Conformance Relations between Input/Output Languages

Robert Sachtleben

October 11, 2023

Abstract

This entry formalises the paper of the same name by Huang et al. [1] and presents a unifying characterisation of well-known conformance relations such as equivalence and language inclusion (reduction) on languages over input/output pairs. This characterisation simplifies comparisons between conformance relations and from it a fundamental necessary and sufficient criterion for conformance testing is developed.

Contents

theory *Input-Output-Language-Conformance*
  imports *HOL−Library.Sublist*
begin

1 Preliminaries

type-synonym ($'a$) *alphabet* = $'a$ *set*
type-synonym ($'x$,$'y$) *word* = ($'x \times 'y$) *list*
type-synonym ($'x$,$'y$) *language* = ($'x$,$'y$) *word set*
type-synonym ($'y$) *output-relation* = ($'y$ *set* $\times$ $'y$ *set*) *set*

fun *is-language* :: $'x$ *alphabet* $\Rightarrow$ $'y$ *alphabet* $\Rightarrow$ ($'x$,$'y$) *language* $\Rightarrow$ *bool* **where**
  is-language X Y L = (
    — nonempty
    ($L \neq \{\}$) $\wedge$
    ($\forall$ $\pi \in L$.
      — over X and Y
      ($\forall$ $xy \in$ *set* π . *fst* $xy \in X$ $\wedge$ *snd* $xy \in Y$) $\wedge$
      — prefix closed
      ($\forall$ π' . *prefix* π' π $\longrightarrow$ $\pi' \in L$)))

lemma *language-contains-nil* :
  assumes *is-language* X Y L
shows $[] \in L$
  using *assms* **by** *auto*

lemma *language-intersection-is-language* :
  assumes *is-language* X Y *L1*
  and *is-language* X Y *L2*
shows *is-language* X Y (*L1* $\cap$ *L2*)
  using *assms*
  using *language-contains-nil*[*OF assms*(*1*)] *language-contains-nil*[*OF assms*(*2*)]
  unfolding *is-language.simps*
  by (*metis IntD1 IntD2 IntI disjoint-iff*)

fun *language-for-state* :: ($'x$,$'y$) *language* $\Rightarrow$ ($'x$,$'y$) *word* $\Rightarrow$ ($'x$,$'y$) *language* **where**
  language-for-state L π = $\{\tau$. $\pi @ \tau \in L\}$

notation *language-for-state* ($\mathcal{L}$[-,-])

lemma *language-for-state-is-language* :
  assumes *is-language* X Y L
  and $\pi \in L$
shows *is-language* X Y $\mathcal{L}[L,\pi]$
proof −
  have $\bigwedge$ τ . $\tau \in \mathcal{L}[L,\pi] \Longrightarrow$ ($\forall$ $xy \in$ *set* τ . *fst* $xy \in X$ $\wedge$ *snd* $xy \in Y$) $\wedge$ ($\forall$ τ' . *prefix* τ' τ $\longrightarrow$ $\tau' \in \mathcal{L}[L,\pi]$)
  proof −
    fix τ **assume** $\tau \in \mathcal{L}[L,\pi]$

then have $\pi @ \tau \in L$ **by** *auto*
then have $\bigwedge$ *xy* . *xy* $\in$ *set* ($\pi @ \tau$) $\Longrightarrow$ *fst xy* $\in X \wedge$ *snd xy* $\in Y$
and $\bigwedge \pi'$. *prefix* π' ($\pi @ \tau$) $\Longrightarrow \pi' \in L$
using *assms*(*1*) **by** *auto*

have $\bigwedge$ *xy* . *xy* $\in$ *set* $\tau \Longrightarrow$ *fst xy* $\in X \wedge$ *snd xy* $\in Y$
using ‹$\bigwedge$ *xy* . *xy* $\in$ *set* ($\pi @ \tau$) $\Longrightarrow$ *fst xy* $\in X \wedge$ *snd xy* $\in Y$› **by** *auto*
moreover have $\bigwedge \tau'$. *prefix* τ' $\tau \Longrightarrow \tau' \in \mathcal{L}[L,\pi]$
by (*simp add*: ‹$\bigwedge\pi'$. *prefix* π' (π @ τ) $\Longrightarrow \pi' \in L$›)
ultimately show ($\forall$ *xy* $\in$ *set* τ . *fst xy* $\in X \wedge$ *snd xy* $\in Y$) $\wedge$ ($\forall$ τ' . *prefix* τ' $\tau \longrightarrow \tau' \in \mathcal{L}[L,\pi]$)
by *simp*
qed
moreover have $\mathcal{L}[L,\pi] \neq \{\}$
using *assms*(*2*)
by (*metis* (*no-types*, *lifting*) *append.right-neutral empty-Collect-eq language-for-state.simps*)

ultimately show *?thesis*
by *simp*
qed

lemma *language-of-state-empty-iff* :
assumes *is-language X Y L*
shows ($\mathcal{L}[L,\pi] = \{\}$) $\longleftrightarrow$ ($\pi \notin L$)
using *assms* **unfolding** *is-language.simps language-for-state.simps*
by (*metis Collect-empty-eq append.right-neutral prefixI*)

fun *are-equivalent-for-language* :: ('*x*,'*y*) *language* $\Rightarrow$ ('*x*,'*y*) *word* $\Rightarrow$ ('*x*,'*y*) *word* $\Rightarrow$ *bool* **where**
are-equivalent-for-language L α β = ($\mathcal{L}[L,\alpha] = \mathcal{L}[L,\beta]$)

abbreviation(*input*) *input-projection* $\pi \equiv$ *map fst* π
abbreviation(*input*) *output-projection* $\pi \equiv$ *map snd* π
notation *input-projection* ($[\text{-}]_I$)
notation *output-projection* ($[\text{-}]_O$)

fun *is-executable* :: ('*x*,'*y*) *language* $\Rightarrow$ ('*x*,'*y*) *word* $\Rightarrow$ '*x list* $\Rightarrow$ *bool* **where**
is-executable L π *xs* = ($\exists$ $\tau \in \mathcal{L}[L,\pi]$. $[\tau]_I$ = *xs*)

fun *executable-sequences* :: ('*x*,'*y*) *language* $\Rightarrow$ ('*x*,'*y*) *word* $\Rightarrow$ '*x list set* **where**
executable-sequences L π = {*xs* . *is-executable L* π *xs*}

fun *executable-inputs* :: ('*x*,'*y*) *language* $\Rightarrow$ ('*x*,'*y*) *word* $\Rightarrow$ '*x set* **where**
executable-inputs L π = {*x* . *is-executable L* π [*x*]}

notation *executable-inputs* (*exec*[-,-])

lemma *executable-sequences-alt-def* : *executable-sequences* L π = {*xs* . ∃ *ys* . *length* *ys* = *length* *xs* ∧ *zip* *xs* *ys* ∈ $\mathcal{L}[L,\pi]$}
proof –
  have ∗:⋀ *A* *xs* . (∃ τ∈*A*. *map* *fst* τ = *xs*) = (∃ *ys*. *length* *ys* = *length* *xs* ∧ *zip* *xs* *ys* ∈ *A*)
    by (*metis* *length-map* *map-fst-zip* *zip-map-fst-snd*)

  show *?thesis*
    unfolding *executable-sequences.simps* *is-executable.simps*
    unfolding ∗
    by *simp*
qed

lemma *executable-inputs-alt-def* : *executable-inputs* L π = {*x* . ∃ *y* . [(*x*,*y*)] ∈ $\mathcal{L}[L,\pi]$}
proof –
  have ∗:⋀ *A* *xs* . (∃ τ∈*A*. *map* *fst* τ = *xs*) = (∃ *ys*. *length* *ys* = *length* *xs* ∧ *zip* *xs* *ys* ∈ *A*)
    by (*metis* *length-map* *map-fst-zip* *zip-map-fst-snd*)

  have ∗∗: ⋀ *A* *x* . (∃ *ys*. *length* *ys* = *length* [*x*] ∧ *zip* [*x*] *ys* ∈ *A*) = (∃ *y* . [(*x*,*y*)] ∈ *A*)
    by (*metis* *length-Suc-conv* *length-map* *zip-Cons-Cons* *zip-Nil*)

  show *?thesis*
    unfolding *executable-inputs.simps* *is-executable.simps*
    unfolding ∗
    unfolding ∗∗
    by *fastforce*
qed

lemma *executable-inputs-in-alphabet* :
  assumes *is-language* *X* *Y* *L*
  and *x* ∈ *exec*[*L*,π]
shows *x* ∈ *X*
  using *assms* **unfolding** *executable-inputs-alt-def* **by** *auto*

fun *output-sequences* :: ('*x*,'*y*) *language* ⇒ ('*x*,'*y*) *word* ⇒ '*x* *list* ⇒ '*y* *list* *set*
where
  output-sequences *L* π *xs* = *output-projection* ‘ {τ ∈ $\mathcal{L}[L,\pi]$. $[\tau]_I$ = *xs*}

lemma *prefix-closure-no-member* :
  assumes *is-language* *X* *Y* *L*
  and π ∉ *L*

```
shows π@τ ∉ L
  by (meson assms(1) assms(2) is-language.elims(2) prefixI)


lemma output-sequences-empty-iff :
  assumes is-language X Y L
shows  (output-sequences L π xs = {}) = ((π ∉ L) ∨ (¬ is-executable L π xs))
  unfolding output-sequences.simps is-executable.simps language-for-state.simps
 using Collect-empty-eq assms image-empty mem-Collect-eq prefix-closure-no-member
by auto



fun outputs :: ('x,'y) language ⇒ ('x,'y) word ⇒ 'x ⇒ 'y set where
  outputs L π x = {y . [(x,y)] ∈ ℒ[L,π]}

notation outputs (out[-,-,-])

lemma outputs-in-alphabet :
  assumes is-language X Y L
shows out[L,π,x] ⊆ Y
  using assms by auto

lemma outputs-executable : (out[L,π,x] = {}) ⟷ (x ∉ exec[L,π])
  by auto



fun is-completely-specified-for :: 'x set ⇒ ('x,'y) language ⇒ bool where
  is-completely-specified-for X L = (∀ π ∈ L . ∀ x ∈ X . out[L,π,x] ≠ {})


lemma prefix-executable :
  assumes is-language X Y L
  and     π ∈ L
  and     i < length π
shows fst (π ! i) ∈ exec[L,take i π]
proof −
  define π' where π' = take i π
  moreover define π'' where π'' = drop (Suc i) π
  moreover define xy where xy = π ! i
  ultimately have π = π'@[xy]@π''
    by (simp add: Cons-nth-drop-Suc assms(3))
  then have π'@[xy] ∈ L
    using assms(1,2) by auto
  then show ?thesis
    unfolding π'-def xy-def
    unfolding executable-inputs-alt-def language-for-state.simps
    by (metis (mono-tags, lifting) CollectI eq-fst-iff)
```

qed

2 Conformance Relations

definition *language-equivalence* :: $('x,'y)$ *language* $\Rightarrow$ $('x,'y)$ *language* $\Rightarrow$ *bool* **where**
language-equivalence L1 L2 = (*L1* = *L2*)

definition *language-inclusion* :: $('x,'y)$ *language* $\Rightarrow$ $('x,'y)$ *language* $\Rightarrow$ *bool* **where**

language-inclusion L1 L2 = (*L1* $\subseteq$ *L2*)

abbreviation(*input*) *reduction L1 L2* $\equiv$ *language-inclusion L1 L2*

definition *quasi-equivalence* :: $('x,'y)$ *language* $\Rightarrow$ $('x,'y)$ *language* $\Rightarrow$ *bool* **where**
quasi-equivalence L1 L2 = ($\forall$ π $\in$ *L1* $\cap$ *L2* . $\forall$ *x* $\in$ *exec*[*L2*,π] . *out*[*L1*,π,*x*] = *out*[*L2*,π,*x*])

definition *quasi-reduction* :: $('x,'y)$ *language* $\Rightarrow$ $('x,'y)$ *language* $\Rightarrow$ *bool* **where**
quasi-reduction L1 L2 = ($\forall$ π $\in$ *L1* $\cap$ *L2* . $\forall$ *x* $\in$ *exec*[*L2*,π] . (*out*[*L1*,π,*x*] $\neq$ {} $\wedge$ *out*[*L1*,π,*x*] $\subseteq$ *out*[*L2*,π,*x*]))

definition *strong-reduction* :: $('x,'y)$ *language* $\Rightarrow$ $('x,'y)$ *language* $\Rightarrow$ *bool* **where**
strong-reduction L1 L2 = (*quasi-reduction L1 L2* $\wedge$ ($\forall$ π $\in$ *L1* $\cap$ *L2* . $\forall$ *x* . *out*[*L2*,π,*x*] = {} $\longrightarrow$ *out*[*L1*,π,*x*] = {}))

definition *semi-equivalence* :: $('x,'y)$ *language* $\Rightarrow$ $('x,'y)$ *language* $\Rightarrow$ *bool* **where**
semi-equivalence L1 L2 = ($\forall$ π $\in$ *L1* $\cap$ *L2* . $\forall$ *x* $\in$ *exec*[*L2*,π] .
(*out*[*L1*,π,*x*] = {} $\vee$ *out*[*L1*,π,*x*] = *out*[*L2*,π,*x*]) $\wedge$
($\exists$ *x'* . *out*[*L1*,π,*x'*] $\cap$ *out*[*L2*,π,*x'*] $\neq$ {}))

definition *semi-reduction* :: $('x,'y)$ *language* $\Rightarrow$ $('x,'y)$ *language* $\Rightarrow$ *bool* **where**
semi-reduction L1 L2 = ($\forall$ π $\in$ *L1* $\cap$ *L2* . $\forall$ *x* $\in$ *exec*[*L2*,π] .
(*out*[*L1*,π,*x*] $\subseteq$ *out*[*L2*,π,*x*]) $\wedge$
($\exists$ *x'* . *out*[*L1*,π,*x'*] $\cap$ *out*[*L2*,π,*x'*] $\neq$ {}))

definition *strong-semi-equivalence* :: $('x,'y)$ *language* $\Rightarrow$ $('x,'y)$ *language* $\Rightarrow$ *bool* **where**
strong-semi-equivalence L1 L2 = ($\forall$ π $\in$ *L1* $\cap$ *L2* . $\forall$ *x* .
(*x* $\in$ *exec*[*L2*,π] $\longrightarrow$ ((*out*[*L1*,π,*x*] = {} $\vee$ *out*[*L1*,π,*x*] = *out*[*L2*,π,*x*]) $\wedge$ ($\exists$ *x'* . *out*[*L1*,π,*x'*] $\cap$ *out*[*L2*,π,*x'*] $\neq$ {}))) $\wedge$
(*x* $\notin$ *exec*[*L2*,π] $\longrightarrow$ *out*[*L1*,π,*x*] = {}))

definition *strong-semi-reduction* :: $('x,'y)$ *language* $\Rightarrow$ $('x,'y)$ *language* $\Rightarrow$ *bool* **where**
strong-semi-reduction L1 L2 = ($\forall$ π $\in$ *L1* $\cap$ *L2* . $\forall$ *x* .
(*x* $\in$ *exec*[*L2*,π] $\longrightarrow$ (*out*[*L1*,π,*x*] $\subseteq$ *out*[*L2*,π,*x*] $\wedge$ ($\exists$ *x'* . *out*[*L1*,π,*x'*] $\cap$ *out*[*L2*,π,*x'*] $\neq$ {}))) $\wedge$
(*x* $\notin$ *exec*[*L2*,π] $\longrightarrow$ *out*[*L1*,π,*x*] = {}))

3 Unifying Characterisations

3.1 ⪯ Conformance

fun *type-1-conforms* :: ('x,'y) *language* ⇒ 'x *alphabet* ⇒ 'y *output-relation* ⇒ ('x,'y) *language* ⇒ *bool* **where**
type-1-conforms L1 X H L2 = (∀ π ∈ *L1* ∩ *L2* . ∀ *x* ∈ *X* . (*out*[*L1*,π,*x*],*out*[*L2*,π,*x*]) ∈ *H*)

notation *type-1-conforms* (- ⪯[-,-] -)

fun *equiv* :: 'y *alphabet* ⇒ 'y *output-relation* **where**
equiv Y = {(*A*,*A*) | *A* . *A* ⊆ *Y*}

fun *red* :: 'y *alphabet* ⇒ 'y *output-relation* **where**
red Y = {(*A*,*B*) | *A B* . *A* ⊆ *B* ∧ *B* ⊆ *Y*}

fun *quasieq* :: 'y *alphabet* ⇒ 'y *output-relation* **where**
quasieq Y = {(*A*,*A*) | *A* . *A* ⊆ *Y*} ∪ {(*A*,{}) | *A* . *A* ⊆ *Y*}

fun *quasired* :: 'y *alphabet* ⇒ 'y *output-relation* **where**
quasired Y = {(*A*,*B*) | *A B* . *A* ≠ {} ∧ *A* ⊆ *B* ∧ *B* ⊆ *Y*} ∪ {(*C*,{}) | *C* . *C* ⊆ *Y*}

fun *strongred* :: 'y *alphabet* ⇒ 'y *output-relation* **where**
strongred Y = {(*A*,*B*) | *A B* . *A* ≠ {} ∧ *A* ⊆ *B* ∧ *B* ⊆ *Y*} ∪ {({},{})}

lemma *red-type-1* :
  assumes *is-language X Y L1*
  and *is-language X Y L2*
shows *reduction L1 L2* ⟷ (*L1* ⪯[*X*,*red Y*] *L2*)
unfolding *language-inclusion-def* **proof**
  show *L1* ⊆ *L2* ⟹ *L1* ⪯[*X*,*red Y*] *L2*
    using *outputs-in-alphabet*[*OF assms*(*2*)]
    unfolding *type-1-conforms.simps red.simps*
    by *auto*

  show *L1* ⪯[*X*,*red Y*] *L2* ⟹ *L1* ⊆ *L2*
  proof
    fix π **assume** π ∈ *L1* **and** *L1* ⪯[*X*,*red Y*] *L2*

    then show π ∈ *L2* **proof** (*induction* π *rule*: *rev-induct*)
      case *Nil*
      then show *?case* **using** *assms*(*2*) **by** *auto*
    next
      case (*snoc xy* π)
      then have π ∈ *L1* **and** π ∈ *L1* ∩ *L2*
        using *assms*(*1*) **by** *auto*

```
      obtain x y where xy = (x,y)
        by fastforce
      then have y ∈ out[L1,π,x]
        using snoc.prems(1)
        by simp
      moreover have x ∈ X and y ∈ Y
        using snoc.prems(1) assms(1) unfolding ‹xy = (x,y)› by auto
      ultimately have y ∈ out[L2,π,x]
        using snoc.prems(2) ‹π ∈ L1 ∩ L2›
        unfolding type-1-conforms.simps
        by fastforce
      then show ?case
        using ‹xy = (x, y)› by auto
    qed
  qed
qed


lemma equiv-by-reduction : (L1 ⪯[X,equiv Y] L2) ⟷ ((L1 ⪯[X,red Y] L2) ∧
(L2 ⪯[X,red Y] L1))
  by fastforce

lemma equiv-type-1 :
  assumes is-language X Y L1
  and     is-language X Y L2
shows (L1 = L2) ⟷ (L1 ⪯[X,equiv Y] L2)
  unfolding equiv-by-reduction
  unfolding red-type-1[OF assms(1,2), symmetric]
  unfolding red-type-1[OF assms(2,1), symmetric]
  unfolding language-inclusion-def
  by blast


lemma quasired-type-1 :
  assumes is-language X Y L1
  and     is-language X Y L2
shows quasi-reduction L1 L2 ⟷ (L1 ⪯[X,quasired Y] L2)
proof
  have ⋀ π x . quasi-reduction L1 L2 ⟹ π ∈ L1 ∩ L2 ⟹ x ∈ X ⟹ (out[L1,π,x],
out[L2,π,x]) ∈ quasired Y
  proof −
    fix π x assume quasi-reduction L1 L2 and π ∈ L1 ∩ L2 and x ∈ X

    show (out[L1,π,x], out[L2,π,x]) ∈ quasired Y
    proof (cases x ∈ exec[L2,π])
      case False
      then show ?thesis
          by (metis (mono-tags, lifting) CollectI UnCI assms(1) outputs-executable
```

```
outputs-in-alphabet quasired.elims)
    next
      case True
      then obtain y where y ∈ out[L2,π,x] by auto
      then have out[L1,π,x] ⊆ out[L2,π,x] and out[L1,π,x] ≠ {}
        using ‹π ∈ L1 ∩ L2› ‹x ∈ X› ‹quasi-reduction L1 L2›
        unfolding quasi-reduction-def by force+
      moreover have out[L2,π,x] ⊆ Y
        by (meson assms(2) outputs-in-alphabet)
      ultimately show ?thesis
        unfolding quasired.simps by blast
    qed
  qed
  then show quasi-reduction L1 L2 ⟹ (L1 ≼[X,quasired Y] L2)
    by auto


  have ⋀ π x . L1 ≼[X,quasired Y] L2 ⟹ π ∈ L1 ∩ L2 ⟹ x ∈ exec[L2,π] ⟹
out[L1,π,x] ⊆ out[L2,π,x]
   and ⋀ π x . L1 ≼[X,quasired Y] L2 ⟹ π ∈ L1 ∩ L2 ⟹ x ∈ exec[L2,π] ⟹
out[L1,π,x] ≠ {}
  proof −
    fix π x assume L1 ≼[X,quasired Y] L2 and π ∈ L1 ∩ L2 and x ∈ exec[L2,π]
    then have x ∈ X
      using executable-inputs-in-alphabet[OF assms(2)] by auto

    have out[L2,π,x] ≠ {}
      using ‹x ∈ exec[L2,π]›
      by (meson outputs-executable)
    moreover have (out[L1,π,x],out[L2,π,x]) ∈ quasired Y
    by (meson ‹L1 ≼[X,quasired Y] L2› ‹π ∈ L1 ∩ L2› ‹x ∈ X› type-1-conforms.elims(2))
    ultimately show out[L1,π,x] ⊆ out[L2,π,x]
              and out[L1,π,x] ≠ {}
      unfolding quasired.simps
      by blast+
  qed
  then show L1 ≼[X,quasired Y] L2 ⟹ quasi-reduction L1 L2
    by (meson quasi-reduction-def)
qed


lemma quasieq-type-1 :
  assumes is-language X Y L1
  and     is-language X Y L2
shows quasi-equivalence L1 L2 ⟷ (L1 ≼[X,quasieq Y] L2)
proof
   have ⋀ π x . quasi-equivalence L1 L2 ⟹ π ∈ L1 ∩ L2 ⟹ x ∈ X ⟹
(out[L1,π,x], out[L2,π,x]) ∈ quasieq Y
  proof −
```

fix π *x* **assume** *quasi-equivalence L1 L2* **and** $\pi \in L1 \cap L2$ **and** $x \in X$

show (*out*[*L1*,π,*x*], *out*[*L2*,π,*x*]) $\in$ *quasieq Y*
proof (*cases* $x \in$ *exec*[*L2*,π])
case *False*
then show *?thesis*
by (*metis* (*mono-tags*, *lifting*) *CollectI UnCI assms*(*1*) *outputs-executable outputs-in-alphabet quasieq.simps*)
next
case *True*
then show *?thesis*
by (*metis* (*mono-tags*, *lifting*) *CollectI UnCI* ‹$\pi \in L1 \cap L2$› ‹*quasi-equivalence L1 L2*› *assms*(*1*) *outputs-in-alphabet quasi-equivalence-def quasieq.simps*)
qed
qed
then show *quasi-equivalence L1 L2* $\Longrightarrow$ (*L1* $\preceq$[*X*,*quasieq Y*] *L2*)
by *auto*

have $\bigwedge$ π *x* . *L1* $\preceq$[*X*,*quasieq Y*] *L2* $\Longrightarrow$ $\pi \in L1 \cap L2$ $\Longrightarrow$ $x \in$ *exec*[*L2*,π] $\Longrightarrow$ *out*[*L1*,π,*x*] = *out*[*L2*,π,*x*]
proof −
fix π *x* **assume** *L1* $\preceq$[*X*,*quasieq Y*] *L2* **and** $\pi \in L1 \cap L2$ **and** $x \in$ *exec*[*L2*,π]
then have $x \in X$
using *executable-inputs-in-alphabet*[*OF assms*(*2*)] **by** *auto*

have *out*[*L2*,π,*x*] $\neq$ {}
using ‹$x \in$ *exec*[*L2*,π]›
by (*meson outputs-executable*)
moreover have (*out*[*L1*,π,*x*],*out*[*L2*,π,*x*]) $\in$ *quasieq Y*
by (*meson* ‹*L1* $\preceq$[*X*,*quasieq Y*] *L2*› ‹$\pi \in L1 \cap L2$› ‹$x \in X$› *type-1-conforms.elims*(*2*))
ultimately show *out*[*L1*,π,*x*] = *out*[*L2*,π,*x*]
unfolding *quasieq.simps*
by *blast*
qed
then show *L1* $\preceq$[*X*,*quasieq Y*] *L2* $\Longrightarrow$ *quasi-equivalence L1 L2*
by (*meson quasi-equivalence-def*)
qed

lemma *strongred-type-1* :
assumes *is-language X Y L1*
and *is-language X Y L2*
shows *strong-reduction L1 L2* $\longleftrightarrow$ (*L1* $\preceq$[*X*,*strongred Y*] *L2*)
proof
have $\bigwedge$ π *x* . *strong-reduction L1 L2* $\Longrightarrow$ $\pi \in L1 \cap L2$ $\Longrightarrow$ $x \in X$ $\Longrightarrow$ (*out*[*L1*,π,*x*], *out*[*L2*,π,*x*]) $\in$ *strongred Y*
proof −
fix π *x* **assume** *strong-reduction L1 L2* **and** $\pi \in L1 \cap L2$ **and** $x \in X$

```
    have out[L2,π,x] ⊆ Y
      using outputs-in-alphabet[OF assms(2)] .

    show (out[L1,π,x], out[L2,π,x]) ∈ strongred Y
    proof (cases x ∈ exec[L2,π])
      case False
      then have out[L2,π,x] = {}
        using outputs-executable by force
      then have out[L1,π,x] = {}
        using ‹strong-reduction L1 L2› ‹π ∈ L1 ∩ L2›
        unfolding strong-reduction-def by blast
      then show ?thesis
        using ‹out[L2,π,x] = {}› by auto
    next
      case True
      then have out[L1,π,x] ≠ {}
        using ‹strong-reduction L1 L2› ‹π ∈ L1 ∩ L2›
        unfolding strong-reduction-def
        by (meson quasi-reduction-def)
      moreover have out[L1,π,x] ⊆ out[L2,π,x]
       by (meson True ‹π ∈ L1 ∩ L2› ‹strong-reduction L1 L2› quasi-reduction-def
strong-reduction-def)
      ultimately show ?thesis
        unfolding strongred.simps
        using outputs-executable outputs-in-alphabet[OF assms(2)]
        by force
    qed
  qed
  then show strong-reduction L1 L2 ⟹ (L1 ≼[X,strongred Y] L2)
    by auto


  have ⋀ π x . L1 ≼[X,strongred Y] L2 ⟹ π ∈ L1 ∩ L2 ⟹ x ∈ exec[L2,π] ⟹
out[L1,π,x] ≠ {}
   and ⋀ π x . L1 ≼[X,strongred Y] L2 ⟹ π ∈ L1 ∩ L2 ⟹ x ∈ exec[L2,π] ⟹
out[L1,π,x] ⊆ out[L2,π,x]
  proof –
   fix π x y assume L1 ≼[X,strongred Y] L2 and π ∈ L1 ∩ L2 and x ∈ exec[L2,π]
    then have x ∈ X
      using executable-inputs-in-alphabet[OF assms(2)] by auto

    have out[L2,π,x] ≠ {}
      using ‹x ∈ exec[L2,π]›
      by (meson outputs-executable)
    moreover have (out[L1,π,x],out[L2,π,x]) ∈ strongred Y
    by (meson ‹L1 ≼[X,strongred Y] L2› ‹π ∈ L1 ∩ L2› ‹x ∈ X› type-1-conforms.elims(2))
    ultimately show out[L1,π,x] ≠ {} and out[L1,π,x] ⊆ out[L2,π,x]
      unfolding strongred.simps
```

by *blast+*
qed
moreover have ⋀ π x . L1 ⪯[X,strongred Y] L2 ⟹ π ∈ L1 ∩ L2 ⟹ out[L2,π,x] = {} ⟹ out[L1,π,x] = {}
proof −
fix π x **assume** L1 ⪯[X,strongred Y] L2 **and** π ∈ L1 ∩ L2 **and** out[L2,π,x] = {}

show out[L1,π,x] = {}
proof (*rule ccontr*)
assume out[L1,π,x] ≠ {}
then have x ∈ X
by (*meson assms*(*1*) *executable-inputs-in-alphabet outputs-executable*)
then have out[L2,π,x] ≠ {}
using ‹L1 ⪯[X,strongred Y] L2› ‹π ∈ L1 ∩ L2› ‹out[L1,π,x] ≠ {}› **by** *fastforce*
then show *False*
using ‹out[L2,π,x] = {}› **by** *simp*
qed
qed
ultimately show L1 ⪯[X,strongred Y] L2 ⟹ strong-reduction L1 L2
unfolding *strong-reduction-def quasi-reduction-def* **by** *blast*
qed

3.2 ≤ Conformance

fun *type-2-conforms* :: ('x,'y) language ⇒ 'x alphabet ⇒ 'y output-relation ⇒ ('x,'y) language ⇒ bool **where**
type-2-conforms L1 X H L2 = (
(∀ π ∈ L1 ∩ L2 . ∀ x ∈ X . (out[L1,π,x],out[L2,π,x]) ∈ H) ∧
(∀ π ∈ L1 ∩ L2 . exec[L2,π] ≠ {} ⟶ (∃ x . out[L1,π,x] ∩ out[L2,π,x] ≠ {})))

notation *type-2-conforms* (- ≤[-,-] -)

fun *semieq* :: 'y alphabet ⇒ 'y output-relation **where**
semieq Y = {(A,A) | A . A ⊆ Y} ∪ {({},A) | A . A ⊆ Y} ∪ {(A,{}) | A . A ⊆ Y}

fun *semired* :: 'y alphabet ⇒ 'y output-relation **where**
semired Y = {(A,B) | A B . A ⊆ B ∧ B ⊆ Y} ∪ {(C,{}) | C . C ⊆ Y}

fun *strongsemieq* :: 'y alphabet ⇒ 'y output-relation **where**
strongsemieq Y = {(A,A) | A . A ⊆ Y} ∪ {({},A) | A . A ⊆ Y}

fun *strongsemired* :: 'y alphabet ⇒ 'y output-relation **where**
strongsemired Y = {(A,B) | A B . A ⊆ B ∧ B ⊆ Y}

lemma *strongsemieq-alt-def* : strongsemieq Y = semieq Y ∩ red Y

by *auto*

lemma *strongsemired-alt-def* : *strongsemired* Y = *red* Y
by *auto*

lemma *semired-type-2* :
assumes *is-language* X Y *L1*
and *is-language* X Y *L2*
shows (*semi-reduction L1 L2*) $\longleftrightarrow$ (*L1* $\leq$[X, *semired* Y] *L2*)
proof
show *semi-reduction L1 L2* $\Longrightarrow$ *L1* $\leq$[X, *semired* Y] *L2*
proof −
assume *semi-reduction L1 L2*
then have *p1*: $\bigwedge$ π x . $\pi \in$ *L1* $\cap$ *L2* $\Longrightarrow$ $x \in$ *exec*[*L2*,π] $\Longrightarrow$ (*out*[*L1*,π,x] $\subseteq$ *out*[*L2*,π,x])
and *p2*: $\bigwedge$ π x . $\pi \in$ *L1* $\cap$ *L2* $\Longrightarrow$ $x \in$ *exec*[*L2*,π] $\Longrightarrow$ $\exists$ x' . *out*[*L1*,π,x'] $\cap$ *out*[*L2*,π,x'] $\neq$ {}
unfolding *semi-reduction-def* **by** *blast*+

have $\bigwedge$ π x . $\pi \in$ *L1* $\cap$ *L2* $\Longrightarrow$ $x \in X$ $\Longrightarrow$ (*out*[*L1*,π,x], *out*[*L2*,π,x]) $\in$ *semired* Y
by (*metis* (*mono-tags*, *lifting*) *CollectI UnCI assms*(*1*) *assms*(*2*) *outputs-executable outputs-in-alphabet p1 semired.simps*)
moreover have $\bigwedge$ π x . $\pi \in$ *L1* $\cap$ *L2* $\Longrightarrow$ *exec*[*L2*,π] $\neq$ {} $\Longrightarrow$ $\exists x$. *out*[*L1*,π,x] $\cap$ *out*[*L2*,π,x] $\neq$ {}
using *p2* **by** *fastforce*
ultimately show *L1* $\leq$[X, *semired* Y] *L2*
by *auto*
qed

show *L1* $\leq$[X,*semired* Y] *L2* $\Longrightarrow$ *semi-reduction L1 L2*
proof −
assume *L1* $\leq$[X,*semired* Y] *L2*
then have *p1* : $\bigwedge$ π x . $\pi \in$ *L1* $\cap$ *L2* $\Longrightarrow$ $x \in X$ $\Longrightarrow$ (*out*[*L1*,π,x],*out*[*L2*,π,x]) $\in$ *semired* Y
and *p2* : $\bigwedge$ π x . $\pi \in$ *L1* $\cap$ *L2* $\Longrightarrow$ *exec*[*L2*,π] $\neq$ {} $\Longrightarrow$ $\exists$ x . *out*[*L1*,π,x] $\cap$ *out*[*L2*,π,x] $\neq$ {}
by *auto*

have $\bigwedge$ π x . $\pi \in$ *L1* $\cap$ *L2* $\Longrightarrow$ $x \in$ *exec*[*L2*,π] $\Longrightarrow$ (*out*[*L1*,π,x] $\subseteq$ *out*[*L2*,π,x])
proof −
fix π x **assume** $\pi \in$ *L1* $\cap$ *L2* **and** $x \in$ *exec*[*L2*,π]
then have (*out*[*L1*,π,x],*out*[*L2*,π,x]) $\in$ *semired* Y
using *p1 executable-inputs-in-alphabet*[*OF assms*(*2*)] **by** *auto*
moreover have *out*[*L2*,π,x] $\neq$ {}
using ‹$x \in$ *exec*[*L2*,π]› **by** *auto*
ultimately show (*out*[*L1*,π,x] $\subseteq$ *out*[*L2*,π,x])
unfolding *semired.simps* **by** *blast*

qed
moreover have $\bigwedge \pi\ x\ .\ \pi \in L1 \cap L2 \Longrightarrow x \in exec[L2,\pi] \Longrightarrow \exists\ x'\ .\ out[L1,\pi,x'] \cap out[L2,\pi,x'] \neq \{\}$
using *p2* **by** *blast*
ultimately show *?thesis*
unfolding *semi-reduction-def* **by** *blast*
qed
qed

lemma *semieq-type-2* :
assumes *is-language X Y L1*
and *is-language X Y L2*
shows (*semi-equivalence L1 L2*) $\longleftrightarrow$ ($L1 \leq[X,\ semieq\ Y]\ L2$)
proof
show *semi-equivalence L1 L2* $\Longrightarrow L1 \leq[X,\ semieq\ Y]\ L2$
proof −
assume *semi-equivalence L1 L2*
then have *p1*: $\bigwedge \pi\ x\ .\ \pi \in L1 \cap L2 \Longrightarrow x \in exec[L2,\pi] \Longrightarrow out[L1,\pi,x] = \{\} \vee out[L1,\pi,x] = out[L2,\pi,x]$
and *p2*: $\bigwedge \pi\ x\ .\ \pi \in L1 \cap L2 \Longrightarrow x \in exec[L2,\pi] \Longrightarrow \exists\ x'\ .\ out[L1,\pi,x'] \cap out[L2,\pi,x'] \neq \{\}$
unfolding *semi-equivalence-def* **by** *blast+*

have $\bigwedge \pi\ x\ .\ \pi \in L1 \cap L2 \Longrightarrow x \in X \Longrightarrow (out[L1,\pi,x],\ out[L2,\pi,x]) \in semieq\ Y$
proof −
fix π x **assume** $\pi \in L1 \cap L2$ **and** $x \in X$
show $(out[L1,\pi,x],\ out[L2,\pi,x]) \in semieq\ Y$
proof (*cases* $x \in exec[L2,\pi]$)
case *True*
then have $out[L2,\pi,x] \neq \{\}$ **by** *auto*
then show *?thesis*
using *p1*[*OF* ‹$\pi \in L1 \cap L2$› *True*]
using *outputs-in-alphabet*[*OF assms*(*2*)]
unfolding *semieq.simps*
by *fastforce*
next
case *False*
then show *?thesis*
by (*metis* (*mono-tags*, *lifting*) *CollectI UnI2 assms*(*1*) *outputs-executable outputs-in-alphabet semieq.elims*)
qed
qed
moreover have $\bigwedge \pi\ x\ .\ \pi \in L1 \cap L2 \Longrightarrow exec[L2,\pi] \neq \{\} \Longrightarrow \exists x.\ out[L1,\pi,x] \cap out[L2,\pi,x] \neq \{\}$
using *p2* **by** *fastforce*
ultimately show $L1 \leq[X,\ semieq\ Y]\ L2$
by *auto*

qed

show *L1* ≤[*X*,*semieq Y*] *L2* ⟹ *semi-equivalence L1 L2*
proof −
assume *L1* ≤[*X*,*semieq Y*] *L2*
then have *p1* : ⋀ π *x* . π ∈ *L1* ∩ *L2* ⟹ *x* ∈ *X* ⟹ (*out*[*L1*,π,*x*],*out*[*L2*,π,*x*]) ∈ *semieq Y*
and *p2* : ⋀ π *x* . π ∈ *L1* ∩ *L2* ⟹ *exec*[*L2*,π] ≠ {} ⟹ ∃ *x* . *out*[*L1*,π,*x*] ∩ *out*[*L2*,π,*x*] ≠ {}
by *auto*

have ⋀ π *x* . π ∈ *L1* ∩ *L2* ⟹ *x* ∈ *exec*[*L2*,π] ⟹ *out*[*L1*,π,*x*] = {} ∨ *out*[*L1*,π,*x*] = *out*[*L2*,π,*x*]
proof −
fix π *x* **assume** π ∈ *L1* ∩ *L2* **and** *x* ∈ *exec*[*L2*,π]
then have (*out*[*L1*,π,*x*],*out*[*L2*,π,*x*]) ∈ *semieq Y*
using *p1 executable-inputs-in-alphabet*[*OF assms*(*2*)] **by** *auto*
moreover have *out*[*L2*,π,*x*] ≠ {}
using ‹*x* ∈ *exec*[*L2*,π]› **by** *auto*
ultimately show *out*[*L1*,π,*x*] = {} ∨ *out*[*L1*,π,*x*] = *out*[*L2*,π,*x*]
unfolding *semieq.simps*
by *blast*
qed
moreover have ⋀ π *x* . π ∈ *L1* ∩ *L2* ⟹ *x* ∈ *exec*[*L2*,π] ⟹ ∃ *x*′ . *out*[*L1*,π,*x*′] ∩ *out*[*L2*,π,*x*′] ≠ {}
using *p2* **by** *blast*
ultimately show *?thesis*
unfolding *semi-equivalence-def* **by** *blast*
qed
qed

lemma *strongsemired-type-2* :
assumes *is-language X Y L1*
and *is-language X Y L2*
shows (*strong-semi-reduction L1 L2*) ⟷ (*L1* ≤[*X*, *strongsemired Y*] *L2*)
proof
show *strong-semi-reduction L1 L2* ⟹ *L1* ≤[*X*, *strongsemired Y*] *L2*
proof −
assume *strong-semi-reduction L1 L2*
then have *p1*: ⋀ π *x* . π ∈ *L1* ∩ *L2* ⟹ *x* ∈ *exec*[*L2*,π] ⟹ (*out*[*L1*,π,*x*] ⊆ *out*[*L2*,π,*x*])
and *p2*: ⋀ π *x* . π ∈ *L1* ∩ *L2* ⟹ *x* ∈ *exec*[*L2*,π] ⟹ ∃ *x*′ . *out*[*L1*,π,*x*′] ∩ *out*[*L2*,π,*x*′] ≠ {}
and *p3*: ⋀ π *x* . π ∈ *L1* ∩ *L2* ⟹ *x* ∉ *exec*[*L2*,π] ⟹ *out*[*L1*,π,*x*] = {}
unfolding *strong-semi-reduction-def* **by** *blast+*

have ⋀ π *x* . π ∈ *L1* ∩ *L2* ⟹ *x* ∈ *X* ⟹ (*out*[*L1*,π,*x*], *out*[*L2*,π,*x*]) ∈ *strongsemired Y*

unfolding *strongsemired.simps*
by (*metis* (*mono-tags*, *lifting*) *CollectI assms*(*2*) *outputs-executable outputs-in-alphabet p1 p3 set-eq-subset*)
moreover have $\bigwedge \pi\ x\ .\ \pi \in L1 \cap L2 \Longrightarrow exec[L2,\pi] \neq \{\} \Longrightarrow \exists x.\ out[L1,\pi,x] \cap out[L2,\pi,x] \neq \{\}$
using *p2* **by** *fastforce*
ultimately show $L1 \leq[X,\ strongsemired\ Y]\ L2$
by *auto*
qed

show $L1 \leq[X,strongsemired\ Y]\ L2 \Longrightarrow strong\text{-}semi\text{-}reduction\ L1\ L2$
proof −
assume $L1 \leq[X,strongsemired\ Y]\ L2$
then have *p1* : $\bigwedge \pi\ x\ .\ \pi \in L1 \cap L2 \Longrightarrow x \in X \Longrightarrow (out[L1,\pi,x],out[L2,\pi,x]) \in strongsemired\ Y$
and *p2* : $\bigwedge \pi\ x\ .\ \pi \in L1 \cap L2 \Longrightarrow exec[L2,\pi] \neq \{\} \Longrightarrow \exists\ x\ .\ out[L1,\pi,x] \cap out[L2,\pi,x] \neq \{\}$
by *auto*

have $\bigwedge \pi\ x\ .\ \pi \in L1 \cap L2 \Longrightarrow x \in exec[L2,\pi] \Longrightarrow (out[L1,\pi,x] \subseteq out[L2,\pi,x])$
proof −
fix π x **assume** $\pi \in L1 \cap L2$ **and** $x \in exec[L2,\pi]$
then have $(out[L1,\pi,x],out[L2,\pi,x]) \in semired\ Y$
using *p1 executable-inputs-in-alphabet*[*OF assms*(*2*)] **by** *auto*
moreover have $out[L2,\pi,x] \neq \{\}$
using ‹$x \in exec[L2,\pi]$› **by** *auto*
ultimately show $(out[L1,\pi,x] \subseteq out[L2,\pi,x])$
unfolding *semired.simps* **by** *blast*
qed
moreover have $\bigwedge \pi\ x\ .\ \pi \in L1 \cap L2 \Longrightarrow x \in exec[L2,\pi] \Longrightarrow \exists\ x'\ .\ out[L1,\pi,x'] \cap out[L2,\pi,x'] \neq \{\}$
using *p2* **by** *blast*
moreover have $\bigwedge \pi\ x\ .\ \pi \in L1 \cap L2 \Longrightarrow x \notin exec[L2,\pi] \Longrightarrow out[L1,\pi,x] = \{\}$
proof −
fix π x **assume** $\pi \in L1 \cap L2$ **and** $x \notin exec[L2,\pi]$

have $(out[L1,\pi,x],out[L2,\pi,x]) \in strongsemired\ Y$
proof (*cases* $x \in exec[L1,\pi]$)
case *True*
then show *?thesis*
by (*meson* ‹$\pi \in L1 \cap L2$› *assms*(*1*) *executable-inputs-in-alphabet p1*)
next
case *False*
then show *?thesis*
using ‹$x \notin exec[L2,\pi]$› **by** *fastforce*
qed
moreover have $out[L2,\pi,x] = \{\}$
using ‹$x \notin exec[L2,\pi]$› **by** *auto*
ultimately show $out[L1,\pi,x] = \{\}$

unfolding *strongsemired.simps*
by *blast*
qed
ultimately show *?thesis*
unfolding *strong-semi-reduction-def* **by** *blast*
qed
qed

lemma *strongsemieq-type-2* :
assumes *is-language X Y L1*
and *is-language X Y L2*
shows (*strong-semi-equivalence L1 L2*) $\longleftrightarrow$ (*L1* $\leq$[*X*, *strongsemieq Y*] *L2*)
proof
show *strong-semi-equivalence L1 L2* $\Longrightarrow$ *L1* $\leq$[*X*, *strongsemieq Y*] *L2*
proof –
assume *strong-semi-equivalence L1 L2*
then have *p1*: $\bigwedge$ π *x* . $\pi \in$ *L1* $\cap$ *L2* $\Longrightarrow$ *x* $\in$ *exec*[*L2*,π] $\Longrightarrow$ *out*[*L1*,π,*x*] = {} $\vee$ *out*[*L1*,π,*x*] = *out*[*L2*,π,*x*]
and *p2*: $\bigwedge$ π *x* . $\pi \in$ *L1* $\cap$ *L2* $\Longrightarrow$ *x* $\in$ *exec*[*L2*,π] $\Longrightarrow$ $\exists$ *x'* . *out*[*L1*,π,*x'*] $\cap$ *out*[*L2*,π,*x'*] $\neq$ {}
and *p3*: $\bigwedge$ π *x* . $\pi \in$ *L1* $\cap$ *L2* $\Longrightarrow$ *x* $\notin$ *exec*[*L2*,π] $\Longrightarrow$ *out*[*L1*,π,*x*] = {}
unfolding *strong-semi-equivalence-def* **by** *blast+*

have $\bigwedge$ π *x* . $\pi \in$ *L1* $\cap$ *L2* $\Longrightarrow$ *x* $\in$ *X* $\Longrightarrow$ (*out*[*L1*,π,*x*], *out*[*L2*,π,*x*]) $\in$ *strongsemieq Y*
proof –
fix π *x* **assume** $\pi \in$ *L1* $\cap$ *L2* **and** *x* $\in$ *X*
show (*out*[*L1*,π,*x*], *out*[*L2*,π,*x*]) $\in$ *strongsemieq Y*
proof (*cases x* $\in$ *exec*[*L2*,π])
case *True*
then have *out*[*L2*,π,*x*] $\neq$ {} **by** *auto*
then show *?thesis*
using *p1*[*OF* ‹$\pi \in$ *L1* $\cap$ *L2*› *True*]
using *outputs-in-alphabet*[*OF assms*(*2*)]
by *fastforce*
next
case *False*
then show *?thesis*
using ‹$\pi \in$ *L1* $\cap$ *L2*› *p3* **by** *fastforce*
qed
qed
moreover have $\bigwedge$ π *x* . $\pi \in$ *L1* $\cap$ *L2* $\Longrightarrow$ *exec*[*L2*,π] $\neq$ {} $\Longrightarrow$ $\exists$ *x*. *out*[*L1*,π,*x*] $\cap$ *out*[*L2*,π,*x*] $\neq$ {}
using *p2* **by** *fastforce*
ultimately show *L1* $\leq$[*X*, *strongsemieq Y*] *L2*
by *auto*
qed

```
  show L1 ≤[X,strongsemieq Y] L2 ⟹ strong-semi-equivalence L1 L2
  proof –
    assume L1 ≤[X,strongsemieq Y] L2
    then have p1 : ⋀ π x . π ∈ L1 ∩ L2 ⟹ x ∈ X ⟹ (out[L1,π,x],out[L2,π,x])
∈ strongsemieq Y
         and  p2 : ⋀ π x . π ∈ L1 ∩ L2 ⟹ exec[L2,π] ≠ {} ⟹ ∃ x . out[L1,π,x]
∩ out[L2,π,x] ≠ {}
      by auto

    have ⋀ π x . π ∈ L1 ∩ L2 ⟹ x ∈ exec[L2,π] ⟹ out[L1,π,x] = {} ∨ out[L1,π,x]
= out[L2,π,x]
    proof –
      fix π x assume π ∈ L1 ∩ L2 and x ∈ exec[L2,π]
      then have (out[L1,π,x],out[L2,π,x]) ∈ semieq Y
        using p1 executable-inputs-in-alphabet[OF assms(2)] by auto
      moreover have out[L2,π,x] ≠ {}
        using ‹x ∈ exec[L2,π]› by auto
      ultimately show out[L1,π,x] = {} ∨ out[L1,π,x] = out[L2,π,x]
        unfolding semieq.simps
        by blast
    qed
    moreover have ⋀ π x . π ∈ L1 ∩ L2 ⟹ x ∈ exec[L2,π] ⟹ ∃ x' . out[L1,π,x']
∩ out[L2,π,x'] ≠ {}
      using p2 by blast
    moreover have ⋀ π x . π ∈ L1 ∩ L2 ⟹ x ∉ exec[L2,π] ⟹ out[L1,π,x] = {}
    proof –
      fix π x assume π ∈ L1 ∩ L2 and x ∉ exec[L2,π]

      have (out[L1,π,x],out[L2,π,x]) ∈ strongsemieq Y
      proof (cases x ∈ exec[L1,π])
        case True
        then show ?thesis
          by (meson ‹π ∈ L1 ∩ L2› assms(1) executable-inputs-in-alphabet p1)
      next
        case False
        then show ?thesis
          using ‹x ∉ exec[L2,π]› by fastforce
      qed
      moreover have out[L2,π,x] = {}
        using ‹x ∉ exec[L2,π]› by auto
      ultimately show out[L1,π,x] = {}
        unfolding strongsemieq.simps
        by blast
    qed
    ultimately show ?thesis
      unfolding strong-semi-equivalence-def by blast
  qed
qed
```

4 Comparing Conformance Relations

lemma *type-1-subset* :
  assumes *L1* $\preceq$[*X*,*H1*] *L2*
  and *H1* $\subseteq$ *H2*
shows *L1* $\preceq$[*X*,*H2*] *L2*
  using *assms* **by** *auto*

lemma *type-1-subsets* :
shows *equiv Y* $\subseteq$ *strongred Y*
  and *equiv Y* $\subseteq$ *quasieq Y*
  and *strongred Y* $\subseteq$ *red Y*
  and *strongred Y* $\subseteq$ *quasired Y*
  and *quasieq Y* $\subseteq$ *quasired Y*
  by *auto*

lemma *type-1-implications* :
shows *L1* $\preceq$[*X*, *equiv Y*] *L2* $\Longrightarrow$ *L1* $\preceq$[*X*, *strongred Y*] *L2*
  and *L1* $\preceq$[*X*, *equiv Y*] *L2* $\Longrightarrow$ *L1* $\preceq$[*X*, *red Y*] *L2*
  and *L1* $\preceq$[*X*, *equiv Y*] *L2* $\Longrightarrow$ *L1* $\preceq$[*X*, *quasired Y*] *L2*
  and *L1* $\preceq$[*X*, *equiv Y*] *L2* $\Longrightarrow$ *L1* $\preceq$[*X*, *quasieq Y*] *L2*
  and *L1* $\preceq$[*X*, *strongred Y*] *L2* $\Longrightarrow$ *L1* $\preceq$[*X*, *red Y*] *L2*
  and *L1* $\preceq$[*X*, *strongred Y*] *L2* $\Longrightarrow$ *L1* $\preceq$[*X*, *quasired Y*] *L2*
  and *L1* $\preceq$[*X*, *quasieq Y*] *L2* $\Longrightarrow$ *L1* $\preceq$[*X*, *quasired Y*] *L2*
  using *type-1-subset*[*OF - type-1-subsets*(*4*), *of L1 X Y L2*]
  using *type-1-subset*[*OF - type-1-subsets*(*5*), *of L1 X Y L2*]
  by *auto*

lemma *type-2-subset* :
  assumes *L1* $\leq$[*X*,*H1*] *L2*
  and *H1* $\subseteq$ *H2*
shows *L1* $\leq$[*X*,*H2*] *L2*
  using *assms* **by** *auto*

lemma *type-2-subsets* :
shows *strongsemieq Y* $\subseteq$ *strongsemired Y*
  and *strongsemieq Y* $\subseteq$ *semieq Y*
  and *semieq Y* $\subseteq$ *semired Y*
  and *strongsemired Y* $\subseteq$ *semired Y*
  and *strongsemired Y* $\subseteq$ *red Y*
  by *auto*

lemma *type-2-implications* :
shows *L1* $\leq$[*X*, *strongsemieq Y*] *L2* $\Longrightarrow$ *L1* $\leq$[*X*, *strongsemired Y*] *L2*
  and *L1* $\leq$[*X*, *strongsemieq Y*] *L2* $\Longrightarrow$ *L1* $\leq$[*X*, *semieq Y*] *L2*
  and *L1* $\leq$[*X*, *strongsemieq Y*] *L2* $\Longrightarrow$ *L1* $\leq$[*X*, *semired Y*] *L2*
  and *L1* $\leq$[*X*, *strongsemired Y*] *L2* $\Longrightarrow$ *L1* $\leq$[*X*, *semired Y*] *L2*
  and *L1* $\leq$[*X*, *semieq Y*] *L2* $\Longrightarrow$ *L1* $\leq$[*X*, *semired Y*] *L2*

by *auto*

lemma *type-1-conformance-to-type-2* :
assumes *is-language X Y L2*
and $L1 \preceq[X,H1]\ L2$
and $H1 \subseteq H2$
and $\bigwedge A\ B\ .\ (A,B) \in H1 \Longrightarrow B \neq \{\} \Longrightarrow A \cap B \neq \{\}$
shows $L1 \leq[X,H2]\ L2$
proof –
have $\bigwedge \pi\ x\ .\ \pi \in L1 \cap L2 \Longrightarrow x \in X \Longrightarrow (out[L1,\pi,x],\ out[L2,\pi,x]) \in H2$
using *assms*(*2*,*3*) **by** *auto*
moreover have $\bigwedge \pi\ .\ \pi \in L1 \cap L2 \Longrightarrow exec[L2,\pi] \neq \{\} \Longrightarrow \exists x.\ out[L1,\pi,x] \cap out[L2,\pi,x] \neq \{\}$
proof –
fix π **assume** $\pi \in L1 \cap L2$ **and** $exec[L2,\pi] \neq \{\}$
then obtain x **where** $x \in exec[L2,\pi]$
by *blast*
then have $x \in X$
by (*meson assms*(*1*) *executable-inputs-in-alphabet*)
then have $(out[L1,\pi,x],\ out[L2,\pi,x]) \in H1$
using ‹$\pi \in L1 \cap L2$› *assms*(*2*) **by** *auto*
moreover have $out[L2,\pi,x] \neq \{\}$
by (*meson* ‹$x \in exec[L2,\pi]$› *outputs-executable*)
ultimately have $out[L1,\pi,x] \cap out[L2,\pi,x] \neq \{\}$
using *assms*(*4*) **by** *blast*
then show $\exists x.\ out[L1,\pi,x] \cap out[L2,\pi,x] \neq \{\}$
by *blast*
qed
ultimately show *?thesis*
by *auto*
qed

lemma *type-1-and-2-mixed-implications* :
assumes *is-language X Y L2*
shows $L1 \leq[X,\ strongsemieq\ Y]\ L2 \Longrightarrow L1 \preceq[X,\ red\ Y]\ L2$
and $L1 \leq[X,\ strongsemired\ Y]\ L2 \Longrightarrow L1 \preceq[X,\ red\ Y]\ L2$
and $L1 \preceq[X,\ quasieq\ Y]\ L2 \Longrightarrow L1 \leq[X,\ semieq\ Y]\ L2$
and $L1 \preceq[X,\ quasired\ Y]\ L2 \Longrightarrow L1 \leq[X,\ semired\ Y]\ L2$
and $L1 \preceq[X,\ equiv\ Y]\ L2 \Longrightarrow L1 \leq[X,\ strongsemieq\ Y]\ L2$
and $L1 \preceq[X,\ strongred\ Y]\ L2 \Longrightarrow L1 \leq[X,\ strongsemired\ Y]\ L2$
proof –

show $L1 \leq[X,\ strongsemieq\ Y]\ L2 \Longrightarrow L1 \preceq[X,\ red\ Y]\ L2$
and $L1 \leq[X,\ strongsemired\ Y]\ L2 \Longrightarrow L1 \preceq[X,\ red\ Y]\ L2$
by *auto*

have $\bigwedge A\ B\ .\ (A,B) \in quasieq\ Y \Longrightarrow B \neq \{\} \Longrightarrow A \cap B \neq \{\}$
by *auto*

moreover have *quasieq Y* ⊆ *semieq Y*
  by *auto*
ultimately show *L1* ⪯[*X*, *quasieq Y*] *L2* ⟹ *L1* ≤[*X*, *semieq Y*] *L2*
  using *type-1-conformance-to-type-2*[*OF assms*] **by** *blast*

have ⋀ *A B* . (*A*,*B*) ∈ *quasired Y* ⟹ *B* ≠ {} ⟹ *A* ∩ *B* ≠ {}
  by *auto*
moreover have *quasired Y* ⊆ *semired Y*
  unfolding *quasired.simps semired.simps* **by** *blast*
ultimately show *L1* ⪯[*X*, *quasired Y*] *L2* ⟹ *L1* ≤[*X*, *semired Y*] *L2*
  using *type-1-conformance-to-type-2*[*OF assms*] **by** *blast*

have ⋀ *A B* . (*A*,*B*) ∈ *equiv Y* ⟹ *B* ≠ {} ⟹ *A* ∩ *B* ≠ {}
  by *auto*
moreover have *equiv Y* ⊆ *strongsemieq Y*
  unfolding *equiv.simps strongsemieq.simps* **by** *blast*
ultimately show *L1* ⪯[*X*, *equiv Y*] *L2* ⟹ *L1* ≤[*X*, *strongsemieq Y*] *L2*
  using *type-1-conformance-to-type-2*[*OF assms*] **by** *blast*

have ⋀ *A B* . (*A*,*B*) ∈ *strongred Y* ⟹ *B* ≠ {} ⟹ *A* ∩ *B* ≠ {}
  by *auto*
moreover have *strongred Y* ⊆ *strongsemired Y*
  unfolding *strongred.simps strongsemired.simps* **by** *blast*
ultimately show *L1* ⪯[*X*, *strongred Y*] *L2* ⟹ *L1* ≤[*X*, *strongsemired Y*] *L2*
  using *type-1-conformance-to-type-2*[*OF assms*] **by** *blast*
qed

4.1 Completely Specified Languages

definition *partiality-component* :: *′y set* ⇒ *′y output-relation* **where**
  partiality-component Y = {(*A*,{}) | *A* . *A* ⊆ *Y*} ∪ {({},*A*) | *A* . *A* ⊆ *Y*}

abbreviation(*input*) Π *Y* ≡ *partiality-component Y*

lemma *conformance-without-partiality* :
shows *strongsemieq Y* − Π *Y* = *semieq Y* − Π *Y*
  and *semieq Y* − Π *Y* = *equiv Y* − Π *Y*
  and *strongsemired Y* − Π *Y* = *semired Y* − Π *Y*
  and *semired Y* − Π *Y* = *red Y* − Π *Y*
  unfolding *partiality-component-def*
  by *fastforce*+

5 Conformance Testing

type-synonym (*′x*,*′y*) *state-cover* = (*′x*,*′y*) *language*
type-synonym (*′x*,*′y*) *transition-cover* = (*′x*,*′y*) *state-cover* × *′x set*

fun *is-state-cover* :: (*′x*,*′y*) *language* ⇒ (*′x*,*′y*) *language* ⇒ (*′x*,*′y*) *state-cover* ⇒

bool **where**
is-state-cover L1 L2 V = (∀ π ∈ *L1* ∩ *L2* . ∃ α ∈ *V* . ℒ[*L1*,π] = ℒ[*L1*,α] ∧ ℒ[*L2*,π] = ℒ[*L2*,α])

lemma *state-cover-subset* :
assumes *is-language X Y L1*
and *is-language X Y L2*
and *is-state-cover L1 L2 V*
and π ∈ *L1* ∩ *L2*
obtains α **where** α ∈ *V*
and α ∈ *L1* ∩ *L2*
and ℒ[*L1*,π] = ℒ[*L1*,α]
and ℒ[*L2*,π] = ℒ[*L2*,α]
proof −
obtain α **where** α ∈ *V*
and ℒ[*L1*,π] = ℒ[*L1*,α]
and ℒ[*L2*,π] = ℒ[*L2*,α]
using *assms*
by (*meson is-state-cover.elims*(*2*))
moreover have ℒ[*L1*,π] ≠ {} **and** ℒ[*L2*,π] ≠ {}
by (*metis Collect-empty-eq-bot Int-iff append.right-neutral assms*(*4*) *empty-def language-for-state.elims*)+
ultimately have α ∈ *L1* ∩ *L2*
using *language-of-state-empty-iff*[*OF assms*(*1*)] *language-of-state-empty-iff*[*OF assms*(*2*)]
by *blast*
then show *?thesis* **using** *that*[*OF* ‹α ∈ *V*› - ‹ℒ[*L1*,π] = ℒ[*L1*,α]› ‹ℒ[*L2*,π] = ℒ[*L2*,α]›]
by *blast*
qed

theorem *sufficient-condition-for-type-1-conformance* :
assumes *is-language X Y L1*
and *is-language X Y L2*
and *is-state-cover L1 L2 V*
shows (*L1* ⪯[*X*,*H*] *L2*) ⟷ (∀ π ∈ *V* . ∀ *x* ∈ *X* . π ∈ *L1* ∩ *L2* ⟶ (*out*[*L1*,π,*x*], *out*[*L2*,π,*x*]) ∈ *H*)
proof
show (*L1* ⪯[*X*,*H*] *L2*) ⟹ (∀ π ∈ *V* . ∀ *x* ∈ *X* . π ∈ *L1* ∩ *L2* ⟶ (*out*[*L1*,π,*x*], *out*[*L2*,π,*x*]) ∈ *H*)
by *auto*

have (⋀ π *x* . π ∈ *V* ⟹ *x* ∈ *X* ⟹ π ∈ *L1* ∩ *L2* ⟹ (*out*[*L1*,π,*x*], *out*[*L2*,π,*x*]) ∈ *H*) ⟹ (⋀ π *x* . π ∈ *L1* ∩ *L2* ⟹ *x* ∈ *X* ⟹ (*out*[*L1*,π,*x*], *out*[*L2*,π,*x*]) ∈ *H*)
proof −
fix π *x* **assume** (⋀ π *x* . π ∈ *V* ⟹ *x* ∈ *X* ⟹ π ∈ *L1* ∩ *L2* ⟹ (*out*[*L1*,π,*x*],

out[*L2*,π,*x*]) ∈ *H*)
and π ∈ *L1* ∩ *L2*
and *x* ∈ *X*

obtain α **where** α ∈ *V* **and** α ∈ *L1* ∩ *L2* **and** ℒ[*L1*,π] = ℒ[*L1*,α] **and** ℒ[*L2*,π] = ℒ[*L2*,α]
using *state-cover-subset*[*OF assms* ‹π ∈ *L1* ∩ *L2*›] **by** *auto*
then have *out*[*L1*,π,*x*] = *out*[*L1*,α,*x*] **and** *out*[*L2*,π,*x*] = *out*[*L2*,α,*x*]
by *force+*
moreover have (*out*[*L1*,α,*x*], *out*[*L2*,α,*x*]) ∈ *H*
using ‹(⋀ π *x* . π ∈ *V* ⟹ *x* ∈ *X* ⟹ π ∈ *L1* ∩ *L2* ⟹ (*out*[*L1*,π,*x*], *out*[*L2*,π,*x*]) ∈ *H*)› ‹α ∈ *V*› ‹*x* ∈ *X*› ‹α ∈ *L1* ∩ *L2*›
by *blast*
ultimately show (*out*[*L1*,π,*x*], *out*[*L2*,π,*x*]) ∈ *H*
by *simp*
qed
then show ∀π∈*V*. ∀*x*∈*X*. π ∈ *L1* ∩ *L2* ⟶ (*out*[*L1*,π,*x*], *out*[*L2*,π,*x*]) ∈ *H* ⟹ *L1* ⪯[*X*,*H*] *L2*
by *auto*
qed

theorem *sufficient-condition-for-type-2-conformance* :
assumes *is-language X Y L1*
and *is-language X Y L2*
and *is-state-cover L1 L2 V*
shows (*L1* ≤[*X*,*H*] *L2*) ⟷ (∀ π ∈ *V* . ∀ *x* ∈ *X* . π ∈ *L1* ∩ *L2* ⟶ (*out*[*L1*,π,*x*], *out*[*L2*,π,*x*]) ∈ *H* ∧ (*out*[*L2*,π,*x*] ≠ {} ⟶ (∃ *x'* ∈ *X* . *out*[*L1*,π,*x'*] ∩ *out*[*L2*,π,*x'*] ≠ {})))
proof

have ⋀ π *x* . (*L1* ≤[*X*,*H*] *L2*) ⟹ π ∈ *V* ⟹ *x* ∈ *X* ⟹ π ∈ *L1* ∩ *L2* ⟹ (*out*[*L1*,π,*x*], *out*[*L2*,π,*x*]) ∈ *H* ∧ (*out*[*L2*,π,*x*] ≠ {} ⟶ (∃ *x'* ∈ *X* . *out*[*L1*,π,*x'*] ∩ *out*[*L2*,π,*x'*] ≠ {}))
proof −
fix π *x* **assume** *L1* ≤[*X*,*H*] *L2* **and** π ∈ *V* **and** *x* ∈ *X* **and** π ∈ *L1* ∩ *L2*

have (*out*[*L1*,π,*x*], *out*[*L2*,π,*x*]) ∈ *H*
using ‹*L1* ≤[*X*,*H*] *L2*› ‹π ∈ *L1* ∩ *L2*› ‹*x* ∈ *X*› **by** *force*
moreover have *out*[*L2*,π,*x*] ≠ {} ⟹ (∃ *x'* ∈ *X* . *out*[*L1*,π,*x'*] ∩ *out*[*L2*,π,*x'*] ≠ {})
by (*metis* (*no-types*, *lifting*) ‹*L1* ≤[*X*,*H*] *L2*› ‹π ∈ *L1* ∩ *L2*› *assms*(*2*) *empty-iff executable-inputs-in-alphabet inf-bot-right outputs-executable type-2-conforms.elims*(*2*))

ultimately show (*out*[*L1*,π,*x*], *out*[*L2*,π,*x*]) ∈ *H* ∧ (*out*[*L2*,π,*x*] ≠ {} ⟶ (∃ *x'* ∈ *X* . *out*[*L1*,π,*x'*] ∩ *out*[*L2*,π,*x'*] ≠ {}))
by *blast*
qed
then show (*L1* ≤[*X*,*H*] *L2*) ⟹ (∀ π ∈ *V* . ∀ *x* ∈ *X* . π ∈ *L1* ∩ *L2* ⟶ (*out*[*L1*,π,*x*], *out*[*L2*,π,*x*]) ∈ *H* ∧ (*out*[*L2*,π,*x*] ≠ {} ⟶ (∃ *x'* ∈ *X* . *out*[*L1*,π,*x'*]

∩ *out*[*L2*,π,*x'*] ≠ {})))
    by *auto*

  have ($\bigwedge$ π *x* . π ∈ *V* ⟹ *x* ∈ *X* ⟹ π ∈ *L1* ∩ *L2* ⟹ (*out*[*L1*,π,*x*], *out*[*L2*,π,*x*]) ∈ *H* ∧ (*out*[*L2*,π,*x*] ≠ {} ⟶ (∃ *x'* ∈ *X* . *out*[*L1*,π,*x'*] ∩ *out*[*L2*,π,*x'*] ≠ {}))) ⟹ ($\bigwedge$ π *x* . π ∈ *L1* ∩ *L2* ⟹ *x* ∈ *X* ⟹ (*out*[*L1*,π,*x*], *out*[*L2*,π,*x*]) ∈ *H*)
   by (*meson assms*(*1*) *assms*(*2*) *assms*(*3*) *sufficient-condition-for-type-1-conformance type-1-conforms.elims*(*2*))
  moreover have ($\bigwedge$ π *x* . π ∈ *V* ⟹ *x* ∈ *X* ⟹ π ∈ *L1* ∩ *L2* ⟹ (*out*[*L1*,π,*x*], *out*[*L2*,π,*x*]) ∈ *H* ∧ (*out*[*L2*,π,*x*] ≠ {} ⟶ (∃ *x'* ∈ *X* . *out*[*L1*,π,*x'*] ∩ *out*[*L2*,π,*x'*] ≠ {}))) ⟹ ($\bigwedge$ π . π ∈ *L1* ∩ *L2* ⟹ *exec*[*L2*,π] ≠ {} ⟹ (∃ *x* . *out*[*L1*,π,*x*] ∩ *out*[*L2*,π,*x*] ≠ {}))
  proof –
    fix π **assume** π ∈ *L1* ∩ *L2*
            and *exec*[*L2*,π] ≠ {}
            and ∗: ($\bigwedge$ π *x* . π ∈ *V* ⟹ *x* ∈ *X* ⟹ π ∈ *L1* ∩ *L2* ⟹ (*out*[*L1*,π,*x*], *out*[*L2*,π,*x*]) ∈ *H* ∧ (*out*[*L2*,π,*x*] ≠ {} ⟶ (∃ *x'* ∈ *X* . *out*[*L1*,π,*x'*] ∩ *out*[*L2*,π,*x'*] ≠ {})))

    then obtain *x* **where** *x* ∈ *X* **and** *out*[*L2*,π,*x*] ≠ {}
    by (*metis all-not-in-conv assms*(*2*) *executable-inputs-in-alphabet outputs-executable*)

    moreover obtain α **where** α ∈ *V*
                        and α ∈ *L1* ∩ *L2*
                        and ℒ[*L1*,π] = ℒ[*L1*,α]
                        and ℒ[*L2*,π] = ℒ[*L2*,α]
      using *state-cover-subset*[*OF assms* ‹π ∈ *L1* ∩ *L2*›] **by** *blast*
    ultimately show (∃ *x* . *out*[*L1*,π,*x*] ∩ *out*[*L2*,π,*x*] ≠ {})
      using ∗
      by (*metis outputs.elims*)
  qed
  ultimately show (∀ π ∈ *V* . ∀ *x* ∈ *X* . π ∈ *L1* ∩ *L2* ⟶ (*out*[*L1*,π,*x*], *out*[*L2*,π,*x*]) ∈ *H* ∧ (*out*[*L2*,π,*x*] ≠ {} ⟶ (∃ *x'* ∈ *X* . *out*[*L1*,π,*x'*] ∩ *out*[*L2*,π,*x'*] ≠ {}))) ⟹ (*L1* ≤[*X*,*H*] *L2*)
    by *auto*
qed

lemma *intersections-card-helper* :
  assumes *finite X*
  and *finite Y*
shows *card* {*A* ∩ *B* | *A B* . *A* ∈ *X* ∧ *B* ∈ *Y*} ≤ *card X* ∗ *card Y*
proof –
  have {*A* ∩ *B* | *A B* . *A* ∈ *X* ∧ *B* ∈ *Y*} = (λ (*A*,*B*) . *A* ∩ *B*) ‘ (*X* × *Y*)
    by *auto*
  then have *card* {*A* ∩ *B* | *A B* . *A* ∈ *X* ∧ *B* ∈ *Y*} ≤ *card* (*X* × *Y*)
    by (*metis* (*no-types*, *lifting*) *assms*(*1*) *assms*(*2*) *card-image-le finite-SigmaI*)
  then show *?thesis*
    by (*simp add*: *card-cartesian-product*)

qed

lemma *prefix-length-take* :
  (*prefix xs ys* ∧ *length xs* ≤ *k*) ⟷ (*prefix xs* (*take k ys*))
proof
  show *prefix xs ys* ∧ *length xs* ≤ *k* ⟹ *prefix xs* (*take k ys*)
    using *prefix-length-prefix take-is-prefix* **by** *fastforce*
  show *prefix xs* (*take k ys*) ⟹ *prefix xs ys* ∧ *length xs* ≤ *k*
   by (*metis le-trans length-take min.cobounded2 prefix-length-le prefix-order.order-trans take-is-prefix*)
qed

lemma *brute-force-state-cover* :
  assumes *is-language X Y L1*
      and *is-language X Y L2*
      and *finite* {ℒ[*L1*,π] | π . π ∈ *L1*}
      and *finite* {ℒ[*L2*,π] | π . π ∈ *L2*}
      and *card* {ℒ[*L1*,π] | π . π ∈ *L1*} ≤ *n*
      and *card* {ℒ[*L2*,π] | π . π ∈ *L2*} ≤ *m*
    shows *is-state-cover L1 L2* {α . *length* α ≤ *m* ∗ *n* − *1* ∧ (∀ *xy* ∈ *set* α . *fst xy* ∈ *X* ∧ *snd xy* ∈ *Y*)}
proof (*rule ccontr*)
  let *?V* = {α. *length* α ≤ *m* ∗ *n* − *1* ∧ (∀ *xy*∈*set* α. *fst xy* ∈ *X* ∧ *snd xy* ∈ *Y*)}
  assume ¬ *is-state-cover L1 L2 ?V*

  define *is-covered* **where** *is-covered* = (λ π . ∃ α ∈ *?V* . ℒ[*L1*,π] = ℒ[*L1*,α] ∧ ℒ[*L2*,π] = ℒ[*L2*,α])
  define *missing-traces* **where** *missing-traces* = {τ . τ ∈ *L1* ∩ *L2* ∧ ¬*is-covered* τ}
  define τ **where** τ = *arg-min length* (λ π . π ∈ *missing-traces*)

  have *missing-traces* ≠ {}
    using ‹¬ *is-state-cover L1 L2 ?V*›
    using *is-covered-def missing-traces-def* **by** *fastforce*
  then have τ ∈ *missing-traces*
          ⋀ τ′ . τ′ ∈ *missing-traces* ⟹ *length* τ ≤ *length* τ′
   using *arg-min-nat-lemma*[**where** *P*=(λ π . π ∈ *missing-traces*) **and** *m*=*length*]

    unfolding τ-*def*[*symmetric*] **by** *blast*+
  then have τ-*props*: τ ∈ *L1* ∩ *L2*
                  ⋀ α . α ∈ *?V* ⟹ ¬(ℒ[*L1*,τ] = ℒ[*L1*,α] ∧ ℒ[*L2*,τ] = ℒ[*L2*,α])
    unfolding *missing-traces-def is-covered-def* **by** *blast*+

have ⋀ *xy* . *xy* ∈ *set* τ ⟹ *fst xy* ∈ *X* ∧ *snd xy* ∈ *Y*
  using ‹τ ∈ *L1* ∩ *L2*› *assms*(*1*) **by** *auto*
moreover have τ ∉ *?V*
  using τ*-props*(*2*) **by** *blast*
ultimately have *length* τ > *m*∗*n*−*1*
  by *simp*

let *?L12* = {ℒ[*L1*,π] | π . π ∈ *L1*} × {ℒ[*L2*,π] | π . π ∈ *L2*}

have *finite ?L12*
  using *assms*(*3*,*4*)
  by *blast*

have *card ?L12* ≤ *m*∗*n*
  using *assms*(*3*,*4*,*5*,*6*)
  by (*metis* (*no-types*, *lifting*) *Sigma-cong card-cartesian-product mult.commute mult-le-mono*)

let *?visited-states* = {(ℒ[*L1*,τ′],ℒ[*L2*,τ′]) | τ′ . τ′ ∈ *set* (*prefixes* τ) ∧ *length* τ′ ≤ *m* ∗ *n* − *1*}

have ⋀ τ′ . τ′ ∈ *set* (*prefixes* τ) ⟹ τ′ ∈ *L1* ∩ *L2*
  by (*meson* τ*-props*(*1*) *assms*(*1*) *assms*(*2*) *in-set-prefixes is-language.elims*(*2*) *language-intersection-is-language*)
then have *?visited-states* ⊆ *?L12*
  by *blast*
then have *card ?visited-states* ≤ *m* ∗ *n*
  using ‹*finite ?L12*› ‹*card ?L12* ≤ *m* ∗ *n*›
  by (*meson card-mono dual-order.trans*)

have *no-index-loop* : ⋀ *i j* . *i* < *j* ⟹ *j* ≤ *length* τ ⟹ ℒ[*L1*, *take i* τ] ≠ ℒ[*L1*, *take j* τ] ∨ ℒ[*L2*, *take i* τ] ≠ ℒ[*L2*, *take j* τ]
proof (*rule ccontr*)
  fix *i j*
  assume *i* < *j* **and** *j* ≤ *length* τ **and** ¬ (ℒ[*L1*,*take i* τ] ≠ ℒ[*L1*,*take j* τ] ∨ ℒ[*L2*,*take i* τ] ≠ ℒ[*L2*,*take j* τ])
  then have ℒ[*L1*,*take i* τ] = ℒ[*L1*,*take j* τ] **and** ℒ[*L2*,*take i* τ] = ℒ[*L2*,*take j* τ]
    by *auto*

have $\{\tau'.\ \tau\ @\ \tau' \in L1\} = \{\tau'.\ take\ j\ \tau\ @\ drop\ j\ \tau\ @\ \tau' \in L1\}$
by (*metis append.assoc append-take-drop-id*)
have $\{\tau'.\ \tau\ @\ \tau' \in L2\} = \{\tau'.\ take\ j\ \tau\ @\ drop\ j\ \tau\ @\ \tau' \in L2\}$
by (*metis append.assoc append-take-drop-id*)

have $\mathcal{L}[L1, take\ i\ \tau\ @\ drop\ j\ \tau] = \mathcal{L}[L1,\tau]$
using ‹$\mathcal{L}[L1, take\ i\ \tau] = \mathcal{L}[L1, take\ j\ \tau]$›
unfolding *language-for-state.simps*
unfolding ‹$\{\tau'.\ \tau\ @\ \tau' \in L1\} = \{\tau'.\ take\ j\ \tau\ @\ drop\ j\ \tau\ @\ \tau' \in L1\}$›
append.assoc **by** *blast*
moreover have $\mathcal{L}[L2, take\ i\ \tau\ @\ drop\ j\ \tau] = \mathcal{L}[L2,\tau]$
using ‹$\mathcal{L}[L2, take\ i\ \tau] = \mathcal{L}[L2, take\ j\ \tau]$›
unfolding *language-for-state.simps*
unfolding ‹$\{\tau'.\ \tau\ @\ \tau' \in L2\} = \{\tau'.\ take\ j\ \tau\ @\ drop\ j\ \tau\ @\ \tau' \in L2\}$›
append.assoc **by** *blast*

have $(take\ i\ \tau\ @\ drop\ j\ \tau) \in missing\text{-}traces$
proof (*rule ccontr*)
assume $take\ i\ \tau\ @\ drop\ j\ \tau \notin missing\text{-}traces$
moreover have $take\ i\ \tau\ @\ drop\ j\ \tau \in L1 \cap L2$
by (*metis IntD1 IntD2 IntI* ‹$\mathcal{L}[L1, take\ i\ \tau] = \mathcal{L}[L1, take\ j\ \tau]$› ‹$\mathcal{L}[L2, take\ i\ \tau] = \mathcal{L}[L2, take\ j\ \tau]$› *τ-props(1) append-take-drop-id language-for-state.elims mem-Collect-eq*)
ultimately obtain α **where** $length\ \alpha \le m * n - 1$
$(\forall xy \in set\ \alpha.\ fst\ xy \in X \wedge snd\ xy \in Y)$
$\mathcal{L}[L1, take\ i\ \tau\ @\ drop\ j\ \tau] = \mathcal{L}[L1,\alpha]$
$\mathcal{L}[L2, take\ i\ \tau\ @\ drop\ j\ \tau] = \mathcal{L}[L2,\alpha]$
unfolding *missing-traces-def is-covered-def*
by *blast*
then have $\tau \notin missing\text{-}traces$
unfolding *missing-traces-def is-covered-def*
using ‹$\tau \in L1 \cap L2$›
unfolding ‹$\mathcal{L}[L1, take\ i\ \tau\ @\ drop\ j\ \tau] = \mathcal{L}[L1,\tau]$›
unfolding ‹$\mathcal{L}[L2, take\ i\ \tau\ @\ drop\ j\ \tau] = \mathcal{L}[L2,\tau]$›
by *blast*
then show *False*
using ‹$\tau \in missing\text{-}traces$› **by** *simp*
qed
moreover have $length\ (take\ i\ \tau\ @\ drop\ j\ \tau) < length\ \tau$
using ‹$i < j$› ‹$j \le length\ \tau$›
by (*induction τ arbitrary: i j; auto*)
ultimately show *False*
using ‹$\bigwedge \tau'.\ \tau' \in missing\text{-}traces \Longrightarrow length\ \tau \le length\ \tau'$›
using *leD* **by** *blast*
qed

have *no-prefix-loop* : $\bigwedge \tau'\ \tau''.\ \tau' \in set\ (prefixes\ \tau) \Longrightarrow \tau'' \in set\ (prefixes\ \tau) \Longrightarrow \tau' \neq \tau'' \Longrightarrow (\mathcal{L}[L1,\tau'], \mathcal{L}[L2,\tau']) \neq (\mathcal{L}[L1,\tau''], \mathcal{L}[L2,\tau''])$
proof −

fix τ' τ'' **assume** $\tau' \in$ *set* (*prefixes* τ) **and** $\tau'' \in$ *set* (*prefixes* τ) **and** $\tau' \neq \tau''$

obtain i **where** $\tau' =$ *take* i τ **and** $i \leq$ *length* τ
using ‹$\tau' \in$ *set* (*prefixes* τ)›
by (*metis append-eq-conv-conj in-set-prefixes linorder-linear prefix-def take-all-iff*)

obtain j **where** $\tau'' =$ *take* j τ **and** $j \leq$ *length* τ
using ‹$\tau'' \in$ *set* (*prefixes* τ)›
by (*metis append-eq-conv-conj in-set-prefixes linorder-linear prefix-def take-all-iff*)

have $i \neq j$
using ‹$\tau' =$ *take* i τ› ‹$\tau' \neq \tau''$› ‹$\tau'' =$ *take* j τ› **by** *blast*
then consider (*a*) $i < j$ | (*b*) $j < i$
using *nat-neq-iff* **by** *blast*
then show ($\mathcal{L}$[*L1*,τ'],$\mathcal{L}$[*L2*,τ']) $\neq$ ($\mathcal{L}$[*L1*,τ''],$\mathcal{L}$[*L2*,τ''])
using *no-index-loop*
using ‹$j \leq$ *length* τ› ‹$i \leq$ *length* τ›
unfolding ‹$\tau' =$ *take* i τ› ‹$\tau'' =$ *take* j τ›
by (*cases*; *blast*)
qed
then have *inj-on* (λ τ' . ($\mathcal{L}$[*L1*,τ'], $\mathcal{L}$[*L2*,τ'])) {τ' . $\tau' \in$ *set* (*prefixes* τ) $\wedge$ *length* $\tau' \leq m * n - 1$}
using *inj-onI*
by (*metis* (*mono-tags*, *lifting*) *mem-Collect-eq*)
then have *card* ((λ τ' . ($\mathcal{L}$[*L1*,τ'], $\mathcal{L}$[*L2*,τ'])) ‘ {τ' . $\tau' \in$ *set* (*prefixes* τ) $\wedge$ *length* $\tau' \leq m * n - 1$}) = *card* {τ' . $\tau' \in$ *set* (*prefixes* τ) $\wedge$ *length* $\tau' \leq m * n - 1$}
using *card-image* **by** *blast*
moreover have *?visited-states* = (λ τ' . ($\mathcal{L}$[*L1*,τ'], $\mathcal{L}$[*L2*,τ'])) ‘ {τ' . $\tau' \in$ *set* (*prefixes* τ) $\wedge$ *length* $\tau' \leq m * n - 1$}
by *auto*
ultimately have *card ?visited-states* = *card* {τ' . $\tau' \in$ *set* (*prefixes* τ) $\wedge$ *length* $\tau' \leq m * n - 1$}
by *simp*
moreover have *card* {τ' . $\tau' \in$ *set* (*prefixes* τ) $\wedge$ *length* $\tau' \leq m * n - 1$} = $m*n$
proof –
have {τ' . $\tau' \in$ *set* (*prefixes* τ) $\wedge$ *length* $\tau' \leq m * n - 1$} = *set* (*prefixes* (*take* ($m*n-1$) τ))
unfolding *in-set-prefixes prefix-length-take*
by *auto*
moreover have *length* (*take* ($m*n-1$) τ) = $m*n-1$
using ‹*length* $\tau > m*n-1$› **by** *auto*
ultimately show *?thesis*
using *length-prefixes distinct-prefixes*
by (*metis* ‹*card* {($\mathcal{L}$[*L1*,τ'], $\mathcal{L}$[*L2*,τ']) |τ'. $\tau' \in$ *set* (*prefixes* τ) $\wedge$ *length* $\tau' \leq m * n - 1$} = *card* {$\tau' \in$ *set* (*prefixes* τ). *length* $\tau' \leq m * n - 1$}› ‹*card* {($\mathcal{L}$[*L1*,τ'], $\mathcal{L}$[*L2*,τ']) |τ'. $\tau' \in$ *set* (*prefixes* τ) $\wedge$ *length* $\tau' \leq m * n - 1$} $\leq m * n$› *distinct-card less-diff-conv not-less-iff-gr-or-eq order-le-less*)

qed

ultimately have *card ?visited-states* = $m{*}n$
by *simp*
then have *?visited-states* = *?L12*
by (*metis* (*no-types*, *lifting*) ‹*card* ($\{\mathcal{L}[L1,\pi] \mid \pi.\ \pi \in L1\} \times \{\mathcal{L}[L2,\pi] \mid \pi.\ \pi \in L2\}) \leq m * n$› ‹*finite* ($\{\mathcal{L}[L1,\pi] \mid \pi.\ \pi \in L1\} \times \{\mathcal{L}[L2,\pi] \mid \pi.\ \pi \in L2\}$)› ‹$\{(\mathcal{L}[L1,\tau'], \mathcal{L}[L2,\tau']) \mid \tau'.\ \tau' \in$ *set* (*prefixes* τ) $\wedge$ *length* $\tau' \leq m * n - 1\} \subseteq \{\mathcal{L}[L1,\pi] \mid \pi.\ \pi \in L1\} \times \{\mathcal{L}[L2,\pi] \mid \pi.\ \pi \in L2\}$› *card-seteq*)

have $(\mathcal{L}[L1,\tau], \mathcal{L}[L2,\tau]) \in$ *?L12*
using ‹$\tau \in L1 \cap L2$›
by *blast*
moreover have $(\mathcal{L}[L1,\tau], \mathcal{L}[L2,\tau]) \notin$ *?visited-states*
proof
assume $(\mathcal{L}[L1,\tau], \mathcal{L}[L2,\tau]) \in$ *?visited-states*
then obtain τ' **where** $(\mathcal{L}[L1,\tau], \mathcal{L}[L2,\tau]) = (\mathcal{L}[L1,\tau'],\mathcal{L}[L2,\tau'])$
and $\tau' \in$ *set* (*prefixes* τ)
and *length* $\tau' \leq m * n - 1$
by *blast*

then have $\tau \neq \tau'$
using ‹*length* $\tau > m{*}n{-}1$› **by** *auto*

show *False*
using ‹$(\mathcal{L}[L1,\tau], \mathcal{L}[L2,\tau]) = (\mathcal{L}[L1,\tau'],\mathcal{L}[L2,\tau'])$›
using *no-prefix-loop*[*OF* - ‹$\tau' \in$ *set* (*prefixes* τ)› ‹$\tau \neq \tau'$›]
by *simp*
qed
ultimately show *False*
unfolding ‹*?visited-states* = *?L12*›
by *blast*
qed

6 Reductions Between Relations

6.1 Quasi-Equivalence via Quasi-Reduction and Absences

fun *absence-completion* :: *'x alphabet* ⇒ *'y alphabet* ⇒ (*'x*,*'y*) *language* ⇒ (*'x*, *'y* × *bool*) *language* **where**
absence-completion X Y L =
(($\lambda\ \pi$. *map* ($\lambda(x,y)$. $(x,(y,True))$) π) ‘ *L*)
$\cup$ {(*map* ($\lambda(x,y)$. $(x,(y,True))$) π)@[$(x,(y,False))$]@τ | $\pi\ x\ y\ \tau$. $\pi \in L \wedge$

out[*L*,π,*x*] ≠ {} ∧ *y* ∈ *Y* ∧ *y* ∉ *out*[*L*,π,*x*] ∧ (∀ (*x*,(*y*,*a*)) ∈ *set* τ . *x* ∈ *X* ∧ *y* ∈ *Y*)}

lemma *absence-completion-is-language* :
  assumes *is-language X Y L*
shows *is-language X* (*Y* × *UNIV*) (*absence-completion X Y L*)
proof −
  let *?L* = (*absence-completion X Y L*)
  have [] ∈ *?L*
    using *language-contains-nil*[*OF assms*] **by** *auto*

  have *?L* ≠ {}
    using *language-contains-nil*[*OF assms*] **by** *auto*
  moreover have ⋀ γ *xy* . γ ∈ *?L* ⟹ *xy* ∈ *set* γ ⟹ *fst xy* ∈ *X* ∧ *snd xy* ∈ (*Y* × *UNIV*)
          and ⋀ γ γ′ . γ ∈ *?L* ⟹ *prefix* γ′ γ ⟹ γ′ ∈ *?L*
  proof −
    fix γ *xy* γ′ **assume** γ ∈ *?L*
    then consider (*a*) γ ∈ ((λ π . *map* (λ(*x*,*y*) . (*x*,(*y*,*True*))) π) ‘ *L*) |
              (*b*) γ ∈ {(*map* (λ(*x*,*y*) . (*x*,(*y*,*True*))) π)@[(*x*,(*y*,*False*))]@τ | π *x y* τ . π ∈ *L* ∧ *out*[*L*,π,*x*] ≠ {} ∧ *y* ∈ *Y* ∧ *y* ∉ *out*[*L*,π,*x*] ∧ (∀ (*x*,(*y*,*a*)) ∈ *set* τ . *x* ∈ *X* ∧ *y* ∈ *Y*)}
      unfolding *absence-completion.simps* **by** *blast*
    then have (*xy* ∈ *set* γ ⟶ *fst xy* ∈ *X* ∧ *snd xy* ∈ (*Y* × *UNIV*)) ∧ (*prefix* γ′ γ ⟶ γ′ ∈ *?L*)
    proof *cases*
      case *a*
      then obtain π **where** ∗:γ = *map* (λ(*x*,*y*) . (*x*,(*y*,*True*))) π **and** π ∈ *L*
        by *auto*
      then have *p1*: ⋀ *xy* . *xy* ∈ *set* π ⟹ *fst xy* ∈ *X* ∧ *snd xy* ∈ *Y*
          and *p2*: ⋀ π′ . *prefix* π′ π ⟹ π′ ∈ *L*
        using *assms* **by** *auto*

      have *xy* ∈ *set* γ ⟹ *fst xy* ∈ *X* ∧ *snd xy* ∈ (*Y* × *UNIV*)
      proof −
        assume *xy* ∈ *set* γ
        then have (*fst xy*, *fst* (*snd xy*)) ∈ *set* π **and** *snd* (*snd xy*) = *True*
          unfolding ∗ **by** *auto*
        then show *fst xy* ∈ *X* ∧ *snd xy* ∈ (*Y* × *UNIV*)
          by (*metis p1 SigmaI UNIV-I fst-conv prod.collapse snd-conv*)
      qed
      moreover have *prefix* γ′ γ ⟹ γ′ ∈ *?L*
      proof −
        assume *prefix* γ′ γ
        then obtain *i* **where** γ′ = *take i* γ
          by (*metis append-eq-conv-conj prefix-def*)
        then have γ′ = *map* (λ(*x*,*y*) . (*x*,(*y*,*True*))) (*take i* π)
          unfolding ∗ **using** *take-map* **by** *blast*
        moreover have *take i* π ∈ *L*

```
      using p2 ‹π ∈ L› take-is-prefix by blast
    ultimately have γ' ∈ ((λ π . map (λ(x,y) . (x,(y,True))) π) ‘ L)
      by simp
    then show γ' ∈ ?L
      by auto
  qed
  ultimately show ?thesis by blast
next
  case b
then obtain π x y τ where ∗: γ = (map (λ(x,y) . (x,(y,True))) π)@[(x,(y,False))]@τ
                        and π ∈ L
                        and out[L,π,x] ≠ {}
                        and y ∈ Y
                        and y ∉ out[L,π,x]
                        and (∀ (x,(y,a)) ∈ set τ . x ∈ X ∧ y ∈ Y)
    by blast
  then have p1: ⋀ xy . xy ∈ set π ⟹ fst xy ∈ X ∧ snd xy ∈ Y
       and p2: ⋀ π' . prefix π' π ⟹ π' ∈ L
    using assms by auto

  have x ∈ X
    using ‹out[L,π,x] ≠ {}› assms
    by (meson executable-inputs-in-alphabet outputs-executable)

  have xy ∈ set γ ⟹ fst xy ∈ X ∧ snd xy ∈ (Y × UNIV)
  proof −
    assume xy ∈ set γ
    then consider (b1) xy ∈ set (map (λ(x,y) . (x,(y,True))) π) |
                  (b2) xy = (x,(y,False)) |
                  (b3) xy ∈ set τ
      unfolding ∗ by force
    then show ?thesis proof cases
      case b1
      then have (fst xy, fst (snd xy)) ∈ set π and snd (snd xy) = True
        unfolding ∗ by auto
      then show fst xy ∈ X ∧ snd xy ∈ (Y × UNIV)
        by (metis p1 SigmaI UNIV-I fst-conv prod.collapse snd-conv)
    next
      case b2
      then show ?thesis
        using ‹x ∈ X› ‹y ∈ Y› by simp
    next
      case b3
      then show ?thesis
        using ‹(∀ (x,(y,a)) ∈ set τ . x ∈ X ∧ y ∈ Y)› by force
    qed
  qed
  moreover have prefix γ' γ ⟹ γ' ∈ ?L
  proof −
```

```
    assume prefix γ' γ
    then obtain i where γ' = take i γ
      by (metis append-eq-conv-conj prefix-def)
    then consider (b1) i ≤ length π |
                  (b2) i > length π
      by linarith
    then show γ' ∈ ?L proof cases
      case b1
      then have i ≤ length (map (λ(x, y). (x, y, True)) π)
        by auto
      then have γ' = map (λ(x,y) . (x,(y,True))) (take i π)
        unfolding * ‹γ' = take i γ›
        by (simp add: take-map)
      moreover have take i π ∈ L
        using p2 ‹π ∈ L› take-is-prefix by blast
      ultimately have γ' ∈ ((λ π . map (λ(x,y) . (x,(y,True))) π) ‘ L)
        by simp
      then show γ' ∈ ?L
        by auto
    next
      case b2
      then have i > length (map (λ(x, y). (x, y, True)) π)
        by auto

      have ⋀ k xs ys . k > length xs ⟹ take k (xs@ys) = xs@(take (k − length
xs) ys)
        by simp
       have take-helper: ⋀ k xs y zs . k > length xs ⟹ take k (xs@[y]@zs) =
xs@[y]@(take (k − length xs − 1) zs)
          by (metis One-nat-def Suc-pred ‹⋀ys xs k. length xs < k ⟹ take k
(xs @ ys) = xs @ take (k − length xs) ys› append-Cons append-Nil take-Suc-Cons
zero-less-diff)

       have **: γ' = (map (λ(x,y) . (x,(y,True))) π)@[(x,(y,False))]@(take (i −
length π − 1) τ)
        unfolding * ‹γ' = take i γ›
         using take-helper[OF ‹i > length (map (λ(x, y). (x, y, True)) π)›] by
simp

      have (∀ (x,(y,a)) ∈ set (take (i − length π − 1) τ) . x ∈ X ∧ y ∈ Y)
        using ‹(∀ (x,(y,a)) ∈ set τ . x ∈ X ∧ y ∈ Y)›
        by (meson in-set-takeD)
      then show ?thesis
        unfolding ** absence-completion.simps
        using ‹π ∈ L› ‹out[L,π,x] ≠ {}› ‹y ∈ Y› ‹y ∉ out[L,π,x]›
        by blast
    qed
  qed
  ultimately show ?thesis by simp
```

```
    qed
    then show xy ∈ set γ ⟹ fst xy ∈ X ∧ snd xy ∈ (Y × UNIV)
          and prefix γ' γ ⟹ γ' ∈ ?L
      by blast+
  qed
  ultimately show ?thesis
    unfolding is-language.simps by blast
qed

lemma absence-completion-inclusion-R :
  assumes is-language X Y L
  and     π ∈ absence-completion X Y L
shows (map (λ(x,y,a) . (x,y)) π ∈ L) ⟷ (∀ (x,y,a) ∈ set π . a = True)
proof −
  define L'a where L'a = ((λ π . map (λ(x,y) . (x,(y,True))) π) ‘ L)
  define L'b where L'b = {(map (λ(x,y) . (x,(y,True))) π)@[(x,(y,False))]@τ | π
x y τ . π ∈ L ∧ out[L,π,x] ≠ {} ∧ y ∈ Y ∧ y ∉ out[L,π,x] ∧ (∀ (x,(y,a)) ∈ set τ
. x ∈ X ∧ y ∈ Y)}


  have ⋀ π xya . π ∈ L'a ⟹ xya ∈ set π ⟹ snd (snd xya) = True
    unfolding L'a-def by auto
  moreover have ⋀ π . π ∈ L'b ⟹ ∃ xya ∈ set π . snd (snd xya) = False
    unfolding L'b-def by auto
  moreover have π ∈ L'a ∪ L'b
    using assms(2) unfolding absence-completion.simps L'a-def L'b-def .
  ultimately have (∀ (x,y,a) ∈ set π . a = True) = (π ∈ L'a)
    by fastforce

  show ?thesis proof (cases (∀ (x,y,a) ∈ set π . a = True))
    case True
    then obtain τ where π = map (λ(x, y). (x, y, True)) τ
                and τ ∈ L
      unfolding ‹(∀ (x,y,a) ∈ set π . a = True) = (π ∈ L'a)› L'a-def
      by blast

    have map (λ(x, y, a). (x, y)) π = τ
      unfolding ‹π = map (λ(x, y). (x, y, True)) τ›
      by (induction τ; auto)
    show ?thesis
      using True ‹τ ∈ L›
      unfolding ‹(∀ (x,y,a) ∈ set π . a = True) = (π ∈ L'a)› L'a-def
      unfolding ‹map (λ(x, y, a). (x, y)) π = τ›
      unfolding ‹π = map (λ(x, y). (x, y, True)) τ›
      by blast
  next
    case False
    then have π ∈ L'b
      using ‹(∀ (x,y,a) ∈ set π . a = True) = (π ∈ L'a)› ‹π ∈ L'a ∪ L'b› by blast
```

then obtain τ x y τ' **where** $\pi = (map\ (\lambda(x,y)\ .\ (x,(y,True)))\ \tau)@[(x,(y,False))]@\tau'$
and $\tau \in L$
and $out[L,\tau,x] \neq \{\}$
and $y \in Y$
and $y \notin out[L,\tau,x]$
and $(\forall\ (x,(y,a)) \in set\ \tau'\ .\ x \in X \wedge y \in Y)$
unfolding *L′b-def* **by** *blast*
then have $\tau@[(x,y)] \notin L$
by *fastforce*
then have $\tau@[(x,y)]@(map\ (\lambda(x,\ y,\ a).\ (x,\ y))\ \tau') \notin L$
using *assms*(*1*)
by (*metis append.assoc prefix-closure-no-member*)
moreover have $map\ (\lambda(x,\ y,\ a).\ (x,\ y))\ \pi = \tau@[(x,y)]@(map\ (\lambda(x,\ y,\ a).\ (x,\ y))\ \tau')$
unfolding ‹$\pi = (map\ (\lambda(x,y)\ .\ (x,(y,True)))\ \tau)@[(x,(y,False))]@\tau'$›
by (*induction* τ; *auto*)
ultimately have $map\ (\lambda(x,\ y,\ a).\ (x,\ y))\ \pi \notin L$
by *simp*
then show *?thesis*
using *False* **by** *blast*
qed
qed

lemma *absence-completion-inclusion-L* :
$(\pi \in L) \longleftrightarrow (map\ (\lambda(x,y)\ .\ (x,y,True))\ \pi \in absence\text{-}completion\ X\ Y\ L)$
proof −

let $?L = absence\text{-}completion\ X\ Y\ L$
define $L'a$ **where** $L'a = ((\lambda\ \pi\ .\ map\ (\lambda(x,y)\ .\ (x,(y,True)))\ \pi)\ `\ L)$
define $L'b$ **where** $L'b = \{(map\ (\lambda(x,y)\ .\ (x,(y,True)))\ \pi)@[(x,(y,False))]@\tau \mid \pi\ x\ y\ \tau\ .\ \pi \in L \wedge out[L,\pi,x] \neq \{\} \wedge y \in Y \wedge y \notin out[L,\pi,x] \wedge (\forall\ (x,(y,a)) \in set\ \tau\ .\ x \in X \wedge y \in Y)\}$
have $?L = L'a \cup L'b$
unfolding *L′a-def L′b-def absence-completion.simps* **by** *blast*

have $\bigwedge\ \pi\ .\ \pi \in L'b \Longrightarrow \exists\ xya \in set\ \pi\ .\ snd\ (snd\ xya) = False$
unfolding *L′b-def* **by** *auto*
then have $(map\ (\lambda(x,y)\ .\ (x,y,True))\ \pi \in ?L) = (map\ (\lambda(x,y)\ .\ (x,y,True))\ \pi \in L'a)$
unfolding ‹$?L = L'a \cup L'b$›
by *fastforce*

have $inj\ (\lambda\ \pi\ .\ map\ (\lambda(x,y)\ .\ (x,(y,True)))\ \pi)$
by (*simp add*: *inj-def*)
then show *?thesis*
unfolding ‹$(map\ (\lambda(x,y)\ .\ (x,y,True))\ \pi \in ?L) = (map\ (\lambda(x,y)\ .\ (x,y,True))\ \pi \in L'a)$›
unfolding *L′a-def*
by (*simp add*: *image-iff inj-def*)

qed

fun *is-present* :: ('x,'y × bool) word ⇒ ('x,'y) language ⇒ bool **where**
  is-present π L = (π ∈ map (λ(x, y). (x, y, True)) ‘ L)

lemma *is-present-rev* :
  assumes *is-present* π L
shows *map* (λ(x, y, a). (x, y)) π ∈ L
proof –
  obtain π' **where** π = map (λ(x, y). (x, y, True)) π' **and** π' ∈ L
    using *assms* **by** *auto*
  moreover have *map* (λ(x, y, a). (x, y)) (map (λ(x, y). (x, y, True)) π') = π'
    by (*induction* π'; *auto*)
  ultimately show *?thesis*
    by *force*
qed

lemma *absence-completion-out* :
  assumes *is-language* X Y L
  and x ∈ X
  and π ∈ *absence-completion* X Y L
shows *is-present* π L ⟹ out[L,map (λ(x, y, a). (x, y)) π,x] ≠ {} ⟹ out[absence-completion X Y L, π, x] = {(y,True) | y . y ∈ out[L,map (λ(x, y, a). (x, y)) π,x]} ∪ {(y,False) | y . y ∈ Y ∧ y ∉ out[L,map (λ(x, y, a). (x, y)) π,x]}
and *is-present* π L ⟹ out[L,map (λ(x, y, a). (x, y)) π,x] = {} ⟹ out[absence-completion X Y L, π, x] = {}
and ¬ *is-present* π L ⟹ out[absence-completion X Y L, π, x] = Y × UNIV
proof –

  let *?L* = *absence-completion* X Y L
  define L'a **where** L'a = ((λ π . map (λ(x,y) . (x,(y,True))) π) ‘ L)
  define L'b **where** L'b = {(map (λ(x,y) . (x,(y,True))) π)@[(x,(y,False))]@τ | π x y τ . π ∈ L ∧ out[L,π,x] ≠ {} ∧ y ∈ Y ∧ y ∉ out[L,π,x] ∧ (∀ (x,(y,a)) ∈ set τ . x ∈ X ∧ y ∈ Y)}
  have *?L* = L'a ∪ L'b
    unfolding *L'a-def L'b-def absence-completion.simps* **by** *blast*
  then have out[?L, π, x] = {y. π @ [(x, y)] ∈ L'a} ∪ {y. π @ [(x, y)] ∈ L'b}
    unfolding *outputs.simps language-for-state.simps* **by** *blast*

  show *is-present* π L ⟹ out[L,map (λ(x, y, a). (x, y)) π,x] ≠ {} ⟹ out[?L, π, x] = {(y,True) | y . y ∈ out[L,map (λ(x, y, a). (x, y)) π,x]} ∪ {(y,False) | y . y ∈ Y ∧ y ∉ out[L,map (λ(x, y, a). (x, y)) π,x]}
  proof –
    assume *is-present* π L **and** out[L,map (λ(x, y, a). (x, y)) π,x] ≠ {}
    then have *map* (λ(x, y, a). (x, y)) π ∈ L
      using *assms(1)* **by** *auto*

have {*y*. π @ [(*x*, *y*)] ∈ *L'a*} = {(*y*,*True*) | *y* . *y* ∈ *out*[*L*,*map* (λ(*x*, *y*, *a*). (*x*, *y*)) π,*x*]}
proof
show {*y*. π @ [(*x*, *y*)] ∈ *L'a*} ⊆ {(*y*, *True*) |*y*. *y* ∈ *out*[*L*,*map* (λ(*x*, *y*, *a*). (*x*, *y*)) π,*x*]}
proof
fix *ya* **assume** *ya* ∈ {*y*. π @ [(*x*, *y*)] ∈ *L'a*}
then have π @ [(*x*, *ya*)] ∈ *map* (λ(*x*, *y*). (*x*, *y*, *True*)) ' *L*
unfolding *L'a-def* **by** *blast*
then obtain γ **where** γ ∈ *L* **and** π @ [(*x*, *ya*)] = *map* (λ(*x*, *y*). (*x*, *y*, *True*)) γ
by *blast*
then have *length* (π @ [(*x*, *ya*)]) = *length* γ
by *auto*
then obtain γ' *xy* **where** γ = γ'@[*xy*]
by (*metis add.right-neutral dual-order.strict-iff-not length-append-singleton less-add-Suc2 rev-exhaust take0 take-all-iff*)
then have (*x*,*ya*) = (λ(*x*, *y*). (*x*, *y*, *True*)) *xy*
using ‹π @ [(*x*, *ya*)] = *map* (λ(*x*, *y*). (*x*, *y*, *True*)) γ› **unfolding** ‹γ = γ'@[*xy*]› **by** *auto*
then have *ya* = (*snd xy*, *True*) **and** *xy* = (*x*,*snd xy*)
by (*simp add*: *split-beta*)+
moreover define *y* **where** *y* = *snd xy*
ultimately have *ya* = (*y*, *True*) **and** *xy* = (*x*,*y*)
by *auto*

have π = *map* (λ(*x*, *y*). (*x*, *y*, *True*)) γ'
using ‹π @ [(*x*, *ya*)] = *map* (λ(*x*, *y*). (*x*, *y*, *True*)) γ› **unfolding** ‹γ = γ'@[*xy*]› **by** *auto*
then have *map* (λ(*x*, *y*, *a*). (*x*, *y*)) π = γ'
by (*induction* π *arbitrary*: γ'; *auto*)

have [(*x*, *y*)] ∈ {τ. *map* (λ(*x*, *y*, *a*). (*x*, *y*)) π @ τ ∈ *L*}
using ‹γ ∈ *L*›
unfolding ‹γ = γ'@[*xy*]› ‹*ya* = (*y*, *True*)› ‹*xy* = (*x*,*y*)›
unfolding ‹*map* (λ(*x*, *y*, *a*). (*x*, *y*)) π = γ'›
by *auto*
then show *ya* ∈ {(*y*, *True*) |*y*. *y* ∈ *out*[*L*,*map* (λ(*x*, *y*, *a*). (*x*, *y*)) π,*x*]}
unfolding ‹*ya* = (*snd xy*, *True*)› *outputs.simps language-for-state.simps*
unfolding ‹*ya* = (*y*, *True*)› ‹*xy* = (*x*,*y*)› ‹*map* (λ(*x*, *y*, *a*). (*x*, *y*)) π = γ'›
by *auto*
qed
show {(*y*, *True*) |*y*. *y* ∈ *out*[*L*,*map* (λ(*x*, *y*, *a*). (*x*, *y*)) π,*x*]} ⊆ {*y*. π @ [(*x*, *y*)] ∈ *L'a*}
proof
fix *ya* **assume** *ya* ∈ {(*y*, *True*) |*y*. *y* ∈ *out*[*L*,*map* (λ(*x*, *y*, *a*). (*x*, *y*)) π,*x*]}
then obtain *y* **where** *ya* = (*y*,*True*) **and** *y* ∈ *out*[*L*,*map* (λ(*x*, *y*, *a*). (*x*, *y*)) π,*x*]

by *blast*
then have $[(x, y)] \in \{\tau.\ map\ (\lambda(x, y, a).\ (x, y))\ \pi\ @\ \tau \in L\}$
unfolding *outputs.simps language-for-state.simps* **by** *auto*
then have $(map\ (\lambda(x, y, a).\ (x, y))\ \pi)\ @\ [(x,y)] \in L$
by *auto*
moreover have $map\ (\lambda(x, y).\ (x, y, True))\ ((map\ (\lambda(x, y, a).\ (x, y))\ \pi)\ @\ [(x,y)]) = \pi\ @\ [(x, (y, True))]$
using ‹*is-present* π *L*› **unfolding** *is-present.simps*
by (*induction* π *arbitrary*: *x y*; *auto*)
ultimately have $\pi\ @\ [(x, (y, True))] \in L'a$
unfolding *L′a-def*
by *force*
then show $ya \in \{y.\ \pi\ @\ [(x, y)] \in L'a\}$
unfolding ‹$ya = (y,True)$›
by *blast*
qed
qed
moreover have $\{y.\ \pi\ @\ [(x, y)] \in L'b\} = \{(y,False) \mid y\ .\ y \in Y \wedge y \notin out[L,map\ (\lambda(x, y, a).\ (x, y))\ \pi,x]\}$
proof
show $\{y.\ \pi\ @\ [(x, y)] \in L'b\} \subseteq \{(y, False)\ |y.\ y \in Y \wedge y \notin out[L,map\ (\lambda(x, y, a).\ (x, y))\ \pi,x]\}$
proof
fix *ya* **assume** $ya \in \{y.\ \pi\ @\ [(x, y)] \in L'b\}$
then have $\pi\ @\ [(x,ya)] \in L'b$
by *auto*
then obtain $\pi'\ x'\ y'\ \tau$ **where** $\pi\ @\ [(x,ya)] = map\ (\lambda(x, y).\ (x, y, True))\ \pi'\ @\ [(x', y', False)]\ @\ \tau$
and $\pi' \in L$
and $out[L,\pi',x'] \neq \{\}$
and $y' \in Y$
and $y' \notin out[L,\pi',x']$
and $(\forall (x, y, a) \in set\ \tau.\ x \in X \wedge y \in Y)$
unfolding *L′b-def* **by** *blast*

obtain π'' **where** $\pi = map\ (\lambda(x, y).\ (x, y, True))\ \pi''$ **and** $\pi'' \in L$
using ‹*is-present* π *L*› **by** *auto*
then have $\bigwedge xya\ .\ xya \in set\ \pi \Longrightarrow snd\ (snd\ xya) = True$
by (*induction* π; *auto*)

have $\tau = []$
proof (*rule ccontr*)
assume $\tau \neq []$
then obtain τ' *xyz* **where** $\tau = \tau' @ [xyz]$
by (*metis append-butlast-last-id*)
then have $\pi = map\ (\lambda(x, y).\ (x, y, True))\ \pi'\ @\ [(x', y', False)]\ @\ \tau'$
using ‹$\pi\ @\ [(x,ya)] = map\ (\lambda(x, y).\ (x, y, True))\ \pi'\ @\ [(x', y', False)]\ @\ \tau$›
by *auto*

then have $(x', y', False) \in set\ \pi$
  by *auto*
then show *False*
  using ‹$\bigwedge$ *xya* . *xya* $\in$ *set* $\pi \Longrightarrow$ *snd* (*snd xya*) = *True*› **by** *force*
qed
then have $x' = x$ **and** $ya = (y', False)$ **and** $\pi = map\ (\lambda(x, y).\ (x, y, True))$ π'
  using ‹π @ [(*x*,*ya*)] = *map* (λ(*x*, *y*). (*x*, *y*, *True*)) π' @ [(x', y', *False*)] @ τ›
  by *auto*

have $*$: *map* (λ(*x*, *y*, *a*). (*x*, *y*)) (*map* (λ(*x*, *y*). (*x*, *y*, *True*)) π') = π'
  by (*induction* π'; *auto*)

have $y' \notin out[L,map\ (\lambda(x, y, a).\ (x, y))\ \pi,x]$
  using ‹$y' \notin out[L,\pi',x']$›
  unfolding *outputs.simps language-for-state.simps*
  unfolding ‹π = *map* (λ(*x*, *y*). (*x*, *y*, *True*)) π'› ‹$x' = x$›
  unfolding $*$ **.**
then show $ya \in \{(y, False)\ |y.\ y \in Y \wedge y \notin out[L,map\ (\lambda(x, y, a).\ (x, y))$ $\pi,x]\}$
  using ‹$y' \in Y$›
  unfolding ‹$ya = (y', False)$› **by** *auto*
qed

show $\{(y, False)\ |y.\ y \in Y \wedge y \notin out[L,map\ (\lambda(x, y, a).\ (x, y))\ \pi,x]\} \subseteq \{y.$ π @ $[(x, y)] \in L'b\}$
proof
  fix *ya* **assume** $ya \in \{(y, False)\ |y.\ y \in Y \wedge y \notin out[L,map\ (\lambda(x, y, a).\ (x,$ $y))\ \pi,x]\}$
  then obtain *y* **where** $ya = (y,False)$
    and $y \in Y$
    and $y \notin out[L,map\ (\lambda(x, y, a).\ (x, y))\ \pi,x]$
    by *blast*

  obtain π' **where** $\pi = map\ (\lambda(x, y).\ (x, y, True))\ \pi'$ **and** $\pi' \in L$
    using ‹*is-present* π *L*› **by** *auto*
  have $*$: *map* (λ(*x*, *y*, *a*). (*x*, *y*)) (*map* (λ(*x*, *y*). (*x*, *y*, *True*)) π') = π'
    by (*induction* π'; *auto*)
  have $out[L,\pi',x] \neq \{\}$
    using ‹$out[L,map\ (\lambda(x, y, a).\ (x, y))\ \pi,x] \neq \{\}$›
    unfolding ‹π = *map* (λ(*x*, *y*). (*x*, *y*, *True*)) π'› $*$ **.**
  have $y \notin out[L,\pi',x]$
    using ‹$y \notin out[L,map\ (\lambda(x, y, a).\ (x, y))\ \pi,x]$›
    unfolding ‹π = *map* (λ(*x*, *y*). (*x*, *y*, *True*)) π'› $*$ **.**

  have π@[(*x*,*ya*)] = *map* (λ(*x*, *y*). (*x*, *y*, *True*)) π' @ [(*x*, *y*, *False*)]
    unfolding ‹$ya = (y,False)$› ‹π = *map* (λ(*x*, *y*). (*x*, *y*, *True*)) π'›
    by *auto*

then show *ya* ∈ {*y*. π @ [(*x*, *y*)] ∈ *L′b*}
unfolding *L′b-def*
using ‹π′ ∈ *L*› ‹*out*[*L*,π′,*x*] ≠ {}› ‹*y* ∈ *Y*› ‹*y* ∉ *out*[*L*,π′,*x*]›
by *force*
qed
qed
ultimately show *?thesis*
unfolding ‹*out*[*?L*, π, *x*] = {*y*. π @ [(*x*, *y*)] ∈ *L′a*} ∪ {*y*. π @ [(*x*, *y*)] ∈ *L′b*}›
by *blast*
qed

show *is-present* π *L* ⟹ *out*[*L*,*map* (λ(*x*, *y*, *a*). (*x*, *y*)) π,*x*] = {} ⟹ *out*[*absence-completion* *X Y L*, π, *x*] = {}
proof −
assume *is-present* π *L* **and** *out*[*L*,*map* (λ(*x*, *y*, *a*). (*x*, *y*)) π,*x*] = {}

obtain π′ **where** π = *map* (λ(*x*, *y*). (*x*, *y*, *True*)) π′ **and** π′ ∈ *L*
using ‹*is-present* π *L*› **by** *auto*
have ∗: *map* (λ(*x*, *y*, *a*). (*x*, *y*)) (*map* (λ(*x*, *y*). (*x*, *y*, *True*)) π′) = π′
by (*induction* π′; *auto*)
then have *map* (λ(*x*, *y*, *a*). (*x*, *y*)) π = π′
using ‹π = *map* (λ(*x*, *y*). (*x*, *y*, *True*)) π′› **by** *blast*

have {*y*. π @ [(*x*, *y*)] ∈ *L′a*} = {}
proof −
have ∄ *y* . π @ [(*x*, *y*)] ∈ *L′a*
proof
assume ∃ *y*. π @ [(*x*, *y*)] ∈ *L′a*
then obtain *ya* **where** π @ [(*x*, *ya*)] ∈ *L′a*
by *blast*
then obtain π′′ **where** π′′ ∈ *L* **and** *map* (λ(*x*, *y*). (*x*, *y*, *True*)) π′′ = π @ [(*x*, *ya*)]
unfolding *L′a-def* **by** *force*
then have (*x*,*ya*) = (λ(*x*, *y*). (*x*, *y*, *True*)) (*last* π′′)
by (*metis* (*mono-tags*, *lifting*) *append-is-Nil-conv last-map last-snoc list.map-disc-iff not-Cons-self2*)
then obtain *y* **where** *ya* = (*y*,*True*)
by (*simp add*: *split-beta*)

have *map* (λ(*x*, *y*). (*x*, *y*, *True*)) π′′ = *map* (λ(*x*, *y*). (*x*, *y*, *True*)) (π′ @ [(*x*, *y*)])
using ‹*map* (λ(*x*, *y*). (*x*, *y*, *True*)) π′′ = π @ [(*x*, *ya*)]›
unfolding ‹π = *map* (λ(*x*, *y*). (*x*, *y*, *True*)) π′› ‹*ya* = (*y*,*True*)› **by** *auto*
moreover have *inj* (λ(*x*, *y*). (*x*, *y*, *True*))
by (*simp add*: *inj-def*)
ultimately have π′′ = π′ @ [(*x*,*y*)]
using *inj-map-eq-map* **by** *blast*

show *False*
using ‹$\pi'' \in L$› ‹$out[L, map\ (\lambda(x, y, a).\ (x, y))\ \pi, x] = \{\}$›
unfolding ‹$map\ (\lambda(x, y, a).\ (x, y))\ \pi = \pi'$› ‹$\pi'' = \pi'\ @\ [(x,y)]$›
by *simp*
qed
then show *?thesis*
by *blast*
qed
moreover have $\{y.\ \pi\ @\ [(x, y)] \in L'b\} = \{\}$
proof –
have $\nexists\ y\ .\ \pi\ @\ [(x, y)] \in L'b$
proof
assume $\exists\, y.\ \pi\ @\ [(x, y)] \in L'b$
then obtain *ya* **where** $\pi\ @\ [(x, ya)] \in L'b$
by *blast*
then obtain $\pi''\ x'\ y'\ \tau$ **where** $\pi\ @\ [(x,ya)] = map\ (\lambda(x, y).\ (x, y, True))$
$\pi''\ @\ [(x', y', False)]\ @\ \tau$
and $\pi'' \in L$
and $out[L,\pi'',x'] \neq \{\}$
and $y' \in Y$
and $y' \notin out[L,\pi'',x']$
and $(\forall\, (x, y, a) \in set\ \tau.\ x \in X \wedge y \in Y)$
unfolding *L'b-def* **by** *blast*

have $\bigwedge\ xya\ .\ xya \in set\ \pi \Longrightarrow snd\ (snd\ xya) = True$
using ‹$\pi = map\ (\lambda(x, y).\ (x, y, True))\ \pi'$›
by (*induction* π; *auto*)

have $\tau = []$
proof (*rule ccontr*)
assume $\tau \neq []$
then obtain τ' *xyz* **where** $\tau = \tau'@[xyz]$
by (*metis append-butlast-last-id*)
then have $\pi = map\ (\lambda(x, y).\ (x, y, True))\ \pi''\ @\ [(x', y', False)]\ @\ \tau'$
using ‹$\pi\ @\ [(x,ya)] = map\ (\lambda(x, y).\ (x, y, True))\ \pi''\ @\ [(x', y', False)]$
$@\ \tau$›
by *auto*
then have $(x', y', False) \in set\ \pi$
by *auto*
then show *False*
using ‹$\bigwedge\ xya\ .\ xya \in set\ \pi \Longrightarrow snd\ (snd\ xya) = True$› **by** *force*
qed
then have $x' = x$ **and** $ya = (y', False)$ **and** $\pi = map\ (\lambda(x, y).\ (x, y, True))$
π''
using ‹$\pi\ @\ [(x,ya)] = map\ (\lambda(x, y).\ (x, y, True))\ \pi''\ @\ [(x', y', False)]$
$@\ \tau$›
by *auto*
moreover have *inj* $(\lambda(x, y).\ (x, y, True))$
by (*simp add*: *inj-def*)

ultimately have $\pi'' = \pi'$
unfolding ‹π = *map* ($\lambda(x, y)$. $(x, y, True)$) π'›
using *map-injective* **by** *blast*
then show *False*
using ‹*out*[L,π'',x'] $\neq$ {}› ‹*out*[L,*map* ($\lambda(x, y, a)$. (x, y)) π,x] = {}›
unfolding ‹*map* ($\lambda(x, y, a)$. (x, y)) $\pi = \pi'$› ‹$x' = x$›
by *blast*
qed
then show *?thesis*
by *blast*
qed
ultimately show *?thesis*
unfolding ‹*out*[*?L*, π, x] = {y. π @ [(x, y)] $\in$ $L'a$} $\cup$ {y. π @ [(x, y)] $\in$ $L'b$}›
by *blast*
qed

show $\neg$ *is-present* π L $\Longrightarrow$ *out*[*absence-completion* X Y L,π,x] = $Y \times$ *UNIV*
proof

show *out*[*absence-completion* X Y L,π,x] $\subseteq$ $Y \times$ *UNIV*
using *absence-completion-is-language*[*OF assms*(*1*)]
by (*meson outputs-in-alphabet*)

assume $\neg$ *is-present* π L
then have $\pi \notin L'a$
unfolding *L'a-def* **by** *auto*
then have $\pi \in L'b$
using ‹$\pi \in$ *?L*› ‹*?L* = $L'a \cup L'b$› **by** *blast*
then obtain π' x' y' τ **where** π = *map* ($\lambda(x, y)$. $(x, y, True)$) π' @ [$(x', y', False)$] @ τ
and $\pi' \in L$
and *out*[L,π',x'] $\neq$ {}
and $y' \in Y$
and $y' \notin$ *out*[L,π',x']
and ($\forall (x, y, a)\in$*set* τ. $x \in X \wedge y \in Y$)
unfolding *L'b-def* **by** *blast*

show $Y \times$ *UNIV* $\subseteq$ *out*[*absence-completion* X Y L,π,x]
proof
fix *ya* **assume** *ya* $\in Y \times$ (*UNIV* :: *bool set*)

have π@[(x,ya)] = *map* ($\lambda(x, y)$. $(x, y, True)$) π' @ [$(x', y', False)$] @ (τ @ [(x,ya)])
using ‹π = *map* ($\lambda(x, y)$. $(x, y, True)$) π' @ [$(x', y', False)$] @ τ›
by *auto*
moreover have ‹($\forall (x, y, a)\in$*set* (τ @ [(x,ya)]) . $x \in X \wedge y \in Y$)›
using ‹($\forall (x, y, a)\in$*set* τ. $x \in X \wedge y \in Y$)› ‹$x \in X$› ‹*ya* $\in Y \times$ (*UNIV* ::

bool set)›
    by *auto*
  ultimately have π@[(*x*,*ya*)] ∈ *L'b*
    unfolding *L'b-def*
    using ‹π' ∈ *L*› ‹*out*[*L*,π',*x'*] ≠ {}› ‹*y'* ∈ *Y*› ‹*y'* ∉ *out*[*L*,π',*x'*]›
    by *blast*
  then show *ya* ∈ *out*[*?L*,π,*x*]
    unfolding ‹*out*[*?L*, π, *x*] = {*y*. π @ [(*x*, *y*)] ∈ *L'a*} ∪ {*y*. π @ [(*x*, *y*)] ∈ *L'b*}›
    by *blast*
  qed
 qed
qed

theorem *quasieq-via-quasired* :
 assumes *is-language X Y L1*
 and *is-language X Y L2*
shows (*L1* ⪯[*X*,*quasieq Y*] *L2*) ⟷ ((*absence-completion X Y L1*) ⪯[*X*, *quasired* (*Y* × *UNIV*)] (*absence-completion X Y L2*))
proof

 define *L1'* **where** *L1'* = *absence-completion X Y L1*
 define *L2'* **where** *L2'* = *absence-completion X Y L2*

 define *L1'a* **where** *L1'a* = ((λ π . *map* (λ(*x*,*y*) . (*x*,(*y*,*True*))) π) ‘ *L1*)
 define *L1'b* **where** *L1'b* = {(*map* (λ(*x*,*y*) . (*x*,(*y*,*True*))) π)@[(*x*,(*y*,*False*))]@τ | π *x y* τ . π ∈ *L1* ∧ *out*[*L1*,π,*x*] ≠ {} ∧ *y* ∈ *Y* ∧ *y* ∉ *out*[*L1*,π,*x*] ∧ (∀ (*x*,(*y*,*a*)) ∈ *set* τ . *x* ∈ *X* ∧ *y* ∈ *Y*)}
 define *L2'a* **where** *L2'a* = ((λ π . *map* (λ(*x*,*y*) . (*x*,(*y*,*True*))) π) ‘ *L2*)
 define *L2'b* **where** *L2'b* = {(*map* (λ(*x*,*y*) . (*x*,(*y*,*True*))) π)@[(*x*,(*y*,*False*))]@τ | π *x y* τ . π ∈ *L2* ∧ *out*[*L2*,π,*x*] ≠ {} ∧ *y* ∈ *Y* ∧ *y* ∉ *out*[*L2*,π,*x*] ∧ (∀ (*x*,(*y*,*a*)) ∈ *set* τ . *x* ∈ *X* ∧ *y* ∈ *Y*)}

 have ⋀ π *xya* . π ∈ *L1'a* ⟹ *xya* ∈ *set* π ⟹ *snd* (*snd xya*) = *True*
  unfolding *L1'a-def* **by** *auto*
 moreover have ⋀ π *xya* . π ∈ *L2'a* ⟹ *xya* ∈ *set* π ⟹ *snd* (*snd xya*) = *True*
  unfolding *L2'a-def* **by** *auto*
 moreover have ⋀ π . π ∈ *L1'b* ⟹ ∃ *xya* ∈ *set* π . *snd* (*snd xya*) = *False*
  unfolding *L1'b-def* **by** *auto*
 moreover have ⋀ π . π ∈ *L2'b* ⟹ ∃ *xya* ∈ *set* π . *snd* (*snd xya*) = *False*
  unfolding *L2'b-def* **by** *auto*
 ultimately have *L1'a* ∩ *L2'b* = {} **and** *L1'b* ∩ *L2'a* = {}
  by *blast+*
 moreover have *L1'* = *L1'a* ∪ *L1'b*
  unfolding *L1'-def L1'a-def L1'b-def* **by** *auto*

```
  moreover have L2′ = L2′a ∪ L2′b
    unfolding L2′-def L2′a-def L2′b-def by auto
  ultimately have L1′ ∩ L2′ = (L1′a ∩ L2′a) ∪ (L1′b ∩ L2′b)
    by blast

  have inj (λ π . map (λ(x,y) . (x,(y,True))) π)
    by (simp add: inj-def)
  then have L1′a ∩ L2′a = ((λ π . map (λ(x,y) . (x,(y,True))) π) ‘ (L1 ∩ L2))
    unfolding L1′a-def L2′a-def
    using image-Int by blast

 have intersection-b: L1′b ∩ L2′b = {(map (λ(x,y) . (x,(y,True))) π)@[(x,(y,False))]@τ
| π x y τ . π ∈ L1 ∩ L2 ∧ out[L1,π,x] ≠ {} ∧ out[L2,π,x] ≠ {} ∧ y ∈ Y ∧ y ∉
out[L1,π,x] ∧ y ∉ out[L2,π,x] ∧ (∀ (x,(y,a)) ∈ set τ . x ∈ X ∧ y ∈ Y)}
    (is L1′b ∩ L2′b = ?L12′b)
  proof
    show ?L12′b ⊆ L1′b ∩ L2′b
      unfolding L1′b-def L2′b-def by blast
    show L1′b ∩ L2′b ⊆ ?L12′b
    proof
      fix γ assume γ ∈ L1′b ∩ L2′b

    obtain π1 x1 y1 τ1 where γ = (map (λ(x,y) . (x,(y,True))) π1)@[(x1,(y1,False))]@τ1

                        and π1 ∈ L1
                        and out[L1,π1,x1] ≠ {}
                        and y1 ∈ Y
                        and y1 ∉ out[L1,π1,x1]
                        and (∀ (x,(y,a)) ∈ set τ1 . x ∈ X ∧ y ∈ Y)
        using ‹γ ∈ L1′b ∩ L2′b› unfolding L1′b-def by blast

    obtain π2 x2 y2 τ2 where γ = (map (λ(x,y) . (x,(y,True))) π2)@[(x2,(y2,False))]@τ2

                        and π2 ∈ L2
                        and out[L2,π2,x2] ≠ {}
                        and y2 ∈ Y
                        and y2 ∉ out[L2,π2,x2]
                        and (∀ (x,(y,a)) ∈ set τ2 . x ∈ X ∧ y ∈ Y)
        using ‹γ ∈ L1′b ∩ L2′b› unfolding L2′b-def by blast

      have ⋀ i . i < length π1 ⟹ snd (snd (γ ! i)) = True
      proof −
        fix i assume i < length π1
        then have i < length (map (λ(x,y) . (x,(y,True))) π1) by auto
        then have γ ! i = (map (λ(x,y) . (x,(y,True))) π1) ! i
          unfolding ‹γ = (map (λ(x,y) . (x,(y,True))) π1)@[(x1,(y1,False))]@τ1›
          by (simp add: nth-append)
        also have ... = (λ(x,y) . (x,(y,True))) (π1 ! i)
          using ‹i < length π1› nth-map by blast
```

finally show *snd* (*snd* (γ ! *i*)) = *True*
by (*metis* (*no-types*, *lifting*) *case-prod-conv old.prod.exhaust snd-conv*)
qed
have γ ! *length* π*1* = (*x1*,(*y1*,*False*))
unfolding ‹γ = (*map* (λ(*x*,*y*) . (*x*,(*y*,*True*))) π*1*)@[(*x1*,(*y1*,*False*))]@τ*1*›
by (*metis append-Cons length-map nth-append-length*)
have ⋀ *i* . *i* < *length* π*2* ⟹ *snd* (*snd* (γ ! *i*)) = *True*
proof −
fix *i* **assume** *i* < *length* π*2*
then have *i* < *length* (*map* (λ(*x*,*y*) . (*x*,(*y*,*True*))) π*2*) **by** *auto*
then have γ ! *i* = (*map* (λ(*x*,*y*) . (*x*,(*y*,*True*))) π*2*) ! *i*
unfolding ‹γ = (*map* (λ(*x*,*y*) . (*x*,(*y*,*True*))) π*2*)@[(*x2*,(*y2*,*False*))]@τ*2*›
by (*simp add*: *nth-append*)
also have . . . = (λ(*x*,*y*) . (*x*,(*y*,*True*))) (π*2* ! *i*)
using ‹*i* < *length* π*2*› *nth-map* **by** *blast*
finally show *snd* (*snd* (γ ! *i*)) = *True*
by (*metis* (*no-types*, *lifting*) *case-prod-conv old.prod.exhaust snd-conv*)
qed
have γ ! *length* π*2* = (*x2*,(*y2*,*False*))
unfolding ‹γ = (*map* (λ(*x*,*y*) . (*x*,(*y*,*True*))) π*2*)@[(*x2*,(*y2*,*False*))]@τ*2*›
by (*metis append-Cons length-map nth-append-length*)

have *length* π*1* = *length* π*2*
by (*metis* ‹⋀*i*. *i* < *length* π*1* ⟹ *snd* (*snd* (γ ! *i*)) = *True*› ‹⋀*i*. *i* < *length* π*2* ⟹ *snd* (*snd* (γ ! *i*)) = *True*› ‹γ ! *length* π*1* = (*x1*, *y1*, *False*)› ‹γ ! *length* π*2* = (*x2*, *y2*, *False*)› *not-less-iff-gr-or-eq snd-conv*)
then have π*1* = π*2*
using ‹γ = (*map* (λ(*x*,*y*) . (*x*,(*y*,*True*))) π*1*)@[(*x1*,(*y1*,*False*))]@τ*1*› ‹*inj* (λ π . *map* (λ(*x*,*y*) . (*x*,(*y*,*True*))) π)›
unfolding ‹γ = (*map* (λ(*x*,*y*) . (*x*,(*y*,*True*))) π*2*)@[(*x2*,(*y2*,*False*))]@τ*2*›
using *map-injective* **by** *fastforce*
then have [(*x1*,(*y1*,*False*))]@τ*1* = [(*x2*,(*y2*,*False*))]@τ*2*
using ‹γ = (*map* (λ(*x*,*y*) . (*x*,(*y*,*True*))) π*1*)@[(*x1*,(*y1*,*False*))]@τ*1*›
unfolding ‹γ = (*map* (λ(*x*,*y*) . (*x*,(*y*,*True*))) π*2*)@[(*x2*,(*y2*,*False*))]@τ*2*›
by *force*
then have *x1* = *x2* **and** *y1* = *y2* **and** τ*1* = τ*2*
by *auto*

show γ ∈ *?L12′b*
using ‹π*1* ∈ *L1*› ‹*out*[*L1*,π*1*,*x1*] ≠ {}› ‹*y1* ∈ *Y*› ‹*y1* ∉ *out*[*L1*,π*1*,*x1*]› ‹(∀ (*x*,(*y*,*a*)) ∈ *set* τ*1* . *x* ∈ *X* ∧ *y* ∈ *Y*)›
using ‹π*2* ∈ *L2*› ‹*out*[*L2*,π*2*,*x2*] ≠ {}› ‹*y2* ∈ *Y*› ‹*y2* ∉ *out*[*L2*,π*2*,*x2*]› ‹(∀ (*x*,(*y*,*a*)) ∈ *set* τ*2* . *x* ∈ *X* ∧ *y* ∈ *Y*)›
unfolding ‹π*1* = π*2*› ‹*x1* = *x2*› ‹*y1* = *y2*› ‹τ*1* = τ*2*› ‹γ = (*map* (λ(*x*,*y*) . (*x*,(*y*,*True*))) π*2*)@[(*x2*,(*y2*,*False*))]@τ*2*›
by *blast*
qed
qed

have *is-language* $X\ (Y \times \mathit{UNIV})\ L1'$
  using *absence-completion-is-language*[*OF assms*(*1*)] **unfolding** *L1′-def* **.**
have *is-language* $X\ (Y \times \mathit{UNIV})\ L2'$
  using *absence-completion-is-language*[*OF assms*(*2*)] **unfolding** *L2′-def* **.**

have $(L1 \preceq[X,\mathit{quasieq}\ Y]\ L2) = \mathit{quasi\text{-}equivalence}\ L1\ L2$
  using *quasieq-type-1*[*OF assms*] **by** *blast*

have $(L1' \preceq[X,\mathit{quasired}\ (Y \times \mathit{UNIV})]\ L2') = \mathit{quasi\text{-}reduction}\ L1'\ L2'$
  using *quasired-type-1*[*OF* ‹*is-language* $X\ (Y \times \mathit{UNIV})\ L1'$› ‹*is-language* $X\ (Y \times \mathit{UNIV})\ L2'$›] **by** *blast*

have $\bigwedge \pi\ x\ .\ \mathit{quasi\text{-}equivalence}\ L1\ L2 \Longrightarrow \pi \in L1' \cap L2' \Longrightarrow x \in \mathit{exec}[L2',\pi] \Longrightarrow (\mathit{out}[L1',\pi,x] \neq \{\} \wedge \mathit{out}[L1',\pi,x] \subseteq \mathit{out}[L2',\pi,x])$
proof −
 fix π x **assume** *quasi-equivalence* $L1\ L2$ **and** $\pi \in L1' \cap L2'$ **and** $x \in \mathit{exec}[L2',\pi]$

  have $x \in X$
    using ‹$x \in \mathit{exec}[L2',\pi]$› *absence-completion-is-language*[*OF assms*(*2*)]
    by (*metis L2′-def executable-inputs-in-alphabet*)
  have $\pi \in \mathit{absence\text{-}completion}\ X\ Y\ L1$ **and** $\pi \in \mathit{absence\text{-}completion}\ X\ Y\ L2$
    using ‹$\pi \in L1' \cap L2'$› **unfolding** *L1′-def L2′-def* **by** *blast+*

  consider $(a)\ \pi \in L1'a \cap L2'a \mid (b)\ \pi \in (L1'b \cap L2'b) - (L1'a \cap L2'a)$
   using ‹$\pi \in L1' \cap L2'$› ‹$L1' \cap L2' = (L1'a \cap L2'a) \cup (L1'b \cap L2'b)$› **by** *blast*
  then show $(\mathit{out}[L1',\pi,x] \neq \{\} \wedge \mathit{out}[L1',\pi,x] \subseteq \mathit{out}[L2',\pi,x])$
  proof *cases*
    case *a*
    then obtain τ **where** $\tau \in L1 \cap L2$
              and $\pi = \mathit{map}\ (\lambda(x,y)\ .\ (x,(y,\mathit{True})))\ \tau$
      using ‹$L1'a \cap L2'a = ((\lambda\ \pi\ .\ \mathit{map}\ (\lambda(x,y)\ .\ (x,(y,\mathit{True})))\ \pi)$ ‘ $(L1 \cap L2))$› **by** *blast*

    have $\mathit{map}\ (\lambda(x,\ y,\ a).\ (x,\ y))\ \pi = \tau$
      unfolding ‹$\pi = \mathit{map}\ (\lambda(x,y)\ .\ (x,(y,\mathit{True})))\ \tau$› **by** (*induction* τ; *auto*)

    have *is-present* π $L1$ **and** *is-present* π $L2$
       using ‹$\tau \in L1 \cap L2$› **unfolding** ‹$\pi = \mathit{map}\ (\lambda(x,y)\ .\ (x,(y,\mathit{True})))\ \tau$› **by** *auto*

    have $\mathit{out}[L2,\mathit{map}\ (\lambda(x,\ y,\ a).\ (x,\ y))\ \pi,x] \neq \{\}$
      using ‹$x \in \mathit{exec}[L2',\pi]$›
     using *absence-completion-out*(*2*)[*OF assms*(*2*) ‹$x \in X$› ‹$\pi \in \mathit{absence\text{-}completion}\ X\ Y\ L2$› ‹*is-present* π $L2$›]
      unfolding *L2′-def*[*symmetric*]
      by (*meson outputs-executable*)
    then have $x \in \mathit{exec}[L2,\mathit{map}\ (\lambda(x,\ y,\ a).\ (x,\ y))\ \pi]$
      by *auto*

then have *out*[*L1*,*map* (λ(*x*, *y*, *a*). (*x*, *y*)) π,*x*] ≠ {} **and** *out*[*L1*,*map* (λ(*x*, *y*, *a*). (*x*, *y*)) π,*x*] = *out*[*L2*,*map* (λ(*x*, *y*, *a*). (*x*, *y*)) π,*x*]
using ‹*quasi-equivalence L1 L2*› ‹τ ∈ *L1* ∩ *L2*›
unfolding *quasi-equivalence-def* ‹*map* (λ(*x*, *y*, *a*). (*x*, *y*)) π = τ› **by** *force+*

have *out*[*L1′*,π,*x*] = *out*[*L2′*,π,*x*]
unfolding *L1′-def L2′-def*
unfolding *absence-completion-out*(*1*)[*OF assms*(*2*) ‹*x* ∈ *X*› ‹π ∈ *absence-completion X Y L2*› ‹*is-present* π *L2*› ‹*out*[*L2*,*map* (λ(*x*, *y*, *a*). (*x*, *y*)) π,*x*] ≠ {}›]
unfolding *absence-completion-out*(*1*)[*OF assms*(*1*) ‹*x* ∈ *X*› ‹π ∈ *absence-completion X Y L1*› ‹*is-present* π *L1*› ‹*out*[*L1*,*map* (λ(*x*, *y*, *a*). (*x*, *y*)) π,*x*] ≠ {}›]
using ‹*quasi-equivalence L1 L2*› ‹τ ∈ *L1* ∩ *L2*› ‹*x* ∈ *exec*[*L2*,*map* (λ(*x*, *y*, *a*). (*x*, *y*)) π]›
unfolding *quasi-equivalence-def*
unfolding ‹*map* (λ(*x*, *y*, *a*). (*x*, *y*)) π = τ›
by *blast*
then show *?thesis*
by (*metis* ‹*x* ∈ *exec*[*L2′*,π]› *dual-order.refl outputs-executable*)
next
case *b*

then obtain π′ *x′* *y′* τ′ **where** π = *map* (λ(*x*, *y*). (*x*, *y*, *True*)) π′ @ [(*x′*, *y′*, *False*)] @ τ′
and π′ ∈ *L1* ∩ *L2*
and *out*[*L1*,π′,*x′*] ≠ {}
and *out*[*L2*,π′,*x′*] ≠ {}
and *y′* ∈ *Y*
and *y′* ∉ *out*[*L1*,π′,*x′*]
and *y′* ∉ *out*[*L2*,π′,*x′*]
and (∀ (*x*, *y*, *a*)∈*set* τ′. *x* ∈ *X* ∧ *y* ∈ *Y*)
unfolding *intersection-b*
by *blast*

have ¬ *is-present* π *L1*
using ‹*L1′a* ≡ *map* (λ(*x*, *y*). (*x*, *y*, *True*)) ‘ *L1*› ‹*L1′a* ∩ *L2′b* = {}› *b* **by** *auto*

have ¬ *is-present* π *L2*
using ‹*L2′a* ≡ *map* (λ(*x*, *y*). (*x*, *y*, *True*)) ‘ *L2*› ‹*L1′b* ∩ *L2′a* = {}› *b* **by** *auto*

show *?thesis*
unfolding *L1′-def L2′-def*
unfolding *absence-completion-out*(*3*)[*OF assms*(*1*) ‹*x* ∈ *X*› ‹π ∈ *absence-completion X Y L1*› ‹¬ *is-present* π *L1*›]
unfolding *absence-completion-out*(*3*)[*OF assms*(*2*) ‹*x* ∈ *X*› ‹π ∈ *absence-completion X Y L2*› ‹¬ *is-present* π *L2*›]

using ‹$y' \in Y$›
by *blast*
qed
qed
then show $L1 \preceq[X,quasieq\ Y]\ L2 \Longrightarrow (absence\text{-}completion\ X\ Y\ L1) \preceq[X,quasired\ (Y \times UNIV)]\ (absence\text{-}completion\ X\ Y\ L2)$
unfolding *L1′-def[symmetric] L2′-def[symmetric]*
unfolding ‹$(L1' \preceq[X,quasired\ (Y \times UNIV)]\ L2') = quasi\text{-}reduction\ L1'\ L2'$›
unfolding ‹$(L1 \preceq[X,quasieq\ Y]\ L2) = quasi\text{-}equivalence\ L1\ L2$›
unfolding *quasi-reduction-def*
by *blast*

have $\bigwedge \pi\ x\ .\ quasi\text{-}reduction\ L1'\ L2' \Longrightarrow \pi \in L1 \cap L2 \Longrightarrow x \in exec[L2,\pi] \Longrightarrow out[L1,\pi,x] = out[L2,\pi,x]$
proof −
fix $\pi\ x$ **assume** $quasi\text{-}reduction\ L1'\ L2'$ **and** $\pi \in L1 \cap L2$ **and** $x \in exec[L2,\pi]$
then have $x \in X$
by (*meson assms*(*2*) *executable-inputs-in-alphabet*)

let $?\pi = map\ (\lambda(x,\ y).\ (x,\ y,\ True))\ \pi$
have $map\ (\lambda(x,\ y,\ a).\ (x,\ y))\ ?\pi = \pi$
by (*induction* π; *auto*)
then have $out[L2,map\ (\lambda(x,\ y,\ a).\ (x,\ y))\ ?\pi,x] \neq \{\}$
using ‹$x \in exec[L2,\pi]$› **by** *auto*

have *is-present* ?π *L1* **and** *is-present* ?π *L2*
using ‹$\pi \in L1 \cap L2$› **by** *auto*

have $?\pi \in L1'a \cap L2'a$
using *L1′a-def* ‹$L2'a \equiv map\ (\lambda(x,\ y).\ (x,\ y,\ True))$ ‘ $L2$› ‹*is-present* (*map* (λ(*x*, *y*). (*x*, *y*, *True*)) π) *L1*› ‹*is-present* (*map* (λ(*x*, *y*). (*x*, *y*, *True*)) π) *L2*› **by** *auto*
then have $?\pi \in absence\text{-}completion\ X\ Y\ L1$ **and** $?\pi \in absence\text{-}completion\ X\ Y\ L2$ **and** $?\pi \in L1' \cap L2'$
unfolding *L1′-def[symmetric] L2′-def[symmetric]*
unfolding ‹$L1' = L1'a \cup L1'b$› ‹$L2' = L2'a \cup L2'b$›
by *blast+*

have $out[L2',?\pi,x] = \{(y,\ True)\ |y.\ y \in out[L2,\pi,x]\} \cup \{(y,\ False)\ |y.\ y \in Y \wedge y \notin out[L2,\pi,x]\}$
using *absence-completion-out*(*1*)[*OF assms*(*2*) ‹$x \in X$› ‹$?\pi \in absence\text{-}completion\ X\ Y\ L2$› ‹*is-present* ?π *L2*› ‹$out[L2,map\ (\lambda(x,\ y,\ a).\ (x,\ y))\ ?\pi,x] \neq \{\}$›]
unfolding *L2′-def[symmetric]* ‹$map\ (\lambda(x,\ y,\ a).\ (x,\ y))\ ?\pi = \pi$› **.**
then have $x \in exec[L2',?\pi]$
using ‹$x \in exec[L2,\pi]$› **by** *fastforce*
then have $out[L1',?\pi,x] \neq \{\}$ **and** $out[L1',?\pi,x] \subseteq out[L2',?\pi,x]$
using ‹*quasi-reduction* *L1′ L2′*› ‹$?\pi \in L1' \cap L2'$›
unfolding *quasi-reduction-def*

by *blast+*

have *out*[*L1*,π,*x*] ≠ {}
by (*metis L1′-def* ‹*is-present* (*map* (λ(*x*, *y*). (*x*, *y*, *True*)) π) *L1*› ‹*map* (λ(*x*, *y*). (*x*, *y*, *True*)) π ∈ *absence-completion X Y L1*› ‹*map* (λ(*x*, *y*, *a*). (*x*, *y*)) (*map* (λ(*x*, *y*). (*x*, *y*, *True*)) π) = π› ‹*out*[*L1′*,*map* (λ(*x*, *y*). (*x*, *y*, *True*)) π,*x*] ≠ {}› ‹*x* ∈ *X*› *absence-completion-out*(*2*) *assms*(*1*))
then have *out*[*L1′*,*?*π,*x*] = {(*y*, *True*) |*y*. *y* ∈ *out*[*L1*,π,*x*]} ∪ {(*y*, *False*) |*y*. *y* ∈ *Y* ∧ *y* ∉ *out*[*L1*,π,*x*]}
using *absence-completion-out*(*1*)[*OF assms*(*1*) ‹*x* ∈ *X*› ‹*?*π ∈ *absence-completion X Y L1*› ‹*is-present ?*π *L1*›]
unfolding *L1′-def*[*symmetric*] ‹*map* (λ(*x*, *y*, *a*). (*x*, *y*)) *?*π = π›
by *blast*

have *out*[*L1*,π,*x*] ⊆ *Y* **and** *out*[*L2*,π,*x*] ⊆ *Y*
by (*meson assms*(*1*,*2*) *outputs-in-alphabet*)+

have ⋀ *y* . *y* ∈ *out*[*L1*,π,*x*] ⟹ *y* ∈ *out*[*L2*,π,*x*]
proof −
fix *y* **assume** *y* ∈ *out*[*L1*,π,*x*]
then have (*y*, *True*) ∈ *out*[*L1′*,*?*π,*x*]
unfolding ‹*out*[*L1′*,*?*π,*x*] = {(*y*, *True*) |*y*. *y* ∈ *out*[*L1*,π,*x*]} ∪ {(*y*, *False*) |*y*. *y* ∈ *Y* ∧ *y* ∉ *out*[*L1*,π,*x*]}› **by** *blast*
then have (*y*, *True*) ∈ *out*[*L2′*,*?*π,*x*]
using ‹*out*[*L1′*,*?*π,*x*] ⊆ *out*[*L2′*,*?*π,*x*]› **by** *blast*
then show *y* ∈ *out*[*L2*,π,*x*]
unfolding ‹*out*[*L2′*,*?*π,*x*] = {(*y*, *True*) |*y*. *y* ∈ *out*[*L2*,π,*x*]} ∪ {(*y*, *False*) |*y*. *y* ∈ *Y* ∧ *y* ∉ *out*[*L2*,π,*x*]}›
by *fastforce*
qed
moreover have ⋀ *y* . *y* ∈ *out*[*L2*,π,*x*] ⟹ *y* ∈ *out*[*L1*,π,*x*]
proof −
fix *y* **assume** *y* ∈ *out*[*L2*,π,*x*]
then have (*y*, *True*) ∈ *out*[*L2′*,*?*π,*x*] **and** (*y*, *False*) ∉ *out*[*L2′*,*?*π,*x*]
unfolding ‹*out*[*L2′*,*?*π,*x*] = {(*y*, *True*) |*y*. *y* ∈ *out*[*L2*,π,*x*]} ∪ {(*y*, *False*) |*y*. *y* ∈ *Y* ∧ *y* ∉ *out*[*L2*,π,*x*]}› **by** *blast+*
moreover have (*y*, *True*) ∈ *out*[*L1′*,*?*π,*x*] ∨ (*y*, *False*) ∈ *out*[*L1′*,*?*π,*x*]
unfolding ‹*out*[*L1′*,*?*π,*x*] = {(*y*, *True*) |*y*. *y* ∈ *out*[*L1*,π,*x*]} ∪ {(*y*, *False*) |*y*. *y* ∈ *Y* ∧ *y* ∉ *out*[*L1*,π,*x*]}›
using ‹*out*[*L2*,π,*x*] ⊆ *Y*› ‹*y* ∈ *out*[*L2*,π,*x*]› **by** *auto*
ultimately have (*y*, *True*) ∈ *out*[*L1′*,*?*π,*x*]
using ‹*out*[*L1′*,*?*π,*x*] ⊆ *out*[*L2′*,*?*π,*x*]› **by** *blast*
then show *y* ∈ *out*[*L1*,π,*x*]
unfolding ‹*out*[*L1′*,*?*π,*x*] = {(*y*, *True*) |*y*. *y* ∈ *out*[*L1*,π,*x*]} ∪ {(*y*, *False*) |*y*. *y* ∈ *Y* ∧ *y* ∉ *out*[*L1*,π,*x*]}›
by *fastforce*
qed
ultimately show *out*[*L1*,π,*x*] = *out*[*L2*,π,*x*]
by *blast*

qed
then show (*absence-completion X Y L1*) $\preceq$[*X*,*quasired* (*Y* × *UNIV*)] (*absence-completion X Y L2*) $\Longrightarrow$ *L1* $\preceq$[*X*,*quasieq Y*] *L2*
unfolding *L1′-def*[*symmetric*] *L2′-def*[*symmetric*]
unfolding ‹(*L1′* $\preceq$[*X*,*quasired* (*Y* × *UNIV*)] *L2′*) = *quasi-reduction L1′ L2′*›
unfolding ‹(*L1* $\preceq$[*X*,*quasieq Y*] *L2*) = *quasi-equivalence L1 L2*›
unfolding *quasi-reduction-def quasi-equivalence-def*
by *blast*
qed

6.2 Quasi-Reduction via Reduction and explicit Undefined Behaviour

fun *bottom-completion* :: ′*x alphabet* ⇒ ′*y alphabet* ⇒ (′*x*,′*y*) *language* ⇒ (′*x*, ′*y option*) *language* **where**
bottom-completion X Y L =
((λ π . *map* (λ(*x*,*y*) . (*x*,*Some y*)) π) ‘ *L*)
∪ {(*map* (λ(*x*,*y*) . (*x*,*Some y*)) π)@[(*x*,*y*)]@τ | π *x y* τ . π ∈ *L* ∧ *out*[*L*,π,*x*] = {} ∧ *x* ∈ *X* ∧ (*y* = *None* ∨ *y* ∈ *Some* ‘ *Y*) ∧ (∀ (*x*,*y*) ∈ *set* τ . *x* ∈ *X* ∧ (*y* = *None* ∨ *y* ∈ *Some* ‘ *Y*))}

lemma *bottom-completion-is-language* :
assumes *is-language X Y L*
shows *is-language X* ({*None*} ∪ *Some* ‘ *Y*) (*bottom-completion X Y L*)
proof −
let *?L* = *bottom-completion X Y L*

have *?L* ≠ {}
using *language-contains-nil*[*OF assms*] **by** *auto*
moreover have ⋀ π . π ∈ *?L* $\Longrightarrow$ (∀ *xy* ∈ *set* π . *fst xy* ∈ *X* ∧ *snd xy* ∈ ({*None*} ∪ *Some* ‘ *Y*)) ∧ (∀ π′ . *prefix* π′ π ⟶ π′ ∈ *?L*)
proof −
fix π **assume** π ∈ *?L*
then consider (*a*) π ∈ ((λ π . *map* (λ(*x*,*y*) . (*x*,*Some y*)) π) ‘ *L*) |
(*b*) π ∈ {(*map* (λ(*x*,*y*) . (*x*,*Some y*)) π)@[(*x*,*y*)]@τ | π *x y* τ . π ∈ *L* ∧ *out*[*L*,π,*x*] = {} ∧ *x* ∈ *X* ∧ (*y* = *None* ∨ *y* ∈ *Some* ‘ *Y*) ∧ (∀ (*x*,*y*) ∈ *set* τ . *x* ∈ *X* ∧ (*y* = *None* ∨ *y* ∈ *Some* ‘ *Y*))}
unfolding *bottom-completion.simps* **by** *blast*
then show (∀ *xy* ∈ *set* π . *fst xy* ∈ *X* ∧ *snd xy* ∈ ({*None*} ∪ *Some* ‘ *Y*)) ∧ (∀ π′ . *prefix* π′ π ⟶ π′ ∈ *?L*)
proof *cases*
case *a*
then obtain π′ **where** π = *map* (λ(*x*, *y*). (*x*, *Some y*)) π′ **and** π′ ∈ *L*
by *auto*
then have (∀ *xy* ∈ *set* π′ . *fst xy* ∈ *X* ∧ *snd xy* ∈ *Y*)
and (∀ π′′ . *prefix* π′′ π′ ⟶ π′′ ∈ *L*)
using *assms* **by** *auto*

have $(\forall\ \pi'\ .\ prefix\ \pi'\ \pi \longrightarrow \pi' \in ((\lambda\ \pi\ .\ map\ (\lambda(x,y)\ .\ (x,Some\ y))\ \pi)\ `\ L))$
using ‹$(\forall\ \pi''\ .\ prefix\ \pi''\ \pi' \longrightarrow \pi'' \in L)$› **unfolding** ‹$\pi = map\ (\lambda(x,\ y).\ (x,\ Some\ y))\ \pi'$›
using *prefix-map-rightE* **by** *force*
then have $(\forall\ \pi'\ .\ prefix\ \pi'\ \pi \longrightarrow \pi' \in\ ?L)$
by *auto*
moreover have $(\forall\ xy \in set\ \pi\ .\ fst\ xy \in X \wedge snd\ xy \in (\{None\} \cup Some\ `\ Y))$
using ‹$(\forall\ xy \in set\ \pi'\ .\ fst\ xy \in X \wedge snd\ xy \in Y)$› **unfolding** ‹$\pi = map\ (\lambda(x,\ y).\ (x,\ Some\ y))\ \pi'$›
by (*induction* π'; *auto*)
ultimately show *?thesis*
by *blast*
next
case *b*
then obtain $\pi'\ x\ y\ \tau$ **where** $\pi = (map\ (\lambda(x,y)\ .\ (x,Some\ y))\ \pi')@[(x,y)]@\tau$
and $\pi' \in L$
and $out[L,\pi',x] = \{\}$
and $x \in X$
and $(y = None \vee y \in Some\ `\ Y)$
and $(\forall\ (x,y) \in set\ \tau\ .\ x \in X \wedge (y = None \vee y \in Some\ `\ Y))$
by *blast*
then have $(\forall\ xy \in set\ \pi'\ .\ fst\ xy \in X \wedge snd\ xy \in Y)$
and $(\forall\ \pi''\ .\ prefix\ \pi''\ \pi' \longrightarrow \pi'' \in L)$
using *assms* **by** *auto*

have $(\forall\ xy \in set\ (map\ (\lambda(x,y)\ .\ (x,Some\ y))\ \pi')\ .\ fst\ xy \in X \wedge snd\ xy \in (\{None\} \cup Some\ `\ Y))$
using ‹$(\forall\ xy \in set\ \pi'\ .\ fst\ xy \in X \wedge snd\ xy \in Y)$›
by (*induction* π'; *auto*)
moreover have $set\ \pi = set\ (map\ (\lambda(x,y)\ .\ (x,Some\ y))\ \pi') \cup \{(x,y)\} \cup set\ \tau$
unfolding ‹$\pi = (map\ (\lambda(x,y)\ .\ (x,Some\ y))\ \pi')@[(x,y)]@\tau$›
by *simp*
ultimately have $(\forall\ xy \in set\ \pi\ .\ fst\ xy \in X \wedge snd\ xy \in (\{None\} \cup Some\ `\ Y))$
using ‹$x \in X$› ‹$(y = None \vee y \in Some\ `\ Y)$› ‹$(\forall\ (x,y) \in set\ \tau\ .\ x \in X \wedge (y = None \vee y \in Some\ `\ Y))$›
by *auto*
moreover have $\bigwedge\ \pi''\ .\ prefix\ \pi''\ \pi \Longrightarrow \pi'' \in\ ?L$
proof −
fix π'' **assume** *prefix* $\pi''\ \pi$
then obtain i **where** $\pi'' = take\ i\ \pi$
by (*metis append-eq-conv-conj prefix-def*)
then consider (*b1*) $i \le length\ \pi'$ |
(*b2*) $i > length\ \pi'$
by *linarith*
then show $\pi'' \in\ ?L$ **proof** *cases*
case *b1*

then have $i \leq$ *length* (*map* (λ(*x*,*y*) . (*x*,*Some y*)) π')
by *auto*
then have π'' = *map* (λ(*x*,*y*) . (*x*,*Some y*)) (*take i* π')
unfolding ‹π'' = *take i* π›
using ‹π = *map* (λ(*x*, *y*). (*x*, *Some y*)) π' @ [(*x*, *y*)] @ τ› *take-map* **by** *fastforce*
moreover have *take i* $\pi' \in L$
using ‹$\pi' \in L$› *take-is-prefix*
using ‹$\forall \pi''$. *prefix* π'' π' $\longrightarrow$ $\pi'' \in L$› **by** *blast*
ultimately have $\pi'' \in$ ((λ π . *map* (λ(*x*,*y*) . (*x*,*Some y*)) π) ‘ *L*)
by *simp*
then show $\pi'' \in$ *?L*
by *auto*
next
case *b2*
then have $i >$ *length* (*map* (λ(*x*,*y*) . (*x*,*Some y*)) π')
by *auto*

have $\bigwedge$ *k xs ys* . *k* > *length xs* $\Longrightarrow$ *take k* (*xs*@*ys*) = *xs*@(*take* (*k* − *length xs*) *ys*)
by *simp*
have *take-helper*: $\bigwedge$ *k xs y zs* . *k* > *length xs* $\Longrightarrow$ *take k* (*xs*@[*y*]@*zs*) = *xs*@[*y*]@(*take* (*k* − *length xs* − *1*) *zs*)
by (*metis One-nat-def Suc-pred* ‹$\bigwedge$*ys xs k*. *length xs* < *k* $\Longrightarrow$ *take k* (*xs* @ *ys*) = *xs* @ *take* (*k* − *length xs*) *ys*› *append-Cons append-Nil take-Suc-Cons zero-less-diff*)

have ∗∗: π'' = (*map* (λ(*x*,*y*) . (*x*,*Some y*)) π')@[(*x*,*y*)]@(*take* (*i* − *length* π' − *1*) τ)
unfolding ‹π = *map* (λ(*x*, *y*). (*x*, *Some y*)) π' @ [(*x*, *y*)] @ τ› ‹π'' = *take i* π›
using *take-helper*[*OF* ‹*i* > *length* (*map* (λ(*x*,*y*) . (*x*,*Some y*)) π')›] **by** *simp*

have ($\forall$ (*x*,*y*) $\in$ *set* (*take* (*i* − *length* π' − *1*) τ) . *x* $\in$ *X* $\wedge$ (*y* = *None* $\vee$ *y* $\in$ *Some* ‘ *Y*))
using ‹($\forall$ (*x*,*y*) $\in$ *set* τ . *x* $\in$ *X* $\wedge$ (*y* = *None* $\vee$ *y* $\in$ *Some* ‘ *Y*))›
by (*meson in-set-takeD*)
then show *?thesis*
unfolding ∗∗ *bottom-completion.simps*
using ‹$\pi' \in L$› ‹*out*[*L*,π',*x*] = {}› ‹*x* $\in$ *X*› ‹(*y* = *None* $\vee$ *y* $\in$ *Some* ‘ *Y*)›
by *blast*
qed
qed
ultimately show *?thesis* **by** *auto*
qed
qed
ultimately show *?thesis*
unfolding *is-language.simps* **by** *blast*

qed

fun *is-not-undefined* :: ('x,'y option) word ⇒ ('x,'y) language ⇒ bool **where**
  is-not-undefined π L = (π ∈ map (λ(x, y). (x, Some y)) ‘ L)

lemma *bottom-id* : map (λ(x,y) . (x, the y)) (map (λ(x, y). (x, Some y)) π) = π
  by (induction π; auto)

fun *maximum-prefix-with-property* :: ('a list ⇒ bool) ⇒ 'a list ⇒ 'a list **where**
  maximum-prefix-with-property P xs = (last (filter P (prefixes xs)))

lemma *maximum-prefix-with-property-props* :
  assumes ∃ ys ∈ set (prefixes xs) . P ys
shows P (maximum-prefix-with-property P xs)
  and (maximum-prefix-with-property P xs) ∈ set (prefixes xs)
  and ⋀ ys . prefix ys xs ⟹ P ys ⟹ length ys ≤ length (maximum-prefix-with-property P xs)
proof –

  have P (maximum-prefix-with-property P xs) ∧
        (maximum-prefix-with-property P xs) ∈ set (prefixes xs) ∧
      (∀ ys . prefix ys xs ⟶ P ys ⟶ length ys ≤ length (maximum-prefix-with-property P xs))
    using *assms*
  proof (induction xs rule: rev-induct)
    case *Nil*
    then show *?case* **by** *auto*
  next
    case (snoc x xs)
    have prefixes (xs @ [x]) = (prefixes xs)@[xs @ [x]]
      by *simp*

    show *?case* **proof** (cases P (xs@[x]))
      case *True*
      then have maximum-prefix-with-property P (xs @ [x]) = (xs @ [x])
      unfolding *maximum-prefix-with-property.simps* ‹prefixes (xs @ [x]) = (prefixes xs)@[xs @ [x]]›
        by *auto*
      show *?thesis*
        using *True*
        unfolding ‹maximum-prefix-with-property P (xs @ [x]) = (xs@[x])›
        using *in-set-prefixes prefix-length-le* **by** *blast*
    next
      case *False*

then have *maximum-prefix-with-property P* (*xs*@[*x*]) = *maximum-prefix-with-property P xs*
unfolding *maximum-prefix-with-property.simps* ‹*prefixes* (*xs* @ [*x*]) = (*prefixes xs*)@[*xs* @ [*x*]]›
by *auto*

have ∃ *a*∈*set* (*prefixes xs*). *P a*
using *snoc.prems False* **unfolding** ‹*prefixes* (*xs* @ [*x*]) = (*prefixes xs*)@[*xs* @ [*x*]]› **by** *auto*

show *?thesis*
using *snoc.IH*[*OF* ‹∃ *a*∈*set* (*prefixes xs*). *P a*›] *False*
unfolding ‹*maximum-prefix-with-property P* (*xs*@[*x*]) = *maximum-prefix-with-property P xs*›
unfolding ‹*prefixes* (*xs* @ [*x*]) = (*prefixes xs*)@[*xs* @ [*x*]]› **by** *auto*
qed
qed
then show *P* (*maximum-prefix-with-property P xs*)
and (*maximum-prefix-with-property P xs*) ∈ *set* (*prefixes xs*)
and ⋀ *ys* . *prefix ys xs* ⟹ *P ys* ⟹ *length ys* ≤ *length* (*maximum-prefix-with-property P xs*)
by *blast+*
qed

lemma *bottom-completion-out* :
assumes *is-language X Y L*
and *x* ∈ *X*
and π ∈ *bottom-completion X Y L*
shows *is-not-undefined* π *L* ⟹ *out*[*L*,*map* (λ(*x*,*y*) . (*x*, *the y*)) π,*x*] ≠ {} ⟹ *out*[*bottom-completion X Y L*, π, *x*] = *Some* ' *out*[*L*, *map* (λ(*x*,*y*) . (*x*, *the y*)) π, *x*]
and *is-not-undefined* π *L* ⟹ *out*[*L*,*map* (λ(*x*,*y*) . (*x*, *the y*)) π,*x*] = {} ⟹ *out*[*bottom-completion X Y L*, π, *x*] = {*None*} ∪ *Some* ' *Y*
and ¬ *is-not-undefined* π *L* ⟹ *out*[*bottom-completion X Y L*, π, *x*] = {*None*} ∪ *Some* ' *Y*
proof −

let *?L* = *bottom-completion X Y L*
define *L'a* **where** *L'a* = ((λ π . *map* (λ(*x*,*y*) . (*x*,*Some y*)) π) ' *L*)
define *L'b* **where** *L'b* = {(*map* (λ(*x*,*y*) . (*x*,*Some y*)) π)@[(*x*,*y*)]@τ | π *x y* τ . π ∈ *L* ∧ *out*[*L*,π,*x*] = {} ∧ *x* ∈ *X* ∧ (*y* = *None* ∨ *y* ∈ *Some* ' *Y*) ∧ (∀ (*x*,*y*) ∈ *set* τ . *x* ∈ *X* ∧ (*y* = *None* ∨ *y* ∈ *Some* ' *Y*))}
have *?L* = *L'a* ∪ *L'b*
unfolding *L'a-def L'b-def bottom-completion.simps* **by** *blast*
then have *out*[*?L*, π, *x*] = {*y*. π @ [(*x*, *y*)] ∈ *L'a*} ∪ {*y*. π @ [(*x*, *y*)] ∈ *L'b*}
unfolding *outputs.simps language-for-state.simps* **by** *blast*

have *is-language X* ({*None*} ∪ *Some* ' *Y*) *?L*
using *bottom-completion-is-language*[*OF assms*(*1*)] **.**

show *is-not-undefined* π *L* $\Longrightarrow$ *out*[*L*,*map* (λ(*x*,*y*) . (*x*, *the y*)) π,*x*] $\neq$ {} $\Longrightarrow$ *out*[*bottom-completion X Y L*, π, *x*] = *Some* ‘ *out*[*L*, *map* (λ(*x*,*y*) . (*x*, *the y*)) π, *x*]
and *is-not-undefined* π *L* $\Longrightarrow$ *out*[*L*,*map* (λ(*x*,*y*) . (*x*, *the y*)) π,*x*] = {} $\Longrightarrow$ *out*[*bottom-completion X Y L*, π, *x*] = {*None*} $\cup$ *Some* ‘ *Y*
proof –
assume *is-not-undefined* π *L*
then obtain π' **where** π = *map* (λ(*x*, *y*). (*x*, *Some y*)) π' **and** $\pi' \in L$
by *auto*
then have *map* (λ(*x*, *y*). (*x*, *the y*)) $\pi = \pi'$
using *bottom-id* **by** *auto*

have {*y*. π @ [(*x*, *y*)] $\in$ *L'a*} = *Some* ‘ *out*[*L*,*map* (λ(*x*,*y*) . (*x*, *the y*)) π,*x*]
proof
show {*y*. π @ [(*x*, *y*)] $\in$ *L'a*} $\subseteq$ *Some* ‘ *out*[*L*,*map* (λ(*x*, *y*). (*x*, *the y*)) π,*x*]
proof
fix *y* **assume** *y* $\in$ {*y*. π @ [(*x*, *y*)] $\in$ *L'a*}
then have π @ [(*x*, *y*)] $\in$ *L'a* **by** *auto*
then obtain π' **where** π @ [(*x*, *y*)] = *map* (λ(*x*,*y*) . (*x*,*Some y*)) π' **and** π' $\in$ *L*
unfolding *L'a-def* **by** *blast*
then have *length* (π @ [(*x*, *y*)]) = *length* π'
by *auto*
then obtain γ' *xy* **where** $\pi' = \gamma'$@[*xy*]
by (*metis add.right-neutral dual-order.strict-iff-not length-append-singleton less-add-Suc2 rev-exhaust take0 take-all-iff*)
then have (*x*,*y*) = (λ(*x*, *y*). (*x*, *Some y*)) *xy*
using ‹π @ [(*x*, *y*)] = *map* (λ(*x*, *y*). (*x*, *Some y*)) π'› **unfolding** ‹$\pi' = \gamma'$@[*xy*]› **by** *auto*
then have *y* = *Some* (*snd xy*) **and** *xy* = (*x*,*snd xy*)
by (*simp add*: *split-beta*)+
moreover define *y'* **where** *y'* = *snd xy*
ultimately have *y* = *Some y'* **and** *xy* = (*x*,*y'*)
by *auto*

have *map* (λ(*x*, *y*). (*x*, *the y*)) $\pi = \gamma'$
using ‹π @ [(*x*, *y*)] = *map* (λ(*x*,*y*) . (*x*,*Some y*)) π'› **unfolding** ‹$\pi' = \gamma'$@[*xy*]›
using *bottom-id* **by** *auto*

have *y'* $\in$ *out*[*L*,*map* (λ(*x*, *y*). (*x*, *the y*)) π,*x*]
using ‹$\pi' \in L$›
unfolding ‹*map* (λ(*x*, *y*). (*x*, *the y*)) $\pi = \gamma'$› ‹$\pi' = \gamma'$@[*xy*]› ‹*xy* = (*x*,*y'*)› **by** *auto*
then show *y* $\in$ *Some* ‘ *out*[*L*,*map* (λ(*x*, *y*). (*x*, *the y*)) π,*x*]
unfolding ‹*y* = *Some y'*› **by** *blast*
qed
show *Some* ‘ *out*[*L*,*map* (λ(*x*, *y*). (*x*, *the y*)) π,*x*] $\subseteq$ {*y*. π @ [(*x*, *y*)] $\in$ *L'a*}

proof
fix *y* **assume** $y \in Some$ ‘ $out[L,map\ (\lambda(x,\ y).\ (x,\ the\ y))\ \pi,x]$
then obtain y' **where** $y = Some\ y'$ **and** $y' \in out[L,map\ (\lambda(x,\ y).\ (x,\ the\ y))\ \pi,x]$
by *blast*
then have $\pi'@[(x,y')] \in L$
unfolding ‹$map\ (\lambda(x,\ y).\ (x,\ the\ y))\ \pi = \pi'$› **by** *auto*
then show $y \in \{y.\ \pi\ @\ [(x,\ y)] \in L'a\}$
unfolding *L′a-def* ‹$\pi = map\ (\lambda(x,\ y).\ (x,\ Some\ y))\ \pi'$›
using ‹$y = Some\ y'$› *image-iff* **by** *fastforce*
qed
qed

show $out[L,map\ (\lambda(x,y)\ .\ (x,\ the\ y))\ \pi,x] \neq \{\} \Longrightarrow out[bottom\text{-}completion\ X\ Y\ L,\ \pi,\ x] = Some$ ‘ $out[L,\ map\ (\lambda(x,y)\ .\ (x,\ the\ y))\ \pi,\ x]$
proof −
assume $out[L,map\ (\lambda(x,y)\ .\ (x,\ the\ y))\ \pi,x] \neq \{\}$
then obtain *ya* **where** $\pi'@[(x,ya)] \in L$
using ‹$\pi' \in L$› **unfolding** ‹$map\ (\lambda(x,y)\ .\ (x,\ the\ y))\ \pi = \pi'$› **by** *auto*

have $\{y.\ \pi\ @\ [(x,\ y)] \in L'b\} = \{\}$
proof (*rule ccontr*)
assume $\{y.\ \pi\ @\ [(x,\ y)] \in L'b\} \neq \{\}$
then obtain *y* **where** $\pi\ @\ [(x,\ y)] \in L'b$ **by** *blast*
then obtain $\pi''\ x'\ y'\ \tau$ **where** $\pi\ @\ [(x,\ y)] = (map\ (\lambda(x,y)\ .\ (x,Some\ y))\ \pi'')@[(x',y')]@\tau$
and $\pi'' \in L$
and $out[L,\pi'',x'] = \{\}$
and $x' \in X$
and $(y' = None \lor y' \in Some$ ‘ $Y)$
and $(\forall\ (x,y) \in set\ \tau\ .\ x \in X \land (y = None \lor y \in Some$ ‘ $Y))$
unfolding *L′b-def*
by *blast*

have $\bigwedge\ y''\ .\ \pi''@[(x',y'')] \notin L$
using ‹$\pi'' \in L$› ‹$out[L,\pi'',x'] = \{\}$›
unfolding *outputs.simps language-for-state.simps* **by** *force*

have $length\ \pi' = length\ \pi''$
proof −
have $length\ \pi' = length\ \pi$
using ‹$map\ (\lambda(x,\ y).\ (x,\ the\ y))\ \pi = \pi'$› *length-map* **by** *blast*

have $\neg\ length\ \pi' < length\ \pi''$
proof

assume *length* π' < *length* π''
then have *length* π'' = *Suc* (*length* π')
by (*metis* (*no-types*, *lifting*) *One-nat-def* ‹π @ [(x, y)] = *map* (λ(x, y). (x, *Some* y)) π'' @ [(x', y')] @ τ› ‹*length* π' = *length* π› *add-diff-cancel-left'* *length-append length-append-singleton length-map list.size*(*3*) *not-less-eq plus-1-eq-Suc zero-less-Suc zero-less-diff*)
then have *length* π'' > *length* π
by (*simp add*: ‹π = *map* (λ(x, y). (x, *Some* y)) π'›)
then show *False*
by (*metis* (*no-types*, *lifting*) *One-nat-def* ‹π @ [(x, y)] = *map* (λ(x, y). (x, *Some* y)) π'' @ [(x', y')] @ τ› *length-Cons length-append length-append-singleton length-map less-add-same-cancel1 list.size*(*3*) *not-less-eq plus-1-eq-Suc zero-less-Suc*)

qed
moreover have ¬ *length* π'' < *length* π'
proof
assume *length* π'' < *length* π'
then have *prefix* ((*map* (λ(x,y) . (x,*Some* y)) π'')@[(x',y')]) (*map* (λ(x,y) . (x,*Some* y)) π')
by (*metis* (*no-types*, *lifting*) ‹π = *map* (λ(x, y). (x, *Some* y)) π'› ‹π @ [(x, y)] = *map* (λ(x, y). (x, *Some* y)) π'' @ [(x', y')] @ τ› *append.assoc length-append-singleton length-map linorder-not-le not-less-eq prefixI prefix-length-prefix*)

then have *prefix* π'' π'
by (*metis append-prefixD bottom-id map-mono-prefix*)
then have *take* (*length* π'') π' = π''
by (*metis append-eq-conv-conj prefix-def*)

have (x',y') = (((*map* (λ(x,y) . (x,*Some* y)) π'')@[(x',y')])) ! (*length* π'')
by (*induction* π'' *arbitrary*: x' y'; *auto*)
then have (x',y') = (*map* (λ(x,y) . (x,*Some* y)) π') ! (*length* π'')
by (*metis* (*no-types*, *lifting*) ‹π = *map* (λ(x, y). (x, *Some* y)) π'› ‹π @ [(x, y)] = *map* (λ(x, y). (x, *Some* y)) π'' @ [(x', y')] @ τ› ‹*length* π'' < *length* π'› *append-Cons length-map nth-append nth-append-length*)
then have *fst* (π' ! (*length* π'')) = x'
by (*simp add*: ‹*length* π'' < *length* π'› *split-beta*)

have *out*[L, *take* (*length* π'') π', *fst* (π' ! (*length* π''))] = {}
unfolding ‹*take* (*length* π'') π' = π''› ‹*fst* (π' ! (*length* π'')) = x'›
using ‹*out*[L,π'',x'] = {}› .
moreover have $\bigwedge$ i . i < *length* π' $\Longrightarrow$ *out*[L, *take* i π', *fst* (π' ! i)] ≠ {}
using *prefix-executable*[*OF assms*(*1*) ‹π' ∈ L›]
by (*meson outputs-executable*)
ultimately show *False*
using ‹*length* π'' < *length* π'› **by** *blast*
qed
ultimately show *?thesis*
by *simp*
qed

then have $\pi'' = \pi'$
by (*metis* ‹π @ [(*x*, *y*)] = *map* (λ(*x*, *y*). (*x*, *Some y*)) π'' @ [(*x'*, *y'*)] @ τ› ‹*map* (λ(*x*, *y*). (*x*, *the y*)) $\pi = \pi'$› *append-eq-append-conv bottom-id length-map*)

show *False*
using ‹π @ [(*x*, *y*)] = *map* (λ(*x*, *y*). (*x*, *Some y*)) π'' @ [(*x'*, *y'*)] @ τ› ‹π'' = π'› ‹*map* (λ(*x*, *y*). (*x*, *the y*)) $\pi = \pi'$› ‹*out*[*L*,π'',*x'*] = {}› ‹*out*[*L*,*map* (λ(*x*, *y*). (*x*, *the y*)) π,*x*] $\neq$ {}›
by *force*
qed
then show *?thesis*
using ‹*out*[*bottom-completion X Y L*,π,*x*] = {*y*. π @ [(*x*, *y*)] $\in$ *L'a*} $\cup$ {*y*. π @ [(*x*, *y*)] $\in$ *L'b*}›
using ‹{*y*. π @ [(*x*, *y*)] $\in$ *L'a*} = *Some* ‘ *out*[*L*,*map* (λ(*x*,*y*) . (*x*, *the y*)) π,*x*]›
by *force*
qed

show *out*[*L*,*map* (λ(*x*,*y*) . (*x*, *the y*)) π,*x*] = {} $\Longrightarrow$ *out*[*bottom-completion X Y L*, π, *x*] = {*None*} $\cup$ *Some* ‘ *Y*
proof –
assume *out*[*L*,*map* (λ(*x*,*y*) . (*x*, *the y*)) π,*x*] = {}
then have {*y*. π @ [(*x*, *y*)] $\in$ *L'a*} = {}
unfolding ‹{*y*. π @ [(*x*, *y*)] $\in$ *L'a*} = *Some* ‘ *out*[*L*,*map* (λ(*x*,*y*) . (*x*, *the y*)) π,*x*]› **by** *blast*
moreover have {*y*. π @ [(*x*, *y*)] $\in$ *L'b*} = {*None*} $\cup$ *Some* ‘ *Y*
proof
show {*y*. π @ [(*x*, *y*)] $\in$ *L'b*} $\subseteq$ {*None*} $\cup$ *Some* ‘ *Y*
proof
fix *y* **assume** *y* $\in$ {*y*. π @ [(*x*, *y*)] $\in$ *L'b*}
then have π @ [(*x*, *y*)] $\in$ *L'b* **by** *blast*
then obtain π'' *x'* *y'* τ **where** π @ [(*x*, *y*)] = (*map* (λ(*x*,*y*) . (*x*,*Some y*)) π'')@[(*x'*,*y'*)]@τ
and $\pi'' \in L$
and *out*[*L*,π'',*x'*] = {}
and *x'* $\in$ *X*
and (*y'* = *None* $\vee$ *y'* $\in$ *Some* ‘ *Y*)
and ($\forall$ (*x*,*y*) $\in$ *set* τ . *x* $\in$ *X* $\wedge$ (*y* = *None* $\vee$ *y* $\in$ *Some* ‘ *Y*))
unfolding *L'b-def*
by *blast*

show *y* $\in$ {*None*} $\cup$ *Some* ‘ *Y*
by (*metis* (*no-types*, *lifting*) *Un-insert-right* ‹*out*[*bottom-completion X Y L*,π,*x*] = {*y*. π @ [(*x*, *y*)] $\in$ *L'a*} $\cup$ {*y*. π @ [(*x*, *y*)] $\in$ *L'b*}› ‹*y* $\in$ {*y*. π @ [(*x*, *y*)] $\in$ *L'b*}› *assms*(*1*) *bottom-completion-is-language insert-subset mk-disjoint-insert outputs-in-alphabet*)
qed

```
        show {None} ∪ Some ‘ Y ⊆ {y. π @ [(x, y)] ∈ L'b}
        proof
          fix y assume y ∈ {None} ∪ Some ‘ Y

          have π @ [(x, y)] = map (λ(x, y). (x, Some y)) π' @ [(x, y)] @ []
            by (simp add: ‹π = map (λ(x, y). (x, Some y)) π'›)
          moreover note ‹π' ∈ L›
          moreover have out[L,π',x] = {}
           using ‹out[L,map (λ(x,y) . (x, the y)) π,x] = {}› unfolding ‹map (λ(x,y)
. (x, the y)) π = π'› .
          moreover note ‹x ∈ X›
          moreover have (y = None ∨ y ∈ Some ‘ Y)
            using ‹y ∈ {None} ∪ Some ‘ Y› by blast
          moreover have (∀ (x, y)∈set []. x ∈ X ∧ (y = None ∨ y ∈ Some ‘ Y))
            by simp
          ultimately show y ∈ {y. π @ [(x, y)] ∈ L'b}
            unfolding L'b-def by blast
        qed
      qed
      ultimately show ?thesis
        using ‹out[bottom-completion X Y L,π,x] = {y. π @ [(x, y)] ∈ L'a} ∪ {y. π
@ [(x, y)] ∈ L'b}›
         using ‹{y. π @ [(x, y)] ∈ L'a} = Some ‘ out[L,map (λ(x,y) . (x, the y))
π,x]›
        by force
    qed
  qed

  show ¬ is-not-undefined π L ⟹ out[bottom-completion X Y L,π,x] = {None}
∪ Some ‘ Y
  proof −
    assume ¬ is-not-undefined π L
    then have π ∉ L'a
      unfolding L'a-def by auto

    have {y. π @ [(x, y)] ∈ L'a} = {}
    proof (rule ccontr)
      assume {y. π @ [(x, y)] ∈ L'a} ≠ {}
      then obtain y where π @ [(x, y)] ∈ L'a by blast
      then obtain γ where π @ [(x, y)] = map (λ(x, y). (x, Some y)) γ and γ ∈
L
        unfolding L'a-def by blast
      then have π = map (λ(x, y). (x, Some y)) (butlast γ)
        by (metis (mono-tags, lifting) butlast-snoc map-butlast)
      moreover have butlast γ ∈ L
        using ‹γ ∈ L› assms(1)
        by (simp add: prefixeq-butlast)
      ultimately show False
        using ‹π ∉ L'a›
```

using *L'a-def* **by** *blast*
qed
then have $out[?L, \pi, x] = \{y.\ \pi\ @\ [(x, y)] \in L'b\}$
using ‹$out[bottom\text{-}completion\ X\ Y\ L,\pi,x] = \{y.\ \pi\ @\ [(x, y)] \in L'a\} \cup \{y.\ \pi\ @\ [(x, y)] \in L'b\}$› **by** *blast*
also have $\ldots = \{None\} \cup Some\ `\ Y$
proof
show $\{y.\ \pi\ @\ [(x, y)] \in L'b\} \subseteq \{None\} \cup Some\ `\ Y$
proof
fix y **assume** $y \in \{y.\ \pi\ @\ [(x, y)] \in L'b\}$
then obtain $\pi'\ x'\ y'\ \tau$ **where** $\pi\ @\ [(x, y)] = (map\ (\lambda(x,y)\ .\ (x,Some\ y))\ \pi')@[(x',y')]@\tau$
and $\pi' \in L$
and $out[L,\pi',x'] = \{\}$
and $x' \in X$
and $(y' = None \vee y' \in Some\ `\ Y)$
and $(\forall\ (x,y) \in set\ \tau\ .\ x \in X \wedge (y = None \vee y \in Some\ `\ Y))$
unfolding *L'b-def*
by *blast*

have $(x,y) \in set\ ([(x',y')]@\tau)$
by (*metis* ‹$\pi\ @\ [(x, y)] = map\ (\lambda(x, y).\ (x, Some\ y))\ \pi'\ @\ [(x', y')]\ @\ \tau$› *append-is-Nil-conv last-appendR last-in-set last-snoc length-Cons list.size*(3) *nat.simps*(3))
then show $y \in \{None\} \cup Some\ `\ Y$
using ‹$(y' = None \vee y' \in Some\ `\ Y)$› ‹$(\forall\ (x,y) \in set\ \tau\ .\ x \in X \wedge (y = None \vee y \in Some\ `\ Y))$› **by** *auto*
qed
show $\{None\} \cup Some\ `\ Y \subseteq \{y.\ \pi\ @\ [(x, y)] \in L'b\}$
proof
fix y **assume** $y \in \{None\} \cup Some\ `\ Y$

have $\pi \in L'b$
using ‹$\pi \notin L'a$› ‹$?L = L'a \cup L'b$› *assms*(3) **by** *fastforce*
then obtain $\pi'\ x'\ y'\ \tau$ **where** $\pi = (map\ (\lambda(x,y)\ .\ (x,Some\ y))\ \pi')@[(x',y')]@\tau$

and $\pi' \in L$
and $out[L,\pi',x'] = \{\}$
and $x' \in X$
and $(y' = None \vee y' \in Some\ `\ Y)$
and $(\forall\ (x,y) \in set\ \tau\ .\ x \in X \wedge (y = None \vee y \in Some\ `\ Y))$
unfolding *L'b-def*
by *blast*

have $\pi\ @\ [(x,y)] = (map\ (\lambda(x,y)\ .\ (x,Some\ y))\ \pi')@[(x',y')]@(\tau@[(x,y)])$
unfolding ‹$\pi = (map\ (\lambda(x,y)\ .\ (x,Some\ y))\ \pi')@[(x',y')]@\tau$› **by** *auto*
moreover note ‹$\pi' \in L$› **and** ‹$out[L,\pi',x'] = \{\}$› **and** ‹$x' \in X$› **and** ‹$(y'$

= *None* ∨ *y'* ∈ *Some* ' *Y*)›
        moreover have (∀ (*x*,*y*) ∈ *set* (τ@[(*x*,*y*)]) . *x* ∈ *X* ∧ (*y* = *None* ∨ *y* ∈ *Some* ' *Y*))
          using ‹∀ (*x*,*y*) ∈ *set* τ . *x* ∈ *X* ∧ (*y* = *None* ∨ *y* ∈ *Some* ' *Y*)› ‹*y* ∈ {*None*} ∪ *Some* ' *Y*› ‹*x* ∈ *X*›
          by *auto*
        ultimately show *y* ∈ {*y*. π @ [(*x*, *y*)] ∈ *L'b*}
          unfolding *L'b-def* **by** *blast*
      qed
    qed
    finally show *out*[*?L*,π,*x*] = {*None*} ∪ *Some* ' *Y* **.**
  qed
qed

theorem *quasired-via-red* :
  assumes *is-language X Y L1*
  and *is-language X Y L2*
shows (*L1* ⪯[*X*,*quasired Y*] *L2*) ⟷ ((*bottom-completion X Y L1*) ⪯[*X*, *red* ({*None*} ∪ *Some* ' *Y*)] (*bottom-completion X Y L2*))
proof −

  define *L1'* **where** *L1'* = *bottom-completion X Y L1*
  define *L2'* **where** *L2'* = *bottom-completion X Y L2*

  define *L1'a* **where** *L1'a* = ((λ π . *map* (λ(*x*,*y*) . (*x*,*Some y*)) π) ' *L1*)
  define *L1'b* **where** *L1'b* = {(*map* (λ(*x*,*y*) . (*x*,*Some y*)) π)@[(*x*,*y*)]@τ | π *x y* τ . π ∈ *L1* ∧ *out*[*L1*,π,*x*] = {} ∧ *x* ∈ *X* ∧ (*y* = *None* ∨ *y* ∈ *Some* ' *Y*) ∧ (∀ (*x*,*y*) ∈ *set* τ . *x* ∈ *X* ∧ (*y* = *None* ∨ *y* ∈ *Some* ' *Y*))}
  define *L2'a* **where** *L2'a* = ((λ π . *map* (λ(*x*,*y*) . (*x*,*Some y*)) π) ' *L2*)
  define *L2'b* **where** *L2'b* = {(*map* (λ(*x*,*y*) . (*x*,*Some y*)) π)@[(*x*,*y*)]@τ | π *x y* τ . π ∈ *L2* ∧ *out*[*L2*,π,*x*] = {} ∧ *x* ∈ *X* ∧ (*y* = *None* ∨ *y* ∈ *Some* ' *Y*) ∧ (∀ (*x*,*y*) ∈ *set* τ . *x* ∈ *X* ∧ (*y* = *None* ∨ *y* ∈ *Some* ' *Y*))}

  let *?L1* = *bottom-completion X Y L1*

  have *?L1* = *L1'a* ∪ *L1'b*
    unfolding *L1'a-def L1'b-def bottom-completion.simps* **by** *blast*
  then have ⋀ π *x* . *out*[*?L1*, π, *x*] = {*y*. π @ [(*x*, *y*)] ∈ *L1'a*} ∪ {*y*. π @ [(*x*, *y*)] ∈ *L1'b*}
    unfolding *outputs.simps language-for-state.simps* **by** *blast*

  let *?L2* = *bottom-completion X Y L2*

  have *?L2* = *L2'a* ∪ *L2'b*
    unfolding *L2'a-def L2'b-def bottom-completion.simps* **by** *blast*
  then have ⋀ π *x* . *out*[*?L2*, π, *x*] = {*y*. π @ [(*x*, *y*)] ∈ *L2'a*} ∪ {*y*. π @ [(*x*, *y*)] ∈ *L2'b*}

unfolding *outputs.simps language-for-state.simps* **by** *blast*

have *is-language X* ({*None*} ∪ *Some* ‘ *Y*) *?L1*
using *bottom-completion-is-language*[*OF assms*(*1*)] **.**
have *is-language X* ({*None*} ∪ *Some* ‘ *Y*) *?L2*
using *bottom-completion-is-language*[*OF assms*(*2*)] **.**
then have ⋀ *π x* . *out*[*bottom-completion X Y L2*,*π*,*x*] ⊆ {*None*} ∪ *Some* ‘ *Y*
by (*meson outputs-in-alphabet*)

have (*?L1* ⪯[*X*, *red* ({*None*} ∪ *Some* ‘ *Y*)] *?L2*) = (∀ *π* ∈ *?L1* ∩ *?L2* . ∀ *x* ∈ *X* . *out*[*?L1*,*π*,*x*] ⊆ *out*[*?L2*,*π*,*x*])
unfolding *type-1-conforms.simps red.simps*
using ‹⋀ *π x* . *out*[*bottom-completion X Y L2*,*π*,*x*] ⊆ {*None*} ∪ *Some* ‘ *Y*› **by** *force*
also have . . . = (∀ *π* ∈ *?L1* ∩ *?L2* . ∀ *x* ∈ *X* . (*out*[*?L2*,*π*,*x*] = {*None*} ∪ *Some* ‘ *Y* ∨ (*out*[*?L1*,*π*,*x*] ≠ {} ∧ *out*[*?L1*,*π*,*x*] ⊆ *out*[*?L2*,*π*,*x*])))
by (*metis* (*no-types*, *lifting*) *IntD1* ‹*is-language X* ({*None*} ∪ *Some* ‘ *Y*) (*bottom-completion X Y L1*)› ‹*is-language X* ({*None*} ∪ *Some* ‘ *Y*) (*bottom-completion X Y L2*)› *assms*(*1*) *bottom-completion-out*(*1*) *bottom-completion-out*(*2*) *bottom-completion-out*(*3*) *image-is-empty outputs-in-alphabet subset-antisym*)
also have . . . = (∀ *π* ∈ *?L1* ∩ *?L2* . ∀ *x* ∈ *X* . (*out*[*?L2*,*π*,*x*] = {*None*} ∪ *Some* ‘ *Y* ∨ (*is-not-undefined π L1* ∧ *is-not-undefined π L2* ∧ *out*[*L1*,*map* (λ(*x*,*y*) . (*x*, *the y*)) *π*,*x*] ≠ {} ∧ *out*[*L1*,*map* (λ(*x*,*y*) . (*x*, *the y*)) *π*,*x*] ⊆ *out*[*L2*,*map* (λ(*x*,*y*) . (*x*, *the y*)) *π*,*x*])))
proof −
have ⋀ *π x* . *π* ∈ *?L1* ∩ *?L2* ⟹ *x* ∈ *X* ⟹ *out*[*?L2*,*π*,*x*] ≠ {*None*} ∪ *Some* ‘ *Y* ⟹
(*out*[*?L1*,*π*,*x*] ≠ {} ∧ *out*[*?L1*,*π*,*x*] ⊆ *out*[*?L2*,*π*,*x*]) = (*is-not-undefined π L1* ∧ *is-not-undefined π L2* ∧ *out*[*L1*,*map* (λ(*x*,*y*) . (*x*, *the y*)) *π*,*x*] ≠ {} ∧ *out*[*L1*,*map* (λ(*x*,*y*) . (*x*, *the y*)) *π*,*x*] ⊆ *out*[*L2*,*map* (λ(*x*,*y*) . (*x*, *the y*)) *π*,*x*])
proof −
fix *π x* **assume** *π* ∈ *?L1* ∩ *?L2* **and** *x* ∈ *X* **and** *out*[*?L2*,*π*,*x*] ≠ {*None*} ∪ *Some* ‘ *Y*
then have *π* ∈ *?L1* **and** *π* ∈ *?L2* **by** *blast+*

have *is-not-undefined π L2*
using *bottom-completion-out*[*OF assms*(*2*) ‹*x* ∈ *X*› ‹*π* ∈ *?L2*›]
using ‹*out*[*bottom-completion X Y L2*,*π*,*x*] ≠ {*None*} ∪ *Some* ‘ *Y*› **by** *fastforce*
have *out*[*L2*,*map* (λ(*x*, *y*). (*x*, *the y*)) *π*,*x*] ≠ {}
using *bottom-completion-out*(*1*,*2*)[*OF assms*(*2*) ‹*x* ∈ *X*› ‹*π* ∈ *?L2*›]
using ‹*is-not-undefined π L2*› ‹*out*[*bottom-completion X Y L2*,*π*,*x*] ≠ {*None*} ∪ *Some* ‘ *Y*› **by** *blast*

show (*out*[*?L1*,*π*,*x*] ≠ {} ∧ *out*[*?L1*,*π*,*x*] ⊆ *out*[*?L2*,*π*,*x*]) = (*is-not-undefined π L1* ∧ *is-not-undefined π L2* ∧ *out*[*L1*,*map* (λ(*x*,*y*) . (*x*, *the y*)) *π*,*x*] ≠ {} ∧ *out*[*L1*,*map* (λ(*x*,*y*) . (*x*, *the y*)) *π*,*x*] ⊆ *out*[*L2*,*map* (λ(*x*,*y*) . (*x*, *the y*)) *π*,*x*])
proof (*cases is-not-undefined π L1*)
case *False*

then have *out*[*?L1*,π,*x*] = {*None*} ∪ *Some* ' *Y*
by (*meson IntD1* ‹π ∈ *bottom-completion X Y L1* ∩ *bottom-completion X Y L2*› ‹*x* ∈ *X*› *assms*(*1*) *bottom-completion-out*(*3*))
then have ¬ (*out*[*?L1*,π,*x*] ⊆ *out*[*?L2*,π,*x*])
by (*metis* ‹*is-language X* ({*None*} ∪ *Some* ' *Y*) (*bottom-completion X Y L2*)› ‹*out*[*bottom-completion X Y L2*,π,*x*] ≠ {*None*} ∪ *Some* ' *Y*› *outputs-in-alphabet subset-antisym*)
then show *?thesis*
using *False* **by** *presburger*
next
case *True*

have (*out*[*?L1*,π,*x*] ≠ {} ∧ *out*[*?L1*,π,*x*] ⊆ *out*[*?L2*,π,*x*]) = (*out*[*L1*,*map* (λ(*x*,*y*) . (*x*, *the y*)) π,*x*] ≠ {} ∧ *out*[*L1*,*map* (λ(*x*,*y*) . (*x*, *the y*)) π,*x*] ⊆ *out*[*L2*,*map* (λ(*x*,*y*) . (*x*, *the y*)) π,*x*])
proof (*cases out*[*L1*,*map* (λ(*x*, *y*). (*x*, *the y*)) π,*x*] = {})
case *True*

have ¬ (*out*[*?L1*,π,*x*] ≠ {} ∧ *out*[*?L1*,π,*x*] ⊆ *out*[*?L2*,π,*x*])
unfolding *bottom-completion-out*(*2*)[*OF assms*(*1*) ‹*x* ∈ *X*› ‹π ∈ *?L1*› ‹*is-not-undefined* π *L1*› *True*]
by (*meson* ‹⋀*x* π. *out*[*bottom-completion X Y L2*,π,*x*] ⊆ {*None*} ∪ *Some* ' *Y*› ‹*out*[*bottom-completion X Y L2*,π,*x*] ≠ {*None*} ∪ *Some* ' *Y*› *subset-antisym*)
moreover have ¬ (*out*[*L1*,*map* (λ(*x*,*y*) . (*x*, *the y*)) π,*x*] ≠ {} ∧ *out*[*L1*,*map* (λ(*x*,*y*) . (*x*, *the y*)) π,*x*] ⊆ *out*[*L2*,*map* (λ(*x*,*y*) . (*x*, *the y*)) π,*x*])
using *True* **by** *simp*
ultimately show *?thesis* **by** *blast*
next
case *False*
show *?thesis*
unfolding *bottom-completion-out*(*1*)[*OF assms*(*1*) ‹*x* ∈ *X*› ‹π ∈ *?L1*› ‹*is-not-undefined* π *L1*› *False*]
unfolding *bottom-completion-out*(*1*)[*OF assms*(*2*) ‹*x* ∈ *X*› ‹π ∈ *?L2*› ‹*is-not-undefined* π *L2*› ‹*out*[*L2*,*map* (λ(*x*, *y*). (*x*, *the y*)) π,*x*] ≠ {}›]
by *blast*
qed
then show *?thesis*
using ‹*is-not-undefined* π *L1*› ‹*is-not-undefined* π *L2*›
by *blast*
qed
qed
then show *?thesis*
by *meson*
qed
also have . . . = ((∀ π ∈ *?L1* ∩ *?L2* . ∀ *x* ∈ *X* . ¬ *is-not-undefined* π *L1* ⟶ *is-not-undefined* π *L2* ⟶ *out*[*?L2*,π,*x*] = {*None*} ∪ *Some* ' *Y*)
∧ (∀ π ∈ *L1* ∩ *L2* . ∀ *x* ∈ *X* . *out*[*L2*,π,*x*] = {} ∨ (*out*[*L1*,π,*x*] ≠ {} ∧ *out*[*L1*,π,*x*] ⊆ *out*[*L2*,π,*x*])))
(**is** *?A* = *?B*)

proof
show $?A \Longrightarrow ?B$
proof −
assume $?A$

have $\bigwedge \pi\ x\ .\ \pi \in ?L1 \cap ?L2 \Longrightarrow x \in X \Longrightarrow \neg$ *is-not-undefined* $\pi\ L1 \Longrightarrow$ *is-not-undefined* $\pi\ L2 \Longrightarrow out[?L2,\pi,x] = \{None\} \cup Some\ `\ Y$
using ‹?A› **by** *blast*
moreover have $\bigwedge \pi\ x\ .\ \pi \in L1 \cap L2 \Longrightarrow x \in X \Longrightarrow out[L2,\pi,x] = \{\} \vee (out[L1,\pi,x] \neq \{\} \wedge out[L1,\pi,x] \subseteq out[L2,\pi,x])$
proof −
fix $\pi\ x$ **assume** $\pi \in L1 \cap L2$ **and** $x \in X$

let $?\pi = map\ (\lambda(x, y).\ (x, Some\ y))\ \pi$

have *is-not-undefined* $?\pi\ L1$ **and** *is-not-undefined* $?\pi\ L2$
using ‹$\pi \in L1 \cap L2$› **by** *auto*
then have $?\pi \in ?L1$ **and** $?\pi \in ?L2$
by *auto*

show $out[L2,\pi,x] = \{\} \vee (out[L1,\pi,x] \neq \{\} \wedge out[L1,\pi,x] \subseteq out[L2,\pi,x])$
proof (*cases* $out[L2,\pi,x] = \{\}$)
case *True*
then show *?thesis* **by** *auto*
next
case *False*
then have $out[bottom\text{-}completion\ X\ Y\ L2,?\pi,x] \neq \{None\} \cup Some\ `\ Y$
using *bottom-completion-out*(1)[*OF assms*(2) ‹$x \in X$› ‹$?\pi \in ?L2$› ‹*is-not-undefined* $?\pi\ L2$›]
unfolding *bottom-id*
by *force*
then have $out[L1,map\ (\lambda(x, y).\ (x, the\ y))\ ?\pi,x] \neq \{\} \wedge out[L1,map\ (\lambda(x, y).\ (x, the\ y))\ ?\pi,x] \subseteq out[L2,map\ (\lambda(x, y).\ (x, the\ y))\ ?\pi,x]$
using ‹?A›
using ‹$?\pi \in ?L1$› ‹$?\pi \in ?L2$› ‹$x \in X$›
by *blast*
then show $out[L2,\pi,x] = \{\} \vee (out[L1,\pi,x] \neq \{\} \wedge out[L1,\pi,x] \subseteq out[L2,\pi,x])$
unfolding *bottom-id* **by** *blast*
qed
qed
ultimately show $?B$
by *meson*
qed
show $?B \Longrightarrow ?A$
proof −
assume $?B$

have $\bigwedge \pi\ x\ .\ \pi \in ?L1 \cap ?L2 \Longrightarrow x \in X \Longrightarrow out[?L2,\pi,x] = \{None\} \cup Some$

‘ *Y* ∨ *is-not-undefined* π *L1* ∧ *is-not-undefined* π *L2* ∧ *out*[*L1*,*map* (λ(*x*, *y*). (*x*, *the y*)) π,*x*] ≠ {} ∧ *out*[*L1*,*map* (λ(*x*, *y*). (*x*, *the y*)) π,*x*] ⊆ *out*[*L2*,*map* (λ(*x*, *y*). (*x*, *the y*)) π,*x*]
proof −
fix π *x* **assume** π ∈ *?L1* ∩ *?L2* **and** *x* ∈ *X*
then have π ∈ *?L1* **and** π ∈ *?L2* **by** *auto*

show *out*[*?L2*,π,*x*] = {*None*} ∪ *Some* ‘ *Y* ∨ *is-not-undefined* π *L1* ∧ *is-not-undefined* π *L2* ∧ *out*[*L1*,*map* (λ(*x*, *y*). (*x*, *the y*)) π,*x*] ≠ {} ∧ *out*[*L1*,*map* (λ(*x*, *y*). (*x*, *the y*)) π,*x*] ⊆ *out*[*L2*,*map* (λ(*x*, *y*). (*x*, *the y*)) π,*x*]
proof (*cases out*[*?L2*,π,*x*] = {*None*} ∪ *Some* ‘ *Y*)
case *True*
then show *?thesis* **by** *blast*
next
case *False*

let *?*π = *map* (λ(*x*, *y*). (*x*, *the y*)) π

have *is-not-undefined* π *L2*
using *False* ‹(∀ π ∈*bottom-completion X Y L1* ∩ *bottom-completion X Y L2*. ∀ *x*∈*X*. ¬ *is-not-undefined* π *L1* ⟶ *is-not-undefined* π *L2* ⟶ *out*[*bottom-completion X Y L2*,π,*x*] = {*None*} ∪ *Some* ‘ *Y*) ∧ (∀ π∈*L1* ∩ *L2*. ∀ *x*∈*X*. *out*[*L2*,π,*x*] = {} ∨ *out*[*L1*,π,*x*] ≠ {} ∧ *out*[*L1*,π,*x*] ⊆ *out*[*L2*,π,*x*])› ‹π ∈ *bottom-completion X Y L1* ∩ *bottom-completion X Y L2*› ‹*x* ∈ *X*›
by (*meson* ‹π ∈ *bottom-completion X Y L2*› *assms*(*2*) *bottom-completion-out*(*3*))
then have *?*π ∈ *L2*
using *bottom-id*
by (*metis* (*mono-tags*, *lifting*) *imageE is-not-undefined.elims*(*2*))

have *is-not-undefined* π *L1*
using *False* ‹(∀ π ∈*bottom-completion X Y L1* ∩ *bottom-completion X Y L2*. ∀ *x*∈*X*. ¬ *is-not-undefined* π *L1* ⟶ *is-not-undefined* π *L2* ⟶ *out*[*bottom-completion X Y L2*,π,*x*] = {*None*} ∪ *Some* ‘ *Y*) ∧ (∀ π∈*L1* ∩ *L2*. ∀ *x*∈*X*. *out*[*L2*,π,*x*] = {} ∨ *out*[*L1*,π,*x*] ≠ {} ∧ *out*[*L1*,π,*x*] ⊆ *out*[*L2*,π,*x*])› ‹π ∈ *bottom-completion X Y L1* ∩ *bottom-completion X Y L2*› ‹*x* ∈ *X*›
using ‹*is-not-undefined* π *L2*› **by** *blast*
then have *?*π ∈ *L1*
using *bottom-id*
by (*metis* (*mono-tags*, *lifting*) *imageE is-not-undefined.elims*(*2*))

have *out*[*L2*,*?*π,*x*] ≠ {}
using *False bottom-completion-out*(*2*)[*OF assms*(*2*) ‹*x* ∈ *X*› ‹π ∈ *?L2*› ‹*is-not-undefined* π *L2*›]
by *blast*
then have *out*[*L1*,*?*π,*x*] ≠ {} **and** *out*[*L1*,*?*π,*x*] ⊆ *out*[*L2*,*?*π,*x*]
using ‹*?B*› ‹*?*π ∈ *L1*› ‹*?*π ∈ *L2*› ‹*x* ∈ *X*›
by (*meson IntI*)+
then show *?thesis*
using ‹*is-not-undefined* π *L1*› ‹*is-not-undefined* π *L2*›

by *blast*
qed
qed
then show *?A*
by *blast*
qed
qed
also have ... $=$ (($\forall$ π $\in$ *?L1* $\cap$ *?L2* . $\forall$ $x \in X$. $\neg$ *is-not-undefined* π *L1* $\longrightarrow$ *is-not-undefined* π *L2* $\longrightarrow$ *out*[*L2*,*map* (λ(*x*, *y*). (*x*, *the y*)) π,*x*] = {})
$\wedge$ ($\forall$ $\pi \in$ *L1* $\cap$ *L2* . $\forall$ $x \in X$. *out*[*L2*,π,*x*] = {} $\vee$ (*out*[*L1*,π,*x*] $\neq$ {} $\wedge$ *out*[*L1*,π,*x*] $\subseteq$ *out*[*L2*,π,*x*])))
(**is** (*?A* $\wedge$ *?B*) = (*?C* $\wedge$ *?B*))
proof −
have *?A* = *?C*
by (*metis IntD2 None-notin-image-Some UnCI assms*(*2*) *bottom-completion-out*(*1*) *bottom-completion-out*(*2*) *insertCI*)
then show *?thesis* **by** *meson*
qed
also have ... = ($\forall$ $\pi \in$ *L1* $\cap$ *L2* . $\forall$ $x \in X$. *out*[*L2*,π,*x*] = {} $\vee$ (*out*[*L1*,π,*x*] $\neq$ {} $\wedge$ *out*[*L1*,π,*x*] $\subseteq$ *out*[*L2*,π,*x*]))
(**is** (*?A* $\wedge$ *?B*) = *?B*)
proof −
have *?B* $\Longrightarrow$ *?A*
proof −
assume *?B*

have $\bigwedge$ π *x* . $\pi \in$ *?L1* $\cap$ *?L2* $\Longrightarrow$ $x \in X$ $\Longrightarrow$ $\neg$ *is-not-undefined* π *L1* $\Longrightarrow$ *is-not-undefined* π *L2* $\Longrightarrow$ *out*[*L2*,*map* (λ(*x*, *y*). (*x*, *the y*)) π,*x*] = {}
proof (*rule ccontr*)
fix π *x* **assume** $\pi \in$ *?L1* $\cap$ *?L2* **and** $x \in X$ **and** $\neg$ *is-not-undefined* π *L1* **and** *is-not-undefined* π *L2*
and *out*[*L2*,*map* (λ(*x*, *y*). (*x*, *the y*)) π,*x*] $\neq$ {}

let *?*π = *map* (λ(*x*, *y*). (*x*, *the y*)) π
have *?*$\pi \in$ *L2*
by (*metis* (*mono-tags*, *lifting*) ‹*is-not-undefined* π *L2*› *bottom-id image-iff is-not-undefined.elims*(*2*))

have $\pi \in$ *?L1*
using ‹$\pi \in$ *?L1* $\cap$ *?L2*› **by** *auto*
moreover have $\pi \notin$ *L1'a*
unfolding *L1'a-def* **using** ‹$\neg$ *is-not-undefined* π *L1*› **by** *auto*
ultimately have $\pi \in$ *L1'b*
unfolding ‹*?L1* = *L1'a* $\cup$ *L1'b*› **by** *blast*
then obtain π' x' y' τ **where** π = (*map* (λ(*x*,*y*) . (*x*,*Some y*)) π')@[(x',y')]@τ

and $\pi' \in$ *L1*
and *out*[*L1*,π',x'] = {}

and $x' \in X$
and $(y' = None \vee y' \in Some\ `\ Y)$
and $(\forall\ (x,y) \in set\ \tau\ .\ x \in X \wedge (y = None \vee y \in Some\ `\ Y))$
unfolding *L1'b-def*
by *blast*

have $?\pi = (\pi'@[(x',\ the\ y')])\ @\ (map\ (\lambda(x,\ y).\ (x,\ the\ y))\ \tau)$
unfolding ‹$\pi = (map\ (\lambda(x,y)\ .\ (x,Some\ y))\ \pi')@[(x',y')]@\tau$›
using *bottom-id* **by** (*induction* π' *arbitrary*: $x'\ y'\ \tau$; *auto*)
then have $\pi'@[(x',\ the\ y')] \in L2$ **and** $\pi' \in L2$
using ‹$?\pi \in L2$›
by (*metis assms*(*2*) *prefix-closure-no-member*)+
then have $out[L2,\pi',x'] \neq \{\}$
by *fastforce*

show *False*
using ‹$?B$› ‹$\pi' \in L1$› ‹$\pi' \in L2$› ‹$x' \in X$› ‹$out[L2,\pi',x'] \neq \{\}$› ‹$out[L1,\pi',x'] = \{\}$›
by *blast*
qed
then show *?A*
by *blast*
qed
then show *?thesis* **by** *meson*
qed
also have $\ldots = (L1 \preceq[X,quasired\ Y]\ L2)$
unfolding *quasired-type-1*[*OF assms*, *symmetric*] *quasi-reduction-def*
by (*meson assms*(*2*) *executable-inputs-in-alphabet outputs-executable*)
finally show *?thesis*
by *meson*
qed

6.3 Strong Reduction via Reduction and Undefinedness Outputs

fun *non-bottom-shortening* :: $('x,'y\ option)\ word \Rightarrow ('x,'y\ option)\ word$ **where**
non-bottom-shortening $\pi = filter\ (\lambda\ (x,y)\ .\ y \neq None)\ \pi$

fun *non-bottom-projection* :: $('x,'y\ option)\ word \Rightarrow ('x,'y)\ word$ **where**
non-bottom-projection $\pi = map\ (\lambda(x,y)\ .\ (x,the\ y))\ (non\text{-}bottom\text{-}shortening\ \pi)$

lemma *non-bottom-projection-split*: *non-bottom-projection* $(\pi'@\pi'') = (non\text{-}bottom\text{-}projection\ \pi')@(non\text{-}bottom\text{-}projection\ \pi'')$
by (*induction* π' *arbitrary*: π''; *auto*)

lemma *non-bottom-projection-id* : *non-bottom-projection* $(map\ (\lambda(x,y)\ .\ (x,Some\ y))\ \pi) = \pi$
by (*induction* π; *auto*)

fun *undefinedness-completion* :: $'x$ *alphabet* $\Rightarrow$ $('x,'y)$ *language* $\Rightarrow$ $('x,\ 'y\ option)$ *language* **where**
undefinedness-completion $X\ L$ =
$\{\pi\ .\ non\text{-}bottom\text{-}projection\ \pi \in L \wedge (\forall\ \pi'\ x\ \pi''\ .\ \pi = \pi'\ @\ [(x,None)]\ @\ \pi'' \longrightarrow x \in X \wedge out[L,\ non\text{-}bottom\text{-}projection\ \pi',\ x] = \{\})\}$

lemma *undefinedness-completion-is-language* :
assumes *is-language* $X\ Y\ L$
shows *is-language* X $(\{None\} \cup Some\ `\ Y)$ $(undefinedness\text{-}completion\ X\ L)$
proof –
let $?L$ = *undefinedness-completion* $X\ L$

have $[] \in L$
using *language-contains-nil*[*OF assms*] **.**
moreover have *non-bottom-projection* $[] = []$
by *auto*
ultimately have $[] \in ?L$
by *simp*
then have $?L \neq \{\}$
by *blast*
moreover have $\bigwedge \pi\ .\ \pi \in ?L \Longrightarrow (\bigwedge xy\ .\ xy \in set\ \pi \Longrightarrow fst\ xy \in X \wedge snd\ xy \in (\{None\} \cup Some\ `\ Y))$
and $\bigwedge \pi\ .\ \pi \in ?L \Longrightarrow (\bigwedge \pi'\ .\ prefix\ \pi'\ \pi \Longrightarrow \pi' \in ?L)$
proof –
fix π **assume** $\pi \in ?L$
then have *p1*: *non-bottom-projection* $\pi \in L$
and *p2*: $\bigwedge \pi'\ x\ \pi''\ .\ \pi = \pi'\ @\ [(x,None)]\ @\ \pi'' \Longrightarrow x \in X \wedge out[L,\ non\text{-}bottom\text{-}projection\ \pi',\ x] = \{\}$
by *auto*

show $\bigwedge xy\ .\ xy \in set\ \pi \Longrightarrow fst\ xy \in X \wedge snd\ xy \in (\{None\} \cup Some\ `\ Y)$
proof –
fix xy **assume** $xy \in set\ \pi$
then obtain $\pi'\ x\ y\ \pi''$ **where** $xy = (x,y)$ **and** $\pi = \pi'\ @\ [(x,y)]\ @\ \pi''$
by (*metis append-Cons append-Nil old.prod.exhaust split-list*)

show $fst\ xy \in X \wedge snd\ xy \in (\{None\} \cup Some\ `\ Y)$
proof (*cases snd xy*)
case *None*
then show *?thesis*
unfolding ‹$xy = (x,y)$› *snd-conv*
using *p2* ‹$\pi = \pi'\ @\ [(x,y)]\ @\ \pi''$›
by *simp*
next
case (*Some* y')
then have $y = Some\ y'$

unfolding ‹$xy = (x,y)$› **by** *auto*
have $(x,y') \in$ *set* (*non-bottom-projection* π)
unfolding ‹$\pi = \pi'$ @ $[(x,y)]$ @ π''› ‹y = *Some* y'›
by *auto*
then show *?thesis*
unfolding ‹$xy = (x,y)$› *snd-conv* ‹y = *Some* y'› *fst-conv*
using *p1 assms*
unfolding *is-language.simps* **by** *fastforce*
qed
qed

show $\bigwedge \pi'$. *prefix* π' $\pi \Longrightarrow \pi' \in$ *?L*
proof –
fix π' **assume** *prefix* π' π
then obtain π'' **where** $\pi = \pi'$@π''
using *prefixE* **by** *blast*

have *non-bottom-projection* π = (*non-bottom-projection* π')@(*non-bottom-projection* π'')
unfolding ‹$\pi = \pi'$@π''›
using *non-bottom-projection-split* **.**
then have *non-bottom-projection* $\pi' \in L$
by (*metis assms p1 prefix-closure-no-member*)
moreover have $\bigwedge \pi'''$ x π'''' . $\pi' = \pi'''$ @ $[(x,None)]$ @ $\pi'''' \Longrightarrow x \in X \wedge$ *out*[L, *non-bottom-projection* π''', x] = {}
using *p2* **unfolding** ‹$\pi = \pi'$@π''›
by (*metis append.assoc*)
ultimately show $\pi' \in$ *?L*
by *fastforce*
qed
qed
ultimately show *?thesis*
by (*meson is-language.elims*(*3*))
qed

lemma *undefinedness-completion-inclusion* :
assumes $\pi \in L$
shows *map* ($\lambda(x,y)$. (x,*Some* y)) $\pi \in$ *undefinedness-completion X L*
proof –
let *?π* = *map* ($\lambda(x,y)$. (x,*Some* y)) π

have $\bigwedge$ a . (a,*None*) $\notin$ *set ?π*
by (*induction* π; *auto*)
then have $\forall$ π' x π'' . *?π* = π' @ $[(x,None)]$ @ $\pi'' \longrightarrow x \in X \wedge$ *out*[L, *non-bottom-projection* π', x] = {}
by (*metis Cons-eq-appendI in-set-conv-decomp*)
moreover have *non-bottom-projection ?π* $\in L$
using ‹$\pi \in L$› **unfolding** *non-bottom-projection-id* **.**

ultimately show *?thesis*
by *auto*
qed

lemma *undefinedness-completion-out-shortening* :
assumes *is-language X Y L*
and $\pi \in$ *undefinedness-completion X L*
and $x \in X$
shows *out*[*undefinedness-completion X L*, π, *x*] = *out*[*undefinedness-completion X L*, *non-bottom-shortening* π, *x*]
using *assms*(*2*,*3*) **proof** (*induction length* π *arbitrary*: π *x rule*: *less-induct*)
case *less*

let *?L* = *undefinedness-completion X L*

show *?case* **proof** (*cases* π *rule*: *rev-cases*)
case *Nil*
then show *?thesis* **by** *auto*
next
case (*snoc* π′ *xy*)

then obtain *x*′ *y*′ **where** *xy* = (*x*′,*y*′) **by** *fastforce*

have *x*′ ∈ *X*
using *snoc less.prems*(*1*) **unfolding** ‹*xy* = (*x*′,*y*′)›
using *undefinedness-completion-is-language*[*OF assms*(*1*)]
by (*metis fst-conv is-language.elims*(*2*) *last-in-set snoc-eq-iff-butlast*)

have π′ ∈ *?L*
using *snoc less.prems*(*1*)
using *undefinedness-completion-is-language*[*OF assms*(*1*)]
using *prefix-closure-no-member* **by** *blast*

show *?thesis* **proof** (*cases y*′)
case *None*

then have *non-bottom-shortening* π = *non-bottom-shortening* π′
unfolding ‹*xy* = (*x*′,*y*′)› *snoc* **by** *auto*
then have *out*[*?L*, *non-bottom-shortening* π, *x*] = *out*[*?L*, *non-bottom-shortening* π′, *x*]
by *simp*
also have . . . = *out*[*?L*, π′, *x*]
using *less.hyps*[*OF* - ‹π′ ∈ *?L*› ‹*x* ∈ *X*›] **unfolding** *snoc*
by (*metis Suc-lessD length-append-singleton not-less-eq*)
also have . . . = *out*[*?L*, π, *x*]
proof

show *out*[*?L*, π′, *x*] ⊆ *out*[*?L*, π, *x*]

proof
  fix *y* **assume** $y \in out[?L, \pi', x]$
  then have $\pi'@[(x,y)] \in ?L$
    by *auto*
  then have *p1*: *non-bottom-projection* $(\pi'@[(x,y)]) \in L$
      and *p2*: $\bigwedge \gamma'\ a\ \gamma''\ .\ \pi'@[(x,y)] = \gamma'\ @\ [(a,None)]\ @\ \gamma'' \Longrightarrow a \in X \wedge$ $out[L,$ *non-bottom-projection* $\gamma', a] = \{\}$
    by *auto*

have *non-bottom-projection* $(\pi@[(x,y)])$ = *non-bottom-projection* $(\pi'@[(x,y)])$
    unfolding *snoc* ‹$xy = (x',y')$› *None* **by** *auto*
  then have *non-bottom-projection* $(\pi@[(x,y)]) \in L$
    using *p1* **by** *simp*
  moreover have $\bigwedge \gamma'\ a\ \gamma''\ .\ \pi@[(x,y)] = \gamma'\ @\ [(a,None)]\ @\ \gamma'' \Longrightarrow a \in$ $X \wedge out[L,$ *non-bottom-projection* $\gamma', a] = \{\}$
  proof –
    fix $\gamma'\ a\ \gamma''$ **assume** $\pi@[(x,y)] = \gamma'\ @\ [(a,None)]\ @\ \gamma''$
    then have $\pi'@[(x',None)]@[(x,y)] = \gamma'\ @\ [(a,None)]\ @\ \gamma''$
      unfolding *snoc* ‹$xy = (x',y')$› *None* **by** *auto*

    show $a \in X \wedge out[L,$ *non-bottom-projection* $\gamma', a] = \{\}$
    proof (*cases* γ'' *rule*: *rev-cases*)
      case *Nil*
      then show *?thesis*
        using ‹$\pi\ @\ [(x,\ y)] = \gamma'\ @\ [(a,\ None)]\ @\ \gamma''$› ‹*non-bottom-shortening* π = *non-bottom-shortening* π'› *p2* **by** *auto*
    next
      case (*snoc* γ''' xy')
      then show *?thesis*
        using ‹$\pi\ @\ [(x,\ y)] = \gamma'\ @\ [(a,\ None)]\ @\ \gamma''$› *less.prems*(*1*) **by** *force*
    qed
  qed
  ultimately show $y \in out[?L, \pi, x]$
    by *auto*
qed

show $out[?L, \pi, x] \subseteq out[?L, \pi', x]$
proof
  fix *y* **assume** $y \in out[?L, \pi, x]$
  then have $\pi'@[(x',None)]@[(x,y)] \in ?L$
    unfolding *snoc* ‹$xy = (x',y')$› *None*
    by *auto*
  then have *p1*: *non-bottom-projection* $(\pi'@[(x',None)]@[(x,y)]) \in L$
      and *p2*: $\bigwedge \gamma'\ a\ \gamma''\ .\ \pi'@[(x',None)]@[(x,y)] = \gamma'\ @\ [(a,None)]\ @\ \gamma''$ $\Longrightarrow a \in X \wedge out[L,$ *non-bottom-projection* $\gamma', a] = \{\}$
    by *auto*

have *non-bottom-projection* $(\pi'@[(x',None)]@[(x,y)])$ = *non-bottom-projection* $(\pi'@[(x,y)])$

```
            by auto
          then have non-bottom-projection (π'@[(x,y)]) ∈ L
            using p1 by auto
          moreover have ⋀ γ' a γ'' . π'@[(x,y)] = γ' @ [(a,None)] @ γ'' ⟹ a ∈
X ∧ out[L, non-bottom-projection γ', a] = {}
          proof –
            fix γ' a γ'' assume π'@[(x,y)] = γ' @ [(a,None)] @ γ''

            show a ∈ X ∧ out[L, non-bottom-projection γ', a] = {}
            proof (cases γ'' rule: rev-cases)
              case Nil
              then show ?thesis
                   by (metis None ‹π' @ [(x, y)] = γ' @ [(a, None)] @ γ''›
‹non-bottom-shortening π = non-bottom-shortening π'› ‹xy = (x', y')› append.assoc
append.right-neutral append1-eq-conv non-bottom-projection.simps p2 snoc)
            next
              case (snoc γ''' xy')
              then show ?thesis
                   using ‹π' @ [(x, y)] = γ' @ [(a, None)] @ γ''› ‹π' ∈ undefined-
ness-completion X L› by force
            qed
          qed
          ultimately show y ∈ out[?L, π', x]
            by auto
        qed
      qed
      finally show ?thesis
        by blast
    next
      case (Some y'')

      have non-bottom-shortening π = (non-bottom-shortening π')@[(x',Some y'')]
        unfolding snoc ‹xy = (x',y')› Some by auto
      then have non-bottom-projection π = (non-bottom-projection π')@[(x',y'')]
        by auto

      have π' @ [(x', Some y'')] ∈ ?L
        using less.prems(1) unfolding snoc ‹xy = (x',y')› Some .
      then have Some y'' ∈ out[?L,π',x']
        by auto
      moreover have out[?L,π',x'] = out[?L,non-bottom-shortening π',x']
        using less.hyps[OF - ‹π' ∈ ?L› ‹x' ∈ X›]
        unfolding snoc ‹xy = (x',y')› Some
        by (metis length-append-singleton lessI)
      ultimately have Some y'' ∈ out[?L,non-bottom-shortening π',x']
        by blast


      show ?thesis
```

proof
show *out*[*?L*,π,*x*] $\subseteq$ *out*[*?L*,*non-bottom-shortening* π,*x*]
proof
fix *y* **assume** $y \in$ *out*[*?L*,π,*x*]
then have π'@[(x',*Some* y'')]@[(*x*,*y*)] $\in$ *?L*
unfolding *snoc* ‹*xy* = (x',y')› *Some* **by** *auto*
then have *p1*: *non-bottom-projection* (π'@[(x',*Some* y'')]@[(*x*,*y*)]) $\in$ *L*
and *p2*: $\bigwedge$ γ' *a* γ'' . π'@[(x',*Some* y'')]@[(*x*,*y*)] = γ' @ [(*a*,*None*)] @ γ'' $\Longrightarrow$ *a* $\in$ *X* $\wedge$ *out*[*L*, *non-bottom-projection* γ', *a*] = {}
by *auto*

have *non-bottom-projection* ((*non-bottom-shortening* π)@[(*x*,*y*)]) = *non-bottom-projection* (π'@[(x',*Some* y'')]@[(*x*,*y*)])
unfolding ‹*non-bottom-shortening* π = (*non-bottom-shortening* π')@[(x',*Some* y'')]›
by *auto*
then have *non-bottom-projection* ((*non-bottom-shortening* π)@[(*x*,*y*)]) $\in$ *L*
using *p1* **by** *simp*
moreover have $\bigwedge$ γ' *a* γ'' . (*non-bottom-shortening* π)@[(*x*,*y*)] = γ' @ [(*a*,*None*)] @ γ'' $\Longrightarrow$ *a* $\in$ *X* $\wedge$ *out*[*L*, *non-bottom-projection* γ', *a*] = {}
proof −
fix γ' *a* γ'' **assume** (*non-bottom-shortening* π)@[(*x*,*y*)] = γ' @ [(*a*,*None*)] @ γ''
moreover have (*a*, *None*) $\notin$ *set* (*non-bottom-shortening* π)
by (*induction* π; *auto*)
moreover have $\bigwedge$ *xs a ys b zs* . *xs*@[*a*] = *ys*@[*b*]@*zs* $\Longrightarrow$ *b* $\notin$ *set xs* $\Longrightarrow$ *zs* = []
by (*metis append-Cons append-Nil butlast.simps*(*2*) *butlast-snoc in-set-butlast-appendI list.distinct*(*1*) *list.sel*(*1*) *list.set-sel*(*1*))
ultimately have γ'' = []
by *fastforce*
then have γ' = *non-bottom-shortening* π
and *x* = *a*
and *y* = *None*
using ‹(*non-bottom-shortening* π)@[(*x*,*y*)] = γ' @ [(*a*,*None*)] @ γ''›
by *auto*

show *a* $\in$ *X* $\wedge$ *out*[*L*, *non-bottom-projection* γ', *a*] = {}
using ‹*x* $\in$ *X*› **unfolding** ‹*x* = *a*›
unfolding ‹γ' = *non-bottom-shortening* π›
by (*metis* (*no-types*, *lifting*) ‹*non-bottom-projection* (*non-bottom-shortening* π @ [(*x*, *y*)]) = *non-bottom-projection* (π' @ [(x', *Some* y'')] @ [(*x*, *y*)])› ‹*x* = *a*› ‹*y* = *None*› *append.assoc append.right-neutral append-same-eq non-bottom-projection-split p2*)
qed
ultimately show *y* $\in$ *out*[*?L*,*non-bottom-shortening* π,*x*]
by *auto*
qed

show *out*[*?L*,*non-bottom-shortening* π,*x*] ⊆ *out*[*?L*,π,*x*]
proof
fix *y* **assume** *y* ∈ *out*[*?L*,*non-bottom-shortening* π,*x*]
then have (*non-bottom-shortening* π′)@[(*x*′,*Some y*′′)]@[(*x*,*y*)] ∈ *?L*
unfolding *snoc* ‹*xy* = (*x*′,*y*′)› *Some* **by** *auto*
then have *p1*: *non-bottom-projection* ((*non-bottom-shortening* π′)@[(*x*′,*Some* *y*′′)]@[(*x*,*y*)]) ∈ *L*
and *p2*: ⋀ γ′ *a* γ′′ . (*non-bottom-shortening* π′)@[(*x*′,*Some y*′′)]@[(*x*,*y*)] = γ′ @ [(*a*,*None*)] @ γ′′ ⟹ *a* ∈ *X* ∧ *out*[*L*, *non-bottom-projection* γ′, *a*] = {}
by *auto*

have *non-bottom-projection* ((*non-bottom-shortening* π′)@[(*x*′,*Some* *y*′′)]@[(*x*,*y*)]) = *non-bottom-projection* (π@[(*x*,*y*)])
unfolding *snoc* ‹*xy* = (*x*′,*y*′)› *Some* **by** *auto*
then have *non-bottom-projection* (π@[(*x*,*y*)]) ∈ *L*
using *p1* **by** *presburger*
moreover have ⋀ γ′ *a* γ′′ . π@[(*x*,*y*)] = γ′ @ [(*a*,*None*)] @ γ′′ ⟹ *a* ∈ *X* ∧ *out*[*L*, *non-bottom-projection* γ′, *a*] = {}
proof
fix γ′ *a* γ′′ **assume** π@[(*x*,*y*)] = γ′ @ [(*a*,*None*)] @ γ′′
then have (*a*,*None*) ∈ *set* (π@[(*x*,*y*)])
by *auto*
then consider (*a*,*None*) ∈ *set* π | (*a*,*None*) = (*x*,*y*)
by *auto*
then show *a* ∈ *X*
by (*metis assms*(*1*) *fst-conv is-language.elims*(*2*) *less.prems*(*1*) *less.prems*(*2*) *undefinedness-completion-is-language*)

show *out*[*L*,*non-bottom-projection* γ′,*a*] = {}
proof (*cases* γ′′ *rule*: *rev-cases*)
case *Nil*
then have π = γ′ **and** *x* = *a* **and** *y* = *None*
using ‹π@[(*x*,*y*)] = γ′ @ [(*a*,*None*)] @ γ′′› **by** *auto*
then show *?thesis*
by (*metis* (*no-types*, *opaque-lifting*) ‹*non-bottom-projection* (*non-bottom-shortening* π′ @ [(*x*′, *Some y*′′)] @ [(*x*, *y*)]) = *non-bottom-projection* (π @ [(*x*, *y*)])› ‹*non-bottom-shortening* π = *non-bottom-shortening* π′ @ [(*x*′, *Some* *y*′′)]› *append.assoc append-Cons append-Nil append-same-eq non-bottom-projection-split* *p2*)
next
case (*snoc* γ′′′ *xy*′)
then have π = γ′ @ [(*a*, *None*)] @ γ′′′
using ‹π@[(*x*,*y*)] = γ′ @ [(*a*,*None*)] @ γ′′› **by** *auto*

have γ′ @ [(*a*, *None*)] ∈ *?L*
using *less.prems*(*1*) **unfolding** ‹π = γ′ @ [(*a*, *None*)] @ γ′′′›
using *undefinedness-completion-is-language*[*OF assms*(*1*)]
by (*metis append-assoc prefix-closure-no-member*)

then show *out*[*L*, *non-bottom-projection* γ', *a*] = {}
by *auto*
qed
qed
ultimately show $y \in$ *out*[*?L*,π,*x*]
by *auto*
qed
qed
qed
qed
qed

lemma *undefinedness-completion-out-projection-not-empty* :
assumes *is-language X Y L*
and $\pi \in$ *undefinedness-completion X L*
and $x \in X$
and *out*[*L*, *non-bottom-projection* π, *x*] $\neq$ {}
shows *out*[*undefinedness-completion X L*, *non-bottom-shortening* π, *x*] = *Some* ‘ *out*[*L*, *non-bottom-projection* π, *x*]
proof

let *?L* = *undefinedness-completion X L*

have π@[(*x*,*None*)] $\notin$ *?L*
using *assms*(*4*) **by** *auto*
then have *None* $\notin$ *out*[*?L*,π,*x*]
by *auto*
then have *None* $\notin$ *out*[*?L*,*non-bottom-shortening* π,*x*]
using *undefinedness-completion-out-shortening*[*OF assms*(*1*,*2*,*3*)] **by** *blast*
then have (*non-bottom-shortening* π)@[(*x*,*None*)] $\notin$ *?L*
by *auto*

show *out*[*?L*, *non-bottom-shortening* π, *x*] $\subseteq$ *Some* ‘ *out*[*L*, *non-bottom-projection* π, *x*]
proof
fix *y* **assume** $y \in$ *out*[*?L*, *non-bottom-shortening* π, *x*]
then have (*non-bottom-shortening* π) @ [(*x*,*y*)] $\in$ *?L* **by** *auto*
then have $y \neq$ *None*
using ‹(*non-bottom-shortening* π)@[(*x*,*None*)] $\notin$ *?L*›
by *meson*
then obtain y' **where** y = *Some* y'
by *auto*

have *non-bottom-projection* ((*non-bottom-shortening* π) @ [(*x*,*y*)]) = (*non-bottom-projection* π) @ [(*x*,y')]
unfolding ‹y = *Some* y'›
by (*induction* π; *auto*)

then have (*non-bottom-projection* π) @ [(*x*,*y'*)] ∈ *L*
using ‹(*non-bottom-shortening* π) @ [(*x*,*y*)] ∈ *?L*› **unfolding** ‹*y* = *Some y'*›
by *auto*
then show *y* ∈ *Some* ‘ *out*[*L*, *non-bottom-projection* π, *x*]
unfolding ‹*y* = *Some y'*› **by** *auto*
qed

show *Some* ‘ *out*[*L*,*non-bottom-projection* π,*x*] ⊆ *out*[*?L*,*non-bottom-shortening* π,*x*]
proof
fix *y* **assume** *y* ∈ *Some* ‘ *out*[*L*,*non-bottom-projection* π,*x*]
then obtain *y'* **where** *y* = *Some y'* **and** *y'* ∈ *out*[*L*,*non-bottom-projection* π,*x*]
by *auto*
then have (*non-bottom-projection* π) @ [(*x*,*y'*)] ∈ *L*
by *auto*
moreover have *non-bottom-projection* ((*non-bottom-shortening* π) @ [(*x*,*y*)]) = (*non-bottom-projection* π) @ [(*x*,*y'*)]
unfolding ‹*y* = *Some y'*›
by (*induction* π; *auto*)
ultimately have *non-bottom-projection* ((*non-bottom-shortening* π) @ [(*x*,*y*)]) ∈ *L*
unfolding ‹*y* = *Some y'*›
by *auto*
moreover have ⋀ π' *x'* π'' . ((*non-bottom-shortening* π) @ [(*x*,*y*)]) = π' @ [(*x'*,*None*)] @ π'' ⟹ *x'* ∈ *X* ∧ *out*[*L*, *non-bottom-projection* π', *x'*] = {}
proof −
fix π' *x'* π'' **assume** ((*non-bottom-shortening* π) @ [(*x*,*y*)]) = π' @ [(*x'*,*None*)] @ π''
then have (*x'*,*None*) ∈ *set* (*non-bottom-shortening* π)
by (*metis* ‹*y* = *Some y'*› *append-Cons in-set-conv-decomp old.prod.inject option.distinct*(*1*) *rotate1.simps*(*2*) *set-ConsD set-rotate1*)
then have *False*
by (*induction* π; *auto*)
then show *x'* ∈ *X* ∧ *out*[*L*, *non-bottom-projection* π', *x'*] = {}
by *blast*
qed
ultimately show *y* ∈ *out*[*?L*,*non-bottom-shortening* π,*x*]
by *auto*
qed
qed

lemma *undefinedness-completion-out-projection-empty* :
assumes *is-language X Y L*
and π ∈ *undefinedness-completion X L*
and *x* ∈ *X*
and *out*[*L*, *non-bottom-projection* π, *x*] = {}
shows *out*[*undefinedness-completion X L*, *non-bottom-shortening* π, *x*] = {*None*}
proof

let *?L* = *undefinedness-completion X L*

have *p1*: *non-bottom-projection* $\pi \in L$
and *p2*: $\bigwedge \pi'\ x\ \pi''\ .\ \pi = \pi' @ [(x,None)] @ \pi'' \Longrightarrow x \in X \wedge out[L, non\text{-}bottom\text{-}projection\ \pi', x] = \{\}$
using *assms*(*2*) by *auto*

have *non-bottom-projection* $(\pi@[(x,None)]) \in L$
using *p1* by *auto*
moreover have $\bigwedge \pi'\ x'\ \pi''\ .\ \pi@[(x,None)] = \pi' @ [(x',None)] @ \pi'' \Longrightarrow x' \in X \wedge out[L, non\text{-}bottom\text{-}projection\ \pi', x'] = \{\}$
proof −
fix $\pi'\ x'\ \pi''$ assume $\pi@[(x,None)] = \pi' @ [(x',None)] @ \pi''$
show $x' \in X \wedge out[L, non\text{-}bottom\text{-}projection\ \pi', x'] = \{\}$
proof (*cases* π'' *rule*: *rev-cases*)
case *Nil*
then show *?thesis*
using ‹$\pi @ [(x, None)] = \pi' @ [(x', None)] @ \pi''$› *assms*(*3*) *assms*(*4*) by *auto*
next
case (*snoc ys y*)
then show *?thesis*
using ‹$\pi @ [(x, None)] = \pi' @ [(x', None)] @ \pi''$› *p2* by *auto*
qed
qed
ultimately have $\pi@[(x,None)] \in ?L$
by *auto*
then show $\{None\} \subseteq out[?L,non\text{-}bottom\text{-}shortening\ \pi,x]$
unfolding *undefinedness-completion-out-shortening*[*OF assms*(*1*,*2*,*3*), *symmetric*]
by *auto*

show $out[?L,non\text{-}bottom\text{-}shortening\ \pi,x] \subseteq \{None\}$
proof (*rule ccontr*)
assume $\neg\ out[?L,non\text{-}bottom\text{-}shortening\ \pi,x] \subseteq \{None\}$
then obtain *y* where $y \in out[?L,non\text{-}bottom\text{-}shortening\ \pi,x]$ and $y \neq None$
by *blast*
then obtain y' where $y = Some\ y'$
by *auto*

have $\pi@[(x,Some\ y')] \in ?L$
using ‹$y \in out[?L,non\text{-}bottom\text{-}shortening\ \pi,x]$›
unfolding ‹$y = Some\ y'$›
unfolding *undefinedness-completion-out-shortening*[*OF assms*(*1*,*2*,*3*), *symmetric*]
by *auto*
then have $(non\text{-}bottom\text{-}projection\ \pi)@[(x,y')] \in L$
by *auto*

then show *False*
using *assms(4)* **by** *auto*
qed
qed

theorem *strongred-via-red* :
assumes *is-language X Y L1*
and *is-language X Y L2*
shows $(L1 \preceq[X,strongred\ Y]\ L2) \longleftrightarrow ((undefinedness\text{-}completion\ X\ L1) \preceq[X,\ red\ (\{None\} \cup Some\ `\ Y)]\ (undefinedness\text{-}completion\ X\ L2))$
proof –

let $?L1 = undefinedness\text{-}completion\ X\ L1$
let $?L2 = undefinedness\text{-}completion\ X\ L2$

have $(L1 \preceq[X,strongred\ Y]\ L2) = (\forall\ \pi \in L1 \cap L2\ .\ \forall\ x \in X\ .\ (out[L1,\pi,x] = \{\} \wedge out[L2,\pi,x] = \{\}) \vee (out[L1,\pi,x] \neq \{\} \wedge out[L1,\pi,x] \subseteq out[L2,\pi,x]))$
(**is** $?A = ?B$)
proof
show $?A \Longrightarrow ?B$
unfolding *strongred-type-1[OF assms, symmetric] strong-reduction-def quasi-reduction-def*
by *(metis outputs-executable)*
show $?B \Longrightarrow ?A$
unfolding *strongred-type-1[OF assms, symmetric] strong-reduction-def quasi-reduction-def*
by *(metis assms(1) assms(2) executable-inputs-in-alphabet outputs-executable subset-empty)*
qed
also have $\ldots = (\forall\ \pi \in ?L1 \cap ?L2\ .\ \forall\ x \in X\ .\ (out[L1,non\text{-}bottom\text{-}projection\ \pi,x] = \{\} \wedge out[L2,non\text{-}bottom\text{-}projection\ \pi,x] = \{\}) \vee (out[L1,non\text{-}bottom\text{-}projection\ \pi,x] \neq \{\} \wedge out[L1,non\text{-}bottom\text{-}projection\ \pi,x] \subseteq out[L2,non\text{-}bottom\text{-}projection\ \pi,x]))$
(**is** $?A = ?B$)
proof
have $\bigwedge \pi\ x\ .\ ?A \Longrightarrow \pi \in ?L1 \cap ?L2 \Longrightarrow x \in X \Longrightarrow (out[L1,non\text{-}bottom\text{-}projection\ \pi,x] = \{\} \wedge out[L2,non\text{-}bottom\text{-}projection\ \pi,x] = \{\}) \vee (out[L1,non\text{-}bottom\text{-}projection\ \pi,x] \neq \{\} \wedge out[L1,non\text{-}bottom\text{-}projection\ \pi,x] \subseteq out[L2,non\text{-}bottom\text{-}projection\ \pi,x])$
proof –
fix π x **assume** $?A$ **and** $\pi \in ?L1 \cap ?L2$ **and** $x \in X$

let $?\pi = non\text{-}bottom\text{-}projection\ \pi$

have $?\pi \in L1$
and $?\pi \in L2$
using $\langle \pi \in ?L1 \cap ?L2 \rangle$ **by** *auto*
then show $(out[L1,?\pi,x] = \{\} \wedge out[L2,?\pi,x] = \{\}) \vee (out[L1,?\pi,x] \neq \{\} \wedge out[L1,?\pi,x] \subseteq out[L2,?\pi,x])$
using $\langle ?A \rangle$ $\langle x \in X \rangle$ **by** *blast*

qed
then show *?A* ⟹ *?B*
by *blast*

have ⋀ π *x* . *?B* ⟹ π ∈ *L1* ∩ *L2* ⟹ *x* ∈ *X* ⟹ (*out*[*L1*,π,*x*] = {} ∧ *out*[*L2*,π,*x*] = {}) ∨ (*out*[*L1*,π,*x*] ≠ {} ∧ *out*[*L1*,π,*x*] ⊆ *out*[*L2*,π,*x*])
proof −
fix π *x* **assume** *?B* **and** π ∈ *L1* ∩ *L2* **and** *x* ∈ *X*

let *?π* = *map* (λ(*x*,*y*) . (*x*,*Some y*)) π

have *?π* ∈ *?L1* **and** *?π* ∈ *?L2*
using ‹π ∈ *L1* ∩ *L2*› *undefinedness-completion-inclusion* **by** *blast+*
then have (*out*[*L1*,*non-bottom-projection ?π*,*x*] = {} ∧ *out*[*L2*,*non-bottom-projection ?π*,*x*] = {}) ∨ (*out*[*L1*,*non-bottom-projection ?π*,*x*] ≠ {} ∧ *out*[*L1*,*non-bottom-projection ?π*,*x*] ⊆ *out*[*L2*,*non-bottom-projection ?π*,*x*])
using ‹*?B*› ‹*x* ∈ *X*› **by** *blast*
then show (*out*[*L1*,π,*x*] = {} ∧ *out*[*L2*,π,*x*] = {}) ∨ (*out*[*L1*,π,*x*] ≠ {} ∧ *out*[*L1*,π,*x*] ⊆ *out*[*L2*,π,*x*])
unfolding *non-bottom-projection-id* **.**
qed
then show *?B* ⟹ *?A*
by *blast*
qed
also have . . . = (∀ π ∈ *?L1* ∩ *?L2* . ∀ *x* ∈ *X* . (*out*[*?L1*,π,*x*] = {*None*} ∧ *out*[*?L2*,π,*x*] = {*None*}) ∨ (*out*[*?L1*,π,*x*] ≠ {*None*} ∧ *out*[*?L1*,π,*x*] ⊆ *out*[*?L2*,π,*x*]))
proof −
have ⋀ π *x* . π ∈ *?L1* ∩ *?L2* ⟹ *x* ∈ *X* ⟹ (*out*[*L1*,*non-bottom-projection* π,*x*] = {} ∧ *out*[*L2*,*non-bottom-projection* π,*x*] = {}) = (*out*[*?L1*,π,*x*] = {*None*} ∧ *out*[*?L2*,π,*x*] = {*None*})
by (*metis IntD1 IntD2 None-notin-image-Some assms*(*1*) *assms*(*2*) *insertCI undefinedness-completion-out-projection-empty undefinedness-completion-out-projection-not-empty undefinedness-completion-out-shortening*)
moreover have ⋀ π *x* . π ∈ *?L1* ∩ *?L2* ⟹ *x* ∈ *X* ⟹ (*out*[*L1*,*non-bottom-projection* π,*x*] ≠ {} ∧ *out*[*L1*,*non-bottom-projection* π,*x*] ⊆ *out*[*L2*,*non-bottom-projection* π,*x*]) = (*out*[*?L1*,π,*x*] ≠ {*None*} ∧ *out*[*?L1*,π,*x*] ⊆ *out*[*?L2*,π,*x*])
proof −
fix π *x* **assume** π ∈ *?L1* ∩ *?L2* **and** *x* ∈ *X*
then have π ∈ *?L1* **and** π ∈ *?L2* **by** *auto*

have (*out*[*L1*,*non-bottom-projection* π,*x*] ≠ {}) = (*out*[*?L1*,π,*x*] ≠ {*None*})
by (*metis None-notin-image-Some* ‹π ∈ *undefinedness-completion X L1*› ‹*x* ∈ *X*› *assms*(*1*) *singletonI undefinedness-completion-out-projection-empty undefinedness-completion-out-projection-not-empty undefinedness-completion-out-shortening*)

show (*out*[*L1*,*non-bottom-projection* π,*x*] ≠ {} ∧ *out*[*L1*,*non-bottom-projection* π,*x*] ⊆ *out*[*L2*,*non-bottom-projection* π,*x*]) = (*out*[*?L1*,π,*x*] ≠ {*None*} ∧ *out*[*?L1*,π,*x*] ⊆ *out*[*?L2*,π,*x*])

proof (*cases out*[*L1*,*non-bottom-projection* π,*x*] ≠ {})
case *False*
then show *?thesis* **using** ‹(*out*[*L1*,*non-bottom-projection* π,*x*] ≠ {}) = (*out*[*?L1*,π,*x*] ≠ {*None*})› **by** *blast*
next
case *True*
have *out*[*undefinedness-completion X L1*,π,*x*] = *Some* ‘ *out*[*L1*,*non-bottom-projection* π,*x*]
using *undefinedness-completion-out-projection-not-empty*[*OF assms*(*1*) ‹π ∈ *?L1*› ‹*x* ∈ *X*› *True*]
unfolding *undefinedness-completion-out-shortening*[*OF assms*(*1*) ‹π ∈ *?L1*› ‹*x* ∈ *X*›,*symmetric*] **.**

show *?thesis* **proof** (*cases out*[*L2*,*non-bottom-projection* π,*x*] = {})
case *True*
then show *?thesis*
by (*metis* ‹(*out*[*L1*,*non-bottom-projection* π,*x*] ≠ {}) = (*out*[*undefinedness-completion X L1*,π,*x*] ≠ {*None*})› ‹π ∈ *undefinedness-completion X L2*› ‹*out*[*undefinedness-completion X L1*,π,*x*] = *Some* ‘ *out*[*L1*,*non-bottom-projection* π,*x*]› ‹*x* ∈ *X*› *assms*(*2*) *image-is-empty subset-empty subset-singletonD undefinedness-completion-out-projection-empty undefinedness-completion-out-shortening*)
next
case *False*

have *out*[*undefinedness-completion X L2*,π,*x*] = *Some* ‘ *out*[*L2*,*non-bottom-projection* π,*x*]
using *undefinedness-completion-out-projection-not-empty*[*OF assms*(*2*) ‹π ∈ *?L2*› ‹*x* ∈ *X*› *False*]
unfolding *undefinedness-completion-out-shortening*[*OF assms*(*2*) ‹π ∈ *?L2*› ‹*x* ∈ *X*›,*symmetric*] **.**

show *?thesis*
unfolding ‹*out*[*undefinedness-completion X L1*,π,*x*] = *Some* ‘ *out*[*L1*,*non-bottom-projection* π,*x*]›
unfolding ‹*out*[*undefinedness-completion X L2*,π,*x*] = *Some* ‘ *out*[*L2*,*non-bottom-projection* π,*x*]›
by (*metis* ‹(*out*[*L1*,*non-bottom-projection* π,*x*] ≠ {}) = (*out*[*undefinedness-completion X L1*,π,*x*] ≠ {*None*})› ‹*out*[*undefinedness-completion X L1*,π,*x*] = *Some* ‘ *out*[*L1*,*non-bottom-projection* π,*x*]› *subset-image-iff these-image-Some-eq*)
qed
qed
qed
ultimately show *?thesis*
by *meson*
qed
also have . . . = (∀ π ∈ *?L1* ∩ *?L2* . ∀ *x* ∈ *X* . *out*[*?L1*,π,*x*] ⊆ *out*[*?L2*,π,*x*])
(**is** *?A* = *?B*)
proof

```
    show ?A ⟹ ?B
      by blast
    show ?B ⟹ ?A
      by (metis IntD2 None-notin-image-Some assms(2) insert-subset undefinedness-completion-out-projection-empty undefinedness-completion-out-projection-not-empty undefinedness-completion-out-shortening)
  qed
  also have ... = (?L1 ⪯[X, red ({None} ∪ Some ‘ Y)] ?L2)
    unfolding type-1-conforms.simps red.simps
   using outputs-in-alphabet[OF undefinedness-completion-is-language[OF assms(2)]]
    by force
  finally show ?thesis .
qed


end
```

References

[1] W.-l. Huang and R. Sachtleben. *Conformance Relations Between Input/Output Languages*, pages 49–67. Springer Nature Switzerland, Cham, 2023.