Locally Nameless Sigma Calculus

Ludovic Henrio and Florian Kammüller and Bianca Lutz and Henry Sudhof

March 17, 2025

Abstract

We present a Theory of Objects based on the original functional ς-calculus by Abadi and Cardelli [1] but with an additional parameter to methods. We prove confluence of the operational semantics following the outline of Nipkow's proof of confluence for the λ-calculus reusing his general `Commutation.thy` [4] a generic diamond lemma reduction. We furthermore formalize a simple type system for our ς-calculus including a proof of type safety. The entire development uses the concept of Locally Nameless representation for binders [2]. We reuse an earlier proof of confluence [3] for a simpler ς-calculus based on de Bruijn indices and lists to represent objects.

Contents

1 List features

theory *ListPre*
imports *Main*
begin

lemma *drop-lem*[*rule-format*]:
fixes n :: *nat* **and** l :: *'a list* **and** g :: *'a list*
assumes *drop n l* = *drop n g* **and** *length l* = *length g* **and** $n <$ *length g*
shows $l!n = g!n$
proof −
from *assms*(*2*−*3*) **have** $n <$ *length l* **by** *simp*
from *Cons-nth-drop-Suc*[*OF this*] *Cons-nth-drop-Suc*[*OF assms*(*3*)] *assms*(*1*)
have $l!n$ # *drop* (*Suc n*) $l = g!n$ # *drop* (*Suc n*) g **by** *simp*
thus *?thesis* **by** *simp*
qed

lemma *mem-append-lem'*: $x \in$ *set* $(l$ @ $[y]) \Longrightarrow x \in$ *set* $l \vee x = y$
by *auto*

lemma *nth-last*: *length* $l = n \Longrightarrow (l$ @ $[x])!n = x$
by *auto*

lemma *take-n*:
fixes n :: *nat* **and** l :: *'a list* **and** g :: *'a list*
assumes *take n l* = *take n g* **and** *Suc n* ≤ *length g* **and** *length l* = *length g*
shows *take* (*Suc n*) $(l[n :=$ *g*!*n*$])$ = *take* (*Suc n*) g
proof −

```
  from assms(2) have ng: n < length g by simp
  with assms(3) have nlupd: n < length (l[n := g!n]) by simp
  hence nl: n < length l by simp
  from
    sym[OF assms(1)] id-take-nth-drop[OF ng] take-Suc-conv-app-nth[OF nlupd]
    nth-list-update-eq[OF nl] take-Suc-conv-app-nth[OF ng]
    upd-conv-take-nth-drop[OF nl] assms(2−3)
  show ?thesis by simp
qed

lemma drop-n-lem:
  fixes n :: nat and l :: 'a list
  assumes Suc n ≤ length l
  shows drop (Suc n) (l[n := x]) = drop (Suc n) l
  using assms by simp

lemma drop-n:
  fixes n :: nat and l :: 'a list and g :: 'a list
  assumes drop n l = drop n g and Suc n ≤ length g and length l = length g
  shows drop (Suc n) (l[n := g!n]) = drop (Suc n) g
proof −
  from assms(2−3) have Suc n ≤ length l by simp
  from drop-n-lem[OF this] assms(1) show ?thesis
    by (simp, (subst drop-Suc)+, (subst drop-tl)+, simp)
qed

lemma nth-fst[rule-format]: length l = n + 1 ⟶ (l @ [x])!0 = l!0
  by (induct l, simp-all)

lemma nth-zero-app:
  fixes l :: 'a list and x :: 'a and y :: 'a
  assumes l ≠ [] and l!0 = x
  shows(l @ [y])!0 = x
proof −
  have l ≠ [] ∧ l!0 = x ⟶ (l @ [y])!0 = x
    by (induct l, simp-all)
  with assms show ?thesis by simp
qed

lemma rev-induct2[consumes 1]:
  fixes xs :: 'a list and ys :: 'a list and P :: 'a list ⇒ 'a list ⇒ bool
  assumes
  length xs = length ys and P [] [] and
  ⋀x xs y ys. ⟦ length xs = length ys; P xs ys ⟧ ⟹ P (xs @ [x]) (ys @ [y])
  shows P xs ys
proof (simplesubst rev-rev-ident[symmetric])
  from assms(1) have lrev: length (rev xs) = length (rev ys) by simp
  from assms have P (rev (rev xs))(rev (rev ys))
    by (induct rule: list-induct2[OF lrev], simp-all)
```

```
  thus P xs (rev (rev ys)) by simp
qed

lemma list-induct3:
  ⋀ys zs. ⟦ length xs = length ys; length zs = length xs; P [] [] [];
            ⋀x xs y ys z zs. ⟦ length xs = length ys;
                               length zs = length xs; P xs ys zs ⟧
                            ⟹ P (x # xs)(y # ys)(z # zs)
  ⟧ ⟹ P xs ys zs
proof (induct xs, simp)
  case (Cons a xs ys zs)
  from ‹length (a#xs) = length ys› ‹length zs = length (a#xs)›
  have ys ≠ [] ∧ zs ≠ [] by auto
  then obtain b ly c lz where ys = b#ly and zs = c#lz
    by (auto simp: neq-Nil-conv)
  with ‹length (a#xs) = length ys› ‹length zs = length (a#xs)›
  obtain length xs = length ly and length lz = length xs
    by auto
  from
    Cons(5)[OF this Cons(1)[OF this ‹P [] [] []›]]
    Cons(5) ‹ys = b#ly› ‹zs = c#lz›
  show ?case by simp
qed

primrec list-insert :: 'a list ⇒ nat ⇒ 'a ⇒ 'a list where
  list-insert (ah#as) i a =
  (case i of
       0      ⇒ a#ah#as
      |Suc j  ⇒ ah#(list-insert as j a)) |

  list-insert [] i a = [a]

lemma insert-eq[simp]: ∀ i≤length l. (list-insert l i a)!i = a
  by (induct l, simp, intro strip, simp split: nat.split)

lemma insert-gt[simp]: ∀ i≤length l. ∀ j<i. (list-insert  l i a)!j = l!j
proof (induct l, simp)
  case (Cons x l) thus ?case
  proof (auto split: nat.split)
    fix n j assume n ≤ length l and j < Suc n
    with Cons(1) show (x#(list-insert l n a))!j = (x#l)!j
      by (cases j) simp-all
  qed
qed

lemma insert-lt[simp]: ∀ j≤length l. ∀ i≤j. (list-insert l i a)!Suc j = l!j
proof (induct l, simp)
  case (Cons x l) thus ?case
  proof (auto split: nat.split)
```

```
    fix n j assume j ≤ Suc (length l) and Suc n ≤ j
    with Cons(1) show (list-insert l n a)!j = l!(j − Suc 0)
      by (cases j) simp-all
  qed
qed

lemma insert-first[simp]: list-insert l 0 b = b#l
  by (induct l, simp-all)

lemma insert-prepend[simp]:
  i = Suc j ⟹ list-insert (a#l) i b = a # list-insert l j b
  by auto

lemma insert-lt2[simp]: ∀ j. ∀ i≤j. (list-insert  l i a)!Suc j = l!j
proof (induct l, simp)
  case (Cons x l) thus ?case
  proof (auto split: nat.split)
    fix n j assume Suc n ≤ j
    with Cons(1) show (list-insert l n a)!j = l!(j − Suc 0)
      by (cases j) simp-all
  qed
qed

lemma insert-commute[simp]:
  ∀ i≤length l. (list-insert (list-insert l i b) 0 a) =
                (list-insert (list-insert l 0 a ) (Suc i) b)
  by (induct l, auto split: nat.split)

lemma insert-length': ⋀i x. length (list-insert l i x) = length (x#l)
  by (induct l, auto split: nat.split)

lemma insert-length[simp]: length (list-insert l i b) = length (list-insert l j c)
  by (simp add: insert-length')

lemma insert-select[simp]: the ((f(l ↦ t)) l) = t
  by auto

lemma dom-insert[simp]: l ∈ dom f ⟹ dom (f(l ↦ t)) = dom f
  by auto

lemma insert-select2[simp]: l1 ≠ l2 ⟹ ((f(l1 ↦ t)) l2) = (f l2)
  by auto

lemma the-insert-select[simp]:
  ⟦ l2 ∈ dom f; l1 ≠ l2 ⟧ ⟹  the ((f(l1 ↦ t)) l2) =  the (f l2)
  by auto

lemma insert-dom-eq: dom f = dom f' ⟹ dom (f(l ↦ x)) = dom (f'(l ↦ x'))
  by auto
```

lemma *insert-dom-less-eq*:
  $[\![x \notin dom\ f;\ x \notin dom\ f';\ dom\ (f(x \mapsto y)) = dom\ (f'(x \mapsto y'))]\!]$
  $\Longrightarrow dom\ f = dom\ f'$
  by *auto*

lemma *one-more-dom*[*rule-format*]:
  $\forall l{\in}dom\ f\ .\ \exists f'.\ f = f'(l \mapsto the(f\ l)) \wedge l \notin dom\ f'$
proof
  fix l **assume** $l \in dom\ f$
  hence $\bigwedge la.\ f\ la = ((\lambda la.\ if\ la = l\ then\ None\ else\ f\ la)(l \mapsto the\ (f\ l)))\ la$
    by *auto*
  hence $f = (\lambda la.\ if\ la = l\ then\ None\ else\ f\ la)(l \mapsto the\ (f\ l))$
    by (*rule ext*)
  thus $\exists f'.\ f = f'(l \mapsto the(f\ l)\ ) \wedge l \notin dom\ f'$ **by** *auto*
qed

end

2 Finite maps with axclasses

theory *FMap* **imports** *ListPre* **begin**

type-synonym $('a, 'b)\ fmap = ('a :: finite) \rightharpoonup 'b$ (**infixl** ‹$-\sim>$› *50*)

class *inftype* =
assumes *infinite*: $\neg finite\ UNIV$

theorem *fset-induct*:
  $P\ \{\} \Longrightarrow (\bigwedge x\ (F{::}('a{::}finite)set).\ x \notin F \Longrightarrow P\ F \Longrightarrow P\ (insert\ x\ F)) \Longrightarrow P\ F$
proof (*rule-tac P=P* **and** *F=F* **in** *finite-induct*)
  from *finite-subset*[*OF subset-UNIV*] **show** *finite F* **by** *auto*
next
  assume $P\ \{\}$ **thus** $P\ \{\}$ **by** *simp*
next
  fix $x\ F$
  assume $\bigwedge x\ F.\ [\![x \notin F;\ P\ F]\!] \Longrightarrow P\ (insert\ x\ F)$ **and** $x \notin F$ **and** $P\ F$
  thus $P\ (insert\ x\ F)$ **by** *simp*
qed

theorem *fmap-unique*: $x = y \Longrightarrow (f{::}('a,'b)fmap)\ x = f\ y$
  by (*erule ssubst*, *rule refl*)

theorem *fmap-case*:
  $(F{::}('a\ -\sim>\ 'b)) = Map.empty \vee (\exists x\ y\ (F'{::}('a\ -\sim>\ 'b)).\ F = F'(x \mapsto y))$
proof (*cases* $F = Map.empty$)
  case *True* **thus** *?thesis* **by** (*rule disjI1*)
next
  case *False* **thus** *?thesis*

```
  proof (simp)
    from ‹F ≠ Map.empty› have ∃ x. F x ≠ None
    proof (rule contrapos-np)
      assume ¬ (∃ x. F x ≠ None)
      hence ∀ x. F x = None by simp
      hence ⋀x. F x = None by simp
      thus F = Map.empty by (rule ext)
    qed
    thus ∃ x y F'. F = F'(x ↦ y)
    proof
      fix x assume F x ≠ None
      hence ⋀y. F y = (F(x ↦ the (F x))) y by auto
      hence F = F(x ↦ the (F x)) by (rule ext)
      thus ?thesis by auto
    qed
  qed
qed


definition
  set-fmap :: 'a -~> 'b ⇒ ('a * 'b)set where
  set-fmap F = {(x, y). x ∈ dom F ∧ F x = Some y}

definition
  pred-set-fmap :: (('a -~> 'b) ⇒ bool) ⇒ (('a * 'b)set) ⇒ bool where
  pred-set-fmap P = (λS. P (λx. if x ∈ fst ' S
                                then (THE y. (∃ z. y = Some z ∧ (x, z) ∈ S))
                                else None))

definition
  fmap-minus-direct :: [('a -~> 'b), ('a * 'b)] ⇒ ('a -~> 'b) (infixl ‹--› 50)
where
  F -- x = (λz. if (fst x = z ∧ ((F (fst x)) = Some (snd x)))
                then None
                else (F z))

lemma insert-lem : insert x A = B ⟹ x ∈ B
  by auto

lemma fmap-minus-fmap:
  fixes F x a b
  assumes (F -- x) a = Some b
  shows F a = Some b
proof (rule ccontr, cases F a)
  case None hence a ∉ dom F by auto
  hence (F -- x) a = None
    unfolding fmap-minus-direct-def by auto
  with ‹(F -- x) a = Some b› show False by simp
next
```

```
  assume F a ≠ Some b
  case (Some y) thus False
  proof (cases fst x = a)
    case True thus False
    proof (cases snd x = y)
      case True with ‹F a = Some y› ‹fst x = a›
      have (F −− x) a = None unfolding fmap-minus-direct-def by auto
      with ‹(F −− x) a = Some b› show False by simp
    next
      case False with ‹F a = Some y› ‹fst x = a›
      have F (fst x) ≠ Some (snd x) by auto
      with ‹(F −− x) a = Some b› have F a = Some b
        unfolding fmap-minus-direct-def by auto
      with ‹F a ≠ Some b› show False by simp
    qed
  next
    case False with ‹(F −− x) a = Some b›
    have F a = Some b unfolding fmap-minus-direct-def by auto
    with ‹F a ≠ Some b› show False by simp
  qed
qed

lemma set-fmap-minus-iff:
  set-fmap ((F::(('a::finite) −~> 'b)) −− x) = set-fmap F − {x}
  unfolding set-fmap-def
proof (auto)
  fix a b assume (F −− x) a = Some b from fmap-minus-fmap[OF this]
  show ∃ y. F a = Some y by blast
next
  fix a b assume (F −− x) a = Some b from fmap-minus-fmap[OF this]
  show F a = Some b by assumption
next
  fix a b assume (F −− (a, b)) a = Some b
  with fmap-minus-fmap[OF this] show False
    unfolding fmap-minus-direct-def by auto
next
  fix a b assume (a,b) ≠ x and F a = Some b
  hence fst x ≠ a ∨ F (fst x) ≠ Some (snd x) by auto
  with ‹F a = Some b› show ∃ y. (F −− x) a = Some y
    unfolding fmap-minus-direct-def by (rule-tac x = b in exI, simp)
next
  fix a b assume (a,b) ≠ x and F a = Some b
  hence fst x ≠ a ∨ F (fst x) ≠ Some (snd x) by auto
  with ‹F a = Some b› show (F −− x) a = Some b
    unfolding fmap-minus-direct-def by simp
qed

lemma set-fmap-minus-insert:
  fixes F :: ('a::finite * 'b)set and  F'':: ('a::finite) −~> 'b and x
```

assumes $x \notin F$ **and** *insert* $x\ F = $ *set-fmap* F'
shows $F =$ *set-fmap* $(F' --\ x)$
proof −
from ‹$x \notin F$› *sym*[*OF* ‹*insert* $x\ F =$ *set-fmap* F'›] *set-fmap-minus-iff*[*of* $F'\ x$]
show *?thesis* **by** *simp*
qed

lemma *notin-fmap-minus*: $x \notin$ *set-fmap* $((F::(('a::finite) -\sim> 'b)) --\ x)$
by (*auto simp*: *set-fmap-minus-iff*)

lemma *fst-notin-fmap-minus-dom*:
fixes $F\ x$ **and** $F' :: ('a::finite) -\sim> 'b$
assumes *insert* $x\ F =$ *set-fmap* F'
shows *fst* $x \notin$ *dom* $(F' --\ x)$
proof (*rule ccontr*, *auto*)
fix y **assume** $(F' --\ x)\ (fst\ x) = Some\ y$
with *notin-fmap-minus*[*of* $x\ F'$]
have $y \neq snd\ x$
unfolding *set-fmap-def* **by** *auto*
moreover
from *insert-lem*[*OF* ‹*insert* $x\ F =$ *set-fmap* F'›]
have $F'\ (fst\ x) = Some\ (snd\ x)$
unfolding *set-fmap-def* **by** *auto*
ultimately show *False*
using *fmap-minus-fmap*[*OF* ‹$(F' --\ x)\ (fst\ x) = Some\ y$›]
by *simp*
qed

lemma *set-fmap-pair*:
$x \in$ *set-fmap* $F \Longrightarrow (fst\ x \in dom\ F \wedge snd\ x = the\ (F\ (fst\ x)))$
by (*simp add*: *set-fmap-def*, *auto*)

lemma *set-fmap-inv1*:
$[\![\ fst\ x \in dom\ F;\ snd\ x = the\ (F\ (fst\ x))\ ]\!] \Longrightarrow (F --\ x)(fst\ x \mapsto snd\ x) = F$
proof (*rule ext*)
fix *xa* **assume** *fst* $x \in$ *dom* F **and** *snd* $x =$ *the* $(F\ (fst\ x))$
thus $((F --\ x)(fst\ x \mapsto snd\ x))\ xa = F\ xa$
unfolding *fmap-minus-direct-def*
by (*case-tac* $xa = fst\ x$, *auto*)
qed

lemma *set-fmap-inv2*:
fst $x \notin$ *dom* $F \Longrightarrow$ *insert* x (*set-fmap* F) = *set-fmap* $(F(fst\ x \mapsto snd\ x))$
unfolding *set-fmap-def*
proof
assume *fst* $x \notin$ *dom* F
thus
insert $x\ \{(x,\ y).\ x \in dom\ F \wedge F\ x = Some\ y\} \subseteq$
$\{(xa,\ y).\ xa \in dom\ (F(fst\ x \mapsto snd\ x)) \wedge (F(fst\ x \mapsto snd\ x))\ xa = Some\ y\}$

```
    by force
next
  have
    ⋀z. z ∈ {(xa, y). xa ∈ dom (F(fst x ↦ snd x))
                  ∧ (F(fst x ↦ snd x)) xa = Some y}
    ⟹ z ∈ insert x {(x, y). x ∈ dom F ∧ F x = Some y}
    proof −
      fix z
      assume
        z: z ∈ {(xa, y). xa ∈ dom (F(fst x ↦ snd x))
                  ∧ (F(fst x ↦ snd x)) xa = Some y}
      hence z = x ∨ ((fst z) ∈ dom F ∧ F (fst z) = Some (snd z))
      proof (cases fst x = fst z)
        case True thus ?thesis using z by auto
      next
        case False thus ?thesis using z by auto
      qed
      thus z ∈ insert x {(x, y). x ∈ dom F ∧ F x = Some y} by fastforce
    qed
  thus
    {(xa, y). xa ∈ dom (F(fst x ↦ snd x)) ∧ (F(fst x ↦ snd x)) xa = Some y} ⊆
    insert x {(x, y). x ∈ dom F ∧ F x = Some y} by auto
qed

lemma rep-fmap-base: P (F::('a  −~> 'b)) = (pred-set-fmap P)(set-fmap F)
  unfolding pred-set-fmap-def set-fmap-def
proof (rule-tac f = P in arg-cong)
  have
    ⋀x. F x =
        (λx. if x ∈ fst ‘ {(x, y). x ∈ dom F ∧ F x = Some y}
             then THE y. ∃ z. y = Some z
                         ∧ (x, z) ∈ {(x, y). x ∈ dom F ∧ F x = Some y}
             else None) x
  proof auto
    fix a b
    assume F a = Some b
    hence ∃!x. x = Some b ∧ a ∈ dom F
    proof (rule-tac a = F a in ex1I)
      assume F a = Some b
      thus F a = Some b ∧ a ∈ dom F
        by (simp add: dom-def)
    next
      fix x assume F a = Some b and x = Some b ∧ a ∈ dom F
      thus x = F a by simp
    qed
    hence (THE y. y = Some b ∧ a ∈ dom F) = Some b ∧ a ∈ dom F
      by (rule theI′)
    thus Some b = (THE y. y = Some b ∧ a ∈ dom F)
      by simp
```

```
  next
    fix x assume nin-x: x ∉ fst ‘ {(x, y). x ∈ dom F ∧ F x = Some y}
    thus F x = None
    proof (cases F x)
      case None thus ?thesis by assumption
    next
      case (Some a)
      hence x ∈ fst ‘ {(x, y). x ∈ dom F ∧ F x = Some y}
        by (simp add: image-def dom-def)
      with nin-x show ?thesis by simp
    qed
  qed
  thus
    F = (λx. if x ∈ fst ‘ {(x, y). x ∈ dom F ∧ F x = Some y}
             then THE y. ∃ z. y = Some z
                         ∧ (x, z) ∈ {(x, y). x ∈ dom F ∧ F x = Some y}
             else None)
    by (rule ext)
qed

lemma rep-fmap:
  ∃ (Fp ::('a * 'b)set) (P'::('a * 'b)set ⇒ bool). P (F::('a -~> 'b)) = P' Fp
proof −
  from rep-fmap-base show ?thesis by blast
qed

theorem finite-fsets: finite (F::('a::finite)set)
proof −
  from finite-subset[OF subset-UNIV] show finite F by auto
qed

lemma finite-dom-fmap: finite (dom (F::('a -~> 'b))::('a::finite)set)
  by (rule finite-fsets)

lemma finite-fmap-ran: finite (ran (F::(('a::finite) -~> 'b)))
  unfolding ran-def
proof −
  from finite-dom-fmap finite-imageI
  have finite ((λx. the (F x)) ‘ (dom F))
    by blast
  moreover
  have {b. ∃ a. F a = Some b} = (λx. the (F x)) ‘ (dom F)
    unfolding image-def dom-def by force
  ultimately
  show finite {b. ∃ a. F a = Some b}  by simp
qed

lemma finite-fset-map: finite (set-fmap (F::(('a::finite) -~> 'b)))
proof −
```

from *finite-cartesian-product*[*OF finite-dom-fmap finite-fmap-ran*]
have *finite* (*dom F* × *ran F*) **.**
moreover
have *set-fmap F* ⊆ *dom F* × *ran F*
unfolding *set-fmap-def dom-def ran-def* **by** *fastforce*
ultimately
show *?thesis* **using** *finite-subset* **by** *auto*
qed

lemma *rep-fmap-imp*:
∀ *F x z*. *x* ∉ *dom* (*F*::('*a* −~> '*b*)) ⟶ *P F* ⟶ *P* (*F*(*x* ↦ *z*))
⟹ (∀ *F x z*. *x* ∉ *fst* ' (*set-fmap F*) ⟶ (*pred-set-fmap P*)(*set-fmap F*)
⟶ (*pred-set-fmap P*) (*insert* (*x*,*z*) (*set-fmap F*)))
proof (*clarify*)
fix *P F x z*
assume
∀ *F x z*. *x* ∉ *dom* (*F*::('*a* −~> '*b*)) ⟶ *P F* ⟶ *P* (*F*(*x* ↦ *z*)) **and**
x ∉ *fst* ' *set-fmap F* **and** (*pred-set-fmap P*)(*set-fmap F*)
hence *notin*: *x* ∉ *dom F*
unfolding *set-fmap-def image-def dom-def* **by** *simp*
moreover
from ‹*pred-set-fmap P* (*set-fmap F*)› **have** *P F* **by** (*simp add*: *rep-fmap-base*)
ultimately
have *P* (*F*(*x* ↦ *z*)) **using** ‹∀ *F x z*. *x* ∉ *dom F* ⟶ *P F* ⟶ *P* (*F*(*x* ↦ *z*))›
by *blast*
hence (*pred-set-fmap P*) (*set-fmap* (*F*(*x* ↦ *z*)))
by (*simp add*: *rep-fmap-base*)
moreover
from *notin*
have (*insert* (*x*,*z*) (*set-fmap F*)) = (*set-fmap* (*F*(*fst* (*x*,*z*) ↦ *snd* (*x*,*z*))))
by (*simp add*: *set-fmap-inv2*)
ultimately
show (*pred-set-fmap P*) (*insert* (*x*,*z*) (*set-fmap F*)) **by** *simp*
qed

lemma *empty-dom*:
fixes *g*
assumes {} = *dom g*
shows *g* = *Map.empty*
proof
fix *x* **from** *assms* **show** *g x* = *None* **by** *auto*
qed

theorem *fmap-induct*[*rule-format*, *case-names empty insert*]:
fixes *P* :: (('*a* :: *finite*) −~> '*b*) ⇒ *bool* **and** *F'* :: ('*a* −~> '*b*)
assumes
P Map.empty **and**
∀ (*F*::('*a* −~> '*b*)) *x z*. *x* ∉ *dom F* ⟶ *P F* ⟶ *P* (*F*(*x* ↦ *z*))
shows *P F'*

```
proof −
  {
    fix F :: ('a × 'b) set assume finite F
    hence ∀ F'. F = set-fmap F' ⟶ pred-set-fmap P (set-fmap F')
    proof (induct F)
      case empty thus ?case
      proof (intro strip)
        fix F' :: 'a −~> 'b assume {} = set-fmap F'
        hence ⋀a. F' a = None unfolding set-fmap-def by auto
        hence F' = Map.empty by (rule ext)
        with ‹P Map.empty› rep-fmap-base[of P Map.empty]
        show pred-set-fmap P (set-fmap F') by simp
      qed
    next
      case (insert x Fa) thus ?case
      proof (intro strip)
        fix Fb :: 'a −~> 'b
        assume insert x Fa = set-fmap Fb
        from
          set-fmap-minus-insert[OF ‹x ∉ Fa› this]
          ‹∀ F'. Fa = set-fmap F' ⟶ pred-set-fmap P (set-fmap F')›
          rep-fmap-base[of P Fb −− x]
        have P (Fb −− x) by blast
        with
          ‹∀ F x z. x ∉ dom F ⟶ P F ⟶ P (F(x ↦ z))›
          fst-notin-fmap-minus-dom[OF ‹insert x Fa = set-fmap Fb›]
        have P ((Fb −− x)(fst x ↦ snd x)) by blast
        moreover
        from
          insert-absorb[OF insert-lem[OF ‹insert x Fa = set-fmap Fb›]]
          set-fmap-minus-iff[of Fb x]
          set-fmap-inv2[OF
           fst-notin-fmap-minus-dom[OF ‹insert x Fa = set-fmap Fb›]]
        have set-fmap Fb = set-fmap ((Fb −− x)(fst x ↦ snd x))
          by simp
        ultimately
        show pred-set-fmap P (set-fmap Fb)
          using rep-fmap-base[of P (Fb −− x)(fst x ↦ snd x)]
          by simp
      qed
    qed
  }
  from this[OF finite-fset-map[of F']]
       rep-fmap-base[of P F']
  show P F' by blast
qed

lemma fmap-induct3[consumes 2, case-names empty insert]:
  ⋀(F2::('a::finite) −~> 'b) (F3::('a −~> 'b)).
```

```
  ⟦ dom (F1::('a -~> 'b)) = dom F2; dom F3 = dom F1;
    P Map.empty Map.empty Map.empty;
    ⋀x a b c (F1::('a -~> 'b)) (F2::('a -~> 'b)) (F3::('a -~> 'b)).
    ⟦ P F1 F2 F3; dom F1 = dom F2; dom F3 = dom F1; x ∉ dom F1 ⟧
    ⟹ P (F1(x ↦ a)) (F2(x ↦ b)) (F3(x ↦ c)) ⟧
  ⟹ P F1 F2 F3
proof (induct F1 rule: fmap-induct)
  case empty
   from ‹dom Map.empty = dom F2› have F2 = Map.empty by (simp add:
empty-dom)
  moreover
   from ‹dom F3 = dom Map.empty› have F3 = Map.empty by (simp add:
empty-dom)
  ultimately
  show ?case using ‹P Map.empty Map.empty Map.empty› by simp
next
  case (insert F x y) thus ?case
  proof (cases F2 = Map.empty)
    case True with ‹dom (F(x ↦ y)) = dom F2›
    have dom (F(x ↦ y)) = {} by auto
    thus ?thesis by auto
  next
    case False thus ?thesis
    proof (cases F3 = Map.empty)
      case True with ‹dom F3 = dom (F(x ↦ y))›
      have dom (F(x ↦ y)) = {} by simp
      thus ?thesis by simp
    next
      case False thus ?thesis
      proof −
        from ‹F2 ≠ Map.empty›
        have ∀ l∈dom F2. ∃ f'. F2 = f'(l ↦ the (F2 l)) ∧ l ∉ dom f'
          by (simp add: one-more-dom)
        moreover
        from ‹dom (F(x ↦ y)) = dom F2› have x ∈ dom F2 by force
        ultimately have ∃ f'. F2 = f'(x ↦ the (F2 x)) ∧ x ∉ dom f' by blast
        then obtain F2' where F2 = F2'(x ↦ the (F2 x)) and x ∉ dom F2'
          by auto

        from ‹F3 ≠ Map.empty›
        have ∀ l∈dom F3. ∃ f'. F3 = f'(l ↦ the (F3 l)) ∧ l ∉ dom f'
          by (simp add: one-more-dom)
        moreover from ‹dom F3 = dom (F(x ↦ y))› have x ∈ dom F3 by force
        ultimately have ∃ f'. F3 = f'(x ↦ the (F3 x)) ∧ x ∉ dom f' by blast
        then obtain F3' where F3 = F3'(x ↦ the (F3 x)) and x ∉ dom F3'
          by auto

        show ?thesis
        proof −
```

```
        from ‹dom (F(x ↦ y)) = dom F2› ‹F2 = F2'(x ↦ the (F2 x))›
        have dom (F(x ↦ y)) = dom (F2'(x ↦ the (F2 x))) by simp
        with ‹x ∉ dom F› ‹x ∉ dom F2'› have dom F = dom F2' by auto

        moreover
        from ‹dom F3 = dom (F(x ↦ y))› ‹F3 = F3'(x ↦ the (F3 x))›
        have dom (F(x ↦ y)) = dom (F3'(x ↦ the (F3 x))) by simp
        with ‹x ∉ dom F› ‹x ∉ dom F3'› have dom F3' = dom F by auto

        ultimately have P F F2' F3' using insert by simp

        with
          ‹⋀F1 F2 F3 x a b c.
            ⟦ P F1 F2 F3; dom F1 = dom F2; dom F3 = dom F1; x ∉ dom F1 ⟧
            ⟹ P (F1(x ↦ a)) (F2(x ↦ b)) (F3(x ↦ c))›
          ‹dom F = dom F2'›
          ‹dom F3' = dom F›
          ‹x ∉ dom F›
        have P (F(x ↦ y)) (F2'(x ↦ the (F2 x))) (F3'(x ↦ the (F3 x)))
          by simp
        with ‹F2 = F2'(x ↦ the (F2 x))› ‹F3 = F3'(x ↦ the (F3 x))›
        show P (F(x ↦ y)) F2 F3 by simp
      qed
    qed
  qed
 qed
qed

lemma fmap-ex-cof2:
 ⋀(P::'c ⇒ 'c ⇒ 'b option ⇒ 'b option ⇒ 'a ⇒ bool)
   (f'::('a::finite) -~> 'b).
 ⟦ dom f' = dom (f::('a -~> 'b));
   ∀ l∈dom f. (∃ L. finite L
               ∧ (∀ s p. s ∉ L ∧ p ∉ L ∧ s ≠ p
                  ⟶ P s p (f l) (f' l) l)) ⟧
 ⟹ ∃ L. finite L ∧ (∀ l∈dom f. (∀ s p. s ∉ L ∧ p ∉ L ∧ s ≠ p
                                ⟶ P s p (f l) (f' l) l))
proof (induct f rule: fmap-induct)
 case empty thus ?case by blast
next
 case (insert f l t P f') note imp = this(2) and pred = this(4)
 define pred-cof where pred-cof L b b' l ⟷ (∀ s p. s ∉ L ∧ p ∉ L ∧ s ≠ p ⟶
P s p b b' l)
   for L b b' l
 from
   map-upd-nonempty[of f l t] ‹dom f' = dom (f(l ↦ t))›
   one-more-dom[of l f']
 obtain f'a where
   f' = f'a(l ↦ the(f' l)) and l ∉ dom f'a and
```

```
  dom (f'a(l ↦ the(f' l))) = dom (f(l ↦ t))
  by auto
from ‹l ∉ dom f›
have
  fla: ∀ la∈dom f. f la = (f(l ↦ t)) la and
  ∀ la∈dom f. f'a la = (f'a(l ↦ the(f' l))) la
  by auto
with ‹f' = f'a(l ↦ the(f' l))›
have f'ala: ∀ la∈dom f. f'a la = f' la by simp
have ∃ L. finite L ∧ (∀ la∈dom f. pred-cof L (f la) (f'a la) la)
  unfolding pred-cof-def
proof
  (intro imp[OF insert-dom-less-eq[OF ‹l ∉ dom f'a› ‹l ∉ dom f›
                                      ‹dom (f'a(l ↦ the(f' l))) = dom (f(l ↦ t))›]],
    intro strip)
  fix la assume la ∈ dom f
  with fla f'ala
  have
    la ∈ dom (f(l ↦ t)) and
    f la = (f(l ↦ t)) la and f'a la = f' la
    by auto
  with pred
  show ∃ L. finite L ∧ (∀ s p. s ∉ L ∧ p ∉ L ∧ s ≠ p ⟶ P s p (f la) (f'a la) la)
    by (elim ssubst, blast)
qed
with fla f'ala obtain L where
  finite L and predf: ∀ la∈dom f. pred-cof L ((f(l ↦ t)) la) (f' la) la
  by auto
moreover
have l ∈ dom (f(l ↦ t)) by simp
with pred obtain L' where
  finite L' and predfl: pred-cof L' ((f(l ↦ t)) l) (f' l) l
  unfolding pred-cof-def
  by blast
ultimately show ?case
proof (rule-tac x = L ∪ L' in exI,
    intro conjI, simp, intro strip)
  fix s p la assume
    sp: s ∉ L ∪ L' ∧ p ∉ L ∪ L' ∧ s ≠ p and indom: la ∈ dom (f(l ↦ t))
  show P s p ((f(l ↦ t)) la) (f' la) la
  proof (cases la = l)
    case True with sp predfl show ?thesis
      unfolding pred-cof-def
      by simp
  next
    case False with indom sp predf show ?thesis
      unfolding pred-cof-def
      by force
  qed
```

qed
qed

lemma *fmap-ex-cof*:
fixes
$P :: 'c \Rightarrow 'c \Rightarrow 'b\ option \Rightarrow ('a::finite) \Rightarrow bool$
assumes
$\forall\, l{\in}dom\ (f::('a\ -\!\!\sim\!\!>\ 'b)).$
$(\exists\, L.\ finite\ L \wedge (\forall\, s\ p.\ s \notin L \wedge p \notin L \wedge s \neq p \longrightarrow P\ s\ p\ (f\ l)\ l))$
shows
$\exists\, L.\ finite\ L \wedge (\forall\, l{\in}dom\ f.\ (\forall\, s\ p.\ s \notin L \wedge p \notin L \wedge s \neq p \longrightarrow P\ s\ p\ (f\ l)\ l))$
using *assms fmap-ex-cof2*[*of f f* $\lambda s\ p\ b\ b'\ l.\ P\ s\ p\ b\ l$] **by** *auto*

lemma *fmap-ball-all2*:
fixes
$Px :: 'c \Rightarrow 'd \Rightarrow bool$ **and**
$P :: 'c \Rightarrow 'd \Rightarrow 'b\ option \Rightarrow bool$
assumes
$\forall\, l{\in}dom\ (f::('a::finite)\ -\!\!\sim\!\!>\ 'b).\ \forall\, (x::'c)\ (y::'d).\ Px\ x\ y \longrightarrow P\ x\ y\ (f\ l)$
shows
$\forall\, x\ y.\ Px\ x\ y \longrightarrow (\forall\, l{\in}dom\ f.\ P\ x\ y\ (f\ l))$
proof (*intro strip*)
fix $x\ y\ l$ **assume** $Px\ x\ y$ **and** $l \in dom\ f$
with *assms* **show** $P\ x\ y\ (f\ l)$ **by** *blast*
qed

lemma *fmap-ball-all2'*:
fixes
$Px :: 'c \Rightarrow 'd \Rightarrow bool$ **and**
$P :: 'c \Rightarrow 'd \Rightarrow 'b\ option \Rightarrow ('a::finite) \Rightarrow bool$
assumes
$\forall\, l{\in}dom\ (f::('a\ -\!\!\sim\!\!>\ 'b)).\ \forall\, (x::'c)\ (y::'d).\ Px\ x\ y \longrightarrow P\ x\ y\ (f\ l)\ l$
shows
$\forall\, x\ y.\ Px\ x\ y \longrightarrow (\forall\, l{\in}dom\ f.\ P\ x\ y\ (f\ l)\ l)$
proof (*intro strip*)
fix $x\ y\ l$ **assume** $Px\ x\ y$ **and** $l \in dom\ f$
with *assms* **show** $P\ x\ y\ (f\ l)\ l$ **by** *blast*
qed

lemma *fmap-ball-all3*:
fixes
$Px :: 'c \Rightarrow 'd \Rightarrow 'e \Rightarrow bool$ **and**
$P :: 'c \Rightarrow 'd \Rightarrow 'e \Rightarrow 'b\ option \Rightarrow 'b\ option \Rightarrow bool$ **and**
$f :: ('a::finite)\ -\!\!\sim\!\!>\ 'b$ **and** $f' :: 'a\ -\!\!\sim\!\!>\ 'b$
assumes
$dom\ f' = dom\ f$ **and**
$\forall\, l{\in}dom\ f.$
$\quad \forall\, (x::'c)\ (y::'d)\ (z::'e).\ Px\ x\ y\ z \longrightarrow P\ x\ y\ z\ (f\ l)\ (f'\ l)$
shows

$\forall x\ y\ z.\ Px\ x\ y\ z \longrightarrow (\forall l{\in}dom\ f.\ P\ x\ y\ z\ (f\ l)\ (f'\ l))$
proof (*intro strip*)
fix $x\ y\ z\ l$ **assume** $Px\ x\ y\ z$ **and** $l \in dom\ f$
with *assms* **show** $P\ x\ y\ z\ (f\ l)\ (f'\ l)$ **by** *blast*
qed

lemma *fmap-ball-all4'*:
fixes
$Px :: 'c \Rightarrow 'd \Rightarrow 'e \Rightarrow 'f \Rightarrow bool$ **and**
$P :: 'c \Rightarrow 'd \Rightarrow 'e \Rightarrow 'f \Rightarrow 'b\ option \Rightarrow ('a{::}finite) \Rightarrow bool$
assumes
$\forall l{\in}dom\ (f{::}('a\ -\!\!\sim\!\!>\ 'b)).$
$\forall (x{::}'c)\ (y{::}'d)\ (z{::}'e)\ (a{::}'f).\ Px\ x\ y\ z\ a \longrightarrow P\ x\ y\ z\ a\ (f\ l)\ l$
shows
$\forall x\ y\ z\ a.\ Px\ x\ y\ z\ a \longrightarrow (\forall l{\in}dom\ f.\ P\ x\ y\ z\ a\ (f\ l)\ l)$
proof (*intro strip*)
fix $x\ y\ z\ a\ l$ **assume** $Px\ x\ y\ z\ a$ **and** $l \in dom\ f$
with *assms* **show** $P\ x\ y\ z\ a\ (f\ l)\ l$ **by** *blast*
qed

end

3 Locally Nameless representation of basic Sigma calculus enriched with formal parameter

theory *Sigma*
imports *../preliminary/FMap*
begin

3.1 Infrastructure for the finite maps

axiomatization *max-label* :: *nat* **where**
LabelAvail: *max-label* > *10*

definition $Label = \{n :: nat.\ n \leq max\text{-}label\}$

typedef *Label* = *Label*
unfolding *Label-def* **by** *auto*

lemmas *finite-Label-set* = *Finite-Set.finite-Collect-le-nat*[*of max-label*]

lemma *Univ-abs-label*:
$(UNIV :: (Label\ set)) = Abs\text{-}Label\ `\ \{n :: nat.\ n \leq max\text{-}label\}$
proof −
have $\forall x :: Label.\ x : Abs\text{-}Label\ `\ \{n :: nat.\ n \leq max\text{-}label\}$
proof
fix $x :: Label$
have $Rep\text{-}Label\ x \in \{n.\ n \leq max\text{-}label\}$

```
    by (fold Label-def, rule Rep-Label)
  hence Abs-Label (Rep-Label x) ∈ Abs-Label ‘ {n. n ≤ max-label}
    by (rule imageI)
  thus x ∈ Abs-Label ‘ {n. n ≤ max-label}
    by (simp add: Rep-Label-inverse)
 qed
 thus ?thesis by force
qed

lemma finite-Label: finite (UNIV :: (Label set))
 by (simp add: Univ-abs-label finite-Label-set)

instance Label :: finite
proof
 show finite (UNIV :: (Label set)) by (rule finite-Label)
qed

consts
Lsuc :: (Label set) ⇒ Label ⇒ Label
Lmin :: (Label set) ⇒ Label
Lmax :: (Label set) ⇒ Label

definition Llt :: [Label, Label] ⇒ bool (infixl ‹<› 50) where
 Llt a b == Rep-Label a < Rep-Label b
definition Lle :: [Label, Label] ⇒ bool (infixl ‹≤› 50) where
 Lle a b == Rep-Label a ≤ Rep-Label b

definition Ltake-eq :: [Label set, (Label ⇀ 'a), (Label ⇀ 'a)] ⇒ bool
where  Ltake-eq L f g  == ∀ l∈L. f l = g l

lemma Ltake-eq-all:
 fixes f g
 assumes dom f = dom g and Ltake-eq (dom f) f g
 shows f = g
proof
 fix x from assms show f x = g x
  unfolding Ltake-eq-def
  by (cases x ∈ dom f, auto)
qed

lemma Ltake-eq-dom:
 fixes L :: Label set and f :: Label -~> 'a
 assumes L ⊆ dom f and card L = card (dom f)
 shows L = (dom f)
proof (auto)
 fix x :: Label assume x ∈ L
 with in-mono[OF assms(1)] show ∃ y. f x = Some y by blast
next
 fix x y assume f x = Some y
```

from *card-seteq*[*OF finite-dom-fmap*[*of f*] ‹$L \subseteq dom\ f$›] ‹$card\ L = card\ (dom\ f)$›
have $L = dom\ f$ **by** *simp*
with ‹$f\ x = Some\ y$› **show** $x \in L$ **by** *force*
qed

3.2 Object-terms in Locally Nameless representation notation, beta-reduction and substitution

datatype *type* = *Object Label* $-\sim>$ (*type* × *type*)

datatype *bVariable* = *Self nat* | *Param nat*
type-synonym *fVariable* = *string*

3.2.1 Enriched Sigma datatype of objects

datatype *sterm* =
Bvar bVariable
| *Fvar fVariable*
| *Obj Label* $-\sim>$ *sterm type*
| *Call sterm Label sterm*
| *Upd sterm Label sterm*

datatype-compat *sterm*

primrec *applyPropOnOption*:: (*sterm* ⇒ *bool*) ⇒ *sterm option* ⇒ *bool* **where**
f1: *applyPropOnOption P None* = *True* |
f2: *applyPropOnOption P* (*Some t*) = *P t*

lemma *sterm-induct*[*case-names Bvar Fvar Obj Call Upd empty insert*]:
fixes
t :: *sterm* **and** *P1* :: *sterm* ⇒ *bool* **and**
f :: *Label* $-\sim>$ *sterm* **and** *P3* :: (*Label* $-\sim>$ *sterm*) ⇒ *bool*
assumes
$\bigwedge b.$ *P1* (*Bvar b*) **and**
$\bigwedge x.$ *P1* (*Fvar x*) **and**
a-obj: $\bigwedge f\ T.$ *P3 f* $\Longrightarrow$ *P1* (*Obj f T*) **and**
$\bigwedge t1\ l\ t2.$ ⟦ *P1 t1*; *P1 t2* ⟧ $\Longrightarrow$ *P1* (*Call t1 l t2*) **and**
$\bigwedge t1\ l\ t2.$ ⟦ *P1 t1*; *P1 t2* ⟧ $\Longrightarrow$ *P1* (*Upd t1 l t2*) **and**
P3 Map.empty **and**
a-f: $\bigwedge t1\ f\ l\ .$ ⟦ $l \notin dom\ f$; *P1 t1*; *P3 f* ⟧ $\Longrightarrow$ (*P3* ($f(l \mapsto t1)$))
shows *P1 t* ∧ *P3 f*
proof −
have ∀ *t* (*f*::*Label* $-\sim>$ *sterm*) *l*. *P1 t* ∧ (*applyPropOnOption P1*) (*f l*)
proof (*intro strip*)
fix *t* :: *sterm* **and** *f'* :: *Label* $-\sim>$ *sterm* **and** *l* :: *Label*
define *foobar* **where** *foobar* = *f' l*
from *assms* **show** *P1 t* ∧ *applyPropOnOption P1 foobar*
proof (*induct-tac t* **and** *foobar rule*: *compat-sterm.induct compat-sterm-option.induct*, *auto*)
fix *f* :: *Label* $-\sim>$ *sterm* **and** *T* :: *type*

```
      assume ⋀x. applyPropOnOption P1 (f x)
      with a-f ‹P3 Map.empty› have P3 f
      proof (induct f rule: fmap-induct, simp)
        case (insert F x z)
        note
          P1F'   = ‹⋀xa. applyPropOnOption P1 ((F(x ↦ z)) xa)› and
          predP3 = ‹⟦ ⋀l f t1. ⟦l ∉ dom f; P1 t1; P3 f⟧ ⟹ P3 (f(l ↦ t1));
                      P3 Map.empty; ⋀x. applyPropOnOption P1 (F x)⟧
                    ⟹ P3 F› and
          P3F'   = ‹⋀l f t f. ⟦ l ∉ dom f; P1 t; P3 f ⟧ ⟹ P3 (f(l ↦ t))›
        have ⋀xa. applyPropOnOption P1 (F xa)
        proof −
          fix xa :: Label show applyPropOnOption P1 (F xa)
          proof (cases xa = x)
            case True with P1F' ‹x ∉ dom F› have F xa = None by force
            thus ?thesis by simp
          next
            case False hence eq: F xa = (F(x ↦ z)) xa by auto
            from P1F'[of xa] show applyPropOnOption P1 (F xa)
              by (simp only: ssubst[OF eq])
          qed
        qed
        from a-f predP3[OF - ‹P3 Map.empty› this] have P3F: P3 F by simp
        from P1F'[of x]
        have applyPropOnOption P1 (Some z) by auto
        hence P1 z by simp
        from a-f[OF ‹x ∉ dom F› this P3F] show ?case by assumption
      qed
      with a-obj show P1 (Obj f T) by simp
    qed
  qed
  with assms show ?thesis
  proof (auto)
    assume ⋀l f t1. ⟦l ∉ dom f; P3 f⟧ ⟹ P3 (f(l ↦ t1))
    with ‹P3 Map.empty› show P3 f by (rule fmap-induct)
  qed
qed

lemma ball-tsp-P3:
  fixes
  P1 :: sterm ⇒ bool and
  P2 :: sterm ⇒ fVariable ⇒ fVariable ⇒ bool and
  P3 :: sterm ⇒ bool and f :: Label −~> sterm
  assumes
  ⋀t. ⟦ P1 t; ∀ s p. s ∉ L ∧ p ∉ L ∧ s ≠ p ⟶ P2 t s p ⟧ ⟹ P3 t and
  ∀ l∈dom f. P1 (the(f l)) and
  ∀ l∈dom f. ∀ s p. s ∉ L ∧ p ∉ L ∧ s ≠ p ⟶ P2 (the(f l)) s p
  shows ∀ l∈dom f. P3 (the(f l))
proof (intro strip)
```

fix l **assume** $l \in dom\ f$ **with** *assms(2)* **have** *P1* $(the(f\ l))$ **by** *blast*
moreover
from *assms(3)* $\langle l \in dom\ f \rangle$ **have** $\forall s\ p.\ s \notin L \wedge p \notin L \wedge s \neq p \longrightarrow P2\ (the(f\ l))\ s\ p$
by *blast*
ultimately
show *P3* $(the(f\ l))$ **using** *assms(1)* **by** *simp*
qed

lemma *ball-tt'sp-P3*:
fixes
P1 :: *sterm* ⇒ *sterm* ⇒ *bool* **and**
P2 :: *sterm* ⇒ *sterm* ⇒ *fVariable* ⇒ *fVariable* ⇒ *bool* **and**
P3 :: *sterm* ⇒ *sterm* ⇒ *bool* **and**
f :: *Label* $-^{\sim}>$ *sterm* **and** f' :: *Label* $-^{\sim}>$ *sterm*
assumes
$\bigwedge t\ t'.\ [\![\ P1\ t\ t';\ \forall s\ p.\ s \notin L \wedge p \notin L \wedge s \neq p \longrightarrow P2\ t\ t'\ s\ p\ ]\!] \Longrightarrow P3\ t\ t'$ **and**
$dom\ f = dom\ f'$ **and**
$\forall l{\in}dom\ f.\ P1\ (the(f\ l))\ (the(f'\ l))$ **and**
$\forall l{\in}dom\ f.\ \forall s\ p.\ s \notin L \wedge p \notin L \wedge s \neq p \longrightarrow P2\ (the(f\ l))\ (the(f'\ l))\ s\ p$
shows $\forall l{\in}dom\ f'.\ P3\ (the(f\ l))\ (the(f'\ l))$
proof (*intro strip*)
fix l **assume** $l \in dom\ f'$ **with** *assms(2−3)* **have** *P1* $(the(f\ l))\ (the(f'\ l))$
by *blast*
moreover
from *assms(2,4)* $\langle l \in dom\ f' \rangle$
have $\forall s\ p.\ s \notin L \wedge p \notin L \wedge s \neq p \longrightarrow P2\ (the\ (f\ l))\ (the(f'\ l))\ s\ p$ **by** *blast*
ultimately
show *P3* $(the(f\ l))\ (the(f'\ l))$ **using** *assms(1)*[*of* $the(f\ l)$ $the(f'\ l)$]
by *simp*
qed

3.2.2 Free variables

primrec
FV :: *sterm* ⇒ *fVariable set*
and
FVoption :: *sterm option* ⇒ *fVariable set*
where
FV-Bvar : $FV(Bvar\ b) = \{\}$
| *FV-Fvar* : $FV(Fvar\ x) = \{x\}$
| *FV-Call* : $FV(Call\ t\ l\ a) = FV\ t \cup FV\ a$
| *FV-Upd* : $FV(Upd\ t\ l\ s) = FV\ t \cup FV\ s$
| *FV-Obj* : $FV(Obj\ f\ T) = (\bigcup\ l \in dom\ f.\ FVoption(f\ l))$
| *FV-None* : $FVoption\ None = \{\}$
| *FV-Some* : $FVoption\ (Some\ t) = FV\ t$

definition *closed* :: *sterm* ⇒ *bool* **where**
$closed\ t \longleftrightarrow FV\ t = \{\}$

lemma *finite-FV-FVoption*: *finite* (*FV t*) ∧ *finite* (*FVoption s*)
by(*induct - t - s rule*: *compat-sterm-sterm-option.induct*, *simp-all*)

lemma *finite-FV*[*simp*]: *finite* (*FV t*)
by (*simp add*: *finite-FV-FVoption*)

lemma *FV-and-cofinite*: ⟦ ∀ *x*. *x* ∉ *L* ⟶ *P x*; *finite L* ⟧
⟹ ∃ *L'*. (*finite L'* ∧ *FV t* ⊆ *L'* ∧ (∀ *x*. *x* ∉ *L'* ⟶ *P x*))
by (*rule-tac x* = *L* ∪ *FV t* **in** *exI*, *auto*)

lemma *exFresh-s-p-cof*:
fixes *L* :: *fVariable set*
assumes *finite L*
shows ∃ *s p*. *s* ∉ *L* ∧ *p* ∉ *L* ∧ *s* ≠ *p*
proof −
from *assms* **have** ∃ *s*. *s* ∉ *L* **by** (*simp only*: *ex-new-if-finite*[*OF infinite-UNIV-listI*])
then obtain *s* **where** *s* ∉ *L* **..**
moreover
from ‹*finite L*› **have** *finite* (*L* ∪ {*s*}) **by** *simp*
hence ∃ *p*. *p* ∉ *L* ∪ {*s*} **by** (*simp only*: *ex-new-if-finite*[*OF infinite-UNIV-listI*])
then obtain *p* **where** *p* ∉ *L* ∪ {*s*} **..**
ultimately show *?thesis* **by** *blast*
qed

lemma *FV-option-lem*: ∀ *l*∈*dom f*. *FV* (*the*(*f l*)) = *FVoption* (*f l*)
by *auto*

3.2.3 Term opening

primrec
sopen :: [*nat*, *sterm*, *sterm*, *sterm*] ⇒ *sterm*
(‹{- → [-,-]} -› [*0*, *0*, *0*, *300*] *300*)
and
sopen-option :: [*nat*, *sterm*, *sterm*, *sterm option*] ⇒ *sterm option*
where
sopen-Bvar:
{*k* → [*s*,*p*]}(*Bvar b*) = (*case b of* (*Self i*) ⇒ (*if* (*k* = *i*) *then s else* (*Bvar b*))
| (*Param i*) ⇒ (*if* (*k* = *i*) *then p else* (*Bvar b*)))
| *sopen-Fvar*: {*k* → [*s*,*p*]}(*Fvar x*) = *Fvar x*
| *sopen-Call*: {*k* → [*s*,*p*]}(*Call t l a*) = *Call* ({*k* → [*s*,*p*]}*t*) *l* ({*k* → [*s*,*p*]}*a*)
| *sopen-Upd* : {*k* → [*s*,*p*]}(*Upd t l u*) = *Upd* ({*k* → [*s*,*p*]}*t*) *l* ({(*Suc k*) → [*s*,*p*]}*u*)
| *sopen-Obj* : {*k* → [*s*,*p*]}(*Obj f T*) = *Obj* (λ*l*. *sopen-option* (*Suc k*) *s p* (*f l*)) *T*
| *sopen-None*: *sopen-option k s p None* = *None*
| *sopen-Some*: *sopen-option k s p* (*Some t*) = *Some* ({*k* → [*s*,*p*]}*t*)

definition *openz* :: [*sterm*, *sterm*, *sterm*] ⇒ *sterm* (‹(-)$^{[-,-]}$› [*50*, *0*, *0*] *50*) **where**
$t^{[s,p]} = \{0 \to [s,p]\}t$

lemma *sopen-eq-Fvar*:
fixes *n s p t x*
assumes $\{n \to [Fvar\ s, Fvar\ p]\}\ t = Fvar\ x$
shows
$(t = Fvar\ x) \vee (x = s \wedge t = (Bvar\ (Self\ n)))$
$\vee\ (x = p \wedge t = (Bvar\ (Param\ n)))$
proof (*cases t*)
case *Obj* **with** *assms* **show** *?thesis* **by** *simp*
next
case *Call* **with** *assms* **show** *?thesis* **by** *simp*
next
case *Upd* **with** *assms* **show** *?thesis* **by** *simp*
next
case (*Fvar y*) **with** *assms* **show** *?thesis*
by (*cases y = x*, *simp-all*)
next
case (*Bvar b*) **thus** *?thesis*
proof (*cases b*)
case (*Self k*) **with** *assms Bvar* **show** *?thesis*
by (*cases k = n*) *simp-all*
next
case (*Param k*) **with** *assms Bvar* **show** *?thesis*
by (*cases k = n*) *simp-all*
qed
qed

lemma *sopen-eq-Fvar′*:
assumes $\{n \to [Fvar\ s, Fvar\ p]\}\ t = Fvar\ x$ **and** $x \neq s$ **and** $x \neq p$
shows $t = Fvar\ x$
proof −
from *sopen-eq-Fvar*[*OF assms*(*1*)] ‹$x \neq s$› ‹$x \neq p$› **show** *?thesis*
by *auto*
qed

lemma *sopen-eq-Bvar*:
fixes *n s p t b*
assumes $\{n \to [Fvar\ s, Fvar\ p]\}\ t = Bvar\ b$
shows $t = Bvar\ b$
proof (*cases t*)
case *Fvar* **with** *assms* **show** *?thesis* **by** (*simp add*: *openz-def*)
next
case *Obj* **with** *assms* **show** *?thesis* **by** (*simp add*: *openz-def*)
next
case *Call* **with** *assms* **show** *?thesis* **by** (*simp add*: *openz-def*)
next

```
  case Upd with assms show ?thesis by (simp add: openz-def)
next
  case (Bvar b') show ?thesis
  proof (cases b')
    case (Self k) with assms Bvar show ?thesis
      by (cases k = n) (simp-all add: openz-def)
  next
    case (Param k) with assms Bvar show ?thesis
      by (cases k = n) (simp-all add: openz-def)
  qed
qed

lemma sopen-eq-Obj:
  fixes n s p t f T
  assumes {n → [Fvar s,Fvar p]} t = Obj f T
  shows
  ∃f'. {n → [Fvar s, Fvar p]} Obj f' T = Obj f T
     ∧  t = Obj f' T
proof (cases t)
  case Fvar with assms show ?thesis by simp
next
  case Obj with assms show ?thesis by simp
next
  case Call with assms show ?thesis by simp
next
  case Upd with assms show ?thesis by simp
next
  case (Bvar b) thus ?thesis
  proof (cases b)
    case (Self k) with assms Bvar show ?thesis
      by (cases k = n) simp-all
  next
    case (Param k) with assms Bvar show ?thesis
      by (cases k = n) simp-all
  qed
qed

lemma sopen-eq-Upd:
  fixes n s p t t1 l t2
  assumes {n → [Fvar s,Fvar p]} t = Upd t1 l t2
  shows
  ∃t1' t2'. {n → [Fvar s,Fvar p]} t1' = t1
          ∧ {(Suc n) → [Fvar s,Fvar p]} t2' = t2 ∧ t = Upd t1' l t2'
proof (cases t)
  case Fvar with assms show ?thesis by simp
next
  case Obj with assms show ?thesis by simp
next
  case Call with assms show ?thesis by simp
```

next
case *Upd* **with** *assms* **show** *?thesis* **by** *simp*
next
case (*Bvar b*) **thus** *?thesis*
proof (*cases b*)
case (*Self k*) **with** *assms Bvar* **show** *?thesis*
by (*cases k = n*) *simp-all*
next
case (*Param k*) **with** *assms Bvar* **show** *?thesis*
by (*cases k = n*) *simp-all*
qed
qed

lemma *sopen-eq-Call*:
fixes *n s p t t1 l t2*
assumes {*n* → [*Fvar s*,*Fvar p*]} *t* = *Call t1 l t2*
shows
∃ *t1′ t2′*. {*n* → [*Fvar s*,*Fvar p*]} *t1′* = *t1*
∧ {*n* → [*Fvar s*,*Fvar p*]} *t2′* = *t2* ∧ *t* = *Call t1′ l t2′*
proof (*cases t*)
case *Fvar* **with** *assms* **show** *?thesis* **by** *simp*
next
case *Obj* **with** *assms* **show** *?thesis* **by** *simp*
next
case *Call* **with** *assms* **show** *?thesis* **by** *simp*
next
case *Upd* **with** *assms* **show** *?thesis* **by** *simp*
next
case (*Bvar b*) **thus** *?thesis*
proof (*cases b*)
case (*Self k*) **with** *assms Bvar* **show** *?thesis*
by (*cases k = n*) *simp-all*
next
case (*Param k*) **with** *assms Bvar* **show** *?thesis*
by (*cases k = n*) *simp-all*
qed
qed

lemma *dom-sopenoption-lem*[*simp*]: *dom* (λ*l*. *sopen-option k s t* (*f l*)) = *dom f*
by (*auto*, *case-tac x* ∈ *dom f*, *auto*)

lemma *sopen-option-lem*:
∀ *l*∈*dom f*. {*n* → [*s*,*p*]} *the*(*f l*) = *the* (*sopen-option n s p* (*f l*))
by *auto*

lemma *pred-sopenoption-lem*:
(∀ *l*∈*dom* (λ*l*. *sopen-option n s p* (*f l*)).
(*P*::*sterm* ⇒ *bool*) (*the* (*sopen-option n s p* (*f l*)))) =
(∀ *l*∈*dom f*. (*P*::*sterm* ⇒ *bool*) ({*n* → [*s*,*p*]} *the* (*f l*)))

by (*simp*, *force*)

lemma *sopen-FV*[*rule-format*]:
$\forall\, n\ s\ p.\ FV\ (\{n \to [s,p]\}\ t) \subseteq FV\ t \cup FV\ s \cup FV\ p$
proof −
fix *u*
have
$(\forall\, n\ s\ p.\ FV\ (\{n \to [s,p]\}\ t) \subseteq FV\ t \cup FV\ s \cup FV\ p)$
$\&(\forall\, n\ s\ p.\ FVoption\ (sopen\text{-}option\ n\ s\ p\ u) \subseteq FVoption\ u \cup FV\ s \cup FV\ p)$
apply (*induct u rule*: *compat-sterm-sterm-option.induct*)
apply (*auto split*: *bVariable.split*)
apply (*metis* (*no-types*, *lifting*) *FV-Some UnE domI sopen-Some subsetCE*)
apply *blast*
apply *blast*
done
from *conjunct1*[*OF this*] **show** *?thesis* **by** *assumption*
qed

lemma *sopen-commute*[*rule-format*]:
$\forall\, n\ k\ s\ p\ s'\ p'.\ n \neq k$
$\longrightarrow \{n \to [Fvar\ s',\ Fvar\ p']\}\ \{k \to [Fvar\ s,\ Fvar\ p]\}\ t$
$= \{k \to [Fvar\ s,\ Fvar\ p]\}\ \{n \to [Fvar\ s',\ Fvar\ p']\}\ t$
proof −
fix *u*
have
$(\forall\, n\ k\ s\ p\ s'\ p'.\ n \neq k$
$\longrightarrow \{n \to [Fvar\ s',\ Fvar\ p']\}\ \{k \to [Fvar\ s,\ Fvar\ p]\}\ t$
$= \{k \to [Fvar\ s,\ Fvar\ p]\}\ \{n \to [Fvar\ s',\ Fvar\ p']\}\ t)$
$\&(\forall\, n\ k\ s\ p\ s'\ p'.\ n \neq k$
$\longrightarrow sopen\text{-}option\ n\ (Fvar\ s')\ (Fvar\ p')\ (sopen\text{-}option\ k\ (Fvar\ s)\ (Fvar\ p)\ u)$
$= sopen\text{-}option\ k\ (Fvar\ s)\ (Fvar\ p)$
$(sopen\text{-}option\ n\ (Fvar\ s')\ (Fvar\ p')\ u))$
by (*rule compat-sterm-sterm-option.induct*, *simp-all split*: *bVariable.split*)
from *conjunct1*[*OF this*] **show** *?thesis* **by** *assumption*
qed

lemma *sopen-fresh-inj*[*rule-format*]:
$\forall\, n\ s\ p\ t'.\ \{n \to [Fvar\ s,\ Fvar\ p]\}\ t = \{n \to [Fvar\ s,\ Fvar\ p]\}\ t'$
$\longrightarrow s \notin FV\ t \longrightarrow s \notin FV\ t' \longrightarrow p \notin FV\ t \longrightarrow p \notin FV\ t' \longrightarrow s \neq p$
$\longrightarrow t = t'$
proof −
{
fix
b s p n t
assume
(*case b of Self i* ⇒ *if n* = *i then Fvar s else Bvar b*
| *Param i* ⇒ *if n* = *i then Fvar p else Bvar b*) = *t*
hence $Fvar\ s = t \vee Fvar\ p = t \vee Bvar\ b = t$
by (*cases b*, *auto*, (*rename-tac nat*, *case-tac n* = *nat*, *auto*)+)

```
}
note cT = this


fix u
have
  (∀ n s p t'. {n → [Fvar s, Fvar p]} t = {n → [Fvar s, Fvar p]} t'
    ⟶ s ∉ FV t ⟶ s ∉ FV t' ⟶ p ∉ FV t ⟶ p ∉ FV t' ⟶ s ≠ p
    ⟶ t = t')
  &(∀ n s p u'. sopen-option n (Fvar s) (Fvar p) u
              = sopen-option n (Fvar s) (Fvar p) u'
    ⟶ s ∉ FVoption u ⟶ s ∉ FVoption u'
    ⟶ p ∉ FVoption u ⟶ p ∉ FVoption u' ⟶ s ≠ p
    ⟶ u = u')
proof (induct - t - u rule: compat-sterm-sterm-option.induct)
  case (Bvar b) thus ?case
  proof (auto)
    fix s p n t
    assume
      a: (case b of Self i ⇒ if n = i then Fvar s else Bvar b
            | Param i ⇒ if n = i then Fvar p else Bvar b)
       = {n → [Fvar s,Fvar p]} t
    note cT[OF this]
    moreover assume s ∉ FV t and p ∉ FV t and s ≠ p
    ultimately show Bvar b = t
    proof (auto)
      {
        fix b'
        assume
          (case b' of Self i ⇒ if n = i then Fvar s else Bvar b'
                | Param i ⇒ if n = i then Fvar p else Bvar b') = Fvar s
        with ‹s ≠ p› have b' = Self n
          by (cases b', auto, (rename-tac nat, case-tac n = nat, auto)+)
      }note fvS = this
      assume eq-s: Fvar s = {n → [Fvar s,Fvar p]} t
      with sym[OF this] ‹s ∉ FV t› ‹s ≠ p› fvS
      have t = Bvar (Self n) by (cases t, auto)
      moreover
      from a sym[OF eq-s] ‹s ≠ p› fvS[of b]
      have Self n = b by simp
      ultimately show Bvar b = t by simp
    next
      {
        fix b'
        assume
          (case b' of Self i ⇒ if n = i then Fvar s else Bvar b'
                | Param i ⇒ if n = i then Fvar p else Bvar b') = Fvar p
        with ‹s ≠ p› have b' = Param n
          by (cases b', auto, (rename-tac nat, case-tac n = nat, auto)+)
```

```
    }note fvP = this
    assume eq-p: Fvar p = {n → [Fvar s,Fvar p]} t
    with sym[OF this] ‹p ∉ FV t› ‹s ≠ p› fvP
    have t = Bvar (Param n) by (cases t, auto)
    moreover
    from a sym[OF eq-p] ‹s ≠ p› fvP[of b]
    have Param n = b by simp
    ultimately show Bvar b = t by simp
   next
    assume Bvar b = {n → [Fvar s, Fvar p]} t
    from sym[OF this] show {n → [Fvar s,Fvar p]} t = t
    proof (cases t, auto)
     fix b'
     assume
      (case b' of Self i ⇒ if n = i then Fvar s else Bvar b'
                | Param i ⇒ if n = i then Fvar p else Bvar b') = Bvar b
     from cT[OF this] have Bvar b = Bvar b' by simp
     thus b = b' by simp
    qed
   qed
  qed
next
  case (Fvar x) thus ?case
  proof (auto)
   fix n s p t
   assume
    a: Fvar x = {n → [Fvar s,Fvar p]} t and
    s ∉ FV ({n → [Fvar s,Fvar p]} t)
   hence s ≠ x by force
   moreover
   assume p ∉ FV ({n → [Fvar s,Fvar p]} t)
   with a have p ≠ x by force
   ultimately
   show {n → [Fvar s,Fvar p]} t = t
    using a
   proof (cases t, auto)
    fix b
    assume
     Fvar x = (case b of Self i ⇒ if n = i then Fvar s else Bvar b
                       | Param i ⇒ if n = i then Fvar p else Bvar b)
    from cT[OF sym[OF this]] ‹s ≠ x› ‹p ≠ x›
    have False by simp
    then show
     (case b of Self i ⇒ if n = i then Fvar s else Bvar b
              | Param i ⇒ if n = i then Fvar p else Bvar b) = Bvar b ..
   qed
  qed
next
  case (Obj f T) thus ?case
```

```
  proof (auto)
    fix n s p t
    assume
      Obj (λl. sopen-option (Suc n) (Fvar s) (Fvar p) (f l)) T
       = {n → [Fvar s,Fvar p]} t and
      ∀ l∈dom f. s ∉ FVoption (f l) and
      ∀ l∈dom f. p ∉ FVoption (f l) and
      s ∉ FV t and p ∉ FV t and s ≠ p
    thus Obj f T = t using Obj
    proof (cases t, auto)

      fix b
      assume
        Obj (λl. sopen-option (Suc n) (Fvar s) (Fvar p) (f l)) T
         = (case b of Self i ⇒ if n = i then Fvar s else Bvar b
                    | Param i ⇒ if n = i then Fvar p else Bvar b)
      from cT[OF sym[OF this]] show False by auto
    next

      fix f'
      assume
        nin-s: ∀ l∈dom f'. s ∉ FVoption (f' l) and
        nin-p: ∀ l∈dom f'. p ∉ FVoption (f' l) and
        ff':   (λl. sopen-option (Suc n) (Fvar s) (Fvar p) (f l))
                 = (λl. sopen-option (Suc n) (Fvar s) (Fvar p) (f' l))
      have ⋀l. f l = f' l
      proof −
        fix  l
        from ff'
        have
          sopen-option (Suc n) (Fvar s) (Fvar p) (f l)
           = sopen-option (Suc n) (Fvar s) (Fvar p) (f' l)
          by (rule cong, simp)
        moreover
        from
          ‹∀ l∈dom f. s ∉ FVoption (f l)›
          ‹∀ l∈dom f. p ∉ FVoption (f l)›
        have s ∉ FVoption (f l) and p ∉ FVoption (f l)
          by (case-tac l ∈ dom f, auto)+
        moreover
        from nin-s nin-p have s ∉ FVoption (f' l) and p ∉ FVoption (f' l)
          by (case-tac l ∈ dom f', auto)+
        ultimately show f l = f' l using Obj[of l] ‹s ≠ p›
          by simp
      qed
      thus f = f' by (rule ext)
    qed
  qed
next
```

```
  case (Call t1 l t2) thus ?case
  proof (auto)
    fix n s p t
    assume
      s ∉ FV t1 and s ∉ FV t2 and p ∉ FV t1 and p ∉ FV t2 and
      s ≠ p and s ∉ FV t and p ∉ FV t and
      Call ({n → [Fvar s,Fvar p]} t1) l ({n → [Fvar s,Fvar p]} t2)
       = {n → [Fvar s,Fvar p]} t
    thus Call t1 l t2 = t using Call
    proof (cases t, auto)

      fix b
      assume
        Call ({n → [Fvar s,Fvar p]} t1) l ({n → [Fvar s,Fvar p]} t2)
         = (case b of Self i ⇒ if n = i then Fvar s else Bvar b
                    | Param i ⇒ if n = i then Fvar p else Bvar b)
      from cT[OF sym[OF this]] show False by auto
    qed
  qed
next
  case (Upd t1 l t2) thus ?case
  proof (auto)
    fix n s p t
    assume
      s ∉ FV t1 and s ∉ FV t2 and p ∉ FV t1 and p ∉ FV t2 and
      s ≠ p and s ∉ FV t and p ∉ FV t and
      Upd ({n → [Fvar s,Fvar p]} t1) l ({Suc n → [Fvar s,Fvar p]} t2)
       = {n → [Fvar s,Fvar p]} t
    thus Upd t1 l t2 = t using Upd
    proof (cases t, auto)

      fix b
      assume
        Upd ({n → [Fvar s,Fvar p]} t1) l ({Suc n → [Fvar s,Fvar p]} t2)
         = (case b of Self i ⇒ if n = i then Fvar s else Bvar b
                    | Param i ⇒ if n = i then Fvar p else Bvar b)
      from cT[OF sym[OF this]] show False by auto
    qed
  qed
next
  case None-sterm thus ?case
  proof (auto)
    fix u s p n
    assume None = sopen-option n (Fvar s) (Fvar p) u
    thus None = u by (cases u, auto)
  qed
next
  case (Some-sterm t) thus ?case
  proof (auto)
```

fix $u\ s\ p\ n$
assume
$Some\ (\{n \to [Fvar\ s,Fvar\ p]\}\ t) = sopen\text{-}option\ n\ (Fvar\ s)\ (Fvar\ p)\ u$ **and**
$s \notin FV\ t$ **and** $p \notin FV\ t$ **and** $s \neq p$ **and**
$s \notin FVoption\ u$ **and** $p \notin FVoption\ u$
with *Some-sterm* **show** $Some\ t = u$ **by** $(cases\ u)\ auto$
qed
qed
from *conjunct1*[*OF this*] **show** *?thesis* **by** *assumption*
qed

3.2.4 Variable closing

primrec
$sclose :: [nat, fVariable, fVariable, sterm] \Rightarrow sterm$
$(\langle\{\text{-} \leftarrow [\text{-},\text{-}]\} \text{-}\rangle [0, 0, 0, 300]\ 300)$
and
$sclose\text{-}option :: [nat, fVariable, fVariable, sterm\ option] \Rightarrow sterm\ option$
where
$sclose\text{-}Bvar: \{k \leftarrow [s,p]\}(Bvar\ b) = Bvar\ b$
| $sclose\text{-}Fvar:$
$\{k \leftarrow [s,p]\}(Fvar\ x) = (if\ x = s\ then\ (Bvar\ (Self\ k))$
$else\ (if\ x = p\ then\ (Bvar\ (Param\ k))$
$else\ (Fvar\ x)))$
| $sclose\text{-}Call: \{k \leftarrow [s,p]\}(Call\ t\ l\ a) = Call\ (\{k \leftarrow [s,p]\}t)\ l\ (\{k \leftarrow [s,p]\}a)$
| $sclose\text{-}Upd : \{k \leftarrow [s,p]\}(Upd\ t\ l\ u) = Upd\ (\{k \leftarrow [s,p]\}t)\ l\ (\{(Suc\ k) \leftarrow [s,p]\}u)$
| $sclose\text{-}Obj : \{k \leftarrow [s,p]\}(Obj\ f\ T) = Obj\ (\lambda l.\ sclose\text{-}option\ (Suc\ k)\ s\ p\ (f\ l))\ T$
| $sclose\text{-}None: sclose\text{-}option\ k\ s\ p\ None = None$
| $sclose\text{-}Some: sclose\text{-}option\ k\ s\ p\ (Some\ t) = Some\ (\{k \leftarrow [s,p]\}t)$

definition $closez :: [fVariable, fVariable, sterm] \Rightarrow sterm\ (\langle\sigma[\text{-},\text{-}]\ \text{-}\rangle [0, 0, 300])$
where
$\sigma[s,p]\ t = \{0 \leftarrow [s,p]\}t$

lemma $dom\text{-}scloseoption\text{-}lem[simp]: dom\ (\lambda l.\ sclose\text{-}option\ k\ s\ t\ (f\ l)) = dom\ f$
by $(auto, case\text{-}tac\ x \in dom\ f, auto)$

lemma *sclose-option-lem*:
$\forall\, l{\in}dom\ f.\ \{n \leftarrow [s,p]\}\ the(f\ l) = the\ (sclose\text{-}option\ n\ s\ p\ (f\ l))$
by *auto*

lemma *pred-scloseoption-lem*:
$(\forall\, l{\in}dom\ (\lambda l.\ sclose\text{-}option\ n\ s\ p\ (f\ l)).$
$(P{::}sterm \Rightarrow bool)\ (the\ (sclose\text{-}option\ n\ s\ p\ (f\ l)))) =$
$(\forall\, l{\in}dom\ f.\ (P{::}sterm \Rightarrow bool)\ (\{n \leftarrow [s,p]\}\ the\ (f\ l)))$
by $(simp, force)$

lemma $sclose\text{-}fresh[simp, rule\text{-}format]:$
$\forall\, n\ s\ p.\ s \notin FV\ t \longrightarrow p \notin FV\ t \longrightarrow \{n \leftarrow [s,p]\}\ t = t$

proof −
have
(∀ *n s p*. *s* ∉ *FV t* ⟶ *p* ∉ *FV t* ⟶ {*n* ← [*s*,*p*]} *t* = *t*)
&(∀ *n s p*. *s* ∉ *FVoption u* ⟶ *p* ∉ *FVoption u*
⟶ *sclose-option n s p u* = *u*)
proof (*induct - t - u rule*: *compat-sterm-sterm-option.induct*, *auto simp*: *bVariable.split*)

fix *f n s p*
assume
nin-s: ∀ *l*∈*dom f*. *s* ∉ *FVoption* (*f l*) **and**
nin-p: ∀ *l*∈*dom f*. *p* ∉ *FVoption* (*f l*)
{
fix *l* **from** *nin-s* **have** *s* ∉ *FVoption* (*f l*)
by (*case-tac l* ∈ *dom f*, *auto*)
}
moreover
{
fix *l* **from** *nin-p* **have** *p* ∉ *FVoption* (*f l*)
by (*case-tac l* ∈ *dom f*, *auto*)
}
moreover
assume
⋀*x*. ∀ *n s*. *s* ∉ *FVoption* (*f x*)
⟶ (∀ *p*. *p* ∉ *FVoption* (*f x*)
⟶ *sclose-option n s p* (*f x*) = *f x*)
ultimately
have ⋀*l*. *sclose-option* (*Suc n*) *s p* (*f l*) = *f l* **by** *auto*
with *ext* **show** (λ*l*. *sclose-option* (*Suc n*) *s p* (*f l*)) = *f* **by** *auto*
qed
from *conjunct1*[*OF this*] **show** *?thesis* **by** *assumption*
qed

lemma *sclose-FV*[*rule-format*]:
∀ *n s p*. *FV* ({*n* ← [*s*,*p*]} *t*) = *FV t* − {*s*} − {*p*}
proof −
have
(∀ *n s p*. *FV* ({*n* ← [*s*,*p*]} *t*) = *FV t* − {*s*} − {*p*})
&(∀ *n s p*. *FVoption* (*sclose-option n s p u*) = *FVoption u* − {*s*} − {*p*})
by (*rule compat-sterm-sterm-option.induct*, *simp-all split*: *bVariable.split*, *blast*+)
from *conjunct1*[*OF this*] **show** *?thesis* **by** *assumption*
qed

lemma *sclose-subset-FV*[*rule-format*]:
FV ({*n* ← [*s*,*p*]} *t*) ⊆ *FV t*
by (*simp add*: *sclose-FV*, *blast*)

lemma *Self-not-in-closed*[*simp*]: *sa* ∉ *FV* ({*n* ← [*sa*,*pa*]} *t*)
by (*simp add*: *sclose-FV*)

lemma *Param-not-in-closed*[*simp*]: *pa* ∉ *FV* ({*n* ← [*sa*,*pa*]} *t*)
by (*simp add*: *sclose-FV*)

3.2.5 Substitution

primrec
ssubst :: [*fVariable*, *sterm*, *sterm*] ⇒ *sterm*
(‹[- → -] -› [0, 0, 300] 300)
and
ssubst-option :: [*fVariable*, *sterm*, *sterm option*] ⇒ *sterm option*
where
ssubst-Bvar: [*z* → *u*](*Bvar v*) = *Bvar v*
| *ssubst-Fvar*: [*z* → *u*](*Fvar x*) = (*if* (*z* = *x*) *then u else* (*Fvar x*))
| *ssubst-Call*: [*z* → *u*](*Call t l s*) = *Call* ([*z* → *u*]*t*) *l* ([*z* → *u*]*s*)
| *ssubst-Upd* : [*z* → *u*](*Upd t l s*) = *Upd* ([*z* → *u*]*t*) *l* ([*z* → *u*]*s*)
| *ssubst-Obj* : [*z* → *u*](*Obj f T*) = *Obj* (λ*l*. *ssubst-option z u* (*f l*)) *T*
| *ssubst-None*: *ssubst-option z u None* = *None*
| *ssubst-Some*: *ssubst-option z u* (*Some t*) = *Some* ([*z* → *u*]*t*)

lemma *dom-ssubstoption-lem*[*simp*]: *dom* (λ*l*. *ssubst-option z u* (*f l*)) = *dom f*
by (*auto*, *case-tac x* ∈ *dom f*, *auto*)

lemma *ssubst-option-lem*:
∀ *l*∈*dom f*. [*z* → *u*] *the*(*f l*) = *the* (*ssubst-option z u* (*f l*))
by *auto*

lemma *pred-ssubstoption-lem*:
(∀ *l*∈*dom* (λ*l*. *ssubst-option x t* (*f l*)).
(*P*::*sterm* ⇒ *bool*) (*the* (*ssubst-option x t* (*f l*)))) =
(∀ *l*∈*dom f*. (*P*::*sterm* ⇒ *bool*) ([*x* → *t*] *the* (*f l*)))
by (*simp*, *force*)

lemma *ssubst-fresh*[*simp*, *rule-format*]:
∀ *s sa*. *sa* ∉ *FV t* ⟶ [*sa* → *s*] *t* = *t*
proof −
have
(∀ *s sa*. *sa* ∉ *FV t* ⟶ [*sa* → *s*] *t* = *t*)
&(∀ *s sa*. *sa* ∉ *FVoption u* ⟶ *ssubst-option sa s u* = *u*)
proof (*induct* - *t* - *u rule*: *compat-sterm-sterm-option.induct*, *auto*)
fix *s sa f*
assume
sa: ∀ *l*∈*dom f*. *sa* ∉ *FVoption* (*f l*) **and**
ssubst: ⋀*x*. ∀ *s sa*. *sa* ∉ *FVoption* (*f x*) ⟶ *ssubst-option sa s* (*f x*) = *f x*
{
fix *l* **from** *sa* **have** *sa* ∉ *FVoption* (*f l*)
by (*case-tac l* ∈ *dom f*, *auto*)
with *ssubst* **have** *ssubst-option sa s* (*f l*) = *f l* **by** *auto*
}

with *ext* **show** $(\lambda l.\ ssubst\text{-}option\ sa\ s\ (f\ l)) = f$ **by** *auto*
qed
from *conjunct1*[*OF this*] **show** *?thesis* **by** *assumption*
qed

lemma *ssubst-commute*[*rule-format*]:
$\forall\, s\ p\ sa\ pa.\ s \neq p \longrightarrow s \notin FV\ pa \longrightarrow p \notin FV\ sa$
$\longrightarrow [s \to sa]\ [p \to pa]\ t = [p \to pa]\ [s \to sa]\ t$
proof −
have
$(\forall\, s\ p\ sa\ pa.\ s \neq p \longrightarrow s \notin FV\ pa \longrightarrow p \notin FV\ sa$
$\longrightarrow [s \to sa]\ [p \to pa]\ t = [p \to pa]\ [s \to sa]\ t)$
$\&(\forall\, s\ p\ sa\ pa.\ s \neq p \longrightarrow s \notin FV\ pa \longrightarrow p \notin FV\ sa$
$\longrightarrow ssubst\text{-}option\ s\ sa\ (ssubst\text{-}option\ p\ pa\ u)$
$= ssubst\text{-}option\ p\ pa\ (ssubst\text{-}option\ s\ sa\ u))$
by (*rule compat-sterm-sterm-option.induct*, *simp-all split*: *bVariable.split*)
from *conjunct1*[*OF this*] **show** *?thesis* **by** *assumption*
qed

lemma *ssubst-FV*[*rule-format*]:
$\forall\, x\ s.\ FV\ ([x \to s]\ t) \subseteq FV\ s \cup (FV\ t - \{x\})$
proof −
have
$(\forall\, x\ s.\ FV\ ([x \to s]\ t) \subseteq FV\ s \cup (FV\ t - \{x\}))$
$\&(\forall\, x\ s.\ FVoption\ (ssubst\text{-}option\ x\ s\ u) \subseteq FV\ s \cup (FVoption\ u - \{x\}))$
by (*rule compat-sterm-sterm-option.induct*, *simp-all split*: *bVariable.split*, *blast*+)
from *conjunct1*[*OF this*] **show** *?thesis* **by** *assumption*
qed

lemma *ssubstoption-insert*:
$l \in dom\ f$
$\Longrightarrow (\lambda(la{::}Label).\ ssubst\text{-}option\ x\ t'\ (if\ la = l\ then\ Some\ t\ else\ f\ la))$
$= (\lambda(la{::}Label).\ ssubst\text{-}option\ x\ t'\ (f\ la))(l \mapsto [x \to t']\ t)$
by (*rule Ltake-eq-all*, *force*, *simp add*: *Ltake-eq-def*)

3.2.6 Local closure

inductive *lc* :: *sterm* ⇒ *bool*
where
lc-Fvar[*simp*, *intro*!]: *lc* (*Fvar x*)
| *lc-Call*[*simp*, *intro*!]: ⟦ *lc t*; *lc a* ⟧ ⟹ *lc* (*Call t l a*)
| *lc-Upd*[*simp*, *intro*!] :
⟦ *lc t*; *finite L*;
$\forall\, s\ p.\ s \notin L \wedge p \notin L \wedge s \neq p \longrightarrow lc\ (u^{[Fvar\ s,\ Fvar\ p]})$ ⟧
⟹ *lc* (*Upd t l u*)
| *lc-Obj*[*simp*, *intro*!] :
⟦ *finite L*; $\forall\, l{\in}dom\ f.$
$\forall\, s\ p.\ s \notin L \wedge p \notin L \wedge s \neq p \longrightarrow lc\ (the(f\ l)^{[Fvar\ s,\ Fvar\ p]})$ ⟧
⟹ *lc* (*Obj f T*)

definition *body* :: *sterm* ⇒ *bool* **where**
body t ⟷ (∃ *L*. *finite L* ∧ (∀ *s p*. *s* ∉ *L* ∧ *p* ∉ *L* ∧ *s* ≠ *p* ⟶ *lc* (*t*[*Fvar s*, *Fvar p*])))

lemma *lc-bvar*: *lc* (*Bvar b*) = *False*
by (*rule iffI*, *erule lc.cases*, *simp-all*)

lemma *lc-obj*:
lc (*Obj f T*) = (∀ *l*∈*dom f*. *body* (*the*(*f l*)))
proof
fix *f T* **assume** *lc* (*Obj f T*)
thus ∀ *l*∈*dom f*. *body* (*the*(*f l*))
unfolding *body-def*
by (*rule lc.cases*, *auto*)
next
fix *f* :: *Label* −~> *sterm* **and** *T* :: *type*
assume ∀ *l*∈*dom f*. *body* (*the* (*f l*))
hence
∃ *L*. *finite L* ∧ (∀ *l*∈*dom f*. ∀ *s p*. *s* ∉ *L* ∧ *p* ∉ *L* ∧ *s* ≠ *p*
⟶ *lc* (*the* (*f l*)[*Fvar s*, *Fvar p*]))
proof (*induct f rule*: *fmap-induct*)
case *empty* **thus** *?case* **by** *blast*
next
case (*insert F x y*) **thus** *?case*
proof −
assume *x* ∉ *dom F* **hence** ∀ *l*∈*dom F*. *the*(*F l*) = *the* ((*F*(*x* ↦ *y*)) *l*)
by *auto*
with ‹∀ *l*∈*dom* (*F*(*x* ↦ *y*)). *body* (*the*((*F*(*x* ↦ *y*)) *l*))›
have ∀ *l*∈*dom F*. *body* (*the* (*F l*)) **by** *force*
from *insert*(*2*)[*OF this*]
obtain *L* **where**
finite L **and**
∀ *l*∈*dom F*. ∀ *s p*. *s* ∉ *L* ∧ *p* ∉ *L* ∧ *s* ≠ *p*
⟶ *lc* (*the* (*F l*)[*Fvar s*, *Fvar p*]) **by** *auto*
moreover
from ‹∀ *l*∈*dom* (*F*(*x* ↦ *y*)). *body* (*the*((*F*(*x* ↦ *y*)) *l*))› **have** *body y* **by** *force*
then obtain *L*′ **where**
finite L′ **and**
∀ *s p*. *s* ∉ *L*′ ∧ *p* ∉ *L*′ ∧ *s* ≠ *p*
⟶ *lc* (*y*[*Fvar s*, *Fvar p*]) **by** (*auto simp*: *body-def*)

ultimately
show
∃ *L*. *finite L* ∧ (∀ *l*∈*dom* (*F*(*x* ↦ *y*)). ∀ *s p*. *s* ∉ *L* ∧ *p* ∉ *L* ∧ *s* ≠ *p*
⟶ *lc* (*the* ((*F*(*x* ↦ *y*)) *l*)[*Fvar s*,*Fvar p*]))
by (*rule-tac x* = *L* ∪ *L*′ **in** *exI*, *auto*)
qed
qed
thus *lc* (*Obj f T*) **by** *auto*

qed

lemma *lc-upd*: *lc* (*Upd t l s*) = (*lc t* ∧ *body s*)
by (*unfold body-def*, *rule iffI*, *erule lc.cases*, *auto*)

lemma *lc-call*: *lc* (*Call t l s*) = (*lc t* ∧ *lc s*)
by (*rule iffI*, *erule lc.cases*, *simp-all*)

lemma *lc-induct*[*consumes 1*, *case-names Fvar Call Upd Obj Bnd*]:
fixes *P1* :: *sterm* ⇒ *bool* **and** *P2* :: *sterm* ⇒ *bool*
assumes
lc t **and**
⋀*x*. *P1* (*Fvar x*) **and**
⋀*t l a*. ⟦ *lc t*; *P1 t*; *lc a*; *P1 a* ⟧ ⟹ *P1* (*Call t l a*) **and**
⋀*t l u*. ⟦ *lc t*; *P1 t*; *P2 u* ⟧ ⟹ *P1* (*Upd t l u*) **and**
⋀*f T*. ∀ *l*∈*dom f*. *P2* (*the*(*f l*)) ⟹ *P1* (*Obj f T*) **and**
⋀*L t*. ⟦ *finite L*;
∀ *s p*. *s* ∉ *L* ∧ *p* ∉ *L* ∧ *s* ≠ *p*
⟶ *lc* ($t^{[Fvar\ s,\ Fvar\ p]}$) ∧ *P1* ($t^{[Fvar\ s,\ Fvar\ p]}$) ⟧
⟹ *P2 t*
shows *P1 t*
using *assms* **by** (*induct rule*: *lc.induct*, *auto*)

3.2.7 Connections between sopen, sclose, ssubst, lc and body and resulting properties

lemma *ssubst-intro*[*rule-format*]:
∀ *n s p sa pa*. *sa* ∉ *FV t* ⟶ *pa* ∉ *FV t* ⟶ *sa* ≠ *pa*
⟶ *sa* ∉ *FV p*
⟶ {*n* → [*s*,*p*]} *t* = [*sa* → *s*] [*pa* → *p*] {*n* → [*Fvar sa*, *Fvar pa*]} *t*
proof −
have
(∀ *n s p sa pa*. *sa* ∉ *FV t* ⟶ *pa* ∉ *FV t* ⟶ *sa* ≠ *pa*
⟶ *sa* ∉ *FV p*
⟶ {*n* → [*s*,*p*]} *t* = [*sa* → *s*] [*pa* → *p*] {*n* → [*Fvar sa*, *Fvar pa*]} *t*)
&(∀ *n s p sa pa*. *sa* ∉ *FVoption u* ⟶ *pa* ∉ *FVoption u* ⟶ *sa* ≠ *pa*
⟶ *sa* ∉ *FV p*
⟶ *sopen-option n s p u*
= *ssubst-option sa s* (*ssubst-option pa p*
(*sopen-option n* (*Fvar sa*) (*Fvar pa*) *u*)))
proof (*induct - t - u rule*: *compat-sterm-sterm-option.induct*)
case *Bvar* **thus** *?case* **by** (*simp split*: *bVariable.split*)
next
case *Fvar* **thus** *?case* **by** *simp*
next
case *Upd* **thus** *?case* **by** *simp*
next
case *Call* **thus** *?case* **by** *simp*
next

```
  case None-sterm thus ?case by simp
next
  case (Obj f T) thus ?case
  proof (clarify)
    fix n s p sa pa
    assume sa ∉ FV (Obj f T) and pa ∉ FV (Obj f T)
    {
      fix l
      from ‹sa ∉ FV (Obj f T)› have sa ∉ FVoption (f l)
        by (case-tac l ∈ dom f, auto)
    }
    moreover
    {
      fix l
      from ‹pa ∉ FV (Obj f T)› have pa: pa ∉ FVoption (f l)
        by (case-tac l ∈ dom f, auto)
    }
    moreover assume sa ≠ pa and sa ∉ FV p
    ultimately
    have
      ⋀l. sopen-option (Suc n) s p (f l)
            = ssubst-option sa s (ssubst-option pa p
              (sopen-option (Suc n) (Fvar sa) (Fvar pa) (f l)))
      using Obj by auto
    with ext
    show
      {n → [s,p]} Obj f T
       = [sa → s] [pa → p] {n → [Fvar sa,Fvar pa]} Obj f T
      by auto
  qed
next
  case (Some-sterm t) thus ?case
  proof (clarify)
    fix n s p sa pa
    assume sa ∉ FVoption (Some t)
    hence sa ∉ FV t by simp
    moreover assume pa ∉ FVoption (Some t)
    hence pa ∉ FV t by simp
    moreover assume sa ≠ pa and sa ∉ FV p
    ultimately
    have {n → [s,p]} t = [sa → s] [pa → p] {n → [Fvar sa,Fvar pa]} t
      using Some-sterm by blast
    thus
      sopen-option n s p (Some t)
       = ssubst-option sa s (ssubst-option pa p
         (sopen-option n (Fvar sa) (Fvar pa) (Some t)))
      by simp
  qed
qed
```

```
  from conjunct1[OF this] show ?thesis by assumption
qed

lemma sopen-lc-FV[rule-format]:
  fixes t
  assumes lc t
  shows ∀ n s p. {n → [Fvar s, Fvar p]} t = t
  using assms
proof
  (induct
    taking: λt. ∀ n s p. {Suc n → [Fvar s, Fvar p]} t = t
    rule: lc-induct)
  case Fvar thus ?case by simp
next
  case Call thus ?case by simp
next
  case Upd thus ?case by simp
next
  case (Obj f T) note pred = this
  show ?case
  proof (intro strip, simp)
    fix n s p
    {
      fix l
      have sopen-option (Suc n) (Fvar s) (Fvar p) (f l) = f l
      proof (cases l ∈ dom f)
        case False hence f l = None by force
        thus ?thesis by force
      next
        case True with pred show ?thesis by force
      qed
    } with ext
    show (λl. sopen-option (Suc n) (Fvar s) (Fvar p) (f l)) = f
      by auto
  qed
next
  case (Bnd L t) note cof = this(2)
  show ?case
  proof (intro strip)
    fix n s p
    from ‹finite L› exFresh-s-p-cof[of L ∪ FV t ∪ {s} ∪ {p}]
    obtain sa pa where
      sapa: sa ∉ L ∪ FV t ∪ {s} ∪ {p} ∧ pa ∉ L ∪ FV t ∪ {s} ∪ {p} ∧ sa ≠ pa
      by auto
    with cof
    have {Suc n → [Fvar s,Fvar p]} (t[Fvar sa, Fvar pa]) = (t[Fvar sa, Fvar pa])
      by auto
    with sopen-commute[OF Suc-not-Zero[of n]]
    have
```

```
    eq: {0 → [Fvar sa,Fvar pa]} {Suc n → [Fvar s,Fvar p]} t
        = {0 → [Fvar sa,Fvar pa]} t
    by (simp add: openz-def)
  from sapa contra-subsetD[OF sopen-FV[of - Fvar s Fvar p t]]
  have
    sa ∉ FV ({Suc n → [Fvar s,Fvar p]} t) and sa ∉ FV t and
    pa ∉ FV ({Suc n → [Fvar s,Fvar p]} t) and pa ∉ FV t and
    sa ≠ pa
    by auto
  from sopen-fresh-inj[OF eq this]
  show {Suc n → [Fvar s,Fvar p]} t = t by assumption
 qed
qed

lemma sopen-lc[simp]:
 fixes t n s p
 assumes lc t
 shows {n → [s,p]} t = t
proof −
 from exFresh-s-p-cof[of FV t ∪ FV p]
 obtain sa pa where
   sa ∉ FV t and pa ∉ FV t and sa ≠ pa and
   sa ∉ FV p and pa ∉ FV p
   by auto
 from ssubst-intro[OF this(1−4)]
 have {n → [s,p]} t = [sa → s] [pa → p] {n → [Fvar sa,Fvar pa]} t
   by simp
 with assms have {n → [s,p]} t = [sa → s] [pa → p] t
   using sopen-lc-FV
   by simp
 with ssubst-fresh[OF ‹pa ∉ FV t›]
 have {n → [s,p]} t = [sa → s] t by simp
 with ssubst-fresh[OF ‹sa ∉ FV t›]
 show {n → [s,p]} t = t by simp
qed

lemma sopen-twice[rule-format]:
 ∀ s p s′ p′ n. lc s ⟶ lc p
    ⟶ {n → [s′,p′]} {n → [s,p]} t = {n → [s,p]} t
proof −
 have
   (∀ s p s′ p′ n. lc s ⟶ lc p
     ⟶ {n → [s′,p′]} {n → [s,p]} t = {n → [s,p]} t)
   &(∀ s p s′ p′ n. lc s ⟶ lc p
     ⟶ sopen-option n s′ p′ (sopen-option n s p u) = sopen-option n s p u)
   by (rule compat-sterm-sterm-option.induct, auto simp: bVariable.split)
 from conjunct1[OF this] show ?thesis by assumption
qed
```

lemma *sopen-sclose-commute*[*rule-format*]:
 $\forall n\ k\ s\ p\ sa\ pa.\ n \neq k \longrightarrow sa \notin FV\ s \longrightarrow sa \notin FV\ p$
  $\longrightarrow pa \notin FV\ s \longrightarrow pa \notin FV\ p$
  $\longrightarrow \{n \rightarrow [s,\ p]\}\ \{k \leftarrow [sa,pa]\}\ t = \{k \leftarrow [sa,pa]\}\ \{n \rightarrow [s,\ p]\}\ t$
proof −
 have
  $(\forall n\ k\ s\ p\ sa\ pa.\ n \neq k \longrightarrow sa \notin FV\ s \longrightarrow sa \notin FV\ p$
   $\longrightarrow pa \notin FV\ s \longrightarrow pa \notin FV\ p$
   $\longrightarrow \{n \rightarrow [s,\ p]\}\ \{k \leftarrow [sa,pa]\}\ t = \{k \leftarrow [sa,pa]\}\ \{n \rightarrow [s,\ p]\}\ t)$
  $\&(\forall n\ k\ s\ p\ sa\ pa.\ n \neq k \longrightarrow sa \notin FV\ s \longrightarrow sa \notin FV\ p$
   $\longrightarrow pa \notin FV\ s \longrightarrow pa \notin FV\ p$
   $\longrightarrow$ *sopen-option* $n\ s\ p$ (*sclose-option* $k\ sa\ pa\ u$)
    $=$ *sclose-option* $k\ sa\ pa$ (*sopen-option* $n\ s\ p\ u$))
  by (*rule compat-sterm-sterm-option.induct*, *simp-all split*: *bVariable.split*)
 from *conjunct1*[*OF this*] **show** *?thesis* **by** *assumption*
qed

lemma *sclose-sopen-eq-t*[*rule-format*]:
 $\forall n\ s\ p.\ s \notin FV\ t \longrightarrow p \notin FV\ t \longrightarrow s \neq p$
  $\longrightarrow \{n \leftarrow [s,p]\}\ \{n \rightarrow [Fvar\ s,\ Fvar\ p]\}\ t = t$
proof −
 have
  $(\forall n\ s\ p.\ s \notin FV\ t \longrightarrow p \notin FV\ t \longrightarrow s \neq p$
   $\longrightarrow \{n \leftarrow [s,p]\}\ \{n \rightarrow [Fvar\ s,\ Fvar\ p]\}\ t = t)$
  $\&(\forall n\ s\ p.\ s \notin FVoption\ u \longrightarrow p \notin FVoption\ u \longrightarrow s \neq p$
   $\longrightarrow$ *sclose-option* $n\ s\ p$ (*sopen-option* n (*Fvar* s) (*Fvar* p) u) $= u$)
 proof (*induct - t - u rule*: *compat-sterm-sterm-option.induct*, *simp-all split*: *bVariable.split*, *auto*)

  fix $f\ n\ s\ p$
  assume
   nin-s: $\forall l{\in}dom\ f.\ s \notin FVoption\ (f\ l)$ **and**
   nin-p: $\forall l{\in}dom\ f.\ p \notin FVoption\ (f\ l)$
  {
   fix l **from** *nin-s* **have** $s \notin FVoption\ (f\ l)$
    by (*case-tac* $l \in dom\ f$, *auto*)
  }
  moreover
  {
   fix l **from** *nin-p* **have** $p \notin FVoption\ (f\ l)$
    by (*case-tac* $l \in dom\ f$, *auto*)
  }
  moreover
  assume
   $s \neq p$ **and**
   $\bigwedge x.\ \forall n\ s.\ s \notin FVoption\ (f\ x)$
    $\longrightarrow (\forall p.\ p \notin FVoption\ (f\ x) \longrightarrow s \neq p$
     $\longrightarrow$ *sclose-option* $n\ s\ p$ (*sopen-option* n (*Fvar* s) (*Fvar* p) ($f\ x$))
      $= f\ x$)

```
    ultimately
    have ⋀l. sclose-option n s p (sopen-option n (Fvar s) (Fvar p) (f l)) = f l
      by auto
    with ext
    show (λl. sclose-option n s p (sopen-option n (Fvar s) (Fvar p) (f l))) = f
      by auto
  qed
  from conjunct1[OF this] show ?thesis by assumption
qed

lemma sopen-sclose-eq-t[simp, rule-format]:
  fixes t
  assumes lc t
  shows ∀ n s p. {n → [Fvar s, Fvar p]} {n ← [s,p]} t = t
  using assms
proof
  (induct
    taking: λt. ∀ n s p. {Suc n → [Fvar s, Fvar p]} {Suc n ← [s,p]} t = t
    rule: lc-induct)
  case Fvar thus ?case by simp
next
  case Call thus ?case by simp
next
  case Upd thus ?case by simp
next
  case (Obj f T) note pred = this
  show ?case
  proof (intro strip, simp)
    fix n s p
    {
      fix l
      have
        sopen-option (Suc n) (Fvar s) (Fvar p) (sclose-option (Suc n) s p (f l))
         = f l
      proof (cases l ∈ dom f)
        case False hence f l = None by force
        thus ?thesis by simp
      next
        case True with pred
        show ?thesis by force
      qed
    }
    with ext
    show (λl. sopen-option (Suc n) (Fvar s) (Fvar p)
                (sclose-option (Suc n) s p (f l))) = f
      by simp
  qed
next
  case (Bnd L t) note cof = this(2)
```

```
  show ?case
  proof (intro strip)
    fix n s p
    from ‹finite L› exFresh-s-p-cof[of L ∪ FV t ∪ {s} ∪ {p}]
    obtain sa pa where
      sapa: sa ∉ L ∪ FV t ∪ {s} ∪ {p} ∧ pa ∉ L ∪ FV t ∪ {s} ∪ {p}
            ∧ sa ≠ pa
      by auto
    with cof
    have
      eq: {Suc n → [Fvar s,Fvar p]} {Suc n ← [s,p]} (t^[Fvar sa, Fvar pa])
          = (t^[Fvar sa, Fvar pa]) by blast

    {
      fix x assume x ∉ FV t
      from contra-subsetD[OF sclose-subset-FV this]
      have x ∉ FV ({Suc n ← [s,p]} t) by simp
      moreover assume x ≠ p and x ≠ s
      ultimately
      have x ∉ FV ({Suc n ← [s,p]} t) ∪ FV (Fvar s) ∪ FV (Fvar p)
        by simp
      from contra-subsetD[OF sopen-FV this]
      have x ∉ FV ({Suc n → [Fvar s,Fvar p]} {Suc n ← [s,p]} t)
        by simp
    } with sapa
    have
      s ∉ FV (Fvar sa) and s ∉ FV (Fvar pa) and
      p ∉ FV (Fvar sa) and p ∉ FV (Fvar pa) and
      sa ∉ FV ({Suc n → [Fvar s,Fvar p]} {Suc n ← [s,p]} t) and
      sa ∉ FV t and
      pa ∉ FV ({Suc n → [Fvar s,Fvar p]} {Suc n ← [s,p]} t) and
      pa ∉ FV t and sa ≠ pa
      by auto

    from
      eq
      sym[OF sopen-sclose-commute[OF not-sym[OF Suc-not-Zero[of n]]
                                                this(1−4)]]
      sopen-commute[OF Suc-not-Zero[of n]]
      sopen-fresh-inj[OF - this(5−9)]
    show {Suc n → [Fvar s,Fvar p]} {Suc n ← [s,p]} t = t
      by (auto simp: openz-def)
  qed
qed

lemma ssubst-sopen-distrib[rule-format]:
  ∀ n s p t'. lc t' ⟶ [x → t'] {n → [s,p]} t
   = {n → [[x → t']s, [x → t']p]} [x → t'] t
proof −
```

have
(∀ *n s p t′*. *lc t′*
⟶ [*x* → *t′*] {*n* → [*s*,*p*]} *t* = {*n* → [[*x* → *t′*]*s*, [*x* → *t′*]*p*]} [*x* → *t′*] *t*)
&(∀ *n s p t′*. *lc t′*
⟶ *ssubst-option x t′* (*sopen-option n s p u*)
= *sopen-option n* ([*x* → *t′*]*s*) ([*x* → *t′*]*p*) (*ssubst-option x t′ u*))
by (*rule compat-sterm-sterm-option.induct*, *simp-all split*: *bVariable.split*)
from *conjunct1*[*OF this*] **show** *?thesis* **by** *assumption*
qed

lemma *ssubst-openz-distrib*:
lc t′ ⟹ [*x* → *t′*] (*t*[*s*,*p*]) = (([*x* → *t′*] *t*)[[*x* → *t′*] *s*, [*x* → *t′*] *p*])
by (*simp add*: *openz-def ssubst-sopen-distrib*)

lemma *ssubst-sopen-commute*: ⟦ *lc t′*; *x* ∉ *FV s*; *x* ∉ *FV p* ⟧
⟹ [*x* → *t′*] {*n* → [*s*,*p*]} *t* = {*n* → [*s*,*p*]} [*x* → *t′*] *t*
by (*frule ssubst-sopen-distrib*[*of t′ x n s p t*], *simp*)

lemma *sopen-commute-gen*:
fixes *s p s′ p′ n k t*
assumes *lc s* **and** *lc p* **and** *lc s′* **and** *lc p′* **and** *n* ≠ *k*
shows {*n* → [*s*,*p*]} {*k* → [*s′*,*p′*]} *t* = {*k* → [*s′*,*p′*]} {*n* → [*s*,*p*]} *t*
proof −
have *finite* (*FV s* ∪ *FV p* ∪ *FV s′* ∪ *FV p′* ∪ *FV t*) **by** *auto*
from *exFresh-s-p-cof*[*OF this*]
obtain *sa pa* **where**
sa ∉ *FV s* ∪ *FV p* ∪ *FV s′* ∪ *FV p′* ∪ *FV t*
∧ *pa* ∉ *FV s* ∪ *FV p* ∪ *FV s′* ∪ *FV p′* ∪ *FV t* ∧ *sa* ≠ *pa* **by** *auto*
moreover
hence *finite* (*FV s* ∪ *FV p* ∪ *FV s′* ∪ *FV p′* ∪ *FV t* ∪ {*sa*} ∪ {*pa*}) **by** *auto*
from *exFresh-s-p-cof*[*OF this*]
obtain *sb pb* **where**
sb ∉ *FV s* ∪ *FV p* ∪ *FV s′* ∪ *FV p′* ∪ *FV t* ∪ {*sa*} ∪ {*pa*}
∧ *pb* ∉ *FV s* ∪ *FV p* ∪ *FV s′* ∪ *FV p′* ∪ *FV t* ∪ {*sa*} ∪ {*pa*}
∧ *sb* ≠ *pb* **by** *auto*
ultimately
have
sa ∉ *FV t* **and** *pa* ∉ *FV t* **and** *sb* ∉ *FV t* **and** *pb* ∉ *FV t* **and**
sa ∉ *FV* ({*n* → [*s*,*p*]} *t*) **and** *pa* ∉ *FV* ({*n* → [*s*,*p*]} *t*) **and**
sb ∉ *FV* ({*k* → [*s′*,*p′*]} *t*) **and** *pb* ∉ *FV* ({*k* → [*s′*,*p′*]} *t*) **and**

sa ≠ *pa* **and** *sb* ≠ *pb* **and** *sb* ≠ *sa* **and** *sb* ≠ *pa* **and**
pb ≠ *sa* **and** *pb* ≠ *pa* **and**

sa ∉ *FV s* **and** *sa* ∉ *FV p* **and** *pa* ∉ *FV s* **and** *pa* ∉ *FV p* **and**
sb ∉ *FV s′* **and** *sb* ∉ *FV p′* **and** *pb* ∉ *FV s′* **and** *pb* ∉ *FV p′* **and**
sa ∉ *FV p′* **and** *sb* ∉ *FV p* **and**

sa ∉ *FV* (*Fvar sb*) **and** *sa* ∉ *FV* (*Fvar pb*) **and**

$pa \notin FV\ (Fvar\ sb)$ **and** $pa \notin FV\ (Fvar\ pb)$ **and**
$pb \notin FV\ (Fvar\ sa)$ **and** $pb \notin FV\ (Fvar\ pa)$ **and**
$sb \notin FV\ (Fvar\ sa)$ **and** $sb \notin FV\ (Fvar\ pa)$ **and**

$lc\ s$ **and** $lc\ p$ **and** $lc\ s'$ **and** $lc\ p'$

using *contra-subsetD*[*OF sopen-FV*] *assms*(*1*−*4*)
by *auto*

from
ssubst-intro[*OF* ‹$sa \notin FV\ t$› ‹$pa \notin FV\ t$› ‹$sa \neq pa$› ‹$sa \notin FV\ p'$›]
ssubst-intro[*OF* ‹$sb \notin FV\ (\{k \to [s',p']\}\ t)$› ‹$pb \notin FV\ (\{k \to [s',p']\}\ t)$›
‹$sb \neq pb$› ‹$sb \notin FV\ p$›]
sym[*OF ssubst-sopen-commute*[*OF* ‹$lc\ s'$›
‹$sa \notin FV\ (Fvar\ sb)$› ‹$sa \notin FV\ (Fvar\ pb)$›]]
sym[*OF ssubst-sopen-commute*[*OF* ‹$lc\ p'$›
‹$pa \notin FV\ (Fvar\ sb)$› ‹$pa \notin FV\ (Fvar\ pb)$›]]
sopen-commute[*OF* ‹$n \neq k$›]
ssubst-commute[*OF* ‹$pb \neq sa$› ‹$pb \notin FV\ s'$› ‹$sa \notin FV\ p$›]
ssubst-commute[*OF* ‹$sb \neq sa$› ‹$sb \notin FV\ s'$› ‹$sa \notin FV\ s$›]
ssubst-commute[*OF* ‹$pb \neq pa$› ‹$pb \notin FV\ p'$› ‹$pa \notin FV\ p$›]
ssubst-commute[*OF* ‹$sb \neq pa$› ‹$sb \notin FV\ p'$› ‹$pa \notin FV\ s$›]
ssubst-sopen-commute[*OF* ‹$lc\ s$› ‹$sb \notin FV\ (Fvar\ sa)$› ‹$sb \notin FV\ (Fvar\ pa)$›]
ssubst-sopen-commute[*OF* ‹$lc\ p$› ‹$pb \notin FV\ (Fvar\ sa)$› ‹$pb \notin FV\ (Fvar\ pa)$›]
sym[*OF ssubst-intro*[*OF* ‹$sb \notin FV\ t$› ‹$pb \notin FV\ t$› ‹$sb \neq pb$› ‹$sb \notin FV\ p$›]]
sym[*OF ssubst-intro*[*OF* ‹$sa \notin FV\ (\{n \to [s,p]\}\ t)$› ‹$pa \notin FV\ (\{n \to [s,p]\}\ t)$›
‹$sa \neq pa$› ‹$sa \notin FV\ p'$›]]
show $\{n \to [s,p]\}\ \{k \to [s',p']\}\ t = \{k \to [s',p']\}\ \{n \to [s,p]\}\ t$
by *force*
qed

lemma *ssubst-preserves-lc*[*simp*, *rule-format*]:
fixes t
assumes $lc\ t$
shows $\forall x\ t'.\ lc\ t' \longrightarrow lc\ ([x \to t']\ t)$
proof −
define *pred-cof*
where $pred\text{-}cof\ L\ t \longleftrightarrow (\forall s\ p.\ s \notin L \wedge p \notin L \wedge s \neq p \longrightarrow lc\ (t^{[Fvar\ s,\ Fvar\ p]}))$
for $L\ t$

{
fix $x\ v\ t$
assume
$lc\ v$ **and**
$\forall x\ v.\ lc\ v \longrightarrow (\exists L.\ finite\ L \wedge pred\text{-}cof\ L\ ([x \to v]\ t))$
hence
$\exists L.\ finite\ L \wedge pred\text{-}cof\ L\ ([x \to v]\ t)$
by *auto*

}note *Lex* = *this*

from *assms* **show** *?thesis*
proof
(*induct*
taking: $\lambda t.\ \forall x\ t'.\ lc\ t' \longrightarrow (\exists L.\ finite\ L \wedge pred\text{-}cof\ L\ ([x \rightarrow t']\ t))$
rule: *lc-induct*)
case *Fvar* **thus** *?case* **by** *simp*
next
case *Call* **thus** *?case* **by** *simp*
next
case (*Upd t l u*) **note** *pred-t* = *this*(*2*) **and** *pred-bnd* = *this*(*3*)
show *?case*
proof (*intro strip*)
fix *x t'* **assume** *lc t'*
note *Lex*[*OF this pred-bnd*]
from *this*[*of x*]
obtain *L* **where** *finite L* **and** *pred-cof L* ($[x \rightarrow t']$ *u*)
by *auto*
with ‹*lc t'*› *pred-t* **show** *lc* ($[x \rightarrow t']$ *Upd t l u*)
unfolding *pred-cof-def*
by *simp*
qed
next
case (*Obj f T*) **note** *pred* = *this*
show *?case*
proof (*intro strip*)
fix *x t'* **assume** *lc t'*
define *pred-fl* **where** *pred-fl s p b l* = *lc* ($[x \rightarrow t']$ *the* $b^{[Fvar\ s,\ Fvar\ p]}$)
for *s p b* **and** *l*::*Label*

from ‹*lc t'*› *fmap-ball-all2*[*OF pred*]
have $\forall\, l{\in}dom\ f.\ \exists L.\ finite\ L \wedge pred\text{-}cof\ L\ ([x \rightarrow t']\ the(f\ l))$
unfolding *pred-cof-def*
by *simp*
with *fmap-ex-cof*[*of f pred-fl*]
obtain *L* **where**
$finite\ L$ **and** $\forall\, l{\in}dom\ f.\ pred\text{-}cof\ L\ ([x \rightarrow t']\ the(f\ l))$
unfolding *pred-cof-def pred-fl-def*
by *auto*
with *pred-ssubstoption-lem*[*of x t' f pred-cof L*]
show *lc* ($[x \rightarrow t']$ *Obj f T*)
unfolding *pred-cof-def*
by *simp*
qed
next
case (*Bnd L t*) **note** *pred* = *this*(*2*)
show *?case*
proof (*intro strip*)

```
      fix x t' assume lc t'
      with ‹finite L› show ∃ L. finite L ∧ pred-cof L ([x → t'] t)
        unfolding pred-cof-def
      proof (
          rule-tac x = L ∪ {x} in exI,
          intro conjI, simp, intro strip)
        fix s p assume sp: s ∉ L ∪ {x} ∧ p ∉ L ∪ {x} ∧ s ≠ p
        hence x ∉ FV (Fvar s) and x ∉ FV (Fvar p)
          by auto
        from sp pred ‹lc t'›
        have lc ([x → t'] (t^[Fvar s,Fvar p]))
          by blast
        with ssubst-sopen-commute[OF ‹lc t'› ‹x ∉ FV (Fvar s)›
                                     ‹x ∉ FV (Fvar p)›]
        show lc ([x → t'] t^[Fvar s,Fvar p])
          by (auto simp: openz-def)
      qed
    qed
  qed
qed

lemma sopen-sclose-eq-ssubst: ⟦ sa ≠ pa; sa ∉ FV p; lc t ⟧
  ⟹ {n → [s,p]} {n ← [sa,pa]} t = [sa → s] [pa → p] t
  by (rule-tac sa1 = sa and pa1 = pa and t1 = {n ← [sa,pa]} t
    in ssubst[OF ssubst-intro], simp+)

lemma ssubst-sclose-commute[rule-format]:
  ∀ x n s p t'. s ∉ FV t' ⟶ p ∉ FV t' ⟶ x ≠ s ⟶ x ≠ p
    ⟶ [x → t'] {n ← [s,p]} t = {n ← [s,p]} [x → t'] t
proof −
  have
    (∀ x n s p t'. s ∉ FV t' ⟶ p ∉ FV t' ⟶ x ≠ s ⟶ x ≠ p
      ⟶ [x → t'] {n ← [s,p]} t = {n ← [s,p]} [x → t'] t)
    &(∀ x n s p t'. s ∉ FV t' ⟶ p ∉ FV t' ⟶ x ≠ s ⟶ x ≠ p
      ⟶ ssubst-option x t' (sclose-option n s p u)
        = sclose-option n s p (ssubst-option x t' u))
    by (rule compat-sterm-sterm-option.induct, simp-all split: bVariable.split)
  from conjunct1[OF this] show ?thesis by assumption
qed

lemma body-lc-FV:
  fixes t s p
  assumes body t
  shows lc (t^[Fvar s, Fvar p])
proof −
  from assms
  obtain L where
    finite L and pred-sp: ∀ s p. s ∉ L ∧ p ∉ L ∧ s ≠ p ⟶ lc (t^[Fvar s,Fvar p])
    unfolding body-def by auto
```

```
  hence finite (L ∪ FV t ∪ {s} ∪ {p}) by simp
  from exFresh-s-p-cof[OF this] obtain sa pa where sapa:
    sa ∉ L ∪ FV t ∪ {s} ∪ {p} ∧ pa ∉ L ∪ FV t ∪ {s} ∪ {p} ∧ sa ≠ pa
    by auto
  hence sa ∉ FV t and pa ∉ FV t and sa ≠ pa and sa ∉ FV (Fvar p) by auto
  from pred-sp sapa have lc (t^[Fvar sa,Fvar pa]) by blast

  with
    ssubst-intro[OF ‹sa ∉ FV t› ‹pa ∉ FV t› ‹sa ≠ pa› ‹sa ∉ FV (Fvar p)›]
    ssubst-preserves-lc
  show lc (t^[Fvar s,Fvar p]) by (auto simp: openz-def)
qed

lemma body-lc:
  fixes t s p
  assumes body t and lc s and lc p
  shows lc (t^[s, p])
proof −
  have finite (FV t ∪ FV p) by simp
  from exFresh-s-p-cof[OF this] obtain sa pa where
    sa ∉ FV t ∪ FV p ∧ pa ∉ FV t ∪ FV p ∧ sa ≠ pa by auto
  hence sa ∉ FV t and pa ∉ FV t and sa ≠ pa and sa ∉ FV p
    by auto

  from body-lc-FV[OF ‹body t›] have lc: lc (t^[Fvar sa,Fvar pa])
    by assumption

  from
    ssubst-intro[OF ‹sa ∉ FV t› ‹pa ∉ FV t› ‹sa ≠ pa› ‹sa ∉ FV p›]
    ssubst-preserves-lc[OF lc] ‹lc s› ‹lc p›
  show lc (t^[s,p]) by (auto simp: openz-def)
qed

lemma lc-body:
  fixes t s p
  assumes lc t and s ≠ p
  shows body (σ[s,p] t)
  unfolding body-def
proof
  have
    ∀ sa pa. sa ∉ FV t ∪ {s} ∪ {p} ∧ pa ∉ FV t ∪ {s} ∪ {p} ∧ sa ≠ pa
      ⟶ lc (σ[s,p] t^[Fvar sa,Fvar pa])
  proof (intro strip)
    fix sa :: fVariable and pa :: fVariable
    assume sa ∉ FV t ∪ {s} ∪ {p} ∧ pa ∉ FV t ∪ {s} ∪ {p} ∧ sa ≠ pa
    hence s ∉ FV (Fvar pa) by auto
    from
      sopen-sclose-eq-ssubst[OF ‹s ≠ p› this ‹lc t›]
```

```
    ssubst-preserves-lc[OF ‹lc t›]
   show lc (σ[s,p] t[Fvar sa,Fvar pa]) by (simp add: openz-def closez-def)
  qed
  thus
   finite (FV t ∪ {s} ∪ {p})
    ∧ (∀ sa pa. sa ∉ FV t ∪ {s} ∪ {p} ∧ pa ∉ FV t ∪ {s} ∪ {p} ∧ sa ≠ pa
       ⟶ lc (σ[s,p] t[Fvar sa,Fvar pa])) by simp
qed

lemma ssubst-preserves-lcE-lem[rule-format]:
  fixes t
  assumes lc t
  shows ∀ x u t'. t = [x → u] t' ⟶ lc u ⟶ lc t'
  using assms
proof
  (induct
    taking:
    λt. ∀ x u t'. t = [x → u] t' ⟶ lc u ⟶ body t'
    rule: lc-induct)
  case Fvar thus ?case by (intro strip, case-tac t', simp-all)
next
  case Call thus ?case by (intro strip, case-tac t', simp-all)
next
  case (Upd t l u) note pred-t = this(2) and pred-u = this(3)
  show ?case
  proof (intro strip)
    fix x v t'' assume Upd t l u = [x → v] t'' and lc v
    from this(1) have t'': (∃ t' u'. t'' = Upd t' l u') ∨ (t'' = Fvar x)
    proof (cases t'', auto)
      fix y
      assume Upd t l u = (if x = y then v else Fvar y)
      thus y = x by (case-tac y = x, auto)
    qed
    show lc t''
    proof (cases t'' = Fvar x)
      case True thus ?thesis by simp
    next
      case False with ‹Upd t l u = [x → v] t''› t''
      show ?thesis
      proof (clarify)
        fix t' u' assume Upd t l u = [x → v] Upd t' l u'
        hence t = [x → v] t' and u = [x → v] u'
          by auto
        with ‹lc v› pred-t pred-u lc-upd[of t' l u']
        show lc (Upd t' l u') by auto
      qed
    qed
  qed
next
```

```
  case (Obj f T) note pred = this
  show ?case
  proof (intro strip)
    fix x v t' assume Obj f T = [x → v] t' and lc v
    from this(1) have t': (∃f'. t' = Obj f' T) ∨ (t' = Fvar x)
    proof (cases t', auto)
      fix y :: fVariable
      assume Obj f T = (if x = y then v else Fvar y)
      thus y = x by (case-tac y = x, auto)
    qed
    show lc t'
    proof (cases t' = Fvar x)
      case True thus ?thesis by simp
    next
      case False with ‹Obj f T = [x → v] t'› t'
      show ?thesis
      proof (clarify)
        fix f' assume Obj f T = [x → v] Obj f' T
        hence
          ssubst: ∀l∈dom f. the(f l) = [x → v] the(f' l) and
          dom f = dom f'
          by auto
        with pred ‹lc v› lc-obj[of f' T]
        show lc (Obj f' T)
          by auto
      qed
    qed
  qed
next
  case (Bnd L t) note pred = this(2)
  show ?case
  proof (intro strip)
    fix x v t' assume t = [x → v] t' and lc v
    from ‹finite L› exFresh-s-p-cof[of L ∪ {x} ∪ FV t']
    obtain s p where
      s ∉ L and p ∉ L and s ≠ p and
      x ∉ FV (Fvar s) and x ∉ FV (Fvar p) and
      s ∉ FV t' and p ∉ FV t'
      by auto
    from
      ‹t = [x → v] t'›
      ssubst-sopen-commute[OF ‹lc v› ‹x ∉ FV (Fvar s)› ‹x ∉ FV (Fvar p)›]
    have (t^[Fvar s, Fvar p]) = [x → v] (t'^[Fvar s, Fvar p])
      by (auto simp: openz-def)
    with
      ‹s ∉ L› ‹p ∉ L› ‹s ≠ p› ‹lc v› pred
    have lc (t'^[Fvar s, Fvar p]) by blast
    from
      lc-body[OF this ‹s ≠ p›]
```

sclose-sopen-eq-t[*OF* ‹$s \notin FV\ t'$› ‹$p \notin FV\ t'$› ‹$s \neq p$›]
show *body* t' **by** (*auto simp*: *openz-def closez-def*)
qed
qed

lemma *ssubst-preserves-lcE*: ⟦ *lc* ($[x \to t']\ t$); *lc* t' ⟧ ⟹ *lc* t
by (*drule-tac* $t = [x \to t']\ t$ **and** $x = x$ **and** $u = t'$ **and** $t' = t$
in *ssubst-preserves-lcE-lem*, *simp*+)

lemma *obj-openz-lc*: ⟦ *lc* (*Obj f T*); *lc p*; $l \in dom\ f$ ⟧ ⟹ *lc* ($the(f\ l)^{[Obj\ f\ T,\ p]}$)
by (*rule-tac* $s = Obj\ f\ T$ **and** $p = p$ **in** *body-lc*, (*simp add*: *lc-obj*)+)

lemma *obj-insert-lc*:
fixes $f\ T\ t\ l$
assumes *lc* (*Obj f T*) **and** *body t*
shows *lc* (*Obj* ($f(l \mapsto t)$) *T*)
proof (*rule ssubst*[*OF lc-obj*], *rule ballI*)
fix l' :: *Label* **assume** $l' \in dom\ (f(l \mapsto t))$
with *assms* **show** *body* (*the* (($f(l \mapsto t)$) l'))
by (*cases* $l' = l$, (*auto simp*: *lc-obj*))
qed

lemma *ssubst-preserves-body*[*simp*]:
fixes $t\ t'\ x$
assumes *body t* **and** *lc* t'
shows *body* ($[x \to t']\ t$)
unfolding *body-def*
proof −
have
$\forall s\ p.\ s \notin FV\ t' \cup \{x\} \wedge p \notin FV\ t' \cup \{x\} \wedge s \neq p$
$\longrightarrow$ *lc* ($[x \to t']\ t^{[Fvar\ s, Fvar\ p]}$)
proof (*intro strip*)
fix s :: *fVariable* **and** p :: *fVariable*
from *body-lc-FV*[*OF* ‹*body t*›]
have *lc* ($\{0 \to [Fvar\ s, Fvar\ p]\}\ t$) **by** (*simp add*: *openz-def*)
from *ssubst-preserves-lc*[*OF this* ‹*lc* t'›]
have *lc* ($[x \to t']\ (t^{[Fvar\ s, Fvar\ p]})$) **by** (*simp add*: *openz-def*)

moreover assume $s \notin FV\ t' \cup \{x\} \wedge p \notin FV\ t' \cup \{x\} \wedge s \neq p$
hence $x \notin FV\ (Fvar\ s)$ **and** $x \notin FV\ (Fvar\ p)$ **by** *auto*
note *ssubst-sopen-commute*[*OF* ‹*lc* t'› *this*]
ultimately
show *lc* ($[x \to t']\ t^{[Fvar\ s, Fvar\ p]}$) **by** (*simp add*: *openz-def*)
qed
thus
$\exists L.$ *finite* $L \wedge (\forall s\ p.\ s \notin L \wedge p \notin L \wedge s \neq p$
$\longrightarrow$ *lc* ($[x \to t']\ t^{[Fvar\ s, Fvar\ p]}$))
by (*rule-tac* $x = FV\ t' \cup \{x\}$ **in** *exI*, *simp*)
qed

lemma *sopen-preserves-body*[*simp*]:
fixes *t s p*
assumes *body t* **and** *lc s* **and** *lc p*
shows *body* ({*n* → [*s*,*p*]} *t*)
unfolding *body-def*
proof −
have
∀ *sa pa*. *sa* ∉ *FV t* ∪ *FV s* ∧ *pa* ∉ *FV p* ∧ *sa* ≠ *pa*
⟶ *lc* ({*n* → [*s*,*p*]} $t^{[Fvar\ sa, Fvar\ pa]}$)
proof (*cases n* = *0*)
case *True* **thus** *?thesis*
using *body-lc*[*OF* ‹*body t*› ‹*lc s*› ‹*lc p*›] *sopen-twice*[*OF* ‹*lc s*› ‹*lc p*›]
by (*simp add*: *openz-def*)
next
case *False* **thus** *?thesis*
proof (*intro strip*)
fix *sa* :: *fVariable* **and** *pa* :: *fVariable*
from *body-lc-FV*[*OF* ‹*body t*›] **have** *lc* ($t^{[Fvar\ sa, Fvar\ pa]}$) **by** *assumption*
moreover
from *sopen-commute-gen*[*OF - -* ‹*lc s*› ‹*lc p*› *not-sym*[*OF* ‹*n* ≠ *0*›]]
have {*n* → [*s*,*p*]} $t^{[Fvar\ sa, Fvar\ pa]}$ = {*n* → [*s*,*p*]} ($t^{[Fvar\ sa, Fvar\ pa]}$)
by (*simp add*: *openz-def*)
ultimately show *lc* ({*n* → [*s*,*p*]} $t^{[Fvar\ sa, Fvar\ pa]}$) **by** *simp*
qed
qed
thus ∃ *L*. *finite L*
∧ (∀ *sa pa*. *sa* ∉ *L* ∧ *pa* ∉ *L* ∧ *sa* ≠ *pa*
⟶ *lc* ({*n* → [*s*,*p*]} $t^{[Fvar\ sa, Fvar\ pa]}$))
by (*rule-tac x* = *FV t* ∪ *FV s* ∪ *FV p* **in** *exI*, *simp*)
qed

3.3 Beta-reduction

inductive *beta* :: [*sterm*, *sterm*] ⇒ *bool* (**infixl** ‹→$_\beta$› *50*)
where
beta[*simp*, *intro*!] :
⟦ *l* ∈ *dom f*; *lc* (*Obj f T*); *lc a* ⟧ ⟹ *Call* (*Obj f T*) *l a* →$_\beta$ (*the* (*f l*)$^{[(Obj\ f\ T),\ a]}$)
| *beta-Upd*[*simp*, *intro*!] :
⟦ *l* ∈ *dom f*; *lc* (*Obj f T*); *body t* ⟧ ⟹ *Upd* (*Obj f T*) *l t* →$_\beta$ *Obj* (*f*(*l* ↦ *t*)) *T*
| *beta-CallL*[*simp*, *intro*!]: ⟦ *t* →$_\beta$ *t*′; *lc u* ⟧ ⟹ *Call t l u* →$_\beta$ *Call t*′ *l u*
| *beta-CallR*[*simp*, *intro*!]: ⟦ *t* →$_\beta$ *t*′; *lc u* ⟧ ⟹ *Call u l t* →$_\beta$ *Call u l t*′
| *beta-UpdL*[*simp*, *intro*!] : ⟦ *t* →$_\beta$ *t*′; *body u* ⟧ ⟹ *Upd t l u* →$_\beta$ *Upd t*′ *l u*
| *beta-UpdR*[*simp*, *intro*!] :
⟦ *finite L*;
∀ *s p*. *s* ∉ *L* ∧ *p* ∉ *L* ∧ *s* ≠ *p* ⟶ (∃ *t*″. $t^{[Fvar\ s, Fvar\ p]}$ →$_\beta$ *t*″∧ *t*′= *σ*[*s*,*p*]*t*″);
lc u ⟧ ⟹ *Upd u l t* →$_\beta$ *Upd u l t*′
| *beta-Obj*[*simp*, *intro*!] :

$[\![\ l \in dom\ f;\ finite\ L;$
$\quad \forall s\ p.\ s \notin L \wedge p \notin L \wedge s \neq p \longrightarrow (\exists t''.\ t^{[Fvar\ s, Fvar\ p]} \rightarrow_\beta t'' \wedge t' = \sigma[s,p]t'');$
$\quad \forall l{\in}dom\ f.\ body\ (the\ (f\ l))\ ]\!]$
$\Longrightarrow Obj\ (f(l \mapsto t))\ T \rightarrow_\beta Obj\ (f(l \mapsto t'))\ T$

inductive-cases *beta-cases* [*elim*!]:
$Call\ s\ l\ t \rightarrow_\beta u$
$Upd\ s\ l\ t \rightarrow_\beta u$
$Obj\ s\ T \rightarrow_\beta t$

abbreviation
beta-reds :: [*sterm*, *sterm*] => *bool* (**infixl** ‹−>>› *50*) **where**
$s\ {-}{>>}\ t == beta\hat{}{**}\ s\ t$
abbreviation
beta-ascii :: [*sterm*, *sterm*] => *bool* (**infixl** ‹−>› *50*) **where**
$s\ {-}{>}\ t == beta\ s\ t$

notation (*latex*)
beta-reds (**infixl** ‹$\rightarrow_\beta{}^*$› *50*)

lemma *beta-induct*[*consumes 1*,
case-names CallL CallR UpdL UpdR Upd Obj beta Bnd]:
fixes
t :: *sterm* **and** t' :: *sterm* **and**
P1 :: *sterm* ⇒ *sterm* ⇒ *bool* **and** *P2* :: *sterm* ⇒ *sterm* ⇒ *bool*
assumes
$t \rightarrow_\beta t'$ **and**
$\bigwedge t\ t'\ u\ l.\ [\![\ t \rightarrow_\beta t';\ P1\ t\ t';\ lc\ u\ ]\!] \Longrightarrow P1\ (Call\ t\ l\ u)\ (Call\ t'\ l\ u)$ **and**
$\bigwedge t\ t'\ u\ l.\ [\![\ t \rightarrow_\beta t';\ P1\ t\ t';\ lc\ u\ ]\!] \Longrightarrow P1\ (Call\ u\ l\ t)\ (Call\ u\ l\ t')$ **and**
$\bigwedge t\ t'\ u\ l.\ [\![\ t \rightarrow_\beta t';\ P1\ t\ t';\ body\ u]\!] \Longrightarrow P1\ (Upd\ t\ l\ u)\ (Upd\ t'\ l\ u)$ **and**
$\bigwedge t\ t'\ u\ l.\ [\![\ P2\ t\ t';\ lc\ u\ ]\!] \Longrightarrow P1\ (Upd\ u\ l\ t)\ (Upd\ u\ l\ t')$ **and**
$\bigwedge l\ f\ T\ t.\ [\![\ l \in dom\ f;\ lc\ (Obj\ f\ T);\ body\ t\ ]\!]$
$\quad \Longrightarrow P1\ (Upd\ (Obj\ f\ T)\ l\ t)\ (Obj\ (f(l \mapsto t))\ T)$ **and**
$\bigwedge l\ f\ t\ t'\ T.\ [\![\ l \in dom\ f;\ P2\ t\ t';\ \forall l{\in}dom\ f.\ body\ (the\ (f\ l))\ ]\!]$
$\quad \Longrightarrow P1\ (Obj\ (f(l \mapsto t))\ T)\ (Obj\ (f(l \mapsto t'))\ T)$ **and**
$\bigwedge l\ f\ T\ a.\ [\![\ l \in dom\ f;\ lc\ (Obj\ f\ T);\ lc\ a\ ]\!]$
$\quad \Longrightarrow P1\ (Call\ (Obj\ f\ T)\ l\ a)\ (the\ (f\ l)^{[Obj\ f\ T, a]})$ **and**
$\bigwedge L\ t\ t'.$
$\quad [\![\ finite\ L;$
$\quad\ \forall s\ p.\ s \notin L \wedge p \notin L \wedge s \neq p$
$\quad\ \longrightarrow (\exists t''.\ t^{[Fvar\ s, Fvar\ p]} \rightarrow_\beta t''$
$\quad\ \wedge\ P1\ (t^{[Fvar\ s, Fvar\ p]})\ t'' \wedge t' = \sigma[s,p]\ t'')\ ]\!]$
$\quad \Longrightarrow P2\ t\ t'$
shows *P1* $t\ t'$
using *assms* **by** (*induct rule*: *beta.induct*, *auto*)

lemma *Fvar-beta*: $Fvar\ x \rightarrow_\beta t \Longrightarrow False$
by (*erule beta.cases*, *auto*)

lemma *Obj-beta*:
assumes $Obj\ f\ T \to_\beta z$
shows
$\exists\, l\ f'\ t\ t'.\ dom\ f = dom\ f' \wedge f = (f'(l \mapsto t)) \wedge l \in dom\ f'$
$\wedge\ (\exists\, L.\ finite\ L$
$\wedge\ (\forall\, s\ p.\ s \notin L \wedge p \notin L \wedge s \neq p$
$\longrightarrow\ (\exists\, t''.\ t^{[Fvar\ s, Fvar\ p]} \to_\beta t'' \wedge t' = \sigma[s,p]t'')))$
$\wedge\ z = Obj\ (f'(l \mapsto t'))\ T$
proof (*cases rule*: *beta-cases*(*3*)[*OF assms*])
case (*1 l fa L t t'*) **thus** *?thesis*
by (*rule-tac* $x = l$ **in** *exI*,
rule-tac $x = fa$ **in** *exI*,
rule-tac $x = t$ **in** *exI*,
rule-tac $x = t'$ **in** *exI*, *auto*)
qed

lemma *Upd-beta*: $Upd\ t\ l\ u \to_\beta z \Longrightarrow$
$(\exists\, t'.\ t \to_\beta t' \wedge z = Upd\ t'\ l\ u)$
$\vee(\exists\, u'\ L.\ finite\ L$
$\wedge\ (\forall\, s\ p.\ s \notin L \wedge p \notin L \wedge s \neq p$
$\longrightarrow\ (\exists\, t''.\ (u^{[Fvar\ s,\ Fvar\ p]}) \to_\beta t'' \wedge u' = \sigma[s,p]t''))$
$\wedge\ z = Upd\ t\ l\ u')$
$\vee(\exists f\ T.\ l \in dom\ f \wedge Obj\ f\ T = t \wedge z = Obj\ (f(l \mapsto u))\ T)$
by (*erule beta-cases*, *auto*)

lemma *Call-beta*: $Call\ t\ l\ u \to_\beta z \Longrightarrow$
$(\exists\, t'.\ t \to_\beta t' \wedge z = Call\ t'\ l\ u) \vee (\exists\, u'.\ u \to_\beta u' \wedge z = Call\ t\ l\ u')$
$\vee(\exists f\ T.\ Obj\ f\ T = t \wedge l \in dom\ f \wedge z = (the\ (f\ l)^{[Obj\ f\ T,\ u]}))$
by (*erule beta-cases*, *auto*)

3.3.1 Properties

lemma *beta-lc*[*simp*]:
fixes $t\ t'$
assumes $t \to_\beta t'$
shows $lc\ t \wedge lc\ t'$
using *assms*
proof
(*induct*
taking: $\lambda t\ t'.\ body\ t \wedge body\ t'$
rule: *beta-induct*)
case *CallL* **thus** *?case* **by** *simp*
next
case *CallR* **thus** *?case* **by** *simp*
next
case *UpdR* **thus** *?case* **by** (*simp add*: *lc-upd*)
next
case *UpdL* **thus** *?case* **by** (*simp add*: *lc-upd*)
next

case *beta* **thus** *?case* **by** (*simp add*: *obj-openz-lc*)
next
case *Upd* **thus** *?case* **by** (*simp add*: *lc-obj lc-upd*)
next
case *Obj* **thus** *?case* **by** (*simp add*: *lc-obj*)
next
case (*Bnd L t t'*) **note** *cof* = *this*(*2*)
from ‹*finite L*› *exFresh-s-p-cof*[*of L* ∪ *FV t*]
obtain *s p* **where**
s ∉ *L* **and** *s* ∉ *FV t* **and** *p* ∉ *L* **and** *p* ∉ *FV t* **and** *s* ≠ *p*
by *auto*
with *cof* **obtain** *t''* **where**
lc (*t*[*Fvar s*, *Fvar p*]) **and** *lc t''* **and**
t' = *σ*[*s*,*p*] *t''* **by** *auto*
from
lc-body[*OF this*(*1*) ‹*s* ≠ *p*›]
sclose-sopen-eq-t[*OF* ‹*s* ∉ *FV t*› ‹*p* ∉ *FV t*› ‹*s* ≠ *p*›]
this(*3*) *lc-body*[*OF this*(*2*) ‹*s* ≠ *p*›]
show *?case* **by** (*simp add*: *openz-def closez-def*)
qed

lemma *beta-ssubst*[*rule-format*]:
fixes *t t'*
assumes *t* →*β* *t'*
shows ∀ *x v*. *lc v* ⟶ [*x* → *v*] *t* →*β* [*x* → *v*] *t'*
proof −
define *pred-cof*
where *pred-cof L t t'* ⟷
(∀ *s p*. *s* ∉ *L* ∧ *p* ∉ *L* ∧ *s* ≠ *p* ⟶ (∃ *t''*. *t*[*Fvar s*, *Fvar p*] →*β* *t''* ∧ *t'* = *σ*[*s*,*p*] *t''*))
for *L t t'*
{
fix *x v t t'*
assume
lc v **and**
∀ *x v*. *lc v* ⟶ (∃ *L*. *finite L* ∧ *pred-cof L* ([*x* → *v*] *t*) ([*x* → *v*] *t'*))
hence
∃ *L*. *finite L* ∧ *pred-cof L* ([*x* → *v*] *t*) ([*x* → *v*] *t'*)
by *auto*
}note *Lex* = *this*

{
fix *x v l* **and** *f* :: *Label* ⇒ *sterm option*
assume *l* ∈ *dom f* **hence** *l* ∈ *dom* (λ*l*. *ssubst-option x v* (*f l*))
by *simp*
}note *domssubst* = *this*
{
fix *x v l T* **and** *f* :: *Label* ⇒ *sterm option*
assume *lc* (*Obj f T*) **and** *lc v* **from** *ssubst-preserves-lc*[*OF this*]

```
  have obj: lc (Obj (λl. ssubst-option x v (f l)) T) by simp
}note lcobj = this

from assms show ?thesis
proof
  (induct
    taking: λt t'. ∀ x v. lc v
                  ⟶ (∃ L. finite L
                        ∧ pred-cof L ([x → v] t) ([x → v] t'))
    rule: beta-induct)
  case CallL thus ?case by simp
next
  case CallR thus ?case by simp
next
  case UpdL thus ?case by simp
next
  case (UpdR t t' u l) note pred = this(1)
  show ?case
  proof (intro strip)
    fix x v assume lc v
    from Lex[OF this pred]
    obtain L where
      finite L and pred-cof L ([x → v] t) ([x → v] t')
      by auto
    with ssubst-preserves-lc[OF ‹lc u› ‹lc v›]
    show [x → v] Upd u l t →β [x → v] Upd u l t'
      unfolding pred-cof-def
      by auto
  qed
next
  case (beta l f T t) thus ?case
  proof (intro strip, simp)
    fix x v assume lc v
    from ssubst-preserves-lc[OF ‹lc t› this] have lc ([x → v] t)
      by simp
    note lem =
      beta.beta[OF domssubst[OF ‹l ∈ dom f›]
      lcobj[OF ‹lc (Obj f T)› ‹lc v›] this]

    from ‹l ∈ dom f› have the (ssubst-option x v (f l)) = [x → v] the (f l)
      by auto
    with lem[of x] ssubst-openz-distrib[OF ‹lc v›]
    show
      Call (Obj (λl. ssubst-option x v (f l)) T) l ([x → v] t)
       →β [x → v] (the (f l)[Obj f T, t])
      by simp
  qed
next
  case (Upd l f T t) thus ?case
```

```
  proof (intro strip, simp)
    fix x v assume lc v
    from ssubst-preserves-body[OF ‹body t› ‹lc v›] have body ([x → v] t)
      by simp
    from
      beta.beta-Upd[OF domssubst[OF ‹l ∈ dom f›]
                       lcobj[OF ‹lc (Obj f T)› ‹lc v›] this]
      ssubstoption-insert[OF ‹l ∈ dom f›]
    show
      Upd (Obj (λl. ssubst-option x v (f l)) T) l ([x → v] t)
       →β Obj (λla. ssubst-option x v (if la = l then Some t else f la)) T
      by simp
  qed
next
  case (Obj l f t t' T) note pred = this(2)
  show ?case
  proof (intro strip, simp)
    fix x v assume lc v
    note Lex[OF this pred]
    from this[of x] obtain L where
      finite L and pred-cof L ([x → v] t) ([x → v] t')
      by auto
    have ∀ l∈dom (λl. ssubst-option x v (f l)). body (the (ssubst-option x v (f l)))
    proof (intro strip, simp)
      fix l' :: Label assume l' ∈ dom f
      with ‹∀ l∈dom f. body (the(f l))› have body (the (f l')) by blast
      note ssubst-preserves-body[OF this ‹lc v›]
      with ‹l' ∈ dom f› ssubst-option-lem
      show body (the (ssubst-option x v (f l'))) by auto
    qed
    from
      beta.beta-Obj[OF domssubst[OF ‹l ∈ dom f›] ‹finite L› - this]
      ssubstoption-insert[OF ‹l ∈ dom f›] ‹pred-cof L ([x → v] t) ([x → v] t')›
    show
      Obj (λla. ssubst-option x v (if la = l then Some t else f la)) T
       →β Obj (λla. ssubst-option x v (if la = l then Some t' else f la)) T
      unfolding pred-cof-def
      by simp
  qed
next
  case (Bnd L t t') note pred = this(2)
  show ?case
  proof (intro strip)
    fix x v assume lc v
    from ‹finite L›
    show ∃ L. finite L ∧ pred-cof L ([x → v] t) ([x → v] t')
    proof (rule-tac x = L ∪ {x} ∪ FV v in exI,
        unfold pred-cof-def, auto)
      fix s p assume s ∉ L and p ∉ L and s ≠ p
```

```
      with pred ‹lc v› obtain t'' where
        t[Fvar s,Fvar p] →β t'' and
        ssubst-beta: [x → v] (t[Fvar s,Fvar p]) →β [x → v] t'' and
        t' = σ[s,p] t''
        by blast
      assume s ≠ x and p ≠ x
      hence x ∉ FV (Fvar s) and x ∉ FV (Fvar p) by auto
      from ssubst-sopen-commute[OF ‹lc v› this] ssubst-beta
      have [x → v] t[Fvar s,Fvar p] →β [x → v] t''
        by (simp add: openz-def)
      moreover
      assume s ∉ FV v and p ∉ FV v
      from
        ssubst-sclose-commute[OF this not-sym[OF ‹s ≠ x›]
                                  not-sym[OF ‹p ≠ x›]]
        ‹t' = σ[s,p] t''›
      have [x → v] t' = σ[s,p] [x → v] t''
        by (simp add: closez-def)
      ultimately
      show ∃ t''. [x → v] t[Fvar s,Fvar p] →β t'' ∧ [x → v] t' = σ[s,p] t''
        by (rule-tac x = [x → v] t'' in exI, simp)
    qed
  qed
 qed
qed

declare if-not-P [simp] not-less-eq [simp]
 — don't add r-into-rtrancl[intro!]

lemma beta-preserves-FV[simp, rule-format]:
 fixes t t' x
 assumes t →β t'
 shows x ∉ FV t ⟶ x ∉ FV t'
using assms
proof
 (induct
   taking: λt t'. x ∉ FV t ⟶ x ∉ FV t'
   rule: beta-induct)
 case CallL thus ?case by simp
next
 case CallR thus ?case by simp
next
 case UpdL thus ?case by simp
next
 case UpdR thus ?case by simp
next
 case Upd thus ?case by simp
next
 case Obj thus ?case by simp
```

next
  case (*beta l f T t*) **thus** *?case*
  proof (*intro strip*)
    assume $x \notin FV\ (Call\ (Obj\ f\ T)\ l\ t)$
    with ‹$l \in dom\ f$› **have** $x \notin FV\ (the\ (f\ l)) \cup FV\ (Obj\ f\ T) \cup FV\ t$
    proof (*auto*)
      fix $y :: sterm$
      assume $x \in FV\ y$ **and** $f\ l = Some\ y$
      hence $x \in FVoption\ (f\ l)$
        by *auto*
      moreover assume $\forall\, l{\in}dom\ f.\ x \notin FVoption\ (f\ l)$
      ultimately show *False* **using** ‹$l \in dom\ f$›
        by *blast*
    qed
    from *contra-subsetD*[*OF sopen-FV this*]
    show $x \notin FV\ (the\ (f\ l)^{[Obj\ f\ T,t]})$ **by** (*simp add*: *openz-def*)
  qed
next
  case (*Bnd L t t'*) **thus** *?case*
  proof (*intro strip*)
    assume $x \notin FV\ t$
    from ‹*finite L*› *exFresh-s-p-cof*[*of L* $\cup$ $\{x\}$]
    obtain *s p* **where** *sp*: $s \notin L \cup \{x\} \wedge p \notin L \cup \{x\} \wedge s \neq p$ **by** *auto*
    with ‹$x \notin FV\ t$› *sopen-FV*[*of 0 Fvar s Fvar p t*]
    have $x \notin FV\ (t^{[Fvar\ s,\ Fvar\ p]})$ **by** (*auto simp*: *openz-def*)
    with *sp Bnd*(*2*) **obtain** t'' **where**
      $x \notin FV\ t''$ **and** $t' = \sigma[s,p]\ t''$
      by *auto*
    with *sclose-subset-FV*[*of 0 s p t''*] **show** $x \notin FV\ t'$
      by (*auto simp*: *closez-def*)
  qed
qed

lemma *rtrancl-beta-lc*[*simp*, *rule-format*]: $t \rightarrow_\beta^*\ t' \Longrightarrow t \neq t' \longrightarrow lc\ t \wedge lc\ t'$
  by (*erule rtranclp.induct*, *simp*,
      drule beta-lc, *blast*)

lemma *rtrancl-beta-lc2*[*simp*]: $[\![\ t \rightarrow_\beta^*\ t';\ lc\ t\ ]\!] \Longrightarrow lc\ t'$
  by (*case-tac* $t = t'$, *simp*+)

lemma *rtrancl-beta-body*:
  fixes *L t t'*
  assumes
  finite L **and**
  $\forall\, s\ p.\ s \notin L \wedge p \notin L \wedge s \neq p$
    $\longrightarrow (\exists\, t''.\ t^{[Fvar\ s,Fvar\ p]} \rightarrow_\beta^*\ t'' \wedge t' = \sigma[s,p]\ t'')$ **and**
  body t
  shows *body t'*
proof (*cases* $t = t'$)

case *True* **with** *assms(3)* **show** *?thesis* **by** *simp*
next
from *exFresh-s-p-cof*[*OF* ‹*finite L*›]
obtain *s p* **where** *sp*: $s \notin L \wedge p \notin L \wedge s \neq p$ **by** *auto*
hence $s \neq p$ **by** *simp*

from *assms(2) sp*
obtain t'' **where** $t^{[Fvar\ s, Fvar\ p]} \rightarrow_\beta{}^*\ t''$ **and** $t' = \sigma[s,p]\ t''$
by *auto*
with ‹*body t*› **have** *lc* t''
proof (*cases* $(t^{[Fvar\ s, Fvar\ p]}) = t''$)
case *True* **with** *body-lc*[*OF* ‹*body t*›] **show** *lc* t'' **by** *auto*
next
case *False* **with** *rtrancl-beta-lc*[*OF* ‹$t^{[Fvar\ s, Fvar\ p]} \rightarrow_\beta{}^*\ t''$›]
show *lc* t'' **by** *auto*
qed
from *lc-body*[*OF this* ‹$s \neq p$›] ‹$t' = \sigma[s,p]\ t''$› **show** *body* t' **by** *simp*
qed

lemma *rtrancl-beta-preserves-FV*[*simp*, *rule-format*]:
$t \rightarrow_\beta{}^*\ t' \Longrightarrow x \notin FV\ t \longrightarrow x \notin FV\ t'$
proof (*induct t t' rule*: *rtranclp.induct*, *simp*)
case (*rtrancl-into-rtrancl a b c*) **thus** *?case*
proof (*clarify*)
assume $x \notin FV\ b$ **and** $x \in FV\ c$
from *beta-preserves-FV*[*OF* ‹$b \rightarrow_\beta c$› *this(1)*] *this(2)*
show *False* **by** *simp*
qed
qed

3.3.2 Congruence rules

lemma *rtrancl-beta-CallL* [*intro*!, *rule-format*]:
$[\![\ t \rightarrow_\beta{}^*\ t';\ lc\ u\ ]\!] \Longrightarrow Call\ t\ l\ u \rightarrow_\beta{}^*\ Call\ t'\ l\ u$
proof (*induct t t' rule*: *rtranclp.induct*, *simp*)
case (*rtrancl-into-rtrancl a b c*) **thus** *?case*
proof (*auto*)
from ‹$b \rightarrow_\beta c$› ‹*lc u*› **have** $Call\ b\ l\ u \rightarrow_\beta Call\ c\ l\ u$ **by** *simp*
with *rtrancl-into-rtrancl*(*2*)[*OF* ‹*lc u*›]
show $Call\ a\ l\ u \rightarrow_\beta{}^*\ Call\ c\ l\ u$ **by** *auto*
qed
qed

lemma *rtrancl-beta-CallR* [*intro*!, *rule-format*]:
$[\![\ t \rightarrow_\beta{}^*\ t';\ lc\ u\ ]\!] \Longrightarrow Call\ u\ l\ t \rightarrow_\beta{}^*\ Call\ u\ l\ t'$
proof (*induct t t' rule*: *rtranclp.induct*, *simp*)
case (*rtrancl-into-rtrancl a b c*) **thus** *?case*
proof (*auto*)
from ‹$b \rightarrow_\beta c$› ‹*lc u*› **have** $Call\ u\ l\ b \rightarrow_\beta Call\ u\ l\ c$ **by** *simp*

with *rtrancl-into-rtrancl*(*2*)[*OF* ‹*lc u*›]
show *Call u l a* $\rightarrow_\beta{}^*$ *Call u l c* **by** *auto*
qed
qed

lemma *rtrancl-beta-Call* [*intro*!, *rule-format*]:
$[\![t \rightarrow_\beta{}^* t'; lc\ t; u \rightarrow_\beta{}^* u'; lc\ u]\!]$
$\Longrightarrow$ *Call t l u* $\rightarrow_\beta{}^*$ *Call t' l u'*
proof (*induct t t' rule*: *rtranclp.induct*, *blast*)
case (*rtrancl-into-rtrancl a b c*) **thus** *?case*
proof (*auto*)
from ‹$u \rightarrow_\beta{}^* u'$› ‹*lc u*› **have** *lc u'* **by** *auto*
with ‹$b \rightarrow_\beta c$› **have** *Call b l u'* $\rightarrow_\beta$ *Call c l u'* **by** *simp*
with *rtrancl-into-rtrancl*(*2*)[*OF* ‹*lc a*› ‹$u \rightarrow_\beta{}^* u'$› ‹*lc u*›]
show *Call a l u* $\rightarrow_\beta{}^*$ *Call c l u'* **by** *auto*
qed
qed

lemma *rtrancl-beta-UpdL*:
$[\![t \rightarrow_\beta{}^* t'; body\ u]\!] \Longrightarrow$ *Upd t l u* $\rightarrow_\beta{}^*$ *Upd t' l u*
proof (*induct t t' rule*: *rtranclp.induct*, *simp*)
case (*rtrancl-into-rtrancl a b c*) **thus** *?case*
proof (*auto*)
from ‹$b \rightarrow_\beta c$› ‹*body u*› **have** *Upd b l u* $\rightarrow_\beta$ *Upd c l u* **by** *simp*
with *rtrancl-into-rtrancl*(*2*)[*OF* ‹*body u*›]
show *Upd a l u* $\rightarrow_\beta{}^*$ *Upd c l u* **by** *auto*
qed
qed

lemma *beta-binder*[*rule-format*]:
fixes *t t'*
assumes $t \rightarrow_\beta t'$
shows
$\forall L\ s\ p.$ *finite* $L \longrightarrow s \notin L \longrightarrow p \notin L \longrightarrow s \neq p$
$\longrightarrow (\exists L'.$ *finite* $L' \wedge (\forall sa\ pa.\ sa \notin L' \wedge pa \notin L' \wedge sa \neq pa$
$\longrightarrow (\exists t''.\ (\sigma[s,p]\ t)^{[Fvar\ sa, Fvar\ pa]} \rightarrow_\beta t''$
$\wedge\ \sigma[s,p]\ t' = \sigma[sa,pa]\ t'')))$
proof (*intro strip*)
fix *L* :: *fVariable set* **and** *s* :: *fVariable* **and** *p* :: *fVariable*
assume $s \neq p$
have
$\forall sa\ pa.\ sa \notin L \cup FV\ t \cup \{s\} \cup \{p\} \wedge pa \notin L \cup FV\ t \cup \{s\} \cup \{p\} \wedge sa \neq pa$
$\longrightarrow (\exists t''.\ (\sigma[s,p]\ t)^{[Fvar\ sa, Fvar\ pa]} \rightarrow_\beta t'' \wedge \sigma[s,p]\ t' = \sigma[sa,pa]\ t'')$
proof (*intro strip*)
fix *sa* :: *fVariable* **and** *pa* :: *fVariable*
from *beta-ssubst*[*OF* ‹$t \rightarrow_\beta t'$›]
have $[p \rightarrow Fvar\ pa]\ t \rightarrow_\beta [p \rightarrow Fvar\ pa]\ t'$ **by** *simp*
from *beta-ssubst*[*OF this*]
have

betasubst: $[s \to Fvar\ sa]\ [p \to Fvar\ pa]\ t \to_\beta [s \to Fvar\ sa]\ [p \to Fvar\ pa]\ t'$
by *simp*

from *beta-lc*[*OF* ‹$t \to_\beta t'$›] **have** *lc t* **and** *lc t'* **by** *auto*

assume
sapa: $sa \notin L \cup FV\ t \cup \{s\} \cup \{p\} \wedge pa \notin L \cup FV\ t \cup \{s\} \cup \{p\} \wedge sa \neq pa$
hence $s \notin FV\ (Fvar\ pa)$ **by** *auto*
from
sopen-sclose-eq-ssubst[*OF* ‹$s \neq p$› *this* ‹*lc t*›]
sopen-sclose-eq-ssubst[*OF* ‹$s \neq p$› *this* ‹*lc t'*›]
betasubst
have $\sigma[s,p]\ t^{[Fvar\ sa,\ Fvar\ pa]} \to_\beta (\sigma[s,p]\ t'^{[Fvar\ sa,\ Fvar\ pa]})$
by (*simp add*: *openz-def closez-def*)

moreover
{
from *sapa* **have** $sa \notin FV\ t$ **by** *simp*
from
contra-subsetD[*OF sclose-subset-FV*
beta-preserves-FV[*OF* ‹$t \to_\beta t'$› *this*]]
have $sa \notin FV\ (\sigma[s,p]\ t')$ **by** (*simp add*: *closez-def*)
moreover
from *sapa* **have** $pa \notin FV\ t$ **by** *simp*
from
contra-subsetD[*OF sclose-subset-FV*
beta-preserves-FV[*OF* ‹$t \to_\beta t'$› *this*]]
have $pa \notin FV\ (\sigma[s,p]\ t')$ **by** (*simp add*: *closez-def*)
ultimately
have $sa \notin FV\ (\sigma[s,p]\ t')$ **and** $pa \notin FV\ (\sigma[s,p]\ t')$ **and** $sa \neq pa$
using *sapa*
by *auto*
note *sym*[*OF sclose-sopen-eq-t*[*OF this*]]
}
ultimately
show
$\exists t''.\ \sigma[s,p]\ t^{[Fvar\ sa,Fvar\ pa]} \to_\beta t'' \wedge \sigma[s,p]\ t' = \sigma[sa,pa]\ t''$
by (*auto simp*: *openz-def closez-def*)
qed
moreover assume *finite L*
ultimately
show
$\exists L'.\ finite\ L' \wedge (\forall sa\ pa.\ sa \notin L' \wedge pa \notin L' \wedge sa \neq pa \longrightarrow (\exists t''.\ \sigma[s,p]\ t^{[Fvar\ sa,Fvar\ pa]} \to_\beta t'' \wedge \sigma[s,p]\ t' = \sigma[sa,pa]\ t''))$
by (*rule-tac* $x = L \cup FV\ t \cup \{s\} \cup \{p\}$ **in** *exI*, *simp*)
qed

lemma *rtrancl-beta-UpdR*:

fixes $L\ t\ t'\ u\ l$
assumes
$\forall s\ p.\ s \notin L \wedge p \notin L \wedge s \neq p$
$\longrightarrow (\exists t''.\ (t^{[Fvar\ s,\ Fvar\ p]}) \rightarrow_\beta{}^*\ t'' \wedge t' = \sigma[s,p]t'')$ **and**
finite L **and** *lc* u
shows *Upd* $u\ l\ t \rightarrow_\beta{}^*$ *Upd* $u\ l\ t'$
proof −
from ‹*finite* L› **have** *finite* $(L \cup FV\ t)$ **by** *simp*
from *exFresh-s-p-cof*[*OF this*]
obtain $s\ p$ **where** *sp*: $s \notin L \cup FV\ t \wedge p \notin L \cup FV\ t \wedge s \neq p$ **by** *auto*
with *assms*(*1*) **obtain** t'' **where** $t^{[Fvar\ s,Fvar\ p]} \rightarrow_\beta{}^*\ t''$ **and** t': $t' = \sigma[s,p]\ t''$
by *auto*
with ‹*lc* u› **have** *Upd* $u\ l\ t \rightarrow_\beta{}^*$ *Upd* $u\ l\ \sigma[s,p]\ t''$
proof (*erule-tac rtranclp-induct*)
from *sp* **have** $s \notin FV\ t$ **and** $p \notin FV\ t$ **and** $s \neq p$ **by** *auto*
from *sclose-sopen-eq-t*[*OF this*]
show *Upd* $u\ l\ t \rightarrow_\beta{}^*$ *Upd* $u\ l\ (\sigma[s,p](t^{[Fvar\ s,Fvar\ p]}))$
by (*simp add*: *openz-def closez-def*)
next
fix y :: *sterm* **and** z :: *sterm*
assume $y \rightarrow_\beta z$
from *sp* **have** $s \notin L$ **and** $p \notin L$ **and** $s \neq p$ **by** *auto*
from *beta-binder*[*OF* ‹$y \rightarrow_\beta z$› ‹*finite* L› *this*]
obtain L' **where**
finite L' **and**
$\forall sa\ pa.\ sa \notin L' \wedge pa \notin L' \wedge sa \neq pa$
$\longrightarrow (\exists t''.\ \sigma[s,p]\ y^{[Fvar\ sa,Fvar\ pa]} \rightarrow_\beta t'' \wedge \sigma[s,p]\ z = \sigma[sa,pa]\ t'')$
by *auto*
from *beta.beta-UpdR*[*OF this* ‹*lc* u›]
have *Upd* $u\ l\ (\sigma[s,p]\ y) \rightarrow_\beta$ *Upd* $u\ l\ (\sigma[s,p]\ z)$ **by** *assumption*
moreover assume *Upd* $u\ l\ t \rightarrow_\beta{}^*$ *Upd* $u\ l\ (\sigma[s,p]\ y)$
ultimately show *Upd* $u\ l\ t \rightarrow_\beta{}^*$ *Upd* $u\ l\ (\sigma[s,p]\ z)$ **by** *simp*
qed
with t' **show** *Upd* $u\ l\ t \rightarrow_\beta{}^*$ *Upd* $u\ l\ t'$ **by** *simp*
qed

lemma *rtrancl-beta-Upd*:
⟦ $u \rightarrow_\beta{}^*\ u'$; *finite* L;
$\forall s\ p.\ s \notin L \wedge p \notin L \wedge s \neq p$
$\longrightarrow (\exists t''.\ t^{[Fvar\ s,Fvar\ p]} \rightarrow_\beta{}^*\ t'' \wedge t' = \sigma[s,p]t'')$;
lc u; *body* t ⟧
$\Longrightarrow$ *Upd* $u\ l\ t \rightarrow_\beta{}^*$ *Upd* $u'\ l\ t'$
proof (*induct* $u\ u'$ *rule*: *rtranclp.induct*)
case *rtrancl-refl* **thus** *?case* **by** (*simp add*: *rtrancl-beta-UpdR*)
next
case (*rtrancl-into-rtrancl a b c*) **thus** *?case*
proof (*auto*)
from *rtrancl-beta-body*[*OF* ‹*finite* L› *rtrancl-into-rtrancl*(*5*) ‹*body* t›] ‹$b \rightarrow_\beta c$›
have *Upd* $b\ l\ t' \rightarrow_\beta$ *Upd* $c\ l\ t'$ **by** *simp*

with *rtrancl-into-rtrancl*(*2*)[*OF* ‹*finite L*› *rtrancl-into-rtrancl*(*5*) ‹*lc a*› ‹*body t*›]
show *Upd a l t* $\rightarrow_\beta^*$ *Upd c l t'* **by** *simp*
qed
qed

lemma *rtrancl-beta-obj*:
fixes *l f L T t t'*
assumes
l ∈ *dom f* **and** *finite L* **and**
∀ *s p*. *s* ∉ *L* ∧ *p* ∉ *L* ∧ *s* ≠ *p*
⟶ (∃ *t''*. $t^{[Fvar\ s, Fvar\ p]}$ $\rightarrow_\beta^*$ *t''* ∧ *t'* = σ[*s*,*p*]*t''*) **and**
∀ *l*∈*dom f*. *body* (*the*(*f l*)) **and** *body t*
shows *Obj* (*f* (*l* ↦ *t*)) *T* $\rightarrow_\beta^*$ *Obj* (*f* (*l* ↦ *t'*)) *T*
proof −
from ‹*finite L*› **have** *finite* (*L* ∪ *FV t*) **by** *simp*
from *exFresh-s-p-cof*[*OF this*]
obtain *s p* **where** *sp*: *s* ∉ *L* ∪ *FV t* ∧ *p* ∉ *L* ∪ *FV t* ∧ *s* ≠ *p* **by** *auto*
with *assms*(*3*) **obtain** *t''* **where** $t^{[Fvar\ s, Fvar\ p]}$ $\rightarrow_\beta^*$ *t''* **and** *t'* = σ[*s*,*p*] *t''*
by *auto*
with ‹*l* ∈ *dom f*› ‹∀ *l*∈*dom f*. *body* (*the*(*f l*))›
have *Obj* (*f*(*l* ↦ *t*)) *T* $\rightarrow_\beta^*$ *Obj* (*f*(*l* ↦ σ[*s*,*p*] *t''*)) *T*
proof (*erule-tac rtranclp-induct*)
from *sp* **have** *s* ∉ *FV t* **and** *p* ∉ *FV t* **and** *s* ≠ *p* **by** *auto*
from *sclose-sopen-eq-t*[*OF this*]
show *Obj* (*f*(*l* ↦ *t*)) *T* $\rightarrow_\beta^*$ *Obj* (*f*(*l* ↦ σ[*s*,*p*] ($t^{[Fvar\ s, Fvar\ p]}$))) *T*
by (*simp add*: *openz-def closez-def*)
next
fix *y* :: *sterm* **and** *z* :: *sterm* **assume** *y* $\rightarrow_\beta$ *z*
from *sp* **have** *s* ∉ *L* **and** *p* ∉ *L* **and** *s* ≠ *p* **by** *auto*
from *beta-binder*[*OF* ‹*y* $\rightarrow_\beta$ *z*› ‹*finite L*› *this*]
obtain *L'* **where**
finite L' **and**
∀ *sa pa*. *sa* ∉ *L'* ∧ *pa* ∉ *L'* ∧ *sa* ≠ *pa*
⟶ (∃ *t''*. σ[*s*,*p*] $y^{[Fvar\ sa, Fvar\ pa]}$ $\rightarrow_\beta$ *t''* ∧ σ[*s*,*p*] *z* = σ[*sa*,*pa*] *t''*)
by *auto*
from *beta.beta-Obj*[*OF* ‹*l* ∈ *dom f*› *this* ‹∀ *l*∈*dom f*. *body* (*the*(*f l*))›]
have *Obj* (*f*(*l* ↦ σ[*s*,*p*] *y*)) *T* $\rightarrow_\beta$ *Obj* (*f*(*l* ↦ σ[*s*,*p*] *z*)) *T*
by *assumption*
moreover assume *Obj* (*f*(*l* ↦ *t*)) *T* $\rightarrow_\beta^*$ *Obj* (*f*(*l* ↦ σ[*s*,*p*] *y*)) *T*
ultimately
show *Obj* (*f*(*l* ↦ *t*)) *T* $\rightarrow_\beta^*$ *Obj* (*f*(*l* ↦ σ[*s*,*p*] *z*)) *T* **by** *simp*
qed
with ‹*t'* = σ[*s*,*p*] *t''*› **show** *Obj* (*f*(*l* ↦ *t*)) *T* $\rightarrow_\beta^*$ *Obj* (*f*(*l* ↦ *t'*)) *T*
by *simp*
qed

lemma *obj-lem*:
fixes *l f T L t'*

assumes
$l \in dom\ f$ **and** *finite* L **and**
$\forall\, s\ p.\ s \notin L \wedge p \notin L \wedge s \neq p$
$\longrightarrow (\exists\, t''.\ ((the(f\ l))^{[Fvar\ s, Fvar\ p]}) \rightarrow_\beta{}^*\ t'' \wedge t' = \sigma[s,p]t'')$ **and**
$\forall\, l{\in}dom\ f.\ body\ (the(f\ l))$
shows $Obj\ f\ T \rightarrow_\beta{}^*\ Obj\ (f(l \mapsto t'))\ T$
proof
(*rule-tac* $P = \lambda y.\ Obj\ y\ T \rightarrow_\beta{}^*\ Obj\ (f(l \mapsto t'))\ T$ **and** $s = (f(l \mapsto the(f\ l)))$
in *subst*)
from ‹$l \in dom\ f$› *fun-upd-idem* **show** $f(l \mapsto the\ (f\ l)) = f$ **by** *force*
next
from ‹$l \in dom\ f$› ‹$\forall\, l{\in}dom\ f.\ body\ (the(f\ l))$› **have** $body\ (the\ (f\ l))$
by *blast*
with
rtrancl-beta-obj[*OF* ‹$l \in dom\ f$› ‹*finite* L› *assms*(3) ‹$\forall\, l{\in}dom\ f.\ body\ (the(f\ l))$›]
show $Obj\ (f(l \mapsto the\ (f\ l)))\ T \rightarrow_\beta{}^*\ Obj\ (f(l \mapsto t'))\ T$ **by** *simp*
qed

lemma *rtrancl-beta-obj-lem00*:
fixes $L\ f\ g$
assumes
finite L **and**
$\forall\, l{\in}dom\ f.\ \forall\, s\ p.\ s \notin L \wedge p \notin L \wedge s \neq p$
$\longrightarrow (\exists\, t''.\ ((the(f\ l))^{[Fvar\ s,\ Fvar\ p]}) \rightarrow_\beta{}^*\ t''$
$\wedge\ the(g\ l) = \sigma[s,p]t'')$ **and**
$dom\ f = dom\ g$ **and** $\forall\, l{\in}dom\ f.\ body\ (the\ (f\ l))$
shows
$\forall\, k \leq (card\ (dom\ f)).$
$(\exists\, ob.\ length\ ob = (k + 1)$
$\wedge\ (\forall\, obi.\ obi \in set\ ob \longrightarrow dom\ (fst(obi)) = dom\ f \wedge ((snd\ obi) \subseteq dom\ f))$
$\wedge\ (fst\ (ob!0) = f)$
$\wedge\ (card\ (snd\ (ob!k)) = k)$
$\wedge\ (\forall\, i < k.\ snd\ (ob!i) \subset snd\ (ob!k))$
$\wedge\ (Obj\ (fst\ (ob!0))\ T \rightarrow_\beta{}^*\ Obj\ (fst\ (ob!k))\ T)$
$\wedge\ (card\ (snd\ (ob!k)) = k$
$\longrightarrow (Ltake\text{-}eq\ (snd\ (ob!k))\ (fst\ (ob!k))\ g)$
$\wedge\ (Ltake\text{-}eq\ ((dom\ f) - (snd\ (ob!k)))\ (fst\ (ob!k))\ f)))$
proof
fix $k :: nat$
show
$k \leq card\ (dom\ f)$
$\longrightarrow (\exists\, ob.\ length\ ob = k + 1$
$\wedge\ (\forall\, obi.\ obi \in set\ ob \longrightarrow dom\ (fst\ obi) = dom\ f \wedge snd\ obi \subseteq dom\ f)$
$\wedge\ fst\ (ob\ !\ 0) = f$
$\wedge\ card\ (snd\ (ob\ !\ k)) = k$
$\wedge\ (\forall\, i{<}k.\ snd\ (ob\ !\ i) \subset snd\ (ob\ !\ k))$
$\wedge\ Obj\ (fst\ (ob\ !\ 0))\ T \rightarrow_\beta{}^*\ Obj\ (fst\ (ob\ !\ k))\ T$
$\wedge\ (card\ (snd\ (ob\ !\ k)) = k$
$\longrightarrow Ltake\text{-}eq\ (snd\ (ob\ !\ k))\ (fst\ (ob\ !\ k))\ g$

```
                ∧ Ltake-eq (dom f − snd (ob ! k)) (fst (ob ! k)) f))
proof (induct k)
  case 0 thus ?case
  by (simp, rule-tac x = [(f,{})] in exI, simp add: Ltake-eq-def)
next
  case (Suc k) thus ?case
  proof (clarify)
    assume Suc k ≤ card (dom f) hence k < card (dom f) by arith
    with Suc.hyps
    obtain ob where
      length ob = k + 1 and
      mem-ob: ∀ obi. obi ∈ set ob
              ⟶ dom (fst obi) = dom f ∧ snd obi ⊆ dom f and
      fst (ob ! 0) = f and
      card (snd (ob ! k)) = k and
      ∀ i<k. snd (ob ! i) ⊂ snd (ob ! k) and
      Obj (fst (ob ! 0)) T →β* Obj (fst (ob ! k)) T and
      card-k: card (snd (ob ! k)) = k
              ⟶ Ltake-eq (snd (ob ! k)) (fst (ob ! k)) g
                ∧ Ltake-eq (dom f − snd (ob ! k)) (fst (ob ! k)) f
      by auto
    from ‹length ob = k + 1› have obkmem: (ob!k) ∈ set ob by auto

    with mem-ob have obksnd: snd(ob!k) ⊆ dom f by blast
    from
      card-psubset[OF finite-dom-fmap this] ‹card (snd(ob!k)) = k›
      ‹k < card (dom f)›
    have snd (ob!k) ⊂ dom f by simp
    then obtain l' where l' ∈ dom f and l' ∉ snd (ob!k) by auto

    from obkmem mem-ob have obkfst: dom (fst(ob!k)) = dom f by blast


  define ob' where ob' = ob @ [((fst(ob!k))(l' ↦ the (g l')), insert l' (snd(ob!k)))]

    from nth-fst[OF ‹length ob = k + 1›] have first: ob'!0 = ob!0
      by (simp add: ob'-def)

    from ‹length ob = k + 1› nth-last[of ob Suc k]
    have last: ob'!Suc k = ((fst(ob!k))(l' ↦ the (g l')), insert l' (snd(ob!k)))
      by (simp add: ob'-def)

    from ‹length ob = k + 1› nth-append[of ob - k] have kth: ob'!k = ob!k
      by (auto simp: ob'-def)

    from ‹card (snd(ob!k)) = k› card-k
    have ass:
      ∀ l∈(snd(ob!k)). fst(ob!k) l = g l
      ∀ l∈(dom f − snd(ob!k)). fst(ob!k) l = f l
```

by (*auto simp*: *Ltake-eq-def*)

from ‹*length ob* = *k* + *1*› **have** *length ob′* = *Suc k* + *1*
by (*auto simp*: *ob′-def*)

moreover
have ∀ *obi*. *obi* ∈ *set ob′* ⟶ *dom* (*fst obi*) = *dom f* ∧ *snd obi* ⊆ *dom f*
unfolding *ob′-def*
proof (*intro strip*)
fix *obi* :: (*Label* −~> *sterm*) × (*Label set*)
assume *obi* ∈ *set* (*ob* @ [((*fst*(*ob*!*k*))(*l′* ↦ *the* (*g l′*)), *insert l′* (*snd* (*ob*!*k*)))])
note *mem-append-lem′*[*OF this*]
thus *dom* (*fst obi*) = *dom f* ∧ *snd obi* ⊆ *dom f*
proof (*rule disjE*, *simp-all*)
assume *obi* ∈ *set ob*
with *mem-ob* **show** *dom* (*fst obi*) = *dom f* ∧ *snd obi* ⊆ *dom f*
by *blast*
next
from *obkfst obksnd* ‹*l′* ∈ *dom f*›
show
insert l′ (*dom* (*fst* (*ob*!*k*))) = *dom f*
∧ *l′* ∈ *dom f* ∧ *snd*(*ob*!*k*) ⊆ *dom f*
by *blast*
qed
qed

moreover
from *first* ‹*fst*(*ob*!*0*) = *f*› **have** *fst*(*ob′*!*0*) = *f* **by** *simp*

moreover
from *obksnd finite-dom-fmap finite-subset*
have *finite* (*snd* (*ob*!*k*)) **by** *auto*
from *card.insert-remove*[*OF this*]
have *card* (*insert l′* (*snd* (*ob*!*k*))) = *Suc* (*card* (*snd*(*ob*!*k*) − {*l′*}))
by *simp*
with ‹*l′* ∉ *snd* (*ob*!*k*)› ‹*card* (*snd*(*ob*!*k*)) = *k*› *last*
have *card*(*snd*(*ob′*!*Suc k*)) = *Suc k* **by** *auto*

moreover
have ∀ *i*<*Suc k*. *snd* (*ob′*!*i*) ⊂ *snd* (*ob′*!*Suc k*)
proof (*intro strip*)
fix *i* :: *nat*
from *last* **have** *snd*(*ob′*!*Suc k*) = *insert l′* (*snd* (*ob*!*k*)) **by** *simp*

```
  with ‹l' ∉ snd (ob!k)› have snd(ob!k) ⊂ snd(ob'!Suc k) by auto
  moreover
  assume i < Suc k
  with ‹length ob = k + 1› have i < length ob by simp
  with nth-append[of ob - i] have ob'!i = ob!i by (simp add: ob'-def)
  ultimately show snd(ob'!i) ⊂ snd(ob'!Suc k)
  proof (cases i < k)
    case True
    with
      ‹∀ i<k. snd(ob!i) ⊂ snd(ob!k)› ‹ob'!i = ob!i›
      ‹snd(ob!k) ⊂ snd(ob'!Suc k)›
    show snd (ob'!i) ⊂ snd (ob'!Suc k) by auto
  next
    case False with ‹i < Suc k› have i = k by arith
    with ‹ob'!i = ob!i› ‹snd(ob!k) ⊂ snd(ob'!Suc k)›
    show snd (ob'!i) ⊂ snd (ob'!Suc k) by auto
  qed
qed


moreover
{
  from ‹l' ∈ dom f› ‹l' ∉ snd(ob!k)› have l' ∈ (dom f − snd(ob!k))
    by auto
  with ass have the(fst(ob!k) l') = the(f l') by auto
  with ‹l' ∈ dom f› assms(2)
  have
    sp: ∀ s p. s ∉ L ∧ p ∉ L ∧ s ≠ p
          ⟶ (∃ t''. the(fst(ob!k) l')[Fvar s,Fvar p] →β* t''
                ∧ the (g l') = σ[s,p] t'')
    by simp

  moreover
  have ∀ l∈dom (fst(ob!k)). body (the(fst(ob!k)) l))
  proof (intro strip)
    fix la :: Label
    assume la ∈ dom (fst(ob!k))
    with obkfst have inf: la ∈ dom f by auto
    with assms(4) have bodyf: body (the(f la)) by auto
    show body (the(fst(ob!k)) la))
    proof (cases la ∈ snd(ob!k))
      case False with inf have la ∈ (dom f − snd(ob!k)) by auto
      with ass have fst(ob!k) la = f la by blast
      with bodyf show body (the (fst(ob!k)) la)) by auto
    next
      from exFresh-s-p-cof[OF ‹finite L›]
      obtain s p where s ∉ L ∧ p ∉ L ∧ s ≠ p by auto
      with assms(2) inf
      obtain t' where
```

```
      the (f la)^[Fvar s,Fvar p] →β* t' and
      the (g la) = σ[s,p] t' by blast
    from body-lc[OF bodyf] have lcf: lc (the (f la)^[Fvar s,Fvar p]) by auto
    hence bodyg: body (the(g la))
    proof (cases (the (f la)^[Fvar s,Fvar p]) = t')
      case True
      with
        lcf lc-body ‹s ∉ L ∧ p ∉ L ∧ s ≠ p›
        ‹the(g la) = σ[s,p] t'›
      show body (the(g la)) by auto
    next
      case False
      with
        rtrancl-beta-lc[OF ‹the (f la)^[Fvar s,Fvar p] →β* t'›]
        lc-body ‹s ∉ L ∧ p ∉ L ∧ s ≠ p› ‹the(g la) = σ[s,p] t'›
      show body (the(g la)) by auto
    qed
    case True with ass bodyg show body (the(fst(ob!k) la)) by simp
  qed
qed

moreover
from ‹l' ∈ dom f› obkfst have l' ∈ dom(fst(ob!k)) by auto
note obj-lem[OF this ‹finite L›]

ultimately
have Obj (fst(ob!k)) T →β* Obj ((fst(ob!k))(l' ↦ the (g l'))) T
  by blast

moreover
from last have fst(ob'!Suc k) = (fst(ob!k))(l' ↦ the (g l'))
  by auto

ultimately
have Obj (fst(ob'!0)) T →β* Obj (fst(ob'!Suc k)) T
  using
    rtranclp-trans[OF ‹Obj (fst (ob!0)) T →β* Obj (fst (ob!k)) T›] first kth
  by auto
}


moreover
from ‹l' ∈ dom f› ‹dom f = dom g›
have
  card (snd(ob'!Suc k)) = Suc k
   ⟶ Ltake-eq (snd (ob'!Suc k)) (fst (ob'!Suc k)) g
      ∧ Ltake-eq (dom f − snd(ob'!Suc k)) (fst(ob'!Suc k)) f
  by (auto simp: Ltake-eq-def last ass)
```

```
    ultimately
    show
      ∃ ob. length ob = Suc k + 1
          ∧ (∀ obi. obi ∈ set ob ⟶ dom (fst obi) = dom f ∧ snd obi ⊆ dom f)
          ∧ fst (ob ! 0) = f
          ∧ card (snd (ob ! Suc k)) = Suc k
          ∧ (∀ i<Suc k. snd (ob ! i) ⊂ snd (ob ! Suc k))
          ∧ Obj (fst (ob ! 0)) T →β* Obj (fst (ob ! Suc k)) T
          ∧ (card (snd (ob ! Suc k)) = Suc k
             ⟶ Ltake-eq (snd (ob ! Suc k)) (fst (ob ! Suc k)) g
                ∧ Ltake-eq (dom f − snd (ob ! Suc k)) (fst (ob ! Suc k)) f)
      by (rule-tac x = ob' in exI, simp)
  qed
 qed
qed

lemma rtrancl-beta-obj-n:
 fixes f g L T
 assumes
 finite L and
 ∀ l∈dom f. ∀ s p. s ∉ L ∧ p ∉ L ∧ s ≠ p
            ⟶ (∃ t''. ((the(f l))[Fvar s, Fvar p]) →β* t''
                  ∧ the(g l) = σ[s,p]t'') and
 dom f = dom g and ∀ l∈dom f. body (the(f l))
 shows Obj f T →β* Obj g T
proof (cases f = Map.empty)
 case True with ‹dom f = dom g› have {} = dom g by simp
 from ‹f = Map.empty› empty-dom[OF this] show ?thesis by simp
next
 from rtrancl-beta-obj-lem00[OF assms]
 obtain ob :: ((Label −~> sterm) × (Label set)) list
   where
   length ob = card(dom f) + 1 and
   ∀ obi. obi ∈ set ob ⟶ dom (fst obi) = dom f ∧ snd obi ⊆ dom f and
   fst(ob!0) = f and
   card (snd(ob!card(dom f))) = card(dom f) and
   Obj (fst(ob!0)) T →β* Obj (fst(ob!card(dom f))) T and
   Ltake-eq (snd(ob!card(dom f))) (fst(ob!card(dom f))) g
   by blast
 from ‹length ob = card (dom f) + 1› have (ob!card(dom f)) ∈ set ob by auto
 with ‹∀ obi. obi ∈ set ob ⟶ dom (fst obi) = dom f ∧ snd obi ⊆ dom f›
 have dom (fst(ob!card(dom f))) = dom f and snd(ob!card(dom f)) ⊆ dom f
   by blast+

 {
   fix l :: Label
   from
     ‹snd(ob!card(dom f)) ⊆ dom f› ‹card (snd(ob!card(dom f))) = card(dom f)›
     Ltake-eq-dom
```

```
    have snd(ob!card(dom f)) = dom f by blast
    with ‹Ltake-eq (snd(ob!card (dom f))) (fst(ob!card (dom f))) g›
    have fst(ob!card(dom f)) l = g l
    proof (cases l ∈ dom f, simp-all add: Ltake-eq-def)
      assume l ∉ dom f
      with ‹dom f = dom g› ‹dom (fst(ob!card(dom f))) = dom f›
      show fst(ob!card(dom f)) l = g l by auto
    qed
  }
  with ext have fst(ob!card(dom f)) = g by auto
  with ‹fst(ob!0) = f› ‹Obj (fst(ob!0)) T →β* Obj (fst(ob!card (dom f))) T›
  show Obj f T →β* Obj g T by simp
qed
```

3.4 Size of sterms

```
definition fsize0 :: (Label −~> sterm) ⇒ (sterm ⇒ nat) ⇒ nat where
  fsize0 f sts =
    foldl (+) 0 (map sts (Finite-Set.fold (λx z. z@[THE y. Some y = f x]) [] (dom
f)))

primrec
 ssize        :: sterm ⇒ nat
and
 ssize-option :: sterm option ⇒ nat
where
  ssize-Bvar : ssize (Bvar b) = 0
| ssize-Fvar : ssize (Fvar x) = 0
| ssize-Call : ssize (Call a l b) = (ssize a) + (ssize b) + Suc 0
| ssize-Upd  : ssize (Upd a l b) = (ssize a) + (ssize b) + Suc 0
| ssize-Obj  : ssize (Obj f T) = Finite-Set.fold (λx y. y + ssize-option (f x)) (Suc
0) (dom f)
| ssize-None : ssize-option (None) = 0
| ssize-Some : ssize-option (Some y) = ssize y + Suc 0


interpretation comp-fun-commute (λx y::nat. y + (f x))
  by (unfold comp-fun-commute-def, force)

lemma SizeOfObjectPos: ssize (Obj (f::Label −~> sterm) T) > 0
proof (simp)
  from finite-dom-fmap have finite (dom f) by auto
  thus 0 < Finite-Set.fold (λx y. y + ssize-option (f x)) (Suc 0) (dom f)
  proof (induct)
    case empty thus ?case by simp
  next
    case (insert A a) thus ?case by auto
  qed
qed
```

end

4 Parallel reduction

theory *ParRed* **imports** *HOL−Proofs−Lambda.Commutation Sigma* **begin**

4.1 Parallel reduction

inductive *par-beta* :: [*sterm*,*sterm*] ⇒ *bool* (**infixl** ‹⇒$_\beta$› *50*)
where
pbeta-Fvar[*simp*,*intro*!]: *Fvar x* ⇒$_\beta$ *Fvar x*
| *pbeta-Obj*[*simp*,*intro*!] :
⟦ *dom f′* = *dom f*; *finite L*;
∀ *l*∈*dom f*. ∀ *s p*. *s* ∉ *L* ∧ *p* ∉ *L* ∧ *s* ≠ *p*
⟶ (∃ *t*. (*the*(*f l*)$^{[Fvar\ s,\ Fvar\ p]}$) ⇒$_\beta$ *t*
∧ *the*(*f′ l*) = *σ*[*s*,*p*] *t*);
∀ *l*∈*dom f*. *body* (*the*(*f l*)) ⟧ ⟹ *Obj f T* ⇒$_\beta$ *Obj f′ T*
| *pbeta-Upd*[*simp*,*intro*!] :
⟦ *t* ⇒$_\beta$ *t′*; *lc t*; *finite L*;
∀ *s p*. *s* ∉ *L* ∧ *p* ∉ *L* ∧ *s* ≠ *p*
⟶ (∃ *t″*. (*u*$^{[Fvar\ s,\ Fvar\ p]}$) ⇒$_\beta$ *t″* ∧ *u′* = *σ*[*s*,*p*] *t″*);
body u ⟧ ⟹ *Upd t l u* ⇒$_\beta$ *Upd t′ l u′*
| *pbeta-Upd′*[*simp*,*intro*!]:
⟦ *Obj f T* ⇒$_\beta$ *Obj f′ T*; *finite L*;
∀ *s p*. *s* ∉ *L* ∧ *p* ∉ *L* ∧ *s* ≠ *p*
⟶ (∃ *t″*. (*t*$^{[Fvar\ s,\ Fvar\ p]}$) ⇒$_\beta$ *t″* ∧ *t′* = *σ*[*s*,*p*] *t″*); *l* ∈ *dom f*;
lc (*Obj f T*); *body t* ⟧ ⟹ (*Upd* (*Obj f T*) *l t*) ⇒$_\beta$ (*Obj* (*f′*(*l* ↦ *t′*)) *T*)
| *pbeta-Call*[*simp*,*intro*!]:
⟦ *t* ⇒$_\beta$ *t′*; *u* ⇒$_\beta$ *u′*; *lc t*; *lc u* ⟧
⟹ *Call t l u* ⇒$_\beta$ *Call t′ l u′*
| *pbeta-beta*[*simp*,*intro*!]:
⟦ *Obj f T* ⇒$_\beta$ *Obj f′ T*; *l* ∈ *dom f*; *p* ⇒$_\beta$ *p′*; *lc* (*Obj f T*); *lc p* ⟧
⟹ *Call* (*Obj f T*) *l p* ⇒$_\beta$ (*the*(*f′ l*)$^{[(Obj\ f'\ T),\ p']}$)

inductive-cases *par-beta-cases* [*elim*!]:
Fvar x ⇒$_\beta$ *t*
Obj f T ⇒$_\beta$ *t*
Call f l p ⇒$_\beta$ *t*
Upd f l t ⇒$_\beta$ *u*

abbreviation
par-beta-ascii :: [*sterm*, *sterm*] => *bool* (**infixl** ‹=>› *50*) **where**
t => *u* == *par-beta t u*

lemma *Obj-par-red*[*consumes 1*, *case-names obj*]:

⟦ *Obj f T* $\Rightarrow_\beta$ *z*;
  $\bigwedge$*lz*. ⟦ *dom lz* = *dom f*; *z* = *Obj lz T*⟧ $\Longrightarrow$ *Q* ⟧ $\Longrightarrow$ *Q*
by (*rule par-beta-cases*(*2*), *assumption*, *auto*)

lemma *Upd-par-red*[*consumes 1*, *case-names upd obj*]:
fixes *t l u z*
assumes
Upd t l u $\Rightarrow_\beta$ *z* **and**
$\bigwedge$*t' u' L*. ⟦ *t* $\Rightarrow_\beta$ *t'*; *finite L*;
  $\forall$ *s p*. *s* $\notin$ *L* $\wedge$ *p* $\notin$ *L* $\wedge$ *s* $\neq$ *p*
   $\longrightarrow$ ($\exists$ *t''*. (*u*[*Fvar s*, *Fvar p*]) $\Rightarrow_\beta$ *t''*
    $\wedge$ *u'* = σ[*s*,*p*]*t''*);
  z = *Upd t' l u'* ⟧ $\Longrightarrow$ *Q* **and**
$\bigwedge$*f f' T u' L*. ⟦ *l* $\in$ *dom f*; *Obj f T* = *t*; *Obj f T* $\Rightarrow_\beta$ *Obj f' T*;
  finite L;
  $\forall$ *s p*. *s* $\notin$ *L* $\wedge$ *p* $\notin$ *L* $\wedge$ *s* $\neq$ *p*
   $\longrightarrow$ ($\exists$ *t''*. (*u*[*Fvar s*, *Fvar p*]) $\Rightarrow_\beta$ *t''*
    $\wedge$ *u'* = σ[*s*,*p*]*t''*);
  z = *Obj* (*f'*(*l* $\mapsto$ *u'*)) *T* ⟧ $\Longrightarrow$ *Q*
shows *Q*
using *assms*
proof (*cases rule*: *par-beta.cases*)
case *pbeta-Upd* **thus** *?thesis* **using** *assms*(*2*) **by** *force*
next
case *pbeta-Upd'*
from *this*(*1*−*2*) *this*(*5*−*6*) *assms*(*3*)[*OF - - this*(*3*−*4*)]
show *?thesis* **by** *force*
qed

lemma *Call-par-red*[*consumes 1*, *case-names call beta*]:
fixes *s l u z*
assumes
Call s l u $\Rightarrow_\beta$ *z* **and**
$\bigwedge$*t u'*. ⟦ *s* $\Rightarrow_\beta$ *t*; *u* $\Rightarrow_\beta$ *u'*; *z* = *Call t l u'* ⟧
$\Longrightarrow$ *Q*
$\bigwedge$*f f' T u'*. ⟦ *Obj f T* = *s*; *Obj f T* $\Rightarrow_\beta$ *Obj f' T*;
  l $\in$ *dom f'*; *u* $\Rightarrow_\beta$ *u'*;
  z = (*the* (*f' l*)[*Obj f' T*, *u'*]) ⟧ $\Longrightarrow$ *Q*
shows *Q*
using *assms*
proof (*cases rule*: *par-beta.cases*)
case *pbeta-Call* **thus** *?thesis* **using** *assms*(*2*) **by** *force*
next
case *pbeta-beta*
from *this*(*1*−*5*) *assms*(*3*)[*OF - this*(*3*)]
show *?thesis* **by** *force*
qed

lemma *pbeta-induct*[*consumes 1*, *case-names Fvar Call Upd Upd' Obj beta Bnd*]:

fixes
$t :: sterm$ **and** $t' :: sterm$ **and**
$P1 :: sterm \Rightarrow sterm \Rightarrow bool$ **and** $P2 :: sterm \Rightarrow sterm \Rightarrow bool$
assumes
$t \Rightarrow_\beta t'$ **and**
$\bigwedge x.\ P1\ (Fvar\ x)\ (Fvar\ x)$ **and**
$\bigwedge t\ t'\ l\ u\ u'.\ [\![\ t \Rightarrow_\beta t';\ P1\ t\ t';\ lc\ t;\ u \Rightarrow_\beta u';\ P1\ u\ u';\ lc\ u\ ]\!]$
$\quad \Longrightarrow P1\ (Call\ t\ l\ u)\ (Call\ t'\ l\ u')$ **and**
$\bigwedge t\ t'\ l\ u\ u'.\ [\![\ t \Rightarrow_\beta t';\ P1\ t\ t';\ lc\ t;\ P2\ u\ u';\ body\ u\ ]\!]$
$\quad \Longrightarrow P1\ (Upd\ t\ l\ u)\ (Upd\ t'\ l\ u')$ **and**
$\bigwedge f\ f'\ T\ t\ t'\ l.\ [\![\ Obj\ f\ T \Rightarrow_\beta Obj\ f'\ T;\ P1\ (Obj\ f\ T)\ (Obj\ f'\ T);$
$\qquad P2\ t\ t';\ l \in dom\ f;\ lc\ (Obj\ f\ T);\ body\ t\ ]\!]$
$\quad \Longrightarrow P1\ (Upd\ (Obj\ f\ T)\ l\ t)\ (Obj\ (f'(l \mapsto t'))\ T)$ **and**
$\bigwedge f\ f'\ T.\ [\![\ dom\ f' = dom\ f;\ \forall l{\in}dom\ f.\ body\ (the(f\ l));$
$\qquad \forall l{\in}dom\ f.\ P2\ (the(f\ l))\ (the(f'\ l))\ ]\!]$
$\quad \Longrightarrow P1\ (Obj\ f\ T)\ (Obj\ f'\ T)$ **and**
$\bigwedge f\ f'\ T\ l\ p\ p'.\ [\![\ Obj\ f\ T \Rightarrow_\beta Obj\ f'\ T;\ P1\ (Obj\ f\ T)\ (Obj\ f'\ T);\ lc\ (Obj\ f\ T);$
$\qquad l \in dom\ f;\ p \Rightarrow_\beta p';\ P1\ p\ p';\ lc\ p\ ]\!]$
$\quad \Longrightarrow P1\ (Call\ (Obj\ f\ T)\ l\ p)\ (the(f'\ l)^{[Obj\ f'\ T,\ p']})$ **and**
$\bigwedge L\ t\ t'.$
$\quad [\![\ finite\ L;$
$\quad\ \forall s\ p.\ s \notin L \wedge p \notin L \wedge s \neq p$
$\quad\ \longrightarrow (\exists t''.\ t^{[Fvar\ s, Fvar\ p]} \Rightarrow_\beta t''$
$\qquad\qquad \wedge\ P1\ (t^{[Fvar\ s, Fvar\ p]})\ t'' \wedge t' = \sigma[s,p]\ t'')\ ]\!]$
$\quad \Longrightarrow P2\ t\ t'$
shows $P1\ t\ t'$
by (*induct rule*: *par-beta.induct*[*OF assms*(*1*)], *auto simp*: *assms*)

4.2 Preservation

lemma *par-beta-lc*[*simp*]:
fixes $t\ t'$
assumes $t \Rightarrow_\beta t'$
shows $lc\ t \wedge lc\ t'$
using *assms*
proof
(*induct*
taking: $\lambda t\ t'.\ body\ t'$
rule: *pbeta-induct*)
case *Fvar* **thus** *?case* **by** *simp*
next
case *Call* **thus** *?case* **by** *simp*
next
case *Upd* **thus** *?case* **by** (*simp add*: *lc-upd*)
next
case *Upd'* **thus** *?case* **by** (*simp add*: *lc-upd lc-obj*)
next
case *Obj* **thus** *?case* **by** (*simp add*: *lc-obj*)
next

```
  case (beta f f' T l p p') thus ?case
    by (clarify, simp add: lc-obj body-lc[of the(f' l) Obj f' T p'])
next
  case (Bnd L t t') note cof = this(2)
  from exFresh-s-p-cof[OF ‹finite L›]
  obtain s p where sp: s ∉ L ∧ p ∉ L ∧ s ≠ p by auto
  with cof obtain t'' where lc t'' and t' = σ[s,p] t'' by blast
  with lc-body[of t'' s p] sp show body t' by force
qed

lemma par-beta-preserves-FV[simp, rule-format]:
  fixes t t' x
  assumes t ⇒β t'
  shows x ∉ FV t ⟶ x ∉ FV t'
using assms
proof
  (induct
    taking: λt t'. x ∉ FV t ⟶ x ∉ FV t'
    rule: pbeta-induct)
  case Fvar thus ?case by simp
next
  case Call thus ?case by simp
next
  case Upd thus ?case by simp
next
  case Upd' thus ?case by simp
next
  case Obj thus ?case by (simp add: FV-option-lem)
next
  case (beta f f' T l p p') thus ?case
  proof (intro strip)
    assume x ∉ FV (Call (Obj f T) l p)
    with
      ‹x ∉ FV (Obj f T) ⟶ x ∉ FV (Obj f' T)›
      ‹x ∉ FV p ⟶ x ∉ FV p'›
    have obj': x ∉ FV (Obj f' T) and p': x ∉ FV p'
      by auto
    from ‹l ∈ dom f› ‹Obj f T ⇒β Obj f' T› have l ∈ dom f'
      by auto
    with
      obj' p' FV-option-lem[of f']
      contra-subsetD[OF sopen-FV[of 0 Obj f' T p' the(f' l)]]
    show x ∉ FV (the (f' l)[Obj f' T,p']) by (auto simp: openz-def)
  qed
next
  case (Bnd L t t') note cof = this(2)
  from ‹finite L› exFresh-s-p-cof[of L ∪ {x}]
  obtain s p where
    s ∉ Land p ∉ L and s ≠ p and
```

```
    x ∉ FV (Fvar s) and x ∉ FV (Fvar p)
    by auto
  with cof obtain t'' where
    tt'': x ∉ FV (t[Fvar s, Fvar p]) ⟶ x ∉ FV t'' and
    t' = σ[s,p] t''
    by auto
  show ?case
  proof (intro strip)
    assume x ∉ FV t
    with
      tt'' ‹x ∉ FV (Fvar s)› ‹x ∉ FV (Fvar p)›
      contra-subsetD[OF sopen-FV[of 0 Fvar s Fvar p t]]
      sclose-subset-FV[of 0 s p t''] ‹t' = σ[s,p] t''›
    show x ∉ FV t' by (auto simp: openz-def closez-def)
  qed
qed

lemma par-beta-body[simp]:
  ⟦ finite L;
    ∀ s p. s ∉ L ∧ p ∉ L ∧ s ≠ p
      ⟶ (∃ t''. t[Fvar s,Fvar p] ⇒β t'' ∧ t' = σ[s,p] t'') ⟧
  ⟹ body t ∧ body t'
proof (intro conjI)
  fix L :: fVariable set and t :: sterm and t' :: sterm
  assume finite L hence finite (L ∪ FV t) by simp
  from exFresh-s-p-cof[OF this]
  obtain s p where sp: s ∉ L ∪ FV t ∧ p ∉ L ∪ FV t ∧ s ≠ p by auto
  hence s ∉ FV t and p ∉ FV t and s ≠ p by auto

  assume
    ∀ s p. s ∉ L ∧ p ∉ L ∧ s ≠ p
      ⟶ (∃ t''. t[Fvar s,Fvar p] => t'' ∧ t' = σ[s,p] t'')
  with sp obtain t'' where t[Fvar s,Fvar p] ⇒β t'' and t' = σ[s,p] t''
    by blast

  from par-beta-lc[OF this(1)] have lc (t[Fvar s,Fvar p]) and lc t''
    by auto

  from
    lc-body[OF this(1) ‹s ≠ p›]
    sclose-sopen-eq-t[OF ‹s ∉ FV t› ‹p ∉ FV t› ‹s ≠ p›]
  show body t
    by (simp add: closez-def openz-def)

  from lc-body[OF ‹lc t''› ‹s ≠ p›] ‹t' = σ[s,p] t''› show body t' by simp
qed
```

4.3 Miscellaneous properties of par_beta

lemma *Fvar-pbeta* [*simp*]: $(Fvar\ x \Rightarrow_\beta t) = (t = Fvar\ x)$ **by** *auto*
lemma *Obj-pbeta*: $Obj\ f\ T \Rightarrow_\beta Obj\ f'\ T$
$\Longrightarrow dom\ f' = dom\ f$
$\wedge\ (\exists L.\ finite\ L$
$\wedge\ (\forall l{\in}dom\ f.\ \forall s\ p.\ s \notin L \wedge p \notin L \wedge s \neq p$
$\longrightarrow\ (\exists t.\ (the(f\ l)^{[Fvar\ s,\ Fvar\ p]}) \Rightarrow_\beta t$
$\wedge\ the(f'\ l) = \sigma[s,p]t)))$
$\wedge\ (\forall l{\in}dom\ f.\ body\ (the(f\ l)))$
by (*rule par-beta-cases*(*2*), *assumption*, *auto*)

lemma *Obj-pbeta-subst*:
$[\![\ finite\ L;$
$\forall s\ p.\ s \notin L \wedge p \notin L \wedge s \neq p$
$\longrightarrow\ (\exists t''.\ (t^{[Fvar\ s,\ Fvar\ p]}) \Rightarrow_\beta t'' \wedge t' = \sigma[s,p]\ t'');$
$Obj\ f\ T \Rightarrow_\beta Obj\ f'\ T;\ lc\ (Obj\ f\ T);\ body\ t\ ]\!]$
$\Longrightarrow Obj\ (f(l \mapsto t))\ T \Rightarrow_\beta Obj\ (f'(l \mapsto t'))\ T$
proof −
fix $L\ f\ f'\ T\ l\ t\ t'$
assume $Obj\ f\ T \Rightarrow_\beta Obj\ f'\ T$ **from** *Obj-pbeta*[*OF this*]
have
dom: $dom\ (f'(l \mapsto t')) = dom\ (f(l \mapsto t))$ **and**
exL: $\exists L.\ finite\ L$
$\wedge\ (\forall l{\in}dom\ f.\ \forall s\ p.\ s \notin L \wedge p \notin L \wedge s \neq p$
$\longrightarrow\ (\exists t.\ the\ (f\ l)^{[Fvar\ s,Fvar\ p]} \Rightarrow_\beta t$
$\wedge\ the\ (f'\ l) = \sigma[s,p]\ t))$ **and**
bodyf: $\forall l{\in}dom\ f.\ body\ (the\ (f\ l))$
by *auto*

assume $body\ t$ **with** *bodyf*
have *body*: $\forall l'{\in}dom\ (f(l \mapsto t)).\ body\ (the\ ((f(l \mapsto t))\ l'))$
by *auto*

assume
$finite\ L$ **and**
$\forall s\ p.\ s \notin L \wedge p \notin L \wedge s \neq p$
$\longrightarrow\ (\exists t''.\ t^{[Fvar\ s,Fvar\ p]} \Rightarrow_\beta t'' \wedge t' = \sigma[s,p]\ t'')$
with *exL*
obtain L' **where**
$finite\ (L' \cup L)$ **and**
$\forall l'{\in}dom\ (f(l \mapsto t)).\ \forall s\ p.\ s \notin L' \cup L \wedge p \notin L' \cup L \wedge s \neq p$
$\longrightarrow\ (\exists t''.\ the\ ((f(l \mapsto t))\ l')^{[Fvar\ s,Fvar\ p]} \Rightarrow_\beta t''$
$\wedge\ the\ ((f'(l \mapsto t'))\ l') = \sigma[s,p]\ t'')$
by *auto*
from *par-beta.pbeta-Obj*[*OF dom this body*]
show $Obj\ (f(l \mapsto t))\ T \Rightarrow_\beta Obj\ (f'(l \mapsto t'))\ T$
by *assumption*
qed

lemma *Upd-pbeta*: *Upd t l u* $\Rightarrow_\beta$ *Upd t′ l u′*
$\Longrightarrow$ $t \Rightarrow_\beta t'$
$\wedge$ ($\exists$ *L*. *finite L*
$\wedge$ ($\forall$ *s p*. $s \notin L \wedge p \notin L \wedge s \neq p$
$\longrightarrow$ ($\exists$ *t″*. ($u^{[Fvar\ s,\ Fvar\ p]}$) $\Rightarrow_\beta$ $t'' \wedge u' = \sigma[s,p]t''$)))
$\wedge$ *lc t* $\wedge$ *body u*
by (*rule par-beta-cases*(*4*), *assumption*, *auto*)

lemma *par-beta-refl*:
fixes *t*
assumes *lc t*
shows $t \Rightarrow_\beta t$
using *assms*
proof −
define *pred-cof*
where *pred-cof L t* $\longleftrightarrow$
($\forall$ *s p*. $s \notin L \wedge p \notin L \wedge s \neq p \longrightarrow (\exists\, t'.\ (t^{[Fvar\ s,\ Fvar\ p]}) \Rightarrow_\beta t' \wedge t = \sigma[s,p]$
t′))
for *L t*
from *assms* **show** *?thesis*
proof
(*induct*
taking: $\lambda t.$ *body t* $\wedge$ ($\exists$ *L*. *finite L* $\wedge$ *pred-cof L t*)
rule: *lc-induct*)
case *Fvar* **thus** *?case* **by** *simp*
next
case *Call* **thus** *?case* **by** *simp*
next
case *Upd* **thus** *?case*
unfolding *pred-cof-def*
by *auto*
next
case (*Obj f T*) **note** *pred* = *this*
define *pred-fl* **where** *pred-fl s p b l* $\longleftrightarrow$ ($\exists\, t'.$ (*the* $b^{[Fvar\ s,\ Fvar\ p]}$) $\Rightarrow_\beta$ $t' \wedge$
the b = $\sigma[s,p]t'$)
for *s p b* **and** *l* :: *Label*

from *fmap-ex-cof*[*of f pred-fl*] *pred*
obtain *L* **where**
finite L **and** $\forall\, l \in dom\ f.$ *body* (*the*(*f l*))
$\wedge$ ($\forall$ *s p*. $s \notin L \wedge p \notin L \wedge s \neq p \longrightarrow$ *pred-fl s p* (*f l*) *l*)
unfolding *pred-cof-def pred-fl-def*
by *auto*
thus *Obj f T* $\Rightarrow_\beta$ *Obj f T*
unfolding *pred-fl-def*
by *auto*
next
case (*Bnd L t*) **note** *pred* = *this*(*2*)

```
    with ‹finite L› show ?case
    proof
      (auto simp: body-def, unfold pred-cof-def,
        rule-tac x = L ∪ FV t in exI, simp, clarify)
      fix s p assume
        s ∉ L and p ∉ L and s ≠ p and
        s ∉ FV t and p ∉ FV t
      from
        this(1−3) pred
        sclose-sopen-eq-t[OF this(4−5) this(3)]
      show ∃ t'. t^[Fvar s,Fvar p] ⇒β t' ∧ t = σ[s,p] t'
        by (rule-tac x = t^[Fvar s,Fvar p] in exI, simp add: openz-def closez-def)
    qed
  qed
qed

lemma par-beta-body-refl:
  fixes u
  assumes body u
  shows ∃ L. finite L ∧ (∀ s p. s ∉ L ∧ p ∉ L ∧ s ≠ p
                        ⟶ (∃ t'. (u^[Fvar s, Fvar p]) ⇒β t' ∧ u = σ[s,p] t'))
proof (rule-tac x = FV u in exI, simp, clarify)
  fix s p assume s ∉ FV u and p ∉ FV u and s ≠ p
  from
    par-beta-refl[OF body-lc[OF assms lc-Fvar[of s] lc-Fvar[of p]]]
    sclose-sopen-eq-t[OF this]
  show ∃ t'. (u^[Fvar s, Fvar p]) ⇒β t' ∧ u = σ[s,p] t'
    by (rule-tac x = u^[Fvar s, Fvar p] in exI, simp add: openz-def closez-def)
qed

lemma par-beta-ssubst[rule-format]:
  fixes t t'
  assumes t ⇒β t'
  shows ∀ x v v'. v ⇒β v' ⟶ [x → v] t ⇒β [x → v'] t'
proof −
  define pred-cof
    where pred-cof L t t' ⟷
      (∀ s p. s ∉ L ∧ p ∉ L ∧ s ≠ p ⟶ (∃ t''. t^[Fvar s, Fvar p] ⇒β t'' ∧ t' = σ[s,p]
t''))
    for L t t'
  {
    fix x v v' t t'
    assume
      v ⇒β v' and
      ∀ x v v'. v ⇒β v' ⟶ (∃ L. finite L ∧ pred-cof L ([x → v] t) ([x → v'] t'))
    hence
      ∃ L. finite L ∧ pred-cof L ([x → v] t) ([x → v'] t')
        by auto
```

```
}note Lex = this

{
 fix x v l and f :: Label ⇒ sterm option
 assume l ∈ dom f hence l ∈ dom (λl. ssubst-option x v (f l))
   by simp
}note domssubst = this
{
 fix x v l T and f :: Label ⇒ sterm option
 assume lc (Obj f T) and lc v from ssubst-preserves-lc[OF this]
 have obj: lc (Obj (λl. ssubst-option x v (f l)) T) by simp
}note lcobj = this

from assms show ?thesis
proof
 (induct
   taking: λt t'. ∀ x v v'. v ⇒β v'
                  ⟶ (∃ L. finite L
                        ∧ pred-cof L ([x → v] t) ([x → v'] t'))
   rule: pbeta-induct)
 case Fvar thus ?case by simp
next
 case Call thus ?case by simp
next
 case (Upd t t' l u u') note pred-t = this(2) and pred-u = this(4)
 show ?case
 proof (intro strip)
   fix x v v' assume v ⇒β v'
   from Lex[OF this pred-u]
   obtain L where
     finite L and pred-cof L ([x → v] u) ([x → v'] u')
     by auto
   with
     ssubst-preserves-lc[of t v x]
     ssubst-preserves-body[of u v x]
     ‹lc t› par-beta-lc[OF ‹v ⇒β v'›] ‹body u›
     ‹v ⇒β v'› pred-t
   show [x → v] Upd t l u ⇒β [x → v'] Upd t' l u'
     unfolding pred-cof-def
     by auto
 qed
next
 case (Upd' f f' T t t' l)
 note pred-obj = this(2) and pred-t = this(3)
 show ?case
 proof (intro strip)
   from ‹Obj f T ⇒β Obj f' T› ‹l ∈ dom f› have l ∈ dom f' by auto
   fix x v v' assume v ⇒β v'
   with
```

```
      domssubst[OF ‹l ∈ dom f›]
      ssubst-preserves-lc[of Obj f T v x]
      ssubst-preserves-body[of t v x]
      ‹lc (Obj f T)› par-beta-lc[OF ‹v ⇒β v'›] ‹body t›
      pred-obj
    have
      [x → v] Obj f T ⇒β [x → v'] Obj f' T and
      lc ([x → v] Obj f T) and body ([x → v] t)
      by auto
    note lem =
      pbeta-Upd'[OF this(1)[simplified] - -
                   domssubst[OF ‹l ∈ dom f›]
                   this(2)[simplified] this(3)]

    from Lex[OF ‹v ⇒β v'› pred-t]
    obtain L where
      finite L and pred-cof L ([x → v] t) ([x → v'] t')
      by auto
    with lem[of L [x → v'] t'] ssubstoption-insert[OF ‹l ∈ dom f'›]
    show [x → v] Upd (Obj f T) l t ⇒β [x → v'] Obj (f'(l ↦ t')) T
      unfolding pred-cof-def
      by auto
  qed
next
  case (beta f f' T l p p')
  note pred-obj = this(2) and pred-p = this(6)
  show ?case
  proof (intro strip)
    fix x v v' assume v ⇒β v'
    from
      par-beta-lc[OF this]
      ssubst-preserves-lc[OF ‹lc p›]
    have lc v and lc v' and lc ([x → v] p) by auto
    note lem =
      pbeta-beta[OF - domssubst[OF ‹l ∈ dom f›] -
                   lcobj[OF ‹lc (Obj f T)› this(1)] this(3)]
    from ‹Obj f T ⇒β Obj f' T› have dom f = dom f' by auto
    with ‹l ∈ dom f› have the (ssubst-option x v' (f' l)) = [x → v'] the (f' l)
      by auto
    with
      lem[of x λl. ssubst-option x v' (f' l) [x → v'] p']
      ‹v ⇒β v'› pred-obj pred-p
      ssubst-openz-distrib[OF ‹lc v'›]
    show
      [x → v] Call (Obj f T) l p ⇒β [x → v'] (the (f' l)^[Obj f' T, p'])
      by simp
  qed
next
  case (Obj f f' T) note pred = fmap-ball-all3[OF this(1) this(3)]
```

```
  show ?case
  proof (intro strip)
    fix x v v'
    define pred-bnd
      where pred-bnd s p b b' l ⟷
        (∃ t''. [x → v] the b^[Fvar s,Fvar p] ⇒β t'' ∧ [x → v'] the b' = σ[s,p] t'')
      for s p b b' and l::Label
    assume v ⇒β v'
    with pred ‹dom f' = dom f› fmap-ex-cof2[of f' f pred-bnd]
    obtain L where
      finite L and
      predf: ∀ l∈dom f. pred-cof L ([x → v] the (f l)) ([x → v'] the (f' l))
      unfolding pred-cof-def pred-bnd-def
      by auto

    have ∀ l∈dom (λl. ssubst-option x v (f l)). body (the (ssubst-option x v (f l)))
    proof (intro strip, simp)
      fix l' :: Label assume l' ∈ dom f
      with ‹∀ l∈dom f. body (the(f l))› have body (the (f l')) by blast
      note ssubst-preserves-body[OF this]
      from
        this[of v x] par-beta-lc[OF ‹v ⇒β v'›]
        ‹l' ∈ dom f› ssubst-option-lem[of f x v]
      show body (the (ssubst-option x v (f l'))) by auto
    qed
    note intro = pbeta-Obj[OF - ‹finite L› - this]
    from
      predf
      ssubst-option-lem[of f x v]
      ssubst-option-lem[of f' x v'] ‹dom f' = dom f›
      dom-ssubstoption-lem[of x v f]
      dom-ssubstoption-lem[of x v' f']
    show [x → v] Obj f T ⇒β [x → v'] Obj f' T
      unfolding pred-cof-def
      by (simp, intro intro[of (λl. ssubst-option x v' (f' l)) T], auto)
  qed
next
  case (Bnd L t t') note pred = this(2)
  show ?case
  proof (intro strip)
    fix x v v' assume v ⇒β v'
    from ‹finite L›
    show ∃ L. finite L ∧ pred-cof L ([x → v] t) ([x → v'] t')
    proof (rule-tac x = L ∪ {x} ∪ FV v' in exI,
        unfold pred-cof-def, auto)
      fix s p assume s ∉ L and p ∉ L and s ≠ p
      with pred
      obtain t'' where
        t^[Fvar s,Fvar p] ⇒β t'' and
```

```
      ∀ x v v'. v ⇒β v' ⟶ [x → v] (t^[Fvar s,Fvar p]) ⇒β [x → v'] t'' and
      t' = σ[s,p] t''
      by blast
    from this(2) ‹v ⇒β v'›
    have ssubst-pbeta: [x → v] (t^[Fvar s,Fvar p]) ⇒β [x → v'] t'' by blast

    assume s ≠ x and p ≠ x
    hence x ∉ FV (Fvar s) and x ∉ FV (Fvar p) by auto
    from
      ssubst-pbeta
      par-beta-lc[OF ‹v ⇒β v'›] ssubst-sopen-commute[OF - this]
    have [x → v] t^[Fvar s,Fvar p] ⇒β [x → v'] t'' by (simp add: openz-def)
    moreover
    assume s ∉ FV v' and p ∉ FV v'
    from
      ssubst-sclose-commute[OF this not-sym[OF ‹s ≠ x›]
                                not-sym[OF ‹p ≠ x›]]
      ‹t' = σ[s,p] t''›
    have [x → v'] t' = σ[s,p] [x → v'] t'' by (simp add: closez-def)
    ultimately
    show ∃ t''. [x → v] t^[Fvar s,Fvar p] ⇒β t'' ∧ [x → v'] t' = σ[s,p] t''
      by (rule-tac x = [x → v'] t'' in exI, simp)
  qed
 qed
qed

lemma renaming-par-beta: t ⇒β t' ⟹ [s → Fvar sa] t ⇒β [s → Fvar sa] t'
  by (erule par-beta-ssubst, simp+)

lemma par-beta-beta:
  fixes l f f' u u'
  assumes
  l ∈ dom f and Obj f T ⇒β Obj f' T and u ⇒β u' and lc (Obj f T) and lc u
  shows (the(f l)^[Obj f T, u]) ⇒β (the(f' l)^[Obj f' T, u'])
proof −
  from Obj-pbeta[OF ‹Obj f T ⇒β Obj f' T›]
  obtain L where
    dom f = dom f' and
    finite L and
    pred-sp: ∀ l∈dom f.∀ s p. s ∉ L ∧ p ∉ L ∧ s ≠ p
                        ⟶ (∃ t''. the (f l)^[Fvar s,Fvar p] ⇒β t''
                              ∧ the (f' l) = σ[s,p] t'') and
    ∀ l∈dom f. body (the (f l))
    by auto

  from this(2) finite-FV[of Obj f T] have fin: finite (L ∪ FV (Obj f T) ∪ FV u)
by simp
```

from *exFresh-s-p-cof*[*OF this*]
obtain *s p* **where**
sp: $s \notin L \cup FV\ (Obj\ f\ T) \cup FV\ u \wedge p \notin L \cup FV\ (Obj\ f\ T) \cup FV\ u \wedge s \neq p$
by *auto*
with ‹$l \in dom\ f$› **obtain** t'' **where**
$the\ (f\ l)^{[Fvar\ s,\ Fvar\ p]} \Rightarrow_\beta t''$ **and** $the\ (f'\ l) = \sigma[s,p]\ t''$
using *pred-sp* **by** *blast*
from *par-beta-lc*[*OF this*(*1*)] **have** *lc* t'' **by** *simp*
from
sopen-sclose-eq-t[*OF this*]
‹$the\ (f\ l)^{[Fvar\ s,\ Fvar\ p]} \Rightarrow_\beta t''$› ‹$the(f'\ l) = \sigma[s,p]\ t''$›
have $the\ (f\ l)^{[Fvar\ s,\ Fvar\ p]} \Rightarrow_\beta (the\ (f'\ l)^{[Fvar\ s,\ Fvar\ p]})$
by (*simp add*: *openz-def closez-def*)
from *par-beta-ssubst*[*OF this*] ‹$u \Rightarrow_\beta u'$›
have $[p \to u]\ (the\ (f\ l)^{[Fvar\ s,\ Fvar\ p]}) \Rightarrow_\beta [p \to u']\ (the\ (f'\ l)^{[Fvar\ s,\ Fvar\ p]})$
by *simp*
note *par-beta-ssubst*[*OF this* ‹$Obj\ f\ T \Rightarrow_\beta Obj\ f'\ T$›]
moreover
from ‹$l \in dom\ f$› *sp*
have $s \notin FV\ (the(f\ l))$ **and** $p \notin FV\ (the(f\ l))$ **and** $s \neq p$ **and** $s \notin FV\ u$
by *force*+
note *ssubst-intro*[*OF this*]
moreover
from ‹$l \in dom\ f$› ‹$dom\ f = dom\ f'$›
have $l \in dom\ f'$ **by** *force*
with
par-beta-preserves-FV[*OF* ‹$Obj\ f\ T \Rightarrow_\beta Obj\ f'\ T$›]
par-beta-preserves-FV[*OF* ‹$u \Rightarrow_\beta u'$›] *sp FV-option-lem*[*of f'*]
have $s \notin FV\ (the\ (f'\ l))$ **and** $p \notin FV\ (the\ (f'\ l))$ **and** $s \neq p$ **and** $s \notin FV\ u'$
by *auto*
note *ssubst-intro*[*OF this*]
ultimately
show $the\ (f\ l)^{[Obj\ f\ T,\ u]} \Rightarrow_\beta (the\ (f'\ l)^{[Obj\ f'\ T,\ u']})$
by (*simp add*: *openz-def closez-def*)
qed

4.4 Inclusions

$beta \subseteq par\text{-}beta \subseteq beta\hat{}*$

lemma *beta-subset-par-beta*: $beta \leq par\text{-}beta$
proof (*clarify*)
define *pred-cof*
where *pred-cof* $L\ t\ t' \longleftrightarrow$
$(\forall s\ p.\ s \notin L \wedge p \notin L \wedge s \neq p \longrightarrow (\exists t''.\ (t^{[Fvar\ s,\ Fvar\ p]}) \Rightarrow_\beta t'' \wedge t' = \sigma[s,p]\ t''))$
for $L\ t\ t'$

fix $t\ t'$ **assume** $t \to_\beta t'$ **thus** $t \Rightarrow_\beta t'$

proof
(*induct*
taking: $\lambda t\ t'.\ body\ t \wedge body\ t' \wedge (\exists L.\ finite\ L \wedge pred\text{-}cof\ L\ t\ t')$
rule: *beta-induct*)
case *CallL* **thus** *?case* **by** (*simp add*: *par-beta-refl*)
next
case *CallR* **thus** *?case* **by** (*simp add*: *par-beta-refl*)
next
case *beta* **thus** *?case* **by** (*simp add*: *par-beta-refl*)
next
case *UpdL*
from
par-beta-lc[*OF this*(*2*)] *this*(*2*)
par-beta-body-refl[*OF this*(*3*)] *this*(*3*)
show *?case* **by** *auto*
next
case (*UpdR t t' u l*)
from *this*(*1*) **obtain** *L* **where**
finite L **and** *pred-cof L t t'* **and** *body t*
by *auto*
from
this(*2*) *pbeta-Upd*[*OF par-beta-refl*[*OF* ‹*lc u*›] ‹*lc u*› *this*(*1*) - *this*(*3*)]
show *?case*
unfolding *pred-cof-def*
by *auto*
next
case (*Upd l f T t*)
from *par-beta-body-refl*[*OF* ‹*body t*›]
obtain *L* **where**
finite L **and**
$\forall s\ p.\ s \notin L \wedge p \notin L \wedge s \neq p$
$\longrightarrow (\exists t'.\ t^{[Fvar\ s,\ Fvar\ p]} \Rightarrow_\beta t' \wedge t = \sigma[s,p]\ t')$
by *auto*
from
pbeta-Upd'[*OF par-beta-refl*[*OF* ‹*lc* (*Obj f T*)›] *this*
‹*l* ∈ *dom f*› ‹*lc* (*Obj f T*)› ‹*body t*›]
show *?case* **by** *assumption*
next
case (*Obj l f t t' T*) **note** *cof* = *this*(*2*) **and** *body* = *this*(*3*)
from *cof* **obtain** *L* **where**
body t **and** *finite L* **and** *pred-cof L t t'*
by *auto*
from *body* **have** *lc* (*Obj f T*) **by** (*simp add*: *lc-obj*)
from
Obj-pbeta-subst[*OF* ‹*finite L*› - *par-beta-refl*[*OF this*] *this* ‹*body t*›]
‹*pred-cof L t t'*›
show *?case*
unfolding *pred-cof-def*
by *auto*

```
  next
    case (Bnd L t t') note pred = this(2)
    from ‹finite L› exFresh-s-p-cof[of L ∪ FV t]
    obtain s p where
      s ∉ L and p ∉ L and s ≠ p and
      s ∉ FV t and p ∉ FV t
      by auto
    with pred obtain t'' where
      t[Fvar s, Fvar p] ⇒β t'' and t' = σ[s,p] t''
      by blast
    from
      par-beta-lc[OF this(1)] this(2) lc-body[OF - ‹s ≠ p›]
    have body σ[s,p](t[Fvar s, Fvar p]) and body t' by auto
    from this(1) sclose-sopen-eq-t[OF ‹s ∉ FV t› ‹p ∉ FV t› ‹s ≠ p›]
    have body t by (simp add: openz-def closez-def)
    with ‹body t'› ‹finite L› pred show ?case
      unfolding pred-cof-def
      by (simp, rule-tac x = L in exI, auto)
  qed
qed

lemma par-beta-subset-beta: par-beta ≤ beta^**
proof (rule predicate2I)
  define pred-cof
    where pred-cof L t t' ⟷
      (∀ s p. s ∉ L ∧ p ∉ L ∧ s ≠ p ⟶ (∃ t''. (t[Fvar s, Fvar p]) →β* t'' ∧ t' =
σ[s,p] t''))
    for L t t'

  fix x y assume x ⇒β y thus x →β* y
  proof (induct
      taking: λt t'. body t' ∧ (∃ L. finite L ∧ pred-cof L t t')
      rule: pbeta-induct)
    case Fvar thus ?case by simp
  next
    case Call thus ?case by auto
  next
    case (Upd t t' l u u')
    from this(4) obtain L where
      finite L and pred-cof L u u' by auto
    from
      this(2)
      rtrancl-beta-Upd[OF ‹t →β* t'› this(1) - ‹lc t› ‹body u›]
    show ?case
      unfolding pred-cof-def
      by simp
  next
    case (Upd' f f' T t t' l)
    from this(3) obtain L where
```

body t' **and** *finite* L **and** *pred-cof* L t t' **by** *auto*
from
this(3)
rtrancl-beta-Upd[*OF* ‹*Obj* f $T \to_\beta^*$ *Obj* f' T› ‹*finite* L› -
‹*lc* (*Obj* f T)› ‹*body* t›]
have *rtranclp*: *Upd* (*Obj* f T) l $t \to_\beta^*$ *Upd* (*Obj* f' T) l t'
unfolding *pred-cof-def*
by *simp*

from
Obj-pbeta[*OF* ‹*Obj* f $T \Rightarrow_\beta$ *Obj* f' T›] ‹$l \in$ *dom* f›
par-beta-lc[*OF* ‹*Obj* f $T \Rightarrow_\beta$ *Obj* f' T›]
have $l \in$ *dom* f' **and** *lc* (*Obj* f' T) **by** *auto*
from *beta-Upd*[*OF this* ‹*body* t'›] *rtranclp*
show *?case* **by** *simp*
next
case (*Obj* f f' T) **note** *body* = *this*(2) **and** *pred* = *this*(3)
define *pred-bnd*
where *pred-bnd* s p b b' $l \longleftrightarrow$ ($\exists\, t''$. *the* $b^{[Fvar\ s, Fvar\ p]} \to_\beta^*$ $t'' \wedge$ *the* $b' = \sigma[s,p]$ t'')
for s p b b' **and** l::*Label*
from
pred ‹*dom* f' = *dom* f› *fmap-ex-cof2*[*of* f' f *pred-bnd*]
obtain L **where**
finite L **and**
$\forall\, l \in$*dom* f. $\forall\, s\ p.\ s \notin L \wedge p \notin L \wedge s \neq p \longrightarrow$ *pred-bnd* s p (f l) (f' l) l
unfolding *pred-cof-def pred-bnd-def*
by *auto*
from
this(2)
rtrancl-beta-obj-n[*OF* ‹*finite* L› - *sym*[*OF* ‹*dom* f' = *dom* f›] *body*]
show *?case*
unfolding *pred-bnd-def*
by *simp*
next
case (*beta* f f' T l p p')
note
rtrancl-beta-Call[*OF* ‹*Obj* f $T \to_\beta^*$ *Obj* f' T› ‹*lc* (*Obj* f T)›
‹$p \to_\beta^*$ p'› ‹*lc* p›]
moreover
from
Obj-pbeta[*OF* ‹*Obj* f $T \Rightarrow_\beta$ *Obj* f' T›] ‹$l \in$ *dom* f›
par-beta-lc[*OF* ‹*Obj* f $T \Rightarrow_\beta$ *Obj* f' T›]
par-beta-lc[*OF* ‹$p \Rightarrow_\beta$ p'›]
have $l \in$ *dom* f' **and** *lc* (*Obj* f' T) **and** *lc* p' **by** *auto*
note *beta.beta*[*OF this*]
ultimately
show *?case*
by (*auto simp*: *rtranclp.rtrancl-into-rtrancl*[*of* - - *Call* (*Obj* f' T) l p'])

next
case (*Bnd L t t'*) **note** *pred* = *this*(*2*)
hence *pred-cof L t t'*
unfolding *pred-cof-def*
by *blast*
moreover
from *pred* ‹*finite L*› *par-beta-body*[*of L t t'*]
have *body t'* **by** *blast*
ultimately
show *?case*
using ‹*finite L*›
by *auto*
qed
qed

4.5 Confluence (directly)

lemma *diamond-binder*:
fixes *L1 L2 t ta tb*
assumes
finite L1 **and**
pred-L1: $\forall\, s\ p.\ s \notin L1 \wedge p \notin L1 \wedge s \neq p$
$\longrightarrow (\exists\, t'.\ (t^{[Fvar\ s,\ Fvar\ p]} \Rightarrow_\beta t'$
$\wedge\ (\forall\, z.\ (t^{[Fvar\ s,\ Fvar\ \ p]} \Rightarrow_\beta z) \longrightarrow (\exists\, u.\ t' \Rightarrow_\beta u \wedge z \Rightarrow_\beta u)))$
$\wedge\ ta = \sigma[s,p]t')$ **and**
finite L2 **and**
pred-L2: $\forall\, s\ p.\ s \notin L2 \wedge p \notin L2 \wedge s \neq p$
$\longrightarrow (\exists\, t'.\ t^{[Fvar\ s,\ Fvar\ p]} \Rightarrow_\beta t' \wedge tb = \sigma[s,p]t')$
shows
$\exists\, L'.$ *finite* L'
$\wedge\ (\exists\, t''.\ (\forall\, s\ p.\ s \notin L' \wedge p \notin L' \wedge s \neq p$
$\longrightarrow (\exists\, u.\ ta^{[Fvar\ s,\ Fvar\ p]} \Rightarrow_\beta u \wedge t'' = \sigma[s,p]u))$
$\wedge\ (\forall\, s\ p.\ s \notin L' \wedge p \notin L' \wedge s \neq p$
$\longrightarrow (\exists\, u.\ tb^{[Fvar\ s,\ Fvar\ p]} \Rightarrow_\beta u \wedge t'' = \sigma[s,p]u)))$
proof −
from ‹*finite L1*› ‹*finite L2*› **have** *finite* (*L1* ∪ *L2*) **by** *simp*
from *exFresh-s-p-cof*[*OF this*]
obtain *s p* **where** *sp*: $s \notin L1 \cup L2 \wedge p \notin L1 \cup L2 \wedge s \neq p$ **by** *auto*
with *pred-L1*
obtain *t'* **where**
$t^{[Fvar\ s,Fvar\ p]} \Rightarrow_\beta t'$ **and**
$\forall\, z.\ t^{[Fvar\ s,Fvar\ p]} \Rightarrow_\beta z \longrightarrow (\exists\, u.\ t' \Rightarrow_\beta u \wedge z \Rightarrow_\beta u)$ **and**
$ta = \sigma[s,p]\ t'$
by *blast*

from *sp pred-L2* **obtain** *t''* **where** $t^{[Fvar\ s,Fvar\ p]} \Rightarrow_\beta t''$ **and** $tb = \sigma[s,p]\ t''$
by *blast*
from ‹$\forall\, z.\ t^{[Fvar\ s,Fvar\ p]} \Rightarrow_\beta z \longrightarrow (\exists\, u.\ t' \Rightarrow_\beta u \wedge z \Rightarrow_\beta u)$› *this*(*1*)

obtain u **where** $t' \Rightarrow_\beta u$ **and** $t'' \Rightarrow_\beta u$ **by** *blast*

from ‹*finite L1*› ‹*finite L2*› **have** *finite* $(L1 \cup L2 \cup FV\ t \cup \{s\} \cup \{p\})$ **by** *simp*
moreover
{
fix x :: *sterm* **and** y :: *sterm*
assume $t^{[Fvar\ s,\ Fvar\ p]} \Rightarrow_\beta y$ **and** $x = \sigma[s,p]\ y$ **and** $y \Rightarrow_\beta u$
hence
$\forall\, sa\ pa.\ sa \notin L1 \cup L2 \cup FV\ t \cup \{s\} \cup \{p\}$
$\quad \wedge\ pa \notin L1 \cup L2 \cup FV\ t \cup \{s\} \cup \{p\} \wedge sa \neq pa$
$\longrightarrow (\exists\, t.\ x^{[Fvar\ sa,Fvar\ pa]} \Rightarrow_\beta t \wedge\ \sigma[s,p]\ u = \sigma[sa,pa]\ t)$
proof (*intro strip*)
fix *sa* :: *fVariable* **and** *pa* :: *fVariable*
assume
sapa: $sa \notin L1 \cup L2 \cup FV\ t \cup \{s\} \cup \{p\}$
$\quad \wedge\ pa \notin L1 \cup L2 \cup FV\ t \cup \{s\} \cup \{p\} \wedge sa \neq pa$
with *sp par-beta-lc*[*OF* ‹$y \Rightarrow_\beta u$›]
have $s \neq p$ **and** $s \notin FV\ (Fvar\ pa)$ **and** *lc y* **and** *lc u* **by** *auto*
from
sopen-sclose-eq-ssubst[*OF this*(*1−3*)]
sopen-sclose-eq-ssubst[*OF this*(*1−2*) *this*(*4*)]
renaming-par-beta ‹$x = \sigma[s,p]\ y$› ‹$y \Rightarrow_\beta u$›
have $x^{[Fvar\ sa,\ Fvar\ pa]} \Rightarrow_\beta (\sigma[s,p]\ u^{[Fvar\ sa,\ Fvar\ pa]})$
by (*auto simp*: *openz-def closez-def*)

moreover
from
sapa par-beta-preserves-FV[*OF* ‹$t\ ^{[Fvar\ s,Fvar\ p]} \Rightarrow_\beta y$›]
sopen-FV[*of 0 Fvar s Fvar p t*]
par-beta-preserves-FV[*OF* ‹$y \Rightarrow_\beta u$›]
sclose-subset-FV[*of 0 s p u*]
have $sa \notin FV\ (\sigma[s,p]\ u)$ **and** $pa \notin FV\ (\sigma[s,p]\ u)$ **and** $sa \neq pa$
by (*auto simp*: *openz-def closez-def*)
from *sym*[*OF sclose-sopen-eq-t*[*OF this*]]
have $\sigma[s,p]\ u = \sigma[sa,pa]\ (\sigma[s,p]\ u^{[Fvar\ sa,\ Fvar\ pa]})$
by (*simp add*: *openz-def closez-def*)

ultimately show $\exists\, t.\ x^{[Fvar\ sa,Fvar\ pa]} \Rightarrow_\beta t \wedge \sigma[s,p]\ u = \sigma[sa,pa]\ t$
by *blast*
qed
}note
this[*OF* ‹$t\ ^{[Fvar\ s,Fvar\ p]} \Rightarrow_\beta t'$› ‹$ta = \sigma[s,p]\ t'$› ‹$t' \Rightarrow_\beta u$›]
this[*OF* ‹$t\ ^{[Fvar\ s,Fvar\ p]} \Rightarrow_\beta t''$› ‹$tb = \sigma[s,p]\ t''$› ‹$t'' \Rightarrow_\beta u$›]
ultimately
show
$\exists\, L'.\ finite\ L'$
$\wedge\ (\exists\, t''.\ (\forall\, s\ p.\ s \notin L' \wedge p \notin L' \wedge s \neq p$
$\longrightarrow (\exists\, t'.\ ta^{[Fvar\ s,Fvar\ p]} \Rightarrow_\beta t' \wedge t'' = \sigma[p]\ t'))$

$\wedge$ ($\forall$ *s p. s* $\notin$ *L'* $\wedge$ *p* $\notin$ *L'* $\wedge$ *s* $\neq$ *p*
$\longrightarrow$ ($\exists$ *t'. tb*$^{[Fvar\ s, Fvar\ p]}$ $\Rightarrow_\beta$ *t'* $\wedge$ *t''* = σ[*s*,*p*] *t'*)))
by (*rule-tac x* = *L1* $\cup$ *L2* $\cup$ *FV t* $\cup$ {*s*} $\cup$ {*p*} **in** *exI*, *simp*, *blast*)
qed

lemma *exL-exMap-lem*:
fixes
f :: *Label* $-^\sim>$ *sterm* **and**
lz :: *Label* $-^\sim>$ *sterm* **and** *f'* :: *Label* $-^\sim>$ *sterm*
assumes *dom f* = *dom lz* **and** *dom f'* = *dom f*
shows
$\forall$ *L1 L2*. *finite L1*
$\longrightarrow$ ($\forall$ *l*$\in$*dom f*. $\forall$ *s p*. *s* $\notin$ *L1* $\wedge$ *p* $\notin$ *L1* $\wedge$ *s* $\neq$ *p*
$\longrightarrow$ ($\exists$ *t*. (*the*(*f l*)$^{[Fvar\ s,\ Fvar\ p]}$ $\Rightarrow_\beta$ *t*
$\wedge$ ($\forall$ *z*. (*the*(*f l*)$^{[Fvar\ s,\ Fvar\ \ p]}$ $\Rightarrow_\beta$ *z*)
$\longrightarrow$ ($\exists$ *u*. *t* $\Rightarrow_\beta$ *u* $\wedge$ *z* $\Rightarrow_\beta$ *u*)))
$\wedge$ *the*(*f' l*) = σ[*s*,*p*]*t*))
$\longrightarrow$ *finite L2*
$\longrightarrow$ ($\forall$ *l*$\in$*dom f*. $\forall$ *s p*. *s* $\notin$ *L2* $\wedge$ *p* $\notin$ *L2* $\wedge$ *s* $\neq$ *p*
$\longrightarrow$ ($\exists$ *t*. *the*(*f l*)$^{[Fvar\ s,\ Fvar\ p]}$ $\Rightarrow_\beta$ *t* $\wedge$ *the*(*lz l*) = σ[*s*,*p*]*t*))
$\longrightarrow$ ($\exists$ *L'*. *finite L'*
$\wedge$ ($\exists$ *lu*. *dom lu* = *dom f*
$\wedge$ ($\forall$ *l*$\in$*dom f*. $\forall$ *s p*. *s* $\notin$ *L'* $\wedge$ *p* $\notin$ *L'* $\wedge$ *s* $\neq$ *p*
$\longrightarrow$ ($\exists$ *t*. (*the*(*f' l*)$^{[Fvar\ s,\ Fvar\ p]}$) $\Rightarrow_\beta$ *t*
$\wedge$ *the*(*lu l*) = σ[*s*,*p*]*t*))
$\wedge$ ($\forall$ *l*$\in$*dom f*. *body* (*the* (*f' l*)))
$\wedge$ ($\forall$ *l*$\in$*dom f*. $\forall$ *s p*. *s* $\notin$ *L'* $\wedge$ *p* $\notin$ *L'* $\wedge$ *s* $\neq$ *p*
$\longrightarrow$ ($\exists$ *t*. (*the*(*lz l*)$^{[Fvar\ s,\ Fvar\ p]}$) $\Rightarrow_\beta$ *t*
$\wedge$ *the*(*lu l*) = σ[*s*,*p*]*t*))
$\wedge$ ($\forall$ *l*$\in$*dom f*. *body* (*the* (*lz l*)))))
using *assms*
proof (*induct rule*: *fmap-induct3*)
case *empty* **thus** *?case* **by** *force*
next
case (*insert x a b c F1 F2 F3*) **thus** *?case*
proof (*intro strip*)
fix *L1* :: *fVariable set* **and** *L2* :: *fVariable set*
{
fix
L :: *fVariable set* **and**
t :: *sterm* **and** *F* :: *Label* $-^\sim>$ *sterm* **and**
P :: *sterm* $\Rightarrow$ *sterm* $\Rightarrow$ *fVariable* $\Rightarrow$ *fVariable* $\Rightarrow$ *bool*
assume
dom F1 = *dom F* **and**
$*$: $\forall$ *l*$\in$*dom* (*F1*(*x* $\mapsto$ *a*)).
$\forall$ *s p*. *s* $\notin$ *L* $\wedge$ *p* $\notin$ *L* $\wedge$ *s* $\neq$ *p*
$\longrightarrow$ *P* (*the* ((*F1*(*x* $\mapsto$ *a*)) *l*)) (*the* ((*F*(*x* $\mapsto$ *t*)) *l*)) *s p*
hence

```
    F: ∀ l∈dom F1. ∀ s p. s ∉ L ∧ p ∉ L ∧ s ≠ p
        ⟶ P (the(F1 l)) (the(F l)) s p
  proof (intro strip)
    fix l :: Label and s :: fVariable and p :: fVariable
    assume l ∈ dom F1 hence l ∈ dom (F1(x ↦ a)) by simp
    moreover assume s ∉ L ∧ p ∉ L ∧ s ≠ p
    ultimately
    have P (the((F1(x ↦ a)) l)) (the((F(x ↦ t)) l)) s p
      using ∗ by blast
    moreover from ‹x ∉ dom F1› ‹l ∈ dom F1› have l ≠ x by auto
    ultimately show P (the(F1 l)) (the(F l)) s p by force
  qed

  from ∗ have ∀ s p. s ∉ L ∧ p ∉ L ∧ s ≠ p ⟶ P a t s p by auto
  note this F
}
note pred = this
note
  tmp =
  pred[of -  L1 (λt t' s p.
                  ∃ t''. (t^[Fvar s,Fvar p] ⇒β t''
                    ∧ (∀ z. t^[Fvar s,Fvar p] ⇒β z ⟶ (∃ u. t'' ⇒β u ∧ z ⇒β u)))
                     ∧ t' = σ[s,p] t'')]
note predc = tmp(1) note predF3 = tmp(2)
note tmp = pred[of - L2
                (λt t' s p. ∃ t''. t^[Fvar s,Fvar p] ⇒β t'' ∧ t' = σ[s,p] t'')]
note predb = tmp(1) note predF2 = tmp(2)

assume
  a: ∀ l∈dom (F1(x ↦ a)). ∀ s p. s ∉ L1 ∧ p ∉ L1 ∧ s ≠ p
    ⟶ (∃ t. (the ((F1(x ↦ a)) l)^[Fvar s,Fvar p] ⇒β t
          ∧ (∀ z. the ((F1(x ↦ a)) l)^[Fvar s,Fvar p] ⇒β z
              ⟶ (∃ u. t ⇒β u ∧ z ⇒β u)))
          ∧ the ((F3(x ↦ c)) l) = σ[s,p] t) and
  b: ∀ l∈dom (F1(x ↦ a)). ∀ s p. s ∉ L2 ∧ p ∉ L2 ∧ s ≠ p
    ⟶ (∃ t. the ((F1(x ↦ a)) l)^[Fvar s,Fvar p] ⇒β t
          ∧ the ((F2(x ↦ b)) l) = σ[s,p] t) and
  finite L1 and finite L2
from
  diamond-binder[OF this(3) predc[OF sym[OF ‹dom F3 = dom F1›] this(1)]
                    this(4) predb[OF ‹dom F1 = dom F2› this(2)]]
obtain La t where
  finite La and
  pred-c: ∀ s p. s ∉ La ∧ p ∉ La ∧ s ≠ p
            ⟶ (∃ u. c^[Fvar s,Fvar p] ⇒β u ∧ t = σ[s,p] u) and
  pred-b: ∀ s p. s ∉ La ∧ p ∉ La ∧ s ≠ p
            ⟶ (∃ u. b^[Fvar s,Fvar p] ⇒β u ∧ t = σ[s,p] u)
  by blast
```

```
{
  from this(1) have finite (La ∪ FV c ∪ FV b) by simp
  from exFresh-s-p-cof[OF this]
  obtain s p where
    sp: s ∉ La ∪ FV c ∪ FV b ∧ p ∉ La ∪ FV c ∪ FV b ∧ s ≠ p
    by auto
  with pred-c obtain u where c^[Fvar s,Fvar p] ⇒β u by blast
  from par-beta-lc[OF this] have lc (c^[Fvar s,Fvar p]) by simp
  with lc-body[of c^[Fvar s,Fvar p] s p] sp sclose-sopen-eq-t[of s c p 0]
  have c: body c by (auto simp: closez-def openz-def)

  from sp pred-b obtain u where b^[Fvar s,Fvar p] ⇒β u by blast
  from par-beta-lc[OF this] have lc (b^[Fvar s,Fvar p]) by simp
  with lc-body[of b^[Fvar s,Fvar p] s p] sp sclose-sopen-eq-t[of s b p 0]
  have body b by (auto simp: closez-def openz-def)
  note c this
}note bodycb = this
from
  predF3[OF sym[OF ‹dom F3 = dom F1›] a]
  predF2[OF ‹dom F1 = dom F2› b]
  ‹finite L1› ‹finite L2›
have
  ∃ L'. finite L'
      ∧ (∃ lu. dom lu = dom F1
      ∧ (∀ l∈dom F1. ∀ s p. s ∉ L' ∧ p ∉ L' ∧ s ≠ p
                    ⟶ (∃ t. the (F3 l)^[Fvar s,Fvar p] ⇒β t
                          ∧ the (lu l) = σ[s,p] t))
      ∧ (∀ l∈dom F1. body (the (F3 l)))
      ∧ (∀ l∈dom F1. ∀ s p. s ∉ L' ∧ p ∉ L' ∧ s ≠ p
                    ⟶ (∃ t. the (F2 l)^[Fvar s,Fvar p] ⇒β t
                          ∧ the (lu l) = σ[s,p] t))
      ∧ (∀ l∈dom F1. body (the (F2 l))))
  by (rule-tac x = L1 in allE[OF insert(1)], simp)
then obtain Lb f where
  finite Lb and dom f = dom F1 and
  pred-F3: ∀ l∈dom F1. ∀ s p. s ∉ La ∪ Lb ∧ p ∉ La ∪ Lb ∧ s ≠ p
                    ⟶ (∃ t. the (F3 l)^[Fvar s,Fvar p] ⇒β t
                          ∧ the (f l) = σ[s,p] t) and
  body-F3: ∀ l∈dom F1. body (the (F3 l)) and
  pred-F2: ∀ l∈dom F1. ∀ s p. s ∉ La ∪ Lb ∧ p ∉ La ∪ Lb ∧ s ≠ p
                    ⟶ (∃ t. the (F2 l)^[Fvar s,Fvar p] ⇒β t
                          ∧ the (f l) = σ[s,p] t) and
  body-F2: ∀ l∈dom F1. body (the (F2 l))
  by auto
from ‹finite La› ‹finite Lb› have finite (La ∪ Lb) by simp
moreover
from ‹dom f = dom F1› have dom (f(x ↦ t)) = dom (F1(x ↦ a)) by simp
moreover
```

from *pred-c pred-F3*
have
$\forall\, l{\in}dom\ (F1(x \mapsto a)).\ \forall\, s\ p.\ s \notin La \cup Lb \wedge p \notin La \cup Lb \wedge s \neq p$
$\longrightarrow (\exists\, t'.\ the\ ((F3(x \mapsto c))\ l)^{[Fvar\ s, Fvar\ p]} \Rightarrow_\beta t'$
$\wedge\ the\ ((f(x \mapsto t))\ l) = \sigma[s,p]\ t')$
by *auto*
moreover
from *bodycb(1) body-F3*
have $\forall\, l{\in}dom\ (F1(x \mapsto a)).\ body\ (the\ ((F3(x \mapsto c))\ l))$
by *simp*
moreover
from *pred-b pred-F2*
have
$\forall\, l{\in}dom\ (F1(x \mapsto a)).\ \forall\, s\ p.\ s \notin La \cup Lb \wedge p \notin La \cup Lb \wedge s \neq p$
$\longrightarrow (\exists\, t'.\ the\ ((F2(x \mapsto b))\ l)^{[Fvar\ s, Fvar\ p]} \Rightarrow_\beta t'$
$\wedge\ the\ ((f(x \mapsto t))\ l) = \sigma[s,p]\ t')$
by *auto*
moreover
from *bodycb(2) body-F2*
have $\forall\, l{\in}dom\ (F1(x \mapsto a)).\ body\ (the\ ((F2(x \mapsto b))\ l))$
by *simp*
ultimately
show
$\exists\, L'.\ finite\ L'$
$\wedge\ (\exists\, lu.\ dom\ lu = dom\ (F1(x \mapsto a))$
$\wedge\ (\forall\, l{\in}dom\ (F1(x \mapsto a)).$
$\forall\, s\ p.\ s \notin L' \wedge p \notin L' \wedge s \neq p$
$\longrightarrow (\exists\, t'.\ the\ ((F3(x \mapsto c))\ l)^{[Fvar\ s, Fvar\ p]} \Rightarrow_\beta t'$
$\wedge\ the\ (lu\ l) = \sigma[s,p]\ t'))$
$\wedge\ (\forall\, l{\in}dom\ (F1(x \mapsto a)).\ body\ (the\ ((F3(x \mapsto c))\ l)))$
$\wedge\ (\forall\, l{\in}dom\ (F1(x \mapsto a)).$
$\forall\, s\ p.\ s \notin L' \wedge p \notin L' \wedge s \neq p$
$\longrightarrow (\exists\, t'.\ the\ ((F2(x \mapsto b))\ l)^{[Fvar\ s, Fvar\ p]} \Rightarrow_\beta t'$
$\wedge\ the\ (lu\ l) = \sigma[s,p]\ t'))$
$\wedge\ (\forall\, l{\in}dom\ (F1(x \mapsto a)).\ body\ (the\ ((F2(x \mapsto b))\ l))))$
by (*rule-tac* $x = La \cup Lb$ **in** *exI*,
simp (*no-asm-simp*) *only*: *conjI simp-thms*(*22*),
rule-tac $x = (f(x \mapsto t))$ **in** *exI*, *simp*)
qed
qed

lemma *exL-exMap*:
$[\![\ dom\ (f{::}Label\ -^{\sim}{>}\ sterm) = dom\ (lz{::}Label\ -^{\sim}{>}\ sterm);$
$dom\ (f'{::}Label\ -^{\sim}{>}\ sterm) = dom\ f;$
$finite\ L1;$
$\forall\, l{\in}dom\ f.\ \forall\, s\ p.\ s \notin L1 \wedge p \notin L1 \wedge s \neq p$
$\longrightarrow (\exists\, t.\ (the(f\ l)^{[Fvar\ s,\ Fvar\ p]} \Rightarrow_\beta t$
$\wedge\ (\forall\, z.\ (the(f\ l)^{[Fvar\ s,\ Fvar\ \ p]} \Rightarrow_\beta z) \longrightarrow (\exists\, u.\ t \Rightarrow_\beta u \wedge z \Rightarrow_\beta u)))$

```
        ∧ the(f' l) = σ[s,p]t);
    finite L2;
    ∀ l∈dom lz. ∀ s p. s ∉ L2 ∧ p ∉ L2 ∧ s ≠ p
     ⟶ (∃ t. the(f l)[Fvar s, Fvar p] ⇒β t ∧ the(lz l) = σ[s,p]t) ⟧
⟹ ∃ L'. finite L'
      ∧ (∃ lu. dom lu = dom f
          ∧ (∀ l∈dom f. ∀ s p. s ∉ L' ∧ p ∉ L' ∧ s ≠ p
                      ⟶ (∃ t. (the(f' l)[Fvar s, Fvar p]) ⇒β t
                          ∧ the(lu l) = σ[s,p]t))
          ∧ (∀ l∈dom f. body (the (f' l)))
          ∧ (∀ l∈dom f. ∀ s p. s ∉ L' ∧ p ∉ L' ∧ s ≠ p
                      ⟶ (∃ t. (the(lz l)[Fvar s, Fvar p]) ⇒β t
                          ∧ the(lu l) = σ[s,p]t))
          ∧ (∀ l∈dom f. body (the (lz l))))
  using exL-exMap-lem[of f lz f'] by simp

lemma diamond-par-beta: diamond par-beta
  unfolding diamond-def commute-def square-def
proof (rule impI [THEN allI [THEN allI]])
  fix x y assume x ⇒β y
  thus ∀ z. x ⇒β z ⟶ (∃ u. y ⇒β u ∧ z ⇒β u)
  proof (induct rule: par-beta.induct)
    case pbeta-Fvar thus ?case by simp
  next
    case (pbeta-Upd t t' L u u' l)
    note pred-t = this(2) and pred-u = this(5)
    show ?case
    proof (intro strip)
      fix z assume Upd t l u ⇒β z
      thus ∃ u. Upd t' l u' ⇒β u ∧ z ⇒β u
      proof (induct rule: Upd-par-red)

        case (upd ta ua La)
        from
          diamond-binder[OF ‹finite L› pred-u this(2−3)]
          this(1) pred-t
          par-beta-lc[OF this(1)] par-beta-lc[OF ‹t ⇒β t'›]
        obtain L' ub tb where
          t' ⇒β tb and lc t' and ta ⇒β tb and
          lc ta and finite L' and
          ∀ s p. s ∉ L' ∧ p ∉ L' ∧ s ≠ p
            ⟶ (∃ u. u'[Fvar s,Fvar p] ⇒β u ∧ ub = σ[s,p] u) and
          ∀ s p. s ∉ L' ∧ p ∉ L' ∧ s ≠ p
            ⟶ (∃ u. ua[Fvar s,Fvar p] ⇒β u ∧ ub = σ[s,p] u)
          by auto
        from
          par-beta.pbeta-Upd[OF this(1−2) this(5−6)]
          par-beta.pbeta-Upd[OF this(3−5) this(7)]
          par-beta-body[OF this(5−6)]
```

```
      par-beta-body[OF this(5) this(7)] ‹z = Upd ta l ua›
    show ?case by (force simp: exI[of - Upd tb l ub])
  next
    case (obj f fa T ua La)

    from diamond-binder[OF ‹finite L› pred-u this(4−5)]
    obtain Lb ub where
      finite Lb and
      ub1: ∀ s p. s ∉ Lb ∧ p ∉ Lb ∧ s ≠ p
            ⟶ (∃ u. ua[Fvar s,Fvar p] ⇒β u ∧ ub = σ[s,p] u) and
      ub2: ∀ s p. s ∉ Lb ∧ p ∉ Lb ∧ s ≠ p
            ⟶ (∃ u. u'[Fvar s,Fvar p] ⇒β u ∧ ub = σ[s,p] u)
      by auto
    from ‹Obj f T = t› ‹Obj f T ⇒β Obj fa T›
    have t ⇒β Obj fa T by simp
    with pred-t obtain a where t' ⇒β a Obj fa T ⇒β a
      by auto
    with
      par-beta-lc[OF this(2)]
      par-beta-body[OF ‹finite Lb› ub1]
    obtain fb where
      t' ⇒β Obj fb T and Obj fa T ⇒β Obj fb T and
      lc (Obj fa T) and body ua
      by auto
    from Obj-pbeta-subst[OF ‹finite Lb› ub1 this(2−4)]
    have Obj (fa(l ↦ ua)) T ⇒β Obj (fb(l ↦ ub)) T by assumption
    moreover
    from
      ‹t ⇒β t'› ‹Obj f T = t›
      par-beta-lc[OF ‹t ⇒β t'›] ‹t' ⇒β Obj fb T›
      par-beta-body[OF ‹finite Lb› ub2]
    obtain f' where
      t' = Obj f' T and Obj f' T ⇒β Obj fb T and
      lc (Obj f' T) and body u'
      by auto
    note par-beta.pbeta-Upd'[OF this(2) ‹finite Lb› ub2 - this(3−4)]
    moreover
    from
      ‹t ⇒β t'› ‹Obj f T = t› ‹t' = Obj f' T›
      ‹l ∈ dom f› Obj-pbeta[of f T f']
    have l ∈ dom f' by simp
    ultimately
    show ?case
      using ‹z = Obj (fa(l ↦ ua)) T› ‹t' = Obj f' T›
      by (rule-tac x = Obj (fb(l ↦ ub)) T in exI, simp)
  qed
qed
next
  case (pbeta-Obj f' f L T) note pred = this(3)
```

```
  show ?case
  proof (clarify)

    fix fa La
    assume
      dom fa = dom f and finite La and
      ∀ l∈dom f. ∀ s p. s ∉ La ∧ p ∉ La ∧ s ≠ p
                  ⟶ (∃ t. the (f l)[Fvar s,Fvar p] ⇒β t
                       ∧ the (fa l) = σ[s,p] t)
    from
      exL-exMap[OF sym[OF this(1)] ‹dom f' = dom f›
                  ‹finite L› pred this(2)]
      this(1) this(3) ‹dom f' = dom f›
    obtain Lb fb where
      dom fb = dom f' and dom fb = dom fa and finite Lb and
      ∀ l∈dom f'. ∀ s p. s ∉ Lb ∧ p ∉ Lb ∧ s ≠ p
                  ⟶ (∃ t. the (f' l)[Fvar s,Fvar p] ⇒β t
                       ∧ the (fb l) = σ[s,p] t) and
      ∀ l∈dom f'. body (the (f' l)) and
      ∀ l∈dom fa. ∀ s p. s ∉ Lb ∧ p ∉ Lb ∧ s ≠ p
                  ⟶ (∃ t. the (fa l)[Fvar s,Fvar p] ⇒β t
                       ∧ the (fb l) = σ[s,p] t) and
      ∀ l∈dom fa. body (the (fa l))
      by auto
    from
      par-beta.pbeta-Obj[OF this(1) this(3−5)]
      par-beta.pbeta-Obj[OF this(2) this(3) this(6−7)]
    show ∃ u. Obj f' T ⇒β u ∧ Obj fa T ⇒β u
      by (rule-tac x = Obj fb T in exI, simp)
  qed
next
  case (pbeta-Upd' f T f' L t t' l)
  note pred-obj = this(2) and pred-bnd = this(4)
  show ?case
  proof (clarify)
    fix z assume Upd (Obj f T) l t ⇒β z
    thus ∃ u. Obj (f'(l ↦ t')) T ⇒β u ∧ z ⇒β u
    proof (induct rule: Upd-par-red)

      case (upd a ta La) note pred-ta = this(3)
      from ‹Obj f T ⇒β a› ‹z = Upd a l ta›
      obtain fa where
        Obj f T ⇒β Obj fa T and z = Upd (Obj fa T) l ta
        by auto
      from this(1) pred-obj
      obtain b where Obj f' T ⇒β b and Obj fa T ⇒β b
        by (elim allE impE exE conjE, simp)
      with
        par-beta-lc[OF this(1)] par-beta-lc[OF this(2)]
```

obtain *fb* **where**
Obj f′ T $\Rightarrow_\beta$ *Obj fb T* **and** *lc* (*Obj f′ T*) **and**
Obj fa T $\Rightarrow_\beta$ *Obj fb T* **and** *lc* (*Obj fa T*)
by *auto*
from *diamond-binder*[*OF* ‹*finite L*› *pbeta-Upd′*(*4*) ‹*finite La*› *pred-ta*]
obtain *Lb tb* **where**
finite Lb **and**
cb1: $\forall$ *s p*. *s* $\notin$ *Lb* $\wedge$ *p* $\notin$ *Lb* $\wedge$ *s* $\neq$ *p*
$\longrightarrow$ ($\exists$ *u*. *t′*[*Fvar s*,*Fvar p*] $\Rightarrow_\beta$ *u* $\wedge$ *tb* = σ[*s*,*p*] *u*) **and**
cb2: $\forall$ *s p*. *s* $\notin$ *Lb* $\wedge$ *p* $\notin$ *Lb* $\wedge$ *s* $\neq$ *p*
$\longrightarrow$ ($\exists$ *u*. *ta*[*Fvar s*,*Fvar p*] $\Rightarrow_\beta$ *u* $\wedge$ *tb* = σ[*s*,*p*] *u*)
by *auto*
from
par-beta-body[*OF this*(*1*−*2*)]
Obj-pbeta-subst[*OF* ‹*finite Lb*› *cb1* ‹*Obj f′ T* $\Rightarrow_\beta$ *Obj fb T*›
‹*lc* (*Obj f′ T*)›]
have *Obj* (*f′*(*l* $\mapsto$ *t′*)) *T* $\Rightarrow_\beta$ *Obj* (*fb*(*l* $\mapsto$ *tb*)) *T*
by *simp*
moreover
from *Obj-pbeta*[*OF* ‹*Obj f T* $\Rightarrow_\beta$ *Obj fa T*›] ‹*l* $\in$ *dom f*›
have *l* $\in$ *dom fa* **by** *simp*
from
par-beta-body[*OF* ‹*finite Lb*› *cb2*]
par-beta.pbeta-Upd′[*OF* ‹*Obj fa T* $\Rightarrow_\beta$ *Obj fb T*› ‹*finite Lb*›
cb2 this ‹*lc* (*Obj fa T*)›]
have *Upd* (*Obj fa T*) *l ta* $\Rightarrow_\beta$ *Obj* (*fb*(*l* $\mapsto$ *tb*)) *T* **by** *simp*
ultimately
show *?case*
using ‹*z* = *Upd* (*Obj fa T*) *l ta*›
by (*rule-tac x* = *Obj* (*fb*(*l* $\mapsto$ *tb*)) *T* **in** *exI*, *simp*)
next

case (*obj f″ fa T′ ta La*)
note *pred-ta* = *this*(*5*) **and** *this*
hence
l $\in$ *dom f* **and** *Obj f T* $\Rightarrow_\beta$ *Obj fa T* **and**
z = *Obj* (*fa*(*l* $\mapsto$ *ta*)) *T*
by *auto*
from *diamond-binder*[*OF* ‹*finite L*› *pred-bnd* ‹*finite La*› *pred-ta*]
obtain *Lb tb* **where**
finite Lb **and**
tb1: $\forall$ *s p*. *s* $\notin$ *Lb* $\wedge$ *p* $\notin$ *Lb* $\wedge$ *s* $\neq$ *p*
$\longrightarrow$ ($\exists$ *u*. *t′*[*Fvar s*,*Fvar p*] $\Rightarrow_\beta$ *u* $\wedge$ *tb* = σ[*s*,*p*] *u*) **and**
tb2: $\forall$ *s p*. *s* $\notin$ *Lb* $\wedge$ *p* $\notin$ *Lb* $\wedge$ *s* $\neq$ *p*
$\longrightarrow$ ($\exists$ *u*. *ta*[*Fvar s*,*Fvar p*] $\Rightarrow_\beta$ *u* $\wedge$ *tb* = σ[*s*,*p*] *u*)
by *auto*
from ‹*Obj f T* $\Rightarrow_\beta$ *Obj fa T*› *pred-obj*
obtain *b* **where** *Obj f′ T* $\Rightarrow_\beta$ *b* **and** *Obj fa T* $\Rightarrow_\beta$ *b*
by (*elim allE impE exE conjE*, *simp*)

```
    with par-beta-lc[OF this(1)] par-beta-lc[OF this(2)]
    obtain fb where
      Obj f' T ⇒β Obj fb T lc (Obj f' T) and
      Obj fa T ⇒β Obj fb T lc (Obj fa T)
      by auto
    from
      par-beta-body[OF ‹finite Lb› tb1]
      Obj-pbeta-subst[OF ‹finite Lb› tb1 this(1−2)]
      par-beta-body[OF ‹finite Lb› tb2]
      Obj-pbeta-subst[OF ‹finite Lb› tb2 this(3−4)]
    have
      Obj (f'(l ↦ t')) T ⇒β Obj (fb(l ↦ tb)) T and
      Obj (fa(l ↦ ta)) T ⇒β Obj (fb(l ↦ tb)) T
      by simp+
    with ‹z = Obj (fa(l ↦ ta)) T› show ?case
      by (rule-tac x = Obj (fb(l ↦ tb)) T in exI, simp)
  qed
 qed
next
 case (pbeta-Call t t' u u' l)
 note pred-t = this(2) and pred-u = this(4)
 show ?case
 proof (intro strip)
  fix z assume Call t l u ⇒β z
  thus ∃ u. Call t' l u' ⇒β u ∧ z ⇒β u
  proof (induct rule: Call-par-red)

   case (call ta ua)
   from
     this(1−2) pred-t pred-u
   obtain tb ub where t' ⇒β tb u' ⇒β ub ta ⇒β tb ua ⇒β ub
     by (elim allE impE exE conjE, simp)
   from
     par-beta-lc[OF this(1)] par-beta-lc[OF this(2)]
     par-beta.pbeta-Call[OF this(1−2)]
     par-beta-lc[OF this(3)] par-beta-lc[OF this(4)]
     par-beta.pbeta-Call[OF this(3−4)]
     ‹z = Call ta l ua›
   show ?case
     by (rule-tac x = Call tb l ub in exI, simp)
  next

   case (beta f fa T ua)
   from this(1−2) have t ⇒β Obj fa T by simp
   with ‹u ⇒β ua› pred-t pred-u
   obtain b ub where
     t' ⇒β b and Obj fa T ⇒β b and u' ⇒β ub and ua ⇒β ub
     by (elim allE impE exE conjE, simp)
   from this(1−2) par-beta-lc[OF this(2)]
```

```
      obtain fb where
        t' ⇒β Obj fb T and
        Obj fa T ⇒β Obj fb T and lc (Obj fa T)
        by auto
      from
        par-beta-beta[OF ‹l ∈ dom fa› this(2) ‹ua ⇒β ub› this(3)]
        par-beta-lc[OF ‹ua ⇒β ub›]
      have the (fa l)[Obj fa T,ua] ⇒β (the (fb l)[Obj fb T,ub]) by simp
      moreover
      from ‹l ∈ dom fa› Obj-pbeta[OF ‹Obj fa T ⇒β Obj fb T›]
      have l ∈ dom fb by simp
      from
        ‹t ⇒β t'› sym[OF ‹Obj f T = t›]
        par-beta-lc[OF ‹t ⇒β t'›] ‹t' ⇒β Obj fb T›
      obtain f' where
        t' = Obj f' T and Obj f' T ⇒β Obj fb T and
        lc (Obj f' T)
        by auto
      from
        Obj-pbeta[OF this(2)] ‹l ∈ dom fb›
        par-beta.pbeta-beta[OF this(2) - ‹u' ⇒β ub› this(3)]
        par-beta-lc[OF ‹u' ⇒β ub›]
      have Call (Obj f' T) l u' ⇒β (the (fb l)[Obj fb T,ub]) by auto
      ultimately
      show ?case
        using ‹t' = Obj f' T› ‹z = (the (fa l)[Obj fa T,ua])›
        by (rule-tac x = (the (fb l)[Obj fb T,ub]) in exI, simp)
    qed
  qed
next
  case (pbeta-beta f T f' l p p')
  note pred-obj = this(2) and pred-p = this(5)
  show ?case
  proof (intro strip)
    fix z assume Call (Obj f T) l p ⇒β z
    thus ∃ u. the (f' l)[Obj f' T,p'] ⇒β u ∧ z ⇒β u
    proof (induct rule: Call-par-red)

      case (call a pa)
      then obtain fa where
        Obj f T ⇒β Obj fa T and z = Call (Obj fa T) l pa
        by auto
      from
        this(1) ‹p ⇒β pa› pred-obj pred-p
      obtain b pb where
        Obj f' T ⇒β b and Obj fa T ⇒β b and
        p' ⇒β pb and pa ⇒β pb
        by (elim allE impE exE conjE, simp)
```

```
      with par-beta-lc[OF this(1)] par-beta-lc[OF this(2)]
      obtain fb where
        Obj f' T ⇒β Obj fb T and lc (Obj f' T) and
        Obj fa T ⇒β Obj fb T and lc (Obj fa T)
        by auto
      from this(1) ‹l ∈ dom f› ‹Obj f T ⇒β Obj f' T› ‹Obj f T ⇒β Obj fa T›
      have l ∈ dom f' and l ∈ dom fa by auto
      from ‹p' ⇒β pb› ‹pa ⇒β pb› par-beta-lc
      have p' ⇒β pb and lc p' and pa ⇒β pb and lc pa by auto
      from
        par-beta.pbeta-beta[OF ‹Obj fa T ⇒β Obj fb T› ‹l ∈ dom fa›
                               this(3) ‹lc (Obj fa T)› this(4)]
        par-beta-beta[OF ‹l ∈ dom f'› ‹Obj f' T ⇒β Obj fb T›
                         this(1) ‹lc (Obj f' T)› this(2)]
        ‹z = Call (Obj fa T) l pa›
      show ?case
        by (rule-tac x = (the (fb l)[Obj fb T,pb]) in exI, simp)
    next

      case (beta f'' fa Ta pa)
      hence Obj f T ⇒β Obj fa T and z = (the (fa l)[Obj fa T,pa])
        by auto
      with ‹p ⇒β pa› pred-obj pred-p
      obtain b pb where
        Obj f' T ⇒β b and Obj fa T ⇒β b and
        p' ⇒β pb and pa ⇒β pb
        by (elim allE impE exE conjE, simp)
      with par-beta-lc
      obtain fb where
        Obj f' T ⇒β Obj fb T and lc (Obj f' T) and lc p' and
        Obj fa T ⇒β Obj fb T and lc (Obj fa T) and lc pa
        by auto
      from ‹l ∈ dom f› ‹Obj f T ⇒β Obj f' T› ‹Obj f T ⇒β Obj fa T›
      have l ∈ dom f' and l ∈ dom fa by auto
      from
        par-beta-beta[OF ‹l ∈ dom f'› ‹Obj f' T ⇒β Obj fb T›
                         ‹p' ⇒β pb› ‹lc (Obj f' T)› ‹lc p'›]
        par-beta-beta[OF ‹l ∈ dom fa› ‹Obj fa T ⇒β Obj fb T›
                         ‹pa ⇒β pb› ‹lc (Obj fa T)› ‹lc pa›]
        ‹z = (the (fa l)[Obj fa T,pa])›
      show ?case
        by (rule-tac x = (the (fb l)[Obj fb T,pb]) in exI, simp)
    qed
  qed
 qed
qed
```

4.6 Confluence (classical not via complete developments)

theorem *beta-confluent*: *confluent beta*
by (*rule diamond-par-beta diamond-to-confluence*
par-beta-subset-beta beta-subset-par-beta)+

end

theory *Environments* **imports** *Main* **begin**

4.7 Type Environments

Some basic properties of our variable environments.

datatype *'a environment* =
Env (*string* ⇀ *'a*)
| *Malformed*

primrec
add :: (*'a environment*) ⇒ *string* ⇒ *'a* ⇒ *'a environment*
(‹-(⟦-:-⟧)› [*90*, *0*, *0*] *91*)
where
add-def: (*Env e*)(⟦*x*:*a*⟧) =
(*if* (*x* ∉ *dom e*) *then* (*Env* (*e*(*x* ↦ *a*))) *else Malformed*)
| *add-mal*: *Malformed*(⟦*x*:*a*⟧) = *Malformed*

primrec
env-dom :: (*'a environment*) ⇒ *string set*
where
env-dom-def: *env-dom* (*Env e*) = *dom e*
| *env-dom-mal*: *env-dom* (*Malformed*) = {}

primrec
env-get :: (*'a environment*) ⇒ *string* ⇒ *'a option* (‹-!-›)
where
env-get-def: *env-get* (*Env e*) *x* = *e x*
| *env-get-mal*: *env-get* (*Malformed*) *x* = *None*

primrec *ok*::(*'a environment*) ⇒ *bool*
where
OK-Env [*intro*]: *ok* (*Env e*) = (*finite* (*dom e*))
| *OK-Mal* [*intro*]: *ok Malformed* = *False*

lemma *subst-add*:

fixes x y
assumes $x \neq y$
shows $e(\!|x{:}a|\!)(\!|y{:}b|\!) = e(\!|y{:}b|\!)(\!|x{:}a|\!)$
proof (*cases e*)
case *Malformed* **thus** *?thesis* **by** *simp*
next
case (*Env f*) **with** *assms* **show** *?thesis*
proof (*cases* $x \in$ *dom f*, *simp*)
case *False* **with** *assms Env* **show** *?thesis*
proof (*cases* $y \in$ *dom f*, *simp-all*, *intro ext*)
fix *xa* :: *string*
case *False* **with** *assms* **show** $(f(x \mapsto a, y \mapsto b))\ xa = (f(y \mapsto b, x \mapsto a))\ xa$
proof (*cases* $xa = x$, *simp*)
case *False* **with** *assms* **show** *?thesis*
by (*cases* $xa = y$, *simp-all*)
qed
qed
qed
qed

lemma *ok-finite*[*simp*]: *ok e* $\Longrightarrow$ *finite* (*env-dom e*)
by (*cases e*, *simp+*)

lemma *ok-ok*[*simp*]: *ok e* $\Longrightarrow \exists x.\ e = (Env\ x)$
by (*cases e*, *simp+*)

lemma *env-defined*:
fixes *x* :: *string* **and** *e* :: *'a environment*
assumes $x \in$ *env-dom e*
shows $\exists\, T\ .\ e!x = Some\ T$
proof (*cases e*)
case *Malformed* **with** *assms* **show** *?thesis* **by** *simp*
next
case *Env* **with** *assms* **show** *?thesis* **by** (*simp*, *force*)
qed

lemma *env-bigger*: $[\![\ a \notin$ *env-dom e*; $x \in$ (*env-dom e*) $]\!] \Longrightarrow x \in$ *env-dom* $(e(\!|a{:}X|\!))$
by (*cases e*, *simp-all*)

lemma *env-bigger2*:
$[\![\ a \notin$ *env-dom e*; $b \notin$ (*env-dom e*); $x \in$ (*env-dom e*); $a \neq b\ ]\!]$
$\Longrightarrow x \in$ *env-dom* $(e(\!|a{:}X|\!)(\!|b{:}Y|\!))$
by (*cases e*, *simp-all*)

lemma *not-malformed*: $x \in (\textit{env-dom}\ e) \Longrightarrow \exists \textit{fun}.\ e = \textit{Env}\ \textit{fun}$
by (*cases e*, *simp-all*)

lemma *not-malformed-smaller*:
fixes $e :: 'a\ \textit{environment}$ **and** $a :: \textit{string}$ **and** $X :: 'a$
assumes *ok* $(e(\!|a{:}X|\!))$
shows *ok e*
proof (*cases e*)
case *Malformed* **with** *assms* **show** *?thesis* **by** *simp*
next
case (*Env f*) **with** *ok-finite*[*OF assms*] *assms* **show** *?thesis*
by (*cases* $a \notin \textit{dom}\ f$, *simp-all*)
qed

lemma *not-in-smaller*:
fixes $e :: 'a\ \textit{environment}$ **and** $a :: \textit{string}$ **and** $X :: 'a$
assumes *ok* $(e(\!|a{:}X|\!))$
shows $a \notin \textit{env-dom}\ e$
proof (*cases e*)
case *Malformed* **thus** *?thesis* **by** *simp*
next
case (*Env f*) **with** *assms* **show** *?thesis*
by (*cases* $a \notin \textit{dom}\ f$, *simp-all*)
qed

lemma *in-add*:
fixes $e :: 'a\ \textit{environment}$ **and** $a :: \textit{string}$ **and** $X :: 'a$
assumes *ok* $(e(\!|a{:}X|\!))$
shows $a \in \textit{env-dom}\ (e(\!|a{:}X|\!))$
proof (*cases e*)
case *Malformed* **with** *assms* **show** *?thesis* **by** *simp*
next
case (*Env f*) **with** *assms* **show** *?thesis*
by (*cases* $a \notin \textit{dom}\ f$, *simp-all*)
qed

lemma *ok-add-reverse*:
fixes
$e :: 'a\ \textit{environment}$ **and** $a :: \textit{string}$ **and** $X :: 'a$ **and**
$b :: \textit{string}$ **and** $Y :: 'a$
assumes *ok* $(e(\!|a{:}X|\!)(\!|b{:}Y|\!))$
shows $(e(\!|b{:}Y|\!)(\!|a{:}X|\!)) = (e(\!|a{:}X|\!)(\!|b{:}Y|\!))$
proof (*cases e*)
case *Malformed* **with** *assms* **show** *?thesis* **by** *simp*

next
case (*Env f*)
with
not-in-smaller[*OF* ‹*ok* (*e*(|*a*:*X*|)(|*b*:*Y*|))›] *in-add*[*OF assms*]
not-in-smaller[*OF not-malformed-smaller*[*OF assms*]]
in-add[*OF not-malformed-smaller*[*OF assms*]]
show *?thesis*
by (*simp*, *intro conjI impI*, *elim conjE*, *auto simp*: *fun-upd-twist*)
qed

lemma *not-in-env-bigger*:
fixes *e* :: *'a environment* **and** *a* :: *string* **and** *X* :: *'a* **and** *x* :: *string*
assumes *x* ∉ (*env-dom e*) **and** *x* ≠ *a*
shows *x* ∉ *env-dom* (*e*(|*a*:*X*|))
proof (*cases e*)
case *Malformed* **thus** *?thesis* **by** *simp*
next
case (*Env f*) **with** *assms* **show** *?thesis*
by (*cases a* ∉ *dom f*, *simp-all*)
qed

lemma *not-in-env-bigger-2*:
fixes
e :: *'a environment* **and** *a* :: *string* **and** *X* :: *'a* **and**
b :: *string* **and** *Y* :: *'a* **and** *x* :: *string*
assumes *x* ∉ (*env-dom e*) **and** *x* ≠ *a* **and** *x* ≠ *b*
shows *x* ∉ *env-dom* (*e*(|*a*:*X*|)(|*b*:*Y*|))
proof (*cases e*)
case *Malformed* **thus** *?thesis* **by** *simp*
next
case (*Env f*) **with** *assms* **show** *?thesis*
by (*cases a* ∉ *dom f*, *simp-all*)
qed

lemma *not-in-env-smaller*:
fixes *e* :: *'a environment* **and** *a* :: *string* **and** *X* :: *'a* **and** *x* :: *string*
assumes *x* ∉ (*env-dom* (*e*(|*a*:*X*|))) **and** *x* ≠ *a* **and** *ok* (*e*(|*a*:*X*|))
shows *x* ∉ *env-dom e*
proof (*cases e*)
case *Malformed* **with** *assms*(*3*) **show** *?thesis* **by** *simp*
next
case (*Env f*) **with** *assms* **show** *?thesis*
by (*cases a* ∉ *dom f*, *simp-all*)
qed

lemma *ok-add-2*:
fixes
e :: *'a environment* **and** *a* :: *string* **and** *X* :: *'a* **and**

```
  b :: string and Y :: 'a
  assumes ok (e(|a:X|)(|b:Y|))
  shows ok e ∧ a ∉ env-dom e ∧ b ∉ env-dom e ∧ a ≠ b
proof −
  {
    assume ok (e(|b:X|)(|b:Y|))
    from not-in-smaller[OF this] in-add[OF not-malformed-smaller[OF this]]
    have False by simp
  } with assms have a ≠ b by auto
  moreover
  from assms ok-add-reverse[OF assms] have ok (e(|b:Y|)(|a:X|)) by simp
  note not-in-smaller[OF not-malformed-smaller[OF this]]
  ultimately
  show ?thesis
    using
      not-malformed-smaller[OF not-malformed-smaller[OF assms]]
      not-in-smaller[OF not-malformed-smaller[OF assms]]
  by simp
qed


lemma in-add-2:
  fixes
  e :: 'a environment and a :: string and X :: 'a and
  b :: string and Y :: 'a
  assumes ok (e(|a:X|)(|b:Y|))
  shows a ∈ env-dom (e(|a:X|)(|b:Y|)) ∧ b ∈ env-dom (e(|a:X|)(|b:Y|))
proof −
  from ok-add-2[OF assms] show ?thesis
    by (elim conjE, intro conjI, (cases e, simp-all)+)
qed


lemma ok-add-3:
  fixes
  e :: 'a environment and a :: string and X :: 'a and
  b :: string and Y :: 'a and c :: string and Z :: 'a
  assumes ok (e(|a:X|)(|b:Y|)(|c:Z|))
  shows
  a ∉ env-dom e ∧ b ∉ env-dom e ∧ c ∉ env-dom e ∧ a ≠ b ∧ b ≠ c ∧ a ≠ c
proof −
  {
    assume ok (e(|a:X|)(|c:Y|)(|c:Z|))
    from not-in-smaller[OF this] in-add[OF not-malformed-smaller[OF this]]
    have False by simp
  } with assms have b ≠ c by auto
  moreover
  from assms ok-add-reverse[OF assms] have ok (e(|a:X|)(|c:Z|)(|b:Y|)) by simp
  note ok-add-2[OF not-malformed-smaller[OF this]]
```

```
  ultimately
  show ?thesis using ok-add-2[OF not-malformed-smaller[OF assms]]
    by simp
qed

lemma in-env-smaller:
  fixes e :: 'a environment and a :: string and X :: 'a and x :: string
  assumes x ∈ (env-dom (e(⦇a:X⦈))) and x ≠ a
  shows x ∈ env-dom e
proof −
  from not-malformed[OF assms(1)] obtain f where f: e(⦇a:X⦈) = Env f by auto
  with assms show ?thesis
  proof (cases e)
    case Malformed with ‹e(⦇a:X⦈) = Env f›
    have False by simp
    then show ?thesis ..
  next
    case (Env f') with assms f show ?thesis
      by (simp, cases a ∈ dom f', simp-all, force)
  qed
qed

lemma in-env-smaller2:
  fixes
  e :: 'a environment and a :: string and X :: 'a and
  b :: string and Y :: 'a and x :: string
  assumes x ∈ (env-dom (e(⦇a:X⦈)(⦇b:Y⦈))) and x ≠ a and x ≠ b
  shows x ∈ env-dom e
 by (simp add: in-env-smaller[OF in-env-smaller[OF assms(1) assms(3)] assms(2)])

lemma get-env-bigger:
  fixes e :: 'a environment and a :: string and X :: 'a and x :: string
  assumes x ∈ (env-dom (e(⦇a:X⦈))) and x ≠ a
  shows e!x = e(⦇a:X⦈)!x
proof −
  from not-malformed[OF assms(1)] obtain f where f: e(⦇a:X⦈) = Env f by auto
  thus ?thesis proof (cases e)
    case Malformed with ‹e(⦇a:X⦈) = Env f›
    show ?thesis by simp
  next
    case (Env f') with assms f show ?thesis
      by (cases a ∉ dom f', auto)
  qed
qed

lemma get-env-bigger2:
  fixes
  e :: 'a environment and a :: string and X :: 'a and
  b :: string and Y :: 'a and x :: string
```

assumes $x \in (\mathit{env\text{-}dom}\ (e(\!|a{:}X|\!)(\!|b{:}Y|\!)))$ **and** $x \neq a$ **and** $x \neq b$
shows $e!x = e(\!|a{:}X|\!)(\!|b{:}Y|\!)!x$
by (*simp add*: *get-env-bigger*[*OF assms*(*1*) *assms*(*3*)]
get-env-bigger[*OF in-env-smaller*[*OF assms*(*1*) *assms*(*3*)] *assms*(*2*)])

lemma *get-env-smaller*: $[\![\ x \in \mathit{env\text{-}dom}\ e;\ a \notin \mathit{env\text{-}dom}\ e\ ]\!] \Longrightarrow e(\!|a{:}X|\!)!x = e!x$
by (*cases e*, *auto*)

lemma *get-env-smaller2*:
$[\![\ x \in \mathit{env\text{-}dom}\ e;\ a \notin \mathit{env\text{-}dom}\ e;\ b \notin \mathit{env\text{-}dom}\ e;\ a \neq b\ ]\!]$
$\Longrightarrow e(\!|a{:}X|\!)(\!|b{:}Y|\!)!x = e!x$
by (*cases e*, *auto*)

lemma *add-get-eq*: $[\![\ xa \notin \mathit{env\text{-}dom}\ e;\ \mathit{ok}\ e;\ \mathit{the}\ e(\!|xa{:}U|\!)!xa = T\ ]\!] \Longrightarrow U = T$
by (*cases e*, *auto*)

lemma *add-get*: $[\![\ xa \notin \mathit{env\text{-}dom}\ e;\ \mathit{ok}\ e\ ]\!] \Longrightarrow \mathit{the}\ e(\!|xa{:}U|\!)!xa = U$
by (*cases e*, *auto*)

lemma *add-get2-1*:
fixes $e :: 'a\ \mathit{environment}$ **and** $x :: \mathit{string}$ **and** $A :: 'a$ **and** $y :: \mathit{string}$ **and** $B :: 'a$
assumes $\mathit{ok}\ (e(\!|x{:}A|\!)(\!|y{:}B|\!))$
shows $\mathit{the}\ e(\!|x{:}A|\!)(\!|y{:}B|\!)!x = A$
proof −
from *ok-add-2*[*OF assms*] **show** *?thesis*
by (*cases e*, *elim conjE*, *simp-all*)
qed

lemma *add-get2-2*:
fixes $e :: 'a\ \mathit{environment}$ **and** $x :: \mathit{string}$ **and** $A :: 'a$ **and** $y :: \mathit{string}$ **and** $B :: 'a$
assumes $\mathit{ok}\ (e(\!|x{:}A|\!)(\!|y{:}B|\!))$
shows $\mathit{the}\ e(\!|x{:}A|\!)(\!|y{:}B|\!)!y = B$
proof −
from *ok-add-2*[*OF assms*] **show** *?thesis*
by (*cases e*, *elim conjE*, *simp-all*)
qed

lemma *ok-add-ok*: $[\![\ \mathit{ok}\ e;\ x \notin \mathit{env\text{-}dom}\ e\ ]\!] \Longrightarrow \mathit{ok}\ (e(\!|x{:}X|\!))$
by (*cases e*, *auto*)

lemma *env-add-dom*:
fixes $e :: 'a\ \mathit{environment}$ **and** $x :: \mathit{string}$
assumes $\mathit{ok}\ e$ **and** $x \notin \mathit{env\text{-}dom}\ e$
shows $\mathit{env\text{-}dom}\ (e(\!|x{:}X|\!)) = \mathit{env\text{-}dom}\ e \cup \{x\}$
proof (*auto simp*: *in-add*[*OF ok-add-ok*[*OF assms*]], *rule ccontr*)
fix y **assume** $y \in \mathit{env\text{-}dom}\ (e(\!|x{:}X|\!))$ **and** $y \notin \mathit{env\text{-}dom}\ e$ **and** $y \neq x$
from *in-env-smaller*[*OF this*(*1*) *this*(*3*)] *this*(*2*) **show** *False* **by** *simp*
next
fix y **assume** $y \in \mathit{env\text{-}dom}\ e$

```
  from env-bigger[OF not-in-smaller[OF ok-add-ok[OF assms]] this]
  show y ∈ env-dom (e(|x:X|)) by assumption
qed

lemma env-add-dom-2:
  fixes e :: 'a environment and x :: string and y :: string
  assumes ok e and x ∉ env-dom e and y ∉ env-dom e and x ≠ y
  shows env-dom (e(|x:X|)(|y:Y|)) = env-dom e ∪ {x,y}
proof −
  from env-add-dom[OF assms(1−2)] assms(3−4)
  have y ∉ env-dom (e(|x:X|)) by simp
  from
    env-add-dom[OF assms(1−2)]
    env-add-dom[OF ok-add-ok[OF assms(1−2)] this]
  show ?thesis by auto
qed

fun
   env-app :: ('a environment) ⇒  ('a environment) ⇒ ('a environment) (‹-+-›)
where
  env-app (Env a) (Env b) =
  (if (ok (Env a) ∧ ok (Env b) ∧ env-dom (Env b) ∩ env-dom (Env a) = {})
   then  Env (a ++ b) else Malformed )

lemma env-app-dom:
  fixes e1 :: 'a environment and e2 :: 'a environment
  assumes ok e1 and env-dom e1 ∩ env-dom e2 = {} and ok e2
  shows env-dom (e1+e2) = env-dom e1 ∪ env-dom e2
proof −
  from ok-ok[OF ‹ok e1›] ok-ok[OF ‹ok e2›]
  obtain f1 f2 where e1 = Env f1 and e2 = Env f2 by auto
  with assms(2) ok-finite[OF ‹ok e1›] ok-finite[OF ‹ok e2›]
  show ?thesis by auto
qed

lemma env-app-same[simp]:
  fixes e1 :: 'a environment and e2 :: 'a environment and x :: string
  assumes
  ok e1 and x ∈ env-dom e1 and
  env-dom e1 ∩ env-dom e2 = {} and ok e2
  shows the (e1+e2!x) = the e1!x
proof −
  from ok-ok[OF ‹ok e1›] ok-ok[OF ‹ok e2›]
  obtain f1 f2 where e1 = Env f1 and e2 = Env f2 by auto
  with assms(2−3) ok-finite[OF ‹ok e1›] ok-finite[OF ‹ok e2›]
  show ?thesis proof (auto)
    fix y :: 'a assume dom f1 ∩ dom f2 = {} and f1 x = Some y
    from map-add-comm[OF this(1)] this(2) have (f1 ++ f2) x = Some y
      by (simp add: map-add-Some-iff)
```

```
    thus the ((f1 ++ f2) x) = y by auto
  qed
qed

lemma env-app-ok[simp]:
  fixes e1 :: 'a environment and e2 :: 'a environment
  assumes ok e1 and env-dom e1 ∩ env-dom e2 = {} and ok e2
  shows ok (e1+e2)
proof −
  from ok-ok[OF ‹ok e1›] ok-ok[OF ‹ok e2›]
  obtain f1 f2 where e1 = Env f1 and e2 = Env f2 by auto
  with assms show ?thesis by (simp,force)
qed

lemma env-app-add[simp]:
  fixes e1 :: 'a environment and e2 :: 'a environment and x :: string
  assumes
  ok e1 and env-dom e1 ∩ env-dom e2 = {} and ok e2 and
  x ∉ env-dom e1 and x ∉ env-dom e2
  shows (e1+e2)(|x:X|) = e1(|x:X|)+e2
proof −
  from ok-ok[OF ‹ok e1›] ok-ok[OF ‹ok e2›]
  obtain f1 f2 where e1 = Env f1 and e2 = Env f2 by auto
  with assms show ?thesis proof (clarify, simp, intro impI ext)
    fix xa :: string
    assume x ∉ dom f1 and x ∉ dom f2
    thus ((f1 ++ f2)(x ↦ X)) xa = (f1(x ↦ X) ++ f2) xa
    proof (cases x = xa, simp-all)
      case False thus (f1 ++ f2) xa = (f1(x ↦ X) ++ f2) xa
        by (simp add: map-add-def split: option.split)
    next
      case True with ‹x ∉ dom f1› ‹x ∉ dom f2›
      have (f1(xa ↦ X) ++ f2) xa = Some X
        by (auto simp: map-add-Some-iff)
      thus Some X = (f1(xa ↦ X) ++ f2) xa by simp
    qed
  qed
qed

lemma env-app-add2[simp]:
  fixes
  e1 :: 'a environment and e2 :: 'a environment and
  x :: string and y :: string
  assumes
  ok e1 and env-dom e1 ∩ env-dom e2 = {} and ok e2 and
  x ∉ env-dom e1 and x ∉ env-dom e2 and y ∉ env-dom e1 and
  y ∉ env-dom e2 and x ≠ y
  shows (e1+e2)(|x:X|)(|y:Y|) = e1(|x:X|)(|y:Y|)+e2
proof −
```

```
  from ok-ok[OF ‹ok e1›] ok-ok[OF ‹ok e2›]
  obtain f1 f2 where e1 = Env f1 and e2 = Env f2 by auto
  with assms show ?thesis proof (clarify, simp, intro impI ext)
    fix xa :: string
    assume x ∉ dom f1 and x ∉ dom f2 and y ∉ dom f1 and y ∉ dom f2
    with ‹x ≠ y›
    show ((f1 ++ f2)(x ↦ X, y ↦ Y)) xa = (f1(x ↦ X, y ↦ Y) ++ f2) xa
    proof (cases x = xa, simp)
      case True
      with ‹x ≠ y› ‹x ∉ dom f1› ‹x ∉ dom f2› ‹y ∉ dom f1› ‹y ∉ dom f2›
      have (f1(xa ↦ X,y ↦ Y) ++ f2) xa = Some X
        by (auto simp: map-add-Some-iff)
      thus Some X = (f1(xa ↦ X,y ↦ Y) ++ f2) xa by simp
    next
      case False thus ?thesis
      proof (cases y = xa, simp-all)
        case False with ‹x ≠ xa›
        show (f1 ++ f2) xa = (f1(x ↦ X,y ↦ Y) ++ f2) xa
          by (simp add: map-add-def split: option.split)
      next
        case True
        with ‹x ≠ y› ‹x ∉ dom f1› ‹x ∉ dom f2› ‹y ∉ dom f1› ‹y ∉ dom f2›
        have (f1(x ↦ X,xa ↦ Y) ++ f2) xa = Some Y
          by (auto simp: map-add-Some-iff)
        thus Some Y = (f1(x ↦ X, xa ↦ Y) ++ f2) xa by simp
      qed
    qed
  qed
qed

end
```

5 First Order Types for Sigma terms

```
theory TypedSigma imports ../preliminary/Environments Sigma begin
```

5.0.1 Types and typing rules

The inductive definition of the typing relation.

```
definition
  return :: (type × type) ⇒ type where
  return a = fst a

definition
  param :: (type × type) ⇒ type where
  param a = snd a

primrec
```

$do :: type \Rightarrow (Label\ set)$
where
$do\ (Object\ l) = (dom\ l)$

primrec
$type\text{-}get :: type \Rightarrow Label \Rightarrow (type \times type)\ option \quad (\langle\text{-}\widehat{}\text{-}\rangle\ 1000)$
where
$(Object\ l)\widehat{}n = (l\ n)$

inductive
$typing :: (type\ environment) \Rightarrow sterm \Rightarrow type \Rightarrow bool$
$(\langle\text{-} \vdash \text{-} : \text{-}\rangle\ [80,\ 0,\ 80]\ 230)$
where
$T\text{-}Var[intro!]:$
$[\![\ ok\ env;\ x \in env\text{-}dom\ env;\ (the\ (env!x)) = T\ ]\!]$
$\Longrightarrow env \vdash (Fvar\ x) : T$
$|\ T\text{-}Obj[intro!]:$
$[\![\ ok\ env;\ dom\ b = do\ A;\ finite\ F;$
$\forall l{\in}do\ A.\ \forall s\ p.\ s \notin F \wedge p \notin F \wedge s \neq p$
$\longrightarrow env(\!|s{:}A|\!)(\!|p{:}param(the\ (A\widehat{}l))|\!)$
$\vdash (the\ (b\ l)^{[Fvar\ s, Fvar\ p]}) : return(the\ (A\widehat{}l))\ ]\!]$
$\Longrightarrow env \vdash (Obj\ b\ A) : A$
$|\ T\text{-}Upd[intro!]:$
$[\![\ finite\ F;$
$\forall s\ p.\ s \notin F \wedge p \notin F \wedge s \neq p$
$\longrightarrow env(\!|s{:}A|\!)(\!|p{:}param(the\ (A\widehat{}l))|\!)$
$\vdash (n^{[Fvar\ s, Fvar\ p]}) : return(the\ (A\widehat{}l));$
$env \vdash a : A;\ l \in do\ A\ ]\!] \Longrightarrow env \vdash Upd\ a\ l\ n : A$
$|\ T\text{-}Call[intro!]:$
$[\![\ env \vdash a : A;\ env \vdash b : param(the\ (A\widehat{}l));\ l \in do\ A\ ]\!]$
$\Longrightarrow env \vdash (Call\ a\ l\ b) : return(the\ (A\widehat{}l))$

inductive-cases $typing\text{-}elims\ [elim!]:$
$e \vdash Obj\ b\ T : T$
$e \vdash Fvar\ x : T$
$e \vdash Call\ a\ l\ b : T$
$e \vdash Upd\ a\ l\ n : T$

5.0.2 Basic lemmas

Basic treats of the type system.

lemma $not\text{-}bvar$: $e \vdash t : T \Longrightarrow \forall i.\ t \neq Bvar\ i$
by $(erule\ typing.cases,\ simp\text{-}all)$

lemma $typing\text{-}regular'$: $e \vdash t : T \Longrightarrow ok\ e$
by $(induct\ rule{:}typing.induct,\ auto)$

lemma *typing-regular''*: $e \vdash t : T \Longrightarrow lc\ t$
by (*induct rule*:*typing.induct*, *auto*)

theorem *typing-regular*: $e \vdash t : T \Longrightarrow ok\ e \wedge lc\ t$
by (*simp add*: *typing-regular' typing-regular''*)

lemma *obj-inv*: $e \vdash Obj\ f\ U : A \Longrightarrow A = U$
by (*erule typing.cases*, *auto*)

lemma *obj-inv-elim*:
$e \vdash Obj\ f\ U : U$
$\Longrightarrow (dom\ f = do\ U)$
$\wedge\ (\exists F.\ finite\ F \wedge (\forall l{\in}do\ U.\ \forall s\ p.\ s \notin F \wedge p \notin F \wedge s \neq p$
$\longrightarrow e(\!|s{:}U|\!)(\!|p{:}param(the\ U\hat{}l)|\!)$
$\vdash (the\ (f\ l)^{[Fvar\ s, Fvar\ p]}) : return(the\ (U\hat{}l))))$
by (*erule typing.cases*, *simp-all*, *blast*)

lemma *typing-induct*[*consumes 1*, *case-names Fvar Call Upd Obj Bnd*]:
fixes
env :: *type environment* **and** t :: *sterm* **and** T :: *type* **and**
P1 :: *type environment* $\Rightarrow$ *sterm* $\Rightarrow$ *type* $\Rightarrow$ *bool* **and**
P2 :: *type environment* $\Rightarrow$ *sterm* $\Rightarrow$ *type* $\Rightarrow$ *Label* $\Rightarrow$ *bool*
assumes
$env \vdash t : T$ **and**
$\bigwedge env\ T\ x.\ [\![\ ok\ env;\ x \in env\text{-}dom\ env;\ the\ env!x = T\ ]\!]$
$\Longrightarrow P1\ env\ (Fvar\ x)\ T$ **and**
$\bigwedge env\ T\ t\ l\ p.\ [\![\ env \vdash t : T;\ P1\ env\ t\ T;\ env \vdash p : param\ (the(T\hat{}l));$
$P1\ env\ p\ (param\ (the(T\hat{}l)));\ l \in do\ T\ ]\!]$
$\Longrightarrow P1\ env\ (Call\ t\ l\ p)\ (return\ (the(T\hat{}l)))$ **and**
$\bigwedge env\ T\ t\ l\ u.\ [\![\ env \vdash t : T;\ P1\ env\ t\ T;\ l \in do\ T;\ P2\ env\ u\ T\ l\ ]\!]$
$\Longrightarrow P1\ env\ (Upd\ t\ l\ u)\ T$ **and**
$\bigwedge env\ T\ f.\ [\![\ ok\ env;\ dom\ f = do\ T;\ \forall l{\in}dom\ f.\ P2\ env\ (the(f\ l))\ T\ l\ ]\!]$
$\Longrightarrow P1\ env\ (Obj\ f\ T)\ T$ **and**
$\bigwedge env\ T\ l\ t\ L.\ [\![\ ok\ env;\ finite\ L;$
$\forall s\ p.\ s \notin L \wedge p \notin L \wedge s \neq p$
$\longrightarrow env(\!|s{:}T|\!)(\!|p{:}param\ (the(T\hat{}l))|\!)$
$\vdash (t^{[Fvar\ s,\ Fvar\ p]}) : return\ (the(T\hat{}l))$
$\wedge\ P1\ (env(\!|s{:}T|\!)(\!|p{:}param\ (the(T\hat{}l))|\!))\ (t^{[Fvar\ s,\ Fvar\ p]})$
$(return\ (the(T\hat{}l)))\ ]\!]$
$\Longrightarrow P2\ env\ t\ T\ l$
shows
P1 env t T
using *assms* **by** (*induct rule*: *typing.induct*, *auto simp*: *typing-regular'*)

lemma *ball-Tltsp*:
fixes
P1 :: *type* $\Rightarrow$ *Label* $\Rightarrow$ *sterm* $\Rightarrow$ *string* $\Rightarrow$ *string* $\Rightarrow$ *bool* **and**
P2 :: *type* $\Rightarrow$ *Label* $\Rightarrow$ *sterm* $\Rightarrow$ *string* $\Rightarrow$ *string* $\Rightarrow$ *bool*

assumes
$\bigwedge l\ t\ t'.\ [\![\ \forall s\ p.\ s \notin F \wedge p \notin F \wedge s \neq p \longrightarrow P1\ T\ l\ t\ s\ p\ ]\!]$
$\Longrightarrow \forall s\ p.\ s \notin F' \wedge p \notin F' \wedge s \neq p \longrightarrow P2\ T\ l\ t\ s\ p$ **and**
$\forall l{\in}do\ T.\ \forall s\ p.\ s \notin F \wedge p \notin F \wedge s \neq p \longrightarrow P1\ T\ l\ (the(f\ l))\ s\ p$
shows $\forall l{\in}do\ T.\ \forall s\ p.\ s \notin F' \wedge p \notin F' \wedge s \neq p \longrightarrow P2\ T\ l\ (the(f\ l))\ s\ p$
proof
fix l **assume** $l \in do\ T$
with *assms(2)*
have $\forall s\ p.\ s \notin F \wedge p \notin F \wedge s \neq p \longrightarrow P1\ T\ l\ (the(f\ l))\ s\ p$
by *simp*
with *assms(1)*
show $\forall s\ p.\ s \notin F' \wedge p \notin F' \wedge s \neq p \longrightarrow P2\ T\ l\ (the(f\ l))\ s\ p$
by *simp*
qed

lemma *ball-ex-finite*:
fixes
$S :: 'a\ set$ **and** $F :: 'b\ set$ **and** $x :: 'a$ **and**
$P :: 'a \Rightarrow 'b \Rightarrow 'b \Rightarrow bool$
assumes
finite S **and** *finite F* **and**
$\forall x{\in}S.\ (\exists F'.\ finite\ F'$
$\wedge\ (\forall s\ p.\ s \notin F' \cup F \wedge p \notin F' \cup F \wedge s \neq p$
$\longrightarrow P\ x\ s\ p))$
shows
$\exists F'.\ finite\ F'$
$\wedge\ (\forall x{\in}S.\ \forall s\ p.\ s \notin F' \cup F \wedge p \notin F' \cup F \wedge s \neq p$
$\longrightarrow P\ x\ s\ p)$
proof −
from *assms* **show** *?thesis*
proof (*induct S*)
case *empty* **thus** *?case* **by** *force*
next
case (*insert x S*)
from *insert(5)*
have
$\forall y{\in}S.\ (\exists F'.\ finite\ F'$
$\wedge\ (\forall s\ p.\ s \notin F' \cup F \wedge p \notin F' \cup F \wedge s \neq p$
$\longrightarrow P\ y\ s\ p))$
by *simp*
from *insert(3)*[*OF* ‹*finite F*› *this*]
obtain *F1* **where**
finite F1 **and**
pred-S: $\forall y{\in}S.\ \forall s\ p.\ s \notin F1 \cup F \wedge p \notin F1 \cup F \wedge s \neq p$
$\longrightarrow P\ y\ s\ p$
by *auto*
from *insert(5)*
obtain *F2* **where**

finite F2 **and**
$\forall\, s\ p.\ s \notin F2 \cup F \wedge p \notin F2 \cup F \wedge s \neq p \longrightarrow P\ x\ s\ p$
by *auto*
with *pred-S* **have**
$\forall\, y{\in}insert\ x\ S.\ \forall\, s\ p.\ s \notin F1 \cup F2 \cup F \wedge p \notin F1 \cup F2 \cup F \wedge s \neq p$
$\longrightarrow P\ y\ s\ p$
by *auto*
moreover
from ‹*finite F1*› ‹*finite F2*› **have** *finite* $(F1 \cup F2)$ **by** *simp*
ultimately
show *?case* **by** *blast*
qed
qed

lemma *bnd-renaming-lem*:
assumes
$s \notin FV\ t'$ **and** $p \notin FV\ t'$ **and** $x \notin FV\ t'$ **and** $y \notin FV\ t'$ **and**
$x \notin$ *env-dom env'* **and** $y \notin$ *env-dom env'* **and** $s \neq p$ **and** $x \neq y$ **and**
$t = \{Suc\ n \to [Fvar\ s,\ Fvar\ p]\}\ t'$ **and** $env = env'(\!|s{:}A|\!)(\!|p{:}B|\!)$ **and**
pred-bnd:
$\forall\, sa\ pa.\ sa \notin F \wedge pa \notin F \wedge sa \neq pa$
$\longrightarrow env(\!|sa{:}T|\!)(\!|pa{:}param(the(T\hat{}l))|\!) \vdash (t^{[Fvar\ sa,Fvar\ pa]}) : return(the(T\hat{}l))$
$\wedge\ (\forall\, env''\ t''\ s'\ p'\ x'\ y'\ A'\ B'\ n'.$
$s' \notin FV\ t'' \longrightarrow p' \notin FV\ t'' \longrightarrow x' \notin FV\ t'' \longrightarrow y' \notin FV\ t'' \longrightarrow$
$x' \notin$ *env-dom env''* $\longrightarrow y' \notin$ *env-dom env''* $\longrightarrow x' \neq y' \longrightarrow s' \neq p'$
$\longrightarrow (t^{[Fvar\ sa,Fvar\ pa]}) = \{n' \to [Fvar\ s',Fvar\ p']\}\ t''$
$\longrightarrow env(\!|sa{:}T|\!)(\!|pa{:}param(the(T\hat{}l))|\!) = env''(\!|s'{:}A'|\!)(\!|p'{:}B'|\!)$
$\longrightarrow env''(\!|x'{:}A'|\!)(\!|y'{:}B'|\!)$
$\vdash \{n' \to [Fvar\ x',Fvar\ y']\}\ t'' : return(the(T\hat{}l)))$ **and**
$FV\ t' \subseteq F'$
shows
$\forall\, sa\ pa.\ sa \notin F \cup \{s,p,x,y\} \cup F' \cup$ *env-dom env'*
$\wedge\ pa \notin F \cup \{s,p,x,y\} \cup F' \cup$ *env-dom env'*
$\wedge\ sa \neq pa$
$\longrightarrow env'(\!|x{:}A|\!)(\!|y{:}B|\!)(\!|sa{:}T|\!)(\!|pa{:}param(the(T\hat{}l))|\!)$
$\vdash (\{Suc\ n \to [Fvar\ x,\ Fvar\ y]\}\ t'^{[Fvar\ sa,Fvar\ pa]}) : return\ (the(T\hat{}l))$
proof (*intro strip, elim conjE*)
fix *sa pa*
assume
nin-sa: $sa \notin F \cup \{s,p,x,y\} \cup F' \cup$ *env-dom env'* **and**
nin-pa: $pa \notin F \cup \{s,p,x,y\} \cup F' \cup$ *env-dom env'* **and** $sa \neq pa$
hence $sa \notin F \wedge pa \notin F \wedge sa \neq pa$ **by** *auto*
moreover
{
fix *a* **assume** $a \notin FV\ t'$ **and** $a \in \{s,p,x,y\}$
with
‹$FV\ t' \subseteq F'$› *nin-sa nin-pa* ‹$sa \neq pa$›
sopen-FV[*of 0 Fvar sa Fvar pa t'*]

have $a \notin FV\ (t'^{[Fvar\ sa, Fvar\ pa]})$ **by** (*auto simp*: *openz-def*)
} note
this[*OF* ‹$s \notin FV\ t'$›] *this*[*OF* ‹$p \notin FV\ t'$›]
this[*OF* ‹$x \notin FV\ t'$›] *this*[*OF* ‹$y \notin FV\ t'$›]
moreover
from
not-in-env-bigger-2[*OF* ‹$x \notin env\text{-}dom\ env'$›]
not-in-env-bigger-2[*OF* ‹$y \notin env\text{-}dom\ env'$›]
nin-sa nin-pa
have
$x \notin env\text{-}dom\ (env'(\!|sa:T|\!)(\!|pa:param(the(T\hat{}l))|\!))$
$\wedge\ y \notin env\text{-}dom\ (env'(\!|sa:T|\!)(\!|pa:param(the(T\hat{}l))|\!))$ **by** *auto*
moreover
from ‹$t = \{Suc\ n \to [Fvar\ s,\ Fvar\ p]\}\ t'$› *sopen-commute*[*OF Suc-not-Zero*]
have $(t^{[Fvar\ sa, Fvar\ pa]}) = \{Suc\ n \to [Fvar\ s, Fvar\ p]\}\ (t'^{[Fvar\ sa, Fvar\ pa]})$
by (*auto simp*: *openz-def*)
moreover
from
subst-add[*of s sa env' A T*] *subst-add*[*of sa p env'*(|*s*:*A*|) *T B*]
subst-add[*of s pa env'*(|*sa*:*T*|) *A param*(*the*(*T*^*l*))]
subst-add[*of p pa env'*(|*sa*:*T*|)(|*s*:*A*|) *B param*(*the*(*T*^*l*))]
‹*env* = *env'*(|*s*:*A*|)(|*p*:*B*|)› *nin-sa nin-pa*
have $env(\!|sa:T|\!)(\!|pa:param(the(T\hat{}l))|\!) = env'(\!|sa:T|\!)(\!|pa:param(the(T\hat{}l))|\!)(\!|s:A|\!)(\!|p:B|\!)$
by *auto*
ultimately
have
$env'(\!|sa:T|\!)(\!|pa:param(the\ T\hat{}l)|\!)(\!|x:A|\!)(\!|y:B|\!)$
$\vdash \{Suc\ n \to [Fvar\ x,\ Fvar\ y]\}\ (t'^{[Fvar\ sa, Fvar\ pa]}) : return(the(T\hat{}l))$
using ‹$s \neq p$› ‹$x \neq y$› *pred-bnd* **by** *auto*
moreover
from
subst-add[*of y sa env'*(|*x*:*A*|) *B T*] *subst-add*[*of x sa env' A T*]
subst-add[*of y pa env'*(|*sa*:*T*|)(|*x*:*A*|) *B param*(*the*(*T*^*l*))]
subst-add[*of x pa env'*(|*sa*:*T*|) *A param*(*the*(*T*^*l*))]
nin-sa nin-pa
have
$env'(\!|x:A|\!)(\!|y:B|\!)(\!|sa:T|\!)(\!|pa:param(the(T\hat{}l))|\!)$
$= env'(\!|sa:T|\!)(\!|pa:param(the(T\hat{}l))|\!)(\!|x:A|\!)(\!|y:B|\!)$
by *auto*
ultimately
show
$env'(\!|x:A|\!)(\!|y:B|\!)(\!|sa:T|\!)(\!|pa:param(the(T\hat{}l))|\!)$
$\vdash (\{Suc\ n \to [Fvar\ x,\ Fvar\ y]\}\ t'^{[Fvar\ sa, Fvar\ pa]}) : return\ (the(T\hat{}l))$
using *sopen-commute*[*OF not-sym*[*OF Suc-not-Zero*]]
by (*simp add*: *openz-def*)
qed

lemma *type-renaming'*[*rule-format*]:

```
  e ⊢ t : C ⟹
  (⋀env t' s p x y A B n. ⟦ s ∉ FV t'; p ∉ FV t'; x ∉ FV t'; y ∉ FV t';
                            x ∉ env-dom env; y ∉ env-dom env; s ≠ p;  x ≠ y;
                            t = {n → [Fvar s,Fvar p]} t'; e = env(|s:A|)(|p:B|) ⟧
    ⟹ env(|x:A|)(|y:B|) ⊢ {n → [Fvar x,Fvar y]} t' : C)
proof (induct set:typing)
  case (T-Call env t1 T t2 l env' t' s p x y A B n)
  with sopen-eq-Call[OF sym[OF ‹Call t1 l t2 = {n → [Fvar s,Fvar p]} t'›]]
  show ?case by auto
next
  case (T-Var env a T env' t' s p x y A B n)
  from ‹ok env› ‹env = env'(|s:A|)(|p:B|)› ok-add-2[of env' s A p B]
  have ok env' by simp
  from
    ok-add-ok[OF ok-add-ok[OF this ‹x ∉ env-dom env'›]
                 not-in-env-bigger[OF ‹y ∉ env-dom env'› not-sym[OF ‹x ≠ y›]]]
  have ok: ok (env'(|x:A|)(|y:B|)) by assumption

  from sopen-eq-Fvar[OF sym[OF ‹Fvar a = {n → [Fvar s,Fvar p]} t'›]]
  show ?case
  proof (elim disjE conjE)
    assume t' = Fvar a with T-Var(4-7)
    obtain a ≠ s and a ≠ p and a ≠ x and a ≠ y by auto
    note in-env-smaller2[OF - this(1-2)]
    from ‹a ∈ env-dom env› ‹env = env'(|s:A|)(|p:B|)› this[of env' A B]
    have a ∈ env-dom env' by simp
    from env-bigger2[OF ‹x ∉ env-dom env'› ‹y ∉ env-dom env'› this ‹x ≠ y›]
    have inenv: a ∈ env-dom (env'(|x:A|)(|y:B|)) by assumption
    note get-env-bigger2[OF - ‹a ≠ s› ‹a ≠ p›]
    from
      this[of env' A B] ‹a ∈ env-dom env› ‹the env!a = T›
      ‹env = env'(|s:A|)(|p:B|)› get-env-bigger2[OF inenv ‹a ≠ x› ‹a ≠ y›]
    have the (env'(|x:A|)(|y:B|)!a) = T by simp
    from typing.T-Var[OF ok inenv this] ‹t' = Fvar a› show ?case by simp
  next
    assume a = s and t' = Bvar (Self n)
    from
      this(1) ‹ok env› ‹env = env'(|s:A|)(|p:B|)› ‹the env!a = T›
      add-get2-1[of env' s A p B]
    have T = A by simp
    moreover
    from ‹t' = Bvar (Self n)› have {n → [Fvar x,Fvar y]} t' = Fvar x by simp
    ultimately
    show ?case using in-add-2[OF ok] typing.T-Var[OF ok - add-get2-1[OF ok]]
      by simp
  next
    note subst = subst-add[OF ‹x ≠ y›]
    from subst[of env' A B] ok have ok': ok (env'(|y:B|)(|x:A|)) by simp
    assume a = p and t' = Bvar (Param n)
```

```
    from
      this(1) ‹ok env› ‹env = env'(|s:A|)(|p:B|)› ‹the env!a = T›
      add-get2-2[of env' s A p B]
    have T = B by simp
    moreover
    from ‹t' = Bvar (Param n)› have {n → [Fvar x,Fvar y]} t' = Fvar y by simp
    ultimately
    show ?case
      using
        subst[of env' A B] in-add-2[OF ok']
        typing.T-Var[OF ok' - add-get2-1[OF ok']]
      by simp
  qed
next
  case (T-Upd F env T l t2 t1 env' t' s p x y A B n)
  from sopen-eq-Upd[OF sym[OF ‹Upd t1 l t2 = {n → [Fvar s,Fvar p]} t'›]]
  obtain t1' t2' where
    t1: t1 = {n → [Fvar s,Fvar p]} t1' and
    t2: t2 = {Suc n → [Fvar s,Fvar p]} t2' and
    t': t' = Upd t1' l t2'
    by auto
  { fix a assume a ∉ FV t' with t' have a ∉ FV t1' by simp }
  note
    t1' = T-Upd(4)[OF this[OF ‹s ∉ FV t'›] this[OF ‹p ∉ FV t'›]
                      this[OF ‹x ∉ FV t'›] this[OF ‹y ∉ FV t'›]
                      ‹x ∉ env-dom env'› ‹y ∉ env-dom env'›
                      ‹s ≠ p› ‹x ≠ y› t1 ‹env = env'(|s:A|)(|p:B|)›]
  from ok-finite[of env'] ok-add-2[OF typing-regular'[OF this]]
  have findom: finite (env-dom env') by simp

  { fix a assume a ∉ FV t' with t' have a ∉ FV t2' by simp }
  note
    bnd-renaming-lem[OF this[OF ‹s ∉ FV t'›] this[OF ‹p ∉ FV t'›]
                        this[OF ‹x ∉ FV t'›] this[OF ‹y ∉ FV t'›]
                        ‹x ∉ env-dom env'› ‹y ∉ env-dom env'›
                        ‹s ≠ p› ‹x ≠ y› t2 ‹env = env'(|s:A|)(|p:B|)›]
  from this[of F T l FV t2'] T-Upd(2)
  have
    ∀ sa pa. sa ∉ F ∪ {s, p, x, y} ∪ FV t2' ∪ env-dom env'
          ∧ pa ∉ F ∪ {s, p, x, y} ∪ FV t2' ∪ env-dom env'
          ∧ sa ≠ pa
      ⟶ env'(|x:A|)(|y:B|)(|sa:T|)(|pa:param(the(T^l))|)
          ⊢ ({Suc n → [Fvar x,Fvar y]} t2'[Fvar sa,Fvar pa]) : return(the(T^l))
    by simp
  from
    typing.T-Upd[OF - this t1' ‹l ∈ do T›]
    ‹finite F› findom t'
  show ?case by simp
next
```

case (*T-Obj env f T F env′ t′ s p x y A B n*)
from ‹*ok env*› ‹*env* = *env′*⦇*s*:*A*⦈⦇*p*:*B*⦈› *ok-add-2*[*of env′ s A p B*]
have *ok env′* **by** *simp*
from
ok-add-ok[*OF ok-add-ok*[*OF this* ‹*x* ∉ *env-dom env′*›]
not-in-env-bigger[*OF* ‹*y* ∉ *env-dom env′*› *not-sym*[*OF* ‹*x* ≠ *y*›]]]
have *ok*: *ok* (*env′*⦇*x*:*A*⦈⦇*y*:*B*⦈) **by** *assumption*
from *sopen-eq-Obj*[*OF sym*[*OF* ‹*Obj f T* = {*n* → [*Fvar s*,*Fvar p*]} *t′*›]]
obtain *f′* **where**
obj: {*n* → [*Fvar s*,*Fvar p*]} *Obj f′ T* = *Obj f T* **and**
t′: *t′* = *Obj f′ T* **by** *auto*
from
this(*1*) ‹*dom f* = *do T*›
sym[*OF dom-sopenoption-lem*[*of Suc n Fvar s Fvar p f′*]]
dom-sopenoption-lem[*of Suc n Fvar x Fvar y f′*]
have *dom*: *dom* (λ*l*. *sopen-option* (*Suc n*) (*Fvar x*) (*Fvar y*) (*f′ l*)) = *do T*
by *simp*

from
‹*finite F*› *finite-FV*[*of Obj f′ T*]
ok-finite[*of env′*] *ok-add-2*[*OF ok*]
have *finF*: *finite* (*F* ∪ {*s*,*p*,*x*,*y*} ∪ *FV* (*Obj f′ T*) ∪ *env-dom env′*)
by *simp*

have
∀ *l*∈*do T*. ∀ *sa pa*. *sa* ∉ *F* ∪ {*s*, *p*, *x*, *y*} ∪ *FV* (*Obj f′ T*) ∪ *env-dom env′*
∧ *pa* ∉ *F* ∪ {*s*, *p*, *x*, *y*} ∪ *FV* (*Obj f′ T*) ∪ *env-dom env′*
∧ *sa* ≠ *pa*
⟶ *env′*⦇*x*:*A*⦈⦇*y*:*B*⦈⦇*sa*:*T*⦈⦇*pa*:*param*(*the*(*T*^*l*))⦈
⊢ (*the*(*sopen-option* (*Suc n*) (*Fvar x*) (*Fvar y*) (*f′ l*))[*Fvar sa*,*Fvar pa*]) :
return(*the*(*T*^*l*))
proof
fix *l* **assume** *l* ∈ *do T* **with** *T-Obj*(*4*)
have *cof*:
∀ *sa pa*. *sa* ∉ *F* ∧ *pa* ∉ *F* ∧ *sa* ≠ *pa*
⟶ *env*⦇*sa*:*T*⦈⦇*pa*:*param*(*the*(*T*^*l*))⦈
⊢ (*the*(*f l*)[*Fvar sa*,*Fvar pa*]) : *return*(*the*(*T*^*l*))
∧ (∀ *env″ t″ s′ p′ x′ y′ A′ B′ n′*.
s′ ∉ *FV t″* ⟶ *p′* ∉ *FV t″* ⟶ *x′* ∉ *FV t″* ⟶ *y′* ∉ *FV t″*
⟶ *x′* ∉ *env-dom env″* ⟶ *y′* ∉ *env-dom env″* ⟶ *x′* ≠ *y′*
⟶ *s′* ≠ *p′*
⟶ (*the*(*f l*)[*Fvar sa*,*Fvar pa*]) = {*n′* → [*Fvar s′*,*Fvar p′*]} *t″*
⟶ *env*⦇*sa*:*T*⦈⦇*pa*:*param*(*the*(*T*^*l*))⦈ = *env″*⦇*s′*:*A′*⦈⦇*p′*:*B′*⦈
⟶ *env″*⦇*x′*:*A′*⦈⦇*y′*:*B′*⦈
⊢ {*n′* → [*Fvar x′*,*Fvar y′*]} *t″* : *return*(*the*(*T*^*l*)))
by *simp*
from
‹*l* ∈ *do T*› ‹*dom f* = *do T*› ‹*Obj f T* = {*n* → [*Fvar s*,*Fvar p*]} *t′*› *obj t′*
dom-sopenoption-lem[*of Suc n Fvar s Fvar p f′*]

```
    have indomf': l ∈ dom f' by auto
    hence
      opened: the (sopen-option (Suc n) (Fvar x) (Fvar y) (f' l))
            = {Suc n → [Fvar x,Fvar y]} the(f' l)
      by force
    from indomf' have FVsubset: FV (the(f' l)) ⊆ FV (Obj f' T) by force
    with
      ‹s ∉ FV t'› ‹p ∉ FV t'› ‹x ∉ FV t'› ‹y ∉ FV t'› obj t'
      indomf' FV-option-lem[of f']
    obtain
      s ∉ FV (the(f' l)) and p ∉ FV (the(f' l)) and
      x ∉ FV (the(f' l)) and y ∉ FV (the(f' l)) and
      the(f l) = {Suc n → [Fvar s,Fvar p]} the(f' l) by auto
    from
      bnd-renaming-lem[OF this(1−4) ‹x ∉ env-dom env'› ‹y ∉ env-dom env'›
                          ‹s ≠ p› ‹x ≠ y› this(5) ‹env = env'⟪s:A⟫⟪p:B⟫›
                          cof FVsubset]
    show
      ∀ sa pa. sa ∉ F ∪ {s, p, x, y} ∪ FV (Obj f' T) ∪ env-dom env'
            ∧ pa ∉ F ∪ {s, p, x, y} ∪ FV (Obj f' T) ∪ env-dom env'
            ∧ sa ≠ pa
        ⟶ env'⟪x:A⟫⟪y:B⟫⟪sa:T⟫⟪pa:param(the(T^l))⟫
            ⊢ (the(sopen-option (Suc n) (Fvar x) (Fvar y) (f' l))[Fvar sa,Fvar pa]) :
              return(the(T^l))
      by (subst opened, assumption)
  qed
  from typing.T-Obj[OF ok dom finF this] t' show ?case by simp
qed

lemma type-renaming:
  ⟦ e⟪s:A⟫⟪p:B⟫ ⊢ {n → [Fvar s,Fvar p]} t : T;
    s ∉ FV t; p ∉ FV t; x ∉ FV t; y ∉ FV t;
    x ∉ env-dom e; y ∉ env-dom e; x ≠ y; s ≠ p⟧
  ⟹ e⟪x:A⟫⟪y:B⟫ ⊢ {n → [Fvar x,Fvar y]} t : T
  by (auto simp: type-renaming')


lemma obj-inv-elim':
  assumes
  e ⊢ Obj f U : U and
  nin-s: s ∉ FV (Obj f U) ∪ env-dom e and
  nin-p: p ∉ FV (Obj f U) ∪ env-dom e and s ≠ p
  shows
  (dom f = do U) ∧ (∀ l∈do U. e⟪s:U⟫⟪p:param(the(U^l))⟫
                          ⊢ (the(f l)[Fvar s,Fvar p]) : return(the(U^l)))
  using assms
proof (cases rule: typing.cases)
  case (T-Obj F)
```

```
  thus ?thesis
  proof (simp, intro strip)
    fix l assume l ∈ do U
    from ‹finite F› finite-FV[of Obj f U] have finite (F ∪ FV (Obj f U) ∪ {s,p})
      by simp
    from exFresh-s-p-cof[OF this]
    obtain sa pa where
      sa ≠ pa and
      nin-sa: sa ∉ F ∪ FV (Obj f U) and
      nin-pa: pa ∉ F ∪ FV (Obj f U) by auto
    with ‹l ∈ do U› T-Obj(4)
    have
      e(|sa:U|)(|pa:param(the(U`l))|)
      ⊢ (the(f l)[Fvar sa,Fvar pa]) : return(the(U`l))
      by simp
    moreover
    from ‹l ∈ do U› ‹dom f = do U›
    have l ∈ dom f by simp
    with nin-s nin-p nin-sa nin-pa FV-option-lem[of f]
    have
      sa ∉ FV (the(f l)) ∧ pa ∉ FV (the(f l))
       ∧ s ∉ FV (the(f l)) ∧ p ∉ FV (the(f l))
       ∧ s ∉ env-dom e ∧ p ∉ env-dom e by auto
    ultimately
    show
      e(|s:U|)(|p:param(the(U`l))|)
      ⊢ (the(f l)[Fvar s,Fvar p]) : return(the(U`l))
      using type-renaming[OF - - - - - - - ‹s ≠ p› ‹sa ≠ pa›]
      by (simp add: openz-def)
  qed
qed

lemma dom-lem: e ⊢ Obj f (Object fun) : Object fun ⟹ dom f = dom fun
  by (erule typing.cases, auto)

lemma abs-typeE:
  assumes e ⊢ Call (Obj f U) l b : T
  shows
  (∃F. finite F
     ∧ (∀s p. s ∉ F ∧ p ∉ F ∧ s ≠ p
         ⟶ e(|s:U|)(|p: param(the(U`l))|) ⊢ (the(f l)[Fvar s,Fvar p]) : T) ⟹ P)
  ⟹ P
  using assms
proof (cases rule: typing.cases)
  case (T-Call A )
  assume
    cof: ∃F. finite F
           ∧ (∀s p. s ∉ F ∧ p ∉ F ∧ s ≠ p
               ⟶ e(|s:U|)(|p: param(the(U`l))|) ⊢ (the(f l)[Fvar s,Fvar p]) : T)
```

$\Longrightarrow P$
from
‹$T = return(the(A\hat{}l))$›
‹$e \vdash Obj\ f\ U : A$› ‹$l \in do\ A$› *obj-inv*[*of e f U A*]
obtain $e \vdash (Obj\ f\ U) : U$ **and** $T = return(the(U\hat{}l))$ **and** $l \in do\ U$
by *simp*
from *obj-inv-elim*[*OF this*(*1*)] *this*(*2*−*3*) *cof* **show** *?thesis* **by** *blast*
qed

5.0.3 Substitution preserves Well-Typedness

lemma *bigger-env-lemma*[*rule-format*]:
assumes $e \vdash t : T$
shows $\forall x\ X.\ x \notin env\text{-}dom\ e \longrightarrow e(\!|x{:}X|\!) \vdash t{:}\ T$
proof −
define *pred-cof*
where *pred-cof L env t T l* $\longleftrightarrow$
$(\forall s\ p.\ s \notin L \wedge p \notin L \wedge s \neq p$
$\longrightarrow env(\!|s{:}T|\!)(\!|p{:}param\ (the(T\hat{}l))|\!) \vdash (t^{[Fvar\ s, Fvar\ p]}) : return\ (the(T\hat{}l)))$
for *L env t T l*
from *assms* **show** *?thesis*
proof (*induct*
taking: $\lambda env\ t\ T\ l.\ \forall x\ X.\ x \notin env\text{-}dom\ env$
$\longrightarrow (\exists L.\ finite\ L \wedge pred\text{-}cof\ L\ (env(\!|x{:}X|\!))\ t\ T\ l)$
rule: *typing-induct*)
case *Call* **thus** *?case* **by** *auto*
next
case (*Fvar env Ta xa*) **thus** *?case*
proof (*intro strip*)
fix *x X* **assume** $x \notin env\text{-}dom\ env$
from
get-env-smaller[*OF* ‹*xa* ∈ *env-dom env*› *this*]
T-Var[*OF ok-add-ok*[*OF* ‹*ok env*› *this*]
env-bigger[*OF this* ‹*xa* ∈ *env-dom env*›]]
‹*the env*!*xa* = *Ta*›
show $env(\!|x{:}X|\!) \vdash Fvar\ xa : Ta$ **by** *simp*
qed
next
case (*Obj env Ta f*) **note** *pred-o* = *this*(*3*)
define *pred-cof′*
where *pred-cof′ x X b l* $\longleftrightarrow$ (∃ *L*. *finite L* ∧ *pred-cof L* (*env*(|*x*:*X*|)) (*the b*) *Ta l*) **for** *x X b l*
from *pred-o*
have *pred*: $\forall x\ X.\ x \notin env\text{-}dom\ env \longrightarrow (\forall l{\in}dom\ f.\ pred\text{-}cof'\ x\ X\ (f\ l)\ l)$
by (*intro fmap-ball-all2′*[*of f* $\lambda x\ X.\ x \notin env\text{-}dom\ env$ *pred-cof*′],
unfold pred-cof-def pred-cof′-def, *simp*)
show *?case*
proof (*intro strip*)
fix *x X*

define *pred-bnd*
where *pred-bnd s p b l* $\longleftrightarrow$
env$(\!|x{:}X|\!)(\!|s{:}Ta|\!)(\!|p{:}param\ (the(Ta\frown l))|\!) \vdash (the\ b^{[Fvar\ s, Fvar\ p]}) : return\ (the(Ta\frown l))$
for *s p b l*
assume $x \notin$ *env-dom env*
with *pred fmap-ex-cof*[*of f pred-bnd*] ‹*dom f* = *do Ta*›
obtain *L* **where**
finite L **and** $\forall\, l{\in}do\ Ta.$ *pred-cof L* (*env*$(\!|x{:}X|\!)$) (*the*(*f l*)) *Ta l*
unfolding *pred-bnd-def pred-cof-def pred-cof′-def*
by *auto*
from
T-Obj[*OF ok-add-ok*[*OF* ‹*ok env*› ‹$x \notin$ *env-dom env*›]
‹*dom f* = *do Ta*› *this*(*1*)]
this(*2*)
show *env*$(\!|x{:}X|\!) \vdash$ *Obj f Ta* : *Ta*
unfolding *pred-cof-def*
by *simp*
qed
next
case (*Upd env Ta t l u*)
note *pred-t* = *this*(*2*) **and** *pred-u* = *this*(*4*)
show *?case*
proof (*intro strip*)
fix *x X* **assume** $x \notin$ *env-dom env*
with *pred-u* **obtain** *L* **where**
finite L **and** *pred-cof L* (*env*$(\!|x{:}X|\!)$) *u Ta l* **by** *auto*
with ‹$l \in$ *do Ta*› ‹$x \notin$ *env-dom env*› *pred-t*
show *env*$(\!|x{:}X|\!) \vdash$ *Upd t l u* : *Ta*
unfolding *pred-cof-def*
by *auto*
qed
next
case (*Bnd env Ta l t L*) **note** *pred* = *this*(*3*)
show *?case*
proof (*intro strip*)
fix *x X* **assume** $x \notin$ *env-dom env*
thus $\exists\, L.$ *finite L* $\wedge$ *pred-cof L* (*env*$(\!|x{:}X|\!)$) *t Ta l*
proof (*rule-tac x* = $L \cup \{x\}$ **in** *exI*, *simp add*: ‹*finite L*›,
unfold pred-cof-def, *auto*)
fix *s p*
assume
$s \notin L$ **and** $p \notin L$ **and** $s \neq p$ **and**
$s \neq x$ **and** $p \neq x$
note
subst-add[*OF not-sym*[*OF* ‹$s \neq x$›]]
subst-add[*OF not-sym*[*OF* ‹$p \neq x$›]]
from
this(*1*)[*of env X Ta*] *this*(*2*)[*of env*$(\!|s{:}Ta|\!)$ *X param* (*the*(*Ta*$\frown$*l*))]

```
      pred ‹s ∉ L› ‹p ∉ L› ‹s ≠ p›
      not-in-env-bigger-2[OF ‹x ∉ env-dom env›
                             not-sym[OF ‹s ≠ x›] not-sym[OF ‹p ≠ x›]]
    show
      env(|x:X|)(|s:Ta|)(|p:param (the(Ta^l))|)
      ⊢ (t[Fvar s,Fvar p]) : return (the(Ta^l))
      by auto
  qed
 qed
 qed
qed

lemma bnd-disj-env-lem:
 assumes
 ok e1 and env-dom e1 ∩ env-dom e2 = {} and ok e2 and
 ∀ s p. s ∉ F ∧ p ∉ F ∧ s ≠ p
  ⟶ e1(|s:T|)(|p:param(the(T^l))|)
     ⊢ (t2[Fvar s,Fvar p]) : return(the(T^l))
     ∧ (env-dom (e1(|s:T|)(|p:param(the(T^l))|)) ∩ env-dom e2 = {}
  ⟶ ok e2
  ⟶ e1(|s:T|)(|p:param(the(T^l))|)+e2
     ⊢ (t2[Fvar s,Fvar p]) : return(the(T^l)))
 shows
 ∀ s p. s ∉ F ∪ env-dom (e1+e2) ∧ p ∉ F ∪ env-dom (e1+e2) ∧ s ≠ p
  ⟶ (e1+e2)(|s:T|)(|p:param(the(T^l))|) ⊢ (t2[Fvar s,Fvar p]) : return(the(T^l))
proof (intro strip, elim conjE)
 fix s p assume
  nin-s: s ∉ F ∪ env-dom (e1+e2) and
  nin-p: p ∉ F ∪ env-dom (e1+e2) and s ≠ p
 from
  this(1−2) env-add-dom-2[OF assms(1) - - this(3)]
  assms(2) env-app-dom[OF assms(1−3)]
 have env-dom (e1(|s:T|)(|p:param(the(T^l))|)) ∩ env-dom e2 = {} by simp
 with
  env-app-add2[OF assms(1−3) - - - - ‹s ≠ p›]
  env-app-dom[OF assms(1−3)] ‹ok e2› assms(4) nin-s nin-p ‹s ≠ p›
 show (e1+e2)(|s:T|)(|p:param(the(T^l))|) ⊢ (t2[Fvar s,Fvar p]) : return(the(T^l))
  by auto
qed


lemma disjunct-env:
 assumes e ⊢ t : A
 shows (env-dom e ∩ env-dom e' = {} ⟹ ok e' ⟹ e + e' ⊢ t : A)
 using assms
proof (induct rule: typing.induct)
 case T-Call thus ?case by auto
next
```

case (*T-Var env x T*)
from
env-app-dom[*OF* ‹*ok env*› ‹*env-dom env* ∩ *env-dom e′* = {}› ‹*ok e′*›]
‹*x* ∈ *env-dom env*›
have *indom*: *x* ∈ *env-dom* (*env*+*e′*) **by** *simp*
from
‹*ok env*› ‹*x* ∈ *env-dom env*› ‹*the env*!*x* = *T*› ‹*env-dom env* ∩ *env-dom e′* = {}›
‹*ok e′*›
have *the* (*env*+*e′*)!*x* = *T* **by** *simp*
from
typing.T-Var[*OF env-app-ok*[*OF* ‹*ok env*› ‹*env-dom env* ∩ *env-dom e′* = {}›
‹*ok e′*›]
indom this]
show *?case* **by** *assumption*
next
case (*T-Upd F env T l t2 t1*)
from
typing.T-Upd[*OF* - *bnd-disj-env-lem*[*OF typing-regular′*[*OF* ‹*env* ⊢ *t1* : *T*›]
‹*env-dom env* ∩ *env-dom e′* = {}› ‹*ok e′*›
T-Upd(*2*)]
T-Upd(*4*)[*OF* ‹*env-dom env* ∩ *env-dom e′* = {}› ‹*ok e′*›]
‹*l* ∈ *do T*›]
‹*finite F*› *ok-finite*[*OF env-app-ok*[*OF typing-regular′*[*OF* ‹*env* ⊢ *t1* : *T*›]
‹*env-dom env* ∩ *env-dom e′* = {}› ‹*ok e′*›]]
show *?case* **by** *simp*
next
case (*T-Obj env f T F*)
from
ok-finite[*OF env-app-ok*[*OF* ‹*ok env*› ‹*env-dom env* ∩ *env-dom e′* = {}› ‹*ok e′*›]]
‹*finite F*›
have *finF*: *finite* (*F* ∪ *env-dom* (*env*+*e′*)) **by** *simp*
note
ball-Tltsp[*of F*
λ*T l t s p*. *env*(|*s*:*T*|)(|*p*:*param*(*the*(*T*^*l*))|) ⊢ (*t*[*Fvar s*,*Fvar p*]) : *return*(*the*(*T*^*l*))
∧ (*env-dom* (*env*(|*s*:*T*|)(|*p*:*param*(*the*(*T*^*l*))|)) ∩ *env-dom e′* = {}
⟶ *ok e′*
⟶ *env*(|*s*:*T*|)(|*p*:*param*(*the*(*T*^*l*))|)+*e′*
⊢ (*t*[*Fvar s*,*Fvar p*]) : *return*(*the*(*T*^*l*)))
T F ∪ *env-dom* (*env*+*e′*)
λ*T l t s p*. (*env*+*e′*)(|*s*:*T*|)(|*p*:*param*(*the*(*T*^*l*))|)
⊢ (*t*[*Fvar s*,*Fvar p*]) : *return*(*the*(*T*^*l*))]
from
this[*OF* - *T-Obj*(*4*)]
bnd-disj-env-lem[*OF* ‹*ok env*› ‹*env-dom env* ∩ *env-dom e′* = {}› ‹*ok e′*›]
typing.T-Obj[*OF env-app-ok*[*OF* ‹*ok env*›
‹*env-dom env* ∩ *env-dom e′* = {}› ‹*ok e′*›]
‹*dom f* = *do T*› *finF*]
show *?case* **by** *simp*
qed

Typed in the Empty Environment implies typed in any Environment

lemma *empty-env*:
assumes (*Env Map.empty*) $\vdash$ t : A **and** *ok env*
shows *env* $\vdash$ t : A
proof −
from ‹*ok env*› **have** *env* = (*Env Map.empty*)+*env* **by** (*cases env*, *auto*)
with *disjunct-env*[*OF assms*(*1*) - *assms*(*2*)] **show** *?thesis* **by** *simp*
qed

lemma *bnd-open-lem*:
assumes
pred-bnd:
$\forall$ *sa pa*. *sa* $\notin$ *F* $\wedge$ *pa* $\notin$ *F* $\wedge$ *sa* $\neq$ *pa*
$\longrightarrow$ *env*⟪*sa*:*T*⟫⟪*pa*:*param*(*the*(*T*^*l*))⟫
$\vdash$ ($t^{[Fvar\ sa,Fvar\ pa]}$) : *return*(*the*(*T*^*l*))
$\wedge$ ($\forall$ *env*′′ *t*′′ *s*′ *p*′ *x*′ *y*′ *A*′ *B*′ *n*′. *s*′ $\notin$ *FV t*′′ $\cup$ *FV x*′ $\cup$ *FV y*′
$\longrightarrow$ *p*′ $\notin$ *FV t*′′ $\cup$ *FV x*′ $\cup$ *FV y*′ $\longrightarrow$ *s*′ $\neq$ *p*′
$\longrightarrow$ *env*′′ $\vdash$ *x*′ : *A*′ $\longrightarrow$ *env*′′ $\vdash$ *y*′ : *B*′
$\longrightarrow$ ($t^{[Fvar\ sa,Fvar\ pa]}$) = {*n*′ $\rightarrow$ [*Fvar s*′,*Fvar p*′]} *t*′′
$\longrightarrow$ *env*⟪*sa*:*T*⟫⟪*pa*:*param*(*the*(*T*^*l*))⟫ = *env*′′⟪*s*′:*A*′⟫⟪*p*′:*B*′⟫
$\longrightarrow$ *env*′′ $\vdash$ {*n*′ $\rightarrow$ [*x*′,*y*′]} *t*′′ : *return*(*the*(*T*^*l*))) **and**
ok env **and** *env* = *env*′⟪*s*:*A*⟫⟪*p*:*B*⟫ **and**
s $\notin$ *FV t*′′ $\cup$ *FV x* $\cup$ *FV y* **and** *p* $\notin$ *FV t*′′ $\cup$ *FV x* $\cup$ *FV y* **and** *s* $\neq$ *p* **and**
env′ $\vdash$ *x* : *A* **and** *env*′ $\vdash$ *y* : *B* **and**
t = {*Suc n* $\rightarrow$ [*Fvar s*,*Fvar p*]} *t*′ **and** *FV t*′ $\subseteq$ *FV t*′′
shows
$\forall$ *sa pa*. *sa* $\notin$ *F* $\cup$ {*s*,*p*} $\cup$ *env-dom env*′
$\wedge$ *pa* $\notin$ *F* $\cup$ {*s*,*p*} $\cup$ *env-dom env*′ $\wedge$ *sa* $\neq$ *pa*
$\longrightarrow$ *env*′⟪*sa*:*T*⟫⟪*pa*:*param*(*the*(*T*^*l*))⟫
$\vdash$ ({*Suc n* $\rightarrow$ [*x*,*y*]} $t'^{[Fvar\ sa,\ Fvar\ pa]}$) : *return*(*the*(*T*^*l*))
proof (*intro strip*, *elim conjE*)
fix *sa pa* **assume**
nin-sa: *sa* $\notin$ *F* $\cup$ {*s*,*p*} $\cup$ *env-dom env*′ **and**
nin-pa: *pa* $\notin$ *F* $\cup$ {*s*,*p*} $\cup$ *env-dom env*′ **and** *sa* $\neq$ *pa*
hence *sa* $\notin$ *F* $\wedge$ *pa* $\notin$ *F* $\wedge$ *sa* $\neq$ *pa* **by** *auto*
moreover
{
fix *a* **assume** *a* $\notin$ *FV t*′′ $\cup$ *FV x* $\cup$ *FV y* **and** *a* $\in$ {*s*,*p*}
with
‹*FV t*′ $\subseteq$ *FV t*′′› *nin-sa nin-pa* ‹*sa* $\neq$ *pa*›
sopen-FV[*of 0 Fvar sa Fvar pa t*′]
have *a* $\notin$ *FV* ($t'^{[Fvar\ sa,Fvar\ pa]}$) $\cup$ *FV x* $\cup$ *FV y* **by** (*auto simp*: *openz-def*)
} note
this[*OF* ‹*s* $\notin$ *FV t*′′ $\cup$ *FV x* $\cup$ *FV y*›]
this[*OF* ‹*p* $\notin$ *FV t*′′ $\cup$ *FV x* $\cup$ *FV y*›]
moreover
{
from ‹*ok env*› ‹*env* = *env*′⟪*s*:*A*⟫⟪*p*:*B*⟫› *ok-add-2*[*of env*′ *s A p B*]

```
    have ok env' by simp
    from nin-sa nin-pa ‹sa ≠ pa› env-add-dom[OF this]
    obtain sa ∉ env-dom env' and pa ∉ env-dom (env'⟪sa:T⟫) by auto
    note
      bigger-env-lemma[OF bigger-env-lemma[OF ‹env' ⊢ x : A› this(1)] this(2)]
      bigger-env-lemma[OF bigger-env-lemma[OF ‹env' ⊢ y : B› this(1)] this(2)]
  }note
      this(1)[of param(the(T^l))]
      this(2)[of param(the(T^l))]
  moreover
  from ‹t = {Suc n → [Fvar s,Fvar p]} t'› sopen-commute[of 0 Suc n sa pa s p t']
  have (t[Fvar sa,Fvar pa]) = {Suc n → [Fvar s,Fvar p]} (t'[Fvar sa,Fvar pa])
    by (simp add: openz-def)
  moreover
  from
    subst-add[of p sa env'⟪s:A⟫ B T] subst-add[of s sa env' A T]
    subst-add[of p pa env'⟪sa:T⟫⟪s:A⟫ B param(the(T^l))]
    subst-add[of s pa env'⟪sa:T⟫ A param(the(T^l))]
    ‹env = env'⟪s:A⟫⟪p:B⟫› nin-sa nin-pa
  have env⟪sa:T⟫⟪pa:param(the(T^l))⟫ = env'⟪sa:T⟫⟪pa:param(the(T^l))⟫⟪s:A⟫⟪p:B⟫
    by auto
  ultimately
  show
    env'⟪sa:T⟫⟪pa:param(the(T^l))⟫
    ⊢ ({Suc n → [x,y]} t'[Fvar sa, Fvar pa]) : return(the(T^l))
    using
      pred-bnd ‹s ≠ p›
      sopen-commute-gen[OF lc-Fvar[of sa] lc-Fvar[of pa]
                           typing-regular''[OF ‹env' ⊢ x : A›]
                           typing-regular''[OF ‹env' ⊢ y : B›]
                           not-sym[OF Suc-not-Zero]]
    by (auto simp: openz-def)
qed



lemma open-lemma':
  shows
  e ⊢ t : C
  ⟹ (⋀env t' s p x y A B n. s ∉ FV t' ∪ FV x ∪ FV y
        ⟹ p ∉ FV t' ∪ FV x ∪ FV y ⟹ s ≠ p
        ⟹ env ⊢ x : A ⟹ env ⊢ y : B
        ⟹ t = {n → [Fvar s,Fvar p]} t'
        ⟹ e = env⟪s:A⟫⟪p:B⟫
        ⟹ env ⊢ {n → [x,y]} t' : C)
proof (induct set:typing)
  case (T-Var env x T env' t' s p y z A B n)
  from sopen-eq-Fvar[OF sym[OF ‹Fvar x = {n → [Fvar s,Fvar p]} t'›]]
  show ?case
```

```
  proof (elim disjE conjE)
    assume t' = Fvar x
    with ‹s ∉ FV t' ∪ FV y ∪ FV z› ‹p ∉ FV t' ∪ FV y ∪ FV z›
    obtain x ≠ s and x ≠ p by auto
    from ‹x ∈ env-dom env› ‹env = env'(|s:A|)(|p:B|)› in-env-smaller2[OF - this]
    have indom: x ∈ env-dom env' by simp
    from
      ‹ok env› ‹the env!x = T› ‹env = env'(|s:A|)(|p:B|)›
      ok-add-2[of env' s A p B] get-env-smaller2[OF this - - ‹s ≠ p›]
    have the env'!x = T by simp
    from
      ‹ok env› ‹env = env'(|s:A|)(|p:B|)› ‹t' = Fvar x›
      ok-add-2[of env' s A p B] typing.T-Var[OF - indom this]
    show ?case by simp
  next
    assume x = s
    with
      ‹ok env› ‹the env!x = T› ‹env = env'(|s:A|)(|p:B|)›
      add-get2-1[of env' s A p B]
    have T = A by simp
    moreover assume t' = Bvar (Self n)
    ultimately show ?thesis using ‹env' ⊢ y : A› by simp
  next
    assume x = p
    with
      ‹ok env› ‹the env!x = T› ‹env = env'(|s:A|)(|p:B|)›
      add-get2-2[of env' s A p B] have T = B by simp
    moreover assume t' = Bvar (Param n)
    ultimately show ?thesis using ‹env' ⊢ z : B› by simp
  qed
next
  case (T-Upd F env T l t2 t1 env' t' s p x y A B n)
  from sopen-eq-Upd[OF sym[OF ‹Upd t1 l t2 = {n → [Fvar s,Fvar p]} t'›]]
  obtain t1' t2' where
    t1': t1 = {n → [Fvar s,Fvar p]} t1' and
    t2': t2 = {Suc n → [Fvar s,Fvar p]} t2' and
    t': t' = Upd t1' l t2' by auto
  hence FV t2' ⊆ FV t' by auto
  from
    ‹s ∉ FV t' ∪ FV x ∪ FV y› ‹p ∉ FV t' ∪ FV x ∪ FV y›
    t' ‹finite F› ok-finite[OF typing-regular'[OF ‹env' ⊢ x : A›]]
    typing.T-Upd[OF - bnd-open-lem[OF T-Upd(2)
                    typing-regular'[OF ‹env ⊢ t1 : T›]
                    ‹env = env'(|s:A|)(|p:B|)›
                    ‹s ∉ FV t' ∪ FV x ∪ FV y›
                    ‹p ∉ FV t' ∪ FV x ∪ FV y› ‹s ≠ p›
                    ‹env' ⊢ x : A› ‹env' ⊢ y : B› t2' this]
    T-Upd(4)[OF - - ‹s ≠ p› ‹env' ⊢ x : A› ‹env' ⊢ y : B›
              t1' ‹env = env'(|s:A|)(|p:B|)›] ‹l ∈ do T›]
```

show *?case* **by** *simp*
next
case (*T-Obj env f T F env' t' s p x y A B n*)
from *sopen-eq-Obj*[*OF sym*[*OF* ‹*Obj f T* = {*n* → [*Fvar s*,*Fvar p*]} *t'*›]]
obtain *f'* **where**
obj: *Obj f T* = {*n* → [*Fvar s*,*Fvar p*]} *Obj f' T* **and**
t': *t'* = *Obj f' T* **by** *auto*
from
sym[*OF this*(*1*)] ‹*dom f* = *do T*›
sym[*OF dom-sopenoption-lem*[*of Suc n Fvar s Fvar p f'*]]
dom-sopenoption-lem[*of Suc n x y f'*]
have *dom*: *dom* (λ*l*. *sopen-option* (*Suc n*) *x y* (*f' l*)) = *do T* **by** *simp*

from ‹*finite F*› *ok-finite*[*OF typing-regular'*[*OF* ‹*env'* ⊢ *x* : *A*›]]
have *finF*: *finite* (*F* ∪ {*s*,*p*} ∪ *env-dom env'*)
by *simp*

have
∀ *l*∈*do T*. ∀ *sa pa*. *sa* ∉ *F* ∪ {*s*,*p*} ∪ *env-dom env'*
∧ *pa* ∉ *F* ∪ {*s*,*p*} ∪ *env-dom env'*
∧ *sa* ≠ *pa*
⟶ *env'*(|*sa*:*T*|)(|*pa*:*param*(*the*(*T*^*l*))|)
⊢ (*the*(*sopen-option* (*Suc n*) *x y* (*f' l*))[*Fvar sa*,*Fvar pa*]) : *return*(*the*(*T*^*l*))
proof
fix *l* **assume** *l* ∈ *do T* **with** *T-Obj*(*4*)
have
cof:
∀ *sa pa*. *sa* ∉ *F* ∧ *pa* ∉ *F* ∧ *sa* ≠ *pa*
⟶ *env*(|*sa*:*T*|)(|*pa*:*param*(*the*(*T*^*l*))|)
⊢ (*the*(*f l*)[*Fvar sa*,*Fvar pa*]) : *return*(*the*(*T*^*l*))
∧ (∀ *env'' t'' s' p' x' y' A' B' n'*.
s' ∉ *FV t''* ∪ *FV x'* ∪ *FV y'* ⟶ *p'* ∉ *FV t''* ∪ *FV x'* ∪ *FV y'*
⟶ *s'* ≠ *p'* ⟶ *env''* ⊢ *x'* : *A'* ⟶ *env''* ⊢ *y'* : *B'*
⟶ (*the*(*f l*)[*Fvar sa*,*Fvar pa*]) = {*n'* → [*Fvar s'*,*Fvar p'*]} *t''*
⟶ *env*(|*sa*:*T*|)(|*pa*:*param*(*the*(*T*^*l*))|) = *env''*(|*s'*:*A'*|)(|*p'*:*B'*|)
⟶ *env''* ⊢ {*n'* → [*x'*,*y'*]} *t''* : *return*(*the*(*T*^*l*)))
by *simp*
from
‹*l* ∈ *do T*› ‹*dom f* = *do T*› ‹*Obj f T* = {*n* → [*Fvar s*,*Fvar p*]} *t'*› *obj t'*
dom-sopenoption-lem[*of Suc n Fvar s Fvar p f'*]
have *indomf'*: *l* ∈ *dom f'* **by** *auto*
with *obj sopen-option-lem*[*of f' Suc n Fvar s Fvar p*] *FV-option-lem*[*of f'*] *t'*
obtain
the(*f l*) = {*Suc n* → [*Fvar s*,*Fvar p*]} *the*(*f' l*) **and**
FV (*the*(*f' l*)) ⊆ *FV t'* **by** *auto*
from
bnd-open-lem[*OF cof* ‹*ok env*› ‹*env* = *env'*(|*s*:*A*|)(|*p*:*B*|)›
‹*s* ∉ *FV t'* ∪ *FV x* ∪ *FV y*› ‹*p* ∉ *FV t'* ∪ *FV x* ∪ *FV y*›
‹*s* ≠ *p*› ‹*env'* ⊢ *x* : *A*› ‹*env'* ⊢ *y* : *B*› *this*]

```
    indomf' sopen-option-lem[of f' Suc n x y] T-Obj(4)
  show
    ∀ sa pa. sa ∉ F ∪ {s,p} ∪ env-dom env'
          ∧ pa ∉ F ∪ {s,p} ∪ env-dom env' ∧ sa ≠ pa
      ⟶ env'⟪sa:T⟫⟪pa:param(the(T^l))⟫
        ⊢ (the(sopen-option (Suc n) x y (f' l))[Fvar sa,Fvar pa]) : return(the(T^l))
    by simp
 qed
 from typing.T-Obj[OF typing-regular'[OF ‹env' ⊢ x : A›] dom finF this] t'
 show ?case by simp
next
 case (T-Call env t1 T t2 l env' t' s p x y A B n)
 from sopen-eq-Call[OF sym[OF ‹Call t1 l t2 = {n → [Fvar s,Fvar p]} t'›]]
 obtain t1' t2' where
   t1: t1 = {n → [Fvar s,Fvar p]} t1' and
   t2: t2 = {n → [Fvar s,Fvar p]} t2' and
   t': t' = Call t1' l t2' by auto
 { fix a assume a ∉ FV t' ∪ FV x ∪ FV y
   with t' have a ∉ FV t1' ∪ FV x ∪ FV y by simp
 }note
    t1' = T-Call(2)[OF this[OF ‹s ∉ FV t' ∪ FV x ∪ FV y›]
                       this[OF ‹p ∉ FV t' ∪ FV x ∪ FV y›]
                       ‹s ≠ p› ‹env' ⊢ x : A› ‹env' ⊢ y : B›
                       t1 ‹env = env'⟪s:A⟫⟪p:B⟫›]
 { fix a assume a ∉ FV t' ∪ FV x ∪ FV y
   with t' have a ∉ FV t2' ∪ FV x ∪ FV y by simp
 }
 from
   typing.T-Call[OF t1' T-Call(4)[OF this[OF ‹s ∉ FV t' ∪ FV x ∪ FV y›]
                                     this[OF ‹p ∉ FV t' ∪ FV x ∪ FV y›]
                                     ‹s ≠ p› ‹env' ⊢ x : A› ‹env' ⊢ y : B›
                                     t2 ‹env = env'⟪s:A⟫⟪p:B⟫›]
                 ‹l ∈ do T›]
   t'
 show ?case by simp
qed

lemma open-lemma:
 ⟦ env⟪s:A⟫⟪p:B⟫ ⊢ {n → [Fvar s,Fvar p]} t : T;
   s ∉ FV t ∪ FV x ∪ FV y;  p ∉ FV t ∪ FV x ∪ FV y; s ≠ p;
   env ⊢ x : A; env ⊢ y : B ⟧
 ⟹ env ⊢ {n → [x,y]} t : T
 by (simp add: open-lemma')
```

5.0.4 Subject reduction

```
lemma type-dom[simp]: env ⊢ (Obj a A) : A ⟹ dom a = do A
 by (erule typing.cases, auto)
```

lemma *select-preserve-type*[*simp*]:
assumes
$env \vdash Obj\ f\ (Object\ t) : Object\ t$ **and** $s \notin FV\ a$ **and** $p \notin FV\ a$ **and**
$env(\!|s{:}(Object\ t)|\!)(\!|p{:}param(the(t\ l2))|\!) \vdash (a^{[Fvar\ s, Fvar\ p]}) : return(the(t\ l2))$ **and**
$l1 \in dom\ t$ **and** $l2 \in dom\ t$
shows
$\exists F.\ finite\ F$
$\wedge\ (\forall s\ p.\ s \notin F \wedge p \notin F \wedge s \neq p$
$\longrightarrow env(\!|s{:}(Object\ t)|\!)(\!|p{:}param(the(t\ l1))|\!)$
$\vdash (the((f(l2 \mapsto a))\ l1)^{[Fvar\ s, Fvar\ p]}) : return(the(t\ l1)))$
proof −
from *ok-finite*[*OF typing-regular'*[*OF* ‹$env \vdash Obj\ f\ (Object\ t) : Object\ t$›]]
have *finF*: *finite* ($\{s,p\} \cup$ *env-dom env*) **by** *simp*

{
note
ok-env = *typing-regular'*[*OF* ‹$env \vdash Obj\ f\ (Object\ t) : Object\ t$›] **and**
ok-env-sp = *typing-regular'*[*OF assms*(*4*)]
fix *sa pa* **assume**
nin-sa: $sa \notin \{s,p\} \cup$ *env-dom env* **and**
nin-pa: $pa \notin \{s,p\} \cup$ *env-dom env* **and** $sa \neq pa$
from *this*(*1*) *ok-add-2*[*OF ok-env-sp*] *env-add-dom-2*[*OF ok-env*]
have $sa \notin$ *env-dom* ($env(\!|s{:}Object\ t|\!)(\!|p{:}param(the(t\ l2))|\!)$) **by** *simp*
from
nin-sa bigger-env-lemma[*OF assms*(*4*) *this*]
subst-add[*of sa p* $env(\!|s{:}Object\ t|\!)$ *Object t param*(*the*(*t l2*))]
subst-add[*of sa s env Object t Object t*]
have
aT-sa: $env(\!|sa{:}Object\ t|\!)(\!|s{:}Object\ t|\!)(\!|p{:}param(the(t\ l2))|\!)$
$\vdash (a^{[Fvar\ s, Fvar\ p]}) : return(the(t\ l2))$ **by** *simp*
from
‹$sa \neq pa$› *nin-sa nin-pa env-add-dom*[*OF ok-env*]
ok-add-2[*OF ok-env-sp*]
obtain
$s \notin$ *env-dom* ($env(\!|sa{:}Object\ t|\!)$) **and**
$p \notin$ *env-dom* ($env(\!|sa{:}Object\ t|\!)$) **and** $s \neq p$ **and**
$sa \notin$ *env-dom env* **and** $pa \notin$ *env-dom* ($env(\!|sa{:}Object\ t|\!)$)
by *auto*
with *env-add-dom-2*[*OF ok-add-ok*[*OF ok-env this*(*4*)] *this*(*1−3*)] *nin-pa*
have $pa \notin$ *env-dom* ($env(\!|sa{:}Object\ t|\!)(\!|s{:}Object\ t|\!)(\!|p{:}param(the(t\ l2))|\!)$)
by *simp*
from
nin-pa bigger-env-lemma[*OF aT-sa this*]
subst-add[*of pa p* $env(\!|sa{:}Object\ t|\!)(\!|s{:}Object\ t|\!)$
param(*the*(*t l2*)) *param*(*the*(*t l2*))]
subst-add[*of pa s* $env(\!|sa{:}Object\ t|\!)$ *param*(*the*(*t l2*)) *Object t*]
have
aT-sapa:
$env(\!|sa{:}Object\ t|\!)(\!|pa{:}param(the(t\ l2))|\!)(\!|s{:}Object\ t|\!)(\!|p{:}param(the(t\ l2))|\!)$

```
    ⊢ {0 → [Fvar s, Fvar p]} a : return(the(t l2)) by (simp add: openz-def)
  from nin-sa nin-pa ‹s ∉ FV a› ‹p ∉ FV a› ok-add-2[OF ok-env-sp]
  obtain
    ninFV-s: s ∉ FV a ∪ FV (Fvar sa) ∪ FV (Fvar pa) and
    ninFV-p: p ∉ FV a ∪ FV (Fvar sa) ∪ FV (Fvar pa) and s ≠ p
    by auto
  from ok-add-2[OF typing-regular'[OF aT-sapa]]
  have ok-env-sapa: ok (env(|sa:Object t|)(|pa:param(the(t l2))|))
    by simp
  with ok-add-reverse[OF this]
  have ok-env-pasa: ok (env(|pa:param(the(t l2))|)(|sa:Object t|))
    by simp

  from
    open-lemma[OF aT-sapa ninFV-s ninFV-p ‹s ≠ p› -
                  T-Var[OF ok-env-sapa in-add[OF ok-env-sapa]
                        add-get2-2[OF ok-env-sapa]]]
    T-Var[OF ok-env-pasa in-add[OF ok-env-pasa] add-get2-2[OF ok-env-pasa]]
    ok-add-reverse[OF ok-env-sapa]
  have
    env(|sa:(Object t)|)(|pa:param(the(t l2))|)
    ⊢ (a^[Fvar sa,Fvar pa]) : return(the(t l2))
    by (simp add: openz-def)
}note alem = this


show ?thesis
proof (cases l1 = l2)
  case True with assms obj-inv-elim'[OF assms(1)] show ?thesis
    by (simp (no-asm-simp), rule-tac x = {s,p} ∪ env-dom env in exI,
        auto simp: finF alem)
next
  case False
  from obj-inv-elim[OF ‹env ⊢ Obj f (Object t) : Object t›]
  obtain F where
    finite F and
    ∀ l∈dom t.
      ∀ s p. s ∉ F ∧ p ∉ F ∧ s ≠ p
        ⟶ env(|s:Object t|)(|p:param(the(Object t`l))|)
          ⊢ (the(f l)^[Fvar s,Fvar p]) : return(the(Object t`l))
    by auto
  from this(2) ‹l1 ∈ dom t›
  have
    ∀ s p. s ∉ F ∧ p ∉ F ∧ s ≠ p
      ⟶ env(|s:Object t|)(|p:param(the(Object t`l1))|)
        ⊢ (the(f l1)^[Fvar s,Fvar p]) : return(the(Object t`l1))
    by auto
  thus ?thesis using ‹finite F› ‹l1 ≠ l2› by (simp,blast)
qed
```

qed

Main Lemma

lemma *subject-reduction*: $e \vdash t : T \Longrightarrow (\bigwedge t'.\ t \to_\beta t' \Longrightarrow e \vdash t' : T)$
proof (*induct set*: *typing*)
case (*T-Var env x T t'*)
from *Fvar-beta*[*OF* ‹*Fvar x* $\to_\beta$ *t'*›] **show** *?case* **by** *simp*
next
case (*T-Upd F env T l t2 t1 t'*)
from *Upd-beta*[*OF* ‹*Upd t1 l t2* $\to_\beta$ *t'*›] **show** *?case*
proof (*elim disjE exE conjE*)
fix *t1'* **assume** *t1* $\to_\beta$ *t1'* **and** *t'* = *Upd t1' l t2*
from
this(*2*) *T-Upd*(*2*)
typing.T-Upd[*OF* ‹*finite F*› - *T-Upd*(*4*)[*OF this*(*1*)] ‹*l* ∈ *do T*›]
show *?case* **by** *simp*
next
fix *t2' F'*
assume
finite F' **and**
pred-F': $\forall s\ p.\ s \notin F' \wedge p \notin F' \wedge s \neq p \longrightarrow (\exists t''.\ t2^{[Fvar\ s, Fvar\ p]} \to_\beta t'' \wedge t2' = \sigma[s,p]\ t'')$ **and**
t': *t'* = *Upd t1 l t2'*
have
$\forall s\ p.\ s \notin F \cup F' \wedge p \notin F \cup F' \wedge s \neq p \longrightarrow env(\!|s{:}T|\!)(\!|p{:}param(the(T\hat{}l))|\!) \vdash (t2'^{[Fvar\ s, Fvar\ p]}) : return(the(T\hat{}l))$
proof (*intro strip*, *elim conjE*)
fix *s p* **assume**
nin-s: *s* ∉ *F* ∪ *F'* **and**
nin-p: *p* ∉ *F* ∪ *F'* **and** *s* ≠ *p*
with *pred-F'* **obtain** *t''* **where** $t2^{[Fvar\ s, Fvar\ p]} \to_\beta t''$ **and** *t2'* = *σ*[*s*,*p*] *t''*
by *auto*
with *beta-lc*[*OF this*(*1*)] *sopen-sclose-eq-t*[*of t'' 0 s p*]
have $t2^{[Fvar\ s, Fvar\ p]} \to_\beta (t2'^{[Fvar\ s, Fvar\ p]})$
by (*simp add*: *openz-def closez-def*)
with *nin-s nin-p* ‹*s* ≠ *p*› *T-Upd*(*2*)
show $env(\!|s{:}T|\!)(\!|p{:}param(the(T\hat{}l))|\!) \vdash (t2'^{[Fvar\ s, Fvar\ p]}) : return(the(T\hat{}l))$
by *auto*
qed
from *t'* ‹*finite F*› ‹*finite F'*› *typing.T-Upd*[*OF* - *this* ‹*env* ⊢ *t1* : *T*› ‹*l* ∈ *do T*›]
show *?case* **by** *simp*
next
fix *f U* **assume**
l ∈ *dom f* **and** *Obj f U* = *t1* **and**
t': *t'* = *Obj* (*f*(*l* ↦ *t2*)) *U*
from *this*(*1*−*2*) ‹*env* ⊢ *t1* : *T*› *obj-inv*[*of env f U T*]
obtain *t* **where**
objT: *env* ⊢ *Obj f* (*Object t*) : (*Object t*) **and**
Object t = *T* **and** *T* = *U*

by (*cases T*, *auto*)
from *obj-inv-elim*[*OF objT*] ‹*Object t* = *T*› ‹*l* ∈ *dom f*›
have *domf'*: *dom* (*f*(*l* ↦ *t2*)) = *do T* **by** *auto*
have
exF: ∀ *l'*∈*do T*.
(∃ *F'*. *finite F'*
∧ (∀ *s p*. *s* ∉ *F'* ∪ (*F* ∪ *FV t2*) ∧ *p* ∉ *F'* ∪ (*F* ∪ *FV t2*) ∧ *s* ≠ *p*
⟶ *env*⦇*s*:*T*⦈⦇*p*:*param*(*the*(*T*^*l'*))⦈
⊢ (*the* ((*f*(*l* ↦ *t2*)) *l'*)[*Fvar s*,*Fvar p*]) : *return*(*the*(*T*^*l'*))))
proof
fix *l'* **assume** *l'* ∈ *do T*
with *dom-lem*[*OF objT*] ‹*l* ∈ *dom f*› ‹*Object t* = *T*›
obtain *ll'*: *l'* ∈ *dom t* **and** *l* ∈ *dom t* **by** *auto*

from ‹*finite F*› **have** *finite* (*F* ∪ *FV t2*) **by** *simp*
from *exFresh-s-p-cof*[*OF this*]
obtain *s p* **where**
nin-s: *s* ∉ *F* ∪ *FV t2* **and**
nin-p: *p* ∉ *F* ∪ *FV t2* **and** *s* ≠ *p*
by *auto*
with *T-Upd*(*2*) ‹*Object t* = *T*›
have
env⦇*s*:*Object t*⦈⦇*p*:*param*(*the*(*t l*))⦈
⊢ (*t2*[*Fvar s*,*Fvar p*]) : *return*(*the*(*t l*))
by *auto*
from
select-preserve-type[*OF objT - - this ll'*] *sym*[*OF* ‹*Object t* = *T*›]
nin-s nin-p ‹*l* ∈ *dom t*›
obtain *F'* **where**
finite F' **and**
∀ *s p*. *s* ∉ *F'* ∧ *p* ∉ *F'* ∧ *s* ≠ *p*
⟶ *env*⦇*s*:*T*⦈⦇*p*:*param*(*the*(*T*^*l'*))⦈
⊢ (*the* ((*f*(*l* ↦ *t2*)) *l'*)[*Fvar s*,*Fvar p*]) : *return*(*the*(*T*^*l'*))
by *auto*
thus
∃ *F'*. *finite F'*
∧ (∀ *s p*. *s* ∉ *F'* ∪ (*F* ∪ *FV t2*) ∧ *p* ∉ *F'* ∪ (*F* ∪ *FV t2*) ∧ *s* ≠ *p*
⟶ *env*⦇*s*:*T*⦈⦇*p*:*param*(*the*(*T*^*l'*))⦈
⊢ (*the* ((*f*(*l* ↦ *t2*)) *l'*)[*Fvar s*,*Fvar p*]) : *return*(*the*(*T*^*l'*)))
by *blast*
qed
{ fix *Ta* **from** *finite-dom-fmap* **have** *finite* (*do Ta*) **by** (*cases Ta*, *auto*) **}**
note *fin-doT* = *this ball-ex-finite*[*of do T F* ∪ *FV t2*]
from *this*(*2*)[*OF this*(*1*)[*of T*] - *exF*] ‹*finite F*›
obtain *F'* **where**
finite F' **and**
∀ *l'*∈*do T*. ∀ *s p*. *s* ∉ *F'* ∪ (*F* ∪ *FV t2*) ∧ *p* ∉ *F'* ∪ (*F* ∪ *FV t2*) ∧ *s* ≠ *p*
⟶ *env*⦇*s*:*T*⦈⦇*p*:*param*(*the*(*T*^*l'*))⦈

$\vdash$ (*the* ((*f*($l \mapsto$ *t2*)) *l'*)$^{[Fvar\ s, Fvar\ p]}$) : *return*(*the*(*T*^*l'*))

by *auto*

moreover

from ‹*finite F'*› ‹*finite F*› **have** *finite* (*F'* ∪ (*F* ∪ *FV t2*)) **by** *simp*

note *typing.T-Obj*[*OF typing-regular'*[*OF* ‹*env* ⊢ *t1* : *T*›] *domf' this*]

ultimately show *?case* **using** *t'* ‹*T* = *U*› **by** *auto*

qed

next

case (*T-Obj env f T F t'*)

from *Obj-beta*[*OF* ‹*Obj f T* →$_\beta$ *t'*›] **show** *?case*

proof (*elim exE conjE*)

fix *l f' a a' F'* **assume**

dom f = *dom f'* **and** *f* = *f'*(*l* ↦ *a*) **and** *l* ∈ *dom f'* **and**

t': *t'* = *Obj* (*f'*(*l* ↦ *a'*)) *T* **and** *finite F'* **and**

red-sp: ∀ *s p*. *s* ∉ *F'* ∧ *p* ∉ *F'* ∧ *s* ≠ *p*

⟶ (∃ *t''*. *a*$^{[Fvar\ s,\ Fvar\ p]}$ →$_\beta$ *t''* ∧ *a'* = *σ*[*s*,*p*] *t''*)

from *this*(*2*) ‹*dom f* = *do T*› **have** *domf'*: *dom* (*f'*(*l* ↦ *a'*)) = *do T* **by** *auto*

have

exF: ∀ *l'*∈*do T*. ∀ *s p*. *s* ∉ *F* ∪ *F'* ∧ *p* ∉ *F* ∪ *F'* ∧ *s* ≠ *p*

⟶ *env*⦇*s*:*T*⦈⦇*p*:*param*(*the*(*T*^*l'*))⦈

⊢ (*the* ((*f'*(*l* ↦ *a'*)) *l'*)$^{[Fvar\ s,Fvar\ p]}$) : *return*(*the*(*T*^*l'*))

proof (*intro strip*, *elim conjE*)

fix *l' s p* **assume**

l' ∈ *do T* **and**

nin-s: *s* ∉ *F* ∪ *F'* **and**

nin-p: *p* ∉ *F* ∪ *F'* **and** *s* ≠ *p*

with *red-sp* **obtain** *t''* **where** *a*$^{[Fvar\ s,Fvar\ p]}$ →$_\beta$ *t''* **and** *a'* = *σ*[*s*,*p*] *t''*

by *auto*

with

beta-lc[*OF this*(*1*)] *sopen-sclose-eq-t*[*of t'' 0 s p*]

‹*f* = *f'*(*l* ↦ *a*)›

have *the* (*f l*)$^{[Fvar\ s,Fvar\ p]}$ →$_\beta$ (*the*((*f'*(*l* ↦ *a'*)) *l*)$^{[Fvar\ s,Fvar\ p]}$)

by (*simp add*: *openz-def closez-def*)

with *T-Obj*(*4*) *nin-s nin-p* ‹*s* ≠ *p*› ‹*l'* ∈ *do T*› ‹*f* = *f'*(*l* ↦ *a*)›

show

env⦇*s*:*T*⦈⦇*p*:*param*(*the*(*T*^*l'*))⦈

⊢ (*the*((*f'*(*l* ↦ *a'*)) *l'*)$^{[Fvar\ s,Fvar\ p]}$) : *return*(*the*(*T*^*l'*))

by *auto*

qed

from *typing.T-Obj*[*OF* ‹*ok env*› *domf'* - *this*] ‹*finite F*› ‹*finite F'*› *t'*

show *?case* **by** (*simp* (*no-asm-simp*))

qed

next

case (*T-Call env t1 T t2 l t'*)

from *Call-beta*[*OF* ‹*Call t1 l t2* →$_\beta$ *t'*›] **show** *?case*

proof (*elim disjE conjE exE*)

fix *t1'* **assume** *t1* →$_\beta$ *t1'* **and** *t'* = *Call t1' l t2*

from

typing.T-Call[*OF T-Call*(*2*)[*OF this*(*1*)]

$\langle env \vdash t2 : param(the(T\hat{}l))\rangle\ \langle l \in do\ T\rangle]$
this(*2*)
show *?case* **by** *simp*
next
fix *t2'* **assume** $t2 \to_\beta t2'$ **and** $t' = Call\ t1\ l\ t2'$
from
typing.T-Call[*OF* $\langle env \vdash t1 : T\rangle$ *T-Call*(*4*)[*OF this*(*1*)] $\langle l \in do\ T\rangle$]
this(*2*)
show *?case* **by** *simp*
next
fix *f U* **assume** $Obj\ f\ U = t1$ **and** $l \in dom\ f$ **and** *t'*: $t' = (the(f\ l)^{[Obj\ f\ U, t2]})$
from
typing.T-Call[*OF* $\langle env \vdash t1 : T\rangle$ $\langle env \vdash t2 : param(the(T\hat{}l))\rangle$ $\langle l \in do\ T\rangle$]
sym[*OF this*(*1*)] $\langle env \vdash t1 : T\rangle$ $\langle env \vdash t2 : param(the(T\hat{}l))\rangle$
obj-inv[*of env f U T*]
obtain
objT: $env \vdash (Obj\ f\ T) : T$ **and** $T = U$ **and**
callT: $env \vdash Call\ (Obj\ f\ T)\ l\ t2 : return(the(T\hat{}l))$
by *auto*
have
$(\exists F.\ finite\ F$
$\wedge\ (\forall s\ p.\ s \notin F \wedge p \notin F \wedge s \neq p$
$\longrightarrow env(\!|s{:}T|\!)(\!|p{:}param(the(T\hat{}l))|\!)$
$\vdash (the(f\ l)^{[Fvar\ s, Fvar\ p]}) : return(the(T\hat{}l)))$
$\Longrightarrow env \vdash (the\ (f\ l)^{[Obj\ f\ T, t2]}) : return\ (the(T\hat{}l)))$
proof (*elim exE conjE*)
fix *F*
assume
finite F **and**
pred-F:
$\forall s\ p.\ s \notin F \wedge p \notin F \wedge s \neq p$
$\longrightarrow env(\!|s{:}T|\!)(\!|p{:}param(the(T\hat{}l))|\!)$
$\vdash (the(f\ l)^{[Fvar\ s, Fvar\ p]}) : return(the(T\hat{}l))$
from *this*(*1*) *finite-FV*[*of Obj f T*]
have *finite* $(F \cup FV\ (Obj\ f\ T) \cup FV\ t2)$ **by** *simp*
from *exFresh-s-p-cof*[*OF this*]
obtain *s p* **where**
nin-s: $s \notin F \cup FV\ (Obj\ f\ T) \cup FV\ t2$ **and**
nin-p: $p \notin F \cup FV\ (Obj\ f\ T) \cup FV\ t2$ **and** $s \neq p$
by *auto*
with *pred-F*
have
type-opened: $env(\!|s{:}T|\!)(\!|p{:}param(the(T\hat{}l))|\!)$
$\vdash \{0 \to [Fvar\ s, Fvar\ p]\}\ the(f\ l) : return(the(T\hat{}l))$
by (*auto simp*: *openz-def*)
from *nin-s nin-p FV-option-lem*[*of f*] *objT* $\langle l \in do\ T\rangle$
obtain
$s \notin FV\ (the(f\ l)) \cup FV\ (Obj\ f\ T) \cup FV\ t2$ **and**
$p \notin FV\ (the(f\ l)) \cup FV\ (Obj\ f\ T) \cup FV\ t2$ **by** *auto*

```
    from
      open-lemma[OF type-opened this ‹s ≠ p›
      objT ‹env ⊢ t2 : param(the(T^l))›]
    show ?thesis by (simp add: openz-def)
  qed
  with abs-typeE[OF callT] t' ‹T = U› show ?case by auto
 qed
qed

theorem subject-reduction': t →β* t' ⟹ e ⊢ t : T ⟹ e ⊢ t' : T
 by (induct set: rtranclp) (iprover intro: subject-reduction)+

lemma type-members-equal:
 fixes A :: type and B :: type
 assumes do A = do B and ∀ i. (A^i) = (B^i)
 shows A = B
proof (cases A)
 case (Object ta) thus ?thesis
 proof (cases B)
  case (Object tb)
  from ‹∀ i. (A^i) = (B^i)› ‹A = Object ta› ‹B = Object tb›
  have ⋀i. ta i = tb i by auto
  with ‹A = Object ta› ‹B = Object tb› show ?thesis by (simp add: ext)
 qed
qed

lemma not-var: Env Map.empty ⊢ a : A ⟹ ∀ x. a ≠ Fvar x
 by (rule allI, case-tac x, auto)

lemma Call-label-range: (Env Map.empty) ⊢ Call (Obj c T) l b : A ⟹ l ∈ dom
c
 by (erule typing-elims, erule typing.cases, simp-all)

lemma  Call-subterm-type: Env Map.empty ⊢ Call t l b: T
 ⟹ (∃ T'. Env Map.empty ⊢ t : T') ∧  (∃ T'. Env Map.empty ⊢ b : T')
 by (erule typing.cases) auto

lemma Upd-label-range: Env Map.empty ⊢ Upd (Obj c T) l x : A ⟹ l ∈ dom c
 by (erule typing-elims, erule typing.cases, simp-all)

lemma Upd-subterm-type:
 Env Map.empty ⊢ Upd t l x : T ⟹ ∃ T'. Env Map.empty ⊢ t : T'
 by (erule typing.cases) auto

lemma no-var: ∃ T. Env Map.empty ⊢ Fvar x : T ⟹ False
 by (case-tac x, auto)

lemma no-bvar: e ⊢ Bvar x : T ⟹ False
 by (erule typing.cases, auto)
```

5.0.5 Unique Type

theorem *type-unique*[*rule-format*]:
assumes *env* ⊢ *a*: *T*
shows ∀ *T'*. *env* ⊢ *a*: *T'* ⟶ *T* = *T'*
using *assms*
proof (*induct rule*: *typing.induct*)
case *T-Var* **thus** *?case* **by** (*auto simp*: *add-get-eq*)
next
case *T-Obj* **show** *?case* **by** (*auto simp*: *sym*[*OF obj-inv*])
next
case *T-Call* **from** *this*(*2*) **show** *?case* **by** *auto*
next
case *T-Upd* **from** *this*(*4*) **show** *?case* **by** *auto*
qed

5.0.6 Progress

Final Type Soundness Lemma

theorem *progress*:
assumes *Env Map.empty* ⊢ *t* : *A* **and** ¬(∃ *c A*. *t* = *Obj c A*)
shows ∃ *b*. *t* →$_\beta$ *b*
proof −
fix *f*
have
(∀ *A*. *Env Map.empty* ⊢ *t* : *A* ⟶ ¬(∃ *c T*. *t* = *Obj c T*) ⟶ (∃ *b*. *t* →$_\beta$ *b*))
&(∀ *A*. *Env Map.empty* ⊢ *Obj f A* : *A* ⟶ ¬(∃ *c T*. *Obj f A* = *Obj c T*)
⟶ (∃ *b*. *Obj f A* →$_\beta$ *b*))
proof (*induct rule*: *sterm-induct*)
case (*Bvar b*) **with** *no-bvar*[*of Env Map.empty b*] **show** *?case*
by *auto*
next
case (*Fvar x*) **with** *Fvar-beta*[*of x*] **show** *?case*
by *auto*
next
case *Obj* **show** *?case* **by** *auto*
next
case *empty* **thus** *?case* **by** *auto*
next
case *insert* **show** *?case* **by** *auto*
next
case (*Call t1 l t2*) **show** *?case*
proof (*clarify*)
fix *T* **assume**
Env Map.empty ⊢ *t1* : *T* **and** *Env Map.empty* ⊢ *t2* : *param*(*the*(*T*^*l*)) **and**
l ∈ *do T*
note *lc* = *typing-regular''*[*OF this*(*1*)] *typing-regular''*[*OF this*(*2*)]
from
‹*Env Map.empty* ⊢ *t1* : *T*›

```
      ‹∀ A. Env Map.empty ⊢ t1 : A ⟶ ¬ (∃ c T. t1 = Obj c T) ⟶ (∃ b. t1
→β b)›
    have (∃ c B. t1 = Obj c B) ∨ (∃ b. t1 →β b) by auto
    thus ∃ b. Call t1 l t2 →β b
    proof (elim disjE exE)
      fix c B assume t1 = Obj c B
      with
        ‹Env Map.empty ⊢ t1 : T› obj-inv[of Env Map.empty c B T]
        ‹l ∈ do T› obj-inv-elim[of Env Map.empty c B]
      have l ∈ dom c by auto
      with ‹t1 = Obj c B› lc beta.beta[of l c B t2]
      show ?thesis by auto
    next
      fix b assume t1 →β b
      from beta.beta-CallL[OF this lc(2)] show ?thesis by auto
    qed
  qed
 next
  case (Upd t1 l t2) show ?case
  proof (clarify)
    fix T F
    assume
      finite F and
      ∀ s p. s ∉ F ∧ p ∉ F ∧ s ≠ p
        ⟶ Env Map.empty(⦇s:T⦈)(⦇p:param(the(T^l))⦈)
            ⊢ (t2[Fvar s,Fvar p]) : return(the(T^l)) and
      Env Map.empty ⊢ t1 : T and
      l ∈ do T
    from typing-regular''[OF T-Upd[OF this]] lc-upd[of t1 l t2]
    obtain lc t1 and body t2 by auto
    from
      ‹Env Map.empty ⊢ t1 : T›
      ‹∀ A. Env Map.empty ⊢ t1 : A ⟶ ¬ (∃ c T. t1 = Obj c T) ⟶ (∃ b. t1
→β b)›
    have (∃ c B. t1 = Obj c B) ∨ (∃ b. t1 →β b) by auto
    thus ∃ b. Upd t1 l t2 →β b
    proof (elim disjE exE)
      fix c B assume t1 = Obj c B
      with
        ‹Env Map.empty ⊢ t1 : T› obj-inv[of Env Map.empty c B T]
        ‹l ∈ do T› obj-inv-elim[of Env Map.empty c B]
      have l ∈ dom c by auto
      with ‹t1 = Obj c B› ‹lc t1› ‹body t2› beta.beta-Upd[of l c B t2]
      show ?thesis by auto
    next
      fix b assume t1 →β b
      from beta.beta-UpdL[OF this ‹body t2›] show ?thesis by auto
    qed
  qed
```

qed
with *assms* **show** *?thesis* **by** *auto*
qed

end

6 Locally Nameless Sigma Calculus

theory *Locally-Nameless-Sigma*
imports *Sigma/ParRed Sigma/TypedSigma*
begin

end

References

[1] M. Abadi and L. Cardelli. "A Theory of Objects". Springer, New York, 1996.

[2] B. Aydemir, A, Charguéraud, B. C. Pierce, R. Pollack, and S. Weirich. Engineering formal metatheory. *Princ. of Programming Languages, POPL'08*, ACM, 2008.

[3] L. Henrio and F. Kammüller. A mechanized model of the theory of objects. *Formal Methods for Open Object-Based Distributed Systems,*. LNCS **4468** Springer, 2007.

[4] Tobias Nipkow. More Church Rosser Proofs. *Journal of Automated Reasoning.* **26**:51–66, 2001.