

# Certification of Absence of Dangling Pointers

By Javier de Dios and Ricardo Peña

July, 2010

## Contents

<b>1</b>	<b>Type Environment and operators</b>	<b>2</b>
1.1	Operators on type environments . . . . .	2
<b>2</b>	<b>Normal form values and heap</b>	<b>20</b>
<b>3</b>	<b>Useful functions and theorems from the Haskell Library or Prelude</b>	<b>23</b>
<b>4</b>	<b>Normalized Safe Expressions</b>	<b>28</b>
4.1	Free Variables . . . . .	30
<b>5</b>	<b>Primitive operators for SVM anf JVM</b>	<b>36</b>
<b>6</b>	<b>State of the SVM</b>	<b>36</b>
6.1	Sizes Table . . . . .	36
6.2	Stack . . . . .	36
6.3	Code Store and SafeImp program . . . . .	37
6.4	Runtime State . . . . .	38
<b>7</b>	<b>Resource-Aware Operational semantics of Safe expressions</b>	<b>39</b>
<b>8</b>	<b>Depth-Aware Operational semantics of Safe expressions</b>	<b>42</b>
<b>9</b>	<b>Closure Heap</b>	<b>50</b>
<b>10</b>	<b>SafeDAssBasic</b>	<b>60</b>
<b>11</b>	<b>Derived Assertions. P2. <math>\text{dom } \Gamma \subseteq \text{dom } E</math></b>	<b>65</b>
<b>12</b>	<b>Derived Assertions. P3. <math>L \text{ dom } G</math></b>	<b>68</b>
<b>13</b>	<b>Derived Assertions. P1. Semantic</b>	<b>76</b>
<b>14</b>	<b>Region Definitions</b>	<b>82</b>

15 Basic Facts	90
16 Derived Assertions. P5. $\text{shareRec } L \ \Gamma \ E \ h$ . P6. $\neg$ identity-Closure	92
17 Derived Assertions. P4. $\text{fv } e \subseteq L$	137
18 Derived Assertions. P7. $S \ L, \ \Gamma, E, h \cap R \ L, \ \Gamma, E, h =$	140
19 Derived Assertions. P8. $\text{closed } E \ L \ h$	198
20 Derived Assertions. P9. $\text{closed } v \ h'$	213
21 Derived Assertions	218
22 Proof rules for explicit deallocation	243
23 Region deallocation	254
24 Proof rules for region deallocation	319

## 1 Type Environment and operators

Non-functional algebraic types may be safe types  $s''$ , condemned types  $d''$  or indanger types  $r''$ .

**datatype**  $Mark = s'' \mid d'' \mid r''$

A type environment is a partial mapping from program variables to marks.

**types**  $TypeEnvironment = string \rightarrow Mark$

Some auxiliary predicates: 'unsafe' is true for condemned and in-danger types, 'safe' is true for safe types.

**constdefs**  $unsafe :: Mark \ option \Rightarrow bool$   
 $unsafe \ x \equiv (x = Some \ d'') \ \vee \ (x = Some \ r'')$

**constdefs**  $safe :: Mark \ option \Rightarrow bool$   
 $safe \ x \equiv (x = Some \ s'')$

### 1.1 Operators on type environments

**constdefs**

$unionEnv :: TypeEnvironment \Rightarrow TypeEnvironment \Rightarrow TypeEnvironment$   
 $unionEnv \ \Gamma 1 \ \Gamma 2 \equiv (\%x. \text{if } x: \text{dom } \Gamma 1 \text{ then } \Gamma 1 \ x \text{ else } \Gamma 2 \ x)$

Operator  $+$ : Disjoint union of environments

**constdefs**  $def\text{-}disjointUnionEnv :: TypeEnvironment \Rightarrow TypeEnvironment \Rightarrow bool$

$def-disjointUnionEnv \Gamma 1 \Gamma 2 \equiv dom \Gamma 1 \cap dom \Gamma 2 = \{\}$

**constdefs**  $disjointUnionEnv :: TypeEnvironment \Rightarrow TypeEnvironment \Rightarrow TypeEnvironment$  (**infix** + 100)  
 $\Gamma 1 + \Gamma 2 \equiv unionEnv \Gamma 1 \Gamma 2$

Operator  $\otimes$  : It allows to join two non-disjoint environments provided they map the common variables to the same types.

**constdefs**  
 $def-nonDisjointUnionEnv :: TypeEnvironment \Rightarrow TypeEnvironment \Rightarrow bool$   
 $def-nonDisjointUnionEnv \Gamma 1 \Gamma 2 \equiv (\forall x \in dom \Gamma 1 \cap dom \Gamma 2. \Gamma 1 x = \Gamma 2 x)$

**constdefs**  
 $nonDisjointUnionEnv :: TypeEnvironment \Rightarrow TypeEnvironment \Rightarrow TypeEnvironment$  (**infixl**  $\otimes$  60)  
 $\Gamma 1 \otimes \Gamma 2 \equiv unionEnv \Gamma 1 \Gamma 2$

Operator  $\otimes$  : It allows to join n-non-disjoint environments provided they map the common variables to the same types.

**fun**  $nonDisjointUnionEnvList :: TypeEnvironment list \Rightarrow TypeEnvironment$  **where**  
 $nonDisjointUnionEnvList \Gamma s = foldl nonDisjointUnionEnv empty \Gamma s$

**fun**  $def-nonDisjointUnionEnvList :: TypeEnvironment list \Rightarrow bool$  **where**  
 $def-nonDisjointUnionEnvList [] = True$

$| def-nonDisjointUnionEnvList (G \# Gs) = (let Gs' = nonDisjointUnionEnvList Gs;$

$def-Gs' = def-nonDisjointUnionEnvList Gs$  in  
 $def-nonDisjointUnionEnv G Gs' \wedge def-Gs')$

Operator  $\oplus$  : It allows to join two non-disjoint environments provided they map the common variables to the same safe types.

**constdefs**  
 $def-nonDisjointUnionSafeEnv :: TypeEnvironment \Rightarrow TypeEnvironment \Rightarrow bool$   
 $def-nonDisjointUnionSafeEnv \Gamma 1 \Gamma 2 \equiv (\forall x \in dom \Gamma 1 \cap dom \Gamma 2. safe (\Gamma 1 x) \wedge safe (\Gamma 2 x))$

**constdefs**  
 $nonDisjointUnionSafeEnv :: TypeEnvironment \Rightarrow TypeEnvironment \Rightarrow TypeEnvironment$  (**infixl**  $\oplus$  60)  
 $\Gamma 1 \oplus \Gamma 2 \equiv unionEnv \Gamma 1 \Gamma 2$

**fun**  $nonDisjointUnionSafeEnvList :: TypeEnvironment list \Rightarrow TypeEnvironment$  **where**  
 $nonDisjointUnionSafeEnvList \Gamma s = foldl nonDisjointUnionSafeEnv empty \Gamma s$

**fun**  $def-nonDisjointUnionSafeEnvList :: TypeEnvironment list \Rightarrow bool$  **where**  
 $def-nonDisjointUnionSafeEnvList [] = True$

| *def-nonDisjointUnionSafeEnvList* ( $G \# Gs$ ) = (*let*  $Gs' = \text{nonDisjointUnionSafeEnvList } Gs$ ;  
 $Gs$  in  
 $\text{def-nonDisjointUnionSafeEnv } G \text{ } Gs' \wedge \text{def-}Gs'$ )

Operator  $\triangleright$  : This is used in rules for *let*. In the union  $\Gamma 1 \triangleright \Gamma 2$  not allowed that variables in  $L$  may have unsafe type in  $\Gamma 1$ . Also, variables common to  $\Gamma 2$  and  $L$  may not have unsafe type in  $\Gamma 2$ . Once well defined, in environment  $\Gamma 1 \triangleright \Gamma 2$  the types assigned in  $\Gamma 2$  to common variables, will prevail.

#### constdefs

*def-pp* :: *TypeEnvironment*  $\Rightarrow$  *TypeEnvironment*  $\Rightarrow$  *string set*  $\Rightarrow$  *bool*  
*def-pp*  $\Gamma 1 \Gamma 2 L \equiv$   
 $(\forall x. x \in \text{dom } \Gamma 1 \longrightarrow \text{unsafe } (\Gamma 1 x) \longrightarrow x \notin L \wedge (x \notin \text{dom } \Gamma 2 \vee (\Gamma 2 x \neq \text{Some } s'' \wedge \Gamma 2 x \neq \text{Some } d'')))) \wedge$   
 $(\forall x. x \in \text{dom } \Gamma 2 \wedge$   
 $(\Gamma 2 x = \text{Some } s'' \vee \Gamma 2 x = \text{Some } d''))$   
 $\longrightarrow x \in L)$

#### constdefs

*pp* :: *TypeEnvironment*  $\Rightarrow$  *TypeEnvironment*  $\Rightarrow$  *string set*  $\Rightarrow$  *TypeEnvironment*  
 $(\rightarrow -)$   
 $\Gamma 1 \triangleright \Gamma 2 L \equiv (\%x. (\text{if } x \notin \text{dom } \Gamma 1 \vee (x \in \text{dom } \Gamma 1 \cap \text{dom } \Gamma 2 \wedge \text{safe } (\Gamma 1 x))$   
 $\text{then } \Gamma 2 x \text{ else } \Gamma 1 x))$

Lemmas for *unionEnv* operator

**lemma** *empty-unionEnv*: *unionEnv*  $m$  *empty* =  $m$   
**apply** (*rule ext*)  
**by** (*simp add: unionEnv-def split: option.split add: dom-def*)

**lemma** *dom-empty-unionEnv*: *dom* (*unionEnv*  $m1$  *empty*) = *dom*  $m1$   
**apply** (*subgoal-tac unionEnv m1 empty = m1*)  
**by** (*simp, rule empty-unionEnv*)

Lemmas for *disjointUnionEnv* operator

**lemma** *empty-disjointUnionEnv*: *disjointUnionEnv*  $m$  *empty* =  $m$   
**by** (*simp add: disjointUnionEnv-def, rule empty-unionEnv*)

**lemma** *dom-empty-disjointUnionEnv*: *dom* (*disjointUnionEnv*  $m1$  *empty*) = *dom*  $m1$   
**by** (*simp add: disjointUnionEnv-def, rule dom-empty-unionEnv*)

#### lemma union-dom-disjointUnionEnv:

*def-disjointUnionEnv*  $\Gamma 1 \Gamma 2 \implies \text{dom } (\text{disjointUnionEnv } \Gamma 1 \Gamma 2) = \text{dom } \Gamma 1 \cup \text{dom } \Gamma 2$

**apply** (*simp add: def-disjointUnionEnv-def add: disjointUnionEnv-def add: unionEnv-def, auto*)

**by** (*split split-if-asm, simp, simp*)

**lemma** *dom-disjointUnionEnv-monotone*:

$$\text{dom } (G + G') = \text{dom } G \cup \text{dom } G'$$

**apply** (*simp add: disjointUnionEnv-def add: unionEnv-def, auto*)

**by** (*split split-if-asm, simp, simp*)

Lemmas for nonDisjointUnionEnv operator

**lemma** *empty-nonDisjointUnionEnv* :

$$\text{empty} \otimes A = A$$

**by** (*simp add: nonDisjointUnionEnv-def add: unionEnv-def*)

**lemma** *refl-nonDisjointUnionEnv* :

$$A \otimes A = A$$

**by** (*simp add: nonDisjointUnionEnv-def add: unionEnv-def*)

**lemma** *list-induct3*:

$$[] \ P \ [] \ 0;$$

$$!!x \ xs. \ P \ (x \# \ xs) \ 0;$$

$$!!i. \ P \ [] \ (\text{Suc } i);$$

$$!!x \ xs \ i. \ P \ xs \ i \implies P \ (x \# \ xs) \ (\text{Suc } i) \ []$$

$$\implies P \ xs \ i$$

**by** (*induct xs arbitrary: i*) (*case-tac x, auto*)<sup>+</sup>

**lemma** *nonDisjointUnionEnv-commutative*:

$$\text{def-nonDisjointUnionEnv } G \ G' \implies (G \otimes G') = (G' \otimes G)$$

**apply** (*simp add: nonDisjointUnionEnv-def add: unionEnv-def*)

**apply** (*rule ext*)

**by** (*simp add: def-nonDisjointUnionEnv-def, clarsimp*)

**lemma** *nonDisjointUnionEnv-assoc*:

$$(G1 \otimes G2) \otimes G3 = G1 \otimes (G2 \otimes G3)$$

**apply** (*simp add: nonDisjointUnionEnv-def add: unionEnv-def*)

**apply** (*rule ext, auto*)

**apply** (*split split-if-asm, simp, simp*)

**apply** (*split split-if-asm, simp, simp*)

**by** (*split split-if-asm, simp, simp add: dom-def*)

**lemma** *nonDisjointUnionEnv-disjointUnionEnv-assoc*:

$$(G1 \otimes G2) + G3 = G1 \otimes (G2 + G3)$$

**apply** (*simp add: nonDisjointUnionEnv-def add: disjointUnionEnv-def add: unionEnv-def*)

**apply** (*rule ext, auto*)

**apply** (*split split-if-asm, simp, simp*)

**apply** (*split split-if-asm, simp, simp*)

**by** (*split split-if-asm, simp, simp add: dom-def*)

**lemma** *foldl-prop1*:

$$\text{foldl } op \otimes (G' \otimes G) \ Gs = G' \otimes \text{foldl } op \otimes G \ Gs$$

**apply** (*induct Gs arbitrary: G*)  
**apply** *simp*  
**by** (*simp-all add: nonDisjointUnionEnv-assoc*)

**lemma** *foldl-prop2*:  
 $\text{foldl } op \otimes (G' \otimes G) \ Gs + G'' = G' \otimes \text{foldl } op \otimes G \ Gs + G''$   
**apply** (*induct Gs arbitrary: G*)  
**apply** (*simp-all add: nonDisjointUnionEnv-assoc*)  
**by** (*rule nonDisjointUnionEnv-disjointUnionEnv-assoc*)

**lemma** *union-dom-nonDisjointUnionEnv*:  
 $\text{dom } (A \otimes B) = \text{dom } A \cup \text{dom } B$   
**apply** (*simp add: nonDisjointUnionEnv-def add: unionEnv-def, auto*)  
**by** (*split split-if-asm, simp-all*)

**lemma** *union-dom-nonDisjointUnionEnv-disjointUnionEnv*:  
 $\text{dom } (A \otimes B + [x \mapsto m]) = \text{dom } A \cup \text{dom } (B + [x \mapsto m])$   
**apply** (*simp add: nonDisjointUnionEnv-def add: unionEnv-def, auto*)  
**apply** (*split split-if-asm, simp*)  
**by** *simp*

**lemma** *def-nonDisjointUnionEnvList-prop1*:  
 $\text{def-nonDisjointUnionEnvList } (x \# xs) \longrightarrow \text{def-nonDisjointUnionEnv } x \ (\text{foldl } op \otimes \text{empty } xs)$   
**by** (*simp add: Let-def*)

**lemma** *dom-foldl-monotone*:  
 $\text{dom } (\text{foldl } op \otimes (\text{empty} \otimes \text{snd } a) \ (\text{map } \text{snd } \text{assert})) =$   
 $\text{dom } (\text{snd } a) \cup \text{dom } (\text{foldl } op \otimes \text{empty} \ (\text{map } \text{snd } \text{assert}))$   
**apply** (*subgoal-tac empty  $\otimes$  snd a = snd a  $\otimes$  empty, simp*)  
**apply** (*subgoal-tac foldl op  $\otimes$  (snd a  $\otimes$  empty) (map snd assert) =*  
 $(\text{snd } a) \otimes \text{foldl } op \otimes \text{empty} \ (\text{map } \text{snd } \text{assert}), \text{simp}$   
**apply** (*rule union-dom-nonDisjointUnionEnv*)  
**apply** (*rule foldl-prop1*)  
**apply** (*subgoal-tac def-nonDisjointUnionEnv empty (snd a)*)  
**apply** (*erule nonDisjointUnionEnv-commutative*)  
**by** (*simp add: def-nonDisjointUnionEnv-def*)

**lemma** *dom-foldl-disjointUnionEnv-monotone*:  
 $\text{dom } (\text{foldl } op \otimes (\text{empty} \otimes \text{snd } a) \ (\text{map } \text{snd } \text{assert}) + [x \mapsto d']) =$   
 $\text{dom } (\text{snd } a) \cup \text{dom } (\text{foldl } op \otimes \text{empty} \ (\text{map } \text{snd } \text{assert})) \cup \text{dom } [x \mapsto d']$   
**apply** (*subgoal-tac empty  $\otimes$  snd a = snd a  $\otimes$  empty, simp*)  
**apply** (*subgoal-tac foldl op  $\otimes$  (snd a  $\otimes$  empty) (map snd assert) =*  
 $(\text{snd } a) \otimes \text{foldl } op \otimes \text{empty} \ (\text{map } \text{snd } \text{assert}), \text{simp}$   
**apply** (*subst dom-disjointUnionEnv-monotone*)  
**apply** (*subst union-dom-nonDisjointUnionEnv*)  
**apply** *simp*  
**apply** (*rule foldl-prop1*)  
**apply** (*subgoal-tac def-nonDisjointUnionEnv empty (snd a)*)

**apply** (*erule nonDisjointUnionEnv-commutative*)  
**by** (*simp add: def-nonDisjointUnionEnv-def*)

**lemma** *dom-monotone*:

*length assert > i  $\implies$  dom (snd (assert ! i))  $\subseteq$  dom (foldl op  $\otimes$  empty (map snd assert))*

**apply** (*induct assert i rule: list-induct3, simp-all*)

**apply** (*subgoal-tac dom (foldl op  $\otimes$  (empty  $\otimes$  snd x) (map snd xs)) =  
dom (snd x)  $\cup$  dom (foldl op  $\otimes$  empty (map snd xs)),simp)*)

**apply** *blast*

**apply** (*rule dom-foldl-monotone*)

**apply** (*subgoal-tac dom (foldl op  $\otimes$  (empty  $\otimes$  snd x) (map snd xs)) =  
dom (snd x)  $\cup$  dom (foldl op  $\otimes$  empty (map snd xs)),simp)*)

**apply** *blast*

**by** (*rule dom-foldl-monotone*)

**lemma** *dom-monotone-foldl-nonDisjointUnionEnv*:

*length assert > i  $\implies$  dom (snd (assert ! i))  $\subseteq$  dom (foldl op  $\otimes$  empty (map snd assert) + [x  $\mapsto$  d'])*

**apply** (*induct assert i rule: list-induct3, simp-all*)

**apply** (*subgoal-tac dom (foldl op  $\otimes$  (empty  $\otimes$  snd xa) (map snd xs) + [x  $\mapsto$  d'])  
=*

*dom (snd xa)  $\cup$  dom (foldl op  $\otimes$  empty (map snd xs))  $\cup$  dom [x  
 $\mapsto$  d'],simp)*

**apply** *blast*

**apply** (*rule dom-foldl-disjointUnionEnv-monotone*)

**apply** (*subgoal-tac dom (foldl op  $\otimes$  (empty  $\otimes$  snd xa) (map snd xs) + [x  $\mapsto$  d'])  
=*

*dom (snd xa)  $\cup$  dom (foldl op  $\otimes$  empty (map snd xs))  $\cup$  dom [x  
 $\mapsto$  d'],simp)*

**apply** (*subgoal-tac dom (foldl op  $\otimes$  empty (map snd xs) + [x  $\mapsto$  d']) =  
dom (foldl op  $\otimes$  empty (map snd xs))  $\cup$  dom [x  $\mapsto$  d'],simp)*

**apply** *blast*

**apply** (*rule dom-disjointUnionEnv-monotone*)

**by** (*rule dom-foldl-disjointUnionEnv-monotone*)

**lemma** *dom- $\Gamma$ i-subseteq-dom- $\Gamma$ -case*:

*length assert > 0  $\implies$*

*( $\forall i < \text{length assert. dom (snd (assert ! i)) \subseteq$   
dom (foldl op  $\otimes$  empty (map snd assert)))*

**apply** (*induct assert*)

**apply** *simp*

**apply** (*case-tac assert = []*)

**apply** (*simp add: nonDisjointUnionEnv-def add: unionEnv-def,simp*)

**apply** (*rule allI*)

**apply** (*subgoal-tac dom (foldl op  $\otimes$  (empty  $\otimes$  snd a) (map snd assert)) =*

```

      dom (snd a)  $\cup$  dom (foldl op  $\otimes$  empty (map snd assert)),simp)
  apply (case-tac i)
  apply (simp,blast)
  apply simp
  apply (erule-tac x=nat in allE)
  apply (blast,simp)
  apply (subgoal-tac empty  $\otimes$  snd a = snd a  $\otimes$  empty,simp)
  apply (subgoal-tac foldl op  $\otimes$  (snd a  $\otimes$  empty) (map snd assert) =
      (snd a)  $\otimes$  foldl op  $\otimes$  empty (map snd assert),simp)
  apply (rule union-dom-nonDisjointUnionEnv)
  apply (rule foldl-prop1)
  apply (subgoal-tac def-nonDisjointUnionEnv empty (snd a))
  apply (erule nonDisjointUnionEnv-commutative)
  by (simp add: def-nonDisjointUnionEnv-def)

```

**lemma** dom- $\Gamma$ i-subseteq-dom- $\Gamma$ -cased:

```

  length assert > 0  $\implies$ 
    ( $\forall i < \text{length assert. dom (snd (assert ! i))} \subseteq \text{dom (foldl op} \otimes \text{empty (map snd}$ 
  assert) +  $[x \mapsto d']]$ )
  apply (induct assert)
  apply simp
  apply (case-tac assert = [])
  apply (simp add: nonDisjointUnionEnv-def add: unionEnv-def)
  apply (simp add: disjointUnionEnv-def add: unionEnv-def, clarsimp)
  apply (rule allI)
  apply (subgoal-tac dom (foldl op  $\otimes$  (empty  $\otimes$  snd a) (map snd assert) +  $[x \mapsto$ 
  d''] =
      dom (snd a)  $\cup$  dom (foldl op  $\otimes$  empty (map snd assert) +  $[x \mapsto$ 
  d'']),simp)
  apply (case-tac i)
  apply (simp,blast)
  apply simp
  apply (erule-tac x=nat in allE)
  apply (blast,simp)
  apply (subgoal-tac empty  $\otimes$  snd a = snd a  $\otimes$  empty,simp)
  apply (subgoal-tac foldl op  $\otimes$  (snd a  $\otimes$  empty) (map snd assert) +  $[x \mapsto d''] =$ 
      (snd a)  $\otimes$  foldl op  $\otimes$  empty (map snd assert) +  $[x \mapsto d'']$ ,simp)
  apply (rule union-dom-nonDisjointUnionEnv-disjointUnionEnv)
  apply (rule foldl-prop2)
  apply (subgoal-tac def-nonDisjointUnionEnv empty (snd a))
  apply (erule nonDisjointUnionEnv-commutative)
  by (simp add: def-nonDisjointUnionEnv-def)

```

**lemma** nonDisjointUnionEnv-prop1:

```

  ( $G \otimes G'$ ) z = Some m  $\implies G$  z = Some m  $\vee (z \notin \text{dom } G \wedge G'$  z = Some m)
  apply (simp add: nonDisjointUnionEnv-def add: unionEnv-def)
  by (split split-if-asm,simp-all)

```



**lemma** *nonDisjointUnionEnv-prop2*:  
 $\llbracket \text{def-nonDisjointUnionEnv } G \ G'; (G \otimes G') \ z = \text{Some } m \rrbracket$   
 $\implies G' \ z = \text{Some } m \vee (z \notin \text{dom } G' \wedge G \ z = \text{Some } m)$   
**apply** (*frule nonDisjointUnionEnv-commutative,simp*)  
**by** (*erule nonDisjointUnionEnv-prop1*)

**lemma** *nonDisjointUnionEnv-prop5*:  
 $\llbracket x \in \text{dom } G; G \ x \neq \text{Some } m \rrbracket \implies (G \otimes G') \ x \neq \text{Some } m$   
**by** (*simp add: nonDisjointUnionEnv-def add: unionEnv-def*)

**lemma** *nonDisjointUnionEnv-prop6*:  
 $\llbracket \text{def-nonDisjointUnionEnv } G \ G'; z \in \text{dom } G'; G' \ z \neq \text{Some } m \rrbracket \implies (G \otimes G') \ z \neq \text{Some } m$   
**apply** (*simp add: def-nonDisjointUnionEnv-def*)  
**apply** (*erule-tac x=z in ballE*)  
**apply** (*simp add: nonDisjointUnionEnv-def add: unionEnv-def*)  
**by** (*simp add: nonDisjointUnionEnv-def add: unionEnv-def*)

**lemma** *nonDisjointUnionEnv-prop6-1*:  
 $\llbracket x \in \text{dom } G; G \ x = \text{Some } m \rrbracket \implies (G \otimes G') \ x = \text{Some } m$   
**by** (*simp add: nonDisjointUnionEnv-def add: unionEnv-def*)

**lemma** *nonDisjointUnionEnv-prop6-2*:  
 $\llbracket \text{def-nonDisjointUnionEnv } G \ G'; G' \ z = \text{Some } m \rrbracket \implies (G \otimes G') \ z = \text{Some } m$   
**apply** (*simp add: def-nonDisjointUnionEnv-def*)  
**apply** (*erule-tac x=z in ballE*)  
**apply** (*simp add: nonDisjointUnionEnv-def add: unionEnv-def*)  
**apply** (*simp add: nonDisjointUnionEnv-def add: unionEnv-def*)  
**by** (*rule impI, simp, simp add: dom-def*)

**lemma** *Otimes-prop1*:  
 $i < \text{length } (\text{map } \text{snd } \text{assert}) \longrightarrow$   
 $\text{length } (\text{map } \text{snd } \text{assert}) > 0 \longrightarrow$   
 $\text{def-nonDisjointUnionEnvList } (\text{map } \text{snd } \text{assert}) \longrightarrow$   
 $\text{foldl } \text{op } \otimes \ \text{empty } (\text{map } \text{snd } \text{assert}) \ z = \text{Some } d'' \longrightarrow$   
 $z \in \text{dom } (\text{snd } (\text{assert } ! \ i)) \longrightarrow$   
 $(\text{snd } (\text{assert } ! \ i)) \ z = \text{Some } d''$   
**apply** (*induct assert i rule: list-induct3, simp-all*)  
**apply** (*intro impI*)  
**apply** (*subgoal-tac def-nonDisjointUnionEnv empty (snd x)*)  
**prefer** 2 **apply** (*simp add: def-nonDisjointUnionEnv-def*)  
**apply** (*subgoal-tac (empty  $\otimes$  (snd x)) = ((snd x)  $\otimes$  empty),simp*)  
**prefer** 2 **apply** (*rule nonDisjointUnionEnv-commutative,simp*)  
**apply** (*subgoal-tac foldl op  $\otimes$  (snd x  $\otimes$  empty) (map snd xs) =*  
 $(\text{snd } x) \otimes \text{foldl op } \otimes \ \text{empty } (\text{map } \text{snd } \text{xs}),\text{simp}$ )  
**prefer** 2 **apply** (*rule foldl-prop1*)

```

apply (frule nonDisjointUnionEnv-prop1)
apply (erule disjE)
  apply (simp add: dom-def)
apply (simp,intro impI,simp)
apply (case-tac xs=[],simp)
apply (simp add: Let-def)
apply (elim conjE)
apply (subgoal-tac def-nonDisjointUnionEnv empty (snd x))
  prefer 2 apply (simp add: def-nonDisjointUnionEnv-def)
apply (subgoal-tac (empty  $\otimes$  snd x) = (snd x  $\otimes$  empty),simp)
  prefer 2 apply (rule nonDisjointUnionEnv-commutative,simp)
apply (subgoal-tac foldl op  $\otimes$  (snd x  $\otimes$  empty) (map snd xs) =
      snd x  $\otimes$  foldl op  $\otimes$  empty (map snd xs),simp)
  prefer 2 apply (rule foldl-prop1)
apply (subgoal-tac def-nonDisjointUnionEnvList (snd x # (map snd xs))  $\longrightarrow$ 
      def-nonDisjointUnionEnv (snd x) (foldl op  $\otimes$  empty (map snd
xs)),simp)
  prefer 2 apply (rule def-nonDisjointUnionEnvList-prop1)
apply (frule-tac G=snd x and G'=foldl op  $\otimes$  empty (map snd xs) in nonDisjointUnionEnv-prop2,simp)
apply (erule disjE)
  apply simp
apply (elim conjE)
apply (subgoal-tac dom (snd (xs ! i))  $\subseteq$  dom (foldl op  $\otimes$  empty (map snd xs)))
  prefer 2 apply (rule dom-monotone,blast)
by blast

```

**lemma** *Otimes-prop2* [*rule-format*]:

```

i < length (map snd assert)  $\longrightarrow$ 
length (map snd assert) > 0  $\longrightarrow$ 
def-nonDisjointUnionEnvList (map snd assert)  $\longrightarrow$ 
x  $\in$  dom (snd (assert ! i))  $\longrightarrow$ 
snd (assert ! i) x  $\neq$  Some s''  $\longrightarrow$ 
foldl op  $\otimes$  empty (map snd assert) x  $\neq$  Some s''
apply (induct assert i rule: list-induct3, simp-all)
apply (intro impI)
apply (simp add: Let-def)
apply (elim conjE)
apply (subgoal-tac def-nonDisjointUnionEnv empty (snd xa))
  prefer 2 apply (simp add: def-nonDisjointUnionEnv-def)
apply (subgoal-tac (empty  $\otimes$  (snd xa)) = ((snd xa)  $\otimes$  empty),simp)
  prefer 2 apply (rule nonDisjointUnionEnv-commutative,simp)
apply (subgoal-tac foldl op  $\otimes$  (snd xa  $\otimes$  empty) (map snd xs) =
      (snd xa)  $\otimes$  foldl op  $\otimes$  empty (map snd xs),simp)
  prefer 2 apply (rule foldl-prop1)
apply (rule nonDisjointUnionEnv-prop5,assumption+)
apply (intro impI,simp)
apply (case-tac xs=[],simp)
apply (simp add: Let-def)
apply (elim conjE)

```

```

apply (subgoal-tac def-nonDisjointUnionEnv empty (snd xa))
  prefer 2 apply (simp add: def-nonDisjointUnionEnv-def)
apply (subgoal-tac (empty  $\otimes$  snd xa) = (snd xa  $\otimes$  empty),simp)
  prefer 2 apply (rule nonDisjointUnionEnv-commutative,simp)
apply (subgoal-tac foldl op  $\otimes$  (snd xa  $\otimes$  empty) (map snd xs) =
      snd xa  $\otimes$  foldl op  $\otimes$  empty (map snd xs),simp)
  prefer 2 apply (rule foldl-prop1)
apply (subgoal-tac def-nonDisjointUnionEnvList (snd xa # (map snd xs))  $\longrightarrow$ 
      def-nonDisjointUnionEnv (snd xa) (foldl op  $\otimes$  empty (map snd
xs)),simp)
  prefer 2 apply (rule def-nonDisjointUnionEnvList-prop1)
apply (subgoal-tac dom (snd (xs ! i))  $\subseteq$  dom (foldl op  $\otimes$  empty (map snd xs)))
  prefer 2 apply (rule dom-monotone,simp)
apply (subgoal-tac  $x \in$  dom (foldl op  $\otimes$  empty (map snd xs)))
  prefer 2 apply blast
by (rule nonDisjointUnionEnv-prop6,assumption+)

```

**lemma** *Otimes-prop3*:

```

   $i < \text{length} (\text{map snd assert}) \longrightarrow$ 
   $\text{length} (\text{map snd assert}) > 0 \longrightarrow$ 
   $\text{def-nonDisjointUnionEnvList} (\text{map snd assert}) \longrightarrow$ 
   $\text{snd} (\text{assert} ! i) x = \text{Some } m \longrightarrow$ 
   $\text{foldl op} \otimes \text{empty} (\text{map snd assert}) x = \text{Some } m$ 
apply (induct assert i rule: list-induct3, simp-all)
apply (intro impI)
apply (simp add: Let-def)
apply (elim conjE)
apply (subgoal-tac def-nonDisjointUnionEnv empty (snd xa))
  prefer 2 apply (simp add: def-nonDisjointUnionEnv-def)
apply (subgoal-tac (empty  $\otimes$  (snd xa)) = ((snd xa)  $\otimes$  empty),simp)
  prefer 2 apply (rule nonDisjointUnionEnv-commutative,simp)
apply (subgoal-tac foldl op  $\otimes$  (snd xa  $\otimes$  empty) (map snd xs) =
      (snd xa)  $\otimes$  foldl op  $\otimes$  empty (map snd xs),simp)
  prefer 2 apply (rule foldl-prop1)
apply (rule nonDisjointUnionEnv-prop6-1, simp add: dom-def, assumption+)
apply (intro impI,simp)
apply (case-tac xs=[],simp)
apply (simp add: Let-def)
apply (elim conjE)
apply (subgoal-tac def-nonDisjointUnionEnv empty (snd xa))
  prefer 2 apply (simp add: def-nonDisjointUnionEnv-def)
apply (subgoal-tac (empty  $\otimes$  snd xa) = (snd xa  $\otimes$  empty),simp)
  prefer 2 apply (rule nonDisjointUnionEnv-commutative,simp)
apply (subgoal-tac foldl op  $\otimes$  (snd xa  $\otimes$  empty) (map snd xs) =
      snd xa  $\otimes$  foldl op  $\otimes$  empty (map snd xs),simp)
  prefer 2 apply (rule foldl-prop1)
apply (subgoal-tac def-nonDisjointUnionEnvList (snd xa # (map snd xs))  $\longrightarrow$ 
      def-nonDisjointUnionEnv (snd xa) (foldl op  $\otimes$  empty (map snd

```

```

xs)),simp)
prefer 2 apply (rule def-nonDisjointUnionEnvList-prop1)
apply (subgoal-tac dom (snd (xs ! i))  $\subseteq$  dom (foldl op  $\otimes$  empty (map snd xs)))
prefer 2 apply (rule dom-monotone,simp)
apply (subgoal-tac  $x \in$  dom (foldl op  $\otimes$  empty (map snd xs)))
prefer 2 apply blast
by (rule nonDisjointUnionEnv-prop6-2,assumption+)

```

**lemma** *nonDisjointUnionEnv-disjointUnionEnv-prop1*:  
 $\llbracket x \in \text{dom } G; G \neq \text{Some } m \rrbracket \implies ((G \otimes G') + G'') x \neq \text{Some } m$   
**apply** (simp add: nonDisjointUnionEnv-def add: disjointUnionEnv-def add: unionEnv-def)  
**apply** (rule impI, auto)  
**apply** (split split-if-asm, simp add: dom-def)  
**by** (simp add: dom-def)

**lemma** *nonDisjointUnionEnv-disjointUnionEnv-prop2*:  
 $\llbracket \text{def-nonDisjointUnionEnv } G \ G'; z \in \text{dom } G'; G' z \neq \text{Some } m \rrbracket \implies ((G \otimes G') + G'') z \neq \text{Some } m$   
**apply** (simp add: def-nonDisjointUnionEnv-def)  
**apply** (erule-tac  $x=z$  **in** ballE)  
**apply** (simp add: nonDisjointUnionEnv-def add: disjointUnionEnv-def add: unionEnv-def)  
  
**apply** (rule impI, auto)  
**apply** (simp add: nonDisjointUnionEnv-def add: disjointUnionEnv-def add: unionEnv-def)  
  
**by** (split split-if-asm, simp add: dom-def,auto)

**lemma** *nonDisjointUnionEnv-disjointUnionEnv-prop3*:  
 $\llbracket x \in \text{dom } G; G x = \text{Some } m \rrbracket \implies ((G \otimes G') + G'') x = \text{Some } m$   
**apply** (simp add: nonDisjointUnionEnv-def add: disjointUnionEnv-def add: unionEnv-def)  
**by** (rule impI,auto)

**lemma** *nonDisjointUnionEnv-disjointUnionEnv-prop4*:  
 $\llbracket \text{def-nonDisjointUnionEnv } G \ G'; G' z = \text{Some } m \rrbracket \implies ((G \otimes G') + G'') z = \text{Some } m$   
**apply** (simp add: def-nonDisjointUnionEnv-def)  
**apply** (erule-tac  $x=z$  **in** ballE)  
**apply** (simp add: nonDisjointUnionEnv-def add: disjointUnionEnv-def add: unionEnv-def)  
**apply** (rule impI,auto)  
**apply** (simp add: nonDisjointUnionEnv-def add: disjointUnionEnv-def add: unionEnv-def)  
**apply** (split split-if-asm,auto)  
**by** (split split-if-asm,auto)

**lemma** *Otimes-prop2-not-s* [rule-format]:  
 $i < \text{length } (\text{map } \text{snd } \text{assert}) \longrightarrow$   
 $\text{length } (\text{map } \text{snd } \text{assert}) > 0 \longrightarrow$

```

def-nonDisjointUnionEnvList (map snd assert) →
y ∈ dom (snd (assert ! i)) →
snd (assert ! i) y ≠ Some s'' →
(foldl op ⊗ empty (map snd assert) + [x ↦ d']) y ≠ Some s''
apply (induct assert i rule: list-induct3, simp-all)
apply (intro impI)
apply (simp add: Let-def)
apply (elim conjE)
apply (subgoal-tac def-nonDisjointUnionEnv empty (snd xa))
prefer 2 apply (simp add: def-nonDisjointUnionEnv-def)
apply (subgoal-tac (empty ⊗ (snd xa)) = ((snd xa) ⊗ empty), simp)
prefer 2 apply (rule nonDisjointUnionEnv-commutative, simp)
apply (subgoal-tac foldl op ⊗ (snd xa ⊗ empty) (map snd xs) =
(snd xa) ⊗ foldl op ⊗ empty (map snd xs), simp)
prefer 2 apply (rule foldl-prop1)
apply (rule nonDisjointUnionEnv-disjointUnionEnv-prop1, assumption+)
apply (intro impI, simp)
apply (case-tac xs=[], simp)
apply (simp add: Let-def)
apply (elim conjE)
apply (subgoal-tac def-nonDisjointUnionEnv empty (snd xa))
prefer 2 apply (simp add: def-nonDisjointUnionEnv-def)
apply (subgoal-tac (empty ⊗ snd xa) = (snd xa ⊗ empty), simp)
prefer 2 apply (rule nonDisjointUnionEnv-commutative, simp)
apply (subgoal-tac foldl op ⊗ (snd xa ⊗ empty) (map snd xs) =
snd xa ⊗ foldl op ⊗ empty (map snd xs), simp)
prefer 2 apply (rule foldl-prop1)
apply (subgoal-tac def-nonDisjointUnionEnvList (snd xa # (map snd xs)) →
def-nonDisjointUnionEnv (snd xa) (foldl op ⊗ empty (map snd
xs)), simp)
prefer 2 apply (rule def-nonDisjointUnionEnvList-prop1)
apply (subgoal-tac dom (snd (xs ! i)) ⊆ dom (foldl op ⊗ empty (map snd xs)))
prefer 2 apply (rule dom-monotone, simp)
apply (subgoal-tac y ∈ dom (foldl op ⊗ empty (map snd xs)))
prefer 2 apply blast
apply (rule nonDisjointUnionEnv-disjointUnionEnv-prop2, assumption+)
by (simp add: disjointUnionEnv-def add: unionEnv-def)

```

**lemma** *Otimes-prop4* [rule-format]:

```

i < length (map snd assert) →
length (map snd assert) > 0 →
def-nonDisjointUnionEnvList (map snd assert) →
snd (assert ! i) xa = Some m →
(foldl op ⊗ empty (map snd assert) + [x ↦ d']) xa = Some m
apply (induct assert i rule: list-induct3, simp-all)
apply (intro impI)
apply (simp add: Let-def)
apply (elim conjE)

```

```

apply (subgoal-tac def-nonDisjointUnionEnv empty (snd xb))
prefer 2 apply (simp add: def-nonDisjointUnionEnv-def)
apply (subgoal-tac (empty  $\otimes$  (snd xb)) = ((snd xb)  $\otimes$  empty),simp)
prefer 2 apply (rule nonDisjointUnionEnv-commutative,simp)
apply (subgoal-tac foldl op  $\otimes$  (snd xb  $\otimes$  empty) (map snd xs) =
      (snd xb)  $\otimes$  foldl op  $\otimes$  empty (map snd xs),simp)
prefer 2 apply (rule foldl-prop1)
apply (rule nonDisjointUnionEnv-disjointUnionEnv-prop3, simp add: dom-def,
assumption+)
apply (intro impI,simp)
apply (case-tac xs=[],simp)
apply (simp add: Let-def)
apply (elim conjE)
apply (subgoal-tac def-nonDisjointUnionEnv empty (snd xb))
prefer 2 apply (simp add: def-nonDisjointUnionEnv-def)
apply (subgoal-tac (empty  $\otimes$  snd xb) = (snd xb  $\otimes$  empty),simp)
prefer 2 apply (rule nonDisjointUnionEnv-commutative,simp)
apply (subgoal-tac foldl op  $\otimes$  (snd xb  $\otimes$  empty) (map snd xs) =
      snd xb  $\otimes$  foldl op  $\otimes$  empty (map snd xs),simp)
prefer 2 apply (rule foldl-prop1)
apply (subgoal-tac def-nonDisjointUnionEnvList (snd xb # (map snd xs))  $\longrightarrow$ 
      def-nonDisjointUnionEnv (snd xb) (foldl op  $\otimes$  empty (map snd
xs)),simp)
prefer 2 apply (rule def-nonDisjointUnionEnvList-prop1)
apply (subgoal-tac dom (snd (xs ! i))  $\subseteq$  dom (foldl op  $\otimes$  empty (map snd xs)))
prefer 2 apply (rule dom-monotone,simp)
apply (subgoal-tac xa  $\in$  dom (foldl op  $\otimes$  empty (map snd xs)))
prefer 2 apply blast
apply (rule nonDisjointUnionEnv-disjointUnionEnv-prop4,assumption)
by (simp add: disjointUnionEnv-def add: unionEnv-def)

```

Lemmas for nonDisjointUnionSafeEnv operator

```

lemma empty-nonDisjointUnionSafeEnv :
  empty  $\oplus$  A = A
by (simp add: nonDisjointUnionSafeEnv-def add: unionEnv-def)

```

Lemmas for triangle operator

```

lemma safe-triangle:
   $\llbracket (\text{def-pp } \Gamma 1 \ \Gamma 2 \ L2); \Gamma 2 \ x = \text{Some } s'' \rrbracket \implies (\text{pp } \Gamma 1 \ \Gamma 2 \ L2) \ x = \text{Some } s''$ 
apply (simp add: pp-def add: def-pp-def add: safe-def add: unsafe-def)
apply (erule conjE)
apply (erule-tac x=x in allE)+
apply auto
by (case-tac y, simp+)

```

```

lemma safe-Gamma2-triangle:
   $\llbracket \text{def-pp } \Gamma 1 \ \Gamma 2 \ L2; \Gamma 2 \ x = \text{Some } s''; x \in L2 \rrbracket \implies \Gamma 1 \ x = \text{Some } s'' \vee x \notin \text{dom } \Gamma 1$ 
apply (simp add: def-pp-def)

```

**apply** (*erule conjE*) +  
**apply** (*erule-tac x=x in allE*) +  
**apply** (*simp add: unsafe-def, auto*)  
**by** (*case-tac y, simp-all*)

**lemma** *unsafe-triangle*:

$\llbracket \text{def-pp } \Gamma 1 \ \Gamma 2 \ L2;$   
 $\text{def-disjointUnionEnv } \Gamma 2 \ (\text{empty}(x1 \mapsto m));$   
 $L2 \subseteq \text{dom } (\text{disjointUnionEnv } \Gamma 2 \ (\text{empty}(x1 \mapsto m)));$   
 $x \neq x1;$   
 $\Gamma 2 \ x \neq \text{Some } s''; x \in L2 \rrbracket \implies (\text{pp } \Gamma 1 \ \Gamma 2 \ L2) \ x \neq \text{Some } s''$   
**apply** (*simp add: def-pp-def*)  
**apply** (*elim conjE*)  
**apply** (*erule-tac x=x in allE, clarsimp*) +  
**apply** (*simp add: unsafe-def add: pp-def*)  
**apply** (*split split-if-asm, clarsimp*)  
**apply** (*simp add: safe-def*)  
**apply** (*erule conjE*)  
**apply** (*subgoal-tac  $\llbracket x \neq x1; x \in L2; L2 \subseteq \text{dom } (\Gamma 2 + [x1 \mapsto m]) \rrbracket \implies x \in \text{dom } \Gamma 2, \text{clarsimp}$* )  
**apply** (*subgoal-tac  $\text{dom } (\text{disjointUnionEnv } \Gamma 2 \ [x1 \mapsto m]) = \text{dom } \Gamma 2 \cup \text{dom } [x1 \mapsto m], \text{simp}$* )  
**apply** *blast*  
**by** (*rule union-dom-disjointUnionEnv, assumption*)

**lemma** *safe-Gamma-triangle-3*:

$\llbracket \text{def-pp } \Gamma 1 \ \Gamma 2 \ L2;$   
 $\text{def-disjointUnionEnv } \Gamma 2 \ (\text{empty}(x1 \mapsto m));$   
 $L2 \subseteq \text{dom } (\text{disjointUnionEnv } \Gamma 2 \ (\text{empty}(x1 \mapsto m)));$   
 $x \neq x1;$   
 $\Gamma 2 \ x \neq \text{Some } s'';$   
 $x \in L2 \rrbracket$   
 $\implies \Gamma 1 \ x = \text{Some } s'' \vee x \notin \text{dom } \Gamma 1$   
**apply** (*frule unsafe-triangle, assumption+*)  
**apply** (*simp add: pp-def*)  
**apply** (*split split-if-asm, clarsimp*)  
**apply** (*simp add: safe-def, clarsimp*)  
**apply** (*simp add: safe-def add: def-pp-def*)  
**apply** (*erule conjE*)  
**apply** (*erule-tac x=x in allE, simp*)  
**apply** (*simp add: unsafe-def, clarsimp*)  
**by** (*case-tac y, simp-all*)

**lemma** *unsafe-Gamma2-triangle*:

$\llbracket \text{def-pp } \Gamma 1 \ \Gamma 2 \ L2; (\Gamma 2 + [x1 \mapsto m]) \ y \neq \text{Some } s''; y \neq x1 \rrbracket \implies \Gamma 2 \ y \neq \text{Some } s''$   
**apply** (*simp add: disjointUnionEnv-def add: unionEnv-def*)  
**apply** (*split split-if-asm*)  
**apply** *simp*

**apply** (*simp add: def-pp-def*)  
**by** *auto*

**lemma** *condemned-Gamma2-triangle*:

$\llbracket \text{def-pp } \Gamma 1 \ \Gamma 2 \ L2; (\Gamma 2 + [x1 \mapsto s'']) \ x = \text{Some } d'' \rrbracket \implies (\text{pp } \Gamma 1 \ \Gamma 2 \ L2) \ x = \text{Some } d''$

**apply** (*simp add: disjointUnionEnv-def add: unionEnv-def*)  
**apply** (*split split-if-asm*)  
**apply** (*simp add: pp-def add: def-pp-def, clarsimp*)  
**apply** (*simp add: safe-def add: unsafe-def*)  
**apply** (*erule-tac x=x in allE*)+ **apply** (*simp add: dom-def*)  
**apply** (*case-tac ya, simp-all*)  
**apply** (*split split-if-asm*)  
**apply** *simp*  
**by** *simp*

**lemma** *unsafe-triangle-unsafe-2*:

$\llbracket \text{def-pp } \Gamma 1 \ \Gamma 2 \ L2; (\text{pp } \Gamma 1 \ \Gamma 2 \ L2) \ x = \text{Some } d'' \rrbracket \implies \Gamma 2 \ x \neq \text{Some } s''$

**apply** (*simp add: pp-def*)  
**apply** (*split split-if-asm*)  
**apply** (*simp add: safe-def*)  
**apply** (*simp add: safe-def*)  
**apply** (*simp add: def-pp-def*)  
**apply** (*simp add: unsafe-def*)  
**by** *auto*

**lemma** *triangle-d-Gamma1-s-or-not-dom-Gamma1*:

$\llbracket \text{def-disjointUnionEnv } \Gamma 2 \ (\text{empty}(x1 \mapsto m));$   
 $L1 \subseteq \text{dom } \Gamma 1;$   
 $L2 \subseteq \text{dom } (\text{disjointUnionEnv } \Gamma 2 \ (\text{empty}(x1 \mapsto m)));$   
 $z \neq x1;$   
 $\text{def-pp } \Gamma 1 \ \Gamma 2 \ L2 ; (\text{pp } \Gamma 1 \ \Gamma 2 \ L2) \ z = \text{Some } d''; z \in L2 \rrbracket$   
 $\implies (\Gamma 1 \ z = \text{Some } s'' \vee z \notin \text{dom } \Gamma 1)$

**apply** (*subgoal-tac  $\Gamma 2 \ z = \text{Some } d''$* )  
**apply** (*subgoal-tac  $z \in \text{dom } \Gamma 2$* )  
**prefer** 2 **apply** *blast*  
**apply** (*simp add: pp-def*)  
**apply** (*case-tac  $z \in L1$* )  
**apply** (*rule disjI1*)  
**apply** (*subgoal-tac  $z \in \text{dom } \Gamma 1$* )  
**prefer** 2 **apply** *blast*  
**apply** (*simp add: safe-def*)  
**apply** (*split split-if-asm, simp, simp*)  
**apply** (*simp add: def-pp-def*)  
**apply** *auto*  
**apply** (*erule-tac  $x=z$  in allE*)+  
**apply** *simp*  
**apply** (*simp add: unsafe-def, auto*)  
**apply** (*simp add: safe-def*)



```

apply (split split-if-asm, simp)
  apply (simp add: def-pp-def, auto)
apply (simp add: def-pp-def, auto)
apply (simp add: unsafe-def, auto)
apply (simp add: def-pp-def)
apply (subgoal-tac  $z \in \text{dom } \Gamma 2$ )
  apply (erule conjE)
  apply (erule-tac  $x=z$  in allE) +
  apply simp
  apply (simp add: unsafe-def add: pp-def)
  apply (split split-if-asm, simp, simp)
apply (subgoal-tac  $\text{dom } (\Gamma 2 + [x1 \mapsto m]) = \text{dom } \Gamma 2 \cup \text{dom } [x1 \mapsto m]$ )
  prefer 2 apply (rule union-dom-disjointUnionEnv) apply simp
apply (subgoal-tac  $\text{dom } [x1 \mapsto m] = \{x1\}, \text{simp}$ )
  apply blast
by simp

```

```

lemma triangle-d-Gamma1-s-Gamma2-d:
   $\llbracket (pp \ \Gamma 1 \ \Gamma 2 \ L2) \ z = \text{Some } d''; \ \Gamma 1 \ z = \text{Some } s'' \rrbracket \implies \Gamma 2 \ z = \text{Some } d''$ 
apply (simp add: pp-def)
apply (split split-if-asm)
  apply simp
by simp

```

```

lemma triangle-prop:
   $\llbracket x \in \text{dom } \Gamma 1; \ \Gamma 1 \ x \neq \text{Some } s'' \rrbracket$ 
   $\implies x \in \text{dom } (pp \ \Gamma 1 \ \Gamma 2 \ L2) \wedge (pp \ \Gamma 1 \ \Gamma 2 \ L2) \ x \neq \text{Some } s''$ 
by (simp add: pp-def, simp add: dom-def add: safe-def)

```

```

lemma triangle-d-Gamma1-d-or-Gamma2-d:
   $\llbracket (pp \ \Gamma 1 \ \Gamma 2 \ L2) \ z = \text{Some } d'' \rrbracket \implies \Gamma 1 \ z = \text{Some } d'' \vee \Gamma 2 \ z = \text{Some } d''$ 
apply (simp add: pp-def add: safe-def)
apply (split split-if-asm)
  apply simp
by simp

```

```

lemma triangle-d-Gamma2-d-Gamma1-s:
   $\llbracket L1 \subseteq \text{dom } \Gamma 1;$ 
   $\text{def-pp } \Gamma 1 \ \Gamma 2 \ L2;$ 
   $(pp \ \Gamma 1 \ \Gamma 2 \ L2) \ z = \text{Some } d'';$ 
   $\Gamma 2 \ z = \text{Some } d''; \ z \in L1 \rrbracket$ 
   $\implies \Gamma 1 \ z = \text{Some } s'' \wedge z \in L2$ 
apply (subgoal-tac  $z \in \text{dom } \Gamma 1$ )
  prefer 2 apply (erule set-mp, assumption)
apply (rule conjI)
  apply (simp add: pp-def)
  apply (split split-if-asm)
  apply (simp-all add: safe-def)

```

```

apply (simp add: def-pp-def)
apply (erule conjE)
apply (erule-tac x=z in allE)+
apply (simp add: unsafe-def)
apply (erule conjE)
apply (simp add: dom-def)
apply (simp add: def-pp-def)
apply (erule conjE)+
apply (erule-tac x=z in allE,simp)+
by (simp add: unsafe-def add: pp-def add: safe-def,clarsimp)

```

```

lemma Gamma2-d-disjointUnionEnv-m-d:
   $\llbracket \text{def-disjointUnionEnv } \Gamma 2 \text{ (empty}(x1 \mapsto m)\text{)}; \Gamma 2 \text{ } z = \text{Some } d' \rrbracket$ 
   $\implies (\text{disjointUnionEnv } \Gamma 2 \text{ (empty}(x1 \mapsto m)\text{)}) z = \text{Some } d''$ 
by (simp add: disjointUnionEnv-def unionEnv-def def-disjointUnionEnv-def, auto)

```

```

lemma dom-Gamma1-dom-triangle:
   $x \in \text{dom } \Gamma 1 \implies x \in \text{dom } (pp \ \Gamma 1 \ \Gamma 2 \ L2)$ 
by (simp add: pp-def,auto)

```

```

lemma safe-triangle-safe-Gamma1:
   $\llbracket (pp \ \Gamma 1 \ \Gamma 2 \ L2) \ x = \text{Some } s''; \ x \in \text{dom } \Gamma 1 \rrbracket \implies \Gamma 1 \ x = \text{Some } s''$ 
apply (simp add: pp-def)
apply (split split-if-asm)
apply (simp add: safe-def)
by simp

```

```

lemma dom-Gamma2-dom-triangle:
   $\llbracket x \neq x1; \ x \in \text{dom } (\Gamma 2 + [x1 \mapsto m]) \rrbracket \implies x \in \text{dom } (pp \ \Gamma 1 \ \Gamma 2 \ L2)$ 
apply (simp add: pp-def add: disjointUnionEnv-def add: unionEnv-def,clarsimp)
apply (split split-if-asm)
apply (simp add: safe-def add: dom-def)
by simp

```

```

lemma unsafe-Gamma2-unsafe-triangle:
   $\llbracket x \neq x1; \ x \in \text{dom } (\Gamma 2 + [x1 \mapsto m]); \ (\Gamma 2 + [x1 \mapsto m]) \ x \neq \text{Some } s'' \rrbracket$ 
   $\implies (pp \ \Gamma 1 \ \Gamma 2 \ L2) \ x \neq \text{Some } s''$ 
apply (simp add: pp-def add: disjointUnionEnv-def add: unionEnv-def,clarsimp)
apply (split split-if-asm)
apply (simp add: safe-def add: dom-def)
by simp

```

**lemma** *def-disjointUnionEnv-monotone*:  
 $\text{def-disjointUnionEnv } \Gamma 2 \ [x1 \mapsto d''] \implies (\Gamma 2 + [x1 \mapsto d'']) \ x1 = \text{Some } d''$   
**by** (*simp add: def-disjointUnionEnv-def add: disjointUnionEnv-def add: unionEnv-def*)

**lemma** *disjointUnionEnv-d-Gamma2-d*:  
 $\llbracket \text{def-disjointUnionEnv } \Gamma 2 \ [x1 \mapsto d''];$   
 $(\Gamma 2 + [x1 \mapsto d'']) \ z = \text{Some } d'';$   
 $z \neq x1 \rrbracket$   
 $\implies \Gamma 2 \ z = \text{Some } d''$   
**apply** (*simp add: disjointUnionEnv-def add: unionEnv-def*)  
**apply** (*split split-if-asm, clarsimp*)  
**by** (*simp add: def-disjointUnionEnv-def*)

**lemma** *Gamma2-d-disjointUnionEnv-d*:  
 $\llbracket \text{def-disjointUnionEnv } \Gamma 2 \ [x1 \mapsto d''];$   
 $\Gamma 2 \ z = \text{Some } d'' \rrbracket$   
 $\implies (\Gamma 2 + [x1 \mapsto d'']) \ z = \text{Some } d''$   
**by** (*simp add: disjointUnionEnv-def unionEnv-def def-disjointUnionEnv-def, auto*)

**lemma** *dom-disjointUnionEnv-subset-dom-extend*:  
 $\llbracket \text{def-disjointUnionEnv } \Gamma 2 \ [x1 \mapsto d''];$   
 $\text{dom } (\text{disjointUnionEnv } \Gamma 2 \ (\text{empty}(x1 \mapsto d''))) \subseteq \text{dom } (E1(x1 \mapsto v1));$   
 $(\Gamma 2 + [x1 \mapsto d'']) \ x = \text{Some } s' \rrbracket$   
 $\implies x \in \text{dom } E1$   
**apply** (*simp add: disjointUnionEnv-def add: unionEnv-def*)  
**apply** (*subgoal-tac x ≠ x1*)  
**apply** *blast*  
**apply** (*simp add: def-disjointUnionEnv-def*)  
**apply** (*split split-if-asm, clarsimp, simp*)  
**by** (*split split-if-asm, clarsimp, simp*)

**lemma** *Gamma2-d-triangle-d*:  
 $\llbracket \text{def-pp } \Gamma 1 \ \Gamma 2 \ L2; \Gamma 2 \ x = \text{Some } d'' \rrbracket \implies (\text{pp } \Gamma 1 \ \Gamma 2 \ L2) \ x = \text{Some } d''$   
**apply** (*simp add: pp-def add: def-pp-def, clarsimp*)  
**apply** (*simp add: safe-def add: unsafe-def*)  
**apply** (*erule-tac x=x in allE*) + **apply** (*simp add: dom-def*)  
**apply** (*elim conjE*)  
**apply** (*case-tac y, simp-all*)  
**done**

**lemma** *disjounitUnionEnv-d-triangle-d*:  
 $\llbracket x \neq x1; \text{def-pp } \Gamma 1 \ \Gamma 2 \ L2; (\Gamma 2 + [x1 \mapsto d'']) \ x = \text{Some } d'' \rrbracket \implies (\text{pp } \Gamma 1 \ \Gamma 2 \ L2)$   
 $x = \text{Some } d''$   
**apply** (*simp add: disjointUnionEnv-def add: unionEnv-def*)

```

apply (split split-if-asm)
apply (simp add: pp-def add: def-pp-def, clarsimp)
apply (simp add: safe-def add: unsafe-def)
apply (erule-tac x=x in allE)+ apply (simp add: dom-def)
apply (case-tac ya, simp-all)
done

```

**end**

## 2 Normal form values and heap

```

theory SafeHeap
imports Main
begin

```

```

types
  Location = nat
  Constructor = string
  FunName = string

```

— Normal form values

```

datatype Val = Loc Location | IntT int | BoolT bool

```

— Destructors of datatype val

```

consts the-IntT :: Val  $\Rightarrow$  int

```

```

primrec
  the-IntT (IntT i) = i

```

```

consts the-BoolT :: Val  $\Rightarrow$  bool

```

```

primrec
  the-BoolT (BoolT b) = b

```

— check if is constant bool

```

constdefs isBool :: Val  $\Rightarrow$  bool
  isBool v  $\equiv$  (case v of (BoolT -)  $\Rightarrow$  True
                | -  $\Rightarrow$  False )

```

A heap is a partial mapping from locations to cells. But, as it is split into regions, the mapping tells also the region where the cell lives. The second component is the highest live region  $k$ . A consistent heap  $(h, k)$  has cells

only in regions  $0 \dots k$ .

#### types

$Cell = Constructor \times Val\ list$   
 $Region = nat$   
 $HeapMap = Location \rightarrow (Region \times Cell)$   
 $Heap = HeapMap \times nat$

#### consts

$restrictToRegion :: Heap \Rightarrow Region \Rightarrow Heap$  (**infix**  $\downarrow 110$ )

#### primrec

$(h,k) \downarrow k0 = (let\ A = \{ p \ .\ p \in dom\ h \ \&\ fst\ (the\ (h\ p)) \leq k0 \}$   
 $\quad in\ (h \upharpoonright A, k0))$

#### definition

$rangeHeap :: HeapMap \Rightarrow Val\ set$  **where**  
 $rangeHeap\ h = \{v.\ EX\ p\ j\ C\ vn.\ h\ p = Some\ (j,C,vn) \wedge v \in set\ vn\}$

#### definition

$fresh :: Location \Rightarrow HeapMap \Rightarrow bool$  **where**  
 $fresh\ p\ h = (p \notin dom\ h \wedge (Loc\ p) \notin rangeHeap\ h)$

#### definition

$domLoc :: HeapMap \Rightarrow Val\ set$  **where**  
 $domLoc\ h = \{l.\ EX\ p.\ p \in dom\ h \wedge l = Loc\ p\}$

**declare**  $rangeHeap-def$  [*simp del*]

**declare**  $fresh-def$  [*simp del*]

**declare**  $domLoc-def$  [*simp del*]

#### constdefs

$getFresh :: HeapMap \Rightarrow Location$   
 $getFresh\ h \equiv SOME\ b.\ fresh\ b\ h$

#### constdefs

$self :: string$  — this identifies the topmost region referenced in a function body  
 $self \equiv "self"$

The constructor table tells, for each constructor, the number of arguments and a description of each one. The second nat gives the alternative  $0..n - 1$  corresponding to this constructor in every **case** of its type

**datatype**  $ArgType = IntArg \mid BoolArg \mid NonRecursive \mid Recursive$

#### types

$ConstructorTableType = (Constructor \times (nat \times nat \times ArgType\ list))\ list$   
 $ConstructorTableFun = Constructor \rightarrow (nat \times nat \times ArgType\ list)$

**consts****consts**

constdefs

```

constdefs descendants :: Location ⇒ Heap ⇒ Location set
descendants l h ≡ case ((fst h) l) of Some c ⇒ getNonBasicValuesCell (snd c)
                | None ⇒ {}

```

```

constdefs isConstant :: Val ⇒ bool
isConstant v ≡ (case v of (IntT -) ⇒ True
                        | (BoolT -) ⇒ True
                        | - ⇒ False)
end

```

### 3 Useful functions and theorems from the Haskell Library or Prelude

```

theory HaskellLib
imports Main
begin

```

Function `mapAccumL` is a powerful combination of `map` and `foldl`. Functions `unzip3` and `unzip` are respectively the inverse of `zip3` and `zip`.

```

consts
  mapAccumL :: ('a => 'b => 'a × 'c) => 'a => 'b list => 'a × 'c list
  zipWith   :: ('a => 'b => 'c) => 'a list => 'b list => 'c list
  unzip3    :: ('a × 'b × 'c) list => 'a list × 'b list × 'c list
  unzip     :: ('a × 'b) list => 'a list × 'b list

```

```

primrec
  mapAccumL f s [] = (s, [])
  mapAccumL f s (x#xs) = (let (s',y) = f s x;
                           (s'',ys) = mapAccumL f s' xs
                           in (s'',y#ys))

```

Some lemmas about `mapAccumL`

```

lemma mapAccumL-non-empty:
  [(s'',ys) = mapAccumL f s xs;
   xs = x#xx]
  ⟹ (∃ s' y ys'.
    (s',y) = f s x
    ∧ ys = y # ys')

```

```

apply clarify
apply (unfold mapAccumL.simps)
apply (rule-tac x=fst (f s x) in exI)
apply (rule-tac x=snd (f s x) in exI)
apply (rule-tac x=snd (mapAccumL f (fst (f s x)) xx) in exI)
apply (rule conjI)

```

**apply** *simp*  
**apply** (*case-tac* *f s x, simp*)  
**by** (*case-tac* *mapAccumL f a xx, simp*)

**lemma** *mapAccumL-non-empty2*:

$\llbracket (s'', ys) = \text{mapAccumL } f \ s \ xs;$   
 $xs = x \# xx$   
 $\rrbracket \implies (\exists \ s' \ y \ ys'.$   
 $\quad (s', y) = f \ s \ x$   
 $\quad \wedge (s'', ys') = \text{mapAccumL } f \ s' \ xx$   
 $\quad \wedge ys = y \ \# \ ys')$

**apply** *clarify*  
**apply** (*unfold* *mapAccumL.simps*)  
**apply** (*rule-tac* *x=fst (f s x) in exI*)  
**apply** (*rule-tac* *x=snd (f s x) in exI*)  
**apply** (*rule-tac* *x=snd (mapAccumL f (fst (f s x)) xx) in exI*)  
**apply** (*rule conjI*)  
**apply** *simp*  
**apply** (*rule conjI*)  
**apply** (*case-tac* *f s x*) **apply** (*simp*)  
**apply** (*case-tac* *mapAccumL f a xx*)  
**apply** (*simp*)  
**apply** (*case-tac* *f s x*) **apply** (*simp*)  
**apply** (*case-tac* *mapAccumL f a xx*)  
**apply** *simp*  
**done**

**axioms** *mapAccumL-non-empty3*:

$\llbracket (s'', ys) = \text{mapAccumL } f \ s \ xs;$   
 $0 < \text{length } xs$   
 $\rrbracket \implies (\exists \ s' \ y \ ys'.$   
 $\quad (s', y) = f \ s \ (xs!0)$   
 $\quad \wedge (s'', ys') = \text{mapAccumL } f \ s' \ (\text{tl } xs))$

**axioms** *mapAccumL-two-elements*:

$\llbracket (s3, ys) = \text{mapAccumL } f \ s \ xs;$   
 $xs = x1 \ \# \ x2 \ \# \ xx$   
 $\rrbracket \implies (\exists \ s1 \ s2 \ y1 \ y2 \ ys3.$   
 $\quad (s1, y1) = f \ s \ x1$   
 $\quad \wedge (s2, y2) = f \ s1 \ x2$   
 $\quad \wedge (s3, ys3) = \text{mapAccumL } f \ s2 \ xx$   
 $\quad \wedge ys = y1 \ \# \ y2 \ \# \ ys3)$

**axioms** *mapAccumL-split*:

$\llbracket (s2, ys) = \text{mapAccumL } f \ s \ xs;$   
 $xs1 \ @ \ xs2 = xs$   
 $\rrbracket \implies (\exists \ s1 \ ys1 \ ys2 .$   
 $\quad (s1, ys1) = \text{mapAccumL } f \ s \ xs1$   
 $\quad \wedge (s2, ys2) = \text{mapAccumL } f \ s1 \ xs2)$



$$\wedge ys = ys1 @ ys2)$$

**axioms** *mapAccumL-one-more*:

$$\begin{aligned} & \llbracket (s1, ys) = \text{mapAccumL } f \ s \ xs; \\ & \quad (s2, y) = f \ s1 \ x \\ & \rrbracket \implies (s2, ys@[y]) = \text{mapAccumL } f \ s \ (xs@[x]) \end{aligned}$$

Some integer arithmetic lemmas

**lemma** *sum-nat*:

$$\llbracket (x1::nat)=x2; (y1::nat)=y2 \rrbracket \implies x1+y1=x2+y2$$

**apply** *arith*

**done**

**axioms** *sum-subtract*:

$$(x::nat)-y+(z-x)=z-y$$

**axioms** *additions1*:

$$\begin{aligned} & \llbracket i < m; \text{Suc } m + n \leq l \rrbracket \implies \\ & \quad m - i < \text{nat } (\text{int } l - 1) - n + 1 - (\text{nat } (\text{int } l - 1) - \text{Suc } m - n + 1) \end{aligned}$$

**axioms** *additions2*:

$$\begin{aligned} & \llbracket i < m; \text{Suc } m + n \leq l \rrbracket \implies \\ & \quad \text{nat } (\text{int } l - 1) - m + (m - \text{Suc } i) = \text{nat } (\text{int } l - 1) - \text{Suc } m + (m - i) \end{aligned}$$

**axioms** *additions3*:

$$\begin{aligned} & \llbracket i < m; \text{Suc } m + n \leq l \rrbracket \implies \\ & \quad \text{nat } (\text{int } l - 1) - \text{Suc } (m + n) + (m - i) = \text{nat } (\text{int } l - 1) - (m + n) + (m - \text{Suc } i) \end{aligned}$$

**axioms** *additions4*:

$$\begin{aligned} & \llbracket \text{Suc } m + n \leq l \rrbracket \implies \\ & \quad \text{nat } (\text{int } l - 1) - m = \text{Suc } (\text{nat } (\text{int } l - 1) - \text{Suc } m) \end{aligned}$$

**axioms** *additions5*:

$$\begin{aligned} & \llbracket \text{Suc } m + n \leq l \rrbracket \implies \\ & \quad \text{Suc } (\text{nat } (\text{int } l - 1) - \text{Suc } (m + n)) = \text{nat } (\text{int } l - 1 - \text{int } n - \text{int } m) \end{aligned}$$

**axioms** *additions6*:

$$\begin{aligned} & \llbracket \text{Suc } m + n \leq l \rrbracket \implies \\ & \quad n + (\text{nat } (\text{int } l - 1) - \text{Suc } (m + n)) < \text{nat } (\text{int } l - 1) \end{aligned}$$

Some lemmas about lists

**lemma** *list-non-empty*:

$$0 < \text{length } xs \implies (\exists \ y \ ys . xs = y \# \ ys)$$

**apply** *auto*

**apply** (*insert neq-Nil-conv* [of *xs*])

**by** *simp*

**axioms** *drop-nth*:

$$n < \text{length } xs \implies (\exists \ y \ ys . \text{drop } n \ xs = y \# \ ys \wedge xs!n = y)$$

**axioms** *drop-nth3*:

$$n < \text{length } xs \implies \text{drop } n \ xs = (xs!n) \# \text{drop } (\text{Suc } n) \ xs$$

**axioms** *drop-take-Suc*:

$$xs = (\text{take } n \text{ } xs) @ (z \# zs) \implies \text{drop } (\text{Suc } n) \text{ } xs = zs$$

**axioms** *drop-nth2*:

$$\begin{aligned} & \llbracket n < \text{length } xs; \text{drop } n \text{ } xs = ys \rrbracket \\ & \implies ys = xs!n \# \text{tl } xs \end{aligned}$$

**axioms** *drop-append2*:

$$\begin{aligned} & \llbracket \text{drop } n \text{ } xs = zs1 @ ys1 @ ys2 @ zs2 @ \text{rest}; \\ & \quad \text{drop } (m - n) \text{ } (zs1 @ ys1 @ ys2 @ zs2) = ys1 @ \text{rest}' \rrbracket \implies \\ & \quad \text{drop } (m + \text{length } ys1 - n) \text{ } (zs1 @ ys1 @ ys2 @ zs2) = ys2 @ zs2 \end{aligned}$$

**axioms** *drop-append3*:

$$\begin{aligned} & \llbracket \text{drop } n \text{ } xs = xs1 @ \text{rest}; \\ & \quad \text{drop } (m - n) \text{ } xs1 = ys1 @ ys2 \rrbracket \implies \\ & \quad \text{drop } m \text{ } xs = ys1 @ ys2 @ \text{rest} \end{aligned}$$

**lemma** *nth-via-drop-append*:  $\text{drop } n \text{ } xs = (y \# ys) @ zs \implies xs!n = y$

**apply** (*induct xs arbitrary: n, simp*)

**by** (*simp add: drop-Cons nth-Cons split: nat.splits*)

**lemma** *drop-Suc-append*:

$$\text{drop } n \text{ } xs = (y \# ys) @ zs \implies \text{drop } (\text{Suc } n) \text{ } xs = ys @ zs$$

**apply** (*induct xs arbitrary: n, simp*)

**apply** (*simp add: drop-Cons*)

**by** (*simp split: nat.splits*)

**lemma** *nth-via-drop-append-2*:  $\text{drop } n \text{ } xs = ((y \# ys) @ ws @ zs) @ ms \implies xs!n = y$

**apply** (*induct xs arbitrary: n, simp*)

**by** (*simp add: drop-Cons nth-Cons split: nat.splits*)

**lemma** *drop-Suc-append-2*:

$$\text{drop } n \text{ } xs = ((y \# ys) @ ws @ zs) @ ms \implies \text{drop } (\text{Suc } n) \text{ } xs = ys @ ws @ zs @ ms$$

**apply** (*induct xs arbitrary: n, simp*)

**apply** (*simp add: drop-Cons*)

**by** (*simp split: nat.splits*)

**axioms** *drop-append-length*:

$$\text{drop } n \text{ } xs = [] @ ys @ zs @ ms \implies \text{drop } (n + \text{length } ys) \text{ } xs = zs @ ms$$

**axioms** *take-length*:

$$n \leq \text{length } xs \implies n = \text{length } (\text{take } n \text{ } xs)$$

**axioms** *take-append2*:

$$n < \text{length } xs \implies x \# \text{take } n \text{ } xs = \text{take } n \text{ } (x \# xs) @ [(x \# xs)!n]$$

**axioms** *take-append3*:

$$\text{Suc } n \leq \text{length } xs \implies \text{take } (\text{Suc } n) \text{ } xs = \text{take } n \text{ } xs @ [xs!n]$$

**axioms** *concat1*:

$$xs @ y \# ys = (xs @ [y]) @ ys$$

**axioms** *concat2*:

$$xs1 = xs2 \implies xs1 @ ys = xs2 @ ys$$

**axioms** *upt-length*:

$$n \leq m \implies \text{length } [n..<m] = m - n$$

Some lemmas about finite maps

**axioms** *map-of-distinct*:

$$\begin{aligned} & \llbracket \text{distinct } (\text{map fst } xys); \\ & \quad l < \text{length } xys; \\ & \quad (x,y) = xys ! l \\ & \rrbracket \implies \text{map-of } xys \text{ } x = \text{Some } y \end{aligned}$$

**axioms** *map-of-distinct2*:

$$\begin{aligned} & \text{map-of } xys \text{ } x = \text{Some } y \\ & \implies (\exists l . l < \text{length } xys \wedge (x,y) = xys ! l) \end{aligned}$$

**axioms** *map-upds-nth*:

$$i < m - n \implies (A([n..<m] \text{ } [\mapsto] \text{ } xs)) (n+i) = \text{Some } (xs ! i)$$

— The unzip3 function of Haskell library

**primrec**

$$\begin{aligned} \text{unzip3 } [] &= ([], [], []) \\ \text{unzip3 } (tup \# tups) &= (\text{let } (xs, ys, zs) = \text{unzip3 } tups; \\ & \quad (x, y, z) = tup \\ & \text{in } (x \# xs, y \# ys, z \# zs)) \end{aligned}$$

**axioms** *unzip3-length*:

$$\text{unzip3 } xs = (ys1, ys2, ys3) \implies \text{length } ys1 = \text{length } ys2$$

**primrec**

$$\begin{aligned} \text{unzip } [] &= ([], []) \\ \text{unzip } (tup \# tups) &= (\text{let } (xs, ys) = \text{unzip } tups; \\ & \quad (x, y) = tup \\ & \text{in } (x \# xs, y \# ys)) \end{aligned}$$

**primrec**

$$\begin{aligned} \text{zipWith } f \text{ } (x \# xs) \text{ } yy &= (\text{case } yy \text{ of} \\ & \quad [] => [] \\ & \quad | y \# ys => f \text{ } x \text{ } y \# \text{zipWith } f \text{ } xs \text{ } ys) \end{aligned}$$

```

zipWith f [] yy = []

axioms zipWith-length:
  length (zipWith f xs ys) = min (length xs) (length ys)

— The Haskell sum type Either

datatype ('a,'b) Either = Left 'a | Right 'b

— insertion sort for list of strings

constdefs
  leString :: string => string => bool
  leString s1 s2 == True

consts
  ins :: string => string list => string list
primrec
  ins s [] = [s]
  ins s (s' # ss) = (if leString s s' then s # s' # ss
                     else s' # ins s ss)

fun sort :: string list => string list
where
  sort ss = foldr ins ss []

fun subList :: 'a list => 'a list => bool
where
  subList xs ys = (∃ hs ts. ys = hs @ xs @ ts)

end

```

## 4 Normalized Safe Expressions

```

theory SafeExpr imports ../SafeImp/SafeHeap ../SafeImp/HaskellLib
begin

```

This is a somewhat simplified copy of the abstract syntax used by the Safe compiler. The idea is that the Haskell code generated by Isabelle for the definition of the *trProg* function, translating from CoreSafe to SafeImp, can be directly used as a phase of the compiler. The simplifications are in expression LetE and in the definition of 'a Der, in order to avoid unnecessary mutual recursion between types. First, we define the key elements of Core-

Safe abstract syntax.

**constdefs**

```
intType :: string
intType == "Int"
boolType :: string
boolType == "Bool"
```

**datatype** *ExpTipo* = *VarT* string  
                   | *ConstrT* string *ExpTipo* list bool string list  
                   | *Rec*

**datatype** *AltDato* = *ConstrA* string (*ExpTipo* list) string

**types**    *DecData* = string × string list × string list × *AltDato* list

**datatype** *Lit* = *LitN* int | *LitB* bool

**datatype** 'a *Patron* = *ConstP* *Lit*  
                   | *VarP* string 'a  
                   | *ConstrP* string ('a *Patron*) list 'a

Now we define the CoreSafe expressions.

**types**

```
ProgVar = string
RegVar  = string
```

**datatype** 'a *Exp* = *ConstE* *Lit* 'a  
                   | *ConstrE* string ('a *Exp*) list *RegVar* 'a  
                   | *VarE*    *ProgVar* 'a  
                   | *CopyE*    *ProgVar* *RegVar* 'a        ( - @ - - 90)  
                   | *ReuseE*   *ProgVar* 'a  
                   | *AppE*    *FunName* ('a *Exp*) list *RegVar* list 'a  
                   | *LetE*    string ('a *Exp*) ('a *Exp*) 'a  
                                   (Let - = - In - - 95)  
  
                   | *CaseE*   ('a *Exp*) ('a *Patron* × 'a *Exp*) list 'a  
                                   (Case - Of - - 95)  
                   | *CaseDE* ('a *Exp*) ('a *Patron* × 'a *Exp*) list 'a  
                                   (CaseD - Of - - 95)

Now, the rest of the abstract syntax.

**datatype** 'a *Der*    = *Simple* ('a *Exp*) int list

**types**

```
'a Izq  = string × ('a Patron × bool) list × string list
'a Def  = ExpTipo list × 'a Izq × 'a Der
'a Prog = DecData list × ('a Def) list × 'a Exp
```

## 4.1 Free Variables

```
fun pat2var :: 'a Patron => string
where
  pat2var (VarP x -) = x
```

```
fun extractP :: 'a Patron => (string × string list)
where
  extractP (ConstrP C ps -) = (
    let xs = map pat2var ps
    in (C,xs))
| extractP - = ([],[])
```

```
fun extractVar :: 'a Patron => string list
where
  extractVar (ConstrP C ps a) = map pat2var ps
| extractVar (ConstP l) = []
| extractVar (VarP v a) = [v]
```

```
consts varProgPat :: 'a Patron => string set
      varProgPats :: 'a Patron list => string set
```

```
primrec
varProgPat (ConstP l)          = {}
varProgPat (VarP x a)          = {x}
varProgPat (ConstrP C pats a) = varProgPats pats

varProgPats []                = {}
varProgPats (pat#pats) = varProgPat pat ∪ varProgPats pats
```

```
consts varProg :: 'a Exp => ProgVar set
      varProgs :: 'a Exp list => ProgVar set
      varProgs' :: 'a Exp list => ProgVar set
      varProgAlts :: ('a Patron × 'a Exp) list => string set
      varProgAlts' :: ('a Patron × 'a Exp) list => string set
      varProgTup :: 'a Patron × 'a Exp => string set
      varProgTup' :: 'a Patron × 'a Exp => string set
```

```
primrec
varProg (ConstE Lit a)          = {}
varProg (ConstrE C exps r a)    = varProgs exps
varProg (VarE x a)              = {x}
varProg (CopyE x r a)           = {x}
varProg (ReuseE x a)            = {x}
varProg (AppE fn exps rs a)     = varProgs' exps
varProg (LetE x1 e1 e2 a)       = varProg e1 ∪ varProg e2 ∪ {x1}
varProg (CaseE exp alts a)      = varProg exp ∪ varProgAlts alts
```

$$\text{varProg } (\text{CaseDE } \text{exp } \text{alts } a) = \text{varProg } \text{exp} \cup \text{varProgAlts}' \text{alts}$$

$$\begin{aligned} \text{varProgs } [] &= \{\} \\ \text{varProgs } (\text{exp} \# \text{exps}) &= \text{varProg } \text{exp} \cup \text{varProgs } \text{exps} \end{aligned}$$

$$\begin{aligned} \text{varProgs}' [] &= \{\} \\ \text{varProgs}' (\text{exp} \# \text{exps}) &= \text{varProg } \text{exp} \cup \text{varProgs}' \text{exps} \end{aligned}$$

$$\begin{aligned} \text{varProgAlts } [] &= \{\} \\ \text{varProgAlts } (\text{alt} \# \text{alts}) &= \text{varProgTup } \text{alt} \cup \text{varProgAlts } \text{alts} \end{aligned}$$

$$\begin{aligned} \text{varProgAlts}' [] &= \{\} \\ \text{varProgAlts}' (\text{alt} \# \text{alts}) &= \text{varProgTup}' \text{alt} \cup \text{varProgAlts}' \text{alts} \end{aligned}$$

$$\text{varProgTup } (\text{pat}, e) = \text{varProgPat } \text{pat} \cup \text{varProg } e$$

$$\text{varProgTup}' (\text{pat}, e) = \text{varProgPat } \text{pat} \cup \text{varProg } e$$

$$\begin{aligned} \text{consts } \text{fv} &:: 'a \text{ Exp} \Rightarrow \text{string set} \\ \text{fvs} &:: 'a \text{ Exp list} \Rightarrow \text{string set} \\ \text{fvs}' &:: 'a \text{ Exp list} \Rightarrow \text{string set} \\ \text{fvAlts} &:: ('a \text{ Patron} \times 'a \text{ Exp}) \text{ list} \Rightarrow \text{string set} \\ \text{fvAlts}' &:: ('a \text{ Patron} \times 'a \text{ Exp}) \text{ list} \Rightarrow \text{string set} \\ \text{fvTup} &:: 'a \text{ Patron} \times 'a \text{ Exp} \Rightarrow \text{string set} \\ \text{fvTup}' &:: 'a \text{ Patron} \times 'a \text{ Exp} \Rightarrow \text{string set} \end{aligned}$$

#### **primrec**

$$\begin{aligned} \text{fv } (\text{ConstE } \text{Lit } a) &= \{\} \\ \text{fv } (\text{ConstrE } C \text{ exps } \text{rv } a) &= \text{fvs } \text{exps} \\ \text{fv } (\text{VarE } x a) &= \{x\} \\ \text{fv } (\text{CopyE } x \text{rv } a) &= \{x\} \\ \text{fv } (\text{ReuseE } x a) &= \{x\} \\ \text{fv } (\text{AppE } \text{fn } \text{exps } \text{rvs } a) &= \text{fvs}' \text{exps} \\ \text{fv } (\text{LetE } x1 \text{ e1 } \text{e2 } a) &= \text{fv } \text{e1} \cup \text{fv } \text{e2} - \{x1\} \\ \text{fv } (\text{CaseE } \text{exp } \text{paterxs } a) &= \text{fvAlts } \text{paterxs} \cup \text{fv } \text{exp} \\ \text{fv } (\text{CaseDE } \text{exp } \text{paterxs } a) &= \text{fvAlts}' \text{paterxs} \cup \text{fv } \text{exp} \end{aligned}$$

$$\begin{aligned} \text{fvs } [] &= \{\} \\ \text{fvs } (\text{exp} \# \text{exps}) &= \text{fv } \text{exp} \cup \text{fvs } \text{exps} \end{aligned}$$

$$\begin{aligned} \text{fvs}' [] &= \{\} \\ \text{fvs}' (\text{exp} \# \text{exps}) &= \text{fv } \text{exp} \cup \text{fvs}' \text{exps} \end{aligned}$$

$$\begin{aligned} \text{fvAlts } [] &= \{\} \\ \text{fvAlts } (\text{alt} \# \text{alts}) &= \text{fvTup } \text{alt} \cup \text{fvAlts } \text{alts} \end{aligned}$$

$$\begin{aligned} \text{fvAlts}' [] &= \{\} \\ \text{fvAlts}' (\text{alt} \# \text{alts}) &= \text{fvTup}' \text{alt} \cup \text{fvAlts}' \text{alts} \end{aligned}$$

$fvTup (pat, e) = fv\ e - set\ (snd\ (extractP\ pat))$

$fvTup' (pat, e) = fv\ e - set\ (snd\ (extractP\ pat))$

**consts**  $fvReg \quad :: 'a\ Exp \Rightarrow string\ set$   
 $fvReg \quad :: 'a\ Exp\ list \Rightarrow string\ set$   
 $fvReg' \quad :: 'a\ Exp\ list \Rightarrow string\ set$   
 $fvAltsReg \quad :: ('a\ Patron \times 'a\ Exp)\ list \Rightarrow string\ set$   
 $fvAltsReg' \quad :: ('a\ Patron \times 'a\ Exp)\ list \Rightarrow string\ set$   
 $fvTupReg \quad :: 'a\ Patron \times 'a\ Exp \Rightarrow string\ set$   
 $fvTupReg' \quad :: 'a\ Patron \times 'a\ Exp \Rightarrow string\ set$

**primrec**

$fvReg\ (ConstE\ Lit\ a) = \{\}$   
 $fvReg\ (ConstrE\ C\ exps\ r\ a) = \{r\}$   
 $fvReg\ (VarE\ x\ a) = \{\}$   
 $fvReg\ (CopyE\ x\ r\ a) = \{r\}$   
 $fvReg\ (ReuseE\ x\ a) = \{\}$   
 $fvReg\ (AppE\ fn\ exps\ rvs\ a) = set\ rvs$   
 $fvReg\ (LetE\ x1\ e1\ e2\ a) = fvReg\ e1 \cup fvReg\ e2$   
 $fvReg\ (CaseE\ exp\ patexps\ a) = fvAltsReg\ patexps$   
 $fvReg\ (CaseDE\ exp\ patexps\ a) = fvAltsReg'\ patexps$

$fvAltsReg\ [] = \{\}$   
 $fvAltsReg\ (alt\ \#alts) = fvTupReg\ alt \cup fvAltsReg\ alts$

$fvAltsReg'\ [] = \{\}$   
 $fvAltsReg'\ (alt\ \#alts) = fvTupReg'\ alt \cup fvAltsReg'\ alts$

$fvTupReg\ (pat, e) = fvReg\ e$

$fvTupReg'\ (pat, e) = fvReg\ e$

**consts**  $boundVar \quad :: 'a\ Exp \Rightarrow string\ set$   
 $boundVars \quad :: 'a\ Exp\ list \Rightarrow string\ set$   
 $boundVars' \quad :: 'a\ Exp\ list \Rightarrow string\ set$   
 $boundVarAlts \quad :: ('a\ Patron \times 'a\ Exp)\ list \Rightarrow string\ set$   
 $boundVarAlts' \quad :: ('a\ Patron \times 'a\ Exp)\ list \Rightarrow string\ set$   
 $boundVarTup \quad :: 'a\ Patron \times 'a\ Exp \Rightarrow string\ set$   
 $boundVarTup' \quad :: 'a\ Patron \times 'a\ Exp \Rightarrow string\ set$

**primrec**

$boundVar\ (ConstE\ Lit\ a) = \{\}$   
 $boundVar\ (ConstrE\ C\ exps\ rv\ a) = boundVars\ exps$   
 $boundVar\ (VarE\ x\ a) = \{\}$   
 $boundVar\ (CopyE\ x\ rv\ a) = \{\}$



$$\begin{aligned}
\text{boundVar } (\text{ReuseE } x \ a) &= \{\} \\
\text{boundVar } (\text{AppE } \text{fn } \text{exps } \text{rvs } a) &= \{\} \\
\text{boundVar } (\text{LetE } x1 \ e1 \ e2 \ a) &= \{x1\} \\
\text{boundVar } (\text{CaseE } \text{exp } \text{patexps } a) &= \text{boundVarAlts } \text{patexps} \\
\text{boundVar } (\text{CaseDE } \text{exp } \text{patexps } a) &= \text{boundVarAlts}' \ \text{patexps}
\end{aligned}$$

$$\begin{aligned}
\text{boundVars } [] &= \{\} \\
\text{boundVars } (\text{exp} \# \text{exps}) &= \text{boundVar } \text{exp} \cup \text{boundVars } \text{exps}
\end{aligned}$$

$$\begin{aligned}
\text{boundVarAlts } [] &= \{\} \\
\text{boundVarAlts } (\text{alt} \# \text{alts}) &= \text{boundVarTup } \text{alt} \cup \text{boundVarAlts } \text{alts}
\end{aligned}$$

$$\begin{aligned}
\text{boundVarAlts}' [] &= \{\} \\
\text{boundVarAlts}' (\text{alt} \# \text{alts}) &= \text{boundVarTup}' \ \text{alt} \cup \text{boundVarAlts}' \ \text{alts}
\end{aligned}$$

$$\text{boundVarTup } (\text{pat}, e) = \text{set } (\text{extractVar } \text{pat})$$

$$\text{boundVarTup}' (\text{pat}, e) = \text{set } (\text{extractVar } \text{pat})$$

A runtime environment consists of two partial mappings: one from program variables to normal form values, and one from region variables to actual regions.

#### types

$$\text{Environment} = (\text{ProgVar} \rightarrow \text{Val}) \times (\text{RegVar} \rightarrow \text{Region})$$

The runtime system provides a 'copy' function which generates a new data structure from a given location by copying those cells pointed to by recursive argument positions of data constructors. The  $\Sigma$  environment provides the textual definitions of previously defined Safe functions. Some auxiliary functions: 'extend' extends an environment with a collection of bindings from variables to values; 'fresh' is a predicate telling whether a variable is fresh with respect to a heap; 'atom2val', given an environment and an atom (a program variable or a literal expression) returns its corresponding value; 'atom2var', given an expression return its corresponding variable; 'atom', is a predicate telling whether an expression is an atom.

**constdefs**  $\text{recursiveArgs} :: \text{Constructor} \Rightarrow \text{bool list}$

$$\begin{aligned}
\text{recursiveArgs } C &\equiv (\text{let} \\
&\quad (-, -, \text{args}) = \text{the } (\text{ConstructorTable } C) \\
&\quad \text{in map } (\%a. a = \text{Recursive}) \ \text{args})
\end{aligned}$$

**function**  $\text{copy}' :: [\text{Region}, \text{HeapMap}, (\text{Val} \times \text{bool})] \Rightarrow (\text{HeapMap} \times \text{Val})$   
**where**

$$\begin{aligned}
&\text{copy}' \ j \ h \ (v, \text{False}) &&= (h, v) \\
| \text{copy}' \ j \ h \ (\text{Val.Loc } p, \text{True}) &&= (\text{let} \\
&\quad (k, C, ps) = \text{the } (h \ p); \\
&\quad bs &= \text{recursiveArgs } C; \\
&\quad pbs &= \text{zip } ps \ bs;
\end{aligned}$$

```

      (h',ps') = mapAccumL (copy' j) h pbs;
      p'       = getFresh h'
      in (h'(p'↦(j,C,ps')), Val.Loc p')
| copy' j h (IntT i, True)   = (h, IntT i)
| copy' j h (BoolT b, True) = (h, BoolT b)
by pat-completeness auto

```

```

function copy :: [Heap, Location, Region] ⇒ (Heap × Location)
where
  copy (h,k) p j = (let
    (h',p') = copy' j h (Val.Loc p, True)
    in case p' of (Val.Loc q) ⇒ ((h',k), q))
by pat-completeness auto
termination by (relation {}) simp

```

```

types FunDefEnv = string → ProgVar list × RegVar list × unit Exp

```

```

consts
  Σf :: FunDefEnv

```

```

constdefs bodyAPP :: FunDefEnv ⇒ string ⇒ unit Exp
  bodyAPP Σ f == (case Σ f of Some (xs,rs,ef) ⇒ ef)

```

```

constdefs varsAPP :: FunDefEnv ⇒ string ⇒ string list
  varsAPP Σ f ≡ (case Σ f of Some (xs,rs,ef) ⇒ xs)

```

```

constdefs regionsAPP :: FunDefEnv ⇒ string ⇒ string list
  regionsAPP Σ f ≡ (case Σ f of Some (xs,rs,ef) ⇒ rs)

```

```

definition
  extend :: [string → Val, ProgVar list, Val list] ⇒ (ProgVar → Val) where
  extend E xs vs = E ++ map-of (zip xs vs)

```

```

definition
  def-extend :: [string → Val, ProgVar list, Val list] ⇒ bool where
  def-extend E xs vs = (set xs ∩ dom E = {}) ∧ length xs = length vs ∧ distinct
  xs ∧ (∀ x ∈ set xs. x ≠ self)

```

```

fun atom2val :: (ProgVar → Val) ⇒ 'a Exp ⇒ Val
where
  atom2val E (ConstE (LitN i) a) = IntT i
| atom2val E (ConstE (LitB b) a) = BoolT b
| atom2val E (VarE x a)          = the (E x)

```

```

fun atom2var :: 'a Exp  $\Rightarrow$  string
where
  atom2var (VarE x a) = x

```

```

fun atom :: 'a Exp  $\Rightarrow$  bool
where
  atom e = (case e of
    (VarE x a)  $\Rightarrow$  True
  | -  $\Rightarrow$  False)

```

Lemmas for extend function

```

lemma extend-monotone:  $x \notin \text{set } xs \implies E\ x = \text{extend } E\ xs\ vs\ x$ 
apply (induct xs vs rule: list-induct2')
apply (simp add: extend-def)
apply (simp add: extend-def)
apply (simp add: extend-def)
apply (subgoal-tac  $x \notin \text{set } xs, \text{simp}$ )
apply (subgoal-tac extend E (xa # xs) (y # ys) = (extend E xs ys)(xa  $\mapsto$  y))
apply simp
apply (simp add: extend-def)
by simp

```

```

lemma list-induct3:
  [| P |] 0;
  !!x xs. P (x # xs) 0;
  !!i. P [|] (Suc i);
  !!x xs i. P xs i  $\implies$  P (x # xs) (Suc i) [|]
 $\implies$  P xs i
by (induct xs arbitrary: i) (case-tac x, auto)+

```

```

lemma extend-monotone-i [rule-format]:
  i < length alts  $\longrightarrow$ 
  length alts > 0  $\longrightarrow$ 
   $x \notin \text{set } (\text{snd } (\text{extractP } (\text{fst } (\text{alts } !\ i)))) \longrightarrow$ 
   $E\ x = \text{extend } E\ (\text{snd } (\text{extractP } (\text{fst } (\text{alts } !\ i))))\ vs\ x$ 
apply (induct alts i rule: list-induct3, simp-all)
apply (rule impI)
apply (erule extend-monotone)
apply (rule impI, simp)
by (case-tac xs = [|], simp-all)

```

```

lemma extend-prop1:
  [| z  $\in \text{dom } (\text{extend } E\ xs\ vs)$ ; z  $\notin \text{set } xs$ ; length xs = length vs |]  $\implies z \in \text{dom } E$ 
apply (simp add: extend-def)
apply (erule disjE)
apply (simp add: dom-def)
by simp

```

end

## 5 Primitive operators for SVM and JVM

```
theory BinOP
imports Main
begin
```

Primitive operators

```
datatype PrimOp = Add | Subtract | Times | Divide | LessThan | LessEqual
                | Equal | GreaterThan | GreaterEqual | NotEqual
```

end

## 6 State of the SVM

```
theory SVMState
imports SafeHeap ../JVMSAFE/BinOP
begin
```

### 6.1 Sizes Table

This gives statically inferred information about the maximum number of heap cells, of heap regions, and of stack words needed by the compiled program.

```
types ncell = nat
      sizeRegions = nat
      sizeStackS = nat
```

```
types SizesTable = ncell  $\times$  sizeRegions  $\times$  sizeStackS
```

### 6.2 Stack

```
types
  CodeLabel = nat
  Continuation = Region  $\times$  CodeLabel
```

```
datatype StackObject = Val Val | Reg Region | Cont Continuation
```

The SVM stack may contain normal form values, region arguments for functions or constructors, and continuations. A continuation  $(k_0, p)$  contains a jump  $p$  to a code sequence and an adjustment  $k_0$  for the heap watermark  $k_0$  of the SVM state.

```
types
```

*Stack* = *StackObject list*  
*StackOffset* = *nat*

### 6.3 Code Store and SafeImp program

— Items are the components of environments and closures

**datatype** *Item* = *ItemConst Val*  
                   | *ItemVar StackOffset*  
                   | *ItemRegSelf*

— The SVM instruction repertory

**datatype** *SafeInstr* = *DECREGION*  
                   | *POPCONT*  
                   | *PUSHCONT CodeLabel*  
                   | *COPY*  
                   | *REUSE*  
                   | *CALL CodeLabel*  
                   | *PRIMOP PrimOp*  
                   | *MATCH StackOffset (CodeLabel list)*  
                   | *MATCHD StackOffset (CodeLabel list)*  
                   | *MATCHN StackOffset nat nat (CodeLabel list)*  
                   | *BUILDENV (Item list)*  
                   | *BUILDCLS Constructor (Item list) Item*  
                   | *SLIDE nat nat*

**fun** *pushcont* :: *SafeInstr* => *bool*  
**where**  
   *pushcont (PUSHCONT p)* = *True*  
   | *pushcont -* = *False*

**fun** *popcont* :: *SafeInstr* => *bool*  
**where**  
   *popcont POPCONT* = *True*  
   | *popcont -* = *False*

A Safe program, when translated into SafeImp, produces four components (1) a map from labels to pairs consisting of a code sequence and a function name. It is given as a list in order to be able to ‘traverse’ the map; (2) a map from function names to pairs consisting of a label and a list of labels. The first points to the starting sequence of the function and the second collects, for each function body, the code labels corresponding to continuations. The map is also given as a list; (3) the code label of the main expression; and (4) a constructor table collecting the properties of all the constructors.

**types**  
*CodeSequence* = *SafeInstr list*

$SVMCode = (CodeLabel \times CodeSequence \times FunName) \text{ list}$   
 $ContinuationMap = (FunName \times CodeLabel \times CodeLabel \text{ list}) \text{ list}$   
 $CodeStore = SVMCode \times ContinuationMap$   
 $SafeImpProg = CodeStore \times CodeLabel \times ConstructorTableType \times SizesTable$

## 6.4 Runtime State

### types

$PC = CodeLabel \times nat$   
 $SVMState = Heap \times Region \times PC \times Stack$

### consts

$incrPC :: PC \Rightarrow PC$

### primrec

$incrPC \ (l, i) = (l, i+1)$

This is the correspondence between primitive operators in CoreSafe and SafeImp.

### constdefs

$primops :: string \rightarrow PrimOp$

$primops \equiv \text{map-of } [("+", Add),$   
 $\quad ("-", Subtract),$   
 $\quad ("*", Times),$   
 $\quad ("% ", Divide),$   
 $\quad ("<", LessThan),$   
 $\quad ("<=", LessEqual),$   
 $\quad ("==", Equal),$   
 $\quad (">", GreaterThan),$   
 $\quad (">=", GreaterEqual)$   
 $\quad ]$

— Define primitive operations

### consts

$execOp :: [PrimOp, Val, Val] \Rightarrow Val$

### primrec

$execOp \ Equal \ b1 \ b2 = BoolT \ (the-IntT(b1) = the-IntT(b2))$   
 $execOp \ NotEqual \ b1 \ b2 = BoolT \ (the-IntT(b1) \neq the-IntT(b2))$   
 $execOp \ GreaterEqual \ b1 \ b2 = BoolT \ (the-IntT(b1) \geq the-IntT(b2))$   
 $execOp \ GreaterThan \ b1 \ b2 = BoolT \ (the-IntT(b1) > the-IntT(b2))$   
 $execOp \ LessThan \ b1 \ b2 = BoolT \ (the-IntT(b1) < the-IntT(b2))$   
 $execOp \ LessEqual \ b1 \ b2 = BoolT \ (the-IntT(b1) \leq the-IntT(b2))$   
  
 $execOp \ Add \ b1 \ b2 = IntT \ (the-IntT(b1) + the-IntT(b2))$   
 $execOp \ Subtract \ b1 \ b2 = IntT \ (the-IntT(b1) - the-IntT(b2))$   
 $execOp \ Times \ b1 \ b2 = IntT \ (the-IntT(b1) * the-IntT(b2))$

*execOp Divide b1 b2 = IntT (the-IntT(b1) div the-IntT(b2))*

**end**

## 7 Resource-Aware Operational semantics of Safe expressions

**theory** *SafeRASemantics*

**imports** *SafeExpr ../SafeImp/SVMState*

**begin**

**types**

*Delta* = (*Region*  $\rightarrow$  *int*)

*MinimumFreshCells* = *int*

*MinimumWords* = *int*

*Resources* = *Delta*  $\times$  *MinimumFreshCells*  $\times$  *MinimumWords*

**constdefs** *sizeVal* :: [*HeapMap*, *Val*]  $\Rightarrow$  *int*

*sizeVal* *h* *v*  $\equiv$  (case *v* of (*Loc* *p*)  $\Rightarrow$  *int* *p*  
| -  $\Rightarrow$  0)

**constdefs** *size* :: [*HeapMap*, *Location*]  $\Rightarrow$  *int* **where**

*size* *h* *p*  $\equiv$  (case *h* *p* of  
Some (*j*, *C*, *vs*)  $\Rightarrow$  (let *rp* = *getRecursiveValuesCell* (*C*, *vs*)  
in 1 + ( $\sum_{i \in rp} \text{sizeVal } h (vs!i)$ )))

**constdefs** *balanceCells* :: *Delta*  $\Rightarrow$  *int* ( $\parallel$  -  $\parallel$  [71] 70)

*balanceCells*  $\delta \equiv (\sum_{n \in \text{ran } \delta} \delta. n)$

**constdefs** *addDelta* :: *Delta*  $\Rightarrow$  *Delta*  $\Rightarrow$  *Delta* (**infix**  $\oplus$  110)

*addDelta*  $\delta 1$   $\delta 2 \equiv$  (%*x*. (if *x*  $\in$  *dom*  $\delta 1 \cap$  *dom*  $\delta 2$   
then (case  $\delta 1$  *x* of (*Some* *y*)  $\Rightarrow$   
case  $\delta 2$  *x* of (*Some* *z*)  $\Rightarrow$  *Some* (*y* + *z*))  
else if *x*  $\in$  *dom*  $\delta 1 -$  *dom*  $\delta 2$   
then  $\delta 1$  *x*  
else if *x*  $\in$  *dom*  $\delta 2 -$  *dom*  $\delta 1$   
then  $\delta 2$  *x*  
else *None*))

**constdefs** *emptyDelta* :: *nat*  $\Rightarrow$  *nat*  $\rightarrow$  *int* ( $\llbracket$  -  $\rrbracket$  [71] 70)

$\llbracket$   $\rrbracket_k \equiv$  (%*i*. if *i*  $\in$  {0..*k*}  
then *Some* 0  
else *None*)

**consts** *def-copy* :: *nat*  $\Rightarrow$  *Heap*  $\Rightarrow$  *bool*

**inductive**

*SafeRASem* :: [*Environment* , *HeapMap* , *nat* , *nat* , *unit Exp* , *HeapMap* , *nat* , *Val* ,  
*Resources* ]  $\Rightarrow$  *bool*  
 ( -  $\vdash$  - , - , - , -  $\Downarrow$  - , - , - , - [71,71,71,71,71,71,71] 70)

**where**

*litInt* : *E*  $\vdash$  *h* , *k* , *td* , (*ConstE* (*LitN* *i*) *a*)  $\Downarrow$  *h* , *k* , *IntT* *i* , ( $\llbracket_k, 0, 1$ )

| *litBool*: *E*  $\vdash$  *h* , *k* , *td* , (*ConstE* (*LitB* *b*) *a*)  $\Downarrow$  *h* , *k* , *BoolT* *b* , ( $\llbracket_k, 0, 1$ )

| *var1* : *E1* *x* = *Some* (*Val.Loc* *p*)  
 $\Rightarrow$  (*E1*, *E2*)  $\vdash$  *h* , *k* , *td* , (*VarE* *x* *a*)  $\Downarrow$  *h* , *k* , *Val.Loc* *p* , ( $\llbracket_k, 0, 1$ )

| *var2* :  $\llbracket E1 \text{ } x = \text{Some } (Val.Loc \text{ } p); E2 \text{ } r = \text{Some } j; j \leq k;$   
*copy* (*h*, *k*) *p* *j* = ((*h'*, *k*), *p'*); *def-copy* *p* (*h*, *k*);  
*m* = *size* *h* *p*  $\rrbracket$   
 $\Rightarrow$  (*E1*, *E2*)  $\vdash$  *h* , *k* , *td* , (*x* @ *r* *a*)  $\Downarrow$  *h'* , *k* , *Val.Loc* *p'* , ( $\llbracket j \mapsto m, m, 2$ )

| *var3* :  $\llbracket E1 \text{ } x = \text{Some } (Val.Loc \text{ } p); h \text{ } p = \text{Some } c; SafeHeap.fresh \text{ } q \text{ } h \rrbracket$   
 $\Rightarrow$  (*E1*, *E2*)  $\vdash$  *h*, *k*, *td*, (*ReuseE* *x* *a*)  $\Downarrow$  ((*h*(*p* := *None*))(*q*  $\mapsto$  *c*)), *k*, *Val.Loc* *q*, ( $\llbracket_k, 0, 1$ )

| *let1* :  $\llbracket \forall C \text{ as } r \text{ } a'. e1 \neq ConstrE \text{ } C \text{ as } r \text{ } a'; x1 \notin dom \text{ } E1;$   
(*E1*, *E2*)  $\vdash$  *h*, *k*, 0, *e1*  $\Downarrow$  *h'*, *k*, *v1*, ( $\delta 1, m1, s1$ );  
(*E1* (*x1*  $\mapsto$  *v1*), *E2*)  $\vdash$  *h'*, *k*, (*td* + 1), *e2*  $\Downarrow$  *h''*, *k*, *v2*, ( $\delta 2, m2, s2$ )  $\rrbracket$   
 $\Rightarrow$  (*E1*, *E2*)  $\vdash$  *h*, *k* , *td* , *Let* *x1* = *e1* *In* *e2* *a*  $\Downarrow$  *h''*, *k*, *v2*,  
( $\delta 1 \oplus \delta 2, \max m1 \text{ } (m2 + \|\delta 1\|), \max (s1 + 2) \text{ } (s2 + 1)$ )

| *let2* :  $\llbracket E2 \text{ } r = \text{Some } j; j \leq k; fresh \text{ } p \text{ } h; x1 \notin dom \text{ } E1; r \neq self;$   
(*E1* (*x1*  $\mapsto$  *Val.Loc* *p*), *E2*)  $\vdash$   
*h*(*p*  $\mapsto$  (*j*, (*C*, *map* (*atom2val* *E1*) *as*))), *k*, (*td* + 1), *e2*  $\Downarrow$  *h'*, *k*, *v2*, ( $\delta, m, s$ )  $\rrbracket$   
 $\Rightarrow$  (*E1*, *E2*)  $\vdash$  *h*, *k* , *td* , *Let* *x1* = *ConstrE* *C* *as* *r* *a'* *In* *e2* *a*  $\Downarrow$  *h'*, *k*, *v2*,  
( $\delta \oplus (empty(j \mapsto 1)), m + 1, s + 1$ )

| *case1*:  $\llbracket i < length \text{ } alts;$   
*E1* *x* = *Some* (*Val.Loc* *p*);  
*h* *p* = *Some* (*j*, *C*, *vs*) ;  
*alts*! *i* = (*pati*, *ei*);  
*pati* = *ConstrP* *C* *ps* *ms*;  
*xs* = (*snd* (*extractP* (*fst* (*alts* ! *i*)))));  
*E1'* = *extend* *E1* *xs* *vs*;  
*def-extend* *E1* *xs* *vs*;  
*nr* = *int* (*length* *vs*);  
(*E1'*, *E2*)  $\vdash$  *h*, *k* , *nat* ((*int* *td*) + *nr*), *ei*  $\Downarrow$  *h'*, *k*, *v*, ( $\delta, m, s$ )  $\rrbracket$   
 $\Rightarrow$  (*E1*, *E2*)  $\vdash$  *h*, *k* , *td* , *Case* (*VarE* *x* *a*) *Of* *alts* *a'*  $\Downarrow$  *h'* , *k*, *v*, ( $\delta, m$ , (*s* + *nr*))



| *case1-1*:  $\llbracket i < \text{length } \text{alts};$   
 $E1 \ x = \text{Some } (\text{IntT } n);$   
 $\text{alts}!i = (\text{pati}, ei);$   
 $\text{pati} = \text{ConstP } (\text{LitN } n);$   
 $(E1, E2) \vdash h, k, td, ei \Downarrow h', k, v, (\delta, m, s) \rrbracket$   
 $\implies (E1, E2) \vdash h, k, td, \text{Case } (\text{VarE } x \ a) \ \text{Of } \text{alts } a' \Downarrow h', k, v, (\delta, m, s)$

| *case1-2*:  $\llbracket i < \text{length } \text{alts};$   
 $E1 \ x = \text{Some } (\text{BoolT } b);$   
 $\text{alts}!i = (\text{pati}, ei);$   
 $\text{pati} = \text{ConstP } (\text{LitB } b);$   
 $(E1, E2) \vdash h, k, td, ei \Downarrow h', k, v, (\delta, m, s) \rrbracket$   
 $\implies (E1, E2) \vdash h, k, td, \text{Case } (\text{VarE } x \ a) \ \text{Of } \text{alts } a' \Downarrow h', k, v, (\delta, m, s)$

| *case2*:  $\llbracket i < \text{length } \text{alts};$   
 $E1 \ x = \text{Some } (\text{Val.Loc } p);$   
 $h \ p = \text{Some } (j, C, vs);$   
 $\text{alts}!i = (\text{pati}, ei);$   
 $\text{pati} = \text{ConstrP } C \ ps \ ms;$   
 $xs = (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! \ i))));$   
 $E1' = \text{extend } E1 \ xs \ vs;$   
 $\text{def-extend } E1 \ xs \ vs;$   
 $nr = \text{int } (\text{length } vs);$   
 $j \leq k;$   
 $(E1', E2) \vdash h(p := \text{None}), k, \text{nat } ((\text{int } td) + nr), ei \Downarrow h', k, v, (\delta, m, s) \rrbracket$   
 $\implies (E1, E2) \vdash h, k, td, \text{CaseD } (\text{VarE } x \ a) \ \text{Of } \text{alts } a' \Downarrow h', k, v,$   
 $(\delta \oplus (\text{empty}(j \mapsto -1)), \max 0 \ (m - 1), s + nr)$

| *app-primops*:  $\llbracket \text{primops } f = \text{Some } \text{oper};$   
 $v1 = \text{atom2val } E1 \ a1;$   
 $v2 = \text{atom2val } E1 \ a2;$   
 $v = \text{execOp } \text{oper } v1 \ v2 \rrbracket$   
 $\implies (E1, E2) \vdash h, k, td, \text{AppE } f \ [a1, a2] \ [] \ a \Downarrow h, k, v, ([k, 0, 2])$

| *app*:  $\llbracket \Sigma f \ f = \text{Some } (xs, rs, e); \text{primops } f = \text{None};$   
 $\text{distinct } xs; \text{distinct } rs; \text{dom } E1 \cap \text{set } xs = \{\};$   
 $\text{length } xs = \text{length } as; \text{length } rs = \text{length } rr;$   
 $E1' = \text{map-of } (\text{zip } xs \ (\text{map } (\text{atom2val } E1) \ as));$   
 $n = \text{length } xs;$   
 $l = \text{length } rs;$   
 $E2' = (\text{map-of } (\text{zip } rs \ (\text{map } (\text{the} \circ E2) \ rr))) \ (\text{self} \mapsto \text{Suc } k);$   
 $(E1', E2') \vdash h, (\text{Suc } k), (n+l), e \Downarrow h', (\text{Suc } k), v, (\delta, m, s);$   
 $h'' = h' \upharpoonright \{p. p \in \text{dom } h' \ \& \ \text{fst } (\text{the } (h' \ p)) \leq k\} \rrbracket$   
 $\implies (E1, E2) \vdash h, k, td, \text{AppE } f \ as \ rr \ a \Downarrow h'', k, v,$   
 $(\delta(k+1 := \text{None}), m, \max((\text{int } n) + (\text{int } l))(s + (\text{int } n) + (\text{int } l) - \text{int } td))$

**end**

## 8 Depth-Aware Operational semantics of Safe expressions

```

theory SafeDepthSemantics
imports SafeRASemantics ../SafeImp/SVMState
begin

```

**inductive**

*SafeDepthSem* :: [*Environment*, *HeapMap*, *nat*, *unit Exp*, *string*, *HeapMap*, *nat*,

*Val*, *nat*]  $\Rightarrow$  *bool*

( $- \vdash -$ ,  $-$ ,  $-$   $\Downarrow$   $-$ ,  $-$ ,  $-$ ,  $-$  [*71*,*71*,*71*,*71*,*71*,*71*,*71*,*71*,*71*] *70*)

**where**

```

  litInt : E  $\vdash$  h, k, (ConstE (LitN i) a)  $\Downarrow$  f h, k, IntT i, 0

| litBool: E  $\vdash$  h, k, (ConstE (LitB b) a)  $\Downarrow$  f h, k, BoolT b, 0

| var1 : E1 x = Some (Val.Loc p)
         $\implies$  (E1,E2)  $\vdash$  h, k, (VarE x a)  $\Downarrow$  f h, k, Val.Loc p, 0

| var2 :  $\llbracket E1 x = \text{Some (Val.Loc p)}; E2 r = \text{Some } j; j \leq k;$ 
         $\text{copy } (h,k) p j = ((h',k),p'); \text{def-copy } p (h,k) \rrbracket$ 
         $\implies$  (E1,E2)  $\vdash$  h, k, (x @ r a)  $\Downarrow$  f h', k, Val.Loc p', 0

| var3 :  $\llbracket E1 x = \text{Some (Val.Loc p)}; h p = \text{Some } c; \text{SafeHeap.fresh } q h \rrbracket$ 
         $\implies$  (E1,E2)  $\vdash$  h,k,(ReuseE x a)  $\Downarrow$  f
        ((h(p:=None))(q  $\mapsto$  c)),k,Val.Loc q,0

| let1 :  $\llbracket \forall C \text{ as } r a'. e1 \neq \text{ConstrE } C \text{ as } r a'; x1 \notin \text{dom } E1;$ 
        (E1,E2)  $\vdash$  h, k, e1  $\Downarrow$  f h', k, v1, n1  $\wedge$ 
        (E1(x1  $\mapsto$  v1),E2)  $\vdash$  h', k, e2  $\Downarrow$  f h'', k, v2, n2  $\rrbracket$ 
         $\implies$  (E1,E2)  $\vdash$  h, k, Let x1 = e1 In e2 a  $\Downarrow$  f h'', k, v2, max n1 n2

| let2 :  $\llbracket E2 r = \text{Some } j; j \leq k; \text{fresh } p h; x1 \notin \text{dom } E1; r \neq \text{self};$ 
        (E1(x1  $\mapsto$  Val.Loc p),E2)  $\vdash$ 
        h(p  $\mapsto$  (j,(C,map (atom2val E1) as))), k, e2  $\Downarrow$  f h', k, v2, n  $\rrbracket$ 
         $\implies$  (E1,E2)  $\vdash$  h,k,Let x1 = ConstrE C as r a' In e2 a  $\Downarrow$  f h',k,v2,n

| case1:  $\llbracket i < \text{length } \text{alts};$ 
        E1 x = Some (Val.Loc p);
        h p = Some (j,C,vs);
        alts!i = (pati, ei);
        pati = ConstrP C ps ms;
        xs = (snd (extractP (fst (alts ! i))));
        E1' = extend E1 xs vs;
        def-extend E1 xs vs;
```

$$\begin{aligned}
& nr = \text{int } (\text{length } vs); \\
& (E1', E2) \vdash h, k, ei \Downarrow f h', k, v, n \\
\implies & (E1, E2) \vdash h, k, \text{Case } (\text{VarE } x \ a) \ \text{Of alts } a' \Downarrow f h', k, v, n
\end{aligned}$$

$$\begin{aligned}
| \text{ case1-1: } & \llbracket i < \text{length } alts; \\
& E1 \ x = \text{Some } (\text{IntT } n); \\
& alts!i = (pati, ei); \\
& pati = \text{ConstP } (\text{LitN } n); \\
& (E1, E2) \vdash h, k, ei \Downarrow f h', k, v, nf \\
\implies & (E1, E2) \vdash h, k, \text{Case } (\text{VarE } x \ a) \ \text{Of alts } a' \Downarrow f h', k, v, nf
\end{aligned}$$

$$\begin{aligned}
| \text{ case1-2: } & \llbracket i < \text{length } alts; \\
& E1 \ x = \text{Some } (\text{BoolT } b); \\
& alts!i = (pati, ei); \\
& pati = \text{ConstP } (\text{LitB } b); \\
& (E1, E2) \vdash h, k, ei \Downarrow f h', k, v, n \\
\implies & (E1, E2) \vdash h, k, \text{Case } (\text{VarE } x \ a) \ \text{Of alts } a' \Downarrow f h', k, v, n
\end{aligned}$$

$$\begin{aligned}
| \text{ case2: } & \llbracket i < \text{length } alts; \\
& E1 \ x = \text{Some } (\text{Val.Loc } p); \\
& h \ p = \text{Some } (j, C, vs); \\
& alts!i = (pati, ei); \\
& pati = \text{ConstrP } C \ ps \ ms; \\
& xs = (\text{snd } (\text{extractP } (\text{fst } (alts \ ! \ i)))); \\
& E1' = \text{extend } E1 \ xs \ vs; \\
& \text{def-extend } E1 \ xs \ vs; \\
& nr = \text{int } (\text{length } vs); \\
& j \leq k; \\
& (E1', E2) \vdash h(p := \text{None}), k, ei \Downarrow f h', k, v, n \\
\implies & (E1, E2) \vdash h, k, \text{CaseD } (\text{VarE } x \ a) \ \text{Of alts } a' \Downarrow f h', k, v, n
\end{aligned}$$

$$\begin{aligned}
| \text{ app-primops: } & \llbracket \text{primops } g = \text{Some } \text{oper}; \\
& v1 = \text{atom2val } E1 \ a1; \\
& v2 = \text{atom2val } E1 \ a2; \\
& v = \text{execOp } \text{oper } v1 \ v2 \\
\implies & (E1, E2) \vdash h, k, \text{AppE } g \ [a1, a2] \ [] \ a \Downarrow f h, k, v, 1
\end{aligned}$$

$$\begin{aligned}
| \text{ app: } & \llbracket \Sigma f \ f = \text{Some } (xs, rs, e); \text{primops } f = \text{None}; \\
& \text{distinct } xs; \text{distinct } rs; \text{dom } E1 \cap \text{set } xs = \{\}; \\
& \text{length } xs = \text{length } as; \text{length } rs = \text{length } rr; \\
& E1' = \text{map-of } (\text{zip } xs \ (\text{map } (\text{atom2val } E1) \ as)); \\
& n = \text{length } xs; \\
& l = \text{length } rs; \\
& E2' = (\text{map-of } (\text{zip } rs \ (\text{map } (\text{the} \circ E2) \ rr))) \ (\text{self} \mapsto \text{Suc } k); \\
& (E1', E2') \vdash h, (\text{Suc } k), e \Downarrow f h', (\text{Suc } k), v, nf; \\
& h'' = h' \restriction \{p. p \in \text{dom } h' \ \& \ \text{fst } (\text{the } (h' \ p)) \leq k\} \\
\implies & (E1, E2) \vdash h, k, (\text{AppE } f \ as \ rr \ a) \Downarrow f h'', k, v, (nf+1)
\end{aligned}$$

$$| \text{ app2: } \llbracket \Sigma f \ g = \text{Some } (xs, rs, e); \text{primops } g = \text{None}; f \neq g;$$

$$\begin{aligned}
& \text{distinct } xs; \text{ distinct } rs; \text{ dom } E1 \cap \text{set } xs = \{\}; \\
& \text{length } xs = \text{length } as; \text{ length } rs = \text{length } rr; \\
& E1' = \text{map-of } (\text{zip } xs \ (\text{map } (\text{atom2val } E1) \ as)); \\
& n = \text{length } xs; \\
& l = \text{length } rs; \\
& E2' = (\text{map-of } (\text{zip } rs \ (\text{map } (\text{the} \circ E2) \ rr))) \ (\text{self} \mapsto \text{Suc } k); \\
& (E1', E2') \vdash h, (\text{Suc } k), e \Downarrow f h', (\text{Suc } k), v, nf; \\
& h'' = h' \mid \{p. p \in \text{dom } h' \ \& \ \text{fst } (\text{the } (h' \ p)) \leq k\} \\
& \implies (E1, E2) \vdash h, k, \text{AppE } g \ as \ rr \ a \Downarrow f h'', k, v, nf
\end{aligned}$$

**lemma** *eqSemRADepth*[*rule-format*]:  $(E1, E2) \vdash h, k, td, e \Downarrow h', k, v, r \longrightarrow$   
 $(\exists n. (E1, E2) \vdash h, k, e \Downarrow f h', k, v, n)$

**apply** (*rule impI*)  
**apply** (*erule SafeRASem.induct*)

**apply** (*rule-tac x=0 in exI*)  
**apply** (*rule SafeDepthSem.litInt*)

**apply** (*rule-tac x=0 in exI*)  
**apply** (*rule SafeDepthSem.litBool*)

**apply** (*rule-tac x=0 in exI*)  
**apply** (*rule SafeDepthSem.var1, assumption*)

**apply** (*rule-tac x=0 in exI*)  
**apply** (*rule SafeDepthSem.var2, simp, simp, assumption, assumption, assumption*)

**apply** (*rule-tac x=0 in exI*)  
**apply** (*rule SafeDepthSem.var3, assumption+*)

**apply** (*erule exE*) +  
**apply** (*rename-tac n1 n2*)  
**apply** (*rule-tac x=max n1 n2 in exI*)  
**apply** (*rule SafeDepthSem.let1*)  
**apply** (*assumption+*)  
**apply** (*rule conjI, simp, simp*)

**apply** (*erule exE*) +  
**apply** (*rename-tac n2*)  
**apply** (*rule-tac x=n2 in exI*)

```

apply (rule SafeDepthSem.let2)
apply (assumption+)

```

```

apply (elim exE)
apply (rename-tac ni)
apply (rule-tac x=ni in exI)
apply (rule SafeDepthSem.case1)
apply (assumption+)

```

```

apply (elim exE)
apply (rename-tac ni)
apply (rule-tac x=ni in exI)
apply (rule SafeDepthSem.case1-1)
apply (assumption+)

```

```

apply (elim exE)
apply (rename-tac ni)
apply (rule-tac x=ni in exI)
apply (rule SafeDepthSem.case1-2)
apply (assumption+)

```

```

apply (elim exE)
apply (rename-tac ni)
apply (rule-tac x=ni in exI)
apply (rule SafeDepthSem.case2)
apply (assumption+)

```

```

apply (rule-tac x=1 in exI)
apply (rule SafeDepthSem.app-primops)
apply assumption+

```

```

apply (elim exE)
apply (rename-tac nf)
apply (case-tac fa=f)

```

```

apply (rule-tac x=nf+1 in exI)
apply (rule-tac s=f and t=fa in subst)
apply simp
apply (rule-tac s=h' | ' {p ∈ dom h'. fst (the (h' p)) ≤ k} and t=h'' in subst)
apply simp
apply clarify
apply (rule-tac h'=h' in SafeDepthSem.app)

```

```

apply assumption +
apply simp
apply simp
apply simp
apply simp
apply assumption
apply simp

```

```

apply (rule-tac  $x = nf$  in exI)
apply (rule-tac  $s = h' \mid \{p \in dom\ h'.\ fst\ (the\ (h'\ p)) \leq k\}$  and  $t = h''$  in subst)
apply simp
apply (rule SafeDepthSem.app2)
apply assumption +
apply simp
apply assumption +
apply simp
done

```

```

lemma eqSemDepthRA[rule-format]:  $(E1, E2) \vdash h, k, e \Downarrow f\ h', k, v, n \longrightarrow$ 
 $(\forall\ td. \exists\ r. (E1, E2) \vdash h, k, td, e \Downarrow h', k, v, r)$ 

```

```

apply (rule impI)
apply (erule SafeDepthSem.induct)

```

```

apply (rule allI)
apply (rule-tac  $x = ([ ]_k, 0, 1)$  in exI)
apply (rule SafeRASem.litInt)

```

```

apply (rule allI)
apply (rule-tac  $x = ([ ]_k, 0, 1)$  in exI)
apply (rule SafeRASem.litBool)

```

```

apply (rule allI)
apply (rule-tac  $x = ([ ]_k, 0, 1)$  in exI)
apply (rule SafeRASem.var1, assumption)

```

```

apply (rule allI)
apply (rule-tac  $x = ([j \mapsto size\ h\ p], size\ h\ p, 2)$  in exI)
apply (rule SafeRASem.var2, assumption +, simp)

```

```

apply (rule allI)
apply (rule-tac  $x = ([ ]_k, 0, 1)$  in exI)

```

**apply** (*rule SafeRASem.var3*) **apply** *assumption+*

**apply** (*rule allI*)  
**apply** (*elim conjE*)  
**apply** (*erule-tac x=0 in allE*)  
**apply** (*erule-tac x=td+1 in allE*)  
**apply** (*elim exE*)  
**apply** (*rename-tac r1 r2*)  
**apply** (*case-tac r1*)  
**apply** (*case-tac r2*)  
**apply** (*rename-tac  $\delta 1$  rest1  $\delta 2$  rest2*)  
**apply** (*case-tac rest1*)  
**apply** (*case-tac rest2*)  
**apply** (*rename-tac m1 ss1 m2 ss2*)  
**apply** (*rule-tac  $x=(\delta 1 \oplus \delta 2, \max m1 (m2 + \|\delta 1\|), \max (ss1+2) (ss2+1))$  in exI*)  
**apply** (*rule SafeRASem.let1,assumption+,simp,simp*)

**apply** (*rule allI*)  
**apply** (*erule-tac x=td+1 in allE*)  
**apply** (*elim exE*)  
**apply** (*rename-tac r2*)  
**apply** (*case-tac r2*)  
**apply** (*rename-tac  $\delta 2$  rest2*)  
**apply** (*case-tac rest2*)  
**apply** (*rename-tac m2 s2*)  
**apply** (*rule-tac  $x=(\delta 2 \oplus (\text{empty}(j \mapsto 1)), m2+1, s2+1)$  in exI*)  
**apply** (*rule SafeRASem.let2,assumption+,clarify*)

**apply** (*rule allI*)  
**apply** (*erule-tac  $x=\text{nat} ((\text{int } td)+ nr)$  in allE*)  
**apply** (*elim exE*)  
**apply** (*rename-tac r*)  
**apply** (*case-tac r*)  
**apply** (*rename-tac  $\delta$  rest*)  
**apply** (*case-tac rest*)  
**apply** (*rename-tac m ss*)  
**apply** (*rule-tac  $x=(\delta, m, (ss+nr))$  in exI*)  
**apply** (*rule SafeRASem.case1,assumption+,clarify*)

**apply** (*rule allI*)  
**apply** (*erule-tac x=td in allE*)  
**apply** (*elim exE*)  
**apply** (*rename-tac r*)  
**apply** (*case-tac r*)  
**apply** (*rename-tac  $\delta$  rest*)

```

apply (case-tac rest)
apply (rename-tac m ss)
apply (rule-tac x=( $\delta, m, ss$ ) in exI)
apply (rule SafeRASem.case1-1, assumption+, clarify)

```

```

apply (rule allI)
apply (erule-tac x=td in allE)
apply (elim exE)
apply (rename-tac r)
apply (case-tac r)
apply (rename-tac  $\delta$  rest)
apply (case-tac rest)
apply (rename-tac m ss)
apply (rule-tac x=( $\delta, m, ss$ ) in exI)
apply (rule SafeRASem.case1-2, assumption+, clarify)

```

```

apply (rule allI)
apply (erule-tac x=nat ((int td)+ nr) in allE)
apply (elim exE)
apply (rename-tac r)
apply (case-tac r)
apply (rename-tac  $\delta$  rest)
apply (case-tac rest)
apply (rename-tac m ss)
apply (rule-tac x=( $\delta \oplus (\text{empty}(j) \mapsto -1)$ ), max 0 (m - 1), ss+nr) in exI)
apply (rule SafeRASem.case2, assumption+, clarify)

```

```

apply (rule allI)
apply (rule-tac x=( $\llbracket k, 0, 2 \rrbracket$ ) in exI)
apply (rule SafeRASem.app-primops, assumption+)

```

```

apply (rule allI)
apply (erule-tac x=length as+ length rs in allE)
apply (elim exE)
apply (rename-tac r)
apply (case-tac r)
apply (rename-tac  $\delta$  rest)
apply (case-tac rest)
apply (rename-tac m ss)
apply (rule-tac x=( $\delta(k+1 := \text{None}), m, \max((\text{int } n) + (\text{int } l))(\text{ss} + (\text{int } n) + (\text{int } l) -$ 
int td)) in exI)
apply (rule SafeRASem.app)
apply assumption+
apply simp
apply simp

```



```

apply (rule allI)
apply (erule-tac x=n+l in allE)
apply (elim exE)
apply (rename-tac r)
apply (case-tac r)
apply (rename-tac  $\delta$  rest)
apply (case-tac rest)
apply (rename-tac m ss)
apply (rule-tac x=( $\delta(k+1:=None), m, \max((int\ n)+(int\ l))(ss+(int\ n)+(int\ l) -$ 
 $int\ td)$ ) in exI)
apply (rule SafeRASem.app)
apply assumption+
apply simp
apply simp
done

constdefs
  SafeBoundSem :: [Environment, HeapMap, nat, unit Exp, string  $\times$  nat,
    HeapMap, nat, Val]  $\Rightarrow$  bool
  (-  $\vdash$  -, -, -,  $\Downarrow$  -, -, -, - [71,71,71,71,71,71,71] 70)

   $E \vdash h, k, e \Downarrow_{tup} h', k', v \equiv$ 
  (let (f,n) = tup in  $k=k' \wedge (\exists\ nf . E \vdash h, k, e \Downarrow_f h', k, v, nf$ 
     $\wedge nf \leq n)$ )

lemma eqSemRABound [rule-format]:
  ( $\forall\ td. \exists\ r. (E1, E2) \vdash h, k, td, e \Downarrow h', k, v, r$ )  $\equiv$ 
  ( $\exists\ n. (E1, E2) \vdash h, k, e \Downarrow (f, n) h', k, v$ )

apply (rule eq-reflection)
apply (rule iffI)

thm eqSemRADepth
apply (erule-tac x=td in allE)
apply (elim exE)
apply (simp add: SafeBoundSem-def)
apply (drule-tac ?f=f in eqSemRADepth)
apply (elim exE)
apply (rule-tac x=x in exI)
apply (rule-tac x=x in exI)
apply (rule conjI, assumption, simp)

apply (simp add: SafeBoundSem-def del:Product-Type.split-paired-Ex)
apply (elim exE)
apply (elim conjE)
apply (rule allI)
apply (drule-tac td=td in eqSemDepthRA)

```

```

apply (elim exE)
apply (rule-tac x=x in exI)
by assumption

```

```

lemma eqSemBoundRA [rule-format]:
   $\exists n. (E1, E2) \vdash h, k, e \Downarrow (f, n) \ h', k, v \equiv$ 
   $\exists r. (E1, E2) \vdash h, k, td, e \Downarrow h', k, v, r$ 
apply (rule eq-reflection)
apply (rule iffI)

```

```

apply (simp add: SafeBoundSem-def del:Product-Type.split-paired-Ex)
apply (elim exE)
apply (elim conjE)
apply (drule-tac td=td in eqSemDepthRA)
apply (elim exE)
apply (rule-tac x=x in exI)
apply assumption

```

```

apply (elim exE)
apply (simp add: SafeBoundSem-def)
apply (drule-tac ?f=f in eqSemRADepth)
apply (elim exE)
apply (rule-tac x=x in exI)
apply (rule-tac x=x in exI)
apply (rule conjI, assumption, simp)
done

```

```

lemma impSemBoundRA [rule-format]:
   $(E1, E2) \vdash h, k, e \Downarrow (f, n) \ h', k, v \implies$ 
   $\exists r. (E1, E2) \vdash h, k, td, e \Downarrow h', k, v, r$ 
apply (simp add: SafeBoundSem-def del:Product-Type.split-paired-Ex)
apply (elim exE, elim conjE)
apply (drule-tac td=td in eqSemDepthRA)
apply (elim exE)
apply (rule-tac x=x in exI)
by assumption

```

**end**

## 9 Closure Heap

```

theory ClosureHeap
imports SafeHeap ../CoreSafe/SafeRASemantics
begin

```

**inductive-set**

$\text{closureL} :: \text{Location} \Rightarrow \text{Heap} \Rightarrow \text{Location set}$   
**for**  $p :: \text{Location}$  **and**  $h :: \text{Heap}$   
**where**  
 $\text{closureL-basic}: p \in \text{closureL } p \ h$   
 $\mid \text{closureL-step}: \llbracket q \in \text{closureL } p \ h; d \in \text{descendants } q \ h \rrbracket \implies d \in \text{closureL } p \ h$

**constdefs**  $\text{closure} :: \text{Environment} \Rightarrow \text{string} \Rightarrow \text{Heap} \Rightarrow \text{Location set}$   
 $\text{closure } E \ x \ h \equiv (\text{case } ((\text{fst } E) \ x) \text{ of } \text{Some } (\text{Loc } p) \Rightarrow \text{closureL } p \ h$   
 $\mid - \Rightarrow \{\})$

**constdefs**  $\text{closureV} :: \text{Val} \Rightarrow \text{Heap} \Rightarrow \text{Location set}$   
 $\text{closureV } v \ h \equiv (\text{case } v \text{ of } \text{IntT } i \Rightarrow \{\}$   
 $\mid \text{BoolT } b \Rightarrow \{\}$   
 $\mid \text{Loc } p \Rightarrow \text{closureL } p \ h)$

**constdefs**  $\text{closureLS} :: \text{Environment} \Rightarrow \text{string set} \Rightarrow \text{Heap} \Rightarrow \text{Location set}$   
 $\text{closureLS } E \ L \ h \equiv (\bigcup_{x \in L. \text{closure } E \ x \ h)$

**constdefs**  $\text{identityClosure} :: \text{Environment} \Rightarrow \text{string} \Rightarrow \text{Heap} \Rightarrow \text{Heap} \Rightarrow \text{bool}$   
 $\text{identityClosure } E \ x \ h \ hh \equiv \text{closure } E \ x \ h = \text{closure } E \ x \ hh \wedge$   
 $(\forall p \in \text{closure } E \ x \ h. (\text{fst } h) \ p = (\text{fst } hh) \ p)$

**constdefs**  $\text{identityClosureL} :: \text{Location} \Rightarrow \text{Heap} \Rightarrow \text{Heap} \Rightarrow \text{bool}$   
 $\text{identityClosureL } q \ h \ hh \equiv \text{closureL } q \ h = \text{closureL } q \ hh \wedge$   
 $(\forall p \in \text{closureL } q \ h. (\text{fst } h) \ p = (\text{fst } hh) \ p)$

**defs**  $\text{def-copy}:$   
 $\text{def-copy } p \ h == (\forall \ q \in \text{closureL } p \ h. \ q \in \text{dom } (\text{fst } h))$

**constdefs**  $\text{scope} :: \text{Environment} \Rightarrow \text{Heap} \Rightarrow \text{Location set}$   
 $\text{scope } E \ h \equiv \text{closureLS } E \ (\text{dom } (\text{fst } E)) \ h$

**constdefs**  $\text{live} :: \text{Environment} \Rightarrow \text{string set} \Rightarrow \text{Heap} \Rightarrow \text{Location set}$   
 $\text{live } E \ L \ h \equiv \text{closureLS } E \ L \ h$

**constdefs**  $\text{closed} :: \text{Environment} \Rightarrow \text{string set} \Rightarrow \text{Heap} \Rightarrow \text{bool}$

$closed\ E\ L\ h \equiv (live\ E\ L\ h) \subseteq domHeap\ h$

**lemma** *closed-Empty* :  $closed\ E\ \{\}\ h$   
**by** (*simp add: closed-def add: live-def add: closureLS-def*)

**lemma** *closed-monotone*:  $closed\ E\ (L1 \cup (L2 - \{x1\}))\ h \implies closed\ E\ L1\ h$   
**by** (*simp add: closed-def add: live-def add: closureLS-def*)

**constdefs** *closed-f* ::  $Val \Rightarrow Heap \Rightarrow bool$   
 $closed-f\ v\ h \equiv (case\ v\ of\ IntT\ i \Rightarrow True$   
 $\quad | BoolT\ b \Rightarrow True$   
 $\quad | Loc\ p \Rightarrow closureL\ p\ h \subseteq domHeap\ h)$

**inductive-set**

$recReachL :: Location \Rightarrow Heap \Rightarrow Location\ set$   
**for**  $p :: Location$  **and**  $h :: Heap$   
**where**  
 $recReachL\ basic: p \in recReachL\ p\ h$   
 $| recReachL\ step: \llbracket q \in recReachL\ p\ h; d \in recDescendants\ q\ h \rrbracket \implies d \in recReachL\ p\ h$

**constdefs** *recReach* ::  $Environment \Rightarrow string \Rightarrow Heap \Rightarrow Location\ set$   
 $recReach\ E\ x\ h \equiv (case\ ((fst\ E)\ x)\ of\ Some\ (Loc\ p) \Rightarrow recReachL\ p\ h$   
 $\quad | - \Rightarrow \{\})$

**lemma** *transit-aux*:  $\llbracket r \in closureL\ d\ h \rrbracket \implies d \in descendants\ qa\ h \longrightarrow r \in closureL\ qa\ h$   
**apply** (*erule closureL.induct*)  
**apply** (*rule impI*)  
**apply** (*subgoal-tac qa \in closureL qa h*)  
**apply** (*erule closureL-step, assumption*)  
**apply** (*rule closureL-basic*)  
**apply** (*rule impI*)  
**apply** (*drule mp*)  
**apply** *assumption*  
**by** (*erule closureL-step, assumption*)

**lemma** *transit-aux2*:  $\llbracket r \in closureL\ d\ h; d \in descendants\ qa\ h \rrbracket \implies r \in closureL\ qa\ h$   
**apply** (*subgoal-tac \llbracket r \in closureL d h \rrbracket \implies d \in descendants qa h \longrightarrow r \in closureL qa h, simp*)  
**by** (*rule transit-aux, assumption*)

**lemma** *transit*:  $\llbracket p \in closureL\ q\ h \rrbracket \implies \forall r. r \in closureL\ p\ h$

$\longrightarrow r \in \text{closureL } q \ h$   
**apply** (*erule closureL.induct*)  
**apply** *simp*  
**apply** *clarsimp*  
**apply** (*erule allE, erule mp*)  
**apply** (*erule thin-rl*)  
**by** (*erule transit-aux2, assumption*)

**lemma** *closureL-transit*:  $\llbracket p \in \text{closureL } q \ h; r \in \text{closureL } p \ h \rrbracket \Longrightarrow r \in \text{closureL } q \ h$   
**by** (*frule transit, simp*)

**lemma** *closure-transit*:  $\llbracket p \in \text{closure } E \ x \ h; q \in \text{closureL } p \ h \rrbracket \Longrightarrow q \in \text{closure } E \ x \ h$   
**apply** (*simp add: closure-def*)  
**apply** (*case-tac fst E x*)  
**apply** *simp*  
**apply** *simp*  
**apply** (*case-tac a*)  
**apply** *simp-all*  
**apply** (*erule closureL-transit*)  
**by** *assumption*

**lemma** *closureL-monotone*:  $p \in \text{closureL } q \ (h1', h2') \Longrightarrow \text{closureL } p \ (h1', h2') \subseteq \text{closureL } q \ (h1', h2')$   
**apply** (*erule closureL.induct*)  
**apply** *simp*  
**apply** (*subgoal-tac closureL d (h1', h2')  $\subseteq$  closureL qa (h1', h2')*)  
**apply** *blast*  
**apply** (*rule subsetI, rule transit-aux2*)  
**apply** *simp*  
**by** *simp*

**lemma** *closure-recReach-monotone*:  
 $\llbracket p \in \text{closure } (E1(x1 \mapsto r), E2) \ y \ (h1', h2');$   
 $\text{closureL } p \ (h1', h2') \cap \text{recReach } (E1(x1 \mapsto r), E2) \ x \ (h1', h2') \neq \{\} \rrbracket$   
 $\Longrightarrow \text{closure } (E1(x1 \mapsto r), E2) \ y \ (h1', h2') \cap \text{recReach } (E1(x1 \mapsto r), E2) \ x \ (h1', h2') \neq \{\}$   
**apply** (*case-tac y  $\neq$  x1*)  
**apply** (*simp add: closure-def*)  
**apply** (*case-tac E1 y*)  
**apply** *simp*  
**apply** *simp*  
**apply** (*case-tac a, simp-all, clarsimp*)  
**apply** (*subgoal-tac p  $\in$  closureL nat (h1', h2')  $\Longrightarrow$  closureL p (h1', h2')  $\subseteq$  closureL nat (h1', h2')*)  
**apply** *blast*  
**apply** (*erule closureL-monotone*)

**apply** (*simp add: closure-def*)  
**apply** (*case-tac r, simp-all, clarsimp*)  
**apply** (*subgoal-tac p ∈ closureL nat (h1', h2') ⇒ closureL p (h1', h2') ⊆ closureL nat (h1', h2')*)  
**apply** *blast*  
**by** (*erule closureL-monotone*)

**lemma** *closure-monotone*:  $\llbracket y \neq x1; p \in \text{closure } (E1(x1 \mapsto r), E2) y (h1', h2') \rrbracket$   
 $\implies \text{closureL } p (h1', h2') \subseteq \text{closure } (E1(x1 \mapsto r), E2) y (h1', h2')$   
**apply** (*simp add: closure-def*)  
**apply** (*case-tac E1 y, simp-all*)  
**apply** (*case-tac a, simp-all*)  
**by** (*erule closureL-monotone*)

**lemma** *identityClosureL-monotone-h*:  
 $\llbracket x \in \text{closureL } q h \rrbracket \implies q \in \text{closureL } p h \longrightarrow \text{identityClosureL } p h hh \longrightarrow x \in \text{closureL } q hh$   
**apply** (*erule closureL.induct*)  
**apply** (*intro impI, rule closureL-basic*)  
**apply** (*intro impI*)  
**apply** *simp*  
**apply** (*subgoal-tac d ∈ descendants qa hh*)  
**apply** (*erule closureL-step, simp*)  
**apply** (*simp add: identityClosureL-def*)  
**apply** (*erule conjE*)  
**apply** (*erule-tac x=qa in ballE*)  
**apply** (*subgoal-tac qa ∈ closureL q h*)  
**apply** (*subgoal-tac d ∈ closureL q h*)  
**apply** (*simp add: descendants-def*)  
**apply** (*erule closureL-step, simp*)  
**apply** *simp*  
**apply** (*subgoal-tac q ∈ closureL p h*)  
**prefer** 2 **apply** *simp*  
**apply** (*subgoal-tac  $\llbracket qa \in \text{closureL } q h; q \in \text{closureL } p h \rrbracket \implies qa \in \text{closureL } p h, \text{simp}$* )  
**by** (*rule closureL-transit*)

**lemma** *identityClosureL-monotone-hh*:  
 $\llbracket x \in \text{closureL } q hh \rrbracket \implies q \in \text{closureL } p h \longrightarrow \text{identityClosureL } p h hh \longrightarrow x \in \text{closureL } q h$   
**apply** (*erule closureL.induct*)  
**apply** (*intro impI, rule closureL-basic*)  
**apply** (*intro impI*)  
**apply** *simp*  
**apply** (*subgoal-tac d ∈ descendants qa h*)  
**apply** (*erule closureL-step, simp*)  
**apply** (*simp add: identityClosureL-def*)  
**apply** (*erule conjE*)  
**apply** (*erule-tac x=qa in ballE*)

**apply** (*subgoal-tac*  $qa \in \text{closureL } q \text{ } hh$ )  
**apply** (*subgoal-tac*  $d \in \text{closureL } q \text{ } hh$ )  
**apply** (*simp add: descendants-def*)  
**apply** (*erule closureL-step, simp*)  
**apply** *simp*  
**apply** (*subgoal-tac*  $q \in \text{closureL } p \text{ } hh$ )  
**prefer** 2 **apply** *simp*  
**apply** (*subgoal-tac*  $\llbracket qa \in \text{closureL } q \text{ } hh; q \in \text{closureL } p \text{ } hh \rrbracket \implies qa \in \text{closureL } p \text{ } hh, \text{simp}$ )  
**by** (*rule closureL-transit*)

**lemma** *identityClosureL-monotone*:

$\llbracket \text{identityClosureL } p \text{ } h \text{ } hh; q \in \text{closureL } p \text{ } h \rrbracket \implies \text{identityClosureL } q \text{ } h \text{ } hh$   
**apply** (*simp (no-asm) add: identityClosureL-def*)  
**apply** (*rule conjI*)  
**apply** (*rule equalityI*)  
**apply** (*rule subsetI*)  
**apply** (*subgoal-tac*  $\llbracket x \in \text{closureL } q \text{ } h \rrbracket \implies q \in \text{closureL } p \text{ } h \longrightarrow \text{identityClosureL } p \text{ } h \text{ } hh \longrightarrow x \in \text{closureL } q \text{ } hh, \text{simp}$ )  
**apply** (*rule identityClosureL-monotone-h, simp*)  
**apply** (*rule subsetI*)  
**apply** (*subgoal-tac*  $\llbracket x \in \text{closureL } q \text{ } hh \rrbracket \implies q \in \text{closureL } p \text{ } h \longrightarrow \text{identityClosureL } p \text{ } h \text{ } hh \longrightarrow x \in \text{closureL } q \text{ } h, \text{simp}$ )  
**apply** (*rule identityClosureL-monotone-hh, simp*)  
**apply** (*simp add: identityClosureL-def*)  
**apply** (*rule ballI*)  
**apply** (*erule conjE*)  
**apply** (*subgoal-tac*  $q \in \text{closureL } p \text{ } h$ )  
**apply** (*frule-tac r=pa and h=h in closureL-transit*)  
**apply** (*rule closureL-basic*)  
**apply** (*erule-tac x=pa in ballE, clarsimp*)  
**apply** (*frule-tac r=pa and h=h and p=q in closureL-transit, clarsimp*)  
**apply** *simp*  
**by** *simp*

**lemma** *identityClosure-closureL-monotone*:

$\llbracket \text{identityClosure } E \text{ } x \text{ } h \text{ } hh; q \in \text{closure } E \text{ } x \text{ } h \rrbracket \implies \text{identityClosureL } q \text{ } h \text{ } hh$   
**apply** (*simp add: identityClosure-def*)  
**apply** (*simp add: closure-def*)  
**apply** (*case-tac fst E x, simp-all*)  
**apply** (*case-tac a, simp-all*)  
**apply** (*subgoal-tac*  $\text{closureL } nat \text{ } h = \text{closureL } nat \text{ } hh \wedge (\forall p \in \text{closureL } nat \text{ } h. \text{fst } h \text{ } p = \text{fst } hh \text{ } p) \longrightarrow \text{identityClosureL } nat \text{ } h \text{ } hh, \text{simp}$ )  
**apply** (*erule identityClosureL-monotone, simp*)  
**by** (*simp add: identityClosureL-def*)

**lemma** *recReachL-subseteq-closureL*:

$\text{recReachL } p \text{ } h \subseteq \text{closureL } p \text{ } h$

```

apply clarify
apply (erule recReachL.induct)
apply (simp add: closure-def)
apply (rule closureL-basic)
apply (simp add: closure-def)
apply (subgoal-tac recDescendants q h  $\subseteq$  descendants q h)
apply (erule closureL-step, blast)
apply (simp add: recDescendants-def add: descendants-def)
apply (case-tac fst h q, simp, simp)
apply (subgoal-tac getRecursiveValuesCell (snd a)  $\subseteq$  getNonBasicValuesCell (snd
a), simp)
apply (simp add: getRecursiveValuesCell-def add: getNonBasicValuesCell-def)
apply (simp add: isRecursive-def add: isNonBasicValue-def)
apply auto
done

```

```

lemma recReach-subseteq-closure:
  recReach E z h  $\subseteq$  closure E z h
apply (simp add: recReach-def)
apply (case-tac fst E z, simp-all)
apply (case-tac a, simp-all)
apply (simp add: closure-def)
by (rule recReachL-subseteq-closureL)

```

```

lemma identityClosure-equals-recReach-hh:
   $p \in \text{recReachL } q \text{ } hh \implies \text{closureL } q \text{ } h = \text{closureL } q \text{ } hh \longrightarrow (\forall p \in \text{closureL } q \text{ } h. (\text{fst } h) \text{ } p = (\text{fst } hh) \text{ } p) \longrightarrow p \in \text{recReachL } q \text{ } h$ 
apply (erule recReachL.induct)
apply (intro impI)
apply (rule recReachL-basic)
apply clarsimp
apply (erule-tac c=qa in equalityCE)
apply (subgoal-tac  $d \in \text{recDescendants } qa \text{ } hh \implies d \in \text{recDescendants } qa \text{ } h, \text{simp}$ )
apply (erule recReachL-step, simp)
apply (simp add: recDescendants-def)
apply (subgoal-tac recReachL q h  $\subseteq$  closureL q h)
apply blast
by (rule recReachL-subseteq-closureL)

```

```

lemma identityClosure-equals-recReach-h:
   $p \in \text{recReachL } q \text{ } h \implies \text{closureL } q \text{ } h = \text{closureL } q \text{ } hh \longrightarrow (\forall p \in \text{closureL } q \text{ } h. (\text{fst } h) \text{ } p = (\text{fst } hh) \text{ } p) \longrightarrow p \in \text{recReachL } q \text{ } hh$ 
apply (erule recReachL.induct)
apply (intro impI)
apply (rule recReachL-basic)
apply clarsimp
apply (erule-tac c=qa in equalityCE)
apply (subgoal-tac  $d \in \text{recDescendants } qa \text{ } h \implies d \in \text{recDescendants } qa \text{ } hh, \text{simp}$ )

```



```

  apply (erule recReachL-step,simp)
  apply (simp add: recDescendants-def)
  apply (subgoal-tac recReachL q hh  $\subseteq$  closureL q hh)
  apply blast
  by (rule recReachL-subseteq-closureL)

```

**lemma** *identityClosure-equals-recReach:*

```

  identityClosure E x h hh  $\implies$  recReach E x h = recReach E x hh
  apply (simp add: identityClosure-def)
  apply (simp add: closure-def)
  apply (case-tac fst E x,simp-all)
  apply (simp add: recReach-def)
  apply (case-tac a,simp-all)
  apply (elim conjE)
  apply (simp-all add: recReach-def)
  apply (rule equalityI)
  apply (rule subsetI)
  apply (subgoal-tac xa  $\in$  recReachL nat h  $\implies$ 
    closureL nat h = closureL nat hh  $\longrightarrow$  ( $\forall p \in$  closureL nat h. (fst
h) p = (fst hh) p)  $\longrightarrow$  xa  $\in$  recReachL nat hh,simp)
  apply (rule identityClosure-equals-recReach-h,simp)
  apply (rule subsetI)
  apply (subgoal-tac xa  $\in$  recReachL nat hh  $\implies$ 
    closureL nat h = closureL nat hh  $\longrightarrow$  ( $\forall p \in$  closureL nat h. (fst
h) p = (fst hh) p)  $\longrightarrow$  xa  $\in$  recReachL nat h,simp)
  by (rule identityClosure-equals-recReach-hh,simp)

```

**lemma** *closure-monotone-extend:*

```

  [ $\forall i < \text{length } \text{alts}. \text{set } (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! i)))) \cap \text{dom } E = \{\} \wedge$ 
    length (snd (extractP (fst (alts ! i)))) = length vs;
    x  $\in$  dom E;
    length alts > 0;
    i < length alts ]
 $\implies$  closure (E, E') x (h, k) = closure (extend E (snd (extractP (fst (alts ! i))))
vs, E') x (h, k)
  apply (erule-tac x=i in allE)
  apply (subgoal-tac x  $\notin$  set (snd (extractP (fst (alts ! i))))
  apply (subgoal-tac
    E x = extend E (snd (extractP (fst (alts ! i)))) vs x)
  apply (simp add: closure-def)
  apply (rule extend-monotone-i)
  apply (simp,simp,simp)
  by blast

```

**lemma** *closure-monotone-extend-2:*

```


$$\begin{aligned} & \llbracket \text{set} (\text{snd} (\text{extractP} (\text{fst} (\text{alts} ! i)))) \cap \text{dom } E = \{\}; \\ & \quad \text{length} (\text{snd} (\text{extractP} (\text{fst} (\text{alts} ! i)))) = \text{length } vs; \\ & \quad x \in \text{dom } E; \\ & \quad \text{length } \text{alts} > 0; \\ & \quad i < \text{length } \text{alts} \rrbracket \\ & \implies \text{closure} (E, E') x (h, k) = \text{closure} (\text{extend } E (\text{snd} (\text{extractP} (\text{fst} (\text{alts} ! i)))) \\ & \text{vs}, E') x (h, k) \\ & \text{apply} (\text{subgoal-tac } x \notin \text{set} (\text{snd} (\text{extractP} (\text{fst} (\text{alts} ! i)))) \\ & \text{apply} (\text{subgoal-tac} \\ & \quad E x = \text{extend } E (\text{snd} (\text{extractP} (\text{fst} (\text{alts} ! i)))) \text{ vs } x) \\ & \text{apply} (\text{simp add:closure-def}) \\ & \text{apply} (\text{rule extend-monotone-i}) \\ & \text{apply} (\text{simp,simp,simp}) \\ & \text{by blast} \end{aligned}$$

```

**lemma** *closure-monotone-extend-3:*

```


$$\begin{aligned} & \llbracket \text{set} (\text{snd} (\text{extractP} (\text{fst} (\text{alts} ! i)))) \cap \text{dom } E = \{\}; \\ & \quad \text{length} (\text{snd} (\text{extractP} (\text{fst} (\text{alts} ! i)))) = \text{length } vs; \\ & \quad x \in \text{dom } E; \\ & \quad \text{length } \text{alts} > 0; \\ & \quad i < \text{length } \text{alts}; \\ & \quad xa \in \text{closure} (\text{extend } E (\text{snd} (\text{extractP} (\text{fst} (\text{alts} ! i)))) \text{ vs}, E') x (h, k) \rrbracket \\ & \implies xa \in \text{closure} (E, E') x (h, k) \\ & \text{apply} (\text{subgoal-tac } x \notin \text{set} (\text{snd} (\text{extractP} (\text{fst} (\text{alts} ! i)))) \\ & \text{apply} (\text{subgoal-tac} \\ & \quad E x = \text{extend } E (\text{snd} (\text{extractP} (\text{fst} (\text{alts} ! i)))) \text{ vs } x) \\ & \text{apply} (\text{simp add:closure-def}) \\ & \text{apply} (\text{rule extend-monotone-i}) \\ & \text{apply} (\text{simp,simp,simp}) \\ & \text{by blast} \end{aligned}$$

```

**lemma** *recReach-monotone-extend:*

```


$$\begin{aligned} & \llbracket \forall i < \text{length } \text{alts}. \text{set} (\text{snd} (\text{extractP} (\text{fst} (\text{alts} ! i)))) \cap \text{dom } E = \{\} \wedge \\ & \quad \text{length} (\text{snd} (\text{extractP} (\text{fst} (\text{alts} ! i)))) = \text{length } vs; \\ & \quad x \in \text{dom } E; \\ & \quad \text{length } \text{alts} > 0; i < \text{length } \text{alts} \rrbracket \\ & \implies \text{recReach} (E, E') x (h, k) = \text{recReach} (\text{extend } E (\text{snd} (\text{extractP} (\text{fst} (\text{alts} ! \\ & i)))) \text{ vs}, E') x (h, k) \\ & \text{apply} (\text{erule-tac } x=i \text{ in } \text{all } E) \\ & \text{apply} (\text{subgoal-tac } x \notin \text{set} (\text{snd} (\text{extractP} (\text{fst} (\text{alts} ! i)))) \\ & \text{apply} (\text{subgoal-tac} \\ & \quad E x = \text{extend } E (\text{snd} (\text{extractP} (\text{fst} (\text{alts} ! i)))) \text{ vs } x) \\ & \text{apply} (\text{simp add:recReach-def}) \\ & \text{apply} (\text{rule extend-monotone-i}) \\ & \text{apply} (\text{simp,simp,simp}) \\ & \text{by blast} \end{aligned}$$

```

**lemma** *recReach-monotone-extend-2:*

```


$$\llbracket \text{set} (\text{snd} (\text{extractP} (\text{fst} (\text{alts} ! i)))) \cap \text{dom } E = \{\};$$


```

```

length (snd (extractP (fst (alts ! i)))) = length vs;
x ∈ dom E;
length alts > 0; i < length alts
⇒ recReach (E, E') x (h, k) = recReach (extend E (snd (extractP (fst (alts !
i)))) vs, E') x (h, k)
apply (subgoal-tac x ∉ set (snd (extractP (fst (alts ! i))))
apply (subgoal-tac
  E x = extend E (snd (extractP (fst (alts ! i)))) vs x)
apply (simp add:recReach-def)
apply (rule extend-monotone-i)
apply (simp,simp,simp)
by blast

```

**lemma** monotone-extend-closures:

```

[[∀ i < length assert. fst (assert ! i) ⊆ dom (snd (assert ! i));
  ∀ i < length alts. set (snd (extractP (fst (alts ! i)))) ∩ dom E1 = {} ∧
    length (snd (extractP (fst (alts ! i)))) = length vs;
  ∀ i < length assert. dom (snd (assert ! i)) ⊆ dom (extend E1 (snd (extractP
(fst (alts ! i)))) vs);
  z ∈ fst (assert ! i); z ∉ set (snd (extractP (fst (alts ! i)))));
  length alts = length assert;
  closure (E1, E2) x (h, k) ∩ recReach (E1, E2) z (h, k) ≠ {};
  x ∈ dom E1;
  length assert > 0;
  i < length alts ]]
⇒
  closure (extend E1 (snd (extractP (fst (alts ! i)))) vs, E2) x (h, k) ∩
  recReach (extend E1 (snd (extractP (fst (alts ! i)))) vs, E2) z (h, k) ≠ {}
apply (subgoal-tac closure (E1, E2) x (h, k) =
  closure (extend E1 (snd (extractP (fst (alts ! i)))) vs, E2) x (h,
k))
prefer 2 apply (simp,rule closure-monotone-extend,simp,simp,simp,simp)
apply (subgoal-tac z ∈ dom E1)
apply (subgoal-tac recReach (E1, E2) z (h, k) =
  recReach (extend E1 (snd (extractP (fst (alts ! i)))) vs, E2) z (h,
k))
prefer 2 apply (rule recReach-monotone-extend,assumption+,simp,simp)
apply (erule-tac x=i in allE,simp)
apply (erule-tac x=i in allE,simp)
apply (erule-tac x=i in allE,simp)
apply (erule-tac x=i in allE,simp)
apply (subgoal-tac z ∈ dom (extend E1 (snd (extractP (fst (alts ! i)))) vs))
prefer 2 apply blast
apply (elim conjE)
by (rule extend-prop1,assumption+)

```

**lemma** *monotone-extend-closures-i*:

$$\begin{aligned} & \llbracket \text{set} (\text{snd} (\text{extractP} (\text{fst} (\text{alts} ! i)))) \cap \text{dom } E1 = \{\} \wedge \text{length} (\text{snd} (\text{extractP} (\text{fst} (\text{alts} ! i)))) = \text{length } vs; \\ & \text{fst} (\text{assert} ! i) \subseteq \text{dom} (\text{snd} (\text{assert} ! i)); \\ & z \in \text{fst} (\text{assert} ! i); z \notin \text{set} (\text{snd} (\text{extractP} (\text{fst} (\text{alts} ! i)))); \\ & \text{length } \text{alts} = \text{length } \text{assert}; \\ & \text{length } \text{alts} > 0; \\ & \text{closure} (E1, E2) x (h, k) \cap \text{recReach} (E1, E2) z (h, k) \neq \{\}; \\ & \text{dom} (\text{snd} (\text{assert} ! i)) \subseteq \text{dom} (\text{extend } E1 (\text{snd} (\text{extractP} (\text{fst} (\text{alts} ! i)))) vs); \\ & x \in \text{dom } E1; \\ & i < \text{length } \text{alts} \rrbracket \\ \implies & \\ & \text{closure} (\text{extend } E1 (\text{snd} (\text{extractP} (\text{fst} (\text{alts} ! i)))) vs, E2) x (h, k) \cap \\ & \text{recReach} (\text{extend } E1 (\text{snd} (\text{extractP} (\text{fst} (\text{alts} ! i)))) vs, E2) z (h, k) \neq \{\} \\ \text{apply } & (\text{elim conjE}) \\ \text{apply } & (\text{subgoal-tac closure} (E1, E2) x (h, k) = \\ & \text{closure} (\text{extend } E1 (\text{snd} (\text{extractP} (\text{fst} (\text{alts} ! i)))) vs, E2) x (h, \\ & k)) \\ \text{prefer } & 2 \text{ apply } (\text{simp, rule closure-monotone-extend-2, simp, assumption+, simp, simp}) \\ \\ \text{apply } & (\text{subgoal-tac } z \in \text{dom } E1) \\ \text{apply } & (\text{subgoal-tac recReach} (E1, E2) z (h, k) = \\ & \text{recReach} (\text{extend } E1 (\text{snd} (\text{extractP} (\text{fst} (\text{alts} ! i)))) vs, E2) z (h, \\ & k)) \\ \text{prefer } & 2 \text{ apply } (\text{rule recReach-monotone-extend-2, simp, assumption+, simp}) \\ \text{apply } & (\text{subgoal-tac } z \in \text{dom} (\text{extend } E1 (\text{snd} (\text{extractP} (\text{fst} (\text{alts} ! i)))) vs) \\ \text{prefer } & 2 \text{ apply blast} \\ \text{by } & (\text{rule extend-prop1, assumption+}) \\ \\ \text{end} & \end{aligned}$$

## 10 SafeDAssBasic

**constdefs** *SSet* :: *string set*  $\Rightarrow$  *TypeEnvironment*  $\Rightarrow$  *Environment*  $\Rightarrow$  *Heap*  $\Rightarrow$  *Location set*

*SSet* *L*  $\Gamma$  *E* *h*  $\equiv$  *let* *LS* =  $\{z \in L. \Gamma z = \text{Some } s''\}$   
*in*  $(\bigcup x \in LS. \text{closure } E x h)$

**constdefs** *SSet1* :: *string set*  $\Rightarrow$  *TypeEnvironment*  $\Rightarrow$  *TypeEnvironment*  $\Rightarrow$  *Mark*  $\Rightarrow$  *Environment*  $\Rightarrow$  *Heap*  $\Rightarrow$  *Location set*

*SSet1* *L*  $\Gamma 1$   $\Gamma$  *m* *E* *h*  $\equiv$   $(\bigcup x \in \{z \in L. \Gamma z = (\text{Some } m) \wedge \Gamma 1 z = (\text{Some } s'')\}. \text{closure } E x h)$

**constdefs** *RSet* :: *string set*  $\Rightarrow$  *TypeEnvironment*  $\Rightarrow$  *Environment*  $\Rightarrow$  *Heap*  $\Rightarrow$  *Location set*

*RSet* *L*  $\Gamma$  *E* *h*  $\equiv$   $\{p \in \text{live } E L h. \exists z \in L. \Gamma z = \text{Some } d'' \wedge \text{closureL } p h \cap \text{recReach } E z h \neq \{\}\}$

**constdefs**  $RSet1 :: string\ set \Rightarrow TypeEnvironment \Rightarrow TypeEnvironment \Rightarrow Mark$   
 $\Rightarrow Environment \Rightarrow Heap \Rightarrow Heap \Rightarrow Location\ set$   
 $RSet1\ L\ \Gamma1\ \Gamma\ m\ E\ h\ hh \equiv \{p \in live\ E\ L\ h. \exists z \in L. \Gamma\ z = Some\ m \wedge \Gamma1\ z =$   
 $Some\ d'' \wedge$

$$closureL\ p\ h \cap recReach\ E\ z\ h \neq \{\}\} \cup \\ \{p \in scope\ E\ h. \neg identityClosureL\ p\ h\ hh\}$$

**constdefs**  $RSet2 :: string\ set \Rightarrow string\ set \Rightarrow TypeEnvironment \Rightarrow Environment$   
 $\Rightarrow Heap \Rightarrow Location\ set$   
 $RSet2\ L\ L'\ \Gamma\ E\ h \equiv \{p \in live\ E\ L'\ h. \exists z \in L. \Gamma\ z = Some\ d'' \wedge$   
 $closureL\ p\ h \cap recReach\ E\ z\ h \neq \{\}\}$

**constdefs**  $shareRec :: string\ set \Rightarrow TypeEnvironment \Rightarrow Environment \Rightarrow Heap \Rightarrow$   
 $Heap \Rightarrow bool$   
 $shareRec\ L\ \Gamma\ E\ h\ hh \equiv (\forall x \in dom\ (fst\ E). \forall\ z \in L. \Gamma\ z = Some\ d'' \wedge$   
 $closure\ E\ x\ h \cap recReach\ E\ z\ h \neq \{\}$   
 $\longrightarrow x \in dom\ \Gamma \wedge \Gamma\ x \neq Some\ s'')$   
 $\wedge$   
 $(\forall x \in dom\ (fst\ E). \neg identityClosure\ E\ x\ h\ hh$   
 $\longrightarrow x \in dom\ \Gamma \wedge \Gamma\ x \neq Some\ s'')$

**constdefs**  $wellFormed :: string\ set \Rightarrow TypeEnvironment \Rightarrow unit\ Exp \Rightarrow bool$   
 $wellFormed\ L\ \Gamma\ e \equiv (\forall\ E1\ E2\ h\ k\ td\ hh\ v\ r.$   
 $(E1, E2) \vdash h, k, td, e \Downarrow hh, k, v, r$   
 $\wedge dom\ \Gamma \subseteq dom\ E1$   
 $\wedge L \subseteq dom\ \Gamma$   
 $\wedge fv\ e \subseteq L$   
 $\longrightarrow (\forall x \in dom\ E1. \forall\ z \in L. \Gamma\ z = Some\ d'' \wedge$   
 $closure\ (E1, E2)\ x\ (h, k) \cap recReach\ (E1, E2)\ z\ (h, k) \neq \{\}$   
 $\longrightarrow x \in dom\ \Gamma \wedge \Gamma\ x \neq Some\ s''))$

**constdefs**  $wellFormedDepth :: string \Rightarrow nat \Rightarrow string\ set \Rightarrow TypeEnvironment \Rightarrow$   
 $unit\ Exp \Rightarrow bool$   
 $wellFormedDepth\ f\ n\ L\ \Gamma\ e \equiv (\forall\ E1\ E2\ h\ k\ hh\ v.$   
 $(E1, E2) \vdash h, k, e \Downarrow (f, n)\ hh, k, v$   
 $\wedge dom\ \Gamma \subseteq dom\ E1$   
 $\wedge L \subseteq dom\ \Gamma$   
 $\wedge fv\ e \subseteq L$   
 $\longrightarrow (\forall x \in dom\ E1. \forall\ z \in L. \Gamma\ z = Some\ d'' \wedge$   
 $closure\ (E1, E2)\ x\ (h, k) \cap recReach\ (E1, E2)\ z\ (h, k) \neq \{\}$   
 $\longrightarrow x \in dom\ \Gamma \wedge \Gamma\ x \neq Some\ s''))$

**lemma**  $imp\ wellFormed\ wellFormedDepth$ :

```

  wellFormed L Γ e  $\implies$  wellFormedDepth f n L Γ e
apply (simp only: wellFormed-def)
apply (simp only: wellFormedDepth-def)
apply (rule allI)+
apply (rule impI)
apply (elim conjE)
apply (frule impSemBoundRA [where td=td])
apply (elim exE)
apply (erule-tac x=E1 in allE)
apply (erule-tac x=E2 in allE)
apply (erule-tac x=h in allE)
apply (erule-tac x=k in allE)
apply (erule-tac x=td in allE)
apply (erule-tac x=hh in allE)
apply (erule-tac x=v in allE)
apply (erule-tac x=r in allE)
apply (drule mp,simp)
by (frule impSemBoundRA,simp)

```

```

constdefs SR :: string set  $\Rightarrow$  TypeEnvironment  $\Rightarrow$  Environment  $\Rightarrow$  Heap  $\Rightarrow$  Heap
 $\Rightarrow$  string set
SR L Γ E h hh  $\equiv$  {x $\in$  dom (fst E).  $\exists$  z  $\in$  L. Γ z = Some d''  $\wedge$  closure E x h  $\cap$ 
recReach E z h  $\neq$  {}}

```

#### definition

```

restrict-neg-map :: ('a  $\leadsto$  'b)  $\Rightarrow$  'a set  $\Rightarrow$  ('a  $\leadsto$  'b) where
restrict-neg-map m A = (λx. if x : A then None else m x)

```

#### consts

```

maps-of :: ('a * 'b) list  $\Rightarrow$  ('a  $\leadsto$  'b) list

```

#### primrec

```

maps-of [] = [empty]
maps-of (p#ps) = [fst p  $\mapsto$  snd p] # maps-of ps

```

#### axioms z-in-SR:

```

 $\neg$  identityClosure (E1, E2) z (h, k) (h', k')  $\implies$  z  $\in$  SR L Γ (E1,E2) (h,k)
(h',k')

```

```

types MarkEnv = string  $\rightarrow$  Mark list

```

```

constdefs SafeDAss ::

```

$unit\ Exp \Rightarrow string\ set \Rightarrow TypeEnvironment \Rightarrow bool\ (- : \mathbb{I} - , - \mathbb{I}\ 1000)$   
 $SafeDAss\ e\ L\ \Gamma \equiv$   
 $fv\ e \subseteq L \wedge L \subseteq dom\ \Gamma \wedge$   
 $(\forall\ E1\ E2\ h\ k\ td\ hh\ v\ r.$   
 $(E1, E2) \vdash h, k, td, e \Downarrow hh, k, v, r$   
 $\wedge dom\ \Gamma \subseteq dom\ E1$   
 $\longrightarrow shareRec\ L\ \Gamma\ (E1, E2)\ (h, k)\ (hh, k)$   
 $\wedge (closed\ (E1, E2)\ L\ (h, k)$   
 $\wedge SSet\ L\ \Gamma\ (E1, E2)\ (h, k) \cap RSet\ L\ \Gamma\ (E1, E2)\ (h, k) = \{\}$   
 $\longrightarrow closed\text{-}f\ v\ (hh, k)))$

**inductive**

$ValidGlobalMarkEnv :: MarkEnv \Rightarrow bool\ (\models -\ 1000)$

**where**

$base: \models empty$   
 $| step: \mathbb{I} \models \Sigma m; f \notin dom\ \Sigma m;$   
 $Lf = set\ (varsAPP\ \Sigma f\ f); \Gamma f = empty\ ((varsAPP\ \Sigma f\ f)\ [\mapsto] ms);$   
 $(bodyAPP\ \Sigma f\ f) : \mathbb{I} Lf, \Gamma f \mathbb{I} \Longrightarrow \models \Sigma m(f \mapsto ms)$

**constdefs**  $SafeDAssCntxt ::$

$unit\ Exp \Rightarrow MarkEnv \Rightarrow string\ set \Rightarrow TypeEnvironment \Rightarrow bool\ (-, - : \mathbb{I} - , - \mathbb{I}\ 1000)$   
 $SafeDAssCntxt\ e\ \Sigma m\ L\ \Gamma \equiv (\models \Sigma m \longrightarrow e : \mathbb{I} L, \Gamma \mathbb{I})$

**constdefs**  $SafeDAssDepth ::$

$unit\ Exp \Rightarrow string \Rightarrow nat \Rightarrow string\ set \Rightarrow TypeEnvironment \Rightarrow bool\ (- :-, - : \mathbb{I} - , - \mathbb{I}\ 1000)$   
 $SafeDAssDepth\ e\ f\ n\ L\ \Gamma \equiv$   
 $fv\ e \subseteq L \wedge L \subseteq dom\ \Gamma \wedge$   
 $(\forall\ E1\ E2\ h\ k\ hh\ v.$   
 $(E1, E2) \vdash h, k, e \Downarrow (f, n)\ hh, k, v$   
 $\wedge dom\ \Gamma \subseteq dom\ E1$   
 $\longrightarrow shareRec\ L\ \Gamma\ (E1, E2)\ (h, k)\ (hh, k)$   
 $\wedge (closed\ (E1, E2)\ L\ (h, k)$   
 $\wedge SSet\ L\ \Gamma\ (E1, E2)\ (h, k) \cap RSet\ L\ \Gamma\ (E1, E2)\ (h, k) = \{\}$   
 $\longrightarrow closed\text{-}f\ v\ (hh, k)))$

**inductive**  $ValidGlobalMarkEnvDepth :: string \Rightarrow nat \Rightarrow MarkEnv \Rightarrow bool$   
 $(\models -, -\ 1000)$

**where**

$base : \mathbb{I} \models \Sigma m; f \notin dom\ \Sigma m \mathbb{I} \Longrightarrow \models_{f, n} \Sigma m$

$\mid \text{depth0} : \llbracket \models \Sigma m; f \notin \text{dom } \Sigma m \rrbracket \implies \models_{f,0} \Sigma m(f \mapsto ms)$   
 $\mid \text{step} : \llbracket \models \Sigma m; f \notin \text{dom } \Sigma m;$   
 $\quad Lf = \text{set } (\text{varsAPP } \Sigma f f); \Gamma f = \text{empty } ((\text{varsAPP } \Sigma f f) [\mapsto] ms);$   
 $\quad (\text{bodyAPP } \Sigma f f) :_{f,n} \llbracket Lf, \Gamma f \rrbracket \rrbracket \implies \models_{f,\text{Suc } n} \Sigma m(f \mapsto ms)$   
 $\mid g : \llbracket \models_{f,n} \Sigma m; g \notin \text{dom } \Sigma m; g \neq f;$   
 $\quad Lg = \text{set } (\text{varsAPP } \Sigma f g); \Gamma g = \text{empty } ((\text{varsAPP } \Sigma f g) [\mapsto] ms);$   
 $\quad (\text{bodyAPP } \Sigma f g) : \llbracket Lg, \Gamma g \rrbracket \rrbracket \implies \models_{f,n} \Sigma m(g \mapsto ms)$

**constdefs** *SafeDAssDepthCntxt* ::

$\text{unit Exp} \Rightarrow \text{MarkEnv} \Rightarrow \text{string} \Rightarrow \text{nat} \Rightarrow \text{string set} \Rightarrow \text{TypeEnvironment} \Rightarrow \text{bool}$   
 $(-, - : -, - \llbracket -, - \rrbracket 1000)$   
 $\text{SafeDAssDepthCntxt } e \Sigma m f n L \Gamma \equiv ( \models_{f,n} \Sigma m \longrightarrow e :_{f,n} \llbracket L, \Gamma \rrbracket )$

Lemmas

**lemma** *equals-recReach*:

$\llbracket z \neq x1; z \in L; \Gamma z = \text{Some } s''; \text{identityClosure } (E1, E2) z h hh \rrbracket$   
 $\implies \text{recReach } (E1, E2) z h = \text{recReach } (E1(x1 \mapsto r), E2) z hh$   
**apply** (*subgoal-tac*  $z \neq x1 \implies \text{recReach } (E1(x1 \mapsto r), E2) z hh = \text{recReach } (E1, E2) z hh, \text{simp}$ )  
**apply** (*erule identityClosure-equals-recReach*)  
**apply** *simp*  
**by** (*simp add: recReach-def*)

**lemma** *monotone-identityClosure*:

$\llbracket x \neq x1; \text{identityClosure } (E1, E2) x (h, k) (h', k');$   
 $\quad \text{identityClosure } (E1(x1 \mapsto r), E2) x (h', k') (hh, kk) \rrbracket$   
 $\implies \text{identityClosure } (E1, E2) x (h, k) (hh, kk)$   
**apply** (*simp add: identityClosure-def*)  
**apply** (*elim conjE*)  
**apply** (*rule conjI*)  
**apply** (*simp add: closure-def*)  
**apply** (*rule ballI*)  
**apply** (*erule-tac*  $x=p$  **in** *ballE*)  
**apply** *simp*  
**apply** (*simp add: closure-def*)  
**by** *simp*

**lemma** *unsafe-Gamma2-identityClosure*:

$\llbracket L2 \subseteq \text{dom } (\text{disjointUnionEnv } \Gamma 2 (\text{empty}(x1 \mapsto m)));$   
 $\text{def-disjointUnionEnv } \Gamma 2 (\text{empty}(x1 \mapsto m));$   
 $\text{dom } (\text{disjointUnionEnv } \Gamma 2 (\text{empty}(x1 \mapsto m))) \subseteq \text{insert } x1 (\text{dom } E1); y \neq x1;$   
 $\forall x \in \text{dom } E1. (\exists z \in L1. \Gamma 1 z = \text{Some } d'' \wedge \text{closure } (E1, E2) x (h, k) \cap \text{recReach}$   
 $(E1, E2) z (h, k) \neq \{\}) \longrightarrow x \in \text{dom } \Gamma 1 \wedge \Gamma 1 x \neq \text{Some } s'';$   
 $\forall x \in \text{dom } E1. \neg \text{identityClosure } (E1, E2) x (h, k) (h', k') \longrightarrow x \in \text{dom } \Gamma 1 \wedge$   
 $\Gamma 1 x \neq \text{Some } s'';$   
 $\text{def-pp } \Gamma 1 \Gamma 2 L2;$



```

     $\Gamma 2 \ y \neq \text{Some } s''; y \in L2 \rrbracket$ 
 $\implies \text{identityClosure } (E1, E2) \ y \ (h, k) \ (h', k')$ 
apply (erule-tac  $x=y$  in ballE)+
prefer 2 apply blast
prefer 2 apply blast
apply (frule-tac  $x=y$  in safe-Gamma-triangle-3, assumption)+
apply (case-tac  $\neg \text{identityClosure } (E1, E2) \ y \ (h, k) \ (h', k')$ )
apply simp
apply simp
done

```

```

lemma closure-subset-live:
 $\llbracket y \neq x1; y \in L2 \rrbracket \implies \text{closure } (E1, E2) \ y \ (h, k) \subseteq \text{live } (E1, E2) \ (L1 \cup (L2 - \{x1\})) \ (h, k)$ 
apply (simp add: live-def add: closureLS-def)
apply blast
done

```

```

lemma closure-live-monotone:
 $\llbracket p \in \text{closure } (E1(x1 \mapsto r), E2) \ y \ (h', k');$ 
 $\text{closure } (E1(x1 \mapsto r), E2) \ y \ (h', k') \subseteq \text{live } (E1, E2) \ (L1 \cup (L2 - \{x1\})) \ (h,$ 
 $k) \rrbracket$ 
 $\implies p \in \text{live } (E1, E2) \ (L1 \cup (L2 - \{x1\})) \ (h, k)$ 
apply blast
done

```

**end**

## 11 Derived Assertions. P2. $\text{dom } \Gamma \subseteq \text{dom } E$

```

theory SafeDAss-P2 imports SafeDAssBasic
begin

```

Lemmas for LET1 and LET2

```

lemma dom-Γ1-subseteq-triangle-Γ1-Γ2-L2:
 $\text{dom } \Gamma 1 \subseteq \text{dom } (pp \ \Gamma 1 \ \Gamma 2 \ L2)$ 
by (simp add: pp-def, force)

```

```

lemma dom-Γ2-subseteq-triangle-Γ1-Γ2-L2:
 $\text{dom } \Gamma 2 \subseteq \text{dom } (pp \ \Gamma 1 \ \Gamma 2 \ L2)$ 
by (simp add: pp-def, clarsimp)

```

```

lemma P2-LET-e1:
 $\llbracket \text{dom } (pp \ \Gamma 1 \ \Gamma 2 \ L2) \subseteq \text{dom } E1 \rrbracket$ 
 $\implies \text{dom } \Gamma 1 \subseteq \text{dom } E1$ 

```

**apply** (*subgoal-tac*  $\text{dom } \Gamma 1 \subseteq \text{dom } (pp \ \Gamma 1 \ \Gamma 2 \ L2)$ )  
**by** (*blast*, *rule dom- $\Gamma 1$ -subseteq-triangle- $\Gamma 1$ - $\Gamma 2$ - $L2$* )

**lemma** *P2-LET-e2*:

$\llbracket$  *def-disjointUnionEnv*  $\Gamma 2 \ [x1 \mapsto m]$ ;  
 $x1 \notin \text{dom } E1$ ;  
 $\text{dom } (pp \ \Gamma 1 \ \Gamma 2 \ L2) \subseteq \text{dom } E1 \rrbracket$   
 $\implies \text{dom } (\Gamma 2 + [x1 \mapsto m]) \subseteq \text{dom } (E1(x1 \mapsto v1))$   
**apply** (*rule subsetI*)  
**apply** (*case-tac*  $x \neq x1$ , *simp*)  
**apply** (*subgoal-tac*  $x \in \text{dom } (pp \ \Gamma 1 \ \Gamma 2 \ L2)$ )  
**apply** *blast*  
**apply** (*frule union-dom-disjointUnionEnv, simp add: def-disjointUnionEnv-def*)  
**apply** (*subgoal-tac*  $\text{dom } \Gamma 2 \subseteq \text{dom } (pp \ \Gamma 1 \ \Gamma 2 \ L2)$ )  
**apply** *blast*  
**apply** (*rule dom- $\Gamma 2$ -subseteq-triangle- $\Gamma 1$ - $\Gamma 2$ - $L2$* )  
**by** *simp*

**lemma** *P2-LET*:

$\llbracket$  *def-disjointUnionEnv*  $\Gamma 2 \ (\text{empty}(x1 \mapsto m))$ ;  
 $x1 \notin \text{dom } E1$ ;  
 $\text{dom } (pp \ \Gamma 1 \ \Gamma 2 \ L2) \subseteq \text{dom } E1 \rrbracket$   
 $\implies \text{dom } \Gamma 1 \subseteq \text{dom } E1 \wedge \text{dom } (\Gamma 2 + (\text{empty}(x1 \mapsto m))) \subseteq \text{dom } (E1(x1 \mapsto r))$   
**apply** (*rule conjI*)  
**apply** (*erule P2-LET-e1*)  
**by** (*erule P2-LET-e2, assumption+*)

Lemmas for CASE

**lemma** *P2-CASE*:

$\llbracket$   $\text{length } \text{assert} > 0$ ;  $i < \text{length } \text{alts}$ ;  $\text{length } \text{assert} = \text{length } \text{alts}$ ;  
 $\text{def-extend } E1 \ (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! \ i)))) \ \text{vs}$ ;  
 $\text{dom } (\text{foldl } op \ \otimes \ \text{empty } (\text{map } \text{snd } \text{assert})) \subseteq \text{dom } E1 \rrbracket$   
 $\implies \text{dom } (\text{snd } (\text{assert } ! \ i)) \subseteq \text{dom } (\text{extend } E1 \ (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! \ i))))$   
 $\text{vs})$   
**apply** (*frule dom- $\Gamma i$ -subseq-dom- $\Gamma$ -case*)  
**apply** (*subgoal-tac*  $\text{dom } E1 \subseteq \text{dom } (\text{extend } E1 \ (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! \ i)))) \ \text{vs})$ )  
**apply** (*erule-tac*  $x=i$  **in** *allE, simp*)  
**by** (*simp add: extend-def, blast*)

**lemma** *P2-CASE-1-1*:

$\llbracket$   $\text{length } \text{assert} > 0$ ;  $i < \text{length } \text{alts}$ ;  $\text{length } \text{assert} = \text{length } \text{alts}$ ;  
 $\text{dom } (\text{foldl } op \ \otimes \ \text{empty } (\text{map } \text{snd } \text{assert})) \subseteq \text{dom } E1 \rrbracket$   
 $\implies \text{dom } (\text{snd } (\text{assert } ! \ i)) \subseteq \text{dom } E1$   
**apply** (*frule dom- $\Gamma i$ -subseq-dom- $\Gamma$ -case*)

**by** (*erule-tac*  $x=i$  **in** *allE,simp*)

Lemmas for CASED

**lemma** *dom-restrict-neg-map*:

$\text{dom } (\text{restrict-neg-map } m \ A) = \text{dom } m - (\text{dom } m \cap A)$

**apply** (*simp add: restrict-neg-map-def,auto*)

**by** (*split split-if-asm,simp,simp*)

**lemma** *dom-G-dom-restrict-neg-map*:

$\text{dom } G = \text{dom } (\text{restrict-neg-map } G \ A) \cup (\text{dom } G \cap A)$

**apply** (*subst dom-restrict-neg-map*)

**by** *blast*

**lemma** *dom-map-of-zip*:

$\text{length } xs = \text{length } ys$

$\implies \text{dom } (\text{map-of } (\text{zip } xs \ ys)) = \text{set } xs$

**by** (*induct xs ys rule: list-induct2',simp-all*)

**lemma** *dom-foldl-disjointUnionEnv-monotone-generic*:

$\text{dom } (\text{foldl } op \otimes (\text{empty} \otimes y) \ ys + [x \mapsto d']) =$

$\text{dom } y \cup \text{dom } (\text{foldl } op \otimes \text{empty } ys) \cup \text{dom } [x \mapsto d']$

**apply** (*subgoal-tac empty  $\otimes y = y \otimes \text{empty}$ ,simp*)

**apply** (*subgoal-tac foldl  $op \otimes (y \otimes \text{empty}) \ ys =$   
 $y \otimes \text{foldl } op \otimes \text{empty } ys$ ,simp*)

**apply** (*subst dom-disjointUnionEnv-monotone*)

**apply** (*subst union-dom-nonDisjointUnionEnv*)

**apply** *simp*

**apply** (*rule foldl-prop1*)

**apply** (*subgoal-tac def-nonDisjointUnionEnv empty y*)

**apply** (*erule nonDisjointUnionEnv-commutative*)

**by** (*simp add: def-nonDisjointUnionEnv-def*)

**lemma** *dom-monotone-foldl-nonDisjointUnionEnv-Gis*:

$\text{length } Gis > i \implies \text{dom } (Gis ! i) \subseteq \text{dom } (\text{foldl } op \otimes \text{empty } Gis + [x \mapsto d'])$

**apply** (*induct Gis i rule: list-induct3, simp-all*)

**apply** (*subgoal-tac dom (foldl  $op \otimes (\text{empty} \otimes xa) \ xs + [x \mapsto d'] =$*

$\text{dom } xa \cup \text{dom } (\text{foldl } op \otimes \text{empty } xs) \cup \text{dom } [x \mapsto d']$ ,simp)

**apply** *blast*

**apply** (*rule dom-foldl-disjointUnionEnv-monotone-generic*)

**apply** (*subgoal-tac dom (foldl  $op \otimes (\text{empty} \otimes xa) \ xs + [x \mapsto d'] =$*

$\text{dom } xa \cup \text{dom } (\text{foldl } op \otimes \text{empty } xs) \cup \text{dom } [x \mapsto d']$ ,simp)

**apply** (*subgoal-tac dom (foldl  $op \otimes \text{empty } xs + [x \mapsto d'] =$*

$\text{dom } (\text{foldl } op \otimes \text{empty } xs) \cup \text{dom } [x \mapsto d']$ ,simp)

**apply** *blast*

**apply** (*rule dom-disjointUnionEnv-monotone*)

**by** (*rule dom-foldl-disjointUnionEnv-monotone-generic*)

**lemma** *dom- $\Gamma$ -case-subseteq-dom- $\Gamma$ i* [rule-format]:  
 $\text{dom } (\text{foldl } \text{op} \otimes \text{empty } \text{Gis} + [x \mapsto d']) \subseteq \text{dom } E1$   
 $\longrightarrow \text{def-disjointUnionEnv } (\text{foldl } \text{op} \otimes \text{empty } \text{Gis}) [x \mapsto d']$   
 $\longrightarrow \text{length } \text{Gis} > 0$   
 $\longrightarrow i < \text{length } \text{Gis}$   
 $\longrightarrow \text{dom } (\text{Gis}!i) \subseteq \text{dom } E1$   
**apply** (rule *impI*) +  
**apply** (subgoal-tac  
 $\text{dom } (\text{Gis}!i) \subseteq \text{dom } (\text{foldl } \text{op} \otimes \text{empty } \text{Gis} + [x \mapsto d'])$ )  
**apply** blast  
**by** (rule *dom-monotone-foldl-nonDisjointUnionEnv-Gis,assumption*)

**lemma** *P2-CASED*:  
 $\llbracket \text{length } \text{assert} > 0; \text{length } \text{assert} = \text{length } \text{alts}; x \in \text{dom } E1;$   
 $\text{length } (\text{snd } (\text{extractP } (\text{fst } (\text{alts}!i)))) = \text{length } \text{vs};$   
 $i < \text{length } \text{alts};$   
 $\text{def-disjointUnionEnv } (\text{foldl } \text{op} \otimes \text{empty}$   
 $\quad (\text{map } (\lambda(Li, \Gamma i). \text{restrict-neg-map } \Gamma i (\text{insert } x (\text{set } Li)))$   
 $\quad (\text{zip } (\text{map } (\text{snd} \circ \text{extractP} \circ \text{fst}) \text{alts}) (\text{map } \text{snd } \text{assert})))$   
 $\quad [x \mapsto d'];$   
 $\text{length } (\text{snd } (\text{extractP } (\text{fst } (\text{alts}!i)))) = \text{length } \text{vs};$   
 $\text{dom } (\text{foldl } \text{op} \otimes \text{empty}$   
 $\quad (\text{map } (\lambda(Li, \Gamma i). \text{restrict-neg-map } \Gamma i (\text{insert } x (\text{set } Li)))$   
 $\quad (\text{zip } (\text{map } (\text{snd} \circ \text{extractP} \circ \text{fst}) \text{alts}) (\text{map } \text{snd } \text{assert}))) +$   
 $\quad [x \mapsto d']) \subseteq \text{dom } E1 \rrbracket$   
 $\implies \text{dom } (\text{snd } (\text{assert}!i)) \subseteq \text{dom } (\text{extend } E1 (\text{snd } (\text{extractP } (\text{fst } (\text{alts}!i))))$   
 $\text{vs})$   
**apply** (subst *dom-G-dom-restrict-neg-map* [where  $A = (\text{insert } x (\text{set } (\text{snd } (\text{extractP}$   
 $(\text{fst } (\text{alts}!i))))$ ]))  
**apply** (simp add: *extend-def*)  
**apply** (rule *conjI*)  
**apply** (frule *dom- $\Gamma$ -case-subseteq-dom- $\Gamma$ i*)  
**apply** (assumption+, simp, simp, simp)  
**apply** force  
**apply** (subst *dom-map-of-zip, simp*)  
**by** blast

**end**

## 12 Derived Assertions. P3. L dom G

**theory** *SafeDAss-P3* **imports** *SafeDAssBasic*  
**begin**

Lemmas for LET

**lemma** *dom- $\Gamma$ 2-subseteq-triangle- $\Gamma$ 1- $\Gamma$ 2-L2*:

$dom \Gamma 2 \subseteq dom (pp \Gamma 1 \Gamma 2 L2)$   
**by** (*simp add: pp-def, clarsimp*)

**lemma** *set-atom2var-as-subeteq- $\Gamma 1$* :

$\forall a \in set \ as. \ atom \ a$   
 $\implies set \ (map \ atom2var \ as) \subseteq$   
 $dom \ (map-of \ (zip \ (map \ atom2var \ as) \ (replicate \ (length \ as) \ s''))$   
**apply** (*induct as, simp, clarsimp*)  
**apply** (*case-tac a, simp-all*)  
**by** *force*

**lemma** *P3-LET-e1*:

$\llbracket L1 \subseteq dom \Gamma 1 \rrbracket$   
 $\implies L1 \subseteq dom (pp \Gamma 1 \Gamma 2 L2)$   
**by** (*simp add: pp-def, auto*)

**lemma** *P3-LET-e2*:

$\llbracket def-disjointUnionEnv \ \Gamma 2 \ (empty(x1 \mapsto m));$   
 $L2 \subseteq dom \ (disjointUnionEnv \ \Gamma 2 \ (empty(x1 \mapsto m))) \rrbracket$   
 $\implies L2 - \{x1\} \subseteq dom (pp \Gamma 1 \Gamma 2 L2)$   
**apply** (*rule subsetI*)  
**apply** (*frule union-dom-disjointUnionEnv*)  
**apply** (*simp add: def-disjointUnionEnv-def*)  
**apply** (*subgoal-tac dom \Gamma 2 \subseteq dom (pp \Gamma 1 \Gamma 2 L2), blast*)  
**by** (*rule dom- $\Gamma 2$ -subeteq-triangle- $\Gamma 1$ - $\Gamma 2$ - $L2$* )

**lemma** *P3-LET*:

$\llbracket def-disjointUnionEnv \ \Gamma 2 \ (empty(x1 \mapsto m));$   
 $L1 \subseteq dom \Gamma 1;$   
 $L2 \subseteq dom \ (disjointUnionEnv \ \Gamma 2 \ (empty(x1 \mapsto m))) \rrbracket$   
 $\implies L1 \cup (L2 - \{x1\}) \subseteq dom (pp \Gamma 1 \Gamma 2 L2)$   
**apply** *clarsimp*  
**apply** (*rule conjI*)  
**apply** (*erule P3-LET-e1*)  
**by** (*erule P3-LET-e2, assumption*)

Lemmas for CASE

**lemma** *P3-CASE*:

$\llbracket length \ assert > 0;$   
 $length \ alts = length \ assert;$   
 $(\forall i < length \ alts. \ fst \ (assert \ ! \ i) \subseteq dom \ (snd \ (assert \ ! \ i)));$   
 $x \in dom \ (nonDisjointUnionEnvList \ (map \ snd \ assert)) \rrbracket$   
 $\implies (\bigcup_{i < length \ alts} \ fst \ (assert \ ! \ i) - set \ (snd \ (extractP \ (fst \ (alts \ ! \ i))))) \cup \{x\}$   
 $\subseteq$   
 $dom \ (nonDisjointUnionEnvList \ (map \ snd \ assert))$   
**apply** (*frule dom- $\Gamma i$ -subeteq-dom- $\Gamma$ -case*)  
**by** (*clarsimp, blast*)

Lemmas for CASED

**lemma** *P3-1-CASED*:

$\llbracket \text{length } \text{assert} > 0;$   
 $\text{length } \text{assert} = \text{length } \text{alts};$   
 $(\forall i < \text{length } \text{alts}. \text{fst } (\text{assert } ! i) \subseteq \text{dom } (\text{snd } (\text{assert } ! i))) \rrbracket$   
 $\implies x \in \text{dom } (\text{foldl } \text{op } \otimes \text{empty}$   
 $\quad (\text{map } (\lambda(Li, \Gamma i). \text{restrict-neg-map } \Gamma i (\text{insert } x (\text{set } Li)))$   
 $\quad (\text{zip } (\text{map } (\text{snd} \circ \text{extractP} \circ \text{fst}) \text{alts}) (\text{map } \text{snd } \text{assert}))) +$   
 $\quad [x \mapsto d']]$   
**apply** (*subst dom-disjointUnionEnv-monotone*)  
**by** (*simp add: dom-def*)

**lemma** *dom-restrict-neg-map*:

$\text{dom } (\text{restrict-neg-map } m A) = \text{dom } m - (\text{dom } m \cap A)$   
**apply** (*simp add: restrict-neg-map-def*)  
**apply** *auto*  
**by** (*split split-if-asm, simp-all*)

**lemma** *dom-foldl-monotone-generic*:

$\text{dom } (\text{foldl } \text{op } \otimes (\text{empty} \otimes x) \text{xs}) =$   
 $\text{dom } x \cup \text{dom } (\text{foldl } \text{op } \otimes \text{empty } \text{xs})$   
**apply** (*subgoal-tac empty  $\otimes x = x \otimes \text{empty}$ , simp*)  
**apply** (*subgoal-tac foldl op  $\otimes (x \otimes \text{empty}) \text{xs} =$*   
 $\quad x \otimes \text{foldl } \text{op } \otimes \text{empty } \text{xs}, \text{simp}$ )  
**apply** (*rule union-dom-nonDisjointUnionEnv*)  
**apply** (*rule foldl-prop1*)  
**apply** (*subgoal-tac def-nonDisjointUnionEnv empty x*)  
**apply** (*erule nonDisjointUnionEnv-commutative*)  
**by** (*simp add: def-nonDisjointUnionEnv-def*)

**lemma** *dom-foldl-disjointUnionEnv-monotone-generic-2*:

$\text{dom } (\text{foldl } \text{op } \otimes (\text{empty} \otimes y) \text{ys} + A) =$   
 $\text{dom } y \cup \text{dom } (\text{foldl } \text{op } \otimes \text{empty } \text{ys}) \cup \text{dom } A$   
**apply** (*subgoal-tac empty  $\otimes y = y \otimes \text{empty}$ , simp*)  
**apply** (*subgoal-tac foldl op  $\otimes (y \otimes \text{empty}) \text{ys} =$*   
 $\quad y \otimes \text{foldl } \text{op } \otimes \text{empty } \text{ys}, \text{simp}$ )  
**apply** (*subst dom-disjointUnionEnv-monotone*)  
**apply** (*subst union-dom-nonDisjointUnionEnv*)  
**apply** *simp*  
**apply** (*rule foldl-prop1*)  
**apply** (*subgoal-tac def-nonDisjointUnionEnv empty y*)  
**apply** (*erule nonDisjointUnionEnv-commutative*)  
**by** (*simp add: def-nonDisjointUnionEnv-def*)

**lemma** *dom- $\Gamma i$ -in- $\Gamma$ cased [rule-format]*:

$\text{length } \text{assert} > 0$   
 $\longrightarrow \text{length } \text{assert} = \text{length } \text{alts}$   
 $\longrightarrow \text{def-disjointUnionEnv}$   
 $\quad (\text{foldl } \text{op } \otimes \text{empty})$

```

      (map (λ(Li, Γi). restrict-neg-map Γi (insert x (set Li)))
        (zip (map (snd ∘ extractP ∘ fst) alts) (map snd assert))))
    [x ↦ d'']
  → (∀ i < length alts. y ∈ dom (snd (assert ! i))
    → y ∉ set (snd (extractP (fst (alts ! i))))
    → y ∈ dom (foldl op ⊗ empty
      (map (λ(Li, Γi). restrict-neg-map Γi (insert x (set Li)))
        (zip (map (snd ∘ extractP ∘ fst) alts) (map snd assert)))) +
      [x ↦ d''])
  apply (induct assert alts rule:list-induct2',simp-all)
  apply (rule impI)+
  apply (case-tac xs = [],simp)
  apply (rule impI)
  apply (subst empty-nonDisjointUnionEnv)
  apply (subst union-dom-disjointUnionEnv)
  apply (subst (asm) empty-nonDisjointUnionEnv)
  apply simp
  apply (subst dom-restrict-neg-map)
  apply force
  apply simp
  apply (drule mp)
  apply (simp add: def-disjointUnionEnv-def)
  apply (subst (asm) dom-foldl-monotone-generic)
  apply blast
  apply (rule allI, rule impI)
  apply (case-tac i,simp-all)
  apply (rule impI)
  apply (subst dom-foldl-disjointUnionEnv-monotone-generic-2)
  apply (subst dom-restrict-neg-map)
  apply force
  apply (rule impI)
  apply (rotate-tac 3)
  apply (erule-tac x=nat in allE,simp)
  apply (subst dom-foldl-disjointUnionEnv-monotone-generic-2)
  apply (subst (asm) union-dom-disjointUnionEnv)
  apply (simp add: def-disjointUnionEnv-def)
  apply (subst (asm) dom-foldl-monotone-generic)
  apply blast
  by blast

```

**lemma** *P3-2-CASED*:

```

  [| length assert > 0;
    length assert = length alts;
    def-disjointUnionEnv
      (foldl op ⊗ empty
        (map (λ(Li, Γi). restrict-neg-map Γi (insert x (set Li)))
          (zip (map (snd ∘ extractP ∘ fst) alts) (map snd assert))))

```

$[x \mapsto d''];$   
 $\forall i < \text{length } \text{alts}. \forall j < \text{length } \text{alts}. i \neq j \longrightarrow (\text{fst } (\text{assert } ! i) \cap \text{set } (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! j)))) = \{\};$   
 $(\forall i < \text{length } \text{alts}. \text{fst } (\text{assert } ! i) \subseteq \text{dom } (\text{snd } (\text{assert } ! i))) \mathbb{I}$   
 $\implies (\bigcup_{i < \text{length } \text{alts}} \text{fst } (\text{assert } ! i) - \text{set } (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! i))))$   
 $\subseteq \text{dom } (\text{foldl } \text{op } \otimes \text{empty}$   
 $\quad (\text{map } (\lambda(Li, \Gamma i). \text{restrict-neg-map } \Gamma i (\text{insert } x (\text{set } Li)))$   
 $\quad (\text{zip } (\text{map } (\text{snd } \circ \text{extractP } \circ \text{fst}) \text{alts}) (\text{map } \text{snd } \text{assert}))) +$   
 $\quad [x \mapsto d'']$   
**apply** (rule subsetI,simp)  
**apply** (rename-tac y)  
**apply** (elim bexE, elim conjE)  
**apply** (erule-tac x=i in allE,simp)  
**apply** (subgoal-tac y ∈ dom (snd (assert ! i)))  
**prefer** 2 **apply** blast  
**apply** (rule dom-Γi-in-Γcased)  
**by** (simp,assumption+)

**lemma** union-dom-nonDisjointUnionSafeEnv:  
 $\text{dom } (\text{nonDisjointUnionSafeEnv } A \ B) = \text{dom } A \cup \text{dom } B$   
**apply** (simp add: nonDisjointUnionSafeEnv-def add: unionEnv-def,auto)  
**by** (split split-if-asm,simp-all)

**lemma** nonDisjointUnionSafeEnv-assoc:  
 $\text{nonDisjointUnionSafeEnv } (\text{nonDisjointUnionSafeEnv } G1 \ G2) \ G3 =$   
 $\text{nonDisjointUnionSafeEnv } G1 \ (\text{nonDisjointUnionSafeEnv } G2 \ G3)$   
**apply** (simp add: nonDisjointUnionSafeEnv-def add: unionEnv-def)  
**apply** (rule ext, auto)  
**apply** (split split-if-asm, simp, simp)  
**apply** (split split-if-asm, simp,simp)  
**by** (split split-if-asm, simp, simp add: dom-def)

**lemma** foldl-nonDisjointUnionSafeEnv-prop:  
 $\text{foldl } \text{nonDisjointUnionSafeEnv } (G' \oplus G) \ Gs = G' \oplus \text{foldl } \text{op } \oplus G \ Gs$   
**apply** (induct Gs arbitrary: G)  
**apply** simp  
**by** (simp-all add: nonDisjointUnionSafeEnv-assoc)

**lemma** nonDisjointUnionSafeEnv-commutative:  
 $\text{def-nonDisjointUnionSafeEnv } G \ G' \implies (G \oplus G') = (G' \oplus G)$   
**apply** (simp add: nonDisjointUnionSafeEnv-def add: unionEnv-def)  
**apply** (rule ext)



**apply** (*simp add: def-nonDisjointUnionSafeEnv-def*)  
**apply** (*simp add: safe-def*)  
**by** *clarsimp*

**lemma** *dom-foldl-nonDisjointUnionSafeEnv-monotone*:  
 $\text{dom } (\text{foldl } \text{nonDisjointUnionSafeEnv } (\text{empty} \oplus x) \text{ } xs) =$   
 $\text{dom } x \cup \text{dom } (\text{foldl } \text{op} \oplus \text{empty } xs)$   
**apply** (*subgoal-tac empty  $\oplus x = x \oplus \text{empty}$ , simp*)  
**apply** (*subgoal-tac foldl op  $\oplus (x \oplus \text{empty}) xs =$*   
 $x \oplus \text{foldl op} \oplus \text{empty } xs, \text{simp}$ )  
**apply** (*rule union-dom-nonDisjointUnionSafeEnv*)  
**apply** (*rule foldl-nonDisjointUnionSafeEnv-prop*)  
**apply** (*rule nonDisjointUnionSafeEnv-commutative*)  
**by** (*simp add: def-nonDisjointUnionSafeEnv-def*)

**lemma** *nonDisjointUnionSafeEnv-empty*:  
 $\text{nonDisjointUnionSafeEnv empty } x = x$   
**apply** (*simp add: nonDisjointUnionSafeEnv-def*)  
**by** (*simp add: unionEnv-def*)

**declare** *dom-fun-upd* [*simp del*]

**lemma** *dom-atom2var-fv*:  
 $(\exists y. x = \text{VarE } y \text{ unit})$   
 $\implies \text{dom } [\text{atom2var } x \mapsto y] = \text{fv } x$   
**apply** (*case-tac x*)  
**apply** (*simp-all add: atom2var.simps*)  
**by** (*simp add: dom-def*)

**declare** *nonDisjointUnionSafeEnvList.simps* [*simp del*]

**lemma** *atom-fvs-VarE*:  
 $\llbracket (\forall a \in \text{set } as. \text{atom } a); xa \in \text{fvs}' as \rrbracket$   
 $\implies (\exists i < \text{length } as. \exists a. as[i] = \text{VarE } xa a)$   
**apply** (*induct as, simp-all*)  
**apply** (*case-tac a, simp-all*)  
**by** *force*

**lemma** *nth-nonDisjointUnionSafeEnvList*:  
 $\llbracket \text{length } xs = \text{length } ms; \text{def-nonDisjointUnionSafeEnvList } (\text{maps-of } (\text{zip } xs \text{ } ms)) \rrbracket$   
 $\implies (\forall i < \text{length } xs. \text{nonDisjointUnionSafeEnvList } (\text{maps-of } (\text{zip } xs \text{ } ms))$   
 $(xs[i]) = \text{Some } (ms[i]))$

```

apply (induct xs ms rule: list-induct2',simp-all)
apply clarsimp
apply (case-tac i)
apply simp
apply (simp add: nonDisjointUnionSafeEnvList.simps)
apply (subgoal-tac empty  $\oplus [x \mapsto y] = [x \mapsto y] \oplus empty$ ,simp)
apply (subgoal-tac foldl op  $\oplus ([x \mapsto y] \oplus empty)$  (maps-of (zip xs ys)) =
       $[x \mapsto y] \oplus foldl op \oplus empty$  (maps-of (zip xs ys)),simp)
apply (simp add: nonDisjointUnionSafeEnv-def)
apply (simp add: unionEnv-def)
apply (simp add: dom-def)
apply (rule foldl-nonDisjointUnionSafeEnv-prop)
apply (subst nonDisjointUnionSafeEnv-empty)
apply (subst nonDisjointUnionSafeEnv-commutative)
apply (simp add: def-nonDisjointUnionSafeEnv-def)
apply (subst nonDisjointUnionSafeEnv-empty)
apply simp
apply clarsimp
apply (simp add: nonDisjointUnionSafeEnvList.simps)
apply (subgoal-tac empty  $\oplus [x \mapsto y] = [x \mapsto y] \oplus empty$ ,simp)
apply (subgoal-tac foldl op  $\oplus ([x \mapsto y] \oplus empty)$  (maps-of (zip xs ys)) =
       $[x \mapsto y] \oplus foldl op \oplus empty$  (maps-of (zip xs ys)),simp)
apply (simp add: Let-def)
apply (erule-tac x=nat in allE,simp)
apply (simp add: nonDisjointUnionSafeEnv-def)
apply (simp add: unionEnv-def)
apply (rule conjI)
apply (rule impI)+
apply (elim conjE)
apply (simp add: def-nonDisjointUnionSafeEnv-def)
apply (erule-tac x=x in ballE)
apply (simp add: safe-def)
apply (simp add: dom-def)
apply clarsimp
apply (rule foldl-nonDisjointUnionSafeEnv-prop)
apply (rule nonDisjointUnionSafeEnv-commutative)
by (simp add: def-nonDisjointUnionSafeEnv-def)

```

**lemma** *dom-nonDisjointUnionSafeEnvList-fvs:*

```

   $\llbracket \forall a \in \text{set } xs. \text{atom } a; \text{length } xs = \text{length } ys \rrbracket$ 
   $\implies \text{fvs}' xs \subseteq \text{dom} (\text{nonDisjointUnionSafeEnvList} (\text{maps-of} (\text{zip} (\text{map } \text{atom2var}$ 
  xs) ys)))
apply (induct xs ys rule: list-induct2',simp-all)
apply (simp add: nonDisjointUnionSafeEnvList.simps)
apply (subst dom-foldl-nonDisjointUnionSafeEnv-monotone)
apply (rule conjI)
apply (case-tac x, simp-all)
apply (simp add: dom-def)

```

```

apply (subst dom-foldl-nonDisjointUnionSafeEnv-monotone)
by blast

declare nonDisjointUnionSafeEnvList.simps [simp add]
declare def-nonDisjointUnionSafeEnvList.simps [simp add] thm dom-map-add
declare atom.simps [simp del]

lemma nonDisjointUnionSafeEnvList-prop1:
  
$$\llbracket \text{nonDisjointUnionSafeEnvList } (\text{maps-of } (\text{zip } (\text{map } \text{atom2var } \text{as}) \text{ms})) \subseteq_m \Gamma;$$

  
$$\text{xa} \in \text{fvs}' \text{ as}; \Gamma \text{ xa} = \text{Some } y;$$

  
$$\text{def-nonDisjointUnionSafeEnvList } (\text{maps-of } (\text{zip } (\text{map } \text{atom2var } \text{as}) \text{ms}));$$

  
$$(\forall a \in \text{set as. } \text{atom } a); \text{length as} = \text{length ms} \rrbracket$$

  
$$\implies \exists i < \text{length as. } \exists a. \text{as}!i = \text{VarE xa } a \wedge \text{ms}!i = y$$

apply (frule atom-fvs-VarE,assumption+)
apply (elim exE, elim conjE, elim exE)
apply (rule-tac x=i in exI,simp)
apply (simp add: map-le-def)
apply (erule-tac x=xa in ballE,simp)
apply (subgoal-tac length (map atom2var as) = length ms)
prefer 2 apply simp
apply (frule nth-nonDisjointUnionSafeEnvList,assumption+)
apply (erule-tac x=i in allE,simp)
apply (frule dom-nonDisjointUnionSafeEnvList-fvs,assumption+,simp)
by blast

lemma P3-APP:
  
$$\llbracket \text{length as} = \text{length ms}; \forall a \in \text{set as. } \text{atom } a;$$

  
$$\text{nonDisjointUnionSafeEnvList } (\text{maps-of } (\text{zip } (\text{map } \text{atom2var } \text{as}) \text{ms})) \subseteq_m \Gamma \rrbracket$$

  
$$\implies \text{fvs}' \text{ as} \subseteq \text{dom } \Gamma$$

apply (induct as ms rule: list-induct2',simp-all)
apply (elim conjE)
apply (frule map-le-implies-dom-le)
apply (frule dom-nonDisjointUnionSafeEnvList-fvs,assumption+)
apply (subgoal-tac
  
$$\text{dom } (\text{nonDisjointUnionSafeEnvList } ([\text{atom2var } x \mapsto y] \# \text{maps-of } (\text{zip } (\text{map } \text{atom2var } \text{xs}) \text{ys}))) =$$

  
$$\text{dom } [\text{atom2var } x \mapsto y] \cup \text{dom } (\text{nonDisjointUnionSafeEnvList } (\text{maps-of } (\text{zip } (\text{map } \text{atom2var } \text{xs}) \text{ys}))))$$

apply simp
apply (rule conjI)
apply (case-tac x,simp-all add: atom.simps)
apply force
apply force
by (rule dom-foldl-nonDisjointUnionSafeEnv-monotone)

```

**lemma** *P3-APP-PRIMOP*:  

$$\llbracket \Gamma 0 = [atom2var\ a1 \mapsto s'', atom2var\ a2 \mapsto s''];$$

$$[atom2var\ a1 \mapsto s'', atom2var\ a2 \mapsto s''] \subseteq_m \Gamma \rrbracket$$

$$\implies \{atom2var\ a1, atom2var\ a2\} \subseteq dom\ \Gamma$$
  
**apply** (*simp add: map-le-def*)  
**apply** (*rule conjI*)  
**apply** (*erule-tac x=atom2var a1 in ballE*)  
**apply** (*split split-if-asm*)  
**apply** (*drule-tac t=Γ (atom2var a1) in sym,force*)  
**apply** (*drule-tac t=Γ (atom2var a1) in sym, simp add: dom-def*)  
**apply** (*simp add: dom-def*)  
**apply** (*split split-if-asm,simp,simp*)  
**apply** (*erule-tac x=atom2var a2 in ballE*)  
**apply** (*split split-if-asm*)  
**apply** (*drule-tac t=Γ (atom2var a2) in sym,force*)  
**apply** (*drule-tac t=Γ (atom2var a2) in sym, simp add: dom-def*)  
**by** (*simp add: dom-def*)

**end**

## 13 Derived Assertions. P1. Semantic

**theory** *SafeDAss-P1* **imports** *SafeDAssBasic*  
**begin**

Lemmas for REUSE

**lemma** *P1-REUSE*:  

$$\llbracket (E1, E2) \vdash h, k, td, (ReuseE\ x\ a) \Downarrow hh, k, v, r \rrbracket$$

$$\implies \exists\ p\ q\ c.\ E1\ x = Some\ (Loc\ p)$$

$$\wedge\ h\ p = Some\ c$$

$$\wedge\ fresh\ q\ h$$

$$\wedge\ hh = (h(p:=None))(q \mapsto c)$$

$$\wedge\ v = Loc\ q$$
  
**apply** (*ind-cases (E1, E2) ⊢ h, k, td, (ReuseE x a) ↓ hh, k, v, r*)  
**by** *force*

Lemmas for COPY

**lemma** *P1-COPY*:  

$$\llbracket (E1, E2) \vdash h, k, td, x @ r\ a \Downarrow hh, k, v, rs \rrbracket$$

$$\implies \exists\ p\ p'\ j.\ E1\ x = Some\ (Loc\ p)$$

$$\wedge\ E2\ r = Some\ j$$

$$\wedge\ j \leq k$$

$$\wedge\ copy\ (h, k)\ p\ j = ((hh, k), p')$$

$$\wedge\ def-copy\ p\ (h, k)$$

$$\wedge\ v = Loc\ p'$$
  
**apply** (*ind-cases (E1, E2) ⊢ h, k, td, x @ r a ↓ hh, k, v, rs*)

by force

Lemmas for LET1 and LET2

**lemma P1-LET:**

$\llbracket \forall C \text{ as } r \ a'. \ e1 \neq \text{ConstrE } C \text{ as } r \ a';$   
 $(E1, E2) \vdash h, k, td, \text{Let } x1 = e1 \text{ In } e2 \ a \Downarrow hh, k, v2, r' \rrbracket$   
 $\implies \exists h' \ v1 \ r'' \ r'''. (E1, E2) \vdash h, k, 0, e1 \Downarrow h', k, v1, r''$   
 $\wedge (E1(x1 \mapsto v1), E2) \vdash h', k, (td+1), e2 \Downarrow hh, k, v2, r'''$   
 $\wedge x1 \notin \text{dom } E1$   
**apply** (ind-cases (E1, E2)  $\vdash h, k, td, \text{Let } x1 = e1 \text{ In } e2 \ a \Downarrow hh, k, v2, r'$ )  
**prefer** 2  
**apply** (erule-tac x=C in allE)  
**apply** (erule-tac x=as in allE)  
**apply** (erule-tac x=r in allE)  
**apply** (erule-tac x=a' in allE)  
**apply** simp  
**apply** (rule-tac x=h' in exI)  
**apply** (rule-tac x=v1 in exI)  
**apply** (rule-tac x=( $\delta 1, m1, s1$ ) in exI)  
**apply** (rule-tac x=( $\delta 2, m2, s2$ ) in exI)  
**by** simp

**lemma P1-f-n-LET:**

$\llbracket \forall C \text{ as } r \ a'. \ e1 \neq \text{ConstrE } C \text{ as } r \ a';$   
 $(E1, E2) \vdash h, k, \text{Let } x1 = e1 \text{ In } e2 \ a \Downarrow (f, n) \ h'', k, v2 \rrbracket$   
 $\implies \exists h' \ v1.$   
 $(E1, E2) \vdash h, k, e1 \Downarrow (f, n) \ h', k, v1$   
 $\wedge (E1(x1 \mapsto v1), E2) \vdash h', k, e2 \Downarrow (f, n) \ h'', k, v2$   
 $\wedge x1 \notin \text{dom } E1$   
**apply** (simp add: SafeBoundSem-def)  
**apply** (elim exE, elim conjE)  
**apply** (erule SafeDepthSem.cases, simp-all)  
**apply** (elim conjE)  
**apply** (rule-tac x=h' in exI)  
**apply** (rule-tac x=v1 in exI)  
**apply** (rule conjI)  
**apply** (rule-tac x=n1 in exI, simp)  
**apply** (rule-tac x=n2 in exI, simp)  
**done**

**lemma P1-LETC:**

$\llbracket (E1, E2) \vdash h, k, td, \text{Let } x1 = \text{ConstrE } C \text{ as } r \ a' \text{ In } e2 \ a \Downarrow hh, k, v, rs \rrbracket$   
 $\implies \exists rs' \ p \ j.$   
 $(E1(x1 \mapsto \text{Val.Loc } p), E2) \vdash h(p \mapsto (j, (C, \text{map } (\text{atom2val } E1) \ as))), k,$   
 $(td+1), e2 \Downarrow hh, k, v, rs'$   
 $\wedge x1 \notin \text{dom } E1$   
 $\wedge \text{fresh } p \ h$

$\wedge E2\ r = \text{Some } j$   
 $\wedge j \leq k$   
 $\wedge r \neq \text{self}$   
**apply** (*ind-cases* ( $E1, E2$ )  $\vdash h, k, td, \text{Let } x1 = \text{ConstrE } C \text{ as } r\ a' \text{ In } e2\ a \Downarrow$   
 $hh, k, v, rs$ )  
**by** *force+*

**lemma** *P1-f-n-LETC*:

$\llbracket (E1, E2) \vdash h, k, \text{Let } x1 = \text{ConstrE } C \text{ as } r\ a' \text{ In } e2\ a \Downarrow (f, n)\ hh, k, v \rrbracket$   
 $\implies \exists rs' p j.$   
 $(E1(x1 \mapsto \text{Loc } p), E2) \vdash h(p \mapsto (j, C, \text{map } (\text{atom2val } E1) \text{ as})) , k, e2$   
 $\Downarrow (f, n)\ hh, k, v$   
 $\wedge x1 \notin \text{dom } E1$   
 $\wedge \text{fresh } p\ h$   
 $\wedge E2\ r = \text{Some } j$   
 $\wedge j \leq k$   
 $\wedge r \neq \text{self}$   
**apply** (*simp add: SafeBoundSem-def*)  
**apply** (*elim exE, elim conjE*)  
**apply** (*erule SafeDepthSem.cases, simp-all*)  
**apply** *force*  
**apply** *force*  
**done**

Lemmas for CASE

**lemma** *P1-CASE*:

$\llbracket E1\ x = \text{Some } (\text{Val.Loc } p);$   
 $(E1, E2) \vdash h, k, td, \text{Case } (\text{VarE } x\ a)\ \text{Of alts } a \Downarrow h', k, v, r \rrbracket$   
 $\implies \exists j\ C\ vs. h\ p = \text{Some } (j, C, vs) \wedge$   
 $(\exists i < \text{length alts}. \exists td\ r.$   
 $\text{def-extend } E1\ (\text{snd } (\text{extractP } (\text{fst } (\text{alts } !\ i))))\ vs$   
 $\wedge (\text{extend } E1\ (\text{snd } (\text{extractP } (\text{fst } (\text{alts } !\ i))))\ vs, E2) \vdash h, k, td, \text{snd } (\text{alts}$   
 $! i) \Downarrow h', k, v, r)$   
**apply** (*ind-cases* ( $E1, E2$ )  $\vdash h, k, td, \text{Case } (\text{VarE } x\ a)\ \text{Of alts } a \Downarrow h', k, v, r, \text{clarsimp}$ )  
**apply** (*rule-tac x=i in exI, force*)  
**by** (*simp-all*)

**lemma** *P1-CASE-1-1*:

$\llbracket E1\ x = \text{Some } (\text{IntT } n);$   
 $(E1, E2) \vdash h, k, td, \text{Case } (\text{VarE } x\ a)\ \text{Of alts } a' \Downarrow hh, k, v, r \rrbracket$   
 $\implies (\exists i < \text{length alts}.$   
 $(\exists td\ r. (E1, E2) \vdash h, k, td, \text{snd } (\text{alts } !\ i) \Downarrow hh, k, v, r$   
 $\wedge \text{fst } (\text{alts } !\ i) = \text{ConstP } (\text{LitN } n)))$   
**apply** (*ind-cases* ( $E1, E2$ )  $\vdash h, k, td, \text{Case } (\text{VarE } x\ a)\ \text{Of alts } a' \Downarrow hh, k, v, r, \text{clarsimp}$ )  
**apply** (*rule-tac x=i in exI, force*)  
**by** (*simp-all*)

**lemma** *P1-CASE-1-2*:

$\llbracket E1\ x = \text{Some}\ (\text{BoolT}\ b);$   
 $(E1, E2) \vdash h, k, td, \text{Case}\ (\text{VarE}\ x\ a)\ \text{Of}\ \text{alts}\ a' \Downarrow hh, k, v, r \rrbracket$   
 $\implies (\exists\ i < \text{length}\ \text{alts}.$   
 $\quad \exists\ td\ r. (E1, E2) \vdash h, k, td, \text{snd}\ (\text{alts}\ !\ i) \Downarrow hh, k, v, r$   
 $\quad \wedge\ \text{fst}\ (\text{alts}\ !\ i) = \text{ConstP}\ (\text{LitB}\ b))$   
**apply** (*ind-cases* ( $E1, E2$ )  $\vdash h, k, td, \text{Case}\ (\text{VarE}\ x\ a)\ \text{Of}\ \text{alts}\ a' \Downarrow hh, k, v, r, \text{clarsimp}$ )  
**apply** (*rule-tac*  $x=i$  **in** *exI, force*)  
**by** *force*

**lemma** *P1-f-n-CASE*:

$\llbracket E1\ x = \text{Some}\ (\text{Val.Loc}\ p);$   
 $(E1, E2) \vdash h, k, \text{Case}\ \text{VarE}\ x\ a\ \text{Of}\ \text{alts}\ a' \Downarrow (f, n)\ hh, k, v \rrbracket$   
 $\implies \exists\ j\ C\ vs. h\ p = \text{Some}\ (j, C, vs) \wedge$   
 $\quad (\exists\ i < \text{length}\ \text{alts}.$   
 $\quad \text{def-extend}\ E1\ (\text{snd}\ (\text{extractP}\ (\text{fst}\ (\text{alts}\ !\ i))))\ vs$   
 $\quad \wedge\ (\text{extend}\ E1\ (\text{snd}\ (\text{extractP}\ (\text{fst}\ (\text{alts}\ !\ i))))\ vs, E2) \vdash h, k, \text{snd}\ (\text{alts}\ !\ i)$   
 $\Downarrow (f, n)\ hh, k, v)$   
**apply** (*simp add: SafeBoundSem-def*)  
**apply** (*elim exE, elim conjE*)  
**apply** (*erule SafeDepthSem.cases, simp-all*)  
**by** *force*

**lemma** *P1-f-n-CASE-1-1*:

$\llbracket E1\ x = \text{Some}\ (\text{IntT}\ n');$   
 $(E1, E2) \vdash h, k, \text{Case}\ \text{VarE}\ x\ a\ \text{Of}\ \text{alts}\ a' \Downarrow (f, n)\ hh, k, v \rrbracket$   
 $\implies (\exists\ i < \text{length}\ \text{alts}.$   
 $\quad ((E1, E2) \vdash h, k, \text{snd}\ (\text{alts}\ !\ i) \Downarrow (f, n)\ hh, k, v$   
 $\quad \wedge\ \text{fst}\ (\text{alts}\ !\ i) = \text{ConstP}\ (\text{LitN}\ n'))$   
**apply** (*simp add: SafeBoundSem-def*)  
**apply** (*elim exE, elim conjE*)  
**apply** (*erule SafeDepthSem.cases, simp-all*)  
**by** *force*

**lemma** *P1-f-n-CASE-1-2*:

$\llbracket E1\ x = \text{Some}\ (\text{BoolT}\ b);$   
 $(E1, E2) \vdash h, k, \text{Case}\ \text{VarE}\ x\ a\ \text{Of}\ \text{alts}\ a' \Downarrow (f, n)\ hh, k, v \rrbracket$   
 $\implies (\exists\ i < \text{length}\ \text{alts}.$   
 $\quad (E1, E2) \vdash h, k, \text{snd}\ (\text{alts}\ !\ i) \Downarrow (f, n)\ hh, k, v$   
 $\quad \wedge\ \text{fst}\ (\text{alts}\ !\ i) = \text{ConstP}\ (\text{LitB}\ b))$   
**apply** (*simp add: SafeBoundSem-def*)  
**apply** (*elim exE, elim conjE*)  
**apply** (*erule SafeDepthSem.cases, simp-all*)  
**by** *force*

Lemmas for CASED

**lemma** *P1-CASED*:

[[ $(E1, E2) \vdash h, k, td, CaseD (VarE x a) Of alts a' \Downarrow hh, kk, v, r$  ]]  
 $\implies \exists p j C vs. E1 x = Some (Loc p) \wedge h p = Some (j, C, vs) \wedge$   
 $(\exists i < length alts. \exists td r.$   
 $def-extend E1 (snd (extractP (fst (alts ! i)))) vs$   
 $\wedge (extend E1 (snd (extractP (fst (alts ! i)))) vs, E2) \vdash h(p := None), k,$   
 $td, snd (alts ! i) \Downarrow hh, k, v, r)$   
**apply** (*ind-cases* ( $E1, E2$ )  $\vdash h, k, td, CaseD (VarE x a) Of alts a' \Downarrow hh, kk, v, r, clarsimp$ )  
**by** (*rule-tac*  $x=i$  **in**  $exI, force$ )

**lemma** *P1-f-n-CASED*:

[[ $(E1, E2) \vdash h, k, CaseD (VarE x a) Of alts a' \Downarrow (f, n) hh, kk, v$  ]]  
 $\implies \exists p j C vs. E1 x = Some (Loc p) \wedge h p = Some (j, C, vs) \wedge$   
 $(\exists i < length alts.$   
 $def-extend E1 (snd (extractP (fst (alts ! i)))) vs$   
 $\wedge (extend E1 (snd (extractP (fst (alts ! i)))) vs, E2) \vdash h(p := None), k,$   
 $snd (alts ! i) \Downarrow (f, n) hh, k, v)$   
**apply** (*simp add: SafeBoundSem-def*)  
**apply** (*elim conjE, elim exE, elim conjE*)  
**apply** (*erule SafeDepthSem.cases, simp-all*)  
**by** *force*

Lemmas for APP

**lemma** *P1-APP*:

[[ $(E1, E2) \vdash h, k, td, AppE f as rs' a \Downarrow hh, k, v, r; primops f = None;$   
 $\Sigma f f = Some (xs, rs, ef)$  ]]  
 $\implies \exists h' \delta m s.$   
 $(map-of (zip xs (map (atom2val E1) as)), map-of (zip rs (map (the \circ E2)$   
 $rs'))(self \mapsto Suc k)) \vdash$   
 $h, Suc k, (length as + length rs), ef \Downarrow h', Suc k, v, (\delta, m, s)$   
 $\wedge length xs = length as$   
 $\wedge distinct xs$   
 $\wedge length rs = length rs'$   
 $\wedge distinct rs$   
 $\wedge hh = h' | \{p. p \in dom h' \& fst (the (h' p)) \leq k\}$   
 $\wedge dom E1 \cap set xs = \{\}$   
**apply** (*ind-cases* ( $E1, E2$ )  $\vdash h, k, td, AppE f as rs' a \Downarrow hh, k, v, r, clarsimp$ )  
**apply** (*rule-tac*  $x=h'$  **in**  $exI$ )  
**apply** (*rule-tac*  $x=\delta$  **in**  $exI$ )  
**apply** (*rule-tac*  $x=m$  **in**  $exI$ )  
**apply** (*rule-tac*  $x=s$  **in**  $exI$ )  
**by** (*rule conjI, simp, simp*)

**lemma** *P1-f-n-APP*:

[[ $(E1, E2) \vdash h, k, AppE f as rs' a \Downarrow (f, n) hh, k, v; primops f = None;$



$\Sigma f f = \text{Some } (xs, rs, ef) \parallel$   
 $\implies \exists h'.$   
 $(\text{map-of } (\text{zip } xs \ (\text{map } (\text{atom2val } E1) \ as)), \text{map-of } (\text{zip } rs \ (\text{map } (\text{the } \circ E2)$   
 $rs'))(\text{self} \mapsto \text{Suc } k)) \vdash$   
 $h, \text{Suc } k, ef \Downarrow(f,n) h', \text{Suc } k, v$   
 $\wedge \text{length } xs = \text{length } as$   
 $\wedge \text{distinct } xs$   
 $\wedge \text{length } rs = \text{length } rs'$   
 $\wedge \text{distinct } rs$   
 $\wedge hh = h' \mid \{p. p \in \text{dom } h' \ \& \ \text{fst } (\text{the } (h' p)) \leq k\}$   
 $\wedge \text{dom } E1 \cap \text{set } xs = \{\}$   
 $\wedge n > 0$   
**apply** (*simp add: SafeBoundSem-def*)  
**apply** (*elim exE, elim conjE*)  
**apply** (*erule SafeDepthSem.cases, simp-all*)  
**apply** *clarsimp*  
**by** *force*

**lemma** *P1-f-n-ge-0-APP:*

$\parallel (E1, E2) \vdash h, k, \text{AppE } f \text{ as } rs' \ a \Downarrow(f, \text{Suc } n) hh, k, v; \text{primops } f = \text{None};$   
 $\Sigma f f = \text{Some } (xs, rs, ef) \parallel$   
 $\implies \exists h'.$   
 $(\text{map-of } (\text{zip } xs \ (\text{map } (\text{atom2val } E1) \ as)), \text{map-of } (\text{zip } rs \ (\text{map } (\text{the } \circ E2)$   
 $rs'))(\text{self} \mapsto \text{Suc } k)) \vdash$   
 $h, \text{Suc } k, ef \Downarrow(f,n) h', \text{Suc } k, v$   
 $\wedge \text{length } xs = \text{length } as$   
 $\wedge \text{distinct } xs$   
 $\wedge \text{length } rs = \text{length } rs'$   
 $\wedge \text{distinct } rs$   
 $\wedge hh = h' \mid \{p. p \in \text{dom } h' \ \& \ \text{fst } (\text{the } (h' p)) \leq k\}$   
 $\wedge \text{dom } E1 \cap \text{set } xs = \{\}$   
**apply** (*simp add: SafeBoundSem-def*)  
**apply** (*elim exE, elim conjE*)  
**apply** (*erule SafeDepthSem.cases, simp-all*)  
**apply** *clarsimp*  
**by** *auto*

**lemma** *P1-f-n-APP-2:*

$\parallel (E1, E2) \vdash h, k, \text{AppE } g \text{ as } rs' \ a \Downarrow(f,n) hh, k, v; \text{primops } g = \text{None}; f \neq g;$   
 $\Sigma f g = \text{Some } (xs, rs, ef) \parallel$   
 $\implies \exists h'.$   
 $(\text{map-of } (\text{zip } xs \ (\text{map } (\text{atom2val } E1) \ as)), \text{map-of } (\text{zip } rs \ (\text{map } (\text{the } \circ E2)$   
 $rs'))(\text{self} \mapsto \text{Suc } k)) \vdash$   
 $h, \text{Suc } k, ef \Downarrow(f,n) h', \text{Suc } k, v$   
 $\wedge \text{length } xs = \text{length } as$   
 $\wedge \text{distinct } xs$   
 $\wedge \text{length } rs = \text{length } rs'$

```

    ∧ distinct rs
    ∧ hh = h' | ' {p. p ∈ dom h' & fst (the (h' p)) ≤ k}
    ∧ dom E1 ∩ set xs = {}
  apply (simp add: SafeBoundSem-def)
  apply (elim exE, elim conjE)
  apply (erule SafeDepthSem.cases, simp-all)
  apply clarsimp
  by force

end

```

## 14 Region Definitions

```

theory SafeRegion-definitions imports SafeRASemantics
                                   SafeDepthSemantics
                                   ../SafeImp/ClosureHeap

begin

types RegionTypeVariable = string

constdefs
  ρself :: string
  ρself ≡ "rho-self"

types VarType = string

datatype TypeExpression = VarT VarType
  | ConstrT string TypeExpression list VarType list

types TypeMapping = (string → TypeExpression)
types RegMapping = (string → string)

types ThetaMapping = TypeMapping × RegMapping

types InstantiationMapping = VarType → nat

```

**types**  $TypeMu = (string \multimap TypeExpression) \times (string \multimap string)$

**consts**  $mu-ext :: TypeMu \Rightarrow TypeExpression \Rightarrow TypeExpression$   
 $mu-exts :: TypeMu \Rightarrow TypeExpression\ list \Rightarrow TypeExpression\ list$

**primrec**

$mu-ext\ \mu\ (VarT\ a) = the\ ((fst\ \mu)\ a)$   
 $mu-ext\ \mu\ (ConstrT\ T\ tm\ \varrho s) = (ConstrT\ T\ (mu-exts\ \mu\ tm)\ (map\ (the\ \circ\ (snd\ \mu))\ \varrho s))$

$mu-exts\ \mu\ [] = []$   
 $mu-exts\ \mu\ (x\#\!xs) = mu-ext\ \mu\ x\ \#\! (mu-exts\ \mu\ xs)$

**fun**  $atoms :: ('a\ Exp)\ list \Rightarrow bool$

**where**

$atoms\ as = (\forall\ i < length\ as.\ (\exists\ c\ a.\ as!\!i = ConstE\ (LitN\ c)\ a) \vee$   
 $(\exists\ b\ a.\ as!\!i = ConstE\ (LitB\ b)\ a) \vee$   
 $(\exists\ x\ a.\ as!\!i = VarE\ x\ a))$

**fun**  $argP-aux :: (string \multimap TypeExpression) \Rightarrow 'a\ Exp \Rightarrow TypeExpression \Rightarrow bool$

**where**

$argP-aux\ \vartheta\ (ConstE\ (LitN\ -)\ -)\ t = (t = (ConstrT\ intType\ []\ []))$   
 $| argP-aux\ \vartheta\ (ConstE\ (LitB\ -)\ -)\ t = (t = (ConstrT\ boolType\ []\ []))$   
 $| argP-aux\ \vartheta\ (VarE\ x\ -)\ t = (\vartheta\ x = Some\ t)$

**fun**

$argP ::$

$TypeExpression\ list \Rightarrow VarType \Rightarrow ('a\ Exp)\ list \Rightarrow RegVar \Rightarrow ThetaMapping \Rightarrow$   
 $bool$

**where**

$argP\ ti\ \varrho\ as\ r\ (\vartheta 1, \vartheta 2) = ($   
 $\quad length\ ti = length\ as \wedge$   
 $\quad atoms\ as \wedge$   
 $\quad (\forall\ i < length\ as.\ argP-aux\ \vartheta 1\ (as!\!i)\ (ti!\!i)) \wedge$   
 $\quad \vartheta 2\ r = Some\ \varrho)$

**types**

$ConstructorSignatureFun = string \multimap TypeExpression\ list \times VarType \times TypeExpression$

**consts**  $constructorSignature :: ConstructorSignatureFun$

**constdefs** *coherentC* :: *Constructor*  $\Rightarrow$  *bool*  
*coherentC* *C*  $\equiv$   
 (let (*nargs*,*n*,*largs*) = the (*ConstructorTable* *C*);  
   (*ts*,*ql*,*t*) = the (*constructorSignature* *C*)  
 in length *ts* = length *largs*  $\wedge$   
   ( $\exists$  *T tm qs*. *t* = *ConstrT* *T tm qs*)  $\wedge$   
   ( $\forall$  *i* < length *ts*. (*ts*!*i* = *ConstrT* *intType* [] []  $\longrightarrow$   
     (*snd* (*snd* (the (*ConstructorTable* *C*))))!*i* = *IntArg*)  
    $\wedge$  (*ts*!*i* = *ConstrT* *boolType* [] []  $\longrightarrow$   
     (*snd* (*snd* (the (*ConstructorTable* *C*))))!*i* = *BoolArg*)  
    $\wedge$  (( $\exists$  *T' tm' qs'*. (*ts*!*i* = *ConstrT* *T' tm' qs'*  $\wedge$  *ts*!*i*  $\neq$  *t*)  $\vee$   
     ( $\exists$  *a*. *ts*!*i* = *VarT* *a*))  $\longrightarrow$   
     (*snd* (*snd* (the (*ConstructorTable* *C*))))!*i* = *NonRecursive*)  
    $\wedge$  (*ts*!*i* = *t*  $\longrightarrow$  (*snd* (*snd* (the (*ConstructorTable* *C*))))!*i* = *Recursive*))))  
  
**constdefs** *coherent* :: *ConstructorSignatureFun*  $\Rightarrow$  *ConstructorTableFun*  $\Rightarrow$  *bool*  
*coherent*  $\Gamma$ *c* *Tc*  $\equiv$  dom  $\Gamma$ *c* = dom *Tc*  $\wedge$  ( $\forall$  *C*  $\in$  dom  $\Gamma$ *c*. *coherentC* *C*)

#### definition

*map-f-comp* :: (*'b*  $\Rightarrow$  *'c*)  $\Rightarrow$  (*'a*  $\leadsto$  *'b*)  $\Rightarrow$  (*'a*  $\leadsto$  *'c*) **where**  
*map-f-comp* *f g* = ( $\lambda$  *k*. case *g k* of *None*  $\Rightarrow$  *None* | *Some v*  $\Rightarrow$  *Some (f v)*)

#### notation (*xsymbols*)

*map-f-comp* (**infixl**  $\circ_f$  55)

#### fun

*argP-app* ::  
*TypeExpression* *list*  $\Rightarrow$  *RegVar* *list*  $\Rightarrow$  (*'a* *Exp*) *list*  $\Rightarrow$  *RegVar* *list*  $\Rightarrow$  *ThetaMapping*  $\Rightarrow$  *bool*

#### where

*argP-app* *ti qs as rs* ( $\vartheta 1, \vartheta 2$ ) = (  
   length *ti* = length *as*  $\wedge$   
   length *qs* = length *rs*  $\wedge$   
   atoms *as*  $\wedge$   
   ( $\forall$  *i* < length *as*. *argP-aux*  $\vartheta 1$  (*as*!*i*) (*ti*!*i*))  $\wedge$   
   ( $\forall$  *i* < length *rs*.  $\vartheta 2$  (*rs*!*i*) = *Some (qs*!*i*)))

**declare** *argP-app.simps* [*simp del*]

**consts** *functionSignature* :: *string*  $\rightarrow$  *TypeExpression* *list*  $\times$  *VarType* *list*  $\times$  *TypeExpression*

**consts** *regions* :: *TypeExpression*  $\Rightarrow$  *string set*  
*regions'* :: *TypeExpression list*  $\Rightarrow$  *string set*

**primrec**

*regions* (*VarT* *a*) = {}  
*regions* (*ConstrT* *T tm* *qs*) = (*regions'* *tm*)  $\cup$  *set qs*

*regions'* [] = {}  
*regions'* (*t#ts*) = *regions t*  $\cup$  *regions' ts*

**constdefs** *regionV* :: *HeapMap*  $\Rightarrow$  *Location*  $\Rightarrow$  *nat*  
*regionV h p*  $\equiv$  (*case h p of Some (j,C,vs)*  $\Rightarrow$  *j*)

**constdefs** *regionsV* :: *HeapMap*  $\Rightarrow$  *Location set*  $\Rightarrow$  *nat set*  
*regionsV h ps*  $\equiv$   $\bigcup p \in ps. \{regionV h p\}$

**consts** *variables* :: *TypeExpression*  $\Rightarrow$  *string set*  
*variables'* :: *TypeExpression list*  $\Rightarrow$  *string set*

**primrec**

*variables* (*VarT* *a*) = {*a*}  
*variables* (*ConstrT* *T tm* *qs*) = (*variables'* *tm*)

*variables'* [] = {}  
*variables'* (*t#ts*) = *variables t*  $\cup$  *variables' ts*

**fun**

*wellT* :: *TypeExpression list*  $\Rightarrow$  *VarType*  $\Rightarrow$  *TypeExpression*  $\Rightarrow$  *bool*

**where**

*wellT tn* *q* (*ConstrT T tm qs*) =  
 ((*length qs* > 0  $\wedge$  *q* = *last qs*  $\wedge$  *distinct qs*  $\wedge$  *last qs*  $\notin$  *regions' tm*)  
 $\wedge$  ( $\forall i < \text{length } tn. \text{regions } (tn!i) \subseteq \text{regions } (\text{ConstrT } T \text{ tm } qs) \wedge$   
*variables* (*tn!i*)  $\subseteq$  *variables* (*ConstrT T tm qs*)))

**constdefs**

*ofake* :: *string*  
*ofake*  $\equiv$  "rho-fake"

**constdefs** *q-ren* :: *string*  $\Rightarrow$  *string*

*q-ren q*  $\equiv$  (*if q* = *qself* *then ofake* *else q*)

```

consts t-ren :: TypeExpression  $\Rightarrow$  TypeExpression
         t-rens :: TypeExpression list  $\Rightarrow$  TypeExpression list
primrec
  t-ren (VarT a) = (VarT a)
  t-ren (ConstrT T tm qs) = ConstrT T (t-rens tm) (map q-ren qs)

  t-rens [] = []
  t-rens (x#xs) = t-ren x # (t-rens xs)

constdefs q-ren-inv :: string  $\Rightarrow$  string
  q-ren-inv q  $\equiv$  (if q = qfake then qself else q)

consts t-ren-inv :: TypeExpression  $\Rightarrow$  TypeExpression
         t-ren-invs :: TypeExpression list  $\Rightarrow$  TypeExpression list
primrec
  t-ren-inv (VarT a) = (VarT a)
  t-ren-inv (ConstrT T ts rs) = ConstrT T (t-ren-invs ts) (map q-ren-inv rs)

  t-ren-invs [] = []
  t-ren-invs (t#tm) = (t-ren-inv t) # t-ren-invs tm

consts notFake :: TypeExpression  $\Rightarrow$  bool
         notFakes :: TypeExpression list  $\Rightarrow$  bool
primrec
  notFake (VarT a) = (a  $\neq$  qfake)
  notFake (ConstrT T tm qs) = ((notFakes tm)  $\wedge$  ( $\forall$  q  $\in$  set qs. q  $\neq$  qfake))

  notFakes [] = True
  notFakes (x#xs) = (notFake x  $\wedge$  notFakes xs)

constdefs mu-ext-def :: TypeMu  $\Rightarrow$  TypeExpression  $\Rightarrow$  bool
  mu-ext-def  $\mu$  t  $\equiv$  notFake (mu-ext  $\mu$  t)

consts mu-exts-def :: TypeMu  $\Rightarrow$  TypeExpression list  $\Rightarrow$  bool
primrec
  mu-exts-def  $\mu$  [] = True
  mu-exts-def  $\mu$  (x#xs) = (mu-ext-def  $\mu$  x  $\wedge$  mu-exts-def  $\mu$  xs)

fun  $\mu$ -ren :: TypeMu  $\Rightarrow$  TypeMu
where

```

$$\mu\text{-ren } (\mu 1, \mu 2) = (\lambda x. \text{Some } (t\text{-ren } (the (\mu 1 x))), \\ \lambda \varrho. \text{Some } (\varrho\text{-ren } (the (\mu 2 \varrho))))$$

**constdefs**  $\eta\text{-ren} :: \text{InstantiationMapping} \Rightarrow \text{InstantiationMapping}$   
 $\eta\text{-ren } \eta \equiv (\lambda x. \text{if } x = \varrho\text{self} \text{ then None else } \eta x) ++$   
 $(\text{if } (\varrho\text{self} \in \text{dom } \eta) \text{ then } [\varrho\text{fake} \mapsto the (\eta \varrho\text{self})] \text{ else empty})$

**inductive**

$\text{consistent-v} :: [\text{TypeExpression}, \text{InstantiationMapping}, \text{Val}, \text{HeapMap}] \Rightarrow \text{bool}$   
**where**  
 $\text{primitiveI} \quad : \text{consistent-v } (\text{ConstrT intType } [] []) \eta (\text{IntT } i) h$   
 $\text{primitiveB} \quad : \text{consistent-v } (\text{ConstrT boolType } [] []) \eta (\text{BoolT } b) h$   
 $\text{variable} \quad : \text{consistent-v } (\text{VarT } a) \eta v h$   
 $\text{algebraic-None} : p \notin \text{dom } h \implies \text{consistent-v } t \eta (\text{Loc } p) h$   
 $\text{algebraic} \quad : \llbracket h p = \text{Some } (j, C, vn);$   
 $\quad \varrho l = \text{last } \varrho s;$   
 $\quad \varrho l \in \text{dom } \eta; \eta (\varrho l) = \text{Some } j;$   
 $\quad \text{constructorSignature } C = \text{Some } (tn', \varrho', \text{ConstrT } T \text{ tm}' \varrho s');$   
 $\quad \text{wellT } tn' (\text{last } \varrho s') (\text{TypeExpression. ConstrT } T \text{ tm}' \varrho s');$   
 $\quad \text{length } vn = \text{length } tn';$   
 $\quad \exists \mu 1 \mu 2. (((the (\mu 2 (\text{last } \varrho s'))), \mu\text{-ext } (\mu 1, \mu 2) (\text{ConstrT } T$   
 $\text{tm}' \varrho s')) =$   
 $\quad (\varrho l, \text{ConstrT } T \text{ tm } \varrho s) \wedge$   
 $\quad (\forall i < \text{length } vn. \text{consistent-v } ((\text{map } (\mu\text{-ext } (\mu 1, \mu 2)) tn')!i) \eta$   
 $(vn!i) h)) \rrbracket$   
 $\implies \text{consistent-v } (\text{ConstrT } T \text{ tm } \varrho s) \eta (\text{Loc } p) h$

**fun**

$\text{consistent} :: \text{ThetaMapping} \Rightarrow \text{InstantiationMapping} \Rightarrow \text{Environment} \Rightarrow \text{HeapMap}$   
 $\Rightarrow \text{bool}$

**where**

$\text{consistent } (\vartheta 1, \vartheta 2) \eta (E1, E2) h =$   
 $((\forall x \in \text{dom } E1. \exists t v. \vartheta 1 x = \text{Some } t$   
 $\quad \wedge E1 x = \text{Some } v$   
 $\quad \wedge \text{consistent-v } t \eta v h)$   
 $\wedge (\forall r \in \text{dom } E2. \exists r' r''. \vartheta 2 r = \text{Some } r'$   
 $\quad \wedge \eta r' = \text{Some } r''$   
 $\quad \wedge E2 r = \text{Some } r''))$   
 $\wedge \text{self} \in \text{dom } E2$   
 $\wedge \vartheta 2 \text{self} = \text{Some } \varrho\text{self})$

**constdefs**

*admissible* :: *InstantiationMapping*  $\Rightarrow$  *nat*  $\Rightarrow$  *bool*

*admissible*  $\eta$   $k \equiv$

$\varrho\text{self} \in \text{dom } \eta \wedge$

$(\forall \varrho \in \text{dom } \eta.$

$\exists k'.$

$\eta \varrho = \text{Some } k' \wedge$

$(\varrho = \varrho\text{self} \longrightarrow k' = k) \wedge$

$(\varrho \neq \varrho\text{self} \longrightarrow k' < k))$

**fun** *extend-heaps* :: *Heap*  $\Rightarrow$  *Heap*  $\Rightarrow$  *bool* ( $- \sqsubseteq -$  1000)

**where**

$(h,k) \sqsubseteq (h',k') = (\forall p \in \text{dom } h. (\text{dom } h' - \text{dom } h) \cap \text{closureL } p (h,k) = \{\}$   
 $\wedge h p = h' p)$

**types** *RegionEnv* = *string*  $\rightarrow$  *TypeExpression list*  $\times$  *VarType list*  $\times$  *TypeExpression*

**constdefs** *typesArgAPP* :: *RegionEnv*  $\Rightarrow$  *string*  $\Rightarrow$  *TypeExpression list*

*typesArgAPP*  $\Sigma$   $f = (\text{case } \Sigma f \text{ of } \text{Some } (ti, \varrho s, tf) \Rightarrow ti)$

**constdefs** *regionsArgAPP* :: *RegionEnv*  $\Rightarrow$  *string*  $\Rightarrow$  *string list*

*regionsArgAPP*  $\Sigma$   $f \equiv (\text{case } \Sigma f \text{ of } \text{Some } (ti, \varrho s, tf) \Rightarrow \varrho s)$

**constdefs** *typeResAPP* :: *RegionEnv*  $\Rightarrow$  *string*  $\Rightarrow$  *TypeExpression*

*typeResAPP*  $\Sigma$   $f \equiv (\text{case } \Sigma f \text{ of } \text{Some } (ti, \varrho s, tf) \Rightarrow tf)$

**fun**

*SafeRegionDAss* :: *unit Exp*  $\Rightarrow$  *ThetaMapping*  $\Rightarrow$  *TypeExpression*  $\Rightarrow$  *bool*

( $- : \{\} - , - \}$  1000)

**where**

*SafeRegionDAss*  $e (\vartheta 1, \vartheta 2) t =$

$(\forall E1 E2 h k td h' v r \eta.$

$(E1, E2) \vdash h, k, td, e \Downarrow h', k, v, r$

$\wedge \text{fv } e \subseteq \text{dom } E1 \wedge \text{fvReg } e \subseteq \text{dom } E2$

$\wedge \text{dom } E1 \subseteq \text{dom } \vartheta 1 \wedge \text{dom } E2 \subseteq \text{dom } \vartheta 2$

$\wedge \text{admissible } \eta k$

$\wedge \text{consistent } (\vartheta 1, \vartheta 2) \eta (E1, E2) h$

$\longrightarrow \text{consistent-v } t \eta v h')$

(\* P1 \*)

(\* P1' \*)

(\* P2 \*)

(\* P3 \*)

(\* P4 \*)



**inductive**

$ValidGlobalRegionEnv :: RegionEnv \Rightarrow bool \ (\models - \ 1000)$

**where**

$base: \models empty$   
 $| \ step: \llbracket \models \Sigma t; f \notin dom \ \Sigma t;$   
 $\vartheta 1 = map-of \ (zip \ (varsAPP \ \Sigma f \ f) \ ti);$   
 $\vartheta 2 = map-of \ (zip \ (regionsAPP \ \Sigma f \ f) \ \varrho s) \ ++ \ [self \mapsto \varrho self];$   
 $(bodyAPP \ \Sigma f \ f) : \llbracket (\vartheta 1, \vartheta 2) , \ tf \rrbracket \Longrightarrow \models \Sigma t(f \mapsto (ti, \varrho s, tf))$

**constdefs**  $SafeRegionDAssCntxt ::$

$unit \ Exp \Rightarrow RegionEnv \Rightarrow ThetaMapping \Rightarrow TypeExpression \Rightarrow bool \ (-, - : \llbracket -$   
 $, - \rrbracket 1000)$   
 $SafeRegionDAssCntxt \ e \ \Sigma t \ \vartheta \ t \equiv (\models \Sigma t \longrightarrow e : \llbracket \vartheta , t \rrbracket)$

**fun**

$SafeRegionDAssDepth :: unit \ Exp \Rightarrow string \Rightarrow nat \Rightarrow ThetaMapping \Rightarrow TypeEx-$   
 $pression \Rightarrow bool$   
 $(- \vdash -, - \llbracket -, - \rrbracket 1000)$

**where**

$SafeRegionDAssDepth \ e \ f \ n \ (\vartheta 1, \vartheta 2) \ t =$   
 $(\forall \ E1 \ E2 \ h \ k \ h' \ v \ \eta.$   
 $(E1, E2) \vdash h, k, e \Downarrow (f, n) \ h', k, v \quad (* P1 *)$   
 $\wedge fv \ e \subseteq dom \ E1 \wedge fvReg \ e \subseteq dom \ E2 \quad (* P1' *)$   
 $\wedge dom \ E1 \subseteq dom \ \vartheta 1 \wedge dom \ E2 \subseteq dom \ \vartheta 2 \quad (* P2 *)$   
 $\wedge admissible \ \eta \ k \quad (* P3 *)$   
 $\wedge consistent \ (\vartheta 1, \vartheta 2) \ \eta \ (E1, E2) \ h \quad (* P4 *)$   
 $\longrightarrow consistent-v \ t \ \eta \ v \ h')$

**inductive**  $ValidGlobalRegionEnvDepth :: string \Rightarrow nat \Rightarrow RegionEnv \Rightarrow bool$   
 $(\models -, - \ 1000)$

**where**

$base : \llbracket \models \Sigma t; f \notin dom \ \Sigma t \rrbracket \Longrightarrow \models_{f, n} \Sigma t$   
 $| \ depth0 : \llbracket \models \Sigma t; f \notin dom \ \Sigma t \rrbracket \Longrightarrow \models_{f, 0} \Sigma t(f \mapsto (ti, \varrho s, tf))$   
 $| \ step : \llbracket \models \Sigma t; f \notin dom \ \Sigma t;$   
 $\vartheta 1 = map-of \ (zip \ (varsAPP \ \Sigma f \ f) \ ti);$   
 $\vartheta 2 = map-of \ (zip \ (regionsAPP \ \Sigma f \ f) \ \varrho s) \ ++ \ [self \mapsto \varrho self];$   
 $(bodyAPP \ \Sigma f \ f) :_{f, n} \llbracket (\vartheta 1, \vartheta 2) , \ tf \rrbracket \Longrightarrow$   
 $\models_{f, Suc \ n} \Sigma t(f \mapsto (ti, \varrho s, tf))$   
 $| \ g : \llbracket \models_{f, n} \Sigma t; g \notin dom \ \Sigma t; g \neq f;$   
 $\vartheta 1 = map-of \ (zip \ (varsAPP \ \Sigma f \ g) \ ti);$   
 $\vartheta 2 = map-of \ (zip \ (regionsAPP \ \Sigma f \ g) \ \varrho s) \ ++ \ [self \mapsto \varrho self];$

$$\begin{aligned} & (bodyAPP \Sigma f g) : \{ (\vartheta 1, \vartheta 2) , tf \} \Rightarrow \\ & \models_{f, n} \Sigma t(g \mapsto (ti, qs, tf)) \end{aligned}$$

**constdefs** *SafeRegionDAssDepthCntxt* ::  
*unit Exp*  $\Rightarrow$  *RegionEnv*  $\Rightarrow$  *string*  $\Rightarrow$  *nat*  $\Rightarrow$  *ThetaMapping*  $\Rightarrow$  *TypeExpression*  $\Rightarrow$   
*bool*  $(- , - :- , - \{ - , - \} 1000)$   
*SafeRegionDAssDepthCntxt* *e*  $\Sigma m$  *f* *n*  $\vartheta$  *t*  $\equiv$   
 $(\models_{f, n} \Sigma m \longrightarrow e :_{f, n} \{ \vartheta , t \})$

**end**

## 15 Basic Facts

**theory** *BasicFacts* **imports** *SafeRegion-definitions*

**begin**

**axioms** *Regions-Lemma-5*:  
 $\{ e : \{ (\vartheta 1, \vartheta 2), t \} \}$   
 $\implies e : \{ ((mu-ext (\mu 1, \mu 2)) \circ_f \vartheta 1, (\mu 2 \circ_m \vartheta 2)), (mu-ext (\mu 1, \mu 2) t) \}$

**axioms** *Regions-Lemma-5-Depth*:  
 $\{ e :_f , n \{ (\vartheta 1, \vartheta 2), t \} \}$   
 $\implies e :_f , n \{ ((mu-ext (\mu 1, \mu 2)) \circ_f \vartheta 1, (\mu 2 \circ_m \vartheta 2)), (mu-ext (\mu 1, \mu 2) t) \}$

**axioms** *qfake-not-in-dom-η*:  
 $qfake \notin dom \eta$

**axioms** *no-cycles*:  
 $h \ p = Some \ (j, C, vn)$   
 $\implies \forall \ i < length \ vn. \ p \notin closureV \ (vn!i) \ (h, k)$

**axioms** *fresh-notin-closureL*:  
 $fresh \ p \ h$   
 $\implies \forall \ q \in dom \ h. \ p \notin closureL \ q \ (h, k)$

**axioms** *semantic-extend-pointers*:  
 $(E1, E2) \vdash h , k , td , e \Downarrow h' , k , v , r$

$$\implies (\forall p \in \text{dom } h. p \notin \text{dom } h' \vee h p = h' p)$$

**axioms** *semantic-no-capture-h*:

$$\begin{aligned} & \llbracket (E1, E2) \vdash h, k, td, e \Downarrow h', k, v, r ; \\ & \quad v' \in \text{rangeHeap } h - \text{domLoc } h \rrbracket \\ & \implies v' \notin \text{domLoc } h' \end{aligned}$$

**axioms** *semantic-no-capture-E1*:

$$\begin{aligned} & \llbracket (E1, E2) \vdash h, k, td, e \Downarrow h', k, v, r ; \\ & \quad E1 \ x = \text{Some } (\text{Loc } p); \\ & \quad p \notin \text{dom } h \rrbracket \\ & \implies p \notin \text{dom } h' \end{aligned}$$

**axioms** *semantic-no-capture-E1-fresh*:

$$\begin{aligned} & \llbracket (E1, E2) \vdash h, k, td, e \Downarrow h', k, v, r ; \\ & \quad \text{fresh } p \ h \rrbracket \\ & \implies \forall x \in \text{dom } E1. \forall q. E1 \ x = \text{Some } (\text{Loc } q) \longrightarrow p \neq q \end{aligned}$$

**axioms** *semantic-no-capture-E1-fresh-2*:

$$\begin{aligned} & \llbracket (E1, E2) \vdash h, k, td, e \Downarrow h', k, v, r ; \\ & \quad \text{fresh } p \ h \rrbracket \\ & \implies \forall x \in \text{dom } E1. \forall q. E1 \ x = \text{Some } (\text{Loc } q) \longrightarrow p \notin \text{closureL } q \ (h, k) \end{aligned}$$

**axioms** *semantic-no-capture-E1-fresh-2-semDepth*:

$$\begin{aligned} & \llbracket (E1, E2) \vdash h, k, e \Downarrow (f, n) \ h', k, v; \\ & \quad \text{fresh } p \ h \rrbracket \\ & \implies \forall x \in \text{dom } E1. \forall q. E1 \ x = \text{Some } (\text{Loc } q) \longrightarrow p \notin \text{closureL } q \ (h, k) \end{aligned}$$

**axioms** *closureV-equals-closureL*:

$$\begin{aligned} & h \ p = \text{Some } (j, C, vs) \\ & \implies \text{closureL } p \ (h, k) = (\bigcup i < \text{length } vs. \text{closureV } (vs!i) \ (h, k)) \cup \{p\} \end{aligned}$$

**axioms** *closureV-subseteq-closureL-None*:

$$\begin{aligned} & h \ p = \text{Some } (j, C, vs) \\ & \implies (\bigcup i < \text{length } vs. \text{closureV } (vs!i) \ (h(p := \text{None}), k)) \subseteq \text{closureL } p \ (h(p := \text{None}), k) \end{aligned}$$

**axioms** *SafeDARegion-Var2-2*:

$$\begin{aligned} & \forall i < \text{length } tn'. \text{consistent-v } (\mu 1, \mu 2) \ (tn' ! i) \ \eta' \ (vn ! i) \ h \\ & \implies \forall i < \text{length } (\text{snd } (\text{mapAccumL } (\text{copy}' \ j) \ h \ (\text{zip } vn \ (\text{recursiveArgs } C))))). \\ & \quad \text{consistent-v } (\text{map } (\mu 1, \mu 2) (\varrho \mapsto \varrho')) \ tn' ! i \ \eta' \\ & \quad (\text{snd } (\text{mapAccumL } (\text{copy}' \ j) \ h \ (\text{zip } vn \ (\text{recursiveArgs } C)))) ! i \ h' \end{aligned}$$

**axioms** *dom-copy'*:

$copy(h, k) p j = ((h', k), p')$   
 $\implies copy'\text{-}dom(j, h, Loc p, True)$

end

## 16 Derived Assertions. P5. shareRec L $\Gamma$ E h. P6. $\neg$ identityClosure

**theory** *SafeDAss-P5-P6* **imports** *SafeDAssBasic*  
*SafeRegion-definitions*  
*BasicFacts*

**begin**

Lemma for REUSE

**lemma** *P5-REUSE*:

$\llbracket \Gamma x = Some\ d'';$   
 $wellFormed\ \{x\}\ \Gamma\ (ReuseE\ x\ ( ));$   
 $(E1, E2) \vdash h, k, td, ReuseE\ x\ ( ) \Downarrow h(p := None)(q \mapsto c), k, Loc\ q, r;$   
 $dom\ \Gamma \subseteq dom\ E1; E1\ x = Some\ (Loc\ p) \rrbracket$   
 $\implies \forall xa \in dom\ E1.$   
 $\quad closure\ (E1, E2)\ xa\ (h, k) \cap recReach\ (E1, E2)\ x\ (h, k) \neq \{\}$   
 $\quad \longrightarrow xa \in dom\ \Gamma \wedge \Gamma\ xa \neq Some\ s''$

**apply** (*simp only: wellFormed-def*)  
**apply** (*erule-tac x=E1 in allE*)  
**apply** (*erule-tac x=E2 in allE*)  
**apply** (*erule-tac x=h in allE*)  
**apply** (*erule-tac x=k in allE*)  
**apply** (*erule-tac x=td in allE*)  
**apply** (*erule-tac x=h(p := None)(q  $\mapsto$  c) in allE*)  
**apply** (*erule-tac x=Loc q in allE*)  
**apply** (*erule-tac x=r in allE*)  
**apply** (*rule ballI, rule impI*)  
**apply** (*rename-tac y*)  
**apply** (*drule mp, simp, simp add: dom-def*)  
**apply** (*erule-tac x=y in ballE*)  
**prefer** 2 **apply** *force*  
**apply** (*erule-tac x=x in ballE*)  
**prefer** 2 **apply** *blast*  
**by** *simp*

**lemma** *reuse-identityClosure-y-in-E1*:

$\llbracket (E1, E2) \vdash h, k, td, ReuseE\ x\ ( ) \Downarrow h(p := None)(q \mapsto c), k, Loc\ q, r;$   
 $p \notin closure\ (E1, E2)\ y\ (h, k); h\ p = Some\ c; fresh\ q\ h \rrbracket$   
 $\implies identityClosure\ (E1, E2)\ y\ (h, k)\ (h(p := None)(q \mapsto c), k)$

```

apply (subgoal-tac  $p \neq q$ )
  prefer 2 apply (simp add: fresh-def, blast)

apply (simp add: identityClosure-def)
apply (simp add: closure-def)
apply (case-tac  $E1$   $y$ , simp-all)
apply (case-tac  $a$ , simp-all)
apply clarsimp
apply (rename-tac  $w$ )
apply (case-tac  $w = p$ )

apply (subgoal-tac  $p \in \text{closureL } p \ (h,k), \text{simp}$ )
apply (rule closureL-basic)

apply (frule semantic-no-capture-E1-fresh-2, simp)

apply (erule-tac  $x=y$  in ballE)
  prefer 2 apply force
apply (erule-tac  $x=w$  in allE, simp)
apply (rule conjI)
apply (rule equalityI)

  apply (rule subsetI)
  apply (erule closureL.induct)
  apply (rule closureL-basic)
  apply (subgoal-tac  $qa \neq q$ )
  apply (rule closureL-step, simp)
  apply (simp add: descendants-def)
  apply (case-tac  $h$   $qa$ , simp-all)
  apply force
  apply force

  apply (rule subsetI)
  apply (erule closureL.induct)
  apply (rule closureL-basic)
  apply (rule closureL-step, simp)
  apply (subgoal-tac  $qa \neq q$ )
  apply (subgoal-tac  $qa \neq p$ )
  apply (simp add: descendants-def)
  apply force
  apply force
by force

```

**lemma** *P6-REUSE*:

$\llbracket \Gamma \ x = \text{Some } d''; \ h \ p = \text{Some } c; \ \text{fresh } q \ h;$   
 $\text{wellFormed } \{x\} \ \Gamma \ (\text{ReuseE } x \ ( ));$

$(E1, E2) \vdash h, k, td, ReuseE\ x\ () \Downarrow h(p := None)(q \mapsto c), k, Loc\ q, r;$   
 $dom\ \Gamma \subseteq dom\ E1; E1\ x = Some\ (Loc\ p)\llbracket$   
 $\implies \forall x \in dom\ E1.$   
 $\quad \neg identityClosure\ (E1, E2)\ x\ (h, k)\ (h(p := None)(q \mapsto c), k)$   
 $\quad \longrightarrow x \in dom\ \Gamma \wedge \Gamma\ x \neq Some\ s''$   
**apply** (rule ballI, rule impI)  
**apply** (rename-tac y)  
**apply** (frule P5-REUSE, assumption+)  
**apply** (erule-tac x=y in ballE)  
**prefer** 2 **apply** simp  
**apply** (case-tac  $p \in closure\ (E1, E2)\ y\ (h, k)$ )  
**apply** (subgoal-tac  $p \in recReach\ (E1, E2)\ x\ (h, k), force$ )  
**apply** (simp add: recReach-def)  
**apply** (rule recReachL-basic)  
**apply** (frule-tac  $q=q$  and  $c=c$  in reuse-identityClosure-y-in-E1)  
**by** (assumption+, simp)

**lemma** P5-P6-REUSE:

$\llbracket \Gamma\ x = Some\ d''; wellFormed\ \{x\}\ \Gamma\ (ReuseE\ x\ ());$   
 $h\ p = Some\ c; fresh\ q\ h;$   
 $(E1, E2) \vdash h, k, td, ReuseE\ x\ () \Downarrow h(p := None)(q \mapsto c), k, Loc\ q, r;$   
 $dom\ \Gamma \subseteq dom\ E1;$   
 $E1\ x = Some\ (Loc\ p)\llbracket$   
 $\implies shareRec\ \{x\}\ \Gamma\ (E1, E2)\ (h, k)\ (h(p := None)(q \mapsto c), k)$   
**apply** (simp add: shareRec-def)  
**apply** (rule conjI)  
**apply** (rule P5-REUSE, assumption+)  
**by** (rule P6-REUSE, assumption+)

Lemma for COPY

**lemma** P5-COPY:

$\llbracket wellFormed\ \{x\}\ \Gamma\ (x\ @\ r\ ());$   
 $(E1, E2) \vdash h, k, td, x\ @\ r\ () \Downarrow hh, k, v, ra;$   
 $dom\ \Gamma \subseteq dom\ E1; E1\ x = Some\ (Loc\ p)\llbracket$   
 $\implies (\forall xa \in dom\ E1. \Gamma\ xa = Some\ d'' \wedge closure\ (E1, E2)\ xa\ (h, k) \cap recReach$   
 $(E1, E2)\ x\ (h, k) \neq \{\})$   
 $\longrightarrow xa \in dom\ \Gamma \wedge \Gamma\ xa \neq Some\ s''$   
**apply** (simp only: wellFormed-def)  
**apply** (erule-tac x=E1 in allE)  
**apply** (erule-tac x=E2 in allE)  
**apply** (erule-tac x=h in allE)  
**apply** (erule-tac x=k in allE)  
**apply** (erule-tac x=td in allE)  
**apply** (erule-tac x=hh in allE)  
**apply** (erule-tac x=v in allE)  
**apply** (erule-tac x=ra in allE)  
**apply** (rule ballI, rule impI)

```

apply (rename-tac y)
apply (drule mp,simp, simp add: dom-def)
apply (erule-tac x=y in ballE)
  prefer 2 apply force
apply (erule-tac x=x in ballE)
  prefer 2 apply blast
by simp

```

**lemma** *P6-COPY*:

```


$$\llbracket \text{wellFormed } \{x\} \Gamma (x @ r ());$$


$$(E1, E2) \vdash h, k, td, x @ r () \Downarrow hh, k, v, ra ;$$


$$\text{dom } \Gamma \subseteq \text{dom } E1; E1 x = \text{Some } (Loc p) \rrbracket$$


$$\implies \forall x \in \text{dom } E1.$$


$$\neg \text{identityClosure } (E1, E2) x (h, k) (hh, k)$$


$$\longrightarrow x \in \text{dom } \Gamma \wedge \Gamma x \neq \text{Some } s''$$

apply (rule ballI, rule impI)
apply (rename-tac y)
apply (frule P5-COPY, assumption+)
apply (erule-tac x=y in ballE)
prefer 2 apply simp
apply (frule-tac L={x} and  $\Gamma=\Gamma$  in z-in-SR)
by (simp add: SR-def)

```

**lemma** *P5-P6-COPY*:

```


$$\llbracket \text{wellFormed } \{x\} \Gamma (x @ r ());$$


$$(E1, E2) \vdash h, k, td, x @ r () \Downarrow hh, k, v, ra ;$$


$$\text{dom } \Gamma \subseteq \text{dom } E1;$$


$$E1 x = \text{Some } (Loc p) \rrbracket$$


$$\implies \text{shareRec } \{x\} \Gamma (E1, E2) (h, k) (hh, k)$$

apply (simp add: shareRec-def)
apply (rule conjI)
  apply (rule P5-COPY, assumption+)
by (rule P6-COPY, assumption+)

```

Lemmas for LET1 and LET2

**lemma**  *$\Gamma 1z-s-\Gamma 2z-d$ -equals-recReach*:

```


$$\llbracket \text{def-disjointUnionEnv } \Gamma 2 (\text{empty}(x1 \mapsto m)); \text{dom } \Gamma 1 \subseteq \text{dom } E1;$$


$$\text{shareRec } L1 \Gamma 1 (E1, E2) (h, k) (h', k');$$


$$x \in \text{dom } E1; z \neq x1;$$


$$\Gamma 1 z = \text{Some } s''; z \in L1 \rrbracket$$


$$\implies \text{recReach } (E1, E2) z (h, k) = \text{recReach } (E1(x1 \mapsto r), E2) z (h', k')$$

apply (simp only: shareRec-def)
apply (elim conjE)
apply (erule-tac x=x in ballE) prefer 2 apply simp
apply (erule-tac x=z in ballE) prefer 2 apply simp apply blast
apply (case-tac  $\neg \text{identityClosure } (E1, E2) z (h, k) (h', k'), \text{simp}$ )
apply simp apply (rule equals-recReach, assumption+)

```

done

**lemma** *P5- $\Gamma 2z-d-\Gamma 1z-s$ :*

```

  [[def-pp  $\Gamma 1 \ \Gamma 2 \ L2$ ; dom  $\Gamma 1 \subseteq \text{dom } E1$ ;
    def-disjointUnionEnv  $\Gamma 2 \ (\text{empty}(x1 \mapsto m))$ ;
    dom (pp  $\Gamma 1 \ \Gamma 2 \ L2$ )  $\subseteq \text{dom } E1$ ;
    shareRec  $L2 \ (\text{disjointUnionEnv } \Gamma 2 \ (\text{empty}(x1 \mapsto m))) \ (E1(x1 \mapsto r), E2) \ (h',$ 
 $k')$  (hh,kk);
    shareRec  $L1 \ \Gamma 1 \ (E1, E2) \ (h, k) \ (h',k')$ ;
     $x \in \text{dom } E1$ ;
     $\Gamma 1 \ z = \text{Some } s''$ ;  $z \in L1$ ;
     $z \in L2$ ;  $x1 \notin L1$ ;  $\Gamma 2 \ z = \text{Some } d''$ ;
    (pp  $\Gamma 1 \ \Gamma 2 \ L2$ )  $z = \text{Some } d''$ ;
     $x1 \notin \text{dom } E1$ ;
     $z \neq x1$ ;
    closure  $(E1, E2) \ x \ (h, k) \cap \text{recReach } (E1, E2) \ z \ (h, k) \neq \{\}$ ]]
   $\implies x \in \text{dom } (\text{pp } \Gamma 1 \ \Gamma 2 \ L2) \wedge (\text{pp } \Gamma 1 \ \Gamma 2 \ L2) \ x \neq \text{Some } s''$ 
apply (frule-tac  $r=r$  in  $\Gamma 1z-s-\Gamma 2z-d-\text{equals-recReach}$ , assumption+)
apply (case-tac identityClosure  $(E1, E2) \ x \ (h, k) \ (h',k')$ )
apply (simp add: identityClosure-def)
apply (elim conjE)
apply (subgoal-tac  $x \neq x1$ ) prefer 2 apply blast
apply (subgoal-tac  $x \neq x1 \implies \text{closure } (E1, E2) \ x \ (h', k') = \text{closure } (E1(x1 \mapsto$ 
 $r), E2) \ x \ (h', k')$ )
prefer 2 apply (simp add: closure-def)
apply simp
apply (simp add: shareRec-def)
apply (elim conjE)
apply (erule-tac  $x=x$  in ballE) +
prefer 2 apply simp
prefer 2 apply simp
prefer 2 apply simp
prefer 2 apply simp
apply (frule Gamma2-d-disjointUnionEnv-m-d,simp)
apply simp
apply (drule-tac  $Q = x \in \text{dom } (\Gamma 2 + [x1 \mapsto m]) \wedge (\Gamma 2 + [x1 \mapsto m]) \ x \neq \text{Some}$ 
 $s''$  in mp)
apply (rule-tac  $x=z$  in bexI)
prefer 2 apply simp
apply (rule conjI,simp) apply blast
apply (elim conjE)
apply (rule conjI)
apply (rule dom-Gamma2-dom-triangle,assumption+)
apply (rule unsafe-Gamma2-unsafe-triangle,assumption+)
apply (simp add: shareRec-def)
apply (elim conjE)
apply (erule-tac  $x=x$  in ballE) +

```



```

prefer 2 apply simp
prefer 2 apply simp
prefer 2 apply simp
prefer 2 apply simp
apply (simp add: identityClosure-def)
apply (elim conjE)
by (rule triangle-prop)

```

```

lemma P5-Γ1z-d:  $\llbracket \text{shareRec } L1 \ \Gamma1 \ (E1, E2) \ (h, k) \ (h', k') \ ;$ 
 $x \in \text{dom } E1; (pp \ \Gamma1 \ \Gamma2 \ L2) \ z = \text{Some } d'';$ 
 $\text{closure } (E1, E2) \ x \ (h, k) \cap \text{recReach } (E1, E2) \ z \ (h, k) \neq \{\}; z \in$ 
 $L1; \Gamma1 \ z = \text{Some } d'' \rrbracket$ 
 $\implies x \in \text{dom } (pp \ \Gamma1 \ \Gamma2 \ L2) \wedge (pp \ \Gamma1 \ \Gamma2 \ L2) \ x \neq \text{Some } s''$ 
apply (simp add: shareRec-def)
apply (elim conjE)
apply (erule-tac x=x in ballE)+
apply clarsimp
apply (drule mp)
apply (rule-tac x=z in bexI)
apply (rule conjI, assumption, clarsimp)
apply simp
apply (erule conjE)
  apply (rule triangle-prop, assumption+)
apply simp
apply simp
done

```

```

lemma P5-LET-L1:  $\llbracket \text{def-pp } \Gamma1 \ \Gamma2 \ L2;$ 
 $L1 \subseteq \text{dom } \Gamma1;$ 
 $\text{dom } (pp \ \Gamma1 \ \Gamma2 \ L2) \subseteq \text{dom } E1; \text{dom } \Gamma1 \subseteq \text{dom } E1;$ 
 $\text{def-disjointUnionEnv } \Gamma2 \ (\text{empty}(x1 \mapsto m));$ 
 $\text{shareRec } L1 \ \Gamma1 \ (E1, E2) \ (h, k) \ (h', k');$ 
 $\text{shareRec } L2 \ (\text{disjointUnionEnv } \Gamma2 \ (\text{empty}(x1 \mapsto m))) \ (E1(x1 \mapsto r),$ 
 $E2) \ (h', k') \ (hh, kk);$ 
 $x \in \text{dom } E1; x1 \notin \text{dom } E1; x1 \notin L1;$ 
 $(pp \ \Gamma1 \ \Gamma2 \ L2) \ z = \text{Some } d'';$ 
 $\text{closure } (E1, E2) \ x \ (h, k) \cap \text{recReach } (E1, E2) \ z \ (h, k) \neq \{\};$ 
 $z \in L1 \rrbracket$ 
 $\implies x \in \text{dom } (pp \ \Gamma1 \ \Gamma2 \ L2) \wedge (pp \ \Gamma1 \ \Gamma2 \ L2) \ x \neq \text{Some } s''$ 
apply (subgoal-tac  $\llbracket (pp \ \Gamma1 \ \Gamma2 \ L2) \ z = \text{Some } d'' \rrbracket \implies \Gamma1 \ z = \text{Some } d'' \vee \Gamma2 \ z = \text{Some } d''$ )
prefer 2 apply (erule triangle-d-Gamma1-d-or-Gamma2-d, simp)
apply (erule disjE)

```

```

apply (rule P5-Γ1z-d, assumption+)

```

```

apply (frule triangle-d-Gamma2-d-Gamma1-s, assumption+)

```

**apply** (*erule conjE*)  
**apply** (*case-tac z≠x1*)  
**apply** (*rule P5-Γ2z-d-Γ1z-s, assumption+*)  
**by** *simp*

**lemma** *P5-z-notin-L1-Γ1z-s-Γ2z-d*:  
 $\llbracket \text{def-pp } \Gamma 1 \ \Gamma 2 \ L2; \text{def-disjointUnionEnv } \Gamma 2 \ [x1 \mapsto m]; \text{dom } (pp \ \Gamma 1 \ \Gamma 2 \ L2) \subseteq \text{dom } E1;$   
 $\text{shareRec } L2 \ (\Gamma 2 + [x1 \mapsto m]) \ (E1(x1 \mapsto r), E2) \ (h', k') \ (hh, kk); \text{shareRec}$   
 $L1 \ \Gamma 1 \ (E1, E2) \ (h, k) \ (h', k');$   
 $x \in \text{dom } E1; \Gamma 1 \ z = \text{Some } s'';$   
 $z \in L2; x1 \notin L1; \Gamma 2 \ z = \text{Some } d''; (pp \ \Gamma 1 \ \Gamma 2 \ L2) \ z = \text{Some } d''; x1 \notin \text{dom}$   
 $E1; z \neq x1;$   
 $\text{closure } (E1, E2) \ x \ (h, k) \cap \text{recReach } (E1(x1 \mapsto r), E2) \ z \ (h', k') \neq \{\};$   
 $\text{recReach } (E1, E2) \ z \ (h, k) = \text{recReach } (E1(x1 \mapsto r), E2) \ z \ (h', k') \rrbracket$   
 $\implies x \in \text{dom } (pp \ \Gamma 1 \ \Gamma 2 \ L2) \wedge (pp \ \Gamma 1 \ \Gamma 2 \ L2) \ x \neq \text{Some } s''$   
**apply** (*case-tac identityClosure* ( $E1, E2$ )  $x \ (h, k) \ (h', k')$ )  
**apply** (*simp add: identityClosure-def*)  
**apply** (*elim conjE*)  
**apply** (*subgoal-tac x≠x1*) **prefer** 2 **apply** *blast*  
**apply** (*subgoal-tac x≠x1*)  $\implies \text{closure } (E1, E2) \ x \ (h', k') = \text{closure } (E1(x1 \mapsto$   
 $r), E2) \ x \ (h', k')$   
**prefer** 2 **apply** (*simp add: closure-def*)  
**apply** *simp*  
**apply** (*simp add: shareRec-def*)  
**apply** (*elim conjE*)  
**apply** (*erule-tac x=x in ballE*)  
**prefer** 2 **apply** *simp*  
**prefer** 2 **apply** *simp*  
**prefer** 2 **apply** *simp*  
**prefer** 2 **apply** *simp*  
**apply** (*frule Gamma2-d-disjointUnionEnv-m-d, assumption+*)  
**apply** (*drule-tac Q=*  $x \in \text{dom } (\Gamma 2 + [x1 \mapsto m]) \wedge (\Gamma 2 + [x1 \mapsto m]) \ x \neq \text{Some}$   
 $s'' \text{ in } mp$ )  
**apply** (*rule-tac x=z in bexI*)  
**prefer** 2 **apply** *simp*  
**apply** (*rule conjI, simp*)  
**apply** *simp*  
**apply** (*elim conjE*)  
**apply** (*rule conjI*)  
**apply** (*rule dom-Gamma2-dom-triangle, assumption+*)  
**apply** (*rule unsafe-Gamma2-unsafe-triangle, assumption+*)  
**apply** (*simp add: shareRec-def*)  
**apply** (*elim conjE*)  
**apply** (*erule-tac x=x in ballE*)  
**prefer** 2 **apply** *simp*  
**prefer** 2 **apply** *simp*  
**prefer** 2 **apply** *simp*

```

prefer 2 apply simp
apply (simp add: identityClosure-def)
apply (elim conjE)
by (rule triangle-prop)

```

**lemma** *P5- $\Gamma 2z-d-\Gamma 1z-s-z-in-L2$ :*

```

   $\llbracket$  def-pp  $\Gamma 1 \ \Gamma 2 \ L2$ ;
    def-disjointUnionEnv  $\Gamma 2 \ (\text{empty}(x1 \mapsto m))$ ;
     $\text{dom } (pp \ \Gamma 1 \ \Gamma 2 \ L2) \subseteq \text{dom } E1$ ;
    shareRec  $L2 \ (\text{disjointUnionEnv } \Gamma 2 \ (\text{empty}(x1 \mapsto m))) \ (E1(x1 \mapsto r), E2) \ (h',$ 
 $k') \ (hh, kk)$ ;
    shareRec  $L1 \ \Gamma 1 \ (E1, E2) \ (h, k) \ (h', k')$ ;
     $x \in \text{dom } E1$ ;
     $\Gamma 1 \ z = \text{Some } s''; z \in L2; x1 \notin L1; \Gamma 2 \ z = \text{Some } d''$ ;
     $(pp \ \Gamma 1 \ \Gamma 2 \ L2) \ z = \text{Some } d''$ ;
     $x1 \notin \text{dom } E1$ ;
 $z \neq x1$ ;
    recReach  $(E1, E2) \ z \ (h, k) = \text{recReach } (E1(x1 \mapsto r), E2) \ z \ (h', k')$ ;
     $\text{closure } (E1, E2) \ x \ (h, k) \cap \text{recReach } (E1, E2) \ z \ (h, k) \neq \{\}$ 
 $\implies x \in \text{dom } (pp \ \Gamma 1 \ \Gamma 2 \ L2) \wedge (pp \ \Gamma 1 \ \Gamma 2 \ L2) \ x \neq \text{Some } s''$ 
apply (case-tac x≠x1)
apply (subgoal-tac x≠x1  $\implies \text{closure } (E1, E2) \ x \ (h', k') = \text{closure } (E1(x1 \mapsto$ 
 $r), E2) \ x \ (h', k')$ )
prefer 2 apply (simp add: closure-def)
apply simp
prefer 2 apply simp
apply (frule P5-z-notin-L1- $\Gamma 1z-s-\Gamma 2z-d$ , assumption+)
done

```

**lemma** *P5- $\Gamma 2z-d-z-notin-\Gamma 1$ :*

```

   $\llbracket$  def-pp  $\Gamma 1 \ \Gamma 2 \ L2$ ;
    def-disjointUnionEnv  $\Gamma 2 \ (\text{empty}(x1 \mapsto m))$ ;
     $\text{dom } (pp \ \Gamma 1 \ \Gamma 2 \ L2) \subseteq \text{dom } E1$ ;
    shareRec  $L2 \ (\text{disjointUnionEnv } \Gamma 2 \ (\text{empty}(x1 \mapsto m))) \ (E1(x1 \mapsto r), E2) \ (h',$ 
 $k') \ (hh, kk)$ ;
    shareRec  $L1 \ \Gamma 1 \ (E1, E2) \ (h, k) \ (h', k')$ ;
     $x \in \text{dom } E1$ ;
     $z \notin \text{dom } \Gamma 1; z \in L2; x1 \notin L1; \Gamma 2 \ z = \text{Some } d''$ ;
     $(pp \ \Gamma 1 \ \Gamma 2 \ L2) \ z = \text{Some } d''$ ;
     $x1 \notin \text{dom } E1$ ;
 $z \neq x1$ ;
    recReach  $(E1, E2) \ z \ (h, k) = \text{recReach } (E1(x1 \mapsto r), E2) \ z \ (h', k')$ ;
     $\text{closure } (E1, E2) \ x \ (h, k) \cap \text{recReach } (E1, E2) \ z \ (h, k) \neq \{\}$ 
 $\implies x \in \text{dom } (pp \ \Gamma 1 \ \Gamma 2 \ L2) \wedge (pp \ \Gamma 1 \ \Gamma 2 \ L2) \ x \neq \text{Some } s''$ 

```

```

apply (case-tac identityClosure (E1, E2) x (h, k) (h', k'))
apply (simp add: identityClosure-def)
apply (elim conjE)
apply (subgoal-tac x≠x1) prefer 2 apply blast
apply (subgoal-tac x≠x1  $\implies$  closure (E1, E2) x (h', k') = closure (E1(x1  $\mapsto$ 
r), E2) x (h', k'))
prefer 2 apply (simp add: closure-def)
apply simp
apply (simp add: shareRec-def)
apply (elim conjE)
apply (erule-tac x=x in ballE)+
prefer 2 apply simp
prefer 2 apply simp
prefer 2 apply simp
prefer 2 apply simp
apply (frule Gamma2-d-disjointUnionEnv-m-d, assumption+)
apply (drule-tac Q= x  $\in$  dom ( $\Gamma 2 + [x1 \mapsto m]$ )  $\wedge$  ( $\Gamma 2 + [x1 \mapsto m]$ ) x  $\neq$  Some
s'' in mp)
apply (rule-tac x=z in bexI)
prefer 2 apply simp
apply (rule conjI, simp)
apply simp
apply (elim conjE)
apply (rule conjI)
apply (rule dom-Gamma2-dom-triangle, assumption+)
apply (rule unsafe-Gamma2-unsafe-triangle, assumption+)
apply (simp add: shareRec-def)
apply (elim conjE)
apply (erule-tac x=x in ballE)+
prefer 2 apply simp
prefer 2 apply simp
prefer 2 apply simp
prefer 2 apply simp
apply (simp add: identityClosure-def)
apply (elim conjE)
by (rule triangle-prop)

```

**lemma** *P5-LET-L2*:

```

 $\llbracket L1 \subseteq \text{dom } \Gamma 1; \text{dom } \Gamma 1 \subseteq \text{dom } E1;$ 
 $L2 \subseteq \text{dom } (\text{disjointUnionEnv } \Gamma 2 \text{ } (\text{empty}(x1 \mapsto m)));$ 
 $\text{def-pp } \Gamma 1 \ \Gamma 2 \ L2;$ 
 $\text{dom } (\text{pp } \Gamma 1 \ \Gamma 2 \ L2) \subseteq \text{dom } E1;$ 
 $\text{def-disjointUnionEnv } \Gamma 2 \text{ } (\text{empty}(x1 \mapsto m));$ 
 $\text{shareRec } L1 \ \Gamma 1 \ (E1, E2) \ (h, k) \ (h', k');$ 
 $\text{shareRec } L2 \ (\text{disjointUnionEnv } \Gamma 2 \text{ } (\text{empty}(x1 \mapsto m))) \ (E1(x1 \mapsto r), E2) \ (h',$ 
 $k') \ (hh, kk);$ 

```

$x \in \text{dom } E1; x1 \notin \text{dom } E1; x1 \notin L1;$   
 $(pp \ \Gamma 1 \ \Gamma 2 \ L2) \ z = \text{Some } d'';$   
 $\text{closure } (E1, E2) \ x \ (h, k) \cap \text{recReach } (E1, E2) \ z \ (h, k) \neq \{\};$   
 $z \in L2; z \neq x1]$   
 $\implies x \in \text{dom } (pp \ \Gamma 1 \ \Gamma 2 \ L2) \wedge (pp \ \Gamma 1 \ \Gamma 2 \ L2) \ x \neq \text{Some } s''$   
**apply** (frule triangle-d-Gamma1-s-or-not-dom-Gamma1, assumption+)  
**apply** (erule disjE)

**apply** (subgoal-tac  $\llbracket (pp \ \Gamma 1 \ \Gamma 2 \ L2) \ z = \text{Some } d''; \ \Gamma 1 \ z = \text{Some } s'' \rrbracket \implies \Gamma 2 \ z = \text{Some } d''$ )  
**prefer** 2 **apply** (rule triangle-d-Gamma1-s-Gamma2-d, assumption+)  
**apply** simp  
**apply** (subgoal-tac  $z \in \text{dom } E1$ )  
**prefer** 2 **apply** blast  
**apply** (case-tac identityClosure  $(E1, E2) \ z \ (h, k) \ (h', k')$ )  
**apply** (frule identityClosure-equals-recReach)  
**apply** (subgoal-tac  $z \neq x1 \implies \text{recReach } (E1, E2) \ z \ (h', k') = \text{recReach } (E1(x1 \mapsto r), E2) \ z \ (h', k')$ )  
**prefer** 2 **apply** (simp add: recReach-def)  
**apply** simp  
**apply** (rule P5-z-notin-L1- $\Gamma 1$ z-s- $\Gamma 2$ z-d, assumption+)  
**apply** (simp add: shareRec-def)

**apply** (case-tac  $z \in \text{dom } E1$ ) **prefer** 2 **apply** blast  
**apply** (case-tac identityClosure  $(E1, E2) \ z \ (h, k) \ (h', k')$ )  
**apply** (frule identityClosure-equals-recReach)  
**apply** (subgoal-tac  $z \neq x1 \implies \text{recReach } (E1, E2) \ z \ (h', k') = \text{recReach } (E1(x1 \mapsto r), E2) \ z \ (h', k')$ )  
**prefer** 2 **apply** (simp add: recReach-def)  
**apply** simp  
**apply** (subgoal-tac  $\Gamma 2 \ z = \text{Some } d''$ )  
**apply** (rule P5- $\Gamma 2$ z-d-z-notin- $\Gamma 1$ , assumption+) **apply** simp  
**apply** (simp add: pp-def)  
**by** (simp add: shareRec-def)

**lemma** P5-Cond2:  
 $\llbracket L1 \subseteq \text{dom } \Gamma 1;$   
 $L2 \subseteq \text{dom } (\text{disjointUnionEnv } \Gamma 2 \ (\text{empty}(x1 \mapsto m)));$   
 $x1 \notin \text{dom } E1; x1 \notin L1;$   
 $\text{dom } (pp \ \Gamma 1 \ \Gamma 2 \ L2) \subseteq \text{dom } E1;$   
 $\text{def-pp } \Gamma 1 \ \Gamma 2 \ L2;$   
 $\text{def-disjointUnionEnv } \Gamma 2 \ (\text{empty}(x1 \mapsto m));$   
 $\text{shareRec } L1 \ \Gamma 1 \ (E1, E2) \ (h, k) \ (h', k');$   
 $\text{shareRec } L2 \ (\text{disjointUnionEnv } \Gamma 2 \ (\text{empty}(x1 \mapsto m))) \ (E1(x1 \mapsto r),$   
 $E2) \ (h', k') \ (hh, kk) \rrbracket$   
 $\implies \forall x \in \text{dom } E1. \neg \text{identityClosure } (E1, E2) \ x \ (h, k) \ (hh, kk) \longrightarrow x \in$   
 $\text{dom } (pp \ \Gamma 1 \ \Gamma 2 \ L2) \wedge (pp \ \Gamma 1 \ \Gamma 2 \ L2) \ x \neq \text{Some } s''$   
**apply** (rule ballI, rule impI)  
**apply** (case-tac  $\neg \text{identityClosure } (E1, E2) \ x \ (h, k) \ (h', k')$ )

```

apply (simp add: shareRec-def)
apply (elim conjE)
apply (erule-tac x=x in ballE)+
prefer 2 apply simp
prefer 2 apply simp
prefer 2 apply simp
prefer 2 apply simp
apply simp
apply (elim conjE)
apply (rule triangle-prop,assumption+)
apply (case-tac  $\neg$  identityClosure (E1(x1  $\mapsto$  r), E2) x (h', k') (hh, kk))
apply (simp add: shareRec-def)
apply (elim conjE)
apply (erule-tac x=x in ballE)+
prefer 2 apply simp
prefer 2 apply simp
prefer 2 apply simp
prefer 2 apply simp
apply simp
apply (elim conjE)
apply (subgoal-tac x  $\neq$  x1) prefer 2 apply blast
apply (rule conjI)
  apply (rule dom-Gamma2-dom-triangle,assumption+)
apply (rule unsafe-Gamma2-unsafe-triangle,assumption+)
apply simp
  apply (subgoal-tac x  $\neq$  x1) prefer 2 apply blast
apply (frule monotone-identityClosure, assumption+)
by simp

```

**lemma** *P5-P6-LET*:

```

  [| L1  $\subseteq$  dom  $\Gamma$ 1; dom  $\Gamma$ 1  $\subseteq$  dom E1;
    L2  $\subseteq$  dom (disjointUnionEnv  $\Gamma$ 2 (empty(x1  $\mapsto$  m)));
    x1  $\notin$  dom E1; x1  $\notin$  L1;
    dom (pp  $\Gamma$ 1  $\Gamma$ 2 L2)  $\subseteq$  dom E1;
    def-pp  $\Gamma$ 1  $\Gamma$ 2 L2;
    def-disjointUnionEnv  $\Gamma$ 2 (empty(x1  $\mapsto$  m));
    shareRec L1  $\Gamma$ 1 (E1, E2) (h, k) (h', k');
    shareRec L2 (disjointUnionEnv  $\Gamma$ 2 (empty(x1  $\mapsto$  m))) (E1(x1  $\mapsto$  r), E2) (h',
    k') (hh, kk) |]
   $\impl$  shareRec (L1  $\cup$  (L2 - {x1})) (pp  $\Gamma$ 1  $\Gamma$ 2 L2) (E1, E2) (h, k) (hh, kk)
apply (simp (no-asm) add: shareRec-def)
apply (rule conjI)
apply (rule ballI, rule impI)
apply (erule bexE)
apply (erule conjE)+
apply simp
apply (erule disjE)

```

**apply** (erule P5-LET-L1, assumption+)

**apply** (erule conjE)+

**apply** (erule P5-LET-L2, assumption+)

**by** (erule P5-Cond2, assumption+)

**lemma**  $\Gamma 1\text{-}z\text{-Some-}s$ :

$z \in \text{atom2var } \text{'set as}$

$\implies (\text{map-of } (\text{zip } (\text{map atom2var as}) (\text{replicate } (\text{length as}) s'')) z = \text{Some } s''$

**by** (induct as, simp, clarsimp)

**lemma** P5-LETC-e1:

$\forall x \in \text{dom } (\text{fst } (E1, E2)).$

$\forall z \in \text{set } (\text{map atom2var as}).$

$\text{map-of } (\text{zip } (\text{map atom2var as}) (\text{replicate } (\text{length as}) s'')) z = \text{Some}$

$d'' \wedge$

$\text{closure } (E1, E2) x (h, k) \cap \text{recReach } (E1, E2) z (h, k) \neq \{\}$   $\longrightarrow$

$x \in \text{dom } (\text{map-of } (\text{zip } (\text{map atom2var as}) (\text{replicate } (\text{length as}) s'')) \wedge$

$\text{map-of } (\text{zip } (\text{map atom2var as}) (\text{replicate } (\text{length as}) s'')) x \neq \text{Some } s''$

**apply** (rule ballI)+

**apply** (rule impI, elim conjE, simp)

**by** (frule  $\Gamma 1\text{-}z\text{-Some-}s$ , simp)

**lemma** ext-h-same-descendants:

$\llbracket \text{fresh } p \ h; q \neq p \rrbracket$

$\implies \text{descendants } q (h, k) =$

$\text{descendants } q (h(p \mapsto c), k)$

**apply** (rule equalityI)

**apply** (rule subsetI)

**apply** (frule-tac  $k=k$  in fresh-notin-closureL)

**apply** (subgoal-tac  $q \in \text{dom } h$ )

**apply** (simp add: descendants-def)

**apply** (erule-tac  $x=q$  in ballE)

**apply** (simp add: descendants-def)

**apply** (case-tac  $h \ q$ , simp-all)

**apply** force

**apply** (simp add: descendants-def)

**apply** (case-tac  $h \ q$ , simp-all)

**apply** force

**apply** (rule subsetI)

**apply** (frule-tac  $k=k$  in fresh-notin-closureL)

**apply** (subgoal-tac  $q \in \text{dom } h$ )

**apply** (simp add: descendants-def)

**apply** (erule-tac  $x=q$  in ballE)

```

  prefer 2 apply simp
  apply (simp add: descendants-def)
  apply (case-tac h q,simp-all)
  apply force
  apply (simp add: descendants-def)
  apply (case-tac h q,simp-all)
  by force

```

```

lemma ext-h-same-recDescendants:
   $\llbracket \text{fresh } p \ h; \ q \neq p \rrbracket$ 
 $\implies \text{recDescendants } q \ (h, k) =$ 
 $\text{recDescendants } q \ (h(p \mapsto c), k)$ 
  apply (rule equalityI)
  apply (rule subsetI)
  apply (frule-tac k=k in fresh-notin-closureL)
  apply (subgoal-tac q  $\in \text{dom } h$ )
  apply (simp add: recDescendants-def)
  apply (erule-tac x=q in ballE)
  apply (simp add: recDescendants-def)
  apply (case-tac h q,simp-all)
  apply force
  apply (simp add: recDescendants-def)
  apply (case-tac h q,simp-all)
  apply force
  apply (rule subsetI)
  apply (frule-tac k=k in fresh-notin-closureL)
  apply (subgoal-tac q  $\in \text{dom } h$ )
  apply (simp add: recDescendants-def)
  apply (erule-tac x=q in ballE)
  prefer 2 apply simp
  apply (simp add: recDescendants-def)
  apply (case-tac h q,simp-all)
  apply force
  apply (simp add: recDescendants-def)
  apply (case-tac h q,simp-all)
  by force

```

```

lemma ext-h-same-closure:
   $\llbracket (E1, E2) \vdash h, k, td, e \Downarrow h', k, v, r; \text{fresh } p \ h \rrbracket$ 
 $\implies \text{closure } (E1, E2) \ x \ (h, k) =$ 
 $\text{closure } (E1, E2) \ x \ (h(p \mapsto c), k)$ 
  apply (simp add: closure-def)
  apply (case-tac E1 x,simp-all)
  apply (case-tac a,simp-all)
  apply clarsimp

```



```

apply (rename-tac q)
apply (frule-tac k=k in semantic-no-capture-E1-fresh-2,assumption+)
apply (erule-tac x=x in ballE)
  prefer 2 apply force
apply (erule-tac x=q in allE,simp)
apply (rule equalityI)
apply (rule subsetI)
apply (erule closureL.induct)
  apply (rule closureL-basic)
  apply (subgoal-tac qa ≠ p)
    apply (subgoal-tac d ∈ descendants qa (h(p ↦ c), k))
    apply (rule closureL-step,assumption+)
    apply (subst (asm) ext-h-same-descendants,assumption+)
apply (simp add: descendants-def)
apply (case-tac h qa, simp-all)
apply (simp add: fresh-def, force)
apply (rule subsetI)
apply (erule closureL.induct)
  apply (rule closureL-basic)
  apply (subgoal-tac d ∈ descendants qa (h, k))
    apply (rule closureL-step,assumption+)
apply (subst ext-h-same-descendants,assumption+)
  apply (case-tac qa ≠ p,simp,simp)
by simp

```

**lemma** *ext-h-same-recReach*:

```

  [ (E1, E2) ⊢ h , k , td , e ↓ h' , k , v , r ;
    fresh p h ]
  ⇒ recReach (E1,E2) x (h,k) =
     recReach (E1,E2) x (h(p ↦ c), k)
apply (simp add: recReach-def)
apply (case-tac E1 x,simp-all)
apply (case-tac a,simp-all)
apply clarsimp
apply (rename-tac q)
apply (frule-tac k=k in semantic-no-capture-E1-fresh-2,assumption+)
apply (erule-tac x=x in ballE)
  prefer 2 apply force
apply (erule-tac x=q in allE,simp)
apply (rule equalityI)
apply (rule subsetI)
apply (erule recReachL.induct)
  apply (rule recReachL-basic)
  apply (subgoal-tac qa ≠ p)
    apply (subgoal-tac d ∈ recDescendants qa (h(p ↦ c), k))
    apply (rule recReachL-step,assumption+)
    apply (subst (asm) ext-h-same-recDescendants,assumption+)
apply (simp add: recDescendants-def)

```

```

apply (case-tac h qa, simp-all)
apply (simp add: fresh-def, force)
apply (rule subsetI)
apply (erule recReachL.induct)
apply (rule recReachL-basic)
apply (subgoal-tac d ∈ recDescendants qa (h, k))
apply (rule recReachL-step, assumption+)
apply (subst ext-h-same-recDescendants, assumption+)
apply (case-tac qa ≠ p, simp, simp)
apply (subgoal-tac recReachL q (h, k) ⊆ closureL q (h, k))
apply blast
apply (rule recReachL-subseteq-closureL)
by simp

```

```

lemma closure-upt-monotone:
   $\llbracket x \in \text{dom } E1; x1 \notin \text{dom } E1 \rrbracket$ 
   $\implies \text{closure } (E1, E2) x (h, k) =$ 
     $\text{closure } (E1(x1 \mapsto \text{Loc } p), E2) x (h, k)$ 
apply (simp add: closure-def)
apply (rule impI)
by (simp add: dom-def)

```

```

lemma recReach-upt-monotone:
   $\llbracket x \in \text{dom } E1; x1 \notin \text{dom } E1 \rrbracket$ 
   $\implies \text{recReach } (E1, E2) x (h, k) =$ 
     $\text{recReach } (E1(x1 \mapsto \text{Loc } p), E2) x (h, k)$ 
apply (simp add: recReach-def)
apply (rule impI)
by (simp add: dom-def)

```

```

lemma cvte-ext-h-inter-closure-recReach:
   $\llbracket (E1, E2) \vdash h, k, td, e \Downarrow h', k, v, r;$ 
     $\text{fresh } p \ h;$ 
     $x1 \notin \text{dom } E1; x \in \text{dom } E1; z \in \text{dom } E1;$ 
     $\text{closure } (E1, E2) x (h, k) \cap \text{recReach } (E1, E2) z (h, k) \neq \{\} \rrbracket$ 
   $\implies \text{closure } (E1(x1 \mapsto \text{Loc } p), E2) x (h(p \mapsto (j, C, \text{map } (\text{atom2val } E1) \text{ as}))),$ 
     $k) \cap$ 
     $\text{recReach } (E1(x1 \mapsto \text{Loc } p), E2) z (h(p \mapsto (j, C, \text{map } (\text{atom2val } E1) \text{ as}))),$ 
     $k) \neq \{\}$ 
apply (subst (asm) ext-h-same-closure, assumption+)
apply (subst (asm) ext-h-same-recReach, assumption+)
apply (subst (asm) closure-upt-monotone, assumption+)
by (subst (asm) recReach-upt-monotone, assumption+)

```

```

lemma ext-h-same-identityClosure-upt:
   $\llbracket \text{fresh } p \ h; x \in \text{dom } E1; \quad$ 
   $(E1, E2) \vdash h, k, td, e \Downarrow h', k, v, r \rrbracket$ 
   $\implies \text{identityClosure } (E1, E2) \ x \ (h, k) \ (h(p \mapsto c), k)$ 
apply (case-tac E1 x)
apply (simp add: dom-def)
apply (case-tac a, simp-all)

apply (simp add: identityClosure-def)
apply (rule conjI)
apply (rule ext-h-same-closure,assumption+)
apply (rule impI)
apply (simp add: closure-def)
apply (frule semantic-no-capture-E1-fresh-2,assumption+)
apply force

apply (simp add: identityClosure-def)
apply (simp add: closure-def)

apply (simp add: identityClosure-def)
apply (simp add: closure-def)
done

lemma P6-LETC-e1:
   $\llbracket (E1, E2) \vdash h, k, td, \text{Let } x1 = \text{ConstrE } C \text{ as } r \ a' \text{ In } e2 \ a \Downarrow hh, k, v, ra; \quad$ 
   $(E1(x1 \mapsto \text{Loc } p), E2) \vdash h(p \mapsto (j, C, \text{map } (\text{atom2val } E1) \text{ as})), k, (td +$ 
   $1), e2 \Downarrow hh, k, v, rs'; \quad$ 
   $x1 \notin L1; x1 \notin \text{dom } E1; \quad$ 
   $\text{def-pp } (\text{map-of } (\text{zip } (\text{map } \text{atom2var } \text{as}) (\text{replicate } (\text{length } \text{as}) \ s''))) \ \Gamma \ 2 \ L2; \quad$ 
   $\text{dom } (\text{pp } (\text{map-of } (\text{zip } (\text{map } \text{atom2var } \text{as}) (\text{replicate } (\text{length } \text{as}) \ s''))) \ \Gamma \ 2 \ L2) \subseteq \text{dom } E1; \quad$ 
   $\text{fresh } p \ h; \quad$ 
   $\forall x \in \text{dom } (\text{fst } (E1(x1 \mapsto \text{Loc } p), E2)).$ 
   $\neg \text{identityClosure } (E1(x1 \mapsto \text{Loc } p), E2) \ x \ (h(p \mapsto (j, C, \text{map } (\text{atom2val } E1) \text{ as})), k) \ (hh, k) \longrightarrow$ 
   $x \in \text{dom } (\Gamma \ 2 + [x1 \mapsto m']) \wedge (\Gamma \ 2 + [x1 \mapsto m']) \ x \neq \text{Some } s''$ 
   $\implies \forall x \in \text{dom } (\text{fst } (E1, E2)).$ 
   $\neg \text{identityClosure } (E1, E2) \ x \ (h, k) \ (h(p \mapsto (j, C, \text{map } (\text{atom2val } E1) \text{ as})), k) \longrightarrow$ 
   $x \in \text{dom } (\text{map-of } (\text{zip } (\text{map } \text{atom2var } \text{as}) (\text{replicate } (\text{length } \text{as}) \ s''))) \wedge$ 
   $\text{map-of } (\text{zip } (\text{map } \text{atom2var } \text{as}) (\text{replicate } (\text{length } \text{as}) \ s'')) \ x \neq \text{Some } s''$ 
apply (rule ballI, rule impI)
apply (subgoal-tac  $x \neq x1$ )
prefer 2 apply force
apply (erule-tac  $x = x$  in ballE)
prefer 2 apply simp
apply (frule ext-h-same-identityClosure-upt)
by (assumption+, simp, force)

```

**lemma** *P5-P6-LETC-e1*:

$\llbracket (E1, E2) \vdash h, k, td, \text{Let } x1 = \text{ConstrE } C \text{ as } r \text{ a' In } e2 \text{ a} \Downarrow hh, k, v, ra ;$   
 $(E1(x1 \mapsto \text{Loc } p), E2) \vdash h(p \mapsto (j, C, \text{map } (\text{atom2val } E1) \text{ as})) , k , (td + 1)$   
 $, e2 \Downarrow hh, k, v, rs' ;$   
 $x1 \notin L1 ; x1 \notin \text{dom } E1 ;$   
 $L1 = \text{set } (\text{map } \text{atom2var } \text{as}) ;$   
 $\Gamma1 = \text{map-of } (\text{zip } (\text{map } \text{atom2var } \text{as}) (\text{replicate } (\text{length } \text{as}) s')) ;$   
 $\text{dom } (\text{pp } (\text{map-of } (\text{zip } (\text{map } \text{atom2var } \text{as}) (\text{replicate } (\text{length } \text{as}) s')))) \Gamma2 L2$   
 $\subseteq \text{dom } E1 ;$   
 $\text{def-pp } (\text{map-of } (\text{zip } (\text{map } \text{atom2var } \text{as}) (\text{replicate } (\text{length } \text{as}) s')))) \Gamma2 L2 ;$   
 $\text{shareRec } L2 (\Gamma2 + [x1 \mapsto m']) (E1(x1 \mapsto \text{Loc } p), E2) (h(p \mapsto (j, C, \text{map}$   
 $(\text{atom2val } E1) \text{ as})), k) (hh, k) ;$   
 $\text{fresh } p \ h \rrbracket$   
 $\implies \text{shareRec } L1 \Gamma1 (E1, E2) (h, k) (h(p \mapsto (j, C, \text{map } (\text{atom2val } E1) \text{ as})),$   
 $k)$   
**apply** (*simp only: shareRec-def*)  
**apply** (*elim conjE*)  
**apply** (*rule conjI*)  
**apply** (*rule P5-LETC-e1*)  
**by** (*rule P6-LETC-e1, assumption+*)

**lemma** *ext-h-same-closure-semDepth*:

$\llbracket (E1, E2) \vdash h, k, e \Downarrow (f, n) h', k, v ;$   
 $\text{fresh } p \ h \rrbracket$   
 $\implies \text{closure } (E1, E2) \ x \ (h, k) =$   
 $\text{closure } (E1, E2) \ x \ (h(p \mapsto c), k)$   
**apply** (*simp add: closure-def*)  
**apply** (*case-tac E1 x, simp-all*)  
**apply** (*case-tac a, simp-all*)  
**apply** *clarsimp*  
**apply** (*rename-tac q*)  
**apply** (*frule-tac k=k in semantic-no-capture-E1-fresh-2-semDepth, assumption+*)  
**apply** (*erule-tac x=x in ballE*)  
**prefer** 2 **apply** *force*  
**apply** (*erule-tac x=q in allE, simp*)  
**apply** (*rule equalityI*)  
**apply** (*rule subsetI*)  
**apply** (*erule closureL.induct*)  
**apply** (*rule closureL-basic*)  
**apply** (*subgoal-tac qa  $\neq$  p*)  
**apply** (*subgoal-tac d  $\in$  descendants qa (h(p  $\mapsto$  c), k)*)  
**apply** (*rule closureL-step, assumption+*)  
**apply** (*subst (asm) ext-h-same-descendants, assumption+*)  
**apply** (*simp add: descendants-def*)  
**apply** (*case-tac h qa, simp-all*)  
**apply** (*simp add: fresh-def, force*)

```

apply (rule subsetI)
apply (erule closureL.induct)
apply (rule closureL-basic)
apply (subgoal-tac  $d \in \text{descendants } qa \ (h, k)$ )
apply (rule closureL-step,assumption+)
apply (subst ext-h-same-descendants,assumption+)
apply (case-tac  $qa \neq p, \text{simp}, \text{simp}$ )
by simp

```

**lemma** *ext-h-same-identityClosure-upt-semDepth:*

```

 $\llbracket \text{fresh } p \ h; x \in \text{dom } E1; \\
(E1, E2) \vdash h, k, e \Downarrow(f,n) \ h', k, v \rrbracket \\
\implies \text{identityClosure } (E1, E2) \ x \ (h, k) \ (h(p \mapsto c), k)$ 
apply (case-tac  $E1 \ x$ )
apply (simp add: dom-def)
apply (case-tac  $a, \text{simp-all}$ )

apply (simp add: identityClosure-def)
apply (rule conjI)
apply (rule ext-h-same-closure-semDepth,assumption+)
apply (rule impI)
apply (simp add: closure-def)
apply (frule semantic-no-capture-E1-fresh-2-semDepth,assumption+)
apply force

```

```

apply (simp add: identityClosure-def)
apply (simp add: closure-def)

```

```

apply (simp add: identityClosure-def)
apply (simp add: closure-def)
done

```

**lemma** *P6-f-n-LETC-e1:*

```

 $\llbracket (E1, E2) \vdash h, k, \text{Let } x1 = \text{ConstrE } C \text{ as } r \ a' \text{ In } e2 \ a \Downarrow(f,n) \ hh, k, v; \\
(E1(x1 \mapsto \text{Loc } p), E2) \vdash h(p \mapsto (j, C, \text{map } (\text{atom2val } E1) \ as)), k, e2 \Downarrow(f,n) \\
hh, k, v; \\
x1 \notin L1; x1 \notin \text{dom } E1; \\
\text{def-pp } (\text{map-of } (\text{zip } (\text{map } \text{atom2var } as) (\text{replicate } (\text{length } as) \ s''))) \ \Gamma 2 \ L2; \\
\text{dom } (\text{pp } (\text{map-of } (\text{zip } (\text{map } \text{atom2var } as) (\text{replicate } (\text{length } as) \ s''))) \ \Gamma 2 \ L2) \\
\subseteq \text{dom } E1; \\
\text{fresh } p \ h; \\
\forall x \in \text{dom } (\text{fst } (E1(x1 \mapsto \text{Loc } p), E2)). \\
\quad \neg \text{identityClosure } (E1(x1 \mapsto \text{Loc } p), E2) \ x \ (h(p \mapsto (j, C, \text{map } (\text{atom2val } \\
E1) \ as)), k) \ (hh, k) \longrightarrow \\
\quad x \in \text{dom } (\Gamma 2 + [x1 \mapsto m']) \wedge (\Gamma 2 + [x1 \mapsto m']) \ x \neq \text{Some } s'' \rrbracket \\
\implies \forall x \in \text{dom } (\text{fst } (E1, E2)). \\
\quad \neg \text{identityClosure } (E1, E2) \ x \ (h, k) \ (h(p \mapsto (j, C, \text{map } (\text{atom2val } E1)$ 
```

$as)), k) \longrightarrow$   
 $x \in \text{dom} (\text{map-of} (\text{zip} (\text{map atom2var as}) (\text{replicate} (\text{length as}) s'')) \wedge$   
 $\text{map-of} (\text{zip} (\text{map atom2var as}) (\text{replicate} (\text{length as}) s'')) x \neq \text{Some } s''$   
**apply** (rule ballI, rule impI)  
**apply** (subgoal-tac  $x \neq x1$ )  
**prefer** 2 **apply** force  
**apply** (erule-tac  $x = x$  in ballE)  
**prefer** 2 **apply** simp  
**apply** (frule ext-h-same-identityClosure-upt-semDepth)  
**by** (assumption+, simp, force)

**lemma** P5-P6-f-n-LETC-e1:

$\llbracket (E1, E2) \vdash h, k, \text{Let } x1 = \text{ConstrE } C \text{ as } r \text{ a' In } e2 \text{ a } \Downarrow(f, n) \text{ hh}, k, v;$   
 $(E1(x1 \mapsto \text{Loc } p), E2) \vdash h(p \mapsto (j, C, \text{map} (\text{atom2val } E1) as)), k, e2 \Downarrow(f, n)$   
 $\text{hh}, k, v;$   
 $x1 \notin L1; x1 \notin \text{dom } E1;$   
 $L1 = \text{set} (\text{map atom2var as});$   
 $\Gamma1 = \text{map-of} (\text{zip} (\text{map atom2var as}) (\text{replicate} (\text{length as}) s''));$   
 $\text{dom} (pp (\text{map-of} (\text{zip} (\text{map atom2var as}) (\text{replicate} (\text{length as}) s'')) \Gamma2 L2)$   
 $\subseteq \text{dom } E1;$   
 $\text{def-pp} (\text{map-of} (\text{zip} (\text{map atom2var as}) (\text{replicate} (\text{length as}) s'')) \Gamma2 L2;$   
 $\text{shareRec } L2 (\Gamma2 + [x1 \mapsto m']) (E1(x1 \mapsto \text{Loc } p), E2) (h(p \mapsto (j, C, \text{map}$   
 $(\text{atom2val } E1) as)), k) (\text{hh}, k);$   
 $\text{fresh } p \text{ h} \rrbracket$   
 $\implies \text{shareRec } L1 \Gamma1 (E1, E2) (h, k) (h(p \mapsto (j, C, \text{map} (\text{atom2val } E1) as)),$   
 $k)$   
**apply** (simp only: shareRec-def)  
**apply** (elim conjE)  
**apply** (rule conjI)  
**apply** (rule P5-LETC-e1)  
**by** (rule P6-f-n-LETC-e1, assumption+)

Lemmas for CASE

**lemma** P5-CASE-shareRec:

$\llbracket (E1, E2) \vdash h, k, td, \text{Case VarE } x () \text{ Of alts } () \Downarrow \text{hh}, k, v, r;$   
 $\text{dom} (\text{foldl op} \otimes \text{empty} (\text{map snd assert})) \subseteq \text{dom } E1;$   
 $\text{insert } x (\bigcup_i < \text{length alts fst (assert ! i)} - \text{set} (\text{snd} (\text{extractP} (\text{fst} (\text{alts ! i}))))))$   
 $\subseteq \text{dom} (\text{foldl op} \otimes \text{empty} (\text{map snd assert}));$   
 $\text{fv} (\text{Case VarE } x () \text{ Of alts } ()) \subseteq \text{insert } x (\bigcup_i < \text{length alts fst (assert ! i)} -$   
 $\text{set} (\text{snd} (\text{extractP} (\text{fst} (\text{alts ! i})))));$   
 $\text{wellFormed} (\text{insert } x (\bigcup_i < \text{length alts fst (assert ! i)} - \text{set} (\text{snd} (\text{extractP} (\text{fst}$   
 $(\text{alts ! i}))))))$   
 $(\text{foldl op} \otimes \text{empty} (\text{map snd assert})) (\text{Case VarE } x () \text{ Of alts } ()) \rrbracket$   
 $\implies \forall xa \in \text{dom} (\text{fst } (E1, E2)).$   
 $\forall z \in \text{insert } x (\bigcup_i < \text{length alts fst (assert ! i)} - \text{set} (\text{snd} (\text{extractP} (\text{fst}$   
 $(\text{alts ! i}))))).$   
 $\text{foldl op} \otimes \text{empty} (\text{map snd assert}) z = \text{Some } d'' \wedge \text{closure } (E1, E2) xa$   
 $(h, k) \cap \text{recReach } (E1, E2) z (h, k) \neq \{\}$

$xa \in \text{dom } (\text{foldl } op \otimes \text{empty } (\text{map } \text{snd } \text{assert})) \wedge \text{foldl } op \otimes \text{empty } (\text{map } \text{snd } \text{assert}) \text{ } xa \neq \text{Some } s''$   
**apply** (*simp only: wellFormed-def*)  
**apply** (*erule-tac x=E1 in allE*)  
**apply** (*erule-tac x=E2 in allE*)  
**apply** (*erule-tac x=h in allE*)  
**apply** (*erule-tac x=k in allE*)  
**apply** (*erule-tac x=td in allE*)  
**apply** (*erule-tac x=hh in allE*)  
**apply** (*erule-tac x=v in allE*)  
**apply** (*erule-tac x=r in allE*)  
**apply** (*drule mp,simp*)  
**by** *simp*

**lemma** *closure-monotone-extend:*

$\llbracket x \in \text{dom } E;$   
 $\text{def-extend } E \text{ (snd (extractP (fst (alts ! i)))) vs};$   
 $\text{length alts} > 0; i < \text{length alts} \rrbracket$   
 $\implies \text{closure } (E, E') \text{ } x \text{ (h, k)} =$   
 $\text{closure } (\text{extend } E \text{ (snd (extractP (fst (alts ! i)))) vs, } E') \text{ } x \text{ (h, k)}$   
**apply** (*simp add: def-extend-def*)  
**apply** (*subgoal-tac x  $\notin$  set (snd (extractP (fst (alts ! i))))*)  
**apply** (*subgoal-tac*  
 $E \text{ } x = \text{extend } E \text{ (snd (extractP (fst (alts ! i)))) vs } x$ )  
**apply** (*simp add: closure-def*)  
**apply** (*rule extend-monotone-i*)  
**apply** (*assumption+,simp,simp*)  
**by** *blast*

**lemma** *identityClosure-monotone-extend:*

$\llbracket x \in \text{dom } E1;$   
 $\text{length alts} > 0; i < \text{length alts};$   
 $\text{def-extend } E1 \text{ (snd (extractP (fst (alts ! i)))) vs};$   
 $\neg \text{identityClosure } (E1, E2) \text{ } x \text{ (h, k) (hh, k)} \rrbracket$   
 $\implies \neg \text{identityClosure } (\text{extend } E1 \text{ (snd (extractP (fst (alts ! i)))) vs, } E2) \text{ } x \text{ (h, k) (hh, k)}$   
**apply** (*simp add: identityClosure-def*)  
**apply** (*rule impI*)  
**apply** (*subgoal-tac*  
 $\text{closure } (E1, E2) \text{ } x \text{ (h, k)} =$   
 $\text{closure } (\text{extend } E1 \text{ (snd (extractP (fst (alts ! i)))) vs, } E2) \text{ } x \text{ (h, k),simp}$ )  
**apply** (*subgoal-tac*  
 $\text{closure } (E1, E2) \text{ } x \text{ (hh, k)} =$   
 $\text{closure } (\text{extend } E1 \text{ (snd (extractP (fst (alts ! i)))) vs, } E2) \text{ } x \text{ (hh, k),simp}$ )  
**apply** (*rule closure-monotone-extend,assumption+,simp,assumption+*)

**by** (*rule closure-monotone-extend,assumption+,simp,assumption+*)

**lemma** *P5-CASE-identityClosure*:

```

  [ length assert = length alts;
    length alts > 0; i < length alts;
    def-extend E1 (snd (extractP (fst (alts ! i)))) vs;
    def-nonDisjointUnionEnvList (map snd assert);
    (∀ x ∈ dom (fst (extend E1 (snd (extractP (fst (alts ! i)))) vs, E2)).
      ¬ identityClosure (extend E1 (snd (extractP (fst (alts ! i)))) vs, E2) x (h,
k) (hh, k) ⟶
      x ∈ dom (snd (assert ! i)) ∧ snd (assert ! i) x ≠ Some s') ]
  ⟹ ∀ x ∈ dom (fst (E1, E2)).
    ¬ identityClosure (E1, E2) x (h, k) (hh, k) ⟶
    x ∈ dom (foldl op ⊗ empty (map snd assert)) ∧ foldl op ⊗ empty (map
snd assert) x ≠ Some s''
apply (rule ballI)
apply (erule-tac x=x in ballE)
apply (rule impI)
apply (drule mp)
apply (rule identityClosure-monotone-extend,simp,assumption+)
apply (elim conjE)
apply (rule conjI)
apply (subgoal-tac length assert > i)
apply (frule dom-monotone)
apply blast
apply simp
apply (rule Otimes-prop2)
apply (simp,simp,assumption+)
apply (subgoal-tac
  E1 x = extend E1 (snd (extractP (fst (alts ! i)))) vs x)
apply (simp add: dom-def)
apply (rule extend-monotone-i,assumption+)
apply (simp add: def-extend-def)
by blast

```

**lemma** *P5-P6-CASE*:

```

  [ (E1, E2) ⊢ h , k , td , Case VarE x () Of alts () ↓ hh , k , v , r ;
    dom (foldl op ⊗ empty (map snd assert)) ⊆ dom E1;
    insert x (⋃i < length alts fst (assert ! i) - set (snd (extractP (fst (alts ! i)))))
⊆ dom (foldl op ⊗ empty (map snd assert));
    fv (Case VarE x () Of alts ()) ⊆ insert x (⋃i < length alts fst (assert ! i) -
set (snd (extractP (fst (alts ! i))))) ;
    i < length alts;

```



$def\_extend\ E1\ (snd\ (extractP\ (fst\ (alts\ !\ i))))\ vs;$   
 $def\_nonDisjointUnionEnvList\ (map\ snd\ assert);\ alts\ \neq\ [];\ length\ assert = length\ alts;$   
 $wellFormed\ (insert\ x\ (\bigcup_{i < length\ alts}\ fst\ (assert\ !\ i) - set\ (snd\ (extractP\ (fst\ (alts\ !\ i))))))$   
 $(foldl\ op\ \otimes\ empty\ (map\ snd\ assert))\ (Case\ VarE\ x\ ()\ Of\ alts\ ());$   
 $(E1,\ E2) \vdash h,\ k,\ td,\ Case\ VarE\ x\ ()\ Of\ alts\ () \Downarrow hh,\ k,\ v,\ r;$   
 $shareRec\ (fst\ (assert\ !\ i))\ (snd\ (assert\ !\ i))\ (extend\ E1\ (snd\ (extractP\ (fst\ (alts\ !\ i))))\ vs,\ E2)\ (h,\ k)\ (hh,\ k)]$   
 $\implies shareRec\ (insert\ x\ (\bigcup_{i < length\ alts}\ fst\ (assert\ !\ i) - set\ (snd\ (extractP\ (fst\ (alts\ !\ i))))))$   
 $(foldl\ op\ \otimes\ empty\ (map\ snd\ assert))\ (E1,\ E2)\ (h,\ k)\ (hh,\ k)$   
**apply** (*simp* (*no-asm*) *only*: *shareRec-def*)  
**apply** (*rule conjI*)  
**apply** (*rule P5-CASE-shareRec,assumption+*)  
**apply** (*simp only*: *shareRec-def*)  
**apply** (*rule P5-CASE-identityClosure,assumption+*)  
**by** (*simp, simp, assumption+, simp*)

**lemma** *P5-f-n-CASE-shareRec*:

$\llbracket (E1,\ E2) \vdash h,\ k,\ Case\ VarE\ x\ ()\ Of\ alts\ () \Downarrow (f,n)\ hh,\ k,\ v;$   
 $dom\ (foldl\ op\ \otimes\ empty\ (map\ snd\ assert)) \subseteq dom\ E1;$   
 $insert\ x\ (\bigcup_{i < length\ alts}\ fst\ (assert\ !\ i) - set\ (snd\ (extractP\ (fst\ (alts\ !\ i))))$   
 $\subseteq dom\ (foldl\ op\ \otimes\ empty\ (map\ snd\ assert));$   
 $fv\ (Case\ VarE\ x\ ()\ Of\ alts\ ()) \subseteq insert\ x\ (\bigcup_{i < length\ alts}\ fst\ (assert\ !\ i) - set\ (snd\ (extractP\ (fst\ (alts\ !\ i)))));$   
 $wellFormedDepth\ f\ n\ (insert\ x\ (\bigcup_{i < length\ alts}\ fst\ (assert\ !\ i) - set\ (snd\ (extractP\ (fst\ (alts\ !\ i))))$   
 $(foldl\ op\ \otimes\ empty\ (map\ snd\ assert))\ (Case\ VarE\ x\ ()\ Of\ alts\ ()) \rrbracket$   
 $\implies \forall xa \in dom\ (fst\ (E1,\ E2)).$   
 $\forall z \in insert\ x\ (\bigcup_{i < length\ alts}\ fst\ (assert\ !\ i) - set\ (snd\ (extractP\ (fst\ (alts\ !\ i))))).$   
 $foldl\ op\ \otimes\ empty\ (map\ snd\ assert)\ z = Some\ d'' \wedge closure\ (E1,\ E2)\ xa$   
 $(h,\ k) \cap recReach\ (E1,\ E2)\ z\ (h,\ k) \neq \{\}$   $\longrightarrow$   
 $xa \in dom\ (foldl\ op\ \otimes\ empty\ (map\ snd\ assert)) \wedge foldl\ op\ \otimes\ empty\ (map\ snd\ assert)\ xa \neq Some\ s''$   
**apply** (*simp only*: *wellFormedDepth-def*)  
**apply** (*erule-tac*  $x=E1$  **in** *allE*)  
**apply** (*erule-tac*  $x=E2$  **in** *allE*)  
**apply** (*erule-tac*  $x=h$  **in** *allE*)  
**apply** (*erule-tac*  $x=k$  **in** *allE*)  
**apply** (*erule-tac*  $x=hh$  **in** *allE*)  
**apply** (*erule-tac*  $x=v$  **in** *allE*)  
**apply** (*erule mp, simp*)  
**by** *simp*

**lemma** *P5-P6-f-n-CASE*:

$\llbracket (E1, E2) \vdash h, k, \text{Case VarE } x () \text{ Of alts } () \Downarrow (f, n) hh, k, v;$   
 $\text{dom (foldl op } \otimes \text{ empty (map snd assert))} \subseteq \text{dom } E1;$   
 $\text{insert } x (\bigcup_{i < \text{length alts}} \text{fst (assert ! } i) - \text{set (snd (extractP (fst (alts ! } i))))})$   
 $\subseteq \text{dom (foldl op } \otimes \text{ empty (map snd assert))};$   
 $\text{fv (Case VarE } x () \text{ Of alts } ()) \subseteq \text{insert } x (\bigcup_{i < \text{length alts}} \text{fst (assert ! } i) -$   
 $\text{set (snd (extractP (fst (alts ! } i))))});$   
 $i < \text{length alts};$   
 $\text{def-extend } E1 (\text{snd (extractP (fst (alts ! } i)))) \text{ vs};$   
 $\text{def-nonDisjointUnionEnvList (map snd assert); alts} \neq []; \text{length assert} = \text{length}$   
 $\text{alts};$   
 $\text{wellFormedDepth } f \ n (\text{insert } x (\bigcup_{i < \text{length alts}} \text{fst (assert ! } i) - \text{set (snd (extractP (fst (alts ! } i))))})$   
 $(\text{foldl op } \otimes \text{ empty (map snd assert)) (Case VarE } x () \text{ Of alts } ());$   
 $\text{shareRec (fst (assert ! } i)) (\text{snd (assert ! } i)) (\text{extend } E1 (\text{snd (extractP (fst (alts ! } i)))) \text{ vs, } E2) (h, k) (hh, k) \rrbracket$   
 $\implies \text{shareRec (insert } x (\bigcup_{i < \text{length alts}} \text{fst (assert ! } i) - \text{set (snd (extractP (fst (alts ! } i))))})$   
 $(\text{foldl op } \otimes \text{ empty (map snd assert)) (E1, E2) (h, k) (hh, k)$   
**apply** (*simp* (*no-asm*) *only*: *shareRec-def*)  
**apply** (*rule conjI*)  
**apply** (*rule P5-f-n-CASE-shareRec,assumption+*)  
**apply** (*simp only*: *shareRec-def*)  
**apply** (*rule P5-CASE-identityClosure,assumption+*)  
**by** (*simp, simp, assumption+, simp*)

**lemma** *P5-CASE-1-1-identityClosure*:

$\llbracket \text{length assert} = \text{length alts};$   
 $\text{length alts} > 0; i < \text{length alts};$   
 $\text{fst (alts ! } i) = \text{ConstP (LitN } n);$   
 $\text{def-nonDisjointUnionEnvList (map snd assert);}$   
 $(\forall x \in \text{dom (fst (E1, E2)). } \neg \text{identityClosure (E1, E2) } x (h, k) (hh, k)$   
 $\longrightarrow x \in \text{dom (snd (assert ! } i)) \wedge \text{snd (assert ! } i) \ x \neq \text{Some } s'') \rrbracket$   
 $\implies \forall x \in \text{dom (fst (E1, E2)).}$   
 $\neg \text{identityClosure (E1, E2) } x (h, k) (hh, k) \longrightarrow$   
 $x \in \text{dom (foldl op } \otimes \text{ empty (map snd assert))} \wedge \text{foldl op } \otimes \text{ empty (map}$   
 $\text{snd assert) } x \neq \text{Some } s''$   
**apply** (*rule ballI*)  
**apply** (*erule-tac x=x in ballE*)  
**apply** (*rule impI*)  
**apply** (*erule mp, simp*)  
**apply** (*rule conjI*)  
**apply** (*subgoal-tac length assert > i*)

**apply** (*frule dom-monotone*)  
**apply** *blast*  
**apply** *simp*  
**apply** (*rule Otimes-prop2*)  
**apply** (*simp, simp, assumption+, simp, simp*)  
**by** (*simp add: dom-def*)

**lemma** *P5-P6-CASE-1-1*:

$\llbracket (E1, E2) \vdash h, k, td, \text{Case VarE } x () \text{ Of alts } () \Downarrow hh, k, v, r ;$   
 $\text{fst } (alts ! i) = \text{ConstP } (\text{LitN } n);$   
 $\text{dom } (\text{foldl } op \otimes \text{empty } (\text{map } snd \text{ assert})) \subseteq \text{dom } E1;$   
 $\text{insert } x (\bigcup_i < \text{length } alts \text{ fst } (\text{assert } ! i) - \text{set } (snd (\text{extractP } (\text{fst } (alts ! i))))))$   
 $\subseteq \text{dom } (\text{foldl } op \otimes \text{empty } (\text{map } snd \text{ assert}));$   
 $\text{fv } (\text{Case VarE } x () \text{ Of alts } ()) \subseteq \text{insert } x (\bigcup_i < \text{length } alts \text{ fst } (\text{assert } ! i) -$   
 $\text{set } (snd (\text{extractP } (\text{fst } (alts ! i)))));$   
 $i < \text{length } alts;$   
 $\text{def-nonDisjointUnionEnvList } (\text{map } snd \text{ assert}); alts \neq []; \text{length } \text{assert} = \text{length}$   
 $alts;$   
 $\text{wellFormed } (\text{insert } x (\bigcup_i < \text{length } alts \text{ fst } (\text{assert } ! i) - \text{set } (snd (\text{extractP } (\text{fst}$   
 $(alts ! i))))))$   
 $(\text{foldl } op \otimes \text{empty } (\text{map } snd \text{ assert})) (\text{Case VarE } x () \text{ Of alts } ());$   
 $(E1, E2) \vdash h, k, td, \text{Case VarE } x () \text{ Of alts } () \Downarrow hh, k, v, r ;$   
 $\text{shareRec } (\text{fst } (\text{assert } ! i)) (snd (\text{assert } ! i)) (E1, E2) (h, k) (hh, k) \rrbracket$   
 $\implies \text{shareRec } (\text{insert } x (\bigcup_i < \text{length } alts \text{ fst } (\text{assert } ! i) - \text{set } (snd (\text{extractP}$   
 $(\text{fst } (alts ! i))))))$   
 $(\text{foldl } op \otimes \text{empty } (\text{map } snd \text{ assert})) (E1, E2) (h, k) (hh, k)$   
**apply** (*simp (no-asm) only: shareRec-def*)  
**apply** (*rule conjI*)  
**apply** (*rule P5-CASE-shareRec, assumption+*)  
**apply** (*simp only: shareRec-def*)  
**apply** (*rule P5-CASE-1-1-identityClosure, assumption+*)  
**by** (*simp, force, force, assumption+, simp*)

**lemma** *P5-P6-f-n-CASE-1-1*:

$\llbracket (E1, E2) \vdash h, k, \text{Case VarE } x () \text{ Of alts } () \Downarrow (f, n) hh, k, v;$   
 $\text{fst } (alts ! i) = \text{ConstP } (\text{LitN } n');$   
 $\text{dom } (\text{foldl } op \otimes \text{empty } (\text{map } snd \text{ assert})) \subseteq \text{dom } E1;$   
 $\text{insert } x (\bigcup_i < \text{length } alts \text{ fst } (\text{assert } ! i) - \text{set } (snd (\text{extractP } (\text{fst } (alts ! i))))))$   
 $\subseteq \text{dom } (\text{foldl } op \otimes \text{empty } (\text{map } snd \text{ assert}));$   
 $\text{fv } (\text{Case VarE } x () \text{ Of alts } ()) \subseteq \text{insert } x (\bigcup_i < \text{length } alts \text{ fst } (\text{assert } ! i) -$   
 $\text{set } (snd (\text{extractP } (\text{fst } (alts ! i)))));$   
 $i < \text{length } alts;$   
 $\text{def-nonDisjointUnionEnvList } (\text{map } snd \text{ assert}); alts \neq []; \text{length } \text{assert} = \text{length}$   
 $alts;$   
 $\text{wellFormedDepth } f \ n \ (\text{insert } x (\bigcup_i < \text{length } alts \text{ fst } (\text{assert } ! i) - \text{set } (snd$   
 $(\text{extractP } (\text{fst } (alts ! i))))))$   
 $(\text{foldl } op \otimes \text{empty } (\text{map } snd \text{ assert})) (\text{Case VarE } x () \text{ Of alts } ());$

```

    shareRec (fst (assert ! i)) (snd (assert ! i)) (E1, E2) (h, k) (hh, k)
     $\implies$  shareRec (insert x ( $\bigcup_{i < \text{length } \text{alts}} \text{fst } (\text{assert } ! i) - \text{set } (\text{snd } (\text{extractP}$ 
(fst (alts ! i))))))
    (foldl op  $\otimes$  empty (map snd assert)) (E1, E2) (h, k) (hh, k)
  apply (simp (no-asm) only: shareRec-def)
  apply (rule conjI)
  apply (rule P5-f-n-CASE-shareRec,assumption+)
  apply (simp only: shareRec-def)
  apply (rule P5-CASE-1-1-identityClosure,assumption+)
  by (simp,force,force,assumption+,simp)

```

**lemma** *P5-CASE-1-2-identityClosure*:

```

 $\llbracket$  length assert = length alts;
    length alts > 0; i < length alts;
    fst (alts ! i) = ConstP (LitB b);
    def-nonDisjointUnionEnvList (map snd assert);
    ( $\forall x \in \text{dom } (\text{fst } (E1, E2)). \neg \text{identityClosure } (E1, E2) x (h, k) (hh, k)$ 
 $\longrightarrow x \in \text{dom } (\text{snd } (\text{assert } ! i)) \wedge \text{snd } (\text{assert } ! i) x \neq \text{Some } s''$ )  $\rrbracket$ 
 $\implies \forall x \in \text{dom } (\text{fst } (E1, E2)).$ 
 $\neg \text{identityClosure } (E1, E2) x (h, k) (hh, k) \longrightarrow$ 
 $x \in \text{dom } (\text{foldl op } \otimes \text{ empty } (\text{map snd assert})) \wedge \text{foldl op } \otimes \text{ empty } (\text{map}$ 
snd assert)  $x \neq \text{Some } s''$ 
  apply (rule ballI)
  apply (erule-tac x=x in ballE)
  apply (rule impI)
  apply (drule mp,simp)
  apply (rule conjI)
  apply (subgoal-tac length assert > i)
  apply (frule dom-monotone)
  apply blast
  apply simp
  apply (rule Otimes-prop2)
  apply (simp,simp,assumption+,simp,simp)
  by (simp add: dom-def)

```

**lemma** *P5-P6-CASE-1-2*:

```

 $\llbracket (E1, E2) \vdash h, k, td, \text{Case VarE } x () \text{ Of alts } () \Downarrow hh, k, v, r;$ 
    fst (alts ! i) = ConstP (LitB b);
    dom (foldl op  $\otimes$  empty (map snd assert))  $\subseteq \text{dom } E1$ ;
    insert x ( $\bigcup_{i < \text{length } \text{alts}} \text{fst } (\text{assert } ! i) - \text{set } (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! i))))$ )
 $\subseteq \text{dom } (\text{foldl op } \otimes \text{ empty } (\text{map snd assert}))$ ;
    fv (Case VarE x () Of alts ())  $\subseteq \text{insert } x (\bigcup_{i < \text{length } \text{alts}} \text{fst } (\text{assert } ! i) -$ 
set (snd (extractP (fst (alts ! i)))));
    i < length alts;

```

$\text{def-nonDisjointUnionEnvList } (\text{map snd assert}); \text{alts} \neq []; \text{length assert} = \text{length alts};$   
 $\text{wellFormed } (\text{insert } x \ (\bigcup_i < \text{length alts } \text{fst } (\text{assert } ! i) - \text{set } (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! i))))))$   
 $\quad (\text{foldl op } \otimes \text{empty } (\text{map snd assert})) \ (\text{Case VarE } x \ () \text{ Of alts } ());$   
 $\quad (E1, E2) \vdash h, k, td, \text{Case VarE } x \ () \text{ Of alts } () \Downarrow hh, k, v, r;$   
 $\quad \text{shareRec } (\text{fst } (\text{assert } ! i)) \ (\text{snd } (\text{assert } ! i)) \ (E1, E2) \ (h, k) \ (hh, k)]$   
 $\quad \Rightarrow \text{shareRec } (\text{insert } x \ (\bigcup_i < \text{length alts } \text{fst } (\text{assert } ! i) - \text{set } (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! i))))))$   
 $\quad (\text{foldl op } \otimes \text{empty } (\text{map snd assert})) \ (E1, E2) \ (h, k) \ (hh, k)$   
**apply** (*simp* (*no-asm*) *only*: *shareRec-def*)  
**apply** (*rule conjI*)  
**apply** (*rule P5-CASE-shareRec,assumption+*)  
**apply** (*simp only*: *shareRec-def*)  
**apply** (*rule P5-CASE-1-2-identityClosure,assumption+*)  
**by** (*simp,force,force,assumption+,simp*)

**lemma** *P5-P6-f-n-CASE-1-2*:

$\llbracket (E1, E2) \vdash h, k, \text{Case VarE } x \ () \text{ Of alts } () \Downarrow (f,n) \ hh, k, v;$   
 $\text{fst } (\text{alts } ! i) = \text{ConstP } (\text{LitB } b);$   
 $\text{dom } (\text{foldl op } \otimes \text{empty } (\text{map snd assert})) \subseteq \text{dom } E1;$   
 $\text{insert } x \ (\bigcup_i < \text{length alts } \text{fst } (\text{assert } ! i) - \text{set } (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! i))))))$   
 $\subseteq \text{dom } (\text{foldl op } \otimes \text{empty } (\text{map snd assert}));$   
 $\text{fv } (\text{Case VarE } x \ () \text{ Of alts } ()) \subseteq \text{insert } x \ (\bigcup_i < \text{length alts } \text{fst } (\text{assert } ! i) - \text{set } (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! i)))));$   
 $i < \text{length alts};$   
 $\text{def-nonDisjointUnionEnvList } (\text{map snd assert}); \text{alts} \neq []; \text{length assert} = \text{length alts};$   
 $\text{wellFormedDepth } f \ n \ (\text{insert } x \ (\bigcup_i < \text{length alts } \text{fst } (\text{assert } ! i) - \text{set } (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! i))))))$   
 $\quad (\text{foldl op } \otimes \text{empty } (\text{map snd assert})) \ (\text{Case VarE } x \ () \text{ Of alts } ());$   
 $\quad \text{shareRec } (\text{fst } (\text{assert } ! i)) \ (\text{snd } (\text{assert } ! i)) \ (E1, E2) \ (h, k) \ (hh, k)]$   
 $\quad \Rightarrow \text{shareRec } (\text{insert } x \ (\bigcup_i < \text{length alts } \text{fst } (\text{assert } ! i) - \text{set } (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! i))))))$   
 $\quad (\text{foldl op } \otimes \text{empty } (\text{map snd assert})) \ (E1, E2) \ (h, k) \ (hh, k)$   
**apply** (*simp* (*no-asm*) *only*: *shareRec-def*)  
**apply** (*rule conjI*)  
**apply** (*rule P5-f-n-CASE-shareRec,assumption+*)  
**apply** (*simp only*: *shareRec-def*)  
**apply** (*rule P5-CASE-1-2-identityClosure,assumption+*)  
**by** (*simp,force,force,assumption+,simp*)

```

lemma closureL-p-None-p:
  closureL p (h(p := None), k) = {p}
apply (rule equalityI)
apply (rule subsetI)
apply (erule closureL.induct,simp)
apply (simp add: descendants-def)
apply (rule subsetI,simp)
by (rule closureL-basic)

lemma recReachL-p-None-p:
  recReachL p (h(p := None), k) = {p}
apply (rule equalityI)
apply (rule subsetI)
apply (erule recReachL.induct,simp)
apply (simp add: recDescendants-def)
apply (rule subsetI,simp)
by (rule recReachL-basic)

lemma descendants-p-None-q:
   $\llbracket d \in \text{descendants } q \text{ (} h(p := \text{None}), k \text{); } q \neq p \rrbracket$ 
 $\implies d \in \text{descendants } q \text{ (} h, k \text{)}$ 
by (simp add: descendants-def)

lemma recDescendants-p-None-q:
   $\llbracket d \in \text{recDescendants } q \text{ (} h(p := \text{None}), k \text{); } q \neq p \rrbracket$ 
 $\implies d \in \text{recDescendants } q \text{ (} h, k \text{)}$ 
by (simp add: recDescendants-def)

lemma closureL-p-None-subseteq-closureL:
   $p \neq q$ 
 $\implies \text{closureL } q \text{ (} h(p := \text{None}), k \text{)} \subseteq \text{closureL } q \text{ (} h, k \text{)}$ 
apply (rule subsetI)
apply (erule closureL.induct)
apply (rule closureL-basic)
apply clarsimp
apply (subgoal-tac d ∈ descendants qa (h,k))
apply (rule closureL-step,simp,simp)
apply (rule descendants-p-None-q,assumption+)
apply (simp add: descendants-def)
by (case-tac qa = p,simp-all)

lemma recReachL-p-None-subseteq-recReachL:
   $p \neq q$ 
 $\implies \text{recReachL } q \text{ (} h(p := \text{None}), k \text{)} \subseteq \text{recReachL } q \text{ (} h, k \text{)}$ 
apply (rule subsetI)
apply (erule recReachL.induct)
apply (rule recReachL-basic)

```

**apply** *clarsimp*  
**apply** (*subgoal-tac*  $d \in \text{recDescendants } qa \ (h,k)$ )  
**apply** (*rule* *recReachL-step,simp,simp*)  
**apply** (*rule* *recDescendants-p-None-q,assumption+*)  
**apply** (*simp* *add: recDescendants-def*)  
**by** (*case-tac*  $qa = p, \text{simp-all}$ )

**lemma** *dom-foldl-monotone-list*:  
 $\text{dom } (\text{foldl } op \otimes (\text{empty} \otimes x) \ xs) =$   
 $\text{dom } x \cup \text{dom } (\text{foldl } op \otimes \text{empty} \ xs)$   
**apply** (*subgoal-tac*  $\text{empty} \otimes x = x \otimes \text{empty}, \text{simp}$ )  
**apply** (*subgoal-tac*  $\text{foldl } op \otimes (x \otimes \text{empty}) \ xs =$   
 $x \otimes \text{foldl } op \otimes \text{empty} \ xs, \text{simp}$ )  
**apply** (*rule* *union-dom-nonDisjointUnionEnv*)  
**apply** (*rule* *foldl-prop1*)  
**apply** (*subgoal-tac* *def-nonDisjointUnionEnv empty x*)  
**apply** (*erule* *nonDisjointUnionEnv-commutative*)  
**by** (*simp* *add: def-nonDisjointUnionEnv-def*)

**lemma** *dom-restrict-neg-map*:  
 $\text{dom } (\text{restrict-neg-map } m \ A) = \text{dom } m - (\text{dom } m \cap A)$   
**apply** (*simp* *add: restrict-neg-map-def*)  
**apply** *auto*  
**by** (*split split-if-asm,simp-all*)

**lemma** *x-notin- $\Gamma$ -cased*:  
 $x \notin \text{dom } (\text{foldl } op \otimes \text{empty}$   
 $\quad (\text{map } (\lambda(Li, \Gamma i). \text{restrict-neg-map } \Gamma i \ (\text{insert } x \ (\text{set } Li)))$   
 $\quad (\text{zip } (\text{map } (\text{snd} \circ \text{extractP} \circ \text{fst}) \ \text{alts}) \ (\text{map } \text{snd } \text{assert}))))$   
**apply** (*induct-tac* *assert alts rule: list-induct2',simp-all*)  
**apply** (*subgoal-tac*  
 $\text{dom } (\text{foldl } op \otimes (\text{empty} \otimes \text{restrict-neg-map } (\text{snd } xa) \ (\text{insert } x \ (\text{set } (\text{snd } (\text{extractP}$   
 $(\text{fst } y))))))$   
 $\quad (\text{map } (\lambda(Li, \Gamma i). \text{restrict-neg-map } \Gamma i \ (\text{insert } x \ (\text{set } Li))) \ (\text{zip } (\text{map}$   
 $(\lambda a. \text{snd } (\text{extractP } (\text{fst } a))) \ ys) \ (\text{map } \text{snd } xs)))) =$   
 $\text{dom } (\text{restrict-neg-map } (\text{snd } xa) \ (\text{insert } x \ (\text{set } (\text{snd } (\text{extractP } (\text{fst } y)))))) \cup$   
 $\text{dom } (\text{foldl } op \otimes \text{empty} \ (\text{map } (\lambda(Li, \Gamma i). \text{restrict-neg-map } \Gamma i \ (\text{insert } x \ (\text{set } Li)))$   
 $\quad (\text{zip } (\text{map } (\lambda a. \text{snd } (\text{extractP } (\text{fst } a))) \ ys) \ (\text{map } \text{snd}$   
 $xs))))), \text{simp}$ )  
**apply** (*subst dom-restrict-neg-map,blast*)  
**by** (*rule dom-foldl-monotone-list*)

**lemma**  *$\Gamma$ -case-x-is-d*:  
 $\llbracket \Gamma = \text{foldl } op \otimes \text{empty}$   
 $\quad (\text{map } (\lambda(Li, \Gamma i). \text{restrict-neg-map } \Gamma i \ (\text{insert } x \ (\text{set } Li))) \ (\text{zip } (\text{map}$   
 $(\text{snd} \circ \text{extractP} \circ \text{fst}) \ \text{alts}) \ (\text{map } \text{snd } \text{assert}))) \rrbracket +$

```

    
$$\Rightarrow \Gamma \ x = \text{Some } d'' \ ]$$

apply (simp add: disjointUnionEnv-def)
apply (simp add: unionEnv-def)
apply (rule impI)
apply (insert x-notin- $\Gamma$ -cased)
by force

```

```

lemma closureL-p-None-p:
  closureL p (h(p := None), k) = {p}
apply (rule equalityI)
apply (rule subsetI)
apply (erule closureL.induct,simp)
apply (simp add: descendants-def)
apply (rule subsetI,simp)
by (rule closureL-basic)

```

```

lemma recReachL-p-None-p:
  recReachL p (h(p := None), k) = {p}
apply (rule equalityI)
apply (rule subsetI)
apply (erule recReachL.induct,simp)
apply (simp add: recDescendants-def)
apply (rule subsetI,simp)
by (rule recReachL-basic)

```

```

lemma descendants-p-None-q:
  
$$\llbracket d \in \text{descendants } q \ (h(p := \text{None}),k); \ q \neq p \rrbracket$$


$$\Rightarrow d \in \text{descendants } q \ (h,k)$$

by (simp add: descendants-def)

```

```

lemma recDescendants-p-None-q:
  
$$\llbracket d \in \text{recDescendants } q \ (h(p := \text{None}),k); \ q \neq p \rrbracket$$


$$\Rightarrow d \in \text{recDescendants } q \ (h,k)$$

by (simp add: recDescendants-def)

```

```

lemma closureL-p-None-subseteq-closureL:
  
$$p \neq q$$


$$\Rightarrow \text{closureL } q \ (h(p := \text{None}), k) \subseteq \text{closureL } q \ (h, k)$$

apply (rule subsetI)
apply (erule closureL.induct)
apply (rule closureL-basic)
apply clarsimp
apply (subgoal-tac d  $\in$  descendants qa (h,k))
apply (rule closureL-step,simp,simp)
apply (rule descendants-p-None-q,assumption+)
apply (simp add: descendants-def)

```



**by** (*case-tac*  $qa = p, \text{simp-all}$ )

**lemma** *recReachL-p-None-subseteq-recReachL*:

$p \neq q$   
 $\implies \text{recReachL } q \ (h(p := \text{None}), k) \subseteq \text{recReachL } q \ (h, k)$   
**apply** (*rule subsetI*)  
**apply** (*erule recReachL.induct*)  
**apply** (*rule recReachL-basic*)  
**apply** *clarsimp*  
**apply** (*subgoal-tac*  $d \in \text{recDescendants } qa \ (h, k)$ )  
**apply** (*rule recReachL-step, simp, simp*)  
**apply** (*rule recDescendants-p-None-q, assumption+*)  
**apply** (*simp add: recDescendants-def*)  
**by** (*case-tac*  $qa = p, \text{simp-all}$ )

**lemma** *p-in-closure-q-p-none*:

$\llbracket p \neq q; \text{closureL } q \ (h, k) \neq \text{closureL } q \ (h(p := \text{None}), k) \rrbracket$   
 $\implies p \in \text{closureL } q \ (h(p := \text{None}), k)$   
**apply** *auto*  
**apply** (*erule closureL.induct*)  
**apply** (*rule closureL-basic*)  
**apply** (*subgoal-tac*  $p \neq qa$ )  
**prefer** 2 **apply** *blast*  
**apply** (*rule closureL-step, simp*)  
**apply** (*simp add: descendants-def*)  
**apply** (*erule closureL-p-None-subseteq-closureL*)  
**by** *blast*

**lemma** *not-identityClosure-h-h-p-none-inter-not-empty-h*:

$\llbracket y \in \text{dom } E1; E1 \ x = \text{Some } (\text{Loc } p); h \ p = \text{Some } (j, C, vn);$   
 $\neg \text{identityClosure } (E1, E2) \ y \ (h, k) \ (h(p := \text{None}), k) \rrbracket$   
 $\implies \text{closure } (E1, E2) \ y \ (h, k) \cap \text{recReach } (E1, E2) \ x \ (h, k) \neq \{\}$   
**apply** (*simp only: identityClosure-def*)  
**apply** (*simp add: closure-def*)  
**apply** (*case-tac*  $E1 \ y, \text{simp-all}$ )  
**apply** (*case-tac*  $a, \text{simp-all}$ )  
**apply** (*simp add: recReach-def*)  
**apply** (*rename-tac*  $q$ )  
**apply** (*case-tac*  $p = q$ )  
**apply** *simp*  
**apply** (*subgoal-tac*  $q \in \text{recReachL } q \ (h, k)$ )  
**apply** (*subgoal-tac*  $\text{recReachL } q \ (h, k) \subseteq \text{closureL } q \ (h, k)$ )  
**apply** *blast*  
**apply** (*rule recReachL-subseteq-closureL*)  
**apply** (*rule recReachL-basic*)  
**apply** (*case-tac*

```

  closureL q (h, k) ≠ closureL q (h(p := None), k),simp-all)
  apply (frule-tac h=h and k=k in p-in-closure-q-p-none,simp)
  apply (frule-tac h=h and k=k in closureL-p-None-subseteq-closureL)
  apply (subgoal-tac
    p ∈ recReachL p (h, k))
  apply blast
  apply (rule recReachL-basic)
  apply (elim bexE)
  apply (case-tac pa = p,simp-all)
  apply (subgoal-tac
    p ∈ recReachL p (h, k))
  apply blast
  by (rule recReachL-basic)

```

**lemma** *dom-foldl-monotone-generic*:

```

  dom (foldl op ⊗ (empty ⊗ x) xs) =
    dom x ∪ dom (foldl op ⊗ empty xs)
  apply (subgoal-tac empty ⊗ x = x ⊗ empty,simp)
  apply (subgoal-tac foldl op ⊗ (x ⊗ empty) xs =
    x ⊗ foldl op ⊗ empty xs,simp)
  apply (rule union-dom-nonDisjointUnionEnv)
  apply (rule foldl-prop1)
  apply (subgoal-tac def-nonDisjointUnionEnv empty x)
  apply (erule nonDisjointUnionEnv-commutative)
  by (simp add: def-nonDisjointUnionEnv-def)

```

**lemma** *dom-Γi-in-Γcased-2* [rule-format]:

```

  length assert > 0
  → x ≠ y
  → length assert = length alts
  → (∀ i < length alts. y ∈ dom (snd (assert ! i))
    → y ∉ set (snd (extractP (fst (alts ! i))))
    → y ∈ dom (foldl op ⊗ empty
      (map (λ(Li, Γi). restrict-neg-map Γi (insert x (set Li)))
        (zip (map (snd ∘ extractP ∘ fst) alts) (map snd assert)))))
  apply (induct assert alts rule:list-induct2',simp-all)
  apply (rule impI)+
  apply (case-tac xs = [],simp)
  apply (rule impI)+
  apply (subst empty-nonDisjointUnionEnv)
  apply (subst dom-restrict-neg-map)
  apply force
  apply simp
  apply (rule allI, rule impI)
  apply (case-tac i,simp-all)
  apply (rule impI)+

```

```

apply (subst dom-foldl-monotone-generic)
apply (subst dom-restrict-neg-map)
apply force
apply (rule impI)
apply (erule-tac x=nat in allE,simp)
apply (rule impI)+
apply (subst dom-foldl-monotone-generic)
by blast

```

**lemma** *x-notin- $\Gamma$ -cased*:

```

  x  $\notin$  dom (foldl op  $\otimes$  empty
    (map ( $\lambda(Li, \Gamma i).$  restrict-neg-map  $\Gamma i$  (insert x (set Li)))
      (zip (map (snd  $\circ$  extractP  $\circ$  fst) alts) (map snd assert))))))
apply (induct-tac assert alts rule: list-induct2',simp-all)
apply (subgoal-tac
  dom (foldl op  $\otimes$  (empty  $\otimes$  restrict-neg-map (snd xa) (insert x (set (snd (extractP
    (fst y)))))))
    (map ( $\lambda(Li, \Gamma i).$  restrict-neg-map  $\Gamma i$  (insert x (set Li))) (zip (map
      ( $\lambda a.$  snd (extractP (fst a))) ys) (map snd xs)))))) =
  dom (restrict-neg-map (snd xa) (insert x (set (snd (extractP (fst y))))))  $\cup$ 
  dom (foldl op  $\otimes$  empty (map ( $\lambda(Li, \Gamma i).$  restrict-neg-map  $\Gamma i$  (insert x (set Li)))
    (zip (map ( $\lambda a.$  snd (extractP (fst a))) ys) (map snd
      xs))))),simp)
apply (subst dom-restrict-neg-map,blast)
by (rule dom-foldl-monotone-generic)

```

**lemma**  *$\Gamma$ -case-x-is-d*:

```

  [  $\Gamma =$  foldl op  $\otimes$  empty
    (map ( $\lambda(Li, \Gamma i).$  restrict-neg-map  $\Gamma i$  (insert x (set Li))) (zip (map
      (snd  $\circ$  extractP  $\circ$  fst) alts) (map snd assert))) +
    [x  $\mapsto$  d''] ]
 $\implies \Gamma x =$  Some d''
apply (simp add: disjointUnionEnv-def)
apply (simp add: unionEnv-def)
apply (rule impI)
apply (insert x-notin- $\Gamma$ -cased)
by force

```

**lemma** *disjointUnionEnv-G-G'-G-x*:

```

  [ x  $\notin$  dom G'; def-disjointUnionEnv G G' ]
 $\implies (G + G') x = G x$ 
apply (simp add: disjointUnionEnv-def)
apply (simp add: unionEnv-def)
apply (simp add: def-disjointUnionEnv-def)
by force

```

```

lemma restrict-neg-map-not-s:
   $\llbracket G \ y \neq \text{Some } s''; x \neq y ; y \notin L \rrbracket$ 
   $\implies \text{restrict-neg-map } G \ (\text{insert } x \ L) \ y \neq \text{Some } s''$ 
by (simp add: restrict-neg-map-def)

declare def-nonDisjointUnionEnvList.simps [simp del]

lemma Otimes-prop-cased-not-s [rule-format]:
  length assert > 0
   $\longrightarrow \text{length } \text{assert} = \text{length } \text{alts}$ 
   $\longrightarrow \text{def-nonDisjointUnionEnvList } (\text{map } (\lambda(Li, \Gamma i). \text{restrict-neg-map } \Gamma i \ (\text{insert } x \ (\text{set } Li))))$ 
   $\quad (\text{zip } (\text{map } (\text{snd} \circ \text{extractP} \circ \text{fst}) \ \text{alts}) \ (\text{map } \text{snd}$ 
  assert)))
   $\longrightarrow \text{def-disjointUnionEnv}$ 
   $\quad (\text{foldl } \text{op} \otimes \text{empty}$ 
   $\quad \quad (\text{map } (\lambda(Li, \Gamma i). \text{restrict-neg-map } \Gamma i \ (\text{insert } x \ (\text{set } Li))))$ 
   $\quad \quad (\text{zip } (\text{map } (\text{snd} \circ \text{extractP} \circ \text{fst}) \ \text{alts}) \ (\text{map } \text{snd } \text{assert}))))$ 
   $\quad [x \mapsto d'']$ 
   $\longrightarrow (\forall \ i < \text{length } \text{alts}. \ y \in \text{dom } (\text{snd } (\text{assert } ! \ i)))$ 
   $\longrightarrow \text{snd } (\text{assert } ! \ i) \ y \neq \text{Some } s''$ 
   $\longrightarrow y \notin \text{set } (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! \ i))))$ 
   $\longrightarrow (\text{foldl } \text{op} \otimes \text{empty}$ 
   $\quad (\text{map } (\lambda(Li, \Gamma i). \text{restrict-neg-map } \Gamma i \ (\text{insert } x \ (\text{set } Li))))$ 
   $\quad (\text{zip } (\text{map } (\text{snd} \circ \text{extractP} \circ \text{fst}) \ \text{alts}) \ (\text{map } \text{snd } \text{assert}))) +$ 
   $\quad [x \mapsto d''] \ y \neq \text{Some } s''$ )
apply (case-tac x = y, simp)
apply (rule impI) +
apply (rule allI)
apply (rule impI) +
apply (subgoal-tac
  (foldl op  $\otimes$  empty
    (map  $(\lambda(Li, \Gamma i). \text{restrict-neg-map } \Gamma i \ (\text{insert } x \ (\text{set } Li)))$ 
    (zip (map  $(\text{snd} \circ \text{extractP} \circ \text{fst}) \ \text{alts}$ ) (map snd assert)))) +
    [x  $\mapsto d''$ ] x = Some d'', simp)
apply (rule-tac
   $\Gamma = (\text{foldl } \text{op} \otimes \text{empty}$ 
    (map  $(\lambda(Li, \Gamma i). \text{restrict-neg-map } \Gamma i \ (\text{insert } x \ (\text{set } Li)))$ 
    (zip (map  $(\text{snd} \circ \text{extractP} \circ \text{fst}) \ \text{alts}$ ) (map snd assert)))) +
    [x  $\mapsto d''$ ] in  $\Gamma$ -case-x-is-d, force)
apply (induct assert alts rule:list-induct2', simp-all)
apply (rule impI) +
apply (case-tac xs = [], simp)
apply (rule impI)
apply (subst empty-nonDisjointUnionEnv)
apply (simp add: disjointUnionEnv-def)
apply (simp add: unionEnv-def)

```

```

  apply (rule impI)+
  apply (simp add: restrict-neg-map-def)
apply simp
apply (drule mp)
  apply (simp add: def-nonDisjointUnionEnvList.simps)
  apply (simp add: Let-def)
apply (drule mp)
  apply (simp add: def-disjointUnionEnv-def)
  apply (subst (asm) dom-foldl-monotone-generic)
  apply blast
apply (rule allI, rule impI)
apply (case-tac i, simp-all)
  apply (rule impI)+
  apply (subst disjointUnionEnv-G-G'-G-x, simp, simp)
  apply (subst nonDisjointUnionEnv-commutative)
    apply (simp add: def-nonDisjointUnionEnv-def)
  apply (subst foldl-prop1)
  apply (subst nonDisjointUnionEnv-prop5)
    apply (subst dom-restrict-neg-map, force)
    apply (rule restrict-neg-map-not-s, assumption+, simp)
  apply (rule impI)+
  apply (rotate-tac 5)
  apply (erule-tac x=nat in allE, simp)
  apply (subst disjointUnionEnv-G-G'-G-x, simp, simp)
  apply (subst nonDisjointUnionEnv-commutative)
    apply (simp add: def-nonDisjointUnionEnv-def)
  apply (subst foldl-prop1)
  apply (subst (asm) disjointUnionEnv-G-G'-G-x, simp)
    apply (simp add: def-disjointUnionEnv-def)
    apply (rule x-notin-Γ-cased)
  apply (subst nonDisjointUnionEnv-prop6)
    apply (simp add: def-nonDisjointUnionEnvList.simps)
    apply (simp add: Let-def)
  apply (subgoal-tac
    y ∈ dom (foldl op ⊗ empty
      (map (λ(Li, Γi). restrict-neg-map Γi (insert x (set Li)))
        (zip (map (snd ∘ extractP ∘ fst) ys) (map snd xs))))), simp)
  apply (rule dom-Γi-in-Γ-cased-2)
by (force, assumption+, simp)

```

**lemma** *closure-monotone-extend-def-extend*:  
 $\llbracket \text{def-extend } E \text{ (snd (extractP (fst (alts ! i)))) } vs;$   
 $x \in \text{dom } E;$

```

    length alts > 0;
    i < length alts ]
  => closure (E, E') x (h, k) = closure (extend E (snd (extractP (fst (alts ! i))))
vs, E') x (h, k)
apply (simp add: def-extend-def)
apply (elim conjE)
apply (subgoal-tac x ∉ set (snd (extractP (fst (alts ! i)))))
apply (subgoal-tac
  E x = extend E (snd (extractP (fst (alts ! i)))) vs x)
apply (simp add: closure-def)
apply (rule extend-monotone-i)
apply (simp, simp, simp)
by blast

```

**lemma** *recReach-monotone-extend-def-extend*:

```

  [ def-extend E (snd (extractP (fst (alts ! i)))) vs;
    x ∈ dom E;
    length alts > 0;
    i < length alts ]
  => recReach (E, E') x (h, k) = recReach (extend E (snd (extractP (fst (alts !
i)))) vs, E') x (h, k)
apply (simp add: def-extend-def)
apply (elim conjE)
apply (subgoal-tac x ∉ set (snd (extractP (fst (alts ! i)))))
apply (subgoal-tac
  E x = extend E (snd (extractP (fst (alts ! i)))) vs x)
apply (simp add: recReach-def)
apply (rule extend-monotone-i)
apply (simp, simp, simp)
by blast

```

**lemma** *identityClosure-h-p-none-no-identityClosure-hh*:

```

  [ y ∈ dom E1; E1 x = Some (Loc p); h p = Some (j, C, vs);
    i < length alts; length assert = length alts; length alts > 0;
    def-extend E1 (snd (extractP (fst (alts ! i)))) vs;
    identityClosure (E1, E2) y (h, k) (h(p := None), k);
    ¬ identityClosure (E1, E2) y (h, k) (hh, k) ]
  => ¬ identityClosure (extend E1 (snd (extractP (fst (alts ! i)))) vs, E2) y (h(p
:= None), k) (hh, k)
apply (simp add: identityClosure-def)
apply (rule impI)
apply (elim conjE)
apply (subgoal-tac
  y ∉ set (snd (extractP (fst (alts ! i)))))
prefer 2 apply (simp add: def-extend-def, elim conjE, blast)
apply (frule-tac E=E1 and vs=vs in extend-monotone-i, simp, assumption+)
apply (subgoal-tac

```

```

  closure (E1, E2) y (h(p := None), k) = closure (E1, E2) y (hh, k),simp)
prefer 2 apply (subst closure-monotone-extend-def-extend,assumption+,simp,assumption+)+
apply (subst (asm) closure-monotone-extend-def-extend [where E=E1],assumption+,simp,assumption+)+
apply (simp add: closure-def)
apply (case-tac E1 y,simp-all)
apply (case-tac extend E1 (snd (extractP (fst (alts ! i)))) vs y, simp-all)
apply (case-tac aa, simp-all)
apply (rename-tac q)
apply (case-tac p = q,simp-all)
apply clarsimp
apply (rule-tac x=pa in bexI)
prefer 2 apply simp
apply (rule conjI)
apply (rule impI)
apply (erule-tac x=p in ballE)
prefer 2 apply simp
apply clarsimp
apply clarsimp
apply clarsimp
apply (rule-tac x=pa in bexI)
prefer 2 apply simp
apply (rule conjI)
apply (erule-tac x=pa in ballE)
prefer 2 apply simp
apply clarsimp
by clarsimp

```

```

lemma dom-restrict-neg-map:
  dom (restrict-neg-map m A) = dom m - (dom m ∩ A)
apply (simp add: restrict-neg-map-def)
apply auto
by (split split-if-asm,simp-all)

```

```

lemma dom-foldl-monotone-generic:
  dom (foldl op ⊗ (empty ⊗ x) xs) =
    dom x ∪ dom (foldl op ⊗ empty xs)
apply (subgoal-tac empty ⊗ x = x ⊗ empty,simp)
apply (subgoal-tac foldl op ⊗ (x ⊗ empty) xs =
      x ⊗ foldl op ⊗ empty xs,simp)
apply (rule union-dom-nonDisjointUnionEnv)
apply (rule foldl-prop1)
apply (subgoal-tac def-nonDisjointUnionEnv empty x)
apply (erule nonDisjointUnionEnv-commutative)
by (simp add: def-nonDisjointUnionEnv-def)

```

```

lemma dom-foldl-disjointUnionEnv-monotone-generic-2:
  dom (foldl op ⊗ (empty ⊗ y) ys + A) =
    dom y ∪ dom (foldl op ⊗ empty ys) ∪ dom A

```

```

apply (subgoal-tac empty  $\otimes$   $y = y \otimes \text{empty}$ ,simp)
apply (subgoal-tac foldl op  $\otimes$  ( $y \otimes \text{empty}$ )  $ys =$ 
       $y \otimes \text{foldl op} \otimes \text{empty } ys$ ,simp)
apply (subst dom-disjointUnionEnv-monotone)
apply (subst union-dom-nonDisjointUnionEnv)
apply simp
apply (rule foldl-prop1)
apply (subgoal-tac def-nonDisjointUnionEnv empty  $y$ )
apply (erule nonDisjointUnionEnv-commutative)
by (simp add: def-nonDisjointUnionEnv-def)

lemma dom- $\Gamma i$ -in- $\Gamma$  cased [rule-format]:
  length assert > 0
   $\longrightarrow$  length assert = length alts
   $\longrightarrow$  def-disjointUnionEnv
    (foldl op  $\otimes$  empty
      (map ( $\lambda(Li, \Gamma i).$  restrict-neg-map  $\Gamma i$  (insert  $x$  (set  $Li$ )))
        (zip (map (snd  $\circ$  extractP  $\circ$  fst) alts) (map snd assert))))
    [ $x \mapsto d''$ ]
   $\longrightarrow$  ( $\forall i < \text{length alts}. y \in \text{dom} (\text{snd} (\text{assert } ! i))$ )
   $\longrightarrow$   $y \notin \text{set} (\text{snd} (\text{extractP} (\text{fst} (\text{alts } ! i))))$ 
   $\longrightarrow$   $y \in \text{dom} (\text{foldl op} \otimes \text{empty}$ 
    (map ( $\lambda(Li, \Gamma i).$  restrict-neg-map  $\Gamma i$  (insert  $x$  (set  $Li$ )))
      (zip (map (snd  $\circ$  extractP  $\circ$  fst) alts) (map snd assert)))) +
    [ $x \mapsto d''$ ]))

apply (induct assert alts rule:list-induct2',simp-all)
apply (rule impI)+
apply (case-tac  $xs = []$ ,simp)
apply (rule impI)
apply (subst empty-nonDisjointUnionEnv)
apply (subst union-dom-disjointUnionEnv)
apply (subst (asm) empty-nonDisjointUnionEnv)
apply simp
apply (subst dom-restrict-neg-map)
apply force
apply simp
apply (drule mp)
apply (simp add: def-disjointUnionEnv-def)
apply (subst (asm) dom-foldl-monotone-generic)
apply blast
apply (rule allI, rule impI)
apply (case-tac  $i$ ,simp-all)
apply (rule impI)
apply (subst dom-foldl-disjointUnionEnv-monotone-generic-2)
apply (subst dom-restrict-neg-map)
apply force
apply (rule impI)
apply (rotate-tac 3)
apply (erule-tac  $x=\text{nat}$  in allE,simp)

```



```

apply (subst dom-foldl-disjointUnionEnv-monotone-generic-2)
apply (subst (asm) union-dom-disjointUnionEnv)
apply (simp add: def-disjointUnionEnv-def)
apply (subst (asm) dom-foldl-monotone-generic)
apply blast
by blast

```

**lemma** *identityClosure-h-h-p-none*:

```

[[ identityClosure (E1, E2) y (h, k) (h(p := None), k);
  ¬ identityClosure (E1, E2) y (h, k) (hh, k);
  i < length alts; length assert = length alts; length alts > 0;
  def-extend E1 (snd (extractP (fst (alts ! i)))) vs;
  y ∈ dom E1; E1 x = Some (Loc p); h p = Some (j, C, vs);
  def-nonDisjointUnionEnvList
    (map (λ(Li, Γi). restrict-neg-map Γi (insert x (set Li))) (zip (map (snd ∘
extractP ∘ fst) alts) (map snd assert)))));
  def-disjointUnionEnv
    (foldl op ⊗ empty
      (map (λ(Li, Γi). restrict-neg-map Γi (insert x (set Li))) (zip (map (snd
∘ extractP ∘ fst) alts) (map snd assert))))
    [x ↦ d''];
  Γ = nonDisjointUnionEnvList
    (map (λ(Li, Γi). restrict-neg-map Γi (set Li ∪ {x})) (zip (map (snd ∘
extractP ∘ fst) alts) (map snd assert))) +
    [x ↦ d''];
  (∀ x ∈ dom (fst (extend E1 (snd (extractP (fst (alts ! i)))) vs, E2)).
    ¬ identityClosure (extend E1 (snd (extractP (fst (alts ! i)))) vs, E2) x (h(p
:= None), k) (hh, k) →
    x ∈ dom (snd (assert ! i)) ∧ snd (assert ! i) x ≠ Some s'') ]
  ⇒ y ∈ dom Γ ∧ Γ y ≠ Some s''
apply (subgoal-tac y ∉ set (snd (extractP (fst (alts ! i)))))
prefer 2 apply (simp add: def-extend-def, blast)
apply (frule-tac hh=hh and x=x in
  identityClosure-h-p-none-no-identityClosure-hh, assumption+)
apply (erule-tac x=y in ballE, simp)
apply (elim conjE)
apply (rule conjI)
apply (rule-tac i=i in dom-Γi-in-Γ-cased, simp, assumption+)
apply (rule-tac alts=alts and x=x and assert=assert in Otimes-prop-cased-not-s)
apply force apply assumption+
by (simp add: extend-def)

```

**lemma** *P6-CASED*:

```

[[ Γ = nonDisjointUnionEnvList
  (map (λ(Li, Γi). restrict-neg-map Γi (set Li ∪ {x}))
    (zip (map (snd ∘ extractP ∘ fst) alts) (map snd assert))) +
  [x ↦ d''];

```

```

def-nonDisjointUnionEnvList
  (map (λ(Li, Γi). restrict-neg-map Γi (insert x (set Li))) (zip (map (snd ∘
extractP ∘ fst) alts) (map snd assert)));
def-disjointUnionEnv
  (foldl op ⊗ empty
    (map (λ(Li, Γi). restrict-neg-map Γi (insert x (set Li))) (zip (map (snd
∘ extractP ∘ fst) alts) (map snd assert))))
  [x ↦ d''];
dom Γ ⊆ dom E1;
E1 x = Some (Loc p); h p = Some (j, C, vs); x ∈ dom Γ;
def-extend E1 (snd (extractP (fst (alts ! i)))) vs;
i < length alts; alts ≠ [];
length assert = length alts;
shareRec (fst (assert ! i)) (snd (assert ! i)) (extend E1 (snd (extractP (fst
(alts ! i)))) vs, E2) (h(p:=None), k) (hh, k);
∀ y ∈ dom (fst (E1, E2)).
  ∀ z ∈ insert x (⋃ i < length alts fst (assert ! i) - set (snd (extractP (fst
(alts ! i))))).
    Γ z = Some d'' ∧
      closure (E1, E2) y (h, k) ∩ recReach (E1, E2) z (h, k) ≠ {} ⟶
        y ∈ dom Γ ∧ Γ y ≠ Some s'';
    y ∈ dom (fst (E1, E2));
    ¬ identityClosure (E1, E2) y (h, k) (hh, k)]
  ⟹ y ∈ dom Γ ∧ Γ y ≠ Some s''
apply (case-tac ¬ identityClosure (E1, E2) y (h, k) (h(p:=None), k))

apply (frule not-identityClosure-h-h-p-none-inter-not-empty-h, simp, assumption+, simp)
apply (erule-tac x=y in ballE)
prefer 2 apply simp
apply (erule-tac x=x in ballE)
prefer 2 apply simp
apply (drule mp)
apply (rule conjI)
apply (rule Γ-case-x-is-d, force)
apply simp
apply simp

apply (rule identityClosure-h-h-p-none)
apply (simp, simp, assumption+, simp, assumption+, simp, assumption+)
by (simp add: shareRec-def)

```

**lemma** P5-CASED:

```

  [(E1, E2) ⊢ h, k, td, CaseD VarE x () Of alts () ↓ hh, k, v, r ;
    Γ = disjointUnionEnv
      (nonDisjointUnionEnvList ((map (λ(Li, Γi). restrict-neg-map Γi (set
Li ∪ {x}))))

```

$(\text{zip } (\text{map } (\text{snd } \circ \text{extractP } \circ \text{fst}) \text{ alts}) (\text{map } \text{snd } \text{assert})))$   
 $(\text{empty}(x \mapsto d''))$ ;  
 $x \in \text{dom } \Gamma$ ;  
 $\text{dom } \Gamma \subseteq \text{dom } E1$ ;  
 $(\bigcup_i < \text{length alts } \text{fst } (\text{assert } ! i) - \text{set } (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! i)))))) \subseteq \text{dom } \Gamma$ ;  
 $\text{fv } (\text{CaseD } \text{VarE } x () \text{ Of alts } ()) \subseteq \text{insert } x (\bigcup_i < \text{length alts } \text{fst } (\text{assert } ! i) - \text{set } (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! i))))))$ ;  
 $\text{wellFormed } (\text{insert } x (\bigcup_i < \text{length alts } \text{fst } (\text{assert } ! i) - \text{set } (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! i))))))$   
 $\Gamma (\text{CaseD } \text{VarE } x () \text{ Of alts } ()) \parallel$   
 $\implies \forall xa \in \text{dom } (\text{fst } (E1, E2)).$   
 $\quad \forall z \in \text{insert } x (\bigcup_i < \text{length alts } \text{fst } (\text{assert } ! i) - \text{set } (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! i))))).$   
 $\quad \Gamma z = \text{Some } d'' \wedge$   
 $\quad \text{closure } (E1, E2) xa (h, k) \cap \text{recReach } (E1, E2) z (h, k) \neq \{\}$   $\longrightarrow$   
 $\quad xa \in \text{dom } \Gamma \wedge \Gamma xa \neq \text{Some } s''$   
**apply** (*simp only: wellFormed-def*)  
**apply** (*erule-tac x=E1 in allE*)  
**apply** (*erule-tac x=E2 in allE*)  
**apply** (*erule-tac x=h in allE*)  
**apply** (*erule-tac x=k in allE*)  
**apply** (*erule-tac x=td in allE*)  
**apply** (*erule-tac x=hh in allE*)  
**apply** (*erule-tac x=v in allE*)  
**apply** (*erule-tac x=r in allE*)  
**apply** (*drule mp, simp*)  
**by** *simp*

**lemma** *P5-P6-CASED:*

$\parallel (E1, E2) \vdash h, k, td, \text{CaseD } \text{VarE } x () \text{ Of alts } () \Downarrow hh, k, v, r ;$   
 $\Gamma = \text{disjointUnionEnv}$   
 $(\text{nonDisjointUnionEnvList } ((\text{map } (\lambda(Li, \Gamma i). \text{restrict-neg-map } \Gamma i (\text{set } Li \cup \{x\})))$   
 $(\text{zip } (\text{map } (\text{snd } \circ \text{extractP } \circ \text{fst}) \text{ alts}) (\text{map } \text{snd } \text{assert})))$   
 $(\text{empty}(x \mapsto d''))$ ;  
 $\text{def-nonDisjointUnionEnvList}$   
 $(\text{map } (\lambda(Li, \Gamma i). \text{restrict-neg-map } \Gamma i (\text{insert } x (\text{set } Li))) (\text{zip } (\text{map } (\text{snd } \circ \text{extractP } \circ \text{fst}) \text{ alts}) (\text{map } \text{snd } \text{assert}))))$ ;  
 $\text{def-disjointUnionEnv}$   
 $(\text{foldl } \otimes \text{empty}$   
 $(\text{map } (\lambda(Li, \Gamma i). \text{restrict-neg-map } \Gamma i (\text{insert } x (\text{set } Li)))$   
 $(\text{zip } (\text{map } (\text{snd } \circ \text{extractP } \circ \text{fst}) \text{ alts}) (\text{map } \text{snd } \text{assert}))))$   
 $[x \mapsto d'']$ ;  
 $\text{dom } \Gamma \subseteq \text{dom } E1$ ;  $E1 x = \text{Some } (\text{Loc } p)$ ;  $h p = \text{Some } (j, C, vs)$ ;  
 $x \in \text{dom } \Gamma$ ;  
 $(\bigcup_i < \text{length alts } \text{fst } (\text{assert } ! i) - \text{set } (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! i)))))) \subseteq \text{dom } \Gamma$  ;

$fv (CaseD \text{ VarE } x () \text{ Of } alts ()) \subseteq insert\ x (\bigcup_{i < length\ alts} fst\ (assert\ !\ i) - set\ (snd\ (extractP\ (fst\ (alts\ !\ i)))));$   
 $i < length\ alts;$   
 $def\ extend\ E1\ (snd\ (extractP\ (fst\ (alts\ !\ i))))\ vs;$   
 $alts \neq []; length\ assert = length\ alts;$   
 $wellFormed\ (insert\ x (\bigcup_{i < length\ alts} fst\ (assert\ !\ i) - set\ (snd\ (extractP\ (fst\ (alts\ !\ i)))));$   
 $\Gamma (CaseD \text{ VarE } x () \text{ Of } alts ());$   
 $(E1, E2) \vdash h, k, td, CaseD \text{ VarE } x () \text{ Of } alts () \Downarrow hh, k, v, r;$   
 $shareRec\ (fst\ (assert\ !\ i))\ (snd\ (assert\ !\ i))\ (extend\ E1\ (snd\ (extractP\ (fst\ (alts\ !\ i))))\ vs, E2)\ (h(p:=None), k)\ (hh, k) \parallel$   
 $\implies shareRec\ (insert\ x (\bigcup_{i < length\ alts} fst\ (assert\ !\ i) - set\ (snd\ (extractP\ (fst\ (alts\ !\ i)))));$   
 $\Gamma (E1, E2)\ (h, k)\ (hh, k)$   
**apply** (*simp* (*no-asm*) *only*: *shareRec-def*)  
**apply** (*rule conjI*)  
**apply** (*rule P5-CASED,assumption+*)  
**apply** (*rule ballI,rule impI*)  
**apply** (*frule P5-CASED,assumption+*)  
**by** (*rule P6-CASED [where p=p],assumption+*)

**lemma** *P5-f-n-CASED*:

$\parallel (E1, E2) \vdash h, k, CaseD \text{ VarE } x () \text{ Of } alts () \Downarrow (f,n) hh, k, v;$   
 $\Gamma = disjointUnionEnv$   
 $(nonDisjointUnionEnvList\ ((map\ (\lambda(Li,\Gamma i). restrict\ neg\ map\ \Gamma i\ (set\ Li \cup \{x\}))))$   
 $(zip\ (map\ (snd\ o\ extractP\ o\ fst)\ alts)\ (map\ snd\ assert))))$   
 $(empty(x \mapsto d''));$   
 $x \in dom\ \Gamma;$   
 $dom\ \Gamma \subseteq dom\ E1;$   
 $(\bigcup_{i < length\ alts} fst\ (assert\ !\ i) - set\ (snd\ (extractP\ (fst\ (alts\ !\ i)))) \subseteq dom\ \Gamma;$   
 $fv (CaseD \text{ VarE } x () \text{ Of } alts ()) \subseteq insert\ x (\bigcup_{i < length\ alts} fst\ (assert\ !\ i) - set\ (snd\ (extractP\ (fst\ (alts\ !\ i)))));$   
 $wellFormedDepth\ f\ n\ (insert\ x (\bigcup_{i < length\ alts} fst\ (assert\ !\ i) - set\ (snd\ (extractP\ (fst\ (alts\ !\ i)))));$   
 $\Gamma (CaseD \text{ VarE } x () \text{ Of } alts ()) \parallel$   
 $\implies \forall xa \in dom\ (fst\ (E1, E2)).$   
 $\forall z \in insert\ x (\bigcup_{i < length\ alts} fst\ (assert\ !\ i) - set\ (snd\ (extractP\ (fst\ (alts\ !\ i))))).$   
 $\Gamma\ z = Some\ d'' \wedge$   
 $closure\ (E1, E2)\ xa\ (h, k) \cap recReach\ (E1, E2)\ z\ (h, k) \neq \{\}$   $\longrightarrow$   
 $xa \in dom\ \Gamma \wedge \Gamma\ xa \neq Some\ s''$   
**apply** (*simp only*: *wellFormedDepth-def*)  
**apply** (*erule-tac x=E1 in allE*)  
**apply** (*erule-tac x=E2 in allE*)  
**apply** (*erule-tac x=h in allE*)  
**apply** (*erule-tac x=k in allE*)

**apply** (*erule-tac*  $x=hh$  **in**  $allE$ )  
**apply** (*erule-tac*  $x=v$  **in**  $allE$ )  
**apply** (*drule*  $mp, simp$ )  
**by**  $simp$

**lemma** *P5-P6-f-n-CASED*:

$\llbracket (E1, E2) \vdash h, k, CaseD \text{ VarE } x () \text{ Of alts } () \Downarrow (f, n) \text{ } hh, k, v;$   
 $\Gamma = disjointUnionEnv$   
 $(nonDisjointUnionEnvList ((map (\lambda(Li, \Gamma i). restrict-neg-map \Gamma i (set$   
 $Li \cup \{x\}))))$   
 $(zip (map (snd \circ extractP \circ fst) alts) (map snd assert))))$   
 $(empty(x \mapsto d'));$   
 $def-nonDisjointUnionEnvList$   
 $(map (\lambda(Li, \Gamma i). restrict-neg-map \Gamma i (insert x (set Li))) (zip (map (snd \circ$   
 $extractP \circ fst) alts) (map snd assert))));$   
 $def-disjointUnionEnv$   
 $(foldl op \otimes empty$   
 $(map (\lambda(Li, \Gamma i). restrict-neg-map \Gamma i (insert x (set Li)))$   
 $(zip (map (snd \circ extractP \circ fst) alts) (map snd assert))))$   
 $[x \mapsto d'];$   
 $dom \Gamma \subseteq dom E1; E1 x = Some (Loc p); h p = Some (j, C, vs);$   
 $x \in dom \Gamma;$   
 $(\bigcup_{i < length alts} fst (assert ! i) - set (snd (extractP (fst (alts ! i))))) \subseteq dom$   
 $\Gamma ;$   
 $fv (CaseD \text{ VarE } x () \text{ Of alts } ()) \subseteq insert x (\bigcup_{i < length alts} fst (assert ! i) -$   
 $set (snd (extractP (fst (alts ! i)))))$   
 $i < length alts;$   
 $def-extend E1 (snd (extractP (fst (alts ! i)))) vs;$   
 $alts \neq []; length assert = length alts;$   
 $wellFormedDepth f n (insert x (\bigcup_{i < length alts} fst (assert ! i) - set (snd$   
 $(extractP (fst (alts ! i)))))$   
 $\Gamma (CaseD \text{ VarE } x () \text{ Of alts } ());$   
 $shareRec (fst (assert ! i)) (snd (assert ! i)) (extend E1 (snd (extractP (fst$   
 $(alts ! i)))) vs, E2) (h(p:=None), k) (hh, k)\rrbracket$   
 $\implies shareRec (insert x (\bigcup_{i < length alts} fst (assert ! i) - set (snd (extractP$   
 $(fst (alts ! i)))))$   
 $\Gamma (E1, E2) (h, k) (hh, k)$   
**apply** (*simp* (*no-asm*) *only*: *shareRec-def*)  
**apply** (*rule* *conjI*)  
**apply** (*rule* *P5-f-n-CASED*, *assumption* +)  
**apply** (*rule* *ballI*, *rule* *impI*)  
**apply** (*frule* *P5-f-n-CASED*, *assumption* +)  
**by** (*rule* *P6-CASED* [**where**  $p=p$ ], *assumption* +)

**axioms** *SafeRASem-extend-heaps*:

$(E1, E2) \vdash h, k, td, e \Downarrow hh, k, v, r$   
 $\implies \text{extend-heaps } (h, k) (hh, k)$

**axioms** *Lemma4-consistent*:

$\text{extend-heaps } (h, k) (h', k')$   
 $\implies \forall \vartheta 1 \vartheta 2 \eta E1 E2.$   
 $\quad \text{consistent } (\vartheta 1, \vartheta 2) \eta (E1, E2) h$   
 $\quad \longrightarrow \text{consistent } (\vartheta 1, \vartheta 2) \eta (E1, E2) h'$

**axioms** *consistent-identityClosure*:

$\text{consistent } (\vartheta 1, \vartheta 2) \eta (E1, E2) h$   
 $\longrightarrow \text{consistent } (\vartheta 1, \vartheta 2) \eta (E1, E2) (h' \mid \{p \in \text{dom } h'. \text{fst } (\text{the } (h' p)) \leq k\})$   
 $\implies \forall x \in \text{dom } E1. \text{identityClosure } (E1, E2) x (h, k) (h' \mid \{p \in \text{dom } h'. \text{fst } (\text{the } (h' p)) \leq k\}, k)$

**lemma** *P5-P6-APP*:

$\llbracket (E1, E2) \vdash h, k, td, \text{AppE f as rs'} () \Downarrow hh, k, v, r ;$   
 $hh = h' \mid \{p \in \text{dom } h'. \text{fst } (\text{the } (h' p)) \leq k\};$   
 $\text{dom } \Gamma \subseteq \text{dom } E1; \forall a \in \text{set as. atom } a;$   
 $\text{length } xs = \text{length } ms; \text{length } xs = \text{length } as;$   
 $\text{wellFormed } (fvs' as) \Gamma (\text{AppE f as rs'} ());$   
 $fvs' as \subseteq \text{dom } \Gamma;$   
 $\text{nonDisjointUnionSafeEnvList } (\text{maps-of } (\text{zip } (\text{map atom2var as}) ms)) \subseteq_m \Gamma \rrbracket$   
 $\implies \text{shareRec } (fvs' as) \Gamma (E1, E2) (h, k) (hh, k)$

**apply** (*simp add: shareRec-def*)

**apply** (*rule conjI*)

**apply** (*simp only: wellFormed-def*)

**apply** (*erule-tac x=E1 in allE*)

**apply** (*erule-tac x=E2 in allE*)

**apply** (*erule-tac x=h in allE*)

**apply** (*erule-tac x=k in allE*)

**apply** (*erule-tac x=td in allE*)

**apply** (*erule-tac x= h' \mid \{p \in \text{dom } h'. \text{fst } (\text{the } (h' p)) \leq k\} in allE*)

**apply** (*erule-tac x=v in allE*)

**apply** (*erule-tac x=r in allE*)

**apply** (*drule mp*)

**apply** (*rule conjI, simp*)

**apply** (*rule conjI, simp*)

**apply** (*rule conjI*)

**apply** *simp*

**apply** *simp*

**apply** *simp*

**apply** (*frule SafeRASem-extend-heaps*)

**apply** (*frule Lemma4-consistent*)

**apply** (*erule-tac*  $x=\vartheta 1$  **in** *allE*)  
**apply** (*erule-tac*  $x=\vartheta 2$  **in** *allE*)  
**apply** (*erule-tac*  $x=\eta$  **in** *allE*)  
**apply** (*erule-tac*  $x=E1$  **in** *allE*)  
**apply** (*erule-tac*  $x=E2$  **in** *allE*)  
**apply** (*frule* *consistent-identityClosure*)  
**by** *simp*

**axioms** *SafeDepthSem-extend-heaps*:

$(E1, E2) \vdash h, k, e \Downarrow (f, n) \ hh, k, v$   
 $\implies \text{extend-heaps } (h, k) \ (hh, k)$

**lemma** *P5-P6-f-n-APP*:

$\llbracket (E1, E2) \vdash h, k, \text{AppE } f \text{ as } rs' () \Downarrow (f, n) \ hh, k, v;$   
 $hh = h' \mid \{p \in \text{dom } h'. \text{fst } (the \ (h' \ p)) \leq k\};$   
 $\text{dom } \Gamma \subseteq \text{dom } E1; \forall a \in \text{set } as. \text{atom } a;$   
 $\text{length } xs = \text{length } ms; \text{length } xs = \text{length } as;$   
 $\text{wellFormedDepth } f \ n \ (fvs' \ as) \ \Gamma \ (\text{AppE } f \text{ as } rs' ());$   
 $fvs' \ as \subseteq \text{dom } \Gamma;$   
 $\text{nonDisjointUnionSafeEnvList } (\text{maps-of } (\text{zip } (\text{map } \text{atom2var } as) \ ms)) \subseteq_m \Gamma \rrbracket$   
 $\implies \text{shareRec } (fvs' \ as) \ \Gamma \ (E1, E2) \ (h, k) \ (hh, k)$

**apply** (*simp* *add: shareRec-def*)

**apply** (*rule* *conjI*)

**apply** (*simp* *only: wellFormedDepth-def*)  
**apply** (*erule-tac*  $x=E1$  **in** *allE*)  
**apply** (*erule-tac*  $x=E2$  **in** *allE*)  
**apply** (*erule-tac*  $x=h$  **in** *allE*)  
**apply** (*erule-tac*  $x=k$  **in** *allE*)  
**apply** (*erule-tac*  $x= h' \mid \{p \in \text{dom } h'. \text{fst } (the \ (h' \ p)) \leq k\}$  **in** *allE*)  
**apply** (*erule-tac*  $x=v$  **in** *allE*)  
**apply** (*drule* *mp*)  
**apply** (*rule* *conjI, simp*)  
**apply** (*rule* *conjI, simp*)  
**apply** (*rule* *conjI*)  
**apply** *simp*  
**apply** *simp*  
**apply** *simp*

**apply** (*frule* *SafeDepthSem-extend-heaps*)

**apply** (*frule* *Lemma4-consistent*)

**apply** (*erule-tac*  $x=\vartheta 1$  **in** *allE*)

**apply** (*erule-tac*  $x=\vartheta 2$  **in** *allE*)

**apply** (*erule-tac*  $x=\eta$  **in** *allE*)

**apply** (*erule-tac*  $x=E1$  **in** *allE*)

**apply** (*erule-tac*  $x=E2$  **in** *allE*)

**apply** (*frule* *consistent-identityClosure*)

by *simp*

**lemma** *P5-P6-f-n-APP-2*:

$\llbracket (E1, E2) \vdash h, k, AppE\ g\ as\ rs'() \Downarrow(f,n)\ hh, k, v;$   
 $hh = h' \mid \{p \in dom\ h'.\ fst\ (the\ (h'\ p)) \leq k\};$   
 $dom\ \Gamma \subseteq dom\ E1; \forall a \in set\ as.\ atom\ a;$   
 $length\ xs = length\ ms; length\ xs = length\ as;$   
 $wellFormedDepth\ f\ n\ (fvs'\ as)\ \Gamma\ (AppE\ g\ as\ rs'());$   
 $fvs'\ as \subseteq dom\ \Gamma;$   
 $nonDisjointUnionSafeEnvList\ (maps-of\ (zip\ (map\ atom2var\ as)\ ms)) \subseteq_m\ \Gamma \rrbracket$   
 $\implies shareRec\ (fvs'\ as)\ \Gamma\ (E1, E2)\ (h, k)\ (hh, k)$   
**apply** (*simp add: shareRec-def*)  
**apply** (*rule conjI*)

**apply** (*simp only: wellFormedDepth-def*)  
**apply** (*erule-tac x=E1 in allE*)  
**apply** (*erule-tac x=E2 in allE*)  
**apply** (*erule-tac x=h in allE*)  
**apply** (*erule-tac x=k in allE*)  
**apply** (*erule-tac x= h' \mid \{p \in dom\ h'.\ fst\ (the\ (h'\ p)) \leq k\} in allE*)  
**apply** (*erule-tac x=v in allE*)  
**apply** (*drule mp*)  
**apply** (*rule conjI, simp*)  
**apply** (*rule conjI, simp*)  
**apply** (*rule conjI*)  
**apply** *simp*  
**apply** *simp*  
**apply** *simp*

**apply** (*frule SafeDepthSem-extend-heaps*)  
**apply** (*frule Lemma4-consistent*)  
**apply** (*erule-tac x=01 in allE*)  
**apply** (*erule-tac x=02 in allE*)  
**apply** (*erule-tac x=\eta in allE*)  
**apply** (*erule-tac x=E1 in allE*)  
**apply** (*erule-tac x=E2 in allE*)  
**apply** (*frule consistent-identityClosure*)  
**by** *simp*

**lemma** *P5-APP-PRIMOP*:

$\llbracket \Gamma 0 = [atom2var\ a1 \mapsto s'', atom2var\ a2 \mapsto s''];$   
 $\Gamma 0 \subseteq_m\ \Gamma \rrbracket$



$\implies (\forall x \in \text{dom } (\text{fst } (E1, E2))).$   
 $\forall z \in \{\text{atom2var } a1, \text{atom2var } a2\}.$   
 $\Gamma z = \text{Some } d'' \wedge$   
 $\text{closure } (E1, E2) x (h, k) \cap \text{recReach } (E1, E2) z (h, k) \neq \{\}$   $\longrightarrow$   
 $x \in \text{dom } \Gamma \wedge \Gamma x \neq \text{Some } s''$ )  
**apply** (rule ballI)+  
**apply** (rule impI)  
**apply** (elim conjE, clarsimp)  
**apply** (erule disjE, simp-all)  
**apply** (simp add: map-le-def)  
**apply** (split split-if-asm, simp, simp)  
**by** (simp add: map-le-def)

**lemma** P6-APP-PRIMOP:  
 $\llbracket \Gamma 0 = [\text{atom2var } a1 \mapsto s'', \text{atom2var } a2 \mapsto s''];$   
 $\Gamma 0 \subseteq_m \Gamma \rrbracket$   
 $\implies \forall x \in \text{dom } (\text{fst } (E1, E2)).$   
 $\neg \text{identityClosure } (E1, E2) x (h, k) (h, k) \longrightarrow$   
 $x \in \text{dom } \Gamma \wedge \Gamma x \neq \text{Some } s''$   
**apply** (rule ballI, rule impI)  
**by** (simp add: identityClosure-def)

**lemma** P5-P6-APP-PRIMOP:  
 $\llbracket \Gamma 0 = [\text{atom2var } a1 \mapsto s'', \text{atom2var } a2 \mapsto s''];$   
 $\Gamma 0 \subseteq_m \Gamma \rrbracket$   
 $\implies \text{shareRec } \{\text{atom2var } a1, \text{atom2var } a2\}$   
 $\Gamma$   
 $(E1, E2) (h, k) (h, k)$   
**apply** (simp only: shareRec-def)  
**apply** (rule conjI)  
**apply** (rule P5-APP-PRIMOP, assumption+, simp)  
**by** (rule P6-APP-PRIMOP, assumption+, simp)

**end**

## 17 Derived Assertions. P4. $\text{fv } e \subseteq L$

**theory** SafeDAss-P4 **imports** SafeDAssBasic  
**begin**

Lemmas for LET

**lemma** fvs-as-subseteq-L1:  
 $\forall a \in \text{set } as. \text{atom } a$   
 $\implies \text{fvs } as \subseteq \text{atom2var } ' \text{ set } as$

**apply** (*induct as,simp-all*)  
**apply** (*rule conjI*)  
**apply** (*case-tac a,simp-all*)  
**by** *blast*

**lemma** *P4-LET*:  
 $\llbracket \text{fv } e1 \subseteq L1; \text{fv } e2 \subseteq L2 \rrbracket$   
 $\implies \text{fv } (\text{Let } x1 = e1 \text{ In } e2 \text{ } a) \subseteq L1 \cup (L2 - \{x1\})$   
**by** (*clarsimp,blast*)

Lemmas for CASE

**lemma** *P4-CASE-aux* [*rule-format*]:  
 $x \in \text{fvAlts } \text{alts}$   
 $\longrightarrow \text{length } \text{alts} > 0$   
 $\longrightarrow (\exists i < \text{length } \text{alts}. x \in \text{fv } (\text{snd } (\text{alts } ! i)) \wedge$   
 $\quad x \notin \text{set } (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! i))))))$   
**apply** (*induct alts,simp*)  
**apply** (*rule impI,simp*)  
**apply** (*erule disjE*)  
**apply** (*case-tac a, simp-all*)  
**apply** (*elim conjE*)  
**apply** (*rule-tac x=0 in exI,simp*)  
**apply** (*case-tac alts,simp-all*)  
**apply** (*erule exE*)  
**apply** (*case-tac i,simp-all*)  
**apply** (*rule-tac x=Suc 0 in exI,simp*)  
**apply** (*rule-tac x=Suc (Suc nat) in exI*)  
**by** *simp*

**lemma** *P4-CASE*:  
 $\llbracket \forall i < \text{length } \text{alts}. x \in \text{fst } (\text{assert } ! i) \wedge$   
 $\quad x \notin \text{set } (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! i))));$   
 $\quad \forall i < \text{length } \text{alts}. \text{fv } (\text{snd } (\text{alts } ! i)) \subseteq \text{fst } (\text{assert } ! i);$   
 $\quad \text{length } \text{alts} > 0 \rrbracket$   
 $\implies \text{fv } (\text{Case VarE } x \text{ } a \text{ } \text{Of } \text{alts } a') \subseteq$   
 $\quad (\bigcup i < \text{length } \text{alts}. \text{fst } (\text{assert } ! i) - \text{set } (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! i)))) \cup$   
 $\quad \{x\}$   
**apply** *auto*  
**apply** (*subgoal-tac length alts > 0*)  
**prefer** 2 **apply** *simp*  
**apply** (*frule P4-CASE-aux,assumption+*)  
**apply** (*erule exE*)  
**apply** (*erule-tac x=i in allE,simp*)  
**apply** (*elim conjE*)  
**apply** (*erule-tac x=i in ballE,simp*)  
**apply** *blast*  
**by** *simp*

Lemmas for CASED

**lemma** *P4-CASED-aux* [rule-format]:  
 $x \in \text{fvAlts}' \text{ alts} \longrightarrow$   
 $\text{length alts} > 0 \longrightarrow$   
 $(\exists i < \text{length alts}. x \in \text{fv} (\text{snd} (\text{alts} ! i)) \wedge$   
 $\quad x \notin \text{set} (\text{snd} (\text{extractP} (\text{fst} (\text{alts} ! i))))))$   
**apply** (*induct alts,simp*)  
**apply** (*rule impI,simp*)  
**apply** (*erule disjE*)  
**apply** (*case-tac a*)  
**apply** (*simp,elim conjE*)  
**apply** (*rule-tac x=0 in exI,simp*)  
**apply** *simp*  
**apply** (*case-tac alts,simp*)  
**apply** (*simp, erule exE*)  
**apply** (*case-tac i,simp*)  
**apply** (*rule-tac x=Suc 0 in exI,simp*)  
**by** (*rule-tac x=Suc (Suc nat) in exI,simp*)

**lemma** *P4-CASED*:  
 $\llbracket \forall i < \text{length alts}. \text{fv} (\text{snd} (\text{alts} ! i)) \subseteq \text{fst} (\text{assert} ! i);$   
 $\quad \text{length alts} > 0 \rrbracket$   
 $\implies \text{fv} (\text{CaseD VarE } x \text{ a Of alts a}) \subseteq$   
 $\quad \text{insert } x (\bigcup_{i < \text{length alts}} \text{fst} (\text{assert} ! i) - \text{set} (\text{snd} (\text{extractP} (\text{fst} (\text{alts} !$   
 $\quad i))))))$   
**apply** *clarsimp*  
**apply** (*subgoal-tac length alts > 0*)  
**prefer** 2 **apply** *simp*  
**apply** (*frule P4-CASED-aux,assumption+*)  
**by** *blast*

**lemma** *P4-APP-PRIMOP*:  
 $\llbracket \text{atom } a1; \text{atom } a2 \rrbracket$   
 $\implies \text{fv} (\text{AppE } f [a1, a2] \text{ rs}' a) \subseteq \{\text{atom2var } a1, \text{atom2var } a2\}$   
**apply** (*simp,rule conjI*)  
**apply** (*case-tac a1,simp-all*)  
**by** (*case-tac a2,simp-all*)

**end**

## 18 Derived Assertions. P7. $S \text{ L}, \Gamma, E, h \cap R \text{ L}, \Gamma, E, h =$

**theory** *SafeDAss-P7* **imports** *SafeDAssBasic*  
*BasicFacts*

**begin**

Lemmas for LET1 Rule

**lemma** *P7-e1-dem1*:

$\llbracket S1 = S1s \cup S1r \cup S1d; S1s \subseteq S; R1 \subseteq R; (S1r \cup S1d) \cap R1 = \{\}; S \cap R = \{\} \rrbracket$   
 $\implies S1 \cap R1 = \{\}$

**apply** *blast*  
**done**

**lemma** *P7-e1-dem2*:

$SSet \ L1 \ \Gamma1 \ E \ h = SSet1 \ L1 \ \Gamma1 \ (pp \ \Gamma1 \ \Gamma2 \ L2) \ s'' \ E \ h \cup$   
 $SSet1 \ L1 \ \Gamma1 \ (pp \ \Gamma1 \ \Gamma2 \ L2) \ d'' \ E \ h \cup$   
 $SSet1 \ L1 \ \Gamma1 \ (pp \ \Gamma1 \ \Gamma2 \ L2) \ r'' \ E \ h$

**apply** (*rule equalityI*)

**apply** (*simp add: SSet-def add: Let-def add: SSet1-def, clarsimp*)

**apply** (*simp add: pp-def, simp add: dom-def add: safe-def*)

**apply** (*rule-tac x=xa in exI, clarsimp*)

**apply** (*erule-tac x=xa in allE, clarsimp*)+

**apply** (*case-tac y, simp-all*)

**apply** (*simp add: SSet-def, simp add: Let-def, simp add: SSet1-def*)

**by** *blast*

**lemma** *P7-e1-dem3* :

$SSet1 \ L1 \ \Gamma1 \ (pp \ \Gamma1 \ \Gamma2 \ L2) \ s'' \ E \ h \subseteq SSet \ (L1 \cup (L2 - \{x1\})) \ (pp \ \Gamma1 \ \Gamma2 \ L2) \ E \ h$

**apply** (*simp add: SSet-def, simp add: Let-def, simp add: SSet1-def*)

**by** *blast*

**lemma** *P7-e1-dem5* :

$\llbracket dom \ \Gamma1 \subseteq dom \ E1 \rrbracket \implies$   
 $((SSet1 \ L1 \ \Gamma1 \ (pp \ \Gamma1 \ \Gamma2 \ L2) \ d'' \ (E1, E2) \ (h, k)) \cup$   
 $(SSet1 \ L1 \ \Gamma1 \ (pp \ \Gamma1 \ \Gamma2 \ L2) \ r'' \ (E1, E2) \ (h, k))) \cap RSet \ L1 \ \Gamma1 \ (E1, E2) \ (h, k)$   
 $\neq \{\}$   
 $\longrightarrow (\exists x \ z. x \in dom \ E1 \wedge z \in L1 \wedge \Gamma1 \ z = Some \ d'' \wedge \Gamma1 \ x = Some \ s'' \wedge$   
 $closure \ (E1, E2) \ x \ (h, k) \cap recReach \ (E1, E2) \ z \ (h, k) \neq \{\})$

**apply** (*simp add: SSet1-def add: RSet-def, auto*)

**apply** (*erule-tac x=xa in allE*)

**apply** (erule impE, assumption+)  
**apply** (subgoal-tac  $\llbracket \text{dom } \Gamma 1 \subseteq \text{dom } E1; \Gamma 1 \text{ } xa = \text{Some } s'' \rrbracket \implies xa \in \text{dom } E1, \text{clarsimp}$ )  
**apply** (erule-tac  $x=z$  in allE)  
**apply** (erule impE, assumption+)  
**apply** (frule closure-transit, assumption, blast)  
**apply** blast  
**apply** (erule-tac  $x=xa$  in allE)  
**apply** (erule impE, assumption+)  
**apply** (subgoal-tac  $\llbracket \text{dom } \Gamma 1 \subseteq \text{dom } E1; \Gamma 1 \text{ } xa = \text{Some } s'' \rrbracket \implies xa \in \text{dom } E1, \text{clarsimp}$ )  
**apply** (erule-tac  $x=z$  in allE)  
**apply** (erule impE, assumption+, simp)  
**apply** (frule closure-transit, assumption, blast)  
**by** blast

**lemma** P7-e1-dem6 :

$\llbracket \text{shareRec } L1 \ \Gamma 1 \ (E1, E2) \ (h, k) \ (h', k') ;$   
 $\text{dom } \Gamma 1 \subseteq \text{dom } E1 ;$   
 $((\text{SSet1 } L1 \ \Gamma 1 \ (pp \ \Gamma 1 \ \Gamma 2 \ L2) \ d'' \ (E1, E2) \ (h, k)) \cup$   
 $(\text{SSet1 } L1 \ \Gamma 1 \ (pp \ \Gamma 1 \ \Gamma 2 \ L2) \ r'' \ (E1, E2) \ (h, k))) \cap \text{RSet } L1 \ \Gamma 1 \ (E1, E2) \ (h, k)$   
 $\neq \{\}$   
 $\longrightarrow (\exists x \ z. \ x \in \text{dom } E1 \wedge z \in L1 \wedge \Gamma 1 \ z = \text{Some } d'' \wedge \Gamma 1 \ x = \text{Some } s'' \wedge$   
 $\text{closure } (E1, E2) \ x \ (h, k) \cap \text{recReach } (E1, E2) \ z \ (h, k) \neq \{\}) \rrbracket$   
 $\implies (\text{SSet1 } L1 \ \Gamma 1 \ (pp \ \Gamma 1 \ \Gamma 2 \ L2) \ d'' \ (E1, E2) \ (h, k) \cup$   
 $\text{SSet1 } L1 \ \Gamma 1 \ (pp \ \Gamma 1 \ \Gamma 2 \ L2) \ r'' \ (E1, E2) \ (h, k)) \cap \text{RSet } L1 \ \Gamma 1 \ (E1, E2) \ (h, k)$   
 $= \{\}$   
**apply** (simp add: shareRec-def add: SSet1-def add: RSet-def)  
**by** blast

**lemma** P7-e1-dem4-1:

$\llbracket \Gamma 1 \ x = \text{Some } d'' \rrbracket \implies (pp \ \Gamma 1 \ \Gamma 2 \ L2) \ x = \text{Some } d''$   
**by** (simp add: pp-def add: safe-def add: dom-def)

**lemma** P7-e1-dem4-2:

$\llbracket xa \in \text{live } E \ L1 \ h \rrbracket \implies xa \in \text{live } E \ (L1 \cup (L2 - \{x1\})) \ h$   
**by** (simp add: live-def add: closureLS-def)

**lemma** P7-e1-dem4 :

$\text{RSet } (L1 \cup (L2 - \{x1\})) \ (pp \ \Gamma 1 \ \Gamma 2 \ L2) \ (E1, E2) \ (h, k) \cap$   
 $\text{RSet } (L1 \cup (L2 - \{x1\})) \ (pp \ \Gamma 1 \ \Gamma 2 \ L2) \ (E1, E2) \ (h, k) = \{\} \implies$   
 $\text{RSet } L1 \ \Gamma 1 \ (E1, E2) \ (h, k) \subseteq \text{RSet } (L1 \cup (L2 - \{x1\})) \ (pp \ \Gamma 1 \ \Gamma 2 \ L2) \ (E1,$   
 $E2) \ (h, k)$   
**apply** (simp add: RSet-def, safe)  
**apply** (erule P7-e1-dem4-2)  
**apply** (rule-tac  $x=z$  in bexI)  
**apply** (rule conjI)

**apply** (erule P7-e1-dem4-1)  
**apply** blast  
**by** blast

**lemma** P7-LET-e1:

$\llbracket \text{shareRec } L1 \ \Gamma 1 \ (E1, E2) \ (h, k) \ (h', k') ;$   
 $\text{dom } \Gamma 1 \subseteq \text{dom } E1 ;$   
 $\text{SSet } (L1 \cup (L2 - \{x1\})) \ (pp \ \Gamma 1 \ \Gamma 2 \ L2) \ (E1, E2) \ (h, k) \cap$   
 $\text{RSet } (L1 \cup (L2 - \{x1\})) \ (pp \ \Gamma 1 \ \Gamma 2 \ L2) \ (E1, E2) \ (h, k) = \{\}$   
 $\implies \text{SSet } L1 \ \Gamma 1 \ (E1, E2) \ (h, k) \cap \text{RSet } L1 \ \Gamma 1 \ (E1, E2) \ (h, k) = \{\}$   
**apply** (rule P7-e1-dem1)  
**apply** (rule P7-e1-dem2)  
**apply** (rule P7-e1-dem3)  
**apply** (rule P7-e1-dem4, assumption+)  
**apply** (erule P7-e1-dem6, assumption)  
**by** (erule P7-e1-dem5, assumption)

**lemma** P7-e2-dem1 :

$\llbracket (x1 \notin L2 \longrightarrow S2 \subseteq S) ;$   
 $(x1 \in L2 \longrightarrow S2 = S2' \cup S2'x1 \wedge S2' \subseteq S \wedge S2'x1 \cap R2 = \{\}) ;$   
 $R2 \subseteq R ;$   
 $S \cap R = \{\}$   
 $\implies S2 \cap R2 = \{\}$   
**by** blast

**lemma** demS2-2-x1-not-L2:

$\llbracket \text{dom } \Gamma 1 \subseteq \text{dom } E1 ;$   
 $\text{def-disjointUnionEnv } \Gamma 2 \ (\text{empty}(x1 \mapsto s'')) ;$   
 $\text{dom } (\text{disjointUnionEnv } \Gamma 2 \ (\text{empty}(x1 \mapsto s''))) \subseteq \text{dom } (E1(x1 \mapsto v1)) ;$   
 $\text{def-pp } \Gamma 1 \ \Gamma 2 \ L2 ;$   
 $x1 \notin L1 ;$   
 $\text{shareRec } L1 \ \Gamma 1 \ (E1, E2) \ (h, k) \ (h', k') \rrbracket$   
 $\implies x1 \notin L2 \longrightarrow \text{SSet } L2 \ (\text{disjointUnionEnv } \Gamma 2 \ (\text{empty}(x1 \mapsto s''))) \ (E1(x1 \mapsto$   
 $r), E2) \ (h', k') \subseteq$   
 $\text{SSet } (L1 \cup (L2 - \{x1\})) \ (pp \ \Gamma 1 \ \Gamma 2 \ L2) \ (E1, E2) \ (h, k)$   
**apply** (rule impI)  
**apply** (simp add: SSet-def, clarsimp)  
**apply** (simp add: Let-def)  
**apply** (erule exE, rename-tac y)  
**apply** (rule-tac x=y in exI, elim conjE)  
**apply** (case-tac y  $\neq$  x1, clarsimp)  
**apply** (subgoal-tac  $(\Gamma 2 + [x1 \mapsto s'']) \ y = \text{Some } s'' \implies \Gamma 2 \ y = \text{Some } s'', \text{clarsimp}$ )

```

apply (simp add: shareRec-def)
apply (elim conjE)
apply (erule-tac x=y in ballE)+
prefer 2 apply blast
prefer 2 apply blast
apply (frule safe-Gamma2-triangle,assumption+)
apply (case-tac  $\neg$  identityClosure (E1, E2) y (h, k) (h', k'),simp)
apply simp
apply (simp add: identityClosure-def) apply (elim conjE)
apply (rule conjI)
apply (erule safe-triangle,assumption+)
apply (subgoal-tac  $y \neq x1 \implies \text{closure } (E1, E2) y (h', k') = \text{closure } (E1(x1 \mapsto r), E2) y (h', k')$ ,simp)
apply (simp add: closure-def)
apply (simp add: disjointUnionEnv-def add: unionEnv-def)
apply (split split-if-asm, simp)
apply simp
apply simp
done

```

**lemma** P7-e2-dem2-1:

```

  [ def-disjointUnionEnv  $\Gamma 2$  (empty( $x1 \mapsto s''$ ));
     $x1 \in L2$  ]
 $\implies \text{SSet } L2 (\text{disjointUnionEnv } \Gamma 2 (\text{empty}(x1 \mapsto s'')) (E1(x1 \mapsto r), E2) (h', k')) =$ 
 $\text{SSet } (L2 - \{x1\}) \Gamma 2 (E1(x1 \mapsto r), E2) (h', k') \cup$ 
 $\text{SSet } \{x1\} (\text{empty}(x1 \mapsto s'')) (E1(x1 \mapsto r), E2) (h', k')$ 
apply (rule equalityI)
apply (rule subsetI)
apply (simp add: SSet-def add: Let-def)
apply (erule exE)
apply (case-tac xa=x1,simp)
apply (rule disjI1, erule conjE)
apply (subgoal-tac ( $\Gamma 2 + [x1 \mapsto s'']$ ) xa = Some s''  $\implies \Gamma 2 xa = \text{Some } s''$ ,simp)
apply (rule-tac x=xa in exI,simp)
apply (simp add: disjointUnionEnv-def add: unionEnv-def)
apply (split split-if-asm, simp)
apply simp
apply (rule subsetI)
apply (erule UnE)
apply (simp add: SSet-def add: Let-def)
apply (erule exE)
apply (subgoal-tac  $\Gamma 2 xa = \text{Some } s'' \implies (\Gamma 2 + [x1 \mapsto s'']) xa = \text{Some } s''$ ,simp)
apply (rule-tac x=xa in exI,simp)
apply (simp add: disjointUnionEnv-def add: unionEnv-def add: dom-def)
apply (simp add: SSet-def add: Let-def)
apply (rule-tac x=x1 in exI)
apply (rule conjI,simp)

```

**apply** (rule conjI)  
**apply** (simp add: def-disjointUnionEnv-def add: disjointUnionEnv-def add: unionEnv-def)  
**by** simp

**lemma** demS2-1-1b:

$\llbracket \text{dom } \Gamma 1 \subseteq \text{dom } E1;$   
 $L2 \subseteq \text{dom } (\text{disjointUnionEnv } \Gamma 2 \text{ (empty}(x1 \mapsto s'')));$   
 $\text{def-disjointUnionEnv } \Gamma 2 \text{ (empty}(x1 \mapsto s''));$   
 $\text{dom } (\text{disjointUnionEnv } \Gamma 2 \text{ (empty}(x1 \mapsto s'')))) \subseteq \text{dom } (E1(x1 \mapsto v1));$   
 $\text{def-pp } \Gamma 1 \Gamma 2 L2;$   
 $x1 \notin L1; x1 \in L2;$   
 $\text{shareRec } L1 \Gamma 1 (E1, E2) (h, k) (h', k') \rrbracket$   
 $\implies \text{SSet } (L2 - \{x1\}) \Gamma 2 (E1(x1 \mapsto r), E2) (h', k') \subseteq$   
 $\text{SSet } (L1 \cup (L2 - \{x1\})) (\text{pp } \Gamma 1 \Gamma 2 L2) (E1, E2) (h, k)$   
**apply** (simp add: SSet-def, clarsimp)  
**apply** (simp add: Let-def)  
**apply** (erule-tac exE, rename-tac y)  
**apply** (rule-tac x=y in exI)  
**apply** (case-tac  $y \neq x1$ , simp)  
**apply** (elim conjE)  
**apply** (simp add: shareRec-def)  
**apply** (elim conjE)  
**apply** (erule-tac x=y in ballE)+  
**prefer** 2 **apply** blast  
**prefer** 2 **apply** blast  
**apply** (frule safe-Gamma2-triangle, assumption+)  
**apply** (case-tac  $\neg \text{identityClosure } (E1, E2) y (h, k) (h', k'), \text{simp}$ )  
**apply** simp  
**apply** (simp add: identityClosure-def) **apply** (elim conjE)  
**apply** (rule conjI)  
**apply** (erule safe-triangle, assumption+)  
**apply** (subgoal-tac  $y \neq x1 \implies \text{closure } (E1, E2) y (h', k') = \text{closure } (E1(x1 \mapsto r), E2) y (h', k'), \text{simp}$ )  
**apply** (simp add: closure-def)  
**by** blast

**lemma** demS2-S2x1-subset-R2-aux:

$\llbracket \text{closureL } x (h', k') \cap \text{recReach } (E1(x1 \mapsto r), E2) z (h', k') \neq \{\};$   
 $x \in \text{closure } (E1(x1 \mapsto r), E2) x1 (h', k') \rrbracket \implies$   
 $\text{closure } (E1(x1 \mapsto r), E2) x1 (h', k') \cap \text{recReach } (E1(x1 \mapsto r), E2) z (h', k')$   
 $\neq \{\}$   
**apply** (simp add: closure-def)  
**apply** (case-tac r, auto)  
**apply** (frule closureL-transit, assumption+)  
**by** blast

**lemma** demS2-S2x1-subset-R2:



```

[[dom  $\Gamma 1 \subseteq \text{dom } E1$ ;
  def-disjointUnionEnv  $\Gamma 2$  ( $\text{empty}(x1 \mapsto s'')$ );
  dom ( $\text{disjointUnionEnv } \Gamma 2$  ( $\text{empty}(x1 \mapsto s'')$ ))  $\subseteq \text{dom } (E1(x1 \mapsto v1))$ ;
  def-pp  $\Gamma 1 \Gamma 2 L2$ ;
   $x1 \notin L1$ ;  $x1 \in L2$ ;
  shareRec  $L2$  ( $\text{disjointUnionEnv } \Gamma 2$  ( $\text{empty}(x1 \mapsto s'')$ )) ( $E1(x1 \mapsto r)$ ,  $E2$ ) ( $h'$ ,
 $k'$ ) ( $hh, kk$ ) ]]
 $\implies \text{SSet } \{x1\}$  ( $\text{empty}(x1 \mapsto s'')$ ) ( $E1(x1 \mapsto r), E2$ ) ( $h'$ ,  $k'$ )  $\cap$ 
 $\text{RSet } L2$  ( $\text{disjointUnionEnv } \Gamma 2$  ( $\text{empty}(x1 \mapsto s'')$ )) ( $E1(x1 \mapsto r), E2$ ) ( $h'$ ,
 $k'$ ) = {}
apply (simp add: SSet-def, auto)
apply (simp add: RSet-def)
apply (erule conjE, erule bexE, erule conjE)
apply (unfold shareRec-def)
apply (elim conjE)
apply (erule-tac x=x1 in ballE) prefer 2 apply simp
apply (erule-tac x=x1 in ballE) prefer 2 apply simp
apply (erule-tac x=z in ballE)
apply (drule-tac P=( $\Gamma 2 + [x1 \mapsto s'']$ )  $z = \text{Some } d'' \wedge$ 
closure ( $E1(x1 \mapsto r)$ ,  $E2$ )  $x1$  ( $h'$ ,  $k'$ )  $\cap \text{recReach } (E1(x1 \mapsto r)$ ,
 $E2$ )  $z$  ( $h'$ ,  $k'$ )  $\neq \{\}$  in mp))
apply (rule conjI, simp)
apply (frule demS2-S2x1-subset-R2-aux, assumption+)
apply (erule conjE)
apply (simp add: def-disjointUnionEnv-def add: disjointUnionEnv-def add: unionEnv-def)
apply simp
done

```

**lemma** *demS2-2-x1-in-L2:*

```

[[dom  $\Gamma 1 \subseteq \text{dom } E1$ ;
   $x1 \notin L1$ ;
   $L2 \subseteq \text{dom } (\text{disjointUnionEnv } \Gamma 2$  ( $\text{empty}(x1 \mapsto s'')$ ));
  def-disjointUnionEnv  $\Gamma 2$  ( $\text{empty}(x1 \mapsto s'')$ );
  dom ( $\text{disjointUnionEnv } \Gamma 2$  ( $\text{empty}(x1 \mapsto s'')$ ))  $\subseteq \text{dom } (E1(x1 \mapsto v1))$ ;
  def-pp  $\Gamma 1 \Gamma 2 L2$ ;
  shareRec  $L1 \Gamma 1$  ( $E1$ ,  $E2$ ) ( $h$ ,  $k$ ) ( $h', k'$ );
  shareRec  $L2$  ( $\text{disjointUnionEnv } \Gamma 2$  ( $\text{empty}(x1 \mapsto s'')$ )) ( $E1(x1 \mapsto r)$ ,  $E2$ ) ( $h'$ ,
 $k'$ ) ( $hh, kk$ );
   $\text{SSet } (L1 \cup (L2 - \{x1\}))$  ( $\text{pp } \Gamma 1 \Gamma 2 L2$ ) ( $E1$ ,  $E2$ ) ( $h$ ,  $k$ )  $\cap$ 
 $\text{RSet } (L1 \cup (L2 - \{x1\}))$  ( $\text{pp } \Gamma 1 \Gamma 2 L2$ ) ( $E1$ ,  $E2$ ) ( $h$ ,  $k$ ) = {}]]
 $\implies x1 \in L2 \implies \text{SSet } L2$  ( $\text{disjointUnionEnv } \Gamma 2$  ( $\text{empty}(x1 \mapsto s'')$ )) ( $E1(x1$ 
 $\mapsto r)$ ,  $E2$ ) ( $h'$ ,  $k'$ ) =  $?S2' \cup ?S2'x1.0 \wedge$ 
 $?S2' \subseteq \text{SSet } (L1 \cup (L2 - \{x1\}))$  ( $\text{pp } \Gamma 1 \Gamma 2 L2$ ) ( $E1$ ,  $E2$ ) ( $h$ ,  $k$ )  $\wedge$ 
 $?S2'x1.0 \cap \text{RSet } L2$  ( $\text{disjointUnionEnv } \Gamma 2$  ( $\text{empty}(x1 \mapsto s'')$ )) ( $E1(x1 \mapsto r)$ ,
 $E2$ ) ( $h'$ ,  $k'$ ) = {}
apply (rule impI, rule conjI)
apply (rule P7-e2-dem2-1, assumption+)
apply (rule conjI)

```

**apply** (rule demS2-1-1b,assumption+)  
**by** (rule demS2-S2x1-subset-R2,assumption+)

**lemma** demR2-subseteq-R :  

$$\begin{aligned}
 & \llbracket \text{def-pp } \Gamma 1 \ \Gamma 2 \ L2; \ L1 \subseteq \text{dom } \Gamma 1; \\
 & \quad L2 \subseteq \text{dom } (\text{disjointUnionEnv } \Gamma 2 \ (\text{empty}(x1 \mapsto s''))); \\
 & \quad \text{dom } (\text{disjointUnionEnv } \Gamma 2 \ (\text{empty}(x1 \mapsto s''))) \subseteq \text{dom } (E1(x1 \mapsto v1)); \\
 & \quad \text{def-disjointUnionEnv } \Gamma 2 \ (\text{empty}(x1 \mapsto s'')); \\
 & \quad \text{shareRec } L2 \ (\text{disjointUnionEnv } \Gamma 2 \ (\text{empty}(x1 \mapsto s''))) \ (E1(x1 \mapsto r), E2) \ (h', \\
 & \quad k') \ (hh, kk); \\
 & \quad \text{shareRec } L1 \ \Gamma 1 \ (E1, E2) \ (h, k) \ (h', k') \rrbracket \\
 & \implies \text{RSet } L2 \ (\text{disjointUnionEnv } \Gamma 2 \ (\text{empty}(x1 \mapsto s''))) \ (E1(x1 \mapsto r), E2) \ (h', \\
 & \quad k') \subseteq \\
 & \quad \text{RSet } (L1 \cup (L2 - \{x1\})) \ (\text{pp } \Gamma 1 \ \Gamma 2 \ L2) \ (E1, E2) \ (h, k)
 \end{aligned}$$
**apply** (rule subsetI, rename-tac p)  
**apply** (simp add: RSet-def)  
**apply** (erule conjE, erule bexE, rename-tac x)  
**apply** (case-tac x=x1)  
**apply** simp  
**apply** (elim conjE)  
**apply** (simp add: disjointUnionEnv-def add: unionEnv-def add: def-disjointUnionEnv-def)

**apply** (erule conjE)  
**apply** (subgoal-tac p  $\in$  live (E1(x1  $\mapsto$  r), E2) L2 (h', k')  
 $\implies \exists y \in L2. p \in \text{closure } (E1(x1 \mapsto r), E2) \ y \ (h', k'), \text{simp}$ )  
**prefer** 2 **apply** (simp add: live-def add: closureLS-def)  
**apply** (erule bexE)  
**apply** (unfold shareRec-def)  
**apply** (elim conjE)  
**apply** (erule-tac x=y and A=dom (fst (E1(x1  $\mapsto$  r), E2)) in ballE)+  
**prefer** 2 **apply** simp **apply** (elim conjE) **apply** blast  
**apply** (erule-tac x=x and A=L2 in ballE) **prefer** 2 **apply** simp  
**prefer** 2 **apply** simp **apply** (elim conjE) **apply** blast  
**apply** (drule-tac P=( $\Gamma 2 + [x1 \mapsto s'']$ ) x = Some d''  $\wedge$  closure (E1(x1  $\mapsto$  r), E2)  
y (h', k')  $\cap$  recReach (E1(x1  $\mapsto$  r), E2) x (h', k')  $\neq$  {}  
in mp)  
**apply** (rule conjI)  
**apply** simp  
**apply** (rule closure-recReach-monotone, assumption+)  
**apply** simp  
**apply** (elim conjE)  
**apply** (rule conjI)  
**apply** (case-tac y = x1)  
**apply** (simp add: def-disjointUnionEnv-def add: disjointUnionEnv-def add:  
unionEnv-def)  
**apply** (subgoal-tac  $\llbracket (\Gamma 2 + [x1 \mapsto s'']) \ y \neq \text{Some } s''; \ y \neq x1 \rrbracket \implies \Gamma 2 \ y \neq \text{Some}$   
s'',simp)  
**prefer** 2 **apply** (rule unsafe-Gamma2-triangle, assumption+)  
**apply** (frule-tac y=y in unsafe-Gamma2-identityClosure) **apply** assumption+

```

apply (simp add: identityClosure-def) apply (elim conjE)
apply (subgoal-tac  $\llbracket y \neq x1; y \in L2 \rrbracket \implies \text{closure } (E1, E2) \ y \ (h, k) \subseteq \text{live } (E1, E2) \ (L1 \cup (L2 - \{x1\})) \ (h, k), \text{simp}$ )
prefer 2 apply (rule closure-subset-live, assumption+)
apply (subgoal-tac  $y \neq x1 \implies \text{closure } (E1, E2) \ y \ (h', k') = \text{closure } (E1(x1 \mapsto r), E2) \ y \ (h', k'), \text{simp}$ )
prefer 2 apply (simp add: closure-def)
apply (rule closure-live-monotone, assumption+)

apply (case-tac y=x1)
apply (simp add: def-disjointUnionEnv-def add: disjointUnionEnv-def add: unionEnv-def)
apply (rule-tac x=x in bexI)
apply (rule conjI)
apply (subgoal-tac  $\llbracket \text{def-pp } \Gamma1 \ \Gamma2 \ L2; (\Gamma2 + [x1 \mapsto s']) \ x = \text{Some } d'' \rrbracket \implies (\text{pp } \Gamma1 \ \Gamma2 \ L2) \ x = \text{Some } d''$ )
prefer 2 apply (rule condemned-Gamma2-triangle, assumption+)
prefer 2 apply simp
apply (subgoal-tac  $\llbracket (\Gamma2 + [x1 \mapsto s']) \ y \neq \text{Some } s''; y \neq x1 \rrbracket \implies \Gamma2 \ y \neq \text{Some } s'', \text{simp}$ )
prefer 2 apply (rule unsafe-Gamma2-triangle, assumption+)
apply (subgoal-tac  $(\Gamma2 + [x1 \mapsto s']) \ x \neq \text{Some } s''$ ) prefer 2 apply simp
apply (frule-tac y=y in unsafe-Gamma2-triangle, assumption+)
apply (frule-tac y=y in unsafe-Gamma2-identityClosure) apply assumption+
apply (subgoal-tac  $\llbracket \text{def-pp } \Gamma1 \ \Gamma2 \ L2; (\Gamma2 + [x1 \mapsto s']) \ x = \text{Some } d'' \rrbracket \implies (\text{pp } \Gamma1 \ \Gamma2 \ L2) \ x = \text{Some } d''$ )
prefer 2 apply (rule condemned-Gamma2-triangle, assumption+)
apply simp
apply (subgoal-tac  $(\text{pp } \Gamma1 \ \Gamma2 \ L2) \ x = \text{Some } d'' \implies \Gamma2 \ x \neq \text{Some } s'', \text{simp}$ )
apply (frule-tac y=x in unsafe-Gamma2-identityClosure) apply assumption+
apply (frule-tac x=x in identityClosure-equals-recReach)
apply (subgoal-tac  $y \neq x1 \implies \text{closure } (E1(x1 \mapsto r), E2) \ y \ (h', k') = \text{closure } (E1, E2) \ y \ (h', k'), \text{simp}$ )
prefer 2 apply (simp add: closure-def)
apply (subgoal-tac  $p \in \text{closure } (E1, E2) \ y \ (h', k') \implies p \in \text{closure } (E1, E2) \ y \ (h, k), \text{simp}$ )
apply (frule-tac x=y in identityClosure-closureL-monotone, simp)
apply (simp add: identityClosure-def add: identityClosureL-def, elim conjE)
apply (subgoal-tac  $x \neq x1 \implies \text{recReach } (E1(x1 \mapsto r), E2) \ x \ (h', k') = \text{recReach } (E1, E2) \ x \ (h', k'), \text{simp}$ )
apply (simp add: recReach-def)
apply (simp add: identityClosure-def)
apply (rule unsafe-triangle-unsafe-2) apply assumption+
done

```

**lemma** *P7-LET1-e2*:

```

 $\llbracket \text{def-pp } \Gamma1 \ \Gamma2 \ L2; L1 \subseteq \text{dom } \Gamma1;$ 
 $\text{dom } \Gamma1 \subseteq \text{dom } E1;$ 

```

$L2 \subseteq \text{dom } (\text{disjointUnionEnv } \Gamma 2 \text{ (empty}(x1 \mapsto s''))$ ;  
 $x1 \notin L1$ ;  
 $\text{def-disjointUnionEnv } \Gamma 2 \text{ (empty}(x1 \mapsto s''))$ ;  
 $\text{dom } (\text{disjointUnionEnv } \Gamma 2 \text{ (empty}(x1 \mapsto s'')) \subseteq \text{dom } (E1(x1 \mapsto v1))$  ;  
 $\text{shareRec } L1 \Gamma 1 (E1, E2) (h, k) (h', k')$ ;  
 $\text{shareRec } L2 (\text{disjointUnionEnv } \Gamma 2 \text{ (empty}(x1 \mapsto s'')) (E1(x1 \mapsto v1), E2)$   
 $(h', k') (hh, kk)$ ;  
 $S\text{Set } (L1 \cup (L2 - \{x1\})) (pp \Gamma 1 \Gamma 2 L2) (E1, E2) (h, k) \cap$   
 $R\text{Set } (L1 \cup (L2 - \{x1\})) (pp \Gamma 1 \Gamma 2 L2) (E1, E2) (h, k) = \{\}$   
 $\implies S\text{Set } L2 (\text{disjointUnionEnv } \Gamma 2 \text{ (empty}(x1 \mapsto s'')) (E1(x1 \mapsto v1), E2) (h',$   
 $k') \cap$   
 $R\text{Set } L2 (\text{disjointUnionEnv } \Gamma 2 \text{ (empty}(x1 \mapsto s'')) (E1(x1 \mapsto v1), E2) (h',$   
 $k') = \{\}$   
**apply** (rule P7-e2-dem1)  
**apply** (rule demS2-2-x1-not-L2,assumption+)  
**apply** (rule demS2-2-x1-in-L2,assumption+)  
**by** (rule demR2-subseteq-R,assumption+)

Lemmas for LET2 Rule

**lemma** P7-e2-let2-dem1 :  
 $\llbracket (x1 \in L2 \longrightarrow R2 = R2'x1 \cup R2d \wedge R2'x1 \cap S2 = \{\} \wedge R2d \cap S2 = \{\})$ ;  
 $(x1 \notin L2 \longrightarrow R2 \subseteq R)$ ;  
 $S2 \subseteq S$ ;  
 $S \cap R = \{\}$   
 $\implies S2 \cap R2 = \{\}$   
**by** blast

**lemma** P7-e2-let2-dem2:  
 $\llbracket \text{def-disjointUnionEnv } \Gamma 2 [x1 \mapsto d'']; x1 \in L2 \rrbracket$   
 $\implies$   
 $R\text{Set } L2 (\text{disjointUnionEnv } \Gamma 2 \text{ (empty}(x1 \mapsto d'')) (E1(x1 \mapsto v1), E2) (h',$   
 $k') =$   
 $\{p \in \text{live } (E1(x1 \mapsto v1), E2) L2 (h', k'). \text{closureL } p (h', k') \cap \text{recReach}$   
 $(E1(x1 \mapsto v1), E2) x1 (h', k') \neq \{\}\} \cup$   
 $\{p \in \text{live } (E1(x1 \mapsto v1), E2) L2 (h', k'). \exists z \in (L2 - \{x1\}). \Gamma 2 z = \text{Some}$   
 $d'' \wedge$   
 $\text{closureL } p (h', k') \cap \text{recReach } (E1(x1 \mapsto$   
 $v1), E2) z (h', k') \neq \{\}\}$   
**apply** (rule equalityI)  
**apply** (rule subsetI)  
**apply** simp  
**apply** (simp only: RSet-def)  
**apply** simp  
**apply** (rule impI)  
**apply** (elim conjE)  
**apply** clarsimp

```

apply (rule-tac  $x=z$  in  $bexI$ )
apply (rule conjI)
apply (case-tac  $z=x1$ )
apply simp
apply (frule disjointUnionEnv-d-Gamma2-d) apply assumption+
apply (case-tac  $z=x1$ )
apply simp
apply simp

apply (rule subsetI)
  apply (simp only: RSet-def)
apply (elim UnE)
apply simp
apply (rule-tac  $x=x1$  in  $bexI$ )
apply simp
apply (rule def-disjointUnionEnv-monotone) apply assumption+
apply simp
apply (erule conjE)
apply (erule bexE)
apply (rule-tac  $x=z$  in  $bexI$ ) prefer 2 apply simp
apply (elim conjE)
apply (rule conjI) apply (rule Gamma2-d-disjointUnionEnv-d, assumption+)
done

```

**lemma** P7-e2-let2-dem4:

```

  [| dom (disjointUnionEnv  $\Gamma 2$  (empty( $x1 \mapsto d''$ )))  $\subseteq$  dom ( $E1(x1 \mapsto v1)$ );
    def-disjointUnionEnv  $\Gamma 2$  [ $x1 \mapsto d''$ ];
    shareRec L2 (disjointUnionEnv  $\Gamma 2$  (empty( $x1 \mapsto d''$ ))) ( $E1(x1 \mapsto v1)$ ,  $E2$ )
    ( $h'$ ,  $k'$ ) ( $hh, kk$ );
     $x1 \in L2$  |]
   $\implies \{p \in \text{live } (E1(x1 \mapsto v1), E2) \text{ L2 } (h', k'). \text{ closureL } p (h', k') \cap \text{recReach}$ 
    ( $E1(x1 \mapsto v1), E2$ )  $x1 (h', k') \neq \{\}$   $\} \cap$ 
    SSet L2 (disjointUnionEnv  $\Gamma 2$  (empty( $x1 \mapsto d''$ ))) ( $E1(x1 \mapsto v1), E2$ ) ( $h'$ ,
     $k'$ ) =  $\{\}$ 
apply auto
apply (simp add: live-def add: closureLS-def) apply (rename-tac  $p$ )
apply (erule bexE, rename-tac  $z$ )
apply (simp add: SSet-def)
apply (simp add: Let-def)
apply (erule exE)
apply (elim conjE)
apply (simp add: shareRec-def)
apply (elim conjE)
apply (erule-tac  $x=xa$  in  $ballE$ )
apply (drule-tac  $Q= xa \in \text{dom } (\Gamma 2 + [x1 \mapsto d'']) \wedge (\Gamma 2 + [x1 \mapsto d'']) \text{ } xa \neq$ 
  Some  $s''$  in  $mp$ )
apply (rule-tac  $x=x1$  in  $bexI$ )

```

**apply** (rule conjI)  
**apply** (rule def-disjointUnionEnv-monotone) **apply** assumption+  
**apply** (subgoal-tac closureL x (h', k')  $\cap$  recReach (E1(x1  $\mapsto$  v1), E2) x1 (h', k')  $\neq$  {})) **prefer** 2 **apply** blast  
**apply** (rule closure-recReach-monotone) **apply** assumption+  
**apply** (elim conjE) **apply** simp  
**apply** (subgoal-tac  $\llbracket \text{dom} (\text{disjointUnionEnv } \Gamma 2 (\text{empty}(x1 \mapsto d'')) \subseteq \text{dom} (E1(x1 \mapsto v1)))$ ;  
 $(\Gamma 2 + [x1 \mapsto d'']) \text{ } xa = \text{Some } s'' \rrbracket \implies xa \in \text{dom } E1$ )  
**apply** simp  
**apply** (rule dom-disjointUnionEnv-subset-dom-extend) **apply** assumption+  
**done**

**lemma** P7-e2-let2-dem10:

$\llbracket \text{dom} (\text{disjointUnionEnv } \Gamma 2 (\text{empty}(x1 \mapsto d'')) \subseteq \text{dom} (E1(x1 \mapsto v1)))$ ;  
 $\text{shareRec } L2 (\text{disjointUnionEnv } \Gamma 2 (\text{empty}(x1 \mapsto d'')) (E1(x1 \mapsto v1), E2)$   
 $(h', k') (hh, kk)$ ;  
 $\text{def-disjointUnionEnv } \Gamma 2 [x1 \mapsto d'']; x1 \in L2 \rrbracket$   
 $\implies \{p \in \text{live} (E1(x1 \mapsto v1), E2) \mid L2 (h', k'). \exists z \in (L2 - \{x1\}). \Gamma 2 z =$   
 $\text{Some } d'' \wedge$   
 $\text{closureL } p (h', k') \cap \text{recReach} (E1(x1 \mapsto$   
 $v1), E2) z (h', k') \neq \{\}\} \cap$   
 $\text{SSet } L2 (\text{disjointUnionEnv } \Gamma 2 (\text{empty}(x1 \mapsto d'')) (E1(x1 \mapsto v1), E2) (h',$   
 $k') = \{\}$   
**apply** auto  
**apply** (rename-tac y)  
**apply** (simp add: SSet-def)  
**apply** (simp add: Let-def)  
**apply** (erule exE)  
**apply** (elim conjE)  
**apply** (simp add: live-def add: closureLS-def)  
**apply** (erule bexE)  
**apply** (simp add: shareRec-def)  
**apply** (elim conjE)  
**apply** (erule-tac x=xa in ballE)  
**apply** (drule-tac  $Q = xa \in \text{dom} (\Gamma 2 + [x1 \mapsto d'']) \wedge (\Gamma 2 + [x1 \mapsto d'']) \text{ } xa \neq$   
 $\text{Some } s''$  in mp)  
**apply** (rule-tac x=z in bexI)  
**apply** (rule conjI)  
**apply** (rule Gamma2-d-disjointUnionEnv-d) **apply** assumption+  
**apply** (rule closure-recReach-monotone) **apply** assumption+  
**apply** blast  
**apply** simp  
**apply** (elim conjE) **apply** simp  
**apply** (subgoal-tac  $\llbracket \text{dom} (\text{disjointUnionEnv } \Gamma 2 (\text{empty}(x1 \mapsto d'')) \subseteq \text{dom} (E1(x1 \mapsto v1)))$ ;  
 $(\Gamma 2 + [x1 \mapsto d'']) \text{ } xa = \text{Some } s'' \rrbracket \implies xa \in \text{dom } E1$ )  
**apply** simp

**apply** (*rule disjointUnionEnv-subset-dom-extend*) **apply** *assumption+*  
**done**

**lemma** *P7-e2-let2-dem7*:

$\llbracket \text{def-disjointUnionEnv } \Gamma 2 \text{ (empty}(x1 \mapsto d'')) ;$   
 $\text{dom (disjointUnionEnv } \Gamma 2 \text{ (empty}(x1 \mapsto d'')) \subseteq \text{dom (E1}(x1 \mapsto v1)) ;$   
 $\text{shareRec L1 } \Gamma 1 \text{ (E1, E2) (h, k) (h', k') ;}$   
 $\text{def-pp } \Gamma 1 \Gamma 2 \text{ L2} \rrbracket \implies$   
 $\text{SSet L2 (disjointUnionEnv } \Gamma 2 \text{ (empty}(x1 \mapsto d'')) \text{ (E1}(x1 \mapsto v1), \text{E2) (h', k')}$   
 $\subseteq$   
 $\text{SSet (L1 } \cup \text{ (L2 - \{x1\}) (pp } \Gamma 1 \Gamma 2 \text{ L2) (E1, E2) (h, k)}$   
**apply** (*simp add: SSet-def, clarsimp*)  
**apply** (*simp add: Let-def*)  
**apply** (*erule exE, rename-tac y*)  
**apply** (*rule-tac x=y in exI, elim conjE*)  
**apply** (*case-tac y  $\neq$  x1, clarsimp*)  
**apply** (*subgoal-tac ( $\Gamma 2 + [x1 \mapsto d'']$ ) y = Some s''  $\implies$   $\Gamma 2$  y = Some s'', clarsimp*)  
**apply** (*simp add: shareRec-def*)  
**apply** (*elim conjE*)  
**apply** (*erule-tac x=y in ballE*) +  
**prefer** 2 **apply** *blast*  
**prefer** 2 **apply** *blast*  
**apply** (*frule safe-Gamma2-triangle, assumption+*)  
**apply** (*case-tac  $\neg$  identityClosure (E1, E2) y (h, k) (h', k'), simp*)  
**apply** *simp*  
**apply** (*simp add: identityClosure-def*) **apply** (*elim conjE*)  
**apply** (*rule conjI*)  
**apply** (*erule safe-triangle, assumption+*)  
**apply** (*subgoal-tac y  $\neq$  x1  $\implies$  closure (E1, E2) y (h', k') = closure (E1(x1  $\mapsto$*   
*r), E2) y (h', k'), simp*)  
**apply** (*simp add: closure-def*)  
**apply** (*simp add: disjointUnionEnv-def add: unionEnv-def*)  
**apply** (*split split-if-asm, simp*)  
**apply** (*simp add: closure-def*)  
**apply** (*simp add: closure-def*)  
**apply** (*simp add: disjointUnionEnv-def add: unionEnv-def*)  
**apply** (*split split-if-asm, simp, simp, simp*)  
**by** (*simp add: disjointUnionEnv-def add: unionEnv-def add: def-disjointUnionEnv-def*)

**lemma** *P7-e2-let2-dem8*:

$\llbracket \text{def-pp } \Gamma 1 \Gamma 2 \text{ L2; } L1 \subseteq \text{dom } \Gamma 1 ;$   
 $L2 \subseteq \text{dom (disjointUnionEnv } \Gamma 2 \text{ (empty}(x1 \mapsto d'')) ;$   
 $\text{dom (disjointUnionEnv } \Gamma 2 \text{ (empty}(x1 \mapsto d'')) \subseteq \text{dom (E1}(x1 \mapsto v1)) ;$   
 $\text{def-disjointUnionEnv } \Gamma 2 \text{ (empty}(x1 \mapsto d'')) ;$

```

    shareRec L2 (disjointUnionEnv  $\Gamma 2$  (empty( $x1 \mapsto d''$ ))) ( $E1(x1 \mapsto r)$ ,  $E2$ ) ( $h'$ ,
     $k'$ ) ( $hh, kk$ );
    shareRec L1  $\Gamma 1$  ( $E1$ ,  $E2$ ) ( $h$ ,  $k$ ) ( $h', k'$ );  $x1 \notin L2$ ]
     $\implies$  RSet L2 (disjointUnionEnv  $\Gamma 2$  (empty( $x1 \mapsto d''$ ))) ( $E1(x1 \mapsto r)$ ,  $E2$ ) ( $h'$ ,
     $k'$ )  $\subseteq$ 
    RSet ( $L1 \cup (L2 - \{x1\})$ ) ( $pp \Gamma 1 \Gamma 2 L2$ ) ( $E1, E2$ ) ( $h$ ,  $k$ )
  apply (rule subsetI, rename-tac p)
  apply (simp add: RSet-def)
  apply (erule conjE, erule bexE, rename-tac x)
  apply (subgoal-tac  $p \in \text{live } (E1(x1 \mapsto r), E2) L2 (h', k')$ 
     $\implies \exists y \in L2. p \in \text{closure } (E1(x1 \mapsto r), E2) y (h', k'), \text{simp}$ )
  prefer 2 apply (simp add: live-def add: closureLS-def)
  apply (erule bexE)
  apply (unfold shareRec-def)
  apply (elim conjE)
  apply (erule-tac  $x=y$  and  $A = \text{dom } (\text{fst } (E1(x1 \mapsto r), E2))$  in ballE)+
  prefer 2 apply simp apply (elim conjE) apply blast
  apply (erule-tac  $x=x$  and  $A=L2$  in ballE) prefer 2 apply simp
  prefer 2 apply simp apply (elim conjE) apply blast
  apply (drule-tac  $P=(\Gamma 2 + [x1 \mapsto d'']) x = \text{Some } d'' \wedge \text{closure } (E1(x1 \mapsto r), E2)$ 
     $y (h', k') \cap \text{recReach } (E1(x1 \mapsto r), E2) x (h', k') \neq \{\}$ 
    in mp)
  apply (rule conjI)
  apply simp
  apply (rule closure-recReach-monotone, assumption+)
  apply simp
  apply (elim conjE)
  apply (rule conjI)
  apply (case-tac  $y \neq x1$ )
  apply (subgoal-tac  $\llbracket (\Gamma 2 + [x1 \mapsto d'']) y \neq \text{Some } s'' \rrbracket \implies \Gamma 2 y \neq \text{Some } s''$ )
  prefer 2 apply (rule unsafe-Gamma2-triangle, assumption+)
  apply (frule-tac  $y=y$  in unsafe-Gamma2-identityClosure) apply assumption+
  apply (simp add: identityClosure-def) apply (elim conjE)
  apply (subgoal-tac  $\llbracket y \neq x1; y \in L2 \rrbracket \implies \text{closure } (E1, E2) y (h, k) \subseteq \text{live } (E1,$ 
     $E2) (L1 \cup (L2 - \{x1\})) (h, k), \text{simp}$ )
  prefer 2 apply (rule closure-subset-live, assumption+)
  apply (subgoal-tac  $y \neq x1 \implies \text{closure } (E1, E2) y (h', k') = \text{closure } (E1(x1 \mapsto$ 
     $r), E2) y (h', k'), \text{simp}$ )
  prefer 2 apply (simp add: closure-def)
  apply blast
  apply simp

  apply (case-tac  $y=x1$ )
  apply (simp add: def-disjointUnionEnv-def add: disjointUnionEnv-def add: unionEnv-def)
  apply (rule-tac  $x=x$  in bexI) prefer 2 apply simp
  apply (rule conjI)
  apply (case-tac  $x=x1$ ) apply simp
  apply (subgoal-tac  $\llbracket x \neq x1; \text{def-pp } \Gamma 1 \Gamma 2 L2; (\Gamma 2 + [x1 \mapsto d'']) x = \text{Some } d'' \rrbracket$ 

```



```

 $\Rightarrow (pp \ \Gamma 1 \ \Gamma 2 \ L2) \ x = Some \ d''$ 
  prefer 2 apply (rule disjointUnionEnv-d-triangle-d, assumption+)
  apply (subgoal-tac  $\llbracket (\Gamma 2 + [x1 \mapsto d'']) \ y \neq Some \ s''; y \neq x1 \rrbracket \Rightarrow \Gamma 2 \ y \neq Some \ s'', simp$ )
  prefer 2 apply (rule unsafe-Gamma2-triangle, assumption+)
  apply (subgoal-tac  $(\Gamma 2 + [x1 \mapsto d'']) \ x \neq Some \ s''$ ) prefer 2 apply simp
  apply (frule-tac  $y=y$  in unsafe-Gamma2-triangle, assumption+)
  apply (frule-tac  $y=y$  in unsafe-Gamma2-identityClosure) apply assumption+
    apply (subgoal-tac  $\llbracket x \neq x1; def\text{-}pp \ \Gamma 1 \ \Gamma 2 \ L2; (\Gamma 2 + [x1 \mapsto d'']) \ x = Some \ d'' \rrbracket \Rightarrow (pp \ \Gamma 1 \ \Gamma 2 \ L2) \ x = Some \ d''$ )
 $\Rightarrow (pp \ \Gamma 1 \ \Gamma 2 \ L2) \ x = Some \ d''$ 
  prefer 2 apply (rule disjointUnionEnv-d-triangle-d, assumption+)
  apply (case-tac  $x=x1$ ) apply simp
  apply simp
  apply (subgoal-tac  $(pp \ \Gamma 1 \ \Gamma 2 \ L2) \ x = Some \ d'' \Rightarrow \Gamma 2 \ x \neq Some \ s''$ )
  apply (frule-tac  $y=x$  in unsafe-Gamma2-identityClosure) apply assumption+
  apply (frule-tac  $x=x$  in identityClosure-equals-recReach)
  apply (subgoal-tac  $y \neq x1 \Rightarrow closure \ (E1(x1 \mapsto r), E2) \ y \ (h', k') = closure \ (E1, E2) \ y \ (h', k'), simp$ )
  prefer 2 apply (simp add: closure-def)
  apply (subgoal-tac  $p \in closure \ (E1, E2) \ y \ (h', k') \Rightarrow p \in closure \ (E1, E2) \ y \ (h, k), simp$ )
  apply (frule-tac  $x=y$  in identityClosure-closureL-monotone, simp)
  apply (simp add: identityClosure-def add: identityClosureL-def, elim conjE)
  apply (subgoal-tac  $x \neq x1 \Rightarrow recReach \ (E1(x1 \mapsto r), E2) \ x \ (h', k') = recReach \ (E1, E2) \ x \ (h', k'), simp$ )
  apply (simp add: recReach-def)
  apply (simp add: identityClosure-def)
  apply (rule unsafe-triangle-unsafe-2) apply assumption+
done

```

**lemma** P7-LET2-e2:

```

 $\llbracket def\text{-}pp \ \Gamma 1 \ \Gamma 2 \ L2; L1 \subseteq dom \ \Gamma 1;$ 
   $dom \ \Gamma 1 \subseteq dom \ E1;$ 
   $L2 \subseteq dom \ (disjointUnionEnv \ \Gamma 2 \ (empty(x1 \mapsto d'')));$ 
   $x1 \notin L1;$ 
   $def\text{-}disjointUnionEnv \ \Gamma 2 \ (empty(x1 \mapsto d''));$ 
   $dom \ (disjointUnionEnv \ \Gamma 2 \ (empty(x1 \mapsto d''))) \subseteq dom \ (E1(x1 \mapsto v1));$ 
   $shareRec \ L1 \ \Gamma 1 \ (E1, E2) \ (h, k) \ (h', k');$ 
   $shareRec \ L2 \ (disjointUnionEnv \ \Gamma 2 \ (empty(x1 \mapsto d''))) \ (E1(x1 \mapsto v1), E2)$ 
 $(h', k') \ (hh, kk);$ 
   $SSet \ (L1 \cup (L2 - \{x1\})) \ (pp \ \Gamma 1 \ \Gamma 2 \ L2) \ (E1, E2) \ (h, k) \cap$ 
   $RSet \ (L1 \cup (L2 - \{x1\})) \ (pp \ \Gamma 1 \ \Gamma 2 \ L2) \ (E1, E2) \ (h, k) = \{\}$ 
 $\Rightarrow SSet \ L2 \ (disjointUnionEnv \ \Gamma 2 \ (empty(x1 \mapsto d''))) \ (E1(x1 \mapsto v1), E2) \ (h',$ 
 $k') \cap$ 
   $RSet \ L2 \ (disjointUnionEnv \ \Gamma 2 \ (empty(x1 \mapsto d''))) \ (E1(x1 \mapsto v1), E2) \ (h',$ 
 $k') = \{\}$ 
  apply (rule P7-e2-let2-dem1)

```

```

apply (rule impI) apply (rule conjI)
apply (rule P7-e2-let2-dem2, assumption+)
apply (rule conjI) apply (rule P7-e2-let2-dem4) apply assumption+
apply (rule P7-e2-let2-dem10) apply assumption+
apply (rule impI)
apply (rule P7-e2-let2-dem8) apply assumption+
apply (rule P7-e2-let2-dem7) apply assumption+
done

```

Lemmas for CASE Rule

```

lemma dom-foldl-monotone-list:
   $\text{dom } (\text{foldl } op \otimes (\text{empty} \otimes x) xs) =$ 
   $\text{dom } x \cup \text{dom } (\text{foldl } op \otimes \text{empty } xs)$ 
apply (subgoal-tac empty  $\otimes x = x \otimes \text{empty}$ ,simp)
apply (subgoal-tac foldl op  $\otimes (x \otimes \text{empty}) xs =$ 
   $x \otimes \text{foldl } op \otimes \text{empty } xs, \text{simp}$ )
apply (rule union-dom-nonDisjointUnionEnv)
apply (rule foldl-prop1)
apply (subgoal-tac def-nonDisjointUnionEnv empty x)
apply (erule nonDisjointUnionEnv-commutative)
by (simp add: def-nonDisjointUnionEnv-def)

```

```

lemma dom-restrict-neg-map:
   $\text{dom } (\text{restrict-neg-map } m \ A) = \text{dom } m - (\text{dom } m \cap A)$ 
apply (simp add: restrict-neg-map-def)
apply auto
by (split split-if-asm,simp-all)

```

```

lemma x-notin- $\Gamma$ -cased:
   $x \notin \text{dom } (\text{foldl } op \otimes \text{empty}$ 
     $(\text{map } (\lambda(Li, \Gamma i). \text{restrict-neg-map } \Gamma i (\text{insert } x (\text{set } Li)))$ 
     $(\text{zip } (\text{map } (\text{snd} \circ \text{extractP} \circ \text{fst}) \text{alts}) (\text{map } \text{snd } \text{assert}))))$ 
apply (induct-tac assert alts rule: list-induct2',simp-all)
apply (subgoal-tac
   $\text{dom } (\text{foldl } op \otimes (\text{empty} \otimes \text{restrict-neg-map } (\text{snd } xa) (\text{insert } x (\text{set } (\text{snd } (\text{extractP}$ 
     $(\text{fst } y))))))$ 
     $(\text{map } (\lambda(Li, \Gamma i). \text{restrict-neg-map } \Gamma i (\text{insert } x (\text{set } Li))) (\text{zip } (\text{map}$ 
     $(\lambda a. \text{snd } (\text{extractP } (\text{fst } a))) \text{ys}) (\text{map } \text{snd } xs)))) =$ 
     $\text{dom } (\text{restrict-neg-map } (\text{snd } xa) (\text{insert } x (\text{set } (\text{snd } (\text{extractP } (\text{fst } y)))))) \cup$ 
     $\text{dom } (\text{foldl } op \otimes \text{empty } (\text{map } (\lambda(Li, \Gamma i). \text{restrict-neg-map } \Gamma i (\text{insert } x (\text{set } Li)))$ 
       $(\text{zip } (\text{map } (\lambda a. \text{snd } (\text{extractP } (\text{fst } a))) \text{ys}) (\text{map } \text{snd}$ 
         $xs))))), \text{simp}$ )
apply (subst dom-restrict-neg-map,blast)
by (rule dom-foldl-monotone-list)

```

```

lemma  $\Gamma$ -case-x-is-d:
   $\llbracket \Gamma = \text{foldl } \text{op} \otimes \text{empty}$ 
     $(\text{map } (\lambda(Li, \Gamma i). \text{restrict-neg-map } \Gamma i (\text{insert } x (\text{set } Li))) (\text{zip } (\text{map}$ 
       $(\text{snd} \circ \text{extractP} \circ \text{fst}) \text{alts}) (\text{map } \text{snd } \text{assert}))) +$ 
     $\llbracket x \mapsto d'' \rrbracket$ 
   $\implies \Gamma x = \text{Some } d''$ 
apply (simp add: disjointUnionEnv-def)
apply (simp add: unionEnv-def)
apply (rule impI)
apply (insert x-notin- $\Gamma$ -cased)
by force

```

```

lemma restrict-neg-map-m:
   $\llbracket G y = \text{Some } m; x \neq y ; y \notin L \rrbracket$ 
   $\implies \text{restrict-neg-map } G (\text{insert } x L) y = \text{Some } m$ 
by (simp add: restrict-neg-map-def)

```

```

lemma disjointUnionEnv-G-G'-G-x:
   $\llbracket x \notin \text{dom } G'; \text{def-disjointUnionEnv } G G' \rrbracket$ 
   $\implies (G + G') x = G x$ 
apply (simp add: disjointUnionEnv-def)
apply (simp add: unionEnv-def)
apply (simp add: def-disjointUnionEnv-def)
by force

```

```

lemma dom- $\Gamma$ i-in- $\Gamma$ cased-2 [rule-format]:
  length assert > 0
   $\longrightarrow x \neq y$ 
   $\longrightarrow \text{length } \text{assert} = \text{length } \text{alts}$ 
   $\longrightarrow (\forall i < \text{length } \text{alts}. y \in \text{dom } (\text{snd } (\text{assert } ! i)))$ 
   $\longrightarrow y \notin \text{set } (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! i))))$ 
   $\longrightarrow y \in \text{dom } (\text{foldl } \text{op} \otimes \text{empty}$ 
     $(\text{map } (\lambda(Li, \Gamma i). \text{restrict-neg-map } \Gamma i (\text{insert } x (\text{set } Li)))$ 
     $(\text{zip } (\text{map } (\text{snd} \circ \text{extractP} \circ \text{fst}) \text{alts}) (\text{map } \text{snd } \text{assert}))))$ 
apply (induct assert alts rule:list-induct2',simp-all)
apply (rule impI)+
apply (case-tac xs = [],simp)
apply (rule impI)+
apply (subst empty-nonDisjointUnionEnv)
apply (subst dom-restrict-neg-map)
apply force
apply simp
apply (rule allI, rule impI)
apply (case-tac i,simp-all)
apply (rule impI)+
apply (subst dom-foldl-monotone-list)
apply (subst dom-restrict-neg-map)
apply force

```

```

apply (rule impI)
apply (erule-tac x=nat in allE,simp)
apply (rule impI)+
apply (subst dom-foldl-monotone-list)
by blast

declare def-nonDisjointUnionEnvList.simps [simp del]

lemma Otimes-prop4 [rule-format]:
  length assert > 0
   $\longrightarrow y \neq x$ 
   $\longrightarrow \text{length } \text{assert} = \text{length } \text{alts}$ 
   $\longrightarrow \text{def-nonDisjointUnionEnvList } (\text{map } (\lambda(Li, \Gamma i). \text{restrict-neg-map } \Gamma i (\text{insert } x (\text{set } Li))))$ 
   $(\text{zip } (\text{map } (\text{snd} \circ \text{extractP} \circ \text{fst}) \text{alts}) (\text{map } \text{snd } \text{assert})))$ 
   $\longrightarrow \text{def-disjointUnionEnv}$ 
   $(\text{foldl } \text{op} \otimes \text{empty}$ 
     $(\text{map } (\lambda(Li, \Gamma i). \text{restrict-neg-map } \Gamma i (\text{insert } x (\text{set } Li)))$ 
     $(\text{zip } (\text{map } (\text{snd} \circ \text{extractP} \circ \text{fst}) \text{alts}) (\text{map } \text{snd } \text{assert}))))$ 
     $[x \mapsto d']$ 
   $\longrightarrow (\forall i < \text{length } \text{alts}. y \in \text{dom } (\text{snd } (\text{assert } ! i)))$ 
   $\longrightarrow \text{snd } (\text{assert } ! i) y = \text{Some } m$ 
   $\longrightarrow y \notin \text{set } (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! i))))$ 
   $\longrightarrow (\text{foldl } \text{op} \otimes \text{empty}$ 
     $(\text{map } (\lambda(Li, \Gamma i). \text{restrict-neg-map } \Gamma i (\text{insert } x (\text{set } Li)))$ 
     $(\text{zip } (\text{map } (\text{snd} \circ \text{extractP} \circ \text{fst}) \text{alts}) (\text{map } \text{snd } \text{assert}))) +$ 
     $[x \mapsto d'] y = \text{Some } m)$ 

apply (induct assert alts rule:list-induct2',simp-all)
apply (rule impI)+
apply (case-tac xs = [],simp)
apply (rule impI)
apply (subst empty-nonDisjointUnionEnv)
apply (simp add: disjointUnionEnv-def)
apply (simp add: unionEnv-def)
apply (rule conjI)
apply (rule impI)+
apply (simp add: restrict-neg-map-def)
apply (rule impI)+
apply (simp add: restrict-neg-map-def)
apply force
apply simp
apply (drule mp)
apply (simp add: def-nonDisjointUnionEnvList.simps)
apply (simp add: Let-def)
apply (drule mp)
apply (simp add: def-disjointUnionEnv-def)
apply (subst (asm) dom-foldl-monotone-list)
apply blast

```

```

apply (rule allI, rule impI)
apply (subgoal-tac  $x \neq y$ )
  prefer 2 apply simp
apply (erule thin-rl)
apply (case-tac i, simp-all)
  apply (rule impI)+
  apply (subst disjointUnionEnv-G-G'-G-x, force, force)
  apply (subst nonDisjointUnionEnv-commutative)
  apply (simp add: def-nonDisjointUnionEnv-def)
  apply (subst foldl-prop1)
  apply (subst nonDisjointUnionEnv-prop6-1)
  apply (subst dom-restrict-neg-map, force)
  apply (rule restrict-neg-map-m, assumption+, simp)
apply (rule impI)+
apply (rotate-tac 5)
apply (erule-tac  $x = \text{nat}$  in allE, simp)
apply (subst disjointUnionEnv-G-G'-G-x, force, simp)
apply (subst nonDisjointUnionEnv-commutative)
  apply (simp add: def-nonDisjointUnionEnv-def)
apply (subst foldl-prop1)
apply (subst (asm) disjointUnionEnv-G-G'-G-x, force)
  apply (simp add: def-disjointUnionEnv-def)
  apply (subst (asm) dom-foldl-monotone-list)
apply blast
apply (subst nonDisjointUnionEnv-prop6-2)
  apply (simp add: def-nonDisjointUnionEnvList.simps)
  apply (simp add: Let-def)
apply (subgoal-tac
   $y \in \text{dom } (\text{foldl } \text{op} \otimes \text{empty}$ 
     $(\text{map } (\lambda(Li, \Gamma i). \text{restrict-neg-map } \Gamma i (\text{insert } x (\text{set } Li)))$ 
     $(\text{zip } (\text{map } (\text{snd} \circ \text{extractP} \circ \text{fst}) \text{ys}) (\text{map } \text{snd } \text{xs}))))$ , simp)
  apply (rule dom- $\Gamma i$ -in- $\Gamma$  cased-2)
by (force, assumption+, simp)

```

```

lemma closureL-p-None-p:
  closureL p (h(p := None), k) = {p}
apply (rule equalityI)
apply (rule subsetI)
apply (erule closureL.induct, simp)
apply (simp add: descendants-def)
apply (rule subsetI, simp)
by (rule closureL-basic)

```

**lemma** *recReachL-p-None-p*:  
 $\text{recReachL } p \ (h(p := \text{None}), k) = \{p\}$   
**apply** (rule *equalityI*)  
**apply** (rule *subsetI*)  
**apply** (erule *recReachL.induct,simp*)  
**apply** (simp add: *recDescendants-def*)  
**apply** (rule *subsetI,simp*)  
**by** (rule *recReachL-basic*)

**lemma** *closure-extend-p-None-subseteq-closure*:  
 $\llbracket E1 \ x = \text{Some } (\text{Loc } p);$   
 $E1 \ x = (\text{extend } E1 \ (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! \ i)))) \ \text{vs}) \ x \rrbracket$   
 $\implies \text{closure } (\text{extend } E1 \ (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! \ i)))) \ \text{vs}, E2) \ x \ (h(p := \text{None}),$   
 $k) \subseteq$   
 $\text{closure } (E1, E2) \ x \ (h, k)$   
**apply** (simp add: *closure-def*)  
**apply** (subst *closureL-p-None-p,simp*)  
**by** (rule *closureL-basic*)

**lemma** *recReach-extend-p-None-subseteq-recReach*:  
 $\llbracket E1 \ x = \text{Some } (\text{Loc } p);$   
 $E1 \ x = (\text{extend } E1 \ (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! \ i)))) \ \text{vs}) \ x \rrbracket$   
 $\implies \text{recReach } (\text{extend } E1 \ (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! \ i)))) \ \text{vs}, E2) \ x \ (h(p :=$   
 $\text{None}), k) \subseteq$   
 $\text{recReach } (E1, E2) \ x \ (h, k)$   
**apply** (simp add: *recReach-def*)  
**apply** (subst *recReachL-p-None-p,simp*)  
**by** (rule *recReachL-basic*)

**lemma** *descendants-p-None-q*:  
 $\llbracket d \in \text{descendants } q \ (h(p := \text{None}), k); \ q \neq p \rrbracket$   
 $\implies d \in \text{descendants } q \ (h, k)$   
**by** (simp add: *descendants-def*)

**lemma** *recDescendants-p-None-q*:  
 $\llbracket d \in \text{recDescendants } q \ (h(p := \text{None}), k); \ q \neq p \rrbracket$   
 $\implies d \in \text{recDescendants } q \ (h, k)$   
**by** (simp add: *recDescendants-def*)

**lemma** *closureL-p-None-subseteq-closureL*:  
 $p \neq q$   
 $\implies \text{closureL } q \ (h(p := \text{None}), k) \subseteq \text{closureL } q \ (h, k)$   
**apply** (rule *subsetI*)  
**apply** (erule *closureL.induct*)  
**apply** (rule *closureL-basic*)  
**apply** *clarsimp*  
**apply** (subgoal-tac  $d \in \text{descendants } q \ (h, k)$ )

```

apply (rule closureL-step,simp,simp)
apply (rule descendants-p-None-q,assumption+)
apply (simp add: descendants-def)
by (case-tac qa = p,simp-all)

```

```

lemma recReachL-p-None-subseteq-recReachL:
   $p \neq q$ 
   $\implies \text{recReachL } q \ (h(p := \text{None}), k) \subseteq \text{recReachL } q \ (h, k)$ 
apply (rule subsetI)
apply (erule recReachL.induct)
apply (rule recReachL-basic)
apply clarsimp
apply (subgoal-tac  $d \in \text{recDescendants } qa \ (h,k)$ )
apply (rule recReachL-step,simp,simp)
apply (rule recDescendants-p-None-q,assumption+)
apply (simp add: recDescendants-def)
by (case-tac qa = p,simp-all)

```

```

lemma closure-p-None-subseteq-closure:
   $\llbracket E1 \ x = \text{Some } (\text{Loc } p);$ 
   $E1 \ y = \text{Some } (\text{Loc } q);$ 
   $p \neq q;$ 
   $E1 \ y = (\text{extend } E1 \ (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! \ i)))) \ \text{vs } y;$ 
   $w \in \text{closure } (\text{extend } E1 \ (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! \ i)))) \ \text{vs}, E2) \ y \ (h(p :=$ 
   $\text{None}), k) \rrbracket$ 
   $\implies w \in \text{closure } (E1, E2) \ y \ (h, k)$ 
apply (simp add: closure-def)
apply (case-tac
   $\text{extend } E1 \ (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! \ i)))) \ \text{vs } y, \text{simp-all})$ 
apply (case-tac a, simp-all)
apply (frule closureL-p-None-subseteq-closureL)
by blast

```

```

lemma recReach-p-None-subseteq-recReach:
   $\llbracket E1 \ x = \text{Some } (\text{Loc } p); E1 \ y = \text{Some } (\text{Loc } q); p \neq q;$ 
   $E1 \ y = (\text{extend } E1 \ (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! \ i)))) \ \text{vs } y;$ 
   $w \in \text{recReach } (\text{extend } E1 \ (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! \ i)))) \ \text{vs}, E2) \ y \ (h(p :=$ 
   $\text{None}), k) \rrbracket$ 
   $\implies w \in \text{recReach } (E1, E2) \ y \ (h, k)$ 
apply (simp add: recReach-def)
apply (case-tac
   $\text{extend } E1 \ (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! \ i)))) \ \text{vs } y, \text{simp-all})$ 
apply (case-tac a, simp-all)
apply (frule recReachL-p-None-subseteq-recReachL)
by blast

```

**lemma** *closureV-subseteq-closureL*:  
 $h\ p = \text{Some } (j, C, vs)$   
 $\implies (\bigcup i < \text{length } vs. \text{closureV } (vs!i) (h, k)) \subseteq \text{closureL } p (h, k)$   
**apply** (*frule closureV-equals-closureL*)  
**by** *blast*

**lemma** *vs-defined*:  
 $\llbracket \text{set } xs \cap \text{dom } E1 = \{\};$   
 $\text{length } xs = \text{length } vs;$   
 $y \in \text{set } xs;$   
 $\text{extend } E1\ xs\ vs\ y = \text{Some } (\text{Loc } q) \rrbracket$   
 $\implies \exists j < \text{length } vs. vs!j = \text{Loc } q$   
**apply** (*simp add: extend-def*)  
**apply** (*induct xs vs rule: list-induct2', simp-all*)  
**by** (*split split-if-asm, force, force*)

**lemma** *closure-Loc-subseteq-closureV-Loc*:  
 $\llbracket vs!i = \text{Loc } q;$   
 $i < \text{length } vs \rrbracket$   
 $\implies \text{closureL } q (h, k) \subseteq (\bigcup i < \text{length } vs \text{closureV } (vs!i) (h, k))$   
**apply** (*rule subsetI*)  
**apply** *clarsimp*  
**apply** (*rule-tac x=i in bexI*)  
**apply** (*simp add: closureV-def*)  
**by** *simp*

**lemma** *patrones*:  
 $\llbracket E1\ x = \text{Some } (\text{Loc } p); h\ p = \text{Some } (j, C, vs);$   
 $i < \text{length } \text{alts}; \text{length } \text{alts} > 0; \text{length } \text{assert} = \text{length } \text{alts};$   
 $\text{set } (\text{snd } (\text{extractP } (\text{fst } (\text{alts}!i)))) \cap \text{dom } E1 = \{\};$   
 $\text{length } (\text{snd } (\text{extractP } (\text{fst } (\text{alts}!i)))) = \text{length } vs;$   
 $y \in \text{set } (\text{snd } (\text{extractP } (\text{fst } (\text{alts}!i)))) \rrbracket$   
 $\implies \text{closure } (\text{extend } E1\ (\text{snd } (\text{extractP } (\text{fst } (\text{alts}!i))))\ vs, E2)\ y (h, k) \subseteq$   
 $\text{closure } (\text{extend } E1\ (\text{snd } (\text{extractP } (\text{fst } (\text{alts}!i))))\ vs, E2)\ x (h, k)$   
**apply** (*rule subsetI*)  
**apply** (*subst (asm) closure-def*)  
**apply** (*case-tac*)  
 $\text{extend } E1\ (\text{snd } (\text{extractP } (\text{fst } (\text{alts}!i))))\ vs\ y, \text{simp-all}$   
**apply** (*case-tac a, simp-all*)  
**apply** (*subgoal-tac x \notin set (snd (extractP (fst (alts!i))))*)  
**prefer** 2 **apply** *blast*  
**apply** (*frule-tac x=x and E=E1 and vs=vs in extend-monotone-i*)



```

apply (simp, simp, simp)
apply (rename-tac q)
apply (frule-tac y=y in vs-defined, force, assumption+)
apply (simp add: closure-def)
apply (frule-tac k=k in closure V-subseteq-closureL)
apply (elim exE, elim conjE)
apply (frule closure-Loc-subseteq-closure V-Loc, assumption+)
by force

```

**lemma** *patrones-2*:

```

  [| E1 x = Some (Loc p); h p = Some (j, C, vs);
    set (snd (extractP (fst (alts ! i)))) ∩ dom E1 = {};
    length (snd (extractP (fst (alts ! i)))) = length vs;
    i < length alts; length alts > 0; length assert = length alts;
    y ∈ set (snd (extractP (fst (alts ! i)))) |]
   $\impl$  closure (extend E1 (snd (extractP (fst (alts ! i)))) vs, E2) y (h(p:=None),
k) ⊆ {p}
apply (rule subsetI)
apply (subst (asm) closure-def)
apply (case-tac
  extend E1 (snd (extractP (fst (alts ! i)))) vs y, simp-all)
apply (case-tac a, simp-all)
apply (rename-tac q)
apply (frule-tac y=y in vs-defined, force, assumption+)
apply (elim exE, elim conjE)
apply (frule-tac h=h(p:=None) and k=k in closure-Loc-subseteq-closure V-Loc, assumption+)
apply (frule-tac h=h and k=k in closure V-subseteq-closureL-None)
apply (subst (asm) closureL-p-None-p)
by blast

```

**lemma** *dom-extend-in-E1-or-xs*:

```

  [| y ∈ dom (extend E1 xs vs); length xs = length vs |]
   $\impl$  y ∈ dom E1 ∨ y ∈ set xs
apply (simp add: extend-def)
apply (erule disjE)
apply (induct xs vs rule: list-induct2')
apply simp-all
by force

```

**lemma** *extend-monotone-x-in-dom-E1-2*:

```

  [| set (snd (extractP (fst (alts ! i)))) ∩ dom E1 = {};
    length (snd (extractP (fst (alts ! i)))) = length vs;
    length assert = length alts; i < length alts;
    x ∈ fst (assert ! i);
    x ∈ dom E1 |]

```

```

     $\Rightarrow E1\ x = \text{extend } E1\ (\text{snd } (\text{extractP } (\text{fst } (\text{alts } !\ i))))\ vs\ x$ 
apply (subgoal-tac  $x \notin \text{set } (\text{snd } (\text{extractP } (\text{fst } (\text{alts } !\ i))))$ )
apply (rule extend-monotone,assumption)
by blast

```

**lemma** *closure-extend-None-subset-closure*:

```

   $\llbracket \text{alts} \neq []; i < \text{length } \text{alts}; \text{length } \text{assert} = \text{length } \text{alts};$ 
   $E1\ x = \text{Some } (\text{Loc } p); h\ p = \text{Some } (j, C, vs);$ 
   $\text{set } (\text{snd } (\text{extractP } (\text{fst } (\text{alts } !\ i)))) \cap \text{dom } E1 = \{\};$ 
   $\text{length } (\text{snd } (\text{extractP } (\text{fst } (\text{alts } !\ i)))) = \text{length } vs;$ 
   $y \in \text{fst } (\text{assert } !\ i);$ 
   $y \notin \text{set } (\text{snd } (\text{extractP } (\text{fst } (\text{alts } !\ i))));$ 
   $y \in \text{dom } E1;$ 
   $q \in \text{closure } (\text{extend } E1\ (\text{snd } (\text{extractP } (\text{fst } (\text{alts } !\ i))))\ vs, E2)\ y\ (h(p :=$ 
   $\text{None}), k) \rrbracket$ 
   $\Rightarrow q \in \text{closure } (E1, E2)\ y\ (h, k)$ 
apply (case-tac  $E1\ y = E1\ x, \text{simp}$ )

apply (frule extend-monotone-x-in-dom-E1-2,assumption+)
apply (subgoal-tac
   $\text{closure } (\text{extend } E1\ (\text{snd } (\text{extractP } (\text{fst } (\text{alts } !\ i))))\ vs, E2)\ y\ (h(p := \text{None}), k)$ 
 $\subseteq$ 
   $\text{closure } (E1, E2)\ y\ (h, k)$ )
prefer 2 apply (rule closure-extend-p-None-subseteq-closure,simp,simp)
apply blast
apply simp
apply (subgoal-tac  $\exists z. E1\ y = \text{Some } z$ )
prefer 2 apply (simp add: dom-def)
apply (elim exE)
apply (frule extend-monotone-x-in-dom-E1-2,assumption+)
apply (case-tac  $z, \text{simp-all}$ )

apply (frule-tac extend-monotone-x-in-dom-E1-2,assumption+)
apply (rule-tac  $i=i$  and  $\text{alts}=\text{alts}$  and  $vs=vs$ 
  in  $\text{closure-p-None-subseteq-closure,assumption+,simp,simp}$ )
apply simp

apply (simp add: closure-def)

apply (simp add: closure-def)
done

```

**lemma** *recReach-extend-None-subset-recReach*:

```

   $\llbracket \text{alts} \neq []; i < \text{length } \text{alts}; \text{length } \text{assert} = \text{length } \text{alts};$ 
   $E1\ x = \text{Some } (\text{Loc } p); h\ p = \text{Some } (j, C, vs);$ 
   $\text{set } (\text{snd } (\text{extractP } (\text{fst } (\text{alts } !\ i)))) \cap \text{dom } E1 = \{\};$ 
   $\text{length } (\text{snd } (\text{extractP } (\text{fst } (\text{alts } !\ i)))) = \text{length } vs;$ 

```

```

    y ∈ fst (assert ! i);
    y ∉ set (snd (extractP (fst (alts ! i))));
    y ∈ dom E1;
    q ∈ recReach (extend E1 (snd (extractP (fst (alts ! i)))) vs, E2) y (h(p :=
None), k)]
    ⇒ q ∈ recReach (E1, E2) y (h, k)
apply (case-tac E1 y = E1 x,simp)

apply (frule extend-monotone-x-in-dom-E1-2,assumption+)
apply (subgoal-tac
  recReach (extend E1 (snd (extractP (fst (alts ! i)))) vs, E2) y (h(p := None),
k) ⊆
  recReach (E1, E2) y (h, k))
prefer 2 apply (rule recReach-extend-p-None-subseteq-recReach,simp,simp)
apply blast
apply simp
apply (subgoal-tac ∃ z. E1 y = Some z)
prefer 2 apply (simp add: dom-def)
apply (elim exE)
apply (frule extend-monotone-x-in-dom-E1-2,assumption+)
apply (case-tac z,simp-all)

apply (frule-tac extend-monotone-x-in-dom-E1-2,assumption+)
apply (rule-tac i=i and alts=alts and vs=vs
  in recReach-p-None-subseteq-recReach,assumption+,simp,simp)
apply simp

apply (simp add: recReach-def)

apply (simp add: recReach-def)
done

lemma closure-monotone-extend-3:
  [ set (snd (extractP (fst (alts ! i)))) ∩ dom E = {};
    length (snd (extractP (fst (alts ! i)))) = length vs;
    x ∈ dom E;
    length alts > 0;
    i < length alts ]
  ⇒ closure (E, E') x (h, k) = closure (extend E (snd (extractP (fst (alts ! i))))
vs, E') x (h, k)
apply (subgoal-tac x ∉ set (snd (extractP (fst (alts ! i)))))
apply (subgoal-tac
  E x = extend E (snd (extractP (fst (alts ! i)))) vs x)
apply (simp add:closure-def)
apply (rule extend-monotone-i)
apply (simp,simp,simp)
by blast

```

**lemma** *recReach-monotone-extend-3*:  

$$\llbracket \text{set} (\text{snd} (\text{extractP} (\text{fst} (\text{alts} ! i)))) \cap \text{dom } E = \{\};$$

$$\text{length} (\text{snd} (\text{extractP} (\text{fst} (\text{alts} ! i)))) = \text{length } vs;$$

$$x \in \text{dom } E;$$

$$\text{length } \text{alts} > 0;$$

$$i < \text{length } \text{alts} \rrbracket$$

$$\implies \text{recReach} (E, E') x (h, k) = \text{recReach} (\text{extend } E (\text{snd} (\text{extractP} (\text{fst} (\text{alts} ! i)))) vs, E') x (h, k)$$
**apply** (*subgoal-tac*  $x \notin \text{set} (\text{snd} (\text{extractP} (\text{fst} (\text{alts} ! i))))$ )  
**apply** (*subgoal-tac*  
 $E x = \text{extend } E (\text{snd} (\text{extractP} (\text{fst} (\text{alts} ! i)))) vs x$ )  
**apply** (*simp add:recReach-def*)  
**apply** (*rule extend-monotone-i*)  
**apply** (*simp, simp, simp*)  
**by** *blast*

**lemma** *pp12*:  

$$\llbracket xa \in \text{closure} (E1, E2) xaa (h, k);$$

$$xb \in \text{closureL } xa (h, k);$$

$$xb \in \text{recReach} (E1, E2) z (h, k) \rrbracket$$

$$\implies$$

$$\text{closure} (E1, E2) xaa (h, k) \cap \text{recReach} (E1, E2) z (h, k) \neq \{\}$$
**apply** (*simp add: closure-def*)  
**apply** (*case-tac E1 xaa*)  
**apply** *simp*  
**apply** *simp*  
**apply** (*case-tac a, simp-all, clarsimp*)  
**apply** (*subgoal-tac*  $xa \in \text{closureL } nat (h, k) \implies \text{closureL } xa (h, k) \subseteq \text{closureL } nat (h, k)$ )  
**apply** *blast*  
**apply** (*erule closureL-monotone*)  
**done**

**lemma** *P7-case-dem1*:  

$$\llbracket Si = S'i \cup S''i;$$

$$Ri = R'i \cup R''i;$$

$$S'i \subseteq S;$$

$$R'i \subseteq R;$$

$$\Gamma x = \text{Some } s'' \longrightarrow S''i \subseteq S \wedge R''i = \{\};$$

$$\Gamma x \neq \text{Some } s'' \longrightarrow S''i \cap R'i = \{\} \wedge R''i = \{\};$$

$$S \cap R = \{\} \rrbracket$$

$$\implies Si \cap Ri = \{\}$$
**by** *blast*

**lemma** *P7-case-dem1-1*:

$\llbracket \text{length } \text{assert} = \text{length } \text{alts}; i < \text{length } \text{assert} \rrbracket \implies$   
 $\text{SSet } (\text{fst } (\text{assert } ! i)) (\text{snd } (\text{assert } ! i)) (\text{extend } E1 (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! i)))) vs, E2) (h, k) =$   
 $\text{SSet } ((\text{fst } (\text{assert } ! i)) \cap (\text{insert } x (\bigcup_{i < \text{length } \text{alts}} \text{fst } (\text{assert } ! i) - \text{set } (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! i))))))$   
 $(\text{snd } (\text{assert } ! i)) (\text{extend } E1 (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! i)))) vs, E2) (h,$   
 $k) \cup$   
 $\text{SSet } ((\text{fst } (\text{assert } ! i)) \cap \text{set } (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! i))))$   
 $(\text{snd } (\text{assert } ! i)) (\text{extend } E1 (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! i)))) vs, E2) (h,$   
 $k)$   
**by** (simp add: SSet-def add: Let-def, blast)

**lemma** P7-case-dem1-2:

$\llbracket \text{length } \text{assert} = \text{length } \text{alts}; i < \text{length } \text{assert} \rrbracket \implies$   
 $\text{RSet } (\text{fst } (\text{assert } ! i)) (\text{snd } (\text{assert } ! i)) (\text{extend } E1 (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! i)))) vs, E2) (h, k) =$   
 $\text{RSet2 } ((\text{fst } (\text{assert } ! i)) \cap (\text{insert } x (\bigcup_{i < \text{length } \text{alts}} \text{fst } (\text{assert } ! i) - \text{set } (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! i))))))$   
 $(\text{fst } (\text{assert } ! i)) (\text{snd } (\text{assert } ! i)) (\text{extend } E1 (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! i)))) vs, E2) (h, k) \cup$   
 $\text{RSet2 } ((\text{fst } (\text{assert } ! i)) \cap \text{set } (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! i))))$   
 $(\text{fst } (\text{assert } ! i)) (\text{snd } (\text{assert } ! i)) (\text{extend } E1 (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! i)))) vs, E2) (h, k)$   
**apply** (simp add: RSet-def add: RSet2-def add: live-def add: closureLS-def)  
**by** blast

**lemma** P7-case-dem1-3:

$\llbracket \forall i < \text{length } \text{assert}. \forall x \in \text{set } (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! i)))) . \text{snd } (\text{assert } ! i)$   
 $x \neq \text{Some } d'';$   
 $\forall i < \text{length } \text{assert}. \text{fst } (\text{assert } ! i) \subseteq \text{dom } (\text{snd } (\text{assert } ! i));$   
 $\forall i < \text{length } \text{alts}. x \in \text{fst } (\text{assert } ! i) \wedge x \notin \text{set } (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! i))));$   
 $\text{dom } (\text{snd } (\text{assert } ! i)) \subseteq \text{dom } (\text{extend } E1 (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! i)))) vs);$   
 $\text{set } (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! i)))) \cap \text{dom } E1 = \{\};$   
 $\text{length } (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! i)))) = \text{length } vs;$   
 $\text{def-nonDisjointUnionEnvList } (\text{map } \text{snd } \text{assert});$   
 $\forall i < \text{length } \text{alts}. \forall j < \text{length } \text{alts}. i \neq j \longrightarrow (\text{fst } (\text{assert } ! i) \cap \text{set } (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! j))))) = \{\};$   
 $\text{length } \text{assert} > 0;$   
 $x \in \text{dom } (\text{nonDisjointUnionEnvList } (\text{map } \text{snd } \text{assert}));$   
 $E1 x = \text{Some } (\text{Loc } p);$   
 $\text{length } \text{assert} = \text{length } \text{alts};$   
 $i < \text{length } \text{assert} \rrbracket \implies$   
 $\text{SSet } ((\text{fst } (\text{assert } ! i)) \cap (\text{insert } x (\bigcup_{i < \text{length } \text{alts}} \text{fst } (\text{assert } ! i) - \text{set } (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! i))))))$   
 $(\text{snd } (\text{assert } ! i)) (\text{extend } E1 (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! i)))) vs, E2) (h,$

```

k)  $\subseteq$ 
  SSet (insert x ( $\bigcup_{i < \text{length } \text{alts}} \text{fst } (\text{assert } ! i) - \text{set } (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! i))))$ ))
    (foldl op  $\otimes$  empty (map snd assert)) (E1, E2) (h, k)
apply (clarsimp)
apply (simp add: SSet-def add: Let-def)
apply (erule exE, elim conjE)
apply (erule disjE)

apply simp
apply (rule-tac x=x in exI)
apply (rule conjI)
apply (rule disjI1) apply simp
apply (rule conjI)
apply (subgoal-tac
  i < length (map snd assert)  $\longrightarrow$ 
  length (map snd assert) > 0  $\longrightarrow$ 
  def-nonDisjointUnionEnvList (map snd assert)  $\longrightarrow$ 
  snd (assert ! i) x = Some s''  $\longrightarrow$ 
  foldl op  $\otimes$  empty (map snd assert) x = Some s'',simp)
apply (rule Otimes-prop3)
apply (subgoal-tac x  $\in$  dom E1)
apply (frule-tac E'=E2 and h=h and k=k in closure-monotone-extend-3,assumption+,simp,assumption+,si
apply (simp add: dom-def)

apply (rule-tac x=xaa in exI)
apply (erule bexE, simp, elim conjE)
apply (case-tac xaa  $\in$  set (snd (extractP (fst (alts ! i))))
apply (case-tac i=ia, simp)
apply (rotate-tac 7)
apply (erule-tac x=ia in allE,simp)
apply (rotate-tac 20)
apply (erule-tac x=i in allE,simp)
apply blast
apply (rule conjI)
apply (rule disjI2)
apply (rule-tac x=i in bexI,simp,simp)
apply (subgoal-tac
  i < length (map snd assert)  $\longrightarrow$ 
  length (map snd assert) > 0  $\longrightarrow$ 
  def-nonDisjointUnionEnvList (map snd assert)  $\longrightarrow$ 
  snd (assert ! i) xaa = Some s''  $\longrightarrow$ 
  foldl op  $\otimes$  empty (map snd assert) xaa = Some s'',simp)
prefer 2 apply (rule Otimes-prop3)
apply (subgoal-tac xaa  $\in$  dom E1)
apply (frule-tac E'=E2 and h=h and k=k in closure-monotone-extend-3,assumption+,simp,assumption+,si
apply (erule-tac x=i in allE,simp)+
apply (subgoal-tac xaa  $\in$  dom (extend E1 (snd (extractP (fst (alts ! i)))) vs))
apply (rule extend-prop1,simp,simp,simp)

```

by *blast*

**lemma** *P7-case-dem1-4*:

```

  [|  $\forall i < \text{length } \text{assert}. \forall x \in \text{set } (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! i)))) . \text{snd } (\text{assert } ! i)$ 
 $x \neq \text{Some } d'';$ 
 $\forall i < \text{length } \text{alts}. x \in \text{fst } (\text{assert } ! i) \wedge x \notin \text{set } (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! i))));$ 
 $\forall i < \text{length } \text{assert}. \text{fst } (\text{assert } ! i) \subseteq \text{dom } (\text{snd } (\text{assert } ! i));$ 
 $\text{dom } (\text{snd } (\text{assert } ! i)) \subseteq \text{dom } (\text{extend } E1 (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! i)))) \text{ vs};$ 
 $\text{set } (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! i)))) \cap \text{dom } E1 = \{\};$ 
 $\text{length } (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! i)))) = \text{length } \text{vs};$ 
 $\forall i < \text{length } \text{alts}. \forall j < \text{length } \text{alts}. i \neq j \longrightarrow (\text{fst } (\text{assert } ! i) \cap \text{set } (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! j)))) = \{\};$ 
 $\text{def-nonDisjointUnionEnvList } (\text{map } \text{snd } \text{assert});$ 
 $\text{length } \text{assert} > 0;$ 
 $x \in \text{dom } (\text{nonDisjointUnionEnvList } (\text{map } \text{snd } \text{assert}));$ 
 $E1 \ x = \text{Some } (\text{Loc } p);$ 
 $h \ p = \text{Some } (j, C, \text{vs});$ 
 $\text{length } \text{assert} = \text{length } \text{alts};$ 
 $i < \text{length } \text{assert} \ ]$ 
 $\implies \text{RSet2 } ((\text{fst } (\text{assert } ! i)) \cap (\text{insert } x (\bigcup_{i < \text{length } \text{alts}} \text{fst } (\text{assert } ! i) - \text{set } (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! i))))))$ 
 $(\text{fst } (\text{assert } ! i)) (\text{snd } (\text{assert } ! i)) (\text{extend } E1 (\text{snd } (\text{extractP } (\text{fst } (\text{alts } !$ 
 $i)))) \text{ vs}, E2) (h, k) \subseteq$ 
 $\text{RSet } (\text{insert } x (\bigcup_{i < \text{length } \text{alts}} \text{fst } (\text{assert } ! i) - \text{set } (\text{snd } (\text{extractP } (\text{fst } (\text{alts } !$ 
 $i))))))$ 
 $(\text{foldl } \text{op } \otimes \text{ empty } (\text{map } \text{snd } \text{assert})) (E1, E2) (h, k)$ 
apply (simp add: RSet-def add: RSet2-def)
apply (unfold live-def)
apply (unfold closureLS-def)
apply simp
apply (rule subsetI)
apply simp
apply (elim conjE)
apply (rename-tac q)
apply (rule conjI)
apply (erule bexE)
apply (rename-tac q y)
apply (case-tac y  $\notin \text{set } (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! i))))$ )
apply (rule disjI2)
apply (rule-tac x=i in bexI) prefer 2 apply simp
apply (rule-tac x=y in bexI) prefer 2 apply simp
apply (subgoal-tac y  $\in \text{dom } E1$ )
apply (frule-tac E'=E2 and h=h and k=k
  in closure-monotone-extend-3)
apply (simp, simp, simp, simp)
apply (erule-tac x=i in allE, simp)
apply (erule-tac x=i in allE, simp)

```

```

apply (erule-tac  $x=i$  in allE,simp)
apply (erule-tac  $x=i$  in allE,simp)
apply (elim conjE)
apply (subgoal-tac  $y \in \text{dom } (\text{extend } E1 \ (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! \ i)))) \ vs))$ )
apply (rule extend-prop1,assumption+)
apply blast
apply simp
apply (subgoal-tac
   $\text{closure } (\text{extend } E1 \ (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! \ i)))) \ vs, \ E2) \ y \ (h, \ k) \subseteq$ 
   $\text{closure } (\text{extend } E1 \ (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! \ i)))) \ vs, \ E2) \ x \ (h, \ k))$ )
prefer 2 apply (rule patrones,assumption+) apply simp apply assumption+
apply (rule disjI1)
apply (subgoal-tac  $x \in \text{dom } E1$ )
apply (erule-tac  $E'=E2$  and  $h=h$  and  $k=k$  in closure-monotone-extend-3,assumption+,simp,
assumption+,simp)
apply blast
apply (simp add: dom-def)

apply (erule bexE)
apply (erule bexE)
apply simp
apply (elim conjE)
apply (erule disjE)
apply (rule disjI2)
apply simp
apply (rule-tac  $x=i$  in bexI) prefer 2 apply simp
apply (rule-tac  $x=x$  in bexI) prefer 2 apply simp
apply (rule conjI)
apply (subgoal-tac
   $i < \text{length } (\text{map } \text{snd } \text{assert}) \longrightarrow$ 
   $\text{length } (\text{map } \text{snd } \text{assert}) > 0 \longrightarrow$ 
   $\text{def-nonDisjointUnionEnvList } (\text{map } \text{snd } \text{assert}) \longrightarrow$ 
   $\text{snd } (\text{assert } ! \ i) \ x = \text{Some } d'' \longrightarrow$ 
   $\text{foldl } \text{op } \otimes \ \text{empty } (\text{map } \text{snd } \text{assert}) \ x = \text{Some } d'', \text{simp}$ )
apply (rule Otimes-prop3)
apply (subgoal-tac  $x \in \text{dom } E1$ )
apply (erule-tac  $E'=E2$  and  $h=h$  and  $k=k$  in recReach-monotone-extend-3,assumption+,simp,
assumption+,simp)
apply (simp add: dom-def)
apply (rule disjI2)
apply (erule bexE)
apply (elim conjE)
apply (rule-tac  $x=ia$  in bexI) prefer 2 apply simp
apply (rule-tac  $x=z$  in bexI) prefer 2 apply simp
apply (rule conjI)
apply (subgoal-tac
   $i < \text{length } (\text{map } \text{snd } \text{assert}) \longrightarrow$ 
   $\text{length } (\text{map } \text{snd } \text{assert}) > 0 \longrightarrow$ 
   $\text{def-nonDisjointUnionEnvList } (\text{map } \text{snd } \text{assert}) \longrightarrow$ 

```



```

    snd (assert ! i) z = Some d''  $\longrightarrow$ 
    foldl op  $\otimes$  empty (map snd assert) z = Some d'',simp)
  apply (rule Otimes-prop3)
  apply (case-tac z  $\in$  set (snd (extractP (fst (alts ! i)))))
  apply (case-tac i=ia, simp)
  apply (rotate-tac 6)
  apply (erule-tac x=ia in allE)
  apply (drule mp,simp)
  apply (rotate-tac 6)
  apply (erule-tac x=i in allE,simp)
  apply simp
  apply (subgoal-tac z  $\in$  dom E1)
  apply (frule-tac E'=E2 and h=h and k=k in recReach-monotone-extend-3,assumption+,simp,
    assumption+,simp)
  apply (erule-tac x=i in allE,simp)
  apply (erule-tac x=i in allE,simp)
  apply (erule-tac x=i in allE,simp)
  apply (elim conjE)
  apply (subgoal-tac z  $\in$  dom (extend E1 (snd (extractP (fst (alts ! i))))) vs))
  apply (rule extend-prop1,assumption+)
  by blast

```

**lemma** P7-case-dem1-5-1:

```

  [ E1 x = Some (Loc p);
    h p = Some (j,C,vs);
     $\forall i < \text{length alts}. x \in \text{fst} (\text{assert ! } i) \wedge x \notin \text{set} (\text{snd} (\text{extractP} (\text{fst} (\text{alts ! } i))));$ 
    (foldl op  $\otimes$  empty (map snd assert)) x = Some s'';
    set (snd (extractP (fst (alts ! i))))  $\cap$  dom E1 = {};
    length (snd (extractP (fst (alts ! i)))) = length vs;
    length assert > 0;
    length assert = length alts;
    i < length assert ]  $\implies$ 
    SSet ((fst (assert ! i))  $\cap$  set (snd (extractP (fst (alts ! i)))))
      (snd (assert ! i)) (extend E1 (snd (extractP (fst (alts ! i))))) vs, E2) (h,
k)  $\subseteq$ 
    SSet (insert x ( $\bigcup_{i < \text{length alts}} \text{fst} (\text{assert ! } i) - \text{set} (\text{snd} (\text{extractP} (\text{fst} (\text{alts}
! i)))))$ )
      (foldl op  $\otimes$  empty (map snd assert)) (E1, E2) (h, k)
  apply (rule subsetI)
  apply (simp add: SSet-def, simp add: Let-def)
  apply (elim exE, elim conjE)
  apply (rule-tac x=x in exI)
  apply (rule conjI,simp)

```

```

apply (rule conjI,simp)
apply (frule patrones [where ?E2.0=E2 and k=k],assumption+)
apply (simp,assumption+)
apply (subgoal-tac
  closure (E1, E2) x (h, k) =
  closure (extend E1 (snd (extractP (fst (alts ! i)))) vs, E2) x (h, k),simp)
apply blast
apply (rule closure-monotone-extend-3)
by (simp, simp, simp add:dom-def,simp,simp)

```

**lemma** P7-case-dem1-5-2:

```

  [| (foldl op ⊗ empty (map snd assert)) x = Some s'';
    ∀ i < length assert. ∀ x ∈ set (snd (extractP (fst (alts ! i)))). snd (assert !
i) x ≠ Some d'';
    length assert = length alts;
    i < length assert |] ⇒
    RSet2 ((fst (assert ! i)) ∩ set (snd (extractP (fst (alts ! i)))))
      (fst (assert ! i)) (snd (assert ! i)) (extend E1 (snd (extractP (fst (alts
! i))))) vs, E2) (h, k) = {}
apply (erule-tac x=i in allE) apply simp
by (simp add: RSet2-def)

```

**lemma** P7-case-dem1-6-1:

```

  [| (foldl op ⊗ empty (map snd assert)) x ≠ Some s'';
    shareRec (fst (assert ! i)) (snd (assert ! i)) (extend E1 (snd (extractP (fst (alts
! i))))) vs, E2) (h, k) (hh, k);
    ∀ i < length alts. x ∈ fst (assert ! i) ∧ x ∉ set (snd (extractP (fst (alts ! i))));
    ∀ i < length assert. ∀ x ∈ set (snd (extractP (fst (alts ! i)))). snd (assert !
i) x ≠ Some d'';
    ∀ i < length assert. fst (assert ! i) ⊆ dom (snd (assert ! i));
    set (snd (extractP (fst (alts ! i)))) ∩ dom E1 = {};
    length (snd (extractP (fst (alts ! i)))) = length vs;
    def-nonDisjointUnionEnvList (map snd assert);
    x ∈ dom (nonDisjointUnionEnvList (map snd assert));
    E1 x = Some (Loc p);
    length assert > 0;
    length assert = length alts;
    i < length assert |] ⇒
    SSet ((fst (assert ! i)) ∩ set (snd (extractP (fst (alts ! i)))))
      (snd (assert ! i)) (extend E1 (snd (extractP (fst (alts ! i))))) vs, E2) (h,
k) ∩
    RSet2 ((fst (assert ! i)) ∩ (insert x (⋃ i < length alts fst (assert ! i) - set
(snd (extractP (fst (alts ! i)))))

```

```

      (fst (assert ! i)) (snd (assert ! i)) (extend E1 (snd (extractP (fst (alts
! i)))) vs, E2) (h, k) = {}
apply (rule equalityI)
prefer 2 apply simp
apply (rule subsetI)
apply (simp add: SSet-def)
apply (simp add: Let-def)
apply (elim conjE)
apply (elim exE)
apply (elim conjE)
apply (rename-tac xij)
apply (case-tac
  (∃ z ∈ fst (assert ! i) - set (snd (extractP (fst (alts ! i))))).
    snd (assert ! i) z = Some d'' ∧
    closure (extend E1 (snd (extractP (fst (alts ! i)))) vs, E2) xij (h, k) ∩
    recReach (extend E1 (snd (extractP (fst (alts ! i)))) vs, E2) z (h, k) ≠ {})
apply (elim bexE)
apply (elim conjE)
apply (simp add: shareRec-def)
apply (elim conjE)
apply (erule-tac x=xij in ballE)
prefer 2 apply (simp add: extend-def) apply force
apply (rotate-tac 19)
apply (erule thin-rl)
apply (drule mp)
apply (rule-tac x=z in bexI)
prefer 2 apply simp
apply (rule conjI)
apply simp
apply simp
apply simp

apply simp

apply (simp only: RSet2-def)
apply (simp only: live-def)
apply (simp only: closureLS-def)
apply simp
apply (elim conjE)
apply (elim bexE)
apply (elim conjE)
apply (erule-tac x=z in ballE)
prefer 2 apply simp
apply simp
apply (elim conjE)
apply (subgoal-tac ∃ w. w ∈ closureL xa (h, k) ∧ w ∈ recReach (extend E1 (snd
(extractP (fst (alts ! i)))) vs, E2) z (h, k))
prefer 2 apply blast

```

```

apply (elim exE,elim conjE)
apply (frule pp12) apply simp apply simp apply simp
done

```

**lemma** *P7-case-dem1-6-2*:

```


$$\llbracket \forall i < \text{length } \text{assert}. \forall x \in \text{set } (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! i)))) . \text{snd } (\text{assert } ! i) x \neq \text{Some } d''; \\ i < \text{length } \text{assert}; \text{length } \text{assert} = \text{length } \text{alts} \rrbracket \\ \implies \text{RSet2 } ((\text{fst } (\text{assert } ! i)) \cap \text{set } (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! i)))) \\ (\text{fst } (\text{assert } ! i)) (\text{snd } (\text{assert } ! i)) (\text{extend } E1 (\text{snd } (\text{extractP } (\text{fst } (\text{alts } \\ ! i)))) \text{vs}, E2) (h, k) = \{\}$$

apply (rule equalityI)
apply (rule subsetI)
apply (simp only: RSet2-def)
apply (simp only: live-def)
apply (simp only: closureLS-def)
apply simp
apply simp
done

```

**lemma** *P7-CASE*:

```


$$\llbracket \text{length } \text{assert} = \text{length } \text{alts}; \\ \forall i < \text{length } \text{assert}. \forall x \in \text{set } (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! i)))) . \text{snd } (\text{assert } ! i) x \neq \text{Some } d''; \\ \forall i < \text{length } \text{assert}. \text{fst } (\text{assert } ! i) \subseteq \text{dom } (\text{snd } (\text{assert } ! i)); \\ \text{dom } (\text{snd } (\text{assert } ! i)) \subseteq \text{dom } (\text{extend } E1 (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! i)))) \text{vs}); \\ \text{set } (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! i)))) \cap \text{dom } E1 = \{\}; \\ \text{length } (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! i)))) = \text{length } \text{vs}; \\ \forall i < \text{length } \text{alts}. \forall j < \text{length } \text{alts}. i \neq j \longrightarrow (\text{fst } (\text{assert } ! i) \cap \text{set } (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! j)))) = \{\}; \\ \forall i < \text{length } \text{alts}. x \in \text{fst } (\text{assert } ! i) \wedge x \notin \text{set } (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! i)))) \\ \text{shareRec } (\text{fst } (\text{assert } ! i)) (\text{snd } (\text{assert } ! i)) (\text{extend } E1 (\text{snd } (\text{extractP } (\text{fst } (\text{alts } \\ ! i)))) \text{vs}, E2) (h, k) (hh, k); \\ \text{def-nonDisjointUnionEnvList } (\text{map } \text{snd } \text{assert}); \\ E1 x = \text{Some } (\text{Loc } p); \\ h p = \text{Some } (j, C, \text{vs}); \\ x \in \text{dom } (\text{nonDisjointUnionEnvList } (\text{map } \text{snd } \text{assert})); \\ \text{length } \text{assert} > 0; \\ \text{SSet } (\text{insert } x (\bigcup_{i < \text{length } \text{alts}} \text{fst } (\text{assert } ! i) - \text{set } (\text{snd } (\text{extractP } (\text{fst } (\text{alts } \\ ! i))))) \\ (\text{foldl } \text{op } \otimes \text{empty } (\text{map } \text{snd } \text{assert})) (E1, E2) (h, k) \cap \\ \text{RSet } (\text{insert } x (\bigcup_{i < \text{length } \text{alts}} \text{fst } (\text{assert } ! i) - \text{set } (\text{snd } (\text{extractP } (\text{fst } (\text{alts } \\ ! i))))) \\ (\text{foldl } \text{op } \otimes \text{empty } (\text{map } \text{snd } \text{assert})) (E1, E2) (h, k) = \{\}; \\ i < \text{length } \text{assert} \rrbracket \\ \implies$$


```

```

      SSet (fst (assert ! i)) (snd (assert ! i)) (extend E1 (snd (extractP (fst (alts
! i)))) vs, E2) (h, k) ∩
      RSet (fst (assert ! i)) (snd (assert ! i)) (extend E1 (snd (extractP (fst (alts
! i)))) vs, E2) (h, k) = {}
apply (rule P7-case-dem1)
apply (rule P7-case-dem1-1,assumption+)
apply (rule P7-case-dem1-2,assumption+)
apply (rule P7-case-dem1-3,assumption+)
apply (rule P7-case-dem1-4,assumption+)
apply (rule impI, rule conjI)
apply (rule P7-case-dem1-5-1,assumption+)
apply (rule P7-case-dem1-5-2,assumption+)
apply (rule impI, rule conjI)
apply (rule P7-case-dem1-6-1,assumption+)
apply (rule P7-case-dem1-6-2,assumption+)
done

```

Lemmas for CASEL Rule

**lemma** P7-casel-dem1:

```

  [| Si = S'i ∪ S''i;
    Ri = R'i ∪ R''i;
    S'i ⊆ S;
    R'i ⊆ R;
    Γ x = Some s'' ⟶ S''i ⊆ S ∧ R''i = {};
    Γ x ≠ Some s'' ⟶ S''i ∩ R'i = {} ∧ R''i = {};
    S ∩ R = {} |]
  ⟹ Si ∩ Ri = {}
by blast

```

**lemma** P7-casel-dem1-1:

```

  [| length assert = length alts; i < length assert |] ⟹
    SSet (fst (assert ! i)) (snd (assert ! i)) (E1, E2) (h, k) =
    SSet ((fst (assert ! i)) ∩ (insert x (⋃i < length alts fst (assert ! i) - set (snd
(extractP (fst (alts ! i)))))))
      (snd (assert ! i)) (E1, E2) (h, k) ∪
    SSet ((fst (assert ! i)) ∩ set (snd (extractP (fst (alts ! i)))))
      (snd (assert ! i)) (E1, E2) (h, k)
by (simp add: SSet-def add: Let-def,blast)

```

**lemma** P7-casel-dem1-2:

```

  [| length assert = length alts; i < length assert |] ⟹
    RSet (fst (assert ! i)) (snd (assert ! i)) (E1, E2) (h, k) =
    RSet2 ((fst (assert ! i)) ∩ (insert x (⋃i < length alts fst (assert ! i) - set
(snd (extractP (fst (alts ! i)))))))
      (fst (assert ! i)) (snd (assert ! i)) (E1, E2) (h, k) ∪
    RSet2 ((fst (assert ! i)) ∩ set (snd (extractP (fst (alts ! i)))))

```

$(fst (assert ! i)) (snd (assert ! i)) (E1, E2) (h, k)$   
**apply** (*simp* add: RSet-def add: RSet2-def add: live-def add: closureLS-def)  
**by** blast

**lemma** *P7-case1-dem1-3*:

$\llbracket$  def-nonDisjointUnionEnvList (map snd assert);  
 length assert > 0;  
 $x \in dom (nonDisjointUnionEnvList (map snd assert));$   
 length assert = length alts;  
 $i < length assert \rrbracket \implies$   
 $SSet ((fst (assert ! i)) \cap (insert x (\bigcup_{i < length alts} fst (assert ! i) - set (snd$   
 $(extractP (fst (alts ! i))))))$   
 $(snd (assert ! i)) (E1, E2) (h, k) \subseteq$   
 $SSet (insert x (\bigcup_{i < length alts} fst (assert ! i) - set (snd (extractP (fst (alts$   
 $! i))))))$   
 $(foldl op \otimes empty (map snd assert)) (E1, E2) (h, k)$   
**apply** (*clarsimp*)  
**apply** (*simp* add: SSet-def add: Let-def)  
**apply** (*erule* exE, *elim* conjE)  
**apply** (*erule* disjE)

**apply** *simp*  
**apply** (*rule-tac*  $x=x$  **in** *exI*)  
**apply** (*rule* conjI)  
**apply** (*rule* disjI1) **apply** *simp*  
**apply** (*rule* conjI)  
**apply** (*subgoal-tac*  
 $i < length (map snd assert) \longrightarrow$   
 $length (map snd assert) > 0 \longrightarrow$   
 $def-nonDisjointUnionEnvList (map snd assert) \longrightarrow$   
 $snd (assert ! i) x = Some s'' \longrightarrow$   
 $foldl op \otimes empty (map snd assert) x = Some s'', simp$ )  
**apply** (*rule* Otimes-prop3)  
**apply** *simp*

**apply** (*rule-tac*  $x=xaa$  **in** *exI*)  
**apply** (*erule* bexE, *simp*, *elim* conjE)  
**apply** (*rule* conjI)  
**apply** (*rule* disjI2)  
**apply** (*rule-tac*  $x=ia$  **in** *bexI*, *simp*, *simp*)  
**apply** (*subgoal-tac*  
 $i < length (map snd assert) \longrightarrow$   
 $length (map snd assert) > 0 \longrightarrow$   
 $def-nonDisjointUnionEnvList (map snd assert) \longrightarrow$   
 $snd (assert ! i) xaa = Some s'' \longrightarrow$   
 $foldl op \otimes empty (map snd assert) xaa = Some s'', simp$ )  
**by** (*rule* Otimes-prop3)

**lemma** *P7-case1-dem1-4*:

```

 $\llbracket \forall i < \text{length } \text{alts}. x \in \text{fst } (\text{assert } ! i) \wedge x \notin \text{set } (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! i))));$ 
 $\text{fst } (\text{assert } ! i) \subseteq \text{dom } (\text{snd } (\text{assert } ! i));$ 
 $\text{dom } (\text{snd } (\text{assert } ! i)) \subseteq \text{dom } E1;$ 
 $\text{def-nonDisjointUnionEnvList } (\text{map } \text{snd } \text{assert});$ 
 $\text{length } \text{assert} > 0;$ 
 $(E1 \ x = \text{Some } aa \wedge aa = \text{IntT } n \wedge \text{fst } (\text{alts } ! i) = \text{ConstP } (\text{LitN } n) \vee$ 
 $E1 \ x = \text{Some } aa \wedge aa = \text{BoolT } b \wedge \text{fst } (\text{alts } ! i) = \text{ConstP } (\text{LitB } b));$ 
 $x \in \text{dom } (\text{nonDisjointUnionEnvList } (\text{map } \text{snd } \text{assert}));$ 
 $\text{length } \text{assert} = \text{length } \text{alts};$ 
 $i < \text{length } \text{assert} \rrbracket$ 
 $\implies \text{RSet2 } ((\text{fst } (\text{assert } ! i)) \cap (\text{insert } x (\bigcup_{i < \text{length } \text{alts}} \text{fst } (\text{assert } ! i) - \text{set}$ 
 $(\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! i))))))$ 
 $(\text{fst } (\text{assert } ! i)) (\text{snd } (\text{assert } ! i)) (E1, E2) (h, k) \subseteq$ 
 $\text{RSet } (\text{insert } x (\bigcup_{i < \text{length } \text{alts}} \text{fst } (\text{assert } ! i) - \text{set } (\text{snd } (\text{extractP } (\text{fst } (\text{alts}$ 
 $! i))))))$ 
 $(\text{foldl } \text{op } \otimes \text{empty } (\text{map } \text{snd } \text{assert})) (E1, E2) (h, k)$ 
apply (simp add: RSet-def add: RSet2-def)
apply (unfold live-def)
apply (unfold closureLS-def)
apply simp
apply (rule subsetI)
apply simp
apply (elim conjE)
apply (rename-tac q)
apply (rule conjI)
apply clarsimp

apply (erule disjE)
apply clarsimp
apply clarsimp

apply (erule bexE)
apply (erule bexE)
apply simp
apply (elim conjE)
apply (erule-tac P=z=x in disjE)
apply (rule disjI2)
apply (rule-tac x=i in bexI) prefer 2 apply simp
apply (rule-tac x=x in bexI) prefer 2 apply simp
apply (rule conjI)
apply (subgoal-tac
 $i < \text{length } (\text{map } \text{snd } \text{assert}) \longrightarrow$ 
 $\text{length } (\text{map } \text{snd } \text{assert}) > 0 \longrightarrow$ 
 $\text{def-nonDisjointUnionEnvList } (\text{map } \text{snd } \text{assert}) \longrightarrow$ 
 $\text{snd } (\text{assert } ! i) \ x = \text{Some } d'' \longrightarrow$ 

```

```

    foldl op  $\otimes$  empty (map snd assert) x = Some d'',simp)
  apply (rule Otimes-prop3)
  apply simp
  apply (rule disjI2)
  apply (rule-tac x=i in bexI)
  prefer 2 apply simp
  apply (rule-tac x=z in bexI)
  prefer 2 apply force
  apply (rule conjI)
  apply (subgoal-tac
    i < length (map snd assert)  $\longrightarrow$ 
    length (map snd assert) > 0  $\longrightarrow$ 
    def-nonDisjointUnionEnvList (map snd assert)  $\longrightarrow$ 
    snd (assert ! i) z = Some d''  $\longrightarrow$ 
    foldl op  $\otimes$  empty (map snd assert) z = Some d'',simp)
  apply (rule Otimes-prop3)
  by simp

```

**lemma** *P7-casel-dem1-5-1:*

```

   $\llbracket (E1\ x = \text{Some}\ aa \wedge aa = \text{IntT}\ n \wedge \text{fst}\ (\text{alts}\ !\ i) = \text{ConstP}\ (\text{LitN}\ n)) \vee$ 
     $(E1\ x = \text{Some}\ aa \wedge aa = \text{BoolT}\ b \wedge \text{fst}\ (\text{alts}\ !\ i) = \text{ConstP}\ (\text{LitB}\ b)) ;$ 
    length assert > 0;
    length assert = length alts;
    i < length assert  $\rrbracket \implies$ 
    SSet ((fst (assert ! i))  $\cap$  set (snd (extractP (fst (alts ! i)))))
      (snd (assert ! i)) (E1,E2) (h, k)  $\subseteq$ 
    SSet (insert x ( $\bigcup_{i < \text{length}\ alts} \text{fst}\ (\text{assert}\ !\ i) - \text{set}\ (\text{snd}\ (\text{extractP}\ (\text{fst}\ (\text{alts}\ !\ i))))$ ))
      (foldl op  $\otimes$  empty (map snd assert)) (E1, E2) (h, k)
  apply (rule subsetI)
  apply (erule disjE)
  by (simp add: SSet-def)+

```

**lemma** *P7-casel-dem1-5-2:*

```

   $\llbracket (\text{foldl}\ op\ \otimes\ \text{empty}\ (\text{map}\ \text{snd}\ \text{assert}))\ x = \text{Some}\ s'';$ 
     $(E1\ x = \text{Some}\ aa \wedge aa = \text{IntT}\ n \wedge \text{fst}\ (\text{alts}\ !\ i) = \text{ConstP}\ (\text{LitN}\ n)) \vee$ 
     $(E1\ x = \text{Some}\ aa \wedge aa = \text{BoolT}\ b \wedge \text{fst}\ (\text{alts}\ !\ i) = \text{ConstP}\ (\text{LitB}\ b));$ 
    length assert = length alts;
    i < length assert  $\rrbracket \implies$ 
    RSet2 ((fst (assert ! i))  $\cap$  set (snd (extractP (fst (alts ! i)))))
      (fst (assert ! i)) (snd (assert ! i)) (E1,E2) (h, k) = {}
  apply (erule disjE)
  by (simp add: RSet2-def)+

```



**lemma** *P7-casel-dem1-6-1*:

$\llbracket (\text{foldl } op \otimes \text{empty } (\text{map } \text{snd } \text{assert})) \ x \neq \text{Some } s'';$   
 $(E1 \ x = \text{Some } aa \wedge aa = \text{IntT } n \wedge \text{fst } (alts \ ! \ i) = \text{ConstP } (\text{LitN } n)) \vee$   
 $(E1 \ x = \text{Some } aa \wedge aa = \text{BoolT } b \wedge \text{fst } (alts \ ! \ i) = \text{ConstP } (\text{LitB } b));$   
 $\text{length } \text{assert} > 0;$   
 $\text{length } \text{assert} = \text{length } alts;$   
 $i < \text{length } \text{assert} \rrbracket \implies$   
 $\text{SSet } ((\text{fst } (\text{assert } ! \ i)) \cap \text{set } (\text{snd } (\text{extractP } (\text{fst } (alts \ ! \ i)))))$   
 $(\text{snd } (\text{assert } ! \ i)) \ (E1, E2) \ (h, k) \cap$   
 $\text{RSet2 } ((\text{fst } (\text{assert } ! \ i)) \cap (\text{insert } x \ (\bigcup_{i < \text{length } alts} \text{fst } (\text{assert } ! \ i) - \text{set}$   
 $(\text{snd } (\text{extractP } (\text{fst } (alts \ ! \ i)))))$   
 $(\text{fst } (\text{assert } ! \ i)) \ (\text{snd } (\text{assert } ! \ i)) \ (E1, E2) \ (h, k) = \{\}$   
**apply** (*rule equalityI*)  
**prefer** 2 **apply** (*simp*)  
**apply** (*rule subsetI*)  
**apply** (*erule disjE*)  
**by** (*simp add: SSet-def*)+

**lemma** *P7-casel-dem1-6-2*:

$\llbracket (E1 \ x = \text{Some } aa \wedge aa = \text{IntT } n \wedge \text{fst } (alts \ ! \ i) = \text{ConstP } (\text{LitN } n)) \vee$   
 $(E1 \ x = \text{Some } aa \wedge aa = \text{BoolT } b \wedge \text{fst } (alts \ ! \ i) = \text{ConstP } (\text{LitB } b));$   
 $\text{length } \text{assert} > 0; i < \text{length } \text{assert}; \text{length } \text{assert} = \text{length } alts \rrbracket$   
 $\implies \text{RSet2 } ((\text{fst } (\text{assert } ! \ i)) \cap \text{set } (\text{snd } (\text{extractP } (\text{fst } (alts \ ! \ i)))))$   
 $(\text{fst } (\text{assert } ! \ i)) \ (\text{snd } (\text{assert } ! \ i)) \ (E1, E2) \ (h, k) = \{\}$   
**apply** (*rule equalityI*)  
**apply** (*rule subsetI*)  
**apply** (*simp only: RSet2-def*)  
**apply** (*simp only: live-def*)  
**apply** (*simp only: closureLS-def*)  
**apply** (*erule disjE*)  
**apply** (*simp*)  
**apply** (*simp*)  
**by** (*simp*)

**lemma** *P7-CASEL*:

$\llbracket \forall \ i < \text{length } alts. \ x \in \text{fst } (\text{assert } ! \ i) \wedge x \notin \text{set } (\text{snd } (\text{extractP } (\text{fst } (alts \ ! \ i))));$   
 $x \in \text{dom } (\text{nonDisjointUnionEnvList } (\text{map } \text{snd } \text{assert}));$   
 $\text{dom } (\text{nonDisjointUnionEnvList } (\text{map } \text{snd } \text{assert})) \subseteq \text{dom } E1;$   
 $(E1 \ x = \text{Some } aa \wedge aa = \text{IntT } n \wedge \text{fst } (alts \ ! \ i) = \text{ConstP } (\text{LitN } n)) \vee$   
 $(E1 \ x = \text{Some } aa \wedge aa = \text{BoolT } b \wedge \text{fst } (alts \ ! \ i) = \text{ConstP } (\text{LitB } b));$   
 $i < \text{length } alts;$

```

    0 < length assert;
    def-nonDisjointUnionEnvList (map snd assert);
    0 < length (map snd assert);
    length assert = length alts;
    fst (assert ! i) ⊆ dom (snd (assert ! i));
    SSet ((⋃i < length alts fst (assert ! i) - set (snd (extractP (fst (alts ! i)))))
    ∪ {x})
    (nonDisjointUnionEnvList (map snd assert)) (E1, E2) (h, k) ∩
    RSet ((⋃i < length alts fst (assert ! i) - set (snd (extractP (fst (alts ! i)))))
    ∪ {x})
    (nonDisjointUnionEnvList (map snd assert)) (E1, E2) (h, k) =
    {};
    dom (snd (assert ! i)) ⊆ dom E1]]
    ⇒ SSet (fst (assert ! i)) (snd (assert ! i)) (E1, E2) (h, k) ∩ RSet (fst (assert
    ! i)) (snd (assert ! i)) (E1, E2) (h, k) = {}
apply (rule P7-casel-dem1)
apply (rule P7-casel-dem1-1,assumption+,simp)
apply (rule P7-casel-dem1-2,assumption+,simp)
apply (rule P7-casel-dem1-3,assumption+,simp)
apply (rule P7-casel-dem1-4,assumption+,simp)
apply (rule impI, rule conjI)
apply (rule P7-casel-dem1-5-1,assumption+,simp)
apply (rule P7-casel-dem1-5-2,assumption+,simp)
apply (rule impI, rule conjI)
apply (rule P7-casel-dem1-6-1,assumption+,simp)
apply (rule P7-casel-dem1-6-2,assumption+,simp,simp)
by simp

```

Lemmas for CASED Rule

**lemma** *P7-cased-dem1*:

```

[[ Si = S'i ∪ S''i;
   Ri = R'i ∪ R''i;
   S'i ⊆ S;
   R'i ⊆ R;
   S''i ∩ Ri = {};
   S'i ∩ R''i = {};
   S ∩ R = {}]]
⇒ Si ∩ Ri = {}
by blast

```

**lemma** *P7-cased-dem1-1*:

```

[[ length assert = length alts; i < length assert]] ⇒
  SSet (fst (assert ! i)) (snd (assert ! i)) (extend E1 (snd (extractP (fst (alts
  ! i)))) vs, E2) (h(p := None), k) =
  SSet ((fst (assert ! i)) ∩ (⋃i < length alts fst (assert ! i) - set (snd (extractP
  (fst (alts ! i)))))
  (snd (assert ! i)) (extend E1 (snd (extractP (fst (alts ! i)))) vs, E2)
  (h(p := None), k) ∪

```

$S\text{Set } ((fst \ (assert \ ! \ i)) \cap \ set \ (snd \ (extractP \ (fst \ (alts \ ! \ i)))))$   
 $(snd \ (assert \ ! \ i)) \ (extend \ E1 \ (snd \ (extractP \ (fst \ (alts \ ! \ i)))) \ vs, \ E2)$   
 $(h(p := None), k)$   
**by**  $(simp \ add: \ S\text{Set-def} \ add: \ Let\text{-def}, blast)$

**lemma** *P7-cased-dem1-2*:

$\llbracket \text{length } assert = \text{length } alts; i < \text{length } assert \rrbracket \implies$   
 $R\text{Set } (fst \ (assert \ ! \ i)) \ (snd \ (assert \ ! \ i)) \ (extend \ E1 \ (snd \ (extractP \ (fst \ (alts \ ! \ i)))) \ vs, \ E2) \ (h(p := None), k) =$   
 $R\text{Set2 } ((fst \ (assert \ ! \ i)) \cap \ (insert \ x \ (\bigcup_{i < \text{length } alts} fst \ (assert \ ! \ i) - \ set$   
 $(snd \ (extractP \ (fst \ (alts \ ! \ i))))))$   
 $(fst \ (assert \ ! \ i)) \ (snd \ (assert \ ! \ i)) \ (extend \ E1 \ (snd \ (extractP \ (fst \ (alts \ ! \ i)))) \ vs, \ E2) \ (h(p := None), k) \cup$   
 $R\text{Set2 } ((fst \ (assert \ ! \ i)) \cap \ set \ (snd \ (extractP \ (fst \ (alts \ ! \ i)))))$   
 $(fst \ (assert \ ! \ i)) \ (snd \ (assert \ ! \ i)) \ (extend \ E1 \ (snd \ (extractP \ (fst \ (alts \ ! \ i)))) \ vs, \ E2) \ (h(p := None), k)$   
**apply**  $(simp \ add: \ R\text{Set-def} \ add: \ R\text{Set2-def} \ add: \ live\text{-def} \ add: \ closureLS\text{-def})$   
**by** *blast*

**lemma** *P7-cased-dem1-3*:

$\llbracket \text{length } assert > 0;$   
 $E1 \ x = Some \ (Loc \ p); h \ p = Some \ (j, C, vs);$   
 $\forall \ z \in \text{dom } \Gamma. \ \Gamma \ z \neq Some \ s'' \longrightarrow (\forall \ i < \text{length } alts. \ z \notin fst \ (assert \ ! \ i));$   
 $\Gamma = disjointUnionEnv$   
 $(nonDisjointUnionEnvList \ ((map \ (\lambda(Li, \Gamma i). \ restrict\text{-neg-map } \Gamma i \ (set \ Li \cup \{x\})))$   
 $(zip \ (map \ (snd \ o \ extractP \ o \ fst) \ alts) \ (map \ snd \ assert))))$   
 $(empty(x \mapsto d''));$   
 $\forall \ i < \text{length } alts. \ \forall \ j < \text{length } alts. \ i \neq j \longrightarrow (fst \ (assert \ ! \ i) \cap \ set \ (snd$   
 $(extractP \ (fst \ (alts \ ! \ j))))) = \{\};$   
 $def\text{-}nonDisjointUnionEnvList$   
 $(map \ (\lambda(Li, \Gamma i). \ restrict\text{-neg-map } \Gamma i \ (insert \ x \ (set \ Li)))$   
 $(zip \ (map \ (snd \ o \ extractP \ o \ fst) \ alts) \ (map \ snd \ assert))));$   
 $def\text{-}disjointUnionEnv$   
 $(nonDisjointUnionEnvList \ ((map \ (\lambda(Li, \Gamma i). \ restrict\text{-neg-map } \Gamma i \ (set$   
 $Li \cup \{x\})))$   
 $(zip \ (map \ (snd \ o \ extractP \ o \ fst) \ alts) \ (map \ snd \ assert))))$   
 $[x \mapsto d''];$   
 $shareRec \ (insert \ x \ (\bigcup_{i < \text{length } alts} fst \ (assert \ ! \ i) - \ set \ (snd \ (extractP \ (fst$   
 $(alts \ ! \ i)))))$   
 $\Gamma \ (E1, \ E2) \ (h, \ k) \ (hh, \ k);$   
 $set \ (snd \ (extractP \ (fst \ (alts \ ! \ i)))) \cap \ \text{dom } E1 = \{\};$   
 $\text{length } (snd \ (extractP \ (fst \ (alts \ ! \ i)))) = \text{length } vs;$   
 $\text{dom } (snd \ (assert \ ! \ i)) \subseteq \text{dom } (extend \ E1 \ (snd \ (extractP \ (fst \ (alts \ ! \ i)))) \ vs);$   
 $\text{length } assert = \text{length } alts;$

```

    i < length assert  $\mathbb{I}$ 
 $\implies$  SSet ((fst (assert ! i))  $\cap$  ( $\bigcup_{i < \text{length alts}} \text{fst}(\text{assert} ! i) - \text{set}(\text{snd}(\text{extractP}$ 
(fst (alts ! i))))))
    (snd (assert ! i)) (extend E1 (snd (extractP (fst (alts ! i)))) vs, E2)
(h(p := None), k)  $\subseteq$ 
    SSet (( $\bigcup_{i < \text{length alts}} \text{fst}(\text{assert} ! i) - \text{set}(\text{snd}(\text{extractP}(\text{fst}(\text{alts} ! i))))$ 
 $\cup \{x\}$ )
     $\Gamma(E1, E2)(h, k)$ 
apply (rule subsetI)
apply (simp add: SSet-def add: Let-def)
apply (rename-tac q)
apply (erule exE, elim conjE)
apply (rename-tac y)
apply (erule bexE, simp, elim conjE)
apply (rule-tac x=y in exI)
apply (rule conjI)
apply (rule disjI2)
apply (rule-tac x=ia in bexI,simp,simp)
apply (case-tac y  $\in$  set (snd (extractP (fst (alts ! i)))))
apply (case-tac i=ia, simp)
apply (rotate-tac 5)
apply (erule-tac x=ia in allE,simp)
apply (rotate-tac 21)
apply (erule-tac x=i in allE,simp)
apply blast
apply (rule conjI)
apply (frule  $\Gamma$ -case-x-is-d)
apply (erule-tac x=x in ballE)
prefer 2 apply force
apply (drule mp, simp)
apply (case-tac y = x,blast)
apply (rule Otimes-prop4)
apply (simp,assumption+,force,assumption+)
apply (subgoal-tac
    y  $\in$  dom (extend E1 (snd (extractP (fst (alts ! i)))) vs))
prefer 2 apply blast
apply (frule dom-extend-in-E1-or-xs,assumption+)
apply simp
apply (rule closure-extend-None-subset-closure)
by assumption+

```

**lemma** P7-cased-dem1-4:

```

 $\mathbb{I}$  length assert > 0;
    E1 x = Some (Loc p); h p = Some (j,C,vs);
 $\Gamma = \text{disjointUnionEnv}$ 
    (nonDisjointUnionEnvList ((map ( $\lambda(Li,\Gamma i).$  restrict-neg-map  $\Gamma i$  (set Li $\cup\{x\}$ ))))

```

```

      (zip (map (snd o extractP o fst) alts) (map snd assert))))
    (empty(x ↦ d''));
  def-nonDisjointUnionEnvList
    (map (λ(Li, Γi). restrict-neg-map Γi (insert x (set Li)))
      (zip (map (snd o extractP o fst) alts) (map snd assert)));
  def-disjointUnionEnv
    (nonDisjointUnionEnvList ((map (λ(Li,Γi). restrict-neg-map Γi (set
Li ∪ {x})))
      (zip (map (snd o extractP o fst) alts) (map snd assert))))
    [x ↦ d''];
  ∀ i < length alts. ∀ j < length alts. i ≠ j → (fst (assert ! i) ∩ set (snd
(extractP (fst (alts ! j))))) = {};
  set (snd (extractP (fst (alts ! i)))) ∩ dom E1 = {};
  length (snd (extractP (fst (alts ! i)))) = length vs;
  dom (snd (assert ! i)) ⊆ dom (extend E1 (snd (extractP (fst (alts ! i)))) vs);
  ∀ i < length assert. fst (assert ! i) ⊆ dom (snd (assert ! i));
  length assert = length alts;
  i < length assert ]
  ⇒ RSet2 ((fst (assert ! i)) ∩ (insert x (⋃ i < length alts fst (assert ! i) - set
(snd (extractP (fst (alts ! i)))))
    (fst (assert ! i)) (snd (assert ! i)) (extend E1 (snd (extractP (fst (alts
! i)))) vs, E2) (h(p := None), k) ⊆
    RSet ((⋃ i < length alts fst (assert ! i) - set (snd (extractP (fst (alts ! i)))))
  ∪ {x})
    Γ (E1, E2) (h, k)
  apply (simp add: RSet-def add: RSet2-def)
  apply (unfold live-def)
  apply (unfold closureLS-def, simp)
  apply (rule subsetI, simp)
  apply (elim conjE)
  apply (rename-tac q)
  apply (erule bexE)
  apply (rename-tac y)
  apply (rule conjI)
  apply (case-tac y ∉ set (snd (extractP (fst (alts ! i)))))
  apply (rule disjI2)
  apply (rule-tac x=i in bexI)
  prefer 2 apply simp
  apply (rule-tac x=y in bexI)
  prefer 2 apply simp
  apply (subgoal-tac
    y ∈ dom (extend E1 (snd (extractP (fst (alts ! i)))) vs))
  prefer 2 apply (erule-tac x=i in allE, simp)+ apply blast
  apply (subgoal-tac y ∈ dom E1)
  prefer 2 apply (rule extend-prop1, simp, simp, simp)
  apply (rule closure-extend-None-subset-closure, assumption+)

  apply simp

```

```

apply (frule-tac  $k=k$  and  $?E2.0=E2$  in patrones-2)
apply (assumption+,force,assumption+)
apply (subgoal-tac  $q = p$ )
  prefer 2 apply blast
apply (rule disjI1)
apply clarsimp
apply (simp add: closure-def)
apply (rule closureL-basic)

apply (erule bexE,elim conjE)
apply (simp, elim conjE)
apply (erule disjE)
apply (rule disjI2,simp)
apply (rule-tac  $x=i$  in bexI)
  prefer 2 apply simp
apply (rule-tac  $x=x$  in bexI)
  prefer 2 apply blast
apply (rule conjI)
apply (frule  $\Gamma$ -case-x-is-d,simp)
apply (case-tac  $p=q$ )
apply (simp add: recReach-def)
apply (subgoal-tac  $p \in \text{closureL } p (h,k)$ )
apply (subgoal-tac  $p \in \text{recReachL } p (h,k)$ )
  apply blast
  apply (rule recReachL-basic)
  apply (rule closureL-basic)
apply (subgoal-tac  $x \in \text{dom } E1$ )
  prefer 2 apply (simp add: dom-def)
apply (subgoal-tac
   $\exists w. w \in \text{closureL } q (h(p := \text{None}), k) \wedge$ 
   $w \in \text{recReach } (\text{extend } E1 (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! i)))) \text{ vs, } E2) x (h(p$ 
   $:= \text{None}), k))$ )
  prefer 2 apply blast
apply (elim exE, elim conjE)
apply (frule-tac  $x=x$  and  $E=E1$  and  $vs=vs$  in extend-monotone-i,simp,blast)
apply (simp add: recReach-def)
apply (subgoal-tac  $w = p$ ,simp)
  prefer 2 apply (subst (asm) recReachL-p-None-p,simp) +
apply (frule-tac  $h=h$  and  $k=k$  in closureL-p-None-subseteq-closureL)
apply (subgoal-tac  $p \in \text{recReachL } p (h,k)$ )
apply (subgoal-tac  $p \in \text{closureL } q (h,k)$ )
  apply blast
  apply blast
  apply (rule recReachL-basic)
apply (rule disjI2)
apply (erule bexE,elim conjE)
apply simp
apply (case-tac  $z \in \text{set } (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! i))))$ )
apply (case-tac  $i=ia$ ,simp)

```

```

apply (rotate-tac 5)
apply (erule-tac  $x=ia$  in allE,simp)
apply (rotate-tac 22)
apply (erule-tac  $x=i$  in allE,simp)
apply blast
apply (rule-tac  $x=i$  in bexI)
  prefer 2 apply simp
apply (rule-tac  $x=z$  in bexI)
  prefer 2 apply simp
apply (rule conjI)
  apply (case-tac  $z = x,simp$ )
    apply (frule  $\Gamma$ -case-x-is-d,simp)
  apply (rule Otimes-prop4)
  apply (simp,assumption+,force,assumption+)
apply (subgoal-tac
   $\exists w. w \in \text{closureL } q (h(p := \text{None}), k) \wedge$ 
     $w \in \text{recReach } (\text{extend } E1 (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! i)))) \text{ vs}, E2) z (h(p :=$ 
     $\text{None}), k))$ 
  prefer 2 apply blast
apply (subgoal-tac  $z \in \text{dom } (\text{extend } E1 (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! i)))) \text{ vs})$ )
  prefer 2 apply (erule-tac  $x=i$  in allE,simp) + apply blast
apply (subgoal-tac  $z \in \text{dom } E1$ )
  prefer 2 apply (rule extend-prop1,simp,simp,simp)
apply (elim exE,elim conjE)
apply (frule-tac  $y=z$  in recReach-extend-None-subset-recReach,assumption+)
apply (case-tac  $p \neq q$ )
  apply (frule-tac  $h=h$  and  $k=k$  in closureL-p-None-subseteq-closureL)
  apply (subgoal-tac  $w \in \text{closureL } q (h,k)$ )
    apply blast
  apply (rotate-tac 26)
  apply (erule thin-rl)
  apply blast
apply simp
apply (subst (asm) closureL-p-None-p) +
apply simp
apply (subgoal-tac  $q \in \text{closureL } q (h,k)$ )
  apply blast
by (rule closureL-basic)

```

**lemma** *P7-cased-dem1-5*:

```

   $\llbracket \text{length } \text{assert} > 0;$ 
     $\text{length } \text{assert} = \text{length } \text{alts};$ 
     $\text{shareRec } (\text{fst } (\text{assert } ! i)) (\text{snd } (\text{assert } ! i)) (\text{extend } E1 (\text{snd } (\text{extractP } (\text{fst}$ 
     $(\text{alts } ! i)))) \text{ vs}, E2) (h(p:=\text{None}), k) (hh, k);$ 
     $\text{dom } (\text{snd } (\text{assert } ! i)) \subseteq \text{dom } (\text{extend } E1 (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! i)))) \text{ vs};$ 

```

```

     $\forall i < \text{length } \text{assert}. \text{fst } (\text{assert } ! i) \subseteq \text{dom } (\text{snd } (\text{assert } ! i));$ 
     $i < \text{length } \text{assert} \parallel$ 
 $\implies \text{SSet } ((\text{fst } (\text{assert } ! i)) \cap \text{set } (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! i))))$ 
 $(\text{snd } (\text{assert } ! i)) (\text{extend } E1 (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! i)))) \text{ vs, } E2)$ 
 $(h(p := \text{None}), k) \cap$ 
 $\text{RSet } (\text{fst } (\text{assert } ! i)) (\text{snd } (\text{assert } ! i)) (\text{extend } E1 (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! i)))) \text{ vs, } E2) (h(p := \text{None}), k) = \{\}$ 
apply (rule equalityI)
prefer 2 apply simp
apply (rule subsetI)
apply (simp add: SSet-def)
apply (simp add: Let-def)
apply (elim conjE)
apply (elim exE, elim conjE)
apply (simp add: RSet-def)
apply (elim conjE)
apply (elim bexE, elim conjE)
apply (simp add: live-def)
apply (simp add: closureLS-def)
apply (elim bexE)
apply (simp add: shareRec-def)
apply (elim conjE)
apply (rotate-tac 16)
apply (erule thin-rl)
apply (erule-tac x=xa in ballE)
apply (drule mp)
apply (rule-tac x=z in bexI, simp)
apply (subgoal-tac
   $\exists w. w \in \text{closureL } x (h(p := \text{None}), k) \wedge$ 
 $w \in \text{recReach } (\text{extend } E1 (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! i)))) \text{ vs, } E2) z (h(p$ 
 $:= \text{None}), k))$ 
prefer 2 apply blast
apply (elim exE, elim conjE)
apply (subgoal-tac
   $w \in \text{closure } (\text{extend } E1 (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! i)))) \text{ vs, } E2) xa (h(p :=$ 
 $\text{None}), k))$ 
apply blast
apply (rule closure-transit, simp, simp)
apply simp
apply simp
apply (erule-tac x=i in allE, simp)+
by blast

```

**lemma** *P7-cased-dem1-6*:

```

 $\llbracket E1 \ x = \text{Some } (\text{Loc } p);$ 
 $h \ p = \text{Some } (j, C, \text{vs});$ 
 $\Gamma = \text{disjointUnionEnv}$ 

```



```

(nonDisjointUnionEnvList ((map (λ(Li,Γi). restrict-neg-map Γi (set Li ∪ {x})))

  (zip (map (snd o extractP o fst) alts) (map snd assert))))
  (empty(x ↦ d''));
  i < length alts; length assert = length alts;
  shareRec (fst (assert ! i)) (snd (assert ! i))
    (extend E1 (snd (extractP (fst (alts ! i)))) vs, E2) (h(p := None), k)
(hh, k);
  set (snd (extractP (fst (alts ! i)))) ∩ dom E1 = {};
  length (snd (extractP (fst (alts ! i)))) = length vs;
  dom (snd (assert ! i)) ⊆ dom (extend E1 (snd (extractP (fst (alts ! i)))) vs);
  ∀ i < length assert. fst (assert ! i) ⊆ dom (snd (assert ! i)) ]
  ⇒ SSet ((fst (assert ! i)) ∩ (⋃ i < length alts fst (assert ! i) - set (snd
(extractP (fst (alts ! i)))))
    (snd (assert ! i)) (extend E1 (snd (extractP (fst (alts ! i)))) vs, E2)
(h(p := None), k) ∩
  RSet2 ((fst (assert ! i)) ∩ set (snd (extractP (fst (alts ! i)))))
    (fst (assert ! i)) (snd (assert ! i)) (extend E1 (snd (extractP (fst (alts
! i))))) vs, E2) (h(p := None), k) = {}
apply (rule equalityI)
prefer 2 apply simp
apply (rule subsetI)
apply (simp add: SSet-def)
apply (simp add: Let-def)
apply (elim conjE)
apply (elim exE, elim conjE)
apply (simp add: RSet2-def)
apply (elim bexE, elim conjE)
apply (elim bexE, elim conjE)
apply (rename-tac q y ia z)
apply (simp add: live-def)
apply (simp add: closureLS-def)
apply (elim bexE, clarsimp)
apply (simp add: shareRec-def)
apply (elim conjE)
apply (rotate-tac 20)
apply (erule-tac x=y in ballE)
apply (drule mp)
apply (rule-tac x=z in bexI)
apply (rule conjI)
apply simp
apply (simp add: closure-def)
apply (case-tac extend E1 (snd (extractP (fst (alts ! i)))) vs y)
apply simp
apply simp
apply (case-tac a, simp-all, clarsimp)
apply (frule closureL-monotone)
apply blast
by blast

```

**lemma** *P7-CASED*:

```

  [ Γ = disjointUnionEnv
    (nonDisjointUnionEnvList ((map (λ(Li,Γi). restrict-neg-map Γi (set
Li ∪ {x})))
      (zip (map (snd o extractP o fst) alts) (map snd assert))))
    (empty(x ↦ d''));
    set (snd (extractP (fst (alts ! i)))) ∩ dom E1 = {};
  def-nonDisjointUnionEnvList
    (map (λ(Li, Γi). restrict-neg-map Γi (insert x (set Li)))
      (zip (map (snd o extractP o fst) alts) (map snd assert)))));
  def-disjointUnionEnv
    (nonDisjointUnionEnvList ((map (λ(Li,Γi). restrict-neg-map Γi (set
Li ∪ {x})))
      (zip (map (snd o extractP o fst) alts) (map snd assert))))
    [x ↦ d''];
    length (snd (extractP (fst (alts ! i)))) = length vs;
    ∀ i < length assert. fst (assert ! i) ⊆ dom (snd (assert ! i));
    dom (snd (assert ! i)) ⊆ dom (extend E1 (snd (extractP (fst (alts ! i)))) vs);
    shareRec (fst (assert ! i)) (snd (assert ! i)) (extend E1 (snd (extractP (fst
alts ! i)))) vs, E2) (h(p := None), k) (hh, k);
    ∀ z ∈ dom Γ. Γ z ≠ Some s'' ⟶ (∀ i < length alts. z ∉ fst (assert ! i));
    ∀ i < length alts. ∀ j < length alts. i ≠ j ⟶ (fst (assert ! i) ∩ set (snd
(extractP (fst (alts ! j))))) = {};
    E1 x = Some (Loc p); h p = Some (j, C, vs); i < length alts;
    0 < length (map snd assert); length assert = length alts;
    ∀ i < length alts. ∀ j. ∀ x ∈ set (snd (extractP (fst (alts ! i)))). snd (assert ! i)
x = Some d'' ⟶ j ∈ RecPos Ci;
    shareRec (insert x (⋃i < length alts fst (assert ! i) - set (snd (extractP (fst
alts ! i)))))
    Γ (E1, E2) (h, k) (hh,k);
    SSet ((⋃i < length alts fst (assert ! i) - set (snd (extractP (fst (alts ! i)))))
  ∪ {x})
    Γ (E1, E2) (h, k) ∩
    RSet ((⋃i < length alts fst (assert ! i) - set (snd (extractP (fst (alts ! i)))))
  ∪ {x})
    Γ (E1, E2) (h, k) =
    {} ]
    ⟹ SSet (fst (assert ! i)) (snd (assert ! i)) (extend E1 (snd (extractP (fst
alts ! i)))) vs, E2) (h(p := None), k) ∩
    RSet (fst (assert ! i)) (snd (assert ! i)) (extend E1 (snd (extractP (fst
alts ! i)))) vs, E2) (h(p := None), k) =
    {}
  apply (rule P7-cased-dem1)
  apply (rule P7-cased-dem1-1,assumption+,simp)
  apply (rule P7-cased-dem1-2,assumption+,simp)

```

```

apply (rule P7-cased-dem1-3,simp,assumption+,simp)
apply (rule P7-cased-dem1-4,simp,assumption+,simp)
apply (rule P7-cased-dem1-5,simp,assumption+,simp)
apply (rule P7-cased-dem1-6,simp,assumption+)
done

```

```

lemma nth-map-of-xs-atom2val:
  [| length xs = length as;
    distinct xs |]
   $\implies \forall i < \text{length } xs.$ 
    map-of (zip xs (map (atom2val E1) as)) (xs!i) =
      Some (atom2val E1 (as!i))
apply clarsimp
apply (induct xs as rule: list-induct2',simp-all)
by (case-tac i,simp,clarsimp)

```

```

lemma closureL-k-equals-closureL-Suc-k:
  closureL p (h, Suc k) = closureL p (h,k)
apply (rule equalityI)
apply (rule subsetI)
apply (erule-tac closureL.induct)
apply (rule closureL-basic)
apply (rule closureL-step,simp)
apply (simp add: descendants-def)
apply (rule subsetI)
apply (erule-tac closureL.induct)
apply (rule closureL-basic)
apply (rule closureL-step,simp)
by (simp add: descendants-def)

```

```

lemma recReachL-k-equals-recReachL-Suc-k:
  recReachL x (h, k) = recReachL x (h, Suc k)
apply (rule equalityI)
apply (rule subsetI)
apply (erule-tac recReachL.induct)
apply (rule recReachL-basic)
apply (rule recReachL-step,simp)
apply (simp add: recDescendants-def)
apply (rule subsetI)
apply (erule-tac recReachL.induct)
apply (rule recReachL-basic)

```

**apply** (rule *recReachL-step,simp*)  
**by** (simp add: *recDescendants-def*)

**lemma** *closure-APP-equals-closure-ef*:

[[ *length xs = length as; distinct xs;*  
*distinct xs;*  
 $(\forall i < \text{length } as. \forall x a. as!i = \text{VarE } x a \longrightarrow x \in \text{dom } E1);$   
 $i < \text{length } as; as ! i = \text{VarE } xa a$  ]]  
 $\implies \text{closure } (E1, E2) xa (h, k) =$   
 $\text{closure } (\text{map-of } (\text{zip } xs (\text{map } (\text{atom2val } E1) as)), \text{map-of } (\text{zip } rs (\text{map } (the$   
 $\circ E2) rs'))(self \mapsto \text{Suc } k)) (xs ! i) (h, \text{Suc } k)$   
**apply** (frule-tac *?E1.0=E1 in nth-map-of-xs-atom2val,assumption+*)  
**apply** (erule-tac *x=i in allE,simp*)  
**apply** (erule-tac *x=i in allE,simp*)  
**apply** (rule *equalityI*)

**apply** (rule *subsetI*)  
**apply** (simp add: *closure-def*)  
**apply** (case-tac *E1 xa,simp-all*)  
**apply** (case-tac *aa, simp-all*)  
**apply** (subst *closureL-k-equals-closureL-Suc-k,assumption*)

**apply** (rule *subsetI*)  
**apply** (simp add: *closure-def*)  
**apply** (case-tac *E1 xa,simp-all*)  
**apply** (simp add: *dom-def*)  
**apply** (case-tac *aa, simp-all*)  
**apply** (insert *closureL-k-equals-closureL-Suc-k*)  
**by** simp

**lemma** *recReach-APP-equals-recReach-ef*:

[[  $i < \text{length } as; as ! i = \text{VarE } z a;$   
 $(\forall i < \text{length } as. \forall x a. as!i = \text{VarE } x a \longrightarrow x \in \text{dom } E1);$   
 $\text{length } xs = \text{length } as; \text{distinct } xs$  ]]  
 $\implies \text{recReach } (E1, E2) z (h, k) =$   
 $\text{recReach } (\text{map-of } (\text{zip } xs (\text{map } (\text{atom2val } E1) as)), \text{map-of } (\text{zip } rs (\text{map}$   
 $(the \circ E2) rs'))(self \mapsto \text{Suc } k)) (xs ! i) (h, \text{Suc } k)$   
**apply** (rule *equalityI*)

**apply** (rule *subsetI*)  
**apply** (frule-tac *?E1.0=E1 in nth-map-of-xs-atom2val, assumption+*)  
**apply** (rotate-tac 6)  
**apply** (erule-tac *x=i in allE,simp*)  
**apply** (simp add: *recReach-def*)  
**apply** (case-tac *E1 z,simp-all*)  
**apply** (case-tac *aa, simp-all*)

**apply** (subgoal-tac  
 recReachL nat (h, k) = recReachL nat (h, Suc k),simp)  
**apply** (rule recReachL-k-equals-recReachL-Suc-k)

**apply** (rule subsetI)  
**apply** (frule-tac ?E1.0=E1 in nth-map-of-xs-atom2val, assumption+)  
**apply** (rotate-tac 6)  
**apply** (erule-tac x=i in allE,simp)  
**apply** (simp add: recReach-def)  
**apply** (case-tac E1 z,simp-all)  
**apply** force  
**apply** (case-tac aa, simp-all)  
**apply** (subgoal-tac  
 recReachL nat (h, k) = recReachL nat (h, Suc k),simp)  
**by** (rule recReachL-k-equals-recReachL-Suc-k)

**lemma** closureL-recReach-APP-equals-closureL-recReach-ef:

$\llbracket z \in fvs' as;$   
 $i < length\ as; as\ !\ i = VarE\ z\ a;$   
 $(\forall\ i < length\ as. \forall\ x\ a. as\ !\ i = VarE\ x\ a \longrightarrow x \in dom\ E1);$   
 $length\ xs = length\ as; distinct\ xs \rrbracket$   
 $\implies closureL\ x\ (h, k) \cap recReach\ (E1, E2)\ z\ (h, k) =$   
 $closureL\ x\ (h, Suc\ k) \cap$   
 $recReach\ (map-of\ (zip\ xs\ (map\ (atom2val\ E1)\ as)), map-of\ (zip\ rs\ (map\ (the$   
 $\circ E2)\ rs'))(self \mapsto Suc\ k))(xs\ !\ i)\ (h, Suc\ k)$   
**apply** (subst closureL-k-equals-closureL-Suc-k)  
**apply** (frule recReach-APP-equals-recReach-ef,assumption+)  
**by** blast

**lemma** nth-set-distinct:

$\llbracket x \in set\ xs; distinct\ xs \rrbracket$   
 $\implies \exists\ i < length\ xs. xs\ !\ i = x$   
**by** (induct xs,simp,force)

**lemma** nth-map-add-map-of-y:

$\llbracket i < length\ xs; ms\ !\ i = y;$   
 $length\ xs = length\ ms; distinct\ xs \rrbracket$   
 $\implies (map-of\ (zip\ xs\ ms))\ (xs\ !\ i) = Some\ y$   
**by** (simp, subst set-zip,force)

**lemma** nth-map-add-map-of:

$\llbracket i < length\ xs; length\ xs = length\ ms; distinct\ xs \rrbracket$   
 $\implies (map-of\ (zip\ xs\ ms))\ (xs\ !\ i) = Some\ (ms\ !\ i)$   
**apply** (subgoal-tac  
 set (zip xs ms) =

$\{(xs!i, ms!i) \mid i. i < \min (\text{length } xs) (\text{length } ms)\}$   
**apply** *force*  
**by** (*rule set-zip*)

**lemma** *map-add-map-of*:  
 $\llbracket x \in \text{set } xs; \text{dom } E1 \cap \text{set } xs = \{\}; \text{length } xs = \text{length } ys \rrbracket$   
 $\implies (E1 ++ \text{map-of } (\text{zip } xs \text{ } ys)) \ x = \text{map-of } (\text{zip } xs \text{ } ys) \ x$   
**apply** (*subgoal-tac E1 x = None*)  
**apply** (*simp only: map-add-def*)  
**apply** (*case-tac map-of (zip xs ys) x, simp-all*)  
**by** *blast*

**lemma** *var-in-fvs*:  
 $\llbracket i < \text{length } as; as ! i = \text{VarE } x \ a \rrbracket$   
 $\implies x \in \text{fvs}' \ as$   
**apply** (*induct as arbitrary: i, simp-all*)  
**apply** *clarsimp*  
**apply** (*case-tac i, simp-all*)  
**apply** (*case-tac aa, simp-all*)  
**by** *auto*

**lemma** *atom-fvs-VarE*:  
 $\llbracket (\forall a \in \text{set } as. \text{atom } a); xa \in \text{fvs}' \ as \rrbracket$   
 $\implies (\exists i < \text{length } as. \exists a. as!i = \text{VarE } xa \ a)$   
**apply** (*induct as, simp-all*)  
**apply** (*case-tac a, simp-all*)  
**by** *force*

**declare** *atom.simps* [*simp del*]

**lemma** *live-APP-equals-live-ef*:  
 $\llbracket (\forall a \in \text{set } as. \text{atom } a); \text{length } xs = \text{length } as;$   
 $\text{length } xs = \text{length } ms;$   
 $\text{dom } E1 \cap \text{set } xs = \{\}; \text{distinct } xs;$   
 $(\forall i < \text{length } as. \forall x \ a. \exists y. as!i = \text{VarE } x \ a \longrightarrow x \in \text{dom } E1) \rrbracket$   
 $\implies \text{live } (E1, E2) (\text{fvs}' \ as) (h, k) =$   
 $\text{live } (\text{map-of } (\text{zip } xs \ (\text{map } (\text{atom2val } E1) \ as)), \text{map-of } (\text{zip } rs \ (\text{map } (\text{the } \circ$   
 $E2) \ rs')) (\text{self} \mapsto \text{Suc } k)) (\text{set } xs) (h, \text{Suc } k)$   
**apply** (*rule equalityI*)

**apply** (*rule subsetI*)  
**apply** (*simp add: live-def*)  
**apply** (*simp add: closureLS-def*)  
**apply** (*erule bexE*)  
**apply** (*frule atom-fvs-VarE, assumption+*)  
**apply** (*elim exE*)  
**apply** (*elim conjE, elim exE*)

```

apply (rule-tac  $x=xs!i$  in  $be xI$ )
  prefer 2 apply simp
apply (frule-tac  $?E1.0=E1$  in closure-APP-equals-closure-ef)
apply (assumption+,force,assumption+)
apply blast

apply (rule subsetI)
apply (simp add: live-def)
apply (simp add: closureLS-def)
apply (elim  $be xE$ )
apply (frule nth-set-distinct,assumption+)
apply (elim  $exE$ ,elim  $conjE$ )
apply (frule-tac  $?E1.0=E1$  in nth-map-of-xs-atom2val, assumption+)
apply (rotate-tac 10)
apply (erule-tac  $x=i$  in  $allE$ ,simp)
apply (erule-tac  $x=as!i$  in  $ballE$ )
  prefer 2 apply simp
apply (subgoal-tac
  length xs = length (map (atom2val E1) as))
apply (frule-tac  $?E1.0=E1$  in map-add-map-of,assumption+)
  prefer 2 apply simp
apply (simp add: atom.simps)
apply (case-tac (as ! i),simp-all)
  apply (simp add: closure-def)
apply (rule-tac  $x=list$  in  $be xI$ )
apply (case-tac E1 list,simp-all) apply force
apply (case-tac aa, simp-all)
apply (subst (asm) closureL-k-equals-closureL-Suc-k)
apply blast
by (frule var-in-fvs,assumption+)

```

```

lemma map-le-nonDisjointUnionSafeEnvList:
   $\llbracket \text{nonDisjointUnionSafeEnvList } \Gamma s \ x = \text{Some } y;$ 
     $(\text{nonDisjointUnionSafeEnvList } \Gamma s) \subseteq_m \Gamma'$ 
   $\implies \Gamma' \ x = \text{Some } y$ 
apply (simp add: map-le-def)
by force

```

```

declare nonDisjointUnionSafeEnvList.simps [simp del]

```

```

lemma nonDisjointUnionSafeEnv-assoc:
  nonDisjointUnionSafeEnv (nonDisjointUnionSafeEnv G1 G2) G3 =
  nonDisjointUnionSafeEnv G1 (nonDisjointUnionSafeEnv G2 G3)
apply (simp add: nonDisjointUnionSafeEnv-def add: unionEnv-def)
apply (rule ext, auto)
  apply (split split-if-asm, simp, simp)
  apply (split split-if-asm, simp,simp)
by (split split-if-asm, simp, simp add: dom-def)

```

```

lemma foldl-nonDisjointUnionSafeEnv-prop:
  foldl nonDisjointUnionSafeEnv ( $G' \oplus G$ ) Gs =  $G' \oplus \text{foldl } op \oplus G \text{ Gs}$ 
apply (induct Gs arbitrary: G)
apply simp
by (simp-all add: nonDisjointUnionSafeEnv-assoc)

lemma nonDisjointUnionSafeEnv-empty:
  nonDisjointUnionSafeEnv empty  $x = x$ 
apply (simp add: nonDisjointUnionSafeEnv-def)
by (simp add: unionEnv-def)

lemma nonDisjointUnionSafeEnv-commutative:
  def-nonDisjointUnionSafeEnv G G'  $\implies (G \oplus G') = (G' \oplus G)$ 
apply (simp add: nonDisjointUnionSafeEnv-def add: unionEnv-def)
apply (rule ext)
apply (simp add: def-nonDisjointUnionSafeEnv-def)
apply (simp add: safe-def)
by clarsimp

declare dom-fun-upd [simp del]

lemma nth-nonDisjointUnionSafeEnvList:
   $\llbracket \text{length } xs = \text{length } ms; \text{def-nonDisjointUnionSafeEnvList } (\text{maps-of } (\text{zip } xs \text{ } ms)) \rrbracket$ 
 $\implies (\forall i < \text{length } xs . \text{nonDisjointUnionSafeEnvList } (\text{maps-of } (\text{zip } xs \text{ } ms))$ 
 $(xs!i) = \text{Some } (ms!i))$ 
apply (induct xs ms rule: list-induct2', simp-all)
apply clarsimp
apply (case-tac i)
apply simp
apply (simp add: nonDisjointUnionSafeEnvList.simps)
apply (subgoal-tac empty  $\oplus [x \mapsto y] = [x \mapsto y] \oplus \text{empty}, \text{simp}$ )
apply (subgoal-tac foldl op  $\oplus ([x \mapsto y] \oplus \text{empty}) (\text{maps-of } (\text{zip } xs \text{ } ys)) =$ 
 $[x \mapsto y] \oplus \text{foldl } op \oplus \text{empty } (\text{maps-of } (\text{zip } xs \text{ } ys)), \text{simp}$ )
apply (simp add: nonDisjointUnionSafeEnv-def)
apply (simp add: unionEnv-def)
apply (simp add: dom-def)
apply (rule foldl-nonDisjointUnionSafeEnv-prop)
apply (subst nonDisjointUnionSafeEnv-empty)
apply (subst nonDisjointUnionSafeEnv-commutative)
apply (simp only: def-nonDisjointUnionSafeEnv-def)
apply (simp only: safe-def)
apply force
apply (subst nonDisjointUnionSafeEnv-empty)
apply simp
apply clarsimp
apply (simp add: nonDisjointUnionSafeEnvList.simps)

```



```

apply (subgoal-tac empty  $\oplus$   $[x \mapsto y] = [x \mapsto y] \oplus \text{empty}$ ,simp)
apply (subgoal-tac foldl op  $\oplus$  ( $[x \mapsto y] \oplus \text{empty}$ ) (maps-of (zip xs ys)) =
       $[x \mapsto y] \oplus \text{foldl op} \oplus \text{empty}$  (maps-of (zip xs ys)),simp)
apply (simp add: Let-def)
apply (erule-tac x=nat in allE,simp)
apply (simp add: nonDisjointUnionSafeEnv-def)
apply (simp add: unionEnv-def)
apply (rule conjI)
apply (rule impI)+
apply (elim conjE)
apply (simp add: def-nonDisjointUnionSafeEnv-def)
apply (erule-tac x=x in ballE)
apply (simp add: safe-def)
apply (simp add: dom-def)
apply clarsimp
apply (rule foldl-nonDisjointUnionSafeEnv-prop)
apply (rule nonDisjointUnionSafeEnv-commutative)
by (simp add: def-nonDisjointUnionSafeEnv-def)

```

```

lemma union-dom-nonDisjointUnionSafeEnv:
  dom (nonDisjointUnionSafeEnv A B) = dom A  $\cup$  dom B
apply (simp add: nonDisjointUnionSafeEnv-def add: unionEnv-def,auto)
by (split split-if-asm,simp-all)

```

```

lemma dom-foldl-nonDisjointUnionSafeEnv-monotone:
  dom (foldl nonDisjointUnionSafeEnv (empty  $\oplus$  x) xs) =
    dom x  $\cup$  dom (foldl op  $\oplus$  empty xs)
apply (subgoal-tac empty  $\oplus$  x = x  $\oplus$  empty,simp)
apply (subgoal-tac foldl op  $\oplus$  (x  $\oplus$  empty) xs =
      x  $\oplus$  foldl op  $\oplus$  empty xs,simp)
apply (rule union-dom-nonDisjointUnionSafeEnv)
apply (rule foldl-nonDisjointUnionSafeEnv-prop)
apply (rule nonDisjointUnionSafeEnv-commutative)
by (simp add: def-nonDisjointUnionSafeEnv-def)

```

```

lemma dom-nonDisjointUnionSafeEnvList-fvs:
   $\llbracket \forall a \in \text{set } xs. \text{atom } a; \text{length } xs = \text{length } ys \rrbracket$ 
   $\implies \text{fvs}' xs \subseteq \text{dom} (\text{nonDisjointUnionSafeEnvList} (\text{maps-of} (\text{zip} (\text{map atom2var}$ 
  xs) ys)))
apply (induct xs ys rule: list-induct2',simp-all)
apply (simp add: nonDisjointUnionSafeEnvList.simps)
apply (subst dom-foldl-nonDisjointUnionSafeEnv-monotone)
apply (rule conjI)
apply (simp add: atom.simps)
apply (case-tac x, simp-all)
apply (simp add: dom-def)
apply (subst dom-foldl-nonDisjointUnionSafeEnv-monotone)
by blast

```

**lemma** *nonDisjointUnionSafeEnvList-prop1*:  
 $\llbracket \text{nonDisjointUnionSafeEnvList } (\text{maps-of } (\text{zip } (\text{map } \text{atom2var } \text{as}) \text{ms})) \subseteq_m \Gamma;$   
 $\text{xa} \in \text{fvs}' \text{ as}; \Gamma \text{ xa} = \text{Some } y;$   
 $\text{def-nonDisjointUnionSafeEnvList } (\text{maps-of } (\text{zip } (\text{map } \text{atom2var } \text{as}) \text{ms}));$   
 $(\forall a \in \text{set as. atom } a); \text{length as} = \text{length ms} \rrbracket$   
 $\implies \exists i < \text{length as. } \exists a. \text{as}!i = \text{VarE xa } a \wedge \text{ms}!i = y$   
**apply** (*frule atom-fvs-VarE,assumption+*)  
**apply** (*elim exE, elim conjE, elim exE*)  
**apply** (*rule-tac x=i in exI,simp*)  
**apply** (*simp add: map-le-def*)  
**apply** (*erule-tac x=xa in ballE,simp*)  
**apply** (*subgoal-tac length (map atom2var as) = length ms*)  
**prefer** 2 **apply** *simp*  
**apply** (*frule nth-nonDisjointUnionSafeEnvList,assumption+*)  
**apply** (*erule-tac x=i in allE,simp*)  
**apply** (*frule dom-nonDisjointUnionSafeEnvList-fvs,assumption+*)  
**by** *blast*

**lemma** *xs-ms-in-set*:  
 $\llbracket \text{length ms} = \text{length as}; \text{length xs} = \text{length as}; \text{distinct xs};$   
 $i < \text{length as}; \text{as} ! i = \text{VarE xa } a; \text{ms} ! i = m \rrbracket$   
 $\implies (\text{xs} ! i, m) \in \text{set } (\text{zip xs ms})$   
**apply** *clarsimp*  
**apply** (*subgoal-tac*  
 $\text{set } (\text{zip xs ms}) =$   
 $\{(xs!i, ms!i) \mid i. i < \min (\text{length xs}) (\text{length ms})\}$ )  
**apply** *force*  
**by** (*rule set-zip*)

**lemma** *xs-ms-in-set-ms*:  
 $\llbracket \text{length ms} = \text{length as}; \text{length xs} = \text{length as}; \text{distinct xs};$   
 $i < \text{length xs}; (\text{xs}!i, m) \in \text{set } (\text{zip xs ms}) \rrbracket$   
 $\implies \text{ms}!i = m$   
**apply** (*simp add: set-conv-nth cong: rev-conj-cong*)  
**apply** (*elim exE, elim conjE*)  
**apply** (*rename-tac j*)  
**apply** (*frule-tac j=j in nth-eq-iff-index-eq*)  
**by** (*simp,simp,simp*)

**lemma** *SSet-APP-equals-SSet-ef*:  
 $\llbracket \text{def-nonDisjointUnionSafeEnvList } (\text{maps-of } (\text{zip } (\text{map } \text{atom2var } \text{as}) \text{ms}));$   
 $\text{nonDisjointUnionSafeEnvList } (\text{maps-of } (\text{zip } (\text{map } \text{atom2var } \text{as}) \text{ms})) \subseteq_m \Gamma;$   
 $\text{length xs} = \text{length as};$   
 $\text{length xs} = \text{length ms}; \text{distinct xs}; (\forall a \in \text{set as. atom } a);$   
 $\text{dom } E1 \cap \text{set xs} = \{\}; \text{fvs}' \text{ as} \subseteq \text{dom } \Gamma; \text{dom } \Gamma \subseteq \text{dom } E1;$   
 $\rrbracket$

```

    (∀ i < length as. ∀ x a. as!i = VarE x a ⟶ x ∈ dom E1)]
  ⟹ SSet (fvs' as) Γ (E1, E2) (h, k) =
    SSet (set xs) (map-of (zip xs ms))
      (map-of (zip xs (map (atom2val E1) as)), map-of (zip rs (map (the ∘
E2) rs'))(self ↦ Suc k)) (h, Suc k)
  apply (rule equalityI)

  apply (rule subsetI)
  apply (simp add: SSet-def)
  apply (simp add: Let-def)
  apply (elim exE, elim conjE)
  apply (frule nonDisjointUnionSafeEnvList-prop1, assumption+, simp)
  apply (elim exE, elim conjE)+
  apply (rule-tac x=xs!i in exI)
  apply (rule conjI)
  apply simp
  apply (rule conjI)

  apply (rule xs-ms-in-set, assumption+)

  apply (subgoal-tac
    closure (E1, E2) xa (h, k) =
      closure (map-of (zip xs (map (atom2val E1) as)), map-of (zip rs (map (the ∘
E2) rs'))(self ↦ Suc k)) (xs ! i) (h, Suc k))
    apply blast
  apply (rule closure-APP-equals-closure-ef)
  apply (assumption+, simp, assumption+)

  apply (rule subsetI)
  apply (simp add: SSet-def)
  apply (simp add: Let-def)
  apply (elim exE, elim conjE)
  apply (frule-tac ?E1.0=E1 in nth-map-of-xs-atom2val, assumption+)
  apply (frule nth-set-distinct, assumption+)
  apply (elim exE)
  apply (rotate-tac 13)
  apply (erule-tac x=i in allE, simp)
  apply (elim conjE)
  apply (erule-tac x=as!i in ballE)
  prefer 2 apply simp
  apply (simp add: atom.simps)
  apply (case-tac (as ! i), simp-all)

  apply (rule-tac x=list in exI)
  apply (rule conjI)
  apply (rule var-in-fvs, assumption+)
  apply (rule conjI)

```

```

apply (subgoal-tac length (map atom2val as) = length ms)
prefer 2 apply simp
apply (frule nth-nonDisjointUnionSafeEnvList,assumption+)
apply (rotate-tac 17)
apply (erule-tac x=i in allE,simp)
apply (frule map-le-nonDisjointUnionSafeEnvList,assumption+)
apply (subgoal-tac i < length xs)
prefer 2 apply simp
apply (frule nth-map-of-xs-atom2val,assumption+)
apply (rotate-tac 19)
apply (erule-tac x=i in allE,simp)
apply (rule xs-ms-in-set-ms,assumption+,simp,simp)

apply clarsimp
apply (subgoal-tac
  closure (E1, E2) list (h, k) =
    closure (map-of (zip xs (map (atom2val E1) as)), map-of (zip rs (map (the ∘
E2) rs'))(self ↦ Suc k)) (xs ! i) (h, Suc k))
apply blast
apply (rule closure-APP-equals-closure-ef)
by (assumption+,simp,assumption+)

lemma RSet-APP-equals-RSet-ef:
  [| def-nonDisjointUnionSafeEnvList (maps-of (zip (map atom2val as) ms));
    nonDisjointUnionSafeEnvList (maps-of (zip (map atom2val as) ms)) ⊆m Γ;
    (∀ a ∈ set as. atom a); length xs = length as;
    (∀ i < length as. ∀ x a. as!i = VarE x a ⟶ x ∈ dom E1);
    length xs = length ms;
    dom E1 ∩ set xs = {}; distinct xs |]
  ⟹ RSet (fvs' as) Γ (E1, E2) (h, k) =
    RSet (set xs) (map-of (zip xs ms))
      (map-of (zip xs (map (atom2val E1) as)), map-of (zip rs (map (the ∘
E2) rs'))(self ↦ Suc k)) (h, Suc k)
apply (rule equalityI)

apply (rule subsetI)
apply (simp add: RSet-def)
apply (elim conjE,elim bexE,elim conjE)
apply (rule conjI)

apply (frule live-APP-equals-live-ef)
apply assumption+ apply force apply blast

apply (frule nonDisjointUnionSafeEnvList-prop1)
apply (assumption+,simp)
apply (elim exE, elim conjE)+
apply (rule-tac x=xs!i in bexI)
prefer 2 apply simp
apply (rule conjI)

```

```

apply (rule xs-ms-in-set,assumption+)

apply (subgoal-tac
  closureL x (h, k)  $\cap$  recReach (E1, E2) z (h, k) =
  closureL x (h, Suc k)  $\cap$  recReach (map-of (zip xs (map (atom2val E1) as)),
  map-of (zip rs (map (the  $\circ$  E2) rs'))(self  $\mapsto$  Suc k))
  (xs ! i) (h, Suc k))
  apply blast
apply (rule closureL-recReach-APP-equals-closureL-recReach-ef)
apply (assumption+,force,simp,assumption+)

apply (rule subsetI)
apply (simp add: RSet-def)
apply (elim conjE,elim bexE,elim conjE)
apply (rule conjI)

apply (frule live-APP-equals-live-ef)
apply (assumption+,force,blast)

apply (frule-tac ?E1.0=E1 in nth-map-of-xs-atom2val,assumption+)
apply (frule nth-set-distinct,assumption+)
apply (elim exE)
apply (rotate-tac 12)
apply (erule-tac x=i in allE,simp)
apply (elim conjE)
apply (erule-tac x=as!i in ballE)
  prefer 2 apply simp
apply (simp add: atom.simps)
apply (case-tac (as ! i),simp-all)
apply (frule var-in-fvs) apply assumption+
apply (rule-tac x=list in bexI)
  prefer 2 apply simp
apply (rule conjI)

apply (subgoal-tac i<length xs)
  prefer 2 apply simp
apply (frule-tac xs=xs and ms=ms in nth-map-add-map-of)
apply (simp,simp)
apply (subgoal-tac length (map atom2var as) = length ms)
  prefer 2 apply simp
apply (frule nth-nonDisjointUnionSafeEnvList,assumption+)
apply (rotate-tac 19)
apply (erule-tac x=i in allE)
apply clarsimp
apply (rule map-le-nonDisjointUnionSafeEnvList,assumption+)

apply (subgoal-tac closureL x (h, k)  $\cap$  recReach (E1, E2) list (h, k) =

```

```

      closureL x (h, Suc k)  $\cap$  recReach (map-of (zip xs (map (atom2val E1) as)),
map-of (zip rs (map (the  $\circ$  E2) rs'))(self  $\mapsto$  Suc k))
      (xs ! i) (h, Suc k))

apply blast
apply (rule closureL-recReach-APP-equals-closureL-recReach-ef)
by (assumption+,force,simp,assumption+)

```

```

lemma fvs-as-in-dom-E1:
   $\llbracket \text{fvs}' \text{ as} \subseteq \text{dom } \Gamma; \text{dom } \Gamma \subseteq \text{dom } E1 \rrbracket$ 
 $\implies (\forall i < \text{length as}. \forall x a. \text{as}!i = \text{VarE } x \ a \longrightarrow x \in \text{dom } E1)$ 
apply (case-tac as = [],simp-all)
apply (induct as,simp-all)
apply (rule allI,rule impI)
apply (case-tac as = [],simp-all)
apply (case-tac a,simp-all,blast)
apply (case-tac i,simp-all)
apply (erule-tac x=0 in allE,simp)
apply (case-tac a,simp-all)
by blast

```

```

lemma P7-APP-ef:
   $\llbracket \text{length xs} = \text{length as}; \text{distinct xs}; \text{length xs} = \text{length ms};$ 
 $\text{nonDisjointUnionSafeEnvList (maps-of (zip (map atom2var as) ms))} \subseteq_m \Gamma;$ 
 $\text{def-nonDisjointUnionSafeEnvList (maps-of (zip (map atom2var as) ms));}$ 
 $\forall a \in \text{set as}. \text{atom } a;$ 
 $\text{dom } E1 \cap \text{set xs} = \{\}; \text{fvs}' \text{ as} \subseteq \text{dom } \Gamma; \text{dom } \Gamma \subseteq \text{dom } E1;$ 
 $\text{SSet (fvs}' \text{ as) } \Gamma (E1, E2) (h, k) \cap \text{RSet (fvs}' \text{ as) } \Gamma (E1, E2) (h, k) = \{\} \rrbracket$ 
 $\implies \text{SSet (set xs) (map-of (zip xs ms)) (map-of (zip xs (map (atom2val E1)$ 
 $\text{as)), map-of (zip rs (map (the } \circ \text{ E2) rs'))(self } \mapsto \text{Suc k))}$ 
 $(h, \text{Suc } k) \cap$ 
 $\text{RSet (set xs) (map-of (zip xs ms)) (map-of (zip xs (map (atom2val E1) as)),}$ 
 $\text{map-of (zip rs (map (the } \circ \text{ E2) rs'))(self } \mapsto \text{Suc k))}$ 
 $(h, \text{Suc } k) = \{\}$ 
apply (frule fvs-as-in-dom-E1,assumption+)
apply (frule SSet-APP-equals-SSet-ef
  [where h=h and k=k],assumption+)
apply (frule RSet-APP-equals-RSet-ef
  [where h=h and k=k],assumption+)
by blast

```

**end**

## 19 Derived Assertions. P8. closed E L h

**theory** SafeDAss-P8 **imports** SafeDAssBasic

**begin**

Lemmas for LET

**lemma** *P8-LET-e1*:

$closed\ (E1, E2)\ (L1 \cup (L2 - \{x1\}))\ (h, k)$

$\implies closed\ (E1, E2)\ L1\ (h, k)$

**by** (*simp add: closed-def add: live-def add: closureLS-def*)

**lemma** *P8-dem2*:

$\llbracket def\text{-}pp\ \Gamma 1\ \Gamma 2\ L2; x \in dom\ \Gamma 1; \Gamma 1\ x \neq Some\ s' \rrbracket$

$\implies x \notin L2$

**apply** (*simp add: def-pp-def add: unsafe-def*)

**apply** (*erule conjE*)

**apply** (*erule-tac x=x in allE, simp*)**+**

**apply** (*elim conjE, auto*)

**by** (*case-tac y, simp-all*)

**lemma** *P8-dem3*:

$\llbracket closed\text{-}f\ v1\ (h', k'); y \in closure\ (E1(x1 \mapsto v1), E2)\ x1\ (h', k') \rrbracket$

$\implies y \in domHeap\ (h', k')$

**apply** (*simp add: closure-def add: closed-f-def*)

**apply** (*case-tac v1, simp-all*)

**by** *blast*

**lemma** *P8-dem4*:

$\llbracket (\bigcup_{x \in L2 - \{x1\}}. closure\ (E1, E2)\ x\ (h, k)) \subseteq dom\ h; x \neq x1; x \in L2; p \in closure\ (E1, E2)\ x\ (h, k) \rrbracket$

$\implies p \in dom\ h$

**by** *auto*

**lemma** *P8-LET-e2*:

$\llbracket def\text{-}pp\ \Gamma 1\ \Gamma 2\ L2;$

$L2 \subseteq dom\ (disjointUnionEnv\ \Gamma 2\ (empty(x1 \mapsto m)));$

$dom\ (\Gamma 2 + (empty(x1 \mapsto m))) \subseteq dom\ (E1(x1 \mapsto v1));$

$shareRec\ L1\ \Gamma 1\ (E1, E2)\ (h, k)\ (h', k);$

$closed\text{-}f\ v1\ (h', k);$

$closed\ (E1, E2)\ (L1 \cup (L2 - \{x1\}))\ (h, k) \rrbracket$

$\implies closed\ (E1(x1 \mapsto v1), E2)\ L2\ (h', k)$

**apply** (*simp add: closed-def add: live-def add: closureLS-def add: domHeap-def*)

**apply** (*erule conjE*)

**apply** *safe apply* (*rename-tac y x*)

**apply** (*case-tac x ≠ x1*)

**apply** (*subgoal-tac x ∈ dom E1, simp*)

**prefer 2 apply** *blast*

**apply** (*case-tac ¬ (identityClosure (E1, E2) x (h, k) (h', k))*)

**apply** (*simp add: shareRec-def*)

**apply** (*elim conjE*)

**apply** (*erule-tac x=x in ballE*)**+**

```

prefer 2 apply simp
prefer 2 apply simp
apply simp
apply (elim conjE)
apply (subgoal-tac  $x \notin L2, \text{simp}$ )
apply (rule P8-dem2, assumption+)
apply (simp add: identityClosure-def)
apply (erule conjE)
apply (erule-tac  $x=y$  in ballE)
apply (subgoal-tac closure ( $E1(x1 \mapsto v1), E2$ )  $x$  ( $h', k$ ) = closure ( $E1, E2$ )  $x$ 
( $h', k$ ), simp)
prefer 2 apply (simp add: closure-def)
apply (frule P8-dem4) apply assumption apply assumption+ apply simp
apply (simp add: dom-def)
apply (subgoal-tac closure ( $E1(x1 \mapsto v1), E2$ )  $x$  ( $h', k$ ) = closure ( $E1, E2$ )  $x$ 
( $h', k$ ), simp)
apply (simp add: closure-def)
apply simp
apply (subgoal-tac  $y \in \text{domHeap } (h', k)$ )
prefer 2 apply (rule P8-dem3, assumption+)
by (simp add: domHeap-def add: dom-def)

```

**lemma** *P8-LET*:

```

 $\llbracket$  def-pp  $\Gamma 1 \ \Gamma 2 \ L2$ ;
 $L2 \subseteq \text{dom } (\text{disjointUnionEnv } \Gamma 2 \ (\text{empty}(x1 \mapsto m)))$ ;
 $\text{dom } (\Gamma 2 + (\text{empty}(x1 \mapsto m))) \subseteq \text{dom } (E1(x1 \mapsto v1))$ ;
 $\text{shareRec } L1 \ \Gamma 1 \ (E1, E2) \ (h, k) \ (h', k)$ ;
 $\text{closed-f } v1 \ (h', k)$ ;
 $\text{closed } (E1, E2) \ (L1 \cup (L2 - \{x1\})) \ (h, k) \rrbracket$ 
 $\implies \text{closed } (E1, E2) \ L1 \ (h, k) \wedge \text{closed } (E1(x1 \mapsto v1), E2) \ L2 \ (h', k)$ 
apply (rule conjI)
apply (erule P8-LET-e1)
by (erule P8-LET-e2)

```

Lemmas for CASE

**lemma** *vs-defined*:

```

 $\llbracket$  set  $xs \cap \text{dom } E1 = \{\}$ ;
 $\text{length } xs = \text{length } vs$ ;
 $y \in \text{set } xs$ ;
 $\text{extend } E1 \ xs \ vs \ y = \text{Some } (\text{Loc } q) \rrbracket$ 
 $\implies \exists j < \text{length } vs. \text{vs}!j = \text{Loc } q$ 
apply (simp add: extend-def)
apply (induct xs vs rule: list-induct2', simp-all)
by (split split-if-asm, force, force)

```

**lemma** *closure-Loc-subseteq-closure V-Loc*:

```

 $\llbracket$   $vs ! i = \text{Loc } q$ ;

```



$i < \text{length } vs$   $\mathbb{I}$   
 $\implies \text{closureL } q \ (h,k) \subseteq (\bigcup_{i < \text{length } vs} \text{closureV } (vs ! i) \ (h, k))$   
**apply** (rule subsetI)  
**apply** clarsimp  
**apply** (rule-tac  $x=i$  **in** bexI)  
**apply** (simp add: closureV-def)  
**by** simp

**lemma** closureV-subseteq-closureL:  
 $h \ p = \text{Some } (j,C,vs)$   
 $\implies (\bigcup_{i < \text{length } vs} \text{closureV } (vs ! i) \ (h,k)) \subseteq \text{closureL } p \ (h,k)$   
**apply** (frule closureV-equals-closureL)  
**by** blast

**lemma** patrones:  
 $\mathbb{I} \ E1 \ x = \text{Some } (\text{Loc } p); h \ p = \text{Some } (j,C,vs);$   
 $i < \text{length } \text{alts}; \text{length } \text{alts} > 0; \text{length } \text{assert} = \text{length } \text{alts};$   
 $\text{set } (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! i)))) \cap \text{dom } E1 = \{\};$   
 $\text{length } (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! i)))) = \text{length } vs;$   
 $y \in \text{set } (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! i)))) \mathbb{I}$   
 $\implies \text{closure } (\text{extend } E1 \ (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! i)))) \ vs, E2) \ y \ (h, k) \subseteq$   
 $\text{closure } (\text{extend } E1 \ (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! i)))) \ vs, E2) \ x \ (h, k)$   
**apply** (rule subsetI)  
**apply** (subst (asm) closure-def)  
**apply** (case-tac  
 $\text{extend } E1 \ (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! i)))) \ vs \ y, \text{simp-all}$   
 $\text{case-tac } a, \text{simp-all}$   
 $\text{subgoal-tac } x \notin \text{set } (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! i))))$   
**prefer** 2 **apply** blast  
 $\text{frule-tac } x=x \text{ and } E=E1 \text{ and } vs=vs \text{ in } \text{extend-monotone-i}$   
 $\text{simp, simp, simp}$   
 $\text{rename-tac } q$   
 $\text{frule-tac } y=y \text{ in } \text{vs-defined, force, assumption+}$   
 $\text{simp add: closure-def}$   
 $\text{frule-tac } k=k \text{ in } \text{closureV-subseteq-closureL}$   
 $\text{elim exE, elim conjE}$   
 $\text{frule closure-Loc-subseteq-closureV-Loc, assumption+}$   
**by** force

**lemma** closure-monotone-extend-4:  
 $\mathbb{I} \ \text{def-extend } E \ (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! i)))) \ vs;$   
 $x \in \text{dom } E;$   
 $\text{length } \text{alts} > 0;$   
 $i < \text{length } \text{alts} \mathbb{I}$   
 $\implies \text{closure } (E, E') \ x \ (h, k) = \text{closure } (\text{extend } E \ (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! i))))$   
 $vs, E') \ x \ (h, k)$

```

apply (subgoal-tac  $x \notin \text{set} (\text{snd} (\text{extractP} (\text{fst} (\text{alts} ! i))))$ )
apply (subgoal-tac
   $E\ x = \text{extend}\ E\ (\text{snd} (\text{extractP} (\text{fst} (\text{alts} ! i))))\ \text{vs}\ x$ )
apply (simp add: closure-def)
apply (rule extend-monotone-i)
apply (simp, simp, simp)
apply (simp add: def-extend-def)
by blast

```

**lemma** P8-CASE-closureLS:

```

 $\llbracket \text{def-extend}\ E1\ (\text{snd} (\text{extractP} (\text{fst} (\text{alts} ! i))))\ \text{vs};$ 
 $\forall\ i < \text{length}\ \text{alts}. x \in \text{fst} (\text{assert} ! i) \wedge x \notin \text{set} (\text{snd} (\text{extractP} (\text{fst} (\text{alts} ! i))));$ 
 $(\bigcup_{i < \text{length}\ \text{alts}} \text{fst} (\text{assert} ! i) - \text{set} (\text{snd} (\text{extractP} (\text{fst} (\text{alts} ! i)))) \cup \{x\}$ 
 $\subseteq \text{dom} (\text{nonDisjointUnionEnvList} (\text{map}\ \text{snd}\ \text{assert}));$ 
 $\text{dom} (\text{foldl}\ \text{op}\ \otimes\ \text{empty} (\text{map}\ \text{snd}\ \text{assert})) \subseteq \text{dom}\ E1;$ 
 $E1\ x = \text{Some}\ (\text{Loc}\ p);$ 
 $h\ p = \text{Some}\ (j, C, \text{vs});$ 
 $i < \text{length}\ \text{assert}; \text{length}\ \text{assert} = \text{length}\ \text{alts}; 0 < \text{length}\ \text{assert}\rrbracket$ 
 $\implies \text{closureLS} (\text{extend}\ E1\ (\text{snd} (\text{extractP} (\text{fst} (\text{alts} ! i))))\ \text{vs}, E2) (\text{fst} (\text{assert} !$ 
 $i)) (h, k) \subseteq$ 
 $\text{closureLS} (E1, E2) (\text{insert}\ x\ (\bigcup_{i < \text{length}\ \text{alts}} \text{fst} (\text{assert} ! i) - \text{set} (\text{snd}$ 
 $(\text{extractP} (\text{fst} (\text{alts} ! i)))))) (h, k)$ 
apply (simp add: closureLS-def)
apply (rule subsetI, simp)
apply (erule bexE)
apply (rule disjI2)
apply (rule-tac  $x=i$  in bexI)
prefer 2 apply simp
apply (case-tac  $xaa \in \text{set} (\text{snd} (\text{extractP} (\text{fst} (\text{alts} ! i))))$ )
apply (rule-tac  $x=x$  in bexI)
prefer 2 apply simp
apply (simp add: def-extend-def)
apply (elim conjE)
apply (frule-tac  $y=xaa$  and  $\text{vs}=\text{vs}$  and  $j=j$  and
 $C=C$  and  $k=k$  and  $?E2.0=E2$  and  $h=h$  in patrones)
apply (assumption+, simp, assumption+)
apply (subgoal-tac
 $\text{closure} (E1, E2)\ x\ (h, k) =$ 
 $\text{closure} (\text{extend}\ E1\ (\text{snd} (\text{extractP} (\text{fst} (\text{alts} ! i))))\ \text{vs}, E2)\ x\ (h, k))$ 
apply blast
apply (rule closure-monotone-extend-4)
apply (simp add: def-extend-def)
apply (simp add: dom-def, simp, simp)
apply (rule-tac  $x=xaa$  in bexI)
prefer 2 apply simp
apply (subgoal-tac
 $xaa \in (\bigcup_{i < \text{length}\ \text{alts}} \text{fst} (\text{assert} ! i) - \text{set} (\text{snd} (\text{extractP} (\text{fst} (\text{alts} ! i))))))$ 
prefer 2 apply blast

```

**apply** (*subgoal-tac*  
 closure (*E1*, *E2*) *xaa* (*h*, *k*) =  
 closure (extend *E1* (snd (extractP (fst (alts ! *i*)))) *vs*, *E2*) *xaa* (*h*, *k*))  
**apply** *simp*  
**apply** (rule *closure-monotone-extend-4*)  
**by** (*simp,blast,simp,simp*)

**lemma** *P8-CASE*:

$\llbracket$  def-extend *E1* (snd (extractP (fst (alts ! *i*)))) *vs*;  
 $\forall i < \text{length alts}. x \in \text{fst}(\text{assert} ! i) \wedge x \notin \text{set}(\text{snd}(\text{extractP}(\text{fst}(\text{alts} ! i))))$ ;  
 $(\bigcup_{i < \text{length alts}} \text{fst}(\text{assert} ! i) - \text{set}(\text{snd}(\text{extractP}(\text{fst}(\text{alts} ! i)))) \cup \{x\}$   
 $\subseteq \text{dom}(\text{nonDisjointUnionEnvList}(\text{map} \text{snd} \text{assert}))$ ;  
 $\text{dom}(\text{foldl op} \otimes \text{empty}(\text{map} \text{snd} \text{assert})) \subseteq \text{dom } E1$ ;  
 $E1 \ x = \text{Some}(\text{Loc } p)$ ;  
 $h \ p = \text{Some}(j, C, vs)$ ;  
 $i < \text{length assert}; \text{length assert} = \text{length alts}; 0 < \text{length assert}$ ;  
 closed (*E1*, *E2*) (insert *x* ( $\bigcup_{i < \text{length alts}} \text{fst}(\text{assert} ! i) - \text{set}(\text{snd}(\text{extractP}(\text{fst}(\text{alts} ! i))))$ )) (*h*, *k*)  $\rrbracket$   
 $\implies$  closed (extend *E1* (snd (extractP (fst (alts ! *i*)))) *vs*, *E2*) (fst (assert ! *i*))  
 (*h*, *k*)  
**apply** (*simp add: closed-def*)  
**apply** (*simp add: live-def*)  
**apply** (*subgoal-tac*  
 closureLS (extend *E1* (snd (extractP (fst (alts ! *i*)))) *vs*, *E2*) (fst (assert ! *i*))  
 (*h*, *k*)  $\subseteq$   
 closureLS (*E1*, *E2*) (insert *x* ( $\bigcup_{i < \text{length alts}} \text{fst}(\text{assert} ! i) - \text{set}(\text{snd}(\text{extractP}(\text{fst}(\text{alts} ! i))))$ )) (*h*, *k*))  
**apply** *blast*  
**by** (rule *P8-CASE-closureLS,simp-all*)

**lemma** *P8-CASE-1-1-closureLS*:

$\llbracket$  fst (alts ! *i*) = ConstP (LitN *n*);  
 $i < \text{length assert}; \text{length assert} = \text{length alts}; 0 < \text{length assert}$   
 $\implies$  closureLS (*E1*, *E2*) (fst (assert ! *i*)) (*h*, *k*)  $\subseteq$   
 closureLS (*E1*, *E2*) (insert *x* ( $\bigcup_{i < \text{length alts}} \text{fst}(\text{assert} ! i) - \text{set}(\text{snd}(\text{extractP}(\text{fst}(\text{alts} ! i))))$ )) (*h*, *k*)  
**apply** (*simp add: closureLS-def*)  
**apply** (rule *subsetI,simp*)  
**apply** (erule *bexE*)  
**apply** (rule *disjI2*)  
**apply** (rule-tac *x=i* in *bexI*)  
**prefer** 2 **apply** *simp*  
**by** (rule-tac *x=xa* in *bexI,simp,simp*)

**lemma** *P8-CASE-1-1*:

$\llbracket$  fst (alts ! *i*) = ConstP (LitN *n*);

$i < \text{length } \text{assert}; \text{length } \text{assert} = \text{length } \text{alts}; 0 < \text{length } \text{assert};$   
 $\text{closed } (E1, E2) (\text{insert } x (\bigcup_{i < \text{length } \text{alts}} \text{fst } (\text{assert } ! i) - \text{set } (\text{snd } (\text{extractP}$   
 $(\text{fst } (\text{alts } ! i)))))) (h, k) \parallel$   
 $\implies \text{closed } (E1, E2) (\text{fst } (\text{assert } ! i)) (h, k)$   
**apply** (simp add: closed-def)  
**apply** (simp add: live-def)  
**apply** (subgoal-tac  
 $\text{closureLS } (E1, E2) (\text{fst } (\text{assert } ! i)) (h, k) \subseteq$   
 $\text{closureLS } (E1, E2) (\text{insert } x (\bigcup_{i < \text{length } \text{alts}} \text{fst } (\text{assert } ! i) - \text{set } (\text{snd } (\text{extractP}$   
 $(\text{fst } (\text{alts } ! i)))))) (h, k))$   
**apply** blast  
**by** (rule P8-CASE-1-1-closureLS,simp-all)

**lemma** P8-CASE-1-2-closureLS:

$\parallel \text{fst } (\text{alts } ! i) = \text{ConstP } (\text{LitB } b);$   
 $i < \text{length } \text{assert}; \text{length } \text{assert} = \text{length } \text{alts}; 0 < \text{length } \text{assert} \parallel$   
 $\implies \text{closureLS } (E1, E2) (\text{fst } (\text{assert } ! i)) (h, k) \subseteq$   
 $\text{closureLS } (E1, E2) (\text{insert } x (\bigcup_{i < \text{length } \text{alts}} \text{fst } (\text{assert } ! i) - \text{set } (\text{snd } (\text{extractP}$   
 $(\text{fst } (\text{alts } ! i)))))) (h, k)$   
**apply** (simp add: closureLS-def)  
**apply** (rule subsetI,simp)  
**apply** (erule bexE)  
**apply** (rule disjI2)  
**apply** (rule-tac  $x=i$  in bexI)  
**prefer** 2 **apply** simp  
**by** (rule-tac  $x=xa$  in bexI,simp,simp)

**lemma** P8-CASE-1-2:

$\parallel \text{fst } (\text{alts } ! i) = \text{ConstP } (\text{LitB } b);$   
 $i < \text{length } \text{assert}; \text{length } \text{assert} = \text{length } \text{alts}; 0 < \text{length } \text{assert};$   
 $\text{closed } (E1, E2) (\text{insert } x (\bigcup_{i < \text{length } \text{alts}} \text{fst } (\text{assert } ! i) - \text{set } (\text{snd } (\text{extractP}$   
 $(\text{fst } (\text{alts } ! i)))))) (h, k) \parallel$   
 $\implies \text{closed } (E1, E2) (\text{fst } (\text{assert } ! i)) (h, k)$   
**apply** (simp add: closed-def)  
**apply** (simp add: live-def)  
**apply** (subgoal-tac  
 $\text{closureLS } (E1, E2) (\text{fst } (\text{assert } ! i)) (h, k) \subseteq$   
 $\text{closureLS } (E1, E2) (\text{insert } x (\bigcup_{i < \text{length } \text{alts}} \text{fst } (\text{assert } ! i) - \text{set } (\text{snd } (\text{extractP}$   
 $(\text{fst } (\text{alts } ! i)))))) (h, k))$   
**apply** blast  
**by** (rule P8-CASE-1-2-closureLS,simp-all)

Lemmas for CASED

**lemma** closureL-p-None-p:

$\text{closureL } p (h(p := \text{None}), k) = \{p\}$   
**apply** (rule equalityI)  
**apply** (rule subsetI)

**apply** (erule closureL.induct,simp)  
**apply** (simp add: descendants-def)  
**apply** (rule subsetI,simp)  
**by** (rule closureL-basic)

**lemma** closure-extend-p-None-subseteq-closure:

$\llbracket E1\ x = \text{Some}\ (\text{Loc}\ p);$   
 $E1\ x = (\text{extend}\ E1\ (\text{snd}\ (\text{extractP}\ (\text{fst}\ (\text{alts}\ !\ i))))\ vs)\ x \rrbracket$   
 $\implies \text{closure}\ (\text{extend}\ E1\ (\text{snd}\ (\text{extractP}\ (\text{fst}\ (\text{alts}\ !\ i))))\ vs,\ E2)\ x\ (h(p := \text{None}),$   
 $k) \subseteq$   
 $\text{closure}\ (E1,\ E2)\ x\ (h,\ k)$   
**apply** (simp add: closure-def)  
**apply** (subst closureL-p-None-p,simp)  
**by** (rule closureL-basic)

**lemma** descendants-p-None-q:

$\llbracket d \in \text{descendants}\ q\ (h(p := \text{None}),k); q \neq p \rrbracket$   
 $\implies d \in \text{descendants}\ q\ (h,k)$   
**by** (simp add: descendants-def)

**lemma** closureL-p-None-subseteq-closureL:

$p \neq q$   
 $\implies \text{closureL}\ q\ (h(p := \text{None}),\ k) \subseteq \text{closureL}\ q\ (h,\ k)$   
**apply** (rule subsetI)  
**apply** (erule closureL.induct)  
**apply** (rule closureL-basic)  
**apply** clarsimp  
**apply** (subgoal-tac  $d \in \text{descendants}\ qa\ (h,k)$ )  
**apply** (rule closureL-step,simp,simp)  
**apply** (rule descendants-p-None-q,assumption+)  
**apply** (simp add: descendants-def)  
**by** (case-tac  $qa = p$ ,simp-all)

**lemma** dom-foldl-monotone-list:

$\text{dom}\ (\text{foldl}\ op\ \otimes\ (\text{empty}\ \otimes\ x)\ xs) =$   
 $\text{dom}\ x \cup \text{dom}\ (\text{foldl}\ op\ \otimes\ \text{empty}\ xs)$   
**apply** (subgoal-tac  $\text{empty}\ \otimes\ x = x\ \otimes\ \text{empty}$ ,simp)  
**apply** (subgoal-tac  $\text{foldl}\ op\ \otimes\ (x\ \otimes\ \text{empty})\ xs =$   
 $x\ \otimes\ \text{foldl}\ op\ \otimes\ \text{empty}\ xs$ ,simp)  
**apply** (rule union-dom-nonDisjointUnionEnv)  
**apply** (rule foldl-prop1)  
**apply** (subgoal-tac  $\text{def-nonDisjointUnionEnv}\ \text{empty}\ x$ )  
**apply** (erule nonDisjointUnionEnv-commutative)  
**by** (simp add: def-nonDisjointUnionEnv-def)

**lemma** dom-restrict-neg-map:

$\text{dom } (\text{restrict-neg-map } m \ A) = \text{dom } m - (\text{dom } m \cap A)$   
**apply** (*simp add: restrict-neg-map-def*)  
**apply** *auto*  
**by** (*split split-if-asm, simp-all*)

**lemma** *x-notin- $\Gamma$ -cased:*

$x \notin \text{dom } (\text{foldl } \text{op} \otimes \text{empty}$   
 $\quad (\text{map } (\lambda(Li, \Gamma i). \text{restrict-neg-map } \Gamma i \ (\text{insert } x \ (\text{set } Li)))$   
 $\quad (\text{zip } (\text{map } (\text{snd} \circ \text{extractP} \circ \text{fst}) \ \text{alts}) \ (\text{map } \text{snd } \text{assert}))))$   
**apply** (*induct-tac assert alts rule: list-induct2', simp-all*)  
**apply** (*subgoal-tac*  
 $\text{dom } (\text{foldl } \text{op} \otimes (\text{empty} \otimes \text{restrict-neg-map } (\text{snd } xa) \ (\text{insert } x \ (\text{set } (\text{snd } (\text{extractP}$   
 $(\text{fst } y))))))$   
 $\quad (\text{map } (\lambda(Li, \Gamma i). \text{restrict-neg-map } \Gamma i \ (\text{insert } x \ (\text{set } Li))) \ (\text{zip } (\text{map}$   
 $(\lambda a. \text{snd } (\text{extractP } (\text{fst } a))) \ ys) \ (\text{map } \text{snd } xs)))) =$   
 $\text{dom } (\text{restrict-neg-map } (\text{snd } xa) \ (\text{insert } x \ (\text{set } (\text{snd } (\text{extractP } (\text{fst } y)))))) \cup$   
 $\text{dom } (\text{foldl } \text{op} \otimes \text{empty } (\text{map } (\lambda(Li, \Gamma i). \text{restrict-neg-map } \Gamma i \ (\text{insert } x \ (\text{set } Li)))$   
 $\quad (\text{zip } (\text{map } (\lambda a. \text{snd } (\text{extractP } (\text{fst } a))) \ ys) \ (\text{map } \text{snd}$   
 $xs))))), \text{simp})$   
**apply** (*subst dom-restrict-neg-map, blast*)  
**by** (*rule dom-foldl-monotone-list*)

**lemma**  *$\Gamma$ -case- $x$ -is- $d$ :*

$\llbracket \Gamma = \text{foldl } \text{op} \otimes \text{empty}$   
 $\quad (\text{map } (\lambda(Li, \Gamma i). \text{restrict-neg-map } \Gamma i \ (\text{insert } x \ (\text{set } Li))) \ (\text{zip } (\text{map}$   
 $(\text{snd} \circ \text{extractP} \circ \text{fst}) \ \text{alts}) \ (\text{map } \text{snd } \text{assert}))) +$   
 $\quad \llbracket x \mapsto d'' \rrbracket$   
 $\implies \Gamma \ x = \text{Some } d''$   
**apply** (*simp add: disjointUnionEnv-def*)  
**apply** (*simp add: unionEnv-def*)  
**apply** (*rule impI*)  
**apply** (*insert x-notin- $\Gamma$ -cased*)  
**by** *force*

**lemma** *P8-CASED:*

$\llbracket E1 \ x = \text{Some } (\text{Loc } p); h \ p = \text{Some } (j, C, vs);$   
 $\text{length } \text{assert} > 0; \text{length } \text{assert} = \text{length } \text{alts};$   
 $\text{length } (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! \ i)))) = \text{length } vs;$   
 $\text{set } (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! \ i)))) \cap \text{dom } E1 = \{\};$   
 $\Gamma = \text{disjointUnionEnv}$   
 $\quad (\text{nonDisjointUnionEnvList } ((\text{map } (\lambda(Li, \Gamma i). \text{restrict-neg-map } \Gamma i \ (\text{set}$   
 $Li \cup \{x\}))))$   
 $\quad (\text{zip } (\text{map } (\text{snd} \circ \text{extractP} \circ \text{fst}) \ \text{alts}) \ (\text{map } \text{snd } \text{assert}))))$   
 $\quad (\text{empty}(x \mapsto d''));$

```

     $\forall z \in \text{dom } \Gamma. \Gamma z \neq \text{Some } s'' \longrightarrow (\forall i < \text{length } \text{alts}. z \notin \text{fst } (\text{assert } ! i));$ 
     $\text{shareRec } (\text{insert } x (\bigcup_i < \text{length } \text{alts } \text{fst } (\text{assert } ! i) - \text{set } (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! i))))))$ 
     $\Gamma (E1, E2) (h, k) (hh, k);$ 
     $\text{closed } (E1, E2) (\text{insert } x (\bigcup_i < \text{length } \text{alts } \text{fst } (\text{assert } ! i) - \text{set } (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! i)))))) (h, k);$ 
     $i < \text{length } \text{alts} \parallel$ 
     $\implies \text{closed } (\text{extend } E1 (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! i)))) \text{ vs, } E2) (\text{fst } (\text{assert } ! i))$ 
     $(h(p := \text{None}), k)$ 
apply (subgoal-tac  $\Gamma x = \text{Some } d''$ )
prefer 2 apply (rule  $\Gamma\text{-case-}x\text{-is-}d, \text{force}$ )
apply (simp add: closed-def)
apply (simp add: live-def)
apply (simp add: closureLS-def)
apply (simp add: domHeap-def)
apply (elim conjE)
apply (rule subsetI)
apply (rename-tac  $q, \text{simp}$ )
apply (elim bexE)
apply (rename-tac  $y$ )
apply (case-tac  $y \in \text{set } (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! i))))$ )

apply (rule conjI)

apply (frule-tac  $?E2.0 = E2$  and  $k = k$  in  $\text{patrones}$ )
apply (assumption+, simp, assumption+)
apply (subgoal-tac  $x \notin \text{set } (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! i))))$ )
prefer 2 apply blast
apply (frule-tac  $E = E1$  and  $\text{vs} = \text{vs}$  in  $\text{extend-monotone-}i, \text{simp}, \text{assumption+}$ )
apply (frule-tac  $\text{alts} = \text{alts}$  and  $\text{vs} = \text{vs}$  and  $i = i$  and  $?E2.0 = E2$  and  $k = k$  and  $h = h$  in
    closure-extend-p-None-subseteq-closure, simp)
apply (simp add: closure-def)
apply (case-tac  $\text{extend } E1 (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! i)))) \text{ vs } y, \text{simp-all}$ )
apply (case-tac  $a, \text{simp-all}$ )
apply (case-tac  $p = \text{nat}, \text{simp-all}$ )

apply (subst (asm) closureL-p-None-p, blast)

apply (frule-tac  $h = h$  and  $k = k$  in closureL-p-None-subseteq-closureL)
apply blast

apply (simp add: closure-def)
apply (case-tac  $\text{extend } E1 (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! i)))) \text{ vs } y, \text{simp-all}$ )
apply (case-tac  $a, \text{simp-all}$ )
apply (frule vs-defined, simp, simp, simp)
apply (elim exE, elim conjE)
apply (frule-tac  $k = k$  in no-cycles)
apply (erule-tac  $x = ja$  in allE, simp)

```

```

apply (simp add: closureV-def)
apply (case-tac p = nat, simp-all)

```

```

apply (subgoal-tac nat ∈ closureL nat (h, k), simp)
apply (rule closureL-basic)

```

```

apply (frule-tac h=h and k=k in closureL-p-None-subseteq-closureL)
apply blast

```

```

apply (simp add: closure-def)
apply (case-tac extend E1 (snd (extractP (fst (alts ! i)))) vs y, simp-all)
apply (case-tac a, simp-all)
apply (subgoal-tac y ∈ dom (extend E1 (snd (extractP (fst (alts ! i)))) vs))
prefer 2 apply force
apply (frule extend-prop1, simp, simp)
apply (simp only: shareRec-def)
apply (elim conjE)
apply (rotate-tac 20)
apply (erule thin-rl)
apply (rotate-tac 19)
apply (erule-tac x=y in ballE)
prefer 2 apply simp
apply (rotate-tac 19)
apply (erule-tac x=x in ballE)
prefer 2 apply force
apply (case-tac
  closure (E1, E2) y (h, k) ∩ recReach (E1, E2) x (h, k) ≠ {})
apply simp

```

```

apply (simp add: recReach-def)
apply (subgoal-tac p ∈ recReachL p (h,k))
prefer 2 apply (rule recReachL-basic)
apply (frule-tac E=E1 and vs=vs
  in extend-monotone-i, simp, simp)
apply (simp add: closure-def)
apply (case-tac p=nat, simp)

```

```

apply (subgoal-tac nat ∈ closureL nat (h,k))
apply (subgoal-tac nat ∈ recReachL nat (h,k))
apply blast
apply (rule recReachL-basic)
apply (rule closureL-basic)

```

```

apply (frule-tac h=h and k=k in closureL-p-None-subseteq-closureL)
apply (subgoal-tac
   $\forall x \in (\bigcup_{x < \text{length } \text{alts}}$ 
     $\bigcup_{x \in \text{fst } (\text{assert } ! x) - \text{set } (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! x))))}$ 
     $\text{case } E1 \ x \text{ of } \text{None} \Rightarrow \{\} \mid \text{Some } (\text{Loc } p) \Rightarrow \text{closureL } p \ (h, k) \mid \text{Some } - \Rightarrow$ 

```



```

{}).
   $x \in \text{dom } h$ )
prefer 2 apply blast
apply (erule-tac  $x=q$  in ballE)
apply force
apply simp
apply (erule-tac  $x=i$  in ballE)
prefer 2 apply simp
apply (rotate-tac 24)
apply (erule-tac  $x=y$  in ballE)
prefer 2 apply simp
apply simp
apply (erule-tac  $h=h$  and  $k=k$  in closureL-p-None-subseteq-closureL)
by blast

```

```

lemma atom-fvs-VarE:
   $\llbracket (\forall a \in \text{set } as. \text{atom } a); xa \in \text{fvs}' as \rrbracket$ 
   $\implies (\exists i < \text{length } as. \exists a. as!i = \text{VarE } xa a)$ 
apply (induct as,simp-all)
apply (case-tac a, simp-all)
by force

```

```

lemma nth-map-of-xs-atom2val:
   $\llbracket \text{length } xs = \text{length } as;$ 
   $\text{distinct } xs \rrbracket$ 
   $\implies \forall i < \text{length } xs.$ 
     $\text{map-of } (\text{zip } xs (\text{map } (\text{atom2val } E1) as)) (xs!i) =$ 
     $\text{Some } (\text{atom2val } E1 (as!i))$ 
apply clarsimp
apply (induct xs as rule: list-induct2',simp-all)
by (case-tac i,simp,clarsimp)

```

```

lemma closureL-k-equals-closureL-Suc-k:
   $\text{closureL } p (h, \text{Suc } k) = \text{closureL } p (h,k)$ 
apply (rule equalityI)
apply (rule subsetI)
apply (erule-tac closureL.induct)
apply (rule closureL-basic)
apply (rule closureL-step,simp)
apply (simp add: descendants-def)
apply (rule subsetI)
apply (erule-tac closureL.induct)

```

```

apply (rule closureL-basic)
apply (rule closureL-step,simp)
by (simp add: descendants-def)

```

```

lemma fvs-as-in-dom-E1:
   $\llbracket \text{fvs}' \text{ as} \subseteq \text{dom } \Gamma; \text{dom } \Gamma \subseteq \text{dom } E1; \\ i < \text{length as}; \text{as} ! i = \text{VarE } x \ a \rrbracket \\ \implies x \in \text{dom } E1$ 
apply (induct as i rule: list-induct3,simp-all)
by blast

```

```

lemma closure-APP-equals-closure-ef:
   $\llbracket \text{length } xs = \text{length as}; \text{distinct } xs; \\ \text{fvs}' \text{ as} \subseteq \text{dom } \Gamma; \text{dom } \Gamma \subseteq \text{dom } E1; \\ \text{distinct } xs; \\ i < \text{length as}; \text{as} ! i = \text{VarE } xa \ a \rrbracket \\ \implies \text{closure } (E1, E2) \text{ xa } (h, k) = \\ \text{closure } (\text{map-of } (\text{zip } xs \ (\text{map } (\text{atom2val } E1) \text{ as})), \text{map-of } (\text{zip } rs \ (\text{map } (the \\ \circ E2) \text{ rs}')) (\text{self} \mapsto \text{Suc } k)) (xs ! i) (h, \text{Suc } k)$ 
apply (frule-tac ?E1.0=E1 in nth-map-of-xs-atom2val,assumption+)
apply (erule-tac x=i in allE,simp)
apply (rule equalityI)

```

```

apply (rule subsetI)
apply (simp add: closure-def)
apply (case-tac E1 xa,simp-all)
apply (case-tac aa, simp-all)
apply (subst closureL-k-equals-closureL-Suc-k,assumption)

```

```

apply (rule subsetI)
apply (simp add: closure-def)
apply (case-tac E1 xa,simp-all)
apply (frule fvs-as-in-dom-E1,assumption+)
apply (simp add: dom-def)
apply (case-tac aa, simp-all)
apply (insert closureL-k-equals-closureL-Suc-k)
by simp

```

```

lemma nth-set-distinct:
   $\llbracket x \in \text{set } xs; \text{distinct } xs \rrbracket \\ \implies \exists i < \text{length } xs. xs ! i = x$ 
by (induct xs,simp,force)

```

```

lemma nth-map-add-map-of:
   $\llbracket i < \text{length } xs; \text{length } xs = \text{length } ms; \text{distinct } xs \rrbracket$ 

```

```

     $\Rightarrow (\Gamma ++ \text{map-of } (\text{zip } xs \ ms)) (xs!i) = \text{Some } (ms!i)$ 
  apply (subst map-add-Some-iff,simp)
  apply (subgoal-tac
    set (zip xs ms) =
    {(xs!i, ms!i) | i. i < min (length xs) (length ms)})
  apply force
  by (rule set-zip)

```

```

lemma map-add-map-of:
   $\llbracket x \in \text{set } xs; \text{dom } E1 \cap \text{set } xs = \{\}; \text{length } xs = \text{length } ys \rrbracket$ 
   $\Rightarrow (E1 ++ \text{map-of } (\text{zip } xs \ ys)) x = \text{map-of } (\text{zip } xs \ ys) x$ 
  apply (subgoal-tac E1 x = None)
  apply (simp only: map-add-def)
  apply (case-tac map-of (zip xs ys) x,simp-all)
  by blast

```

```

lemma var-in-fvs:
   $\llbracket i < \text{length } as; as ! i = \text{VarE } x \ a \rrbracket$ 
   $\Rightarrow x \in \text{fvs}' as$ 
  apply (induct as arbitrary: i, simp-all)
  apply clarsimp
  apply (case-tac i,simp-all)
  apply (case-tac aa, simp-all)
  by auto

```

```

declare atom.simps [simp del]

```

```

lemma live-APP-equals-live-ef:
   $\llbracket (\forall a \in \text{set } as. \text{atom } a); \text{length } xs = \text{length } as;$ 
     $\text{length } xs = \text{length } ms;$ 
     $\text{fvs}' as \subseteq \text{dom } \Gamma; \text{dom } \Gamma \subseteq \text{dom } E1;$ 
     $\text{dom } E1 \cap \text{set } xs = \{\}; \text{distinct } xs \rrbracket$ 
   $\Rightarrow \text{live } (E1, E2) (\text{fvs}' as) (h, k) =$ 
     $\text{live } (\text{map-of } (\text{zip } xs \ (\text{map } (\text{atom2val } E1) as)), \text{map-of } (\text{zip } rs \ (\text{map } (\text{the } \circ$ 
     $E2) rs')) (\text{self} \mapsto \text{Suc } k)) (\text{set } xs) (h, \text{Suc } k)$ 
  apply (rule equalityI)

```

```

  apply (rule subsetI)
  apply (simp add: live-def)
  apply (simp add: closureLS-def)
  apply (erule bexE)
  apply (frule atom-fvs-VarE,assumption+)
  apply (elim exE)
  apply (elim conjE, elim exE)
  apply (rule-tac x=xs!i in bexI)

```

```

  prefer 2 apply simp
  apply (frule closure-APP-equals-closure-ef)
  apply (assumption+,force)

  apply (rule subsetI)
  apply (simp add: live-def)
  apply (simp add: closureLS-def)
  apply (elim bexE)
  apply (frule nth-set-distinct,assumption+)
  apply (elim exE,elim conjE)
  apply (frule-tac ?E1.0=E1 in nth-map-of-xs-atom2val, assumption+)
  apply (rotate-tac 11)
  apply (erule-tac x=i in allE,simp)
  apply (erule-tac x=as!i in ballE)
  prefer 2 apply simp
  apply (subgoal-tac
    length xs = length (map (atom2val E1) as))
  apply (frule map-add-map-of,assumption+)
  prefer 2 apply simp
  apply (simp add: atom.simps)
  apply (case-tac (as ! i),simp-all)
  apply (simp add: closure-def)
  apply (rule-tac x=list in bexI)
  apply (case-tac E1 list,simp-all)
  apply (frule fvs-as-in-dom-E1,assumption+)
  apply (simp add: dom-def)
  apply (case-tac aa, simp-all)
  apply (subst (asm) closureL-k-equals-closureL-Suc-k)
  apply simp
  by (frule var-in-fvs,assumption+)

```

**lemma** *P8-APP-ef*:

```

  ⌊ (∀ a ∈ set as. atom a); length xs = length as;
    length xs = length ms;
    dom E1 ∩ set xs = {}; distinct xs;
    fvs' as ⊆ dom Γ; dom Γ ⊆ dom E1;
    closed (E1, E2) (fvs' as) (h, k) ⌋
  ⇒ closed (map-of (zip xs (map (atom2val E1) as)), map-of (zip rs (map (the
  ◦ E2) rs'))(self ↦ Suc k)) (set xs) (h, Suc k)
  apply (simp add: closed-def)
  apply (frule live-APP-equals-live-ef)
  apply (assumption+)
  apply (subgoal-tac domHeap (h, k) = domHeap (h, Suc k))
  apply force
  by (simp add: domHeap-def)

```

end

## 20 Derived Assertions. P9. closed v h'

**theory** *SafeDAss-P9* **imports** *SafeDAssBasic*  
*SafeRegion-definitions*  
*BasicFacts*

**begin**

Lemma for REUSE

**lemma** *closureV-equals-reuse*:

$\llbracket p \notin \text{closureV } v \ (h, k);$   
 $q \notin \text{closureV } v \ (h, k) \rrbracket$   
 $\implies \text{closureV } v \ (h, k) = \text{closureV } v \ (h(p := \text{None})(q \mapsto (j, C, vn)), k)$   
**apply** (*rule equalityI*)  
**apply** (*rule subsetI*)  
**apply** (*simp add: closureV-def*)  
**apply** (*case-tac v, simp-all*)  
**apply** (*rename-tac q*)  
**apply** (*erule closureL.induct*)  
**apply** (*rule closureL-basic*)  
**apply** (*rule closureL-step, simp*)  
**apply** (*simp add: descendants-def*)  
**apply** (*subgoal-tac qa  $\neq$  q, simp*)  
**prefer** 2 **apply** *blast*  
**apply** (*subgoal-tac qa  $\neq$  p, simp*)  
**apply** *blast*  
**apply** (*rule subsetI*)  
**apply** (*simp add: closureV-def*)  
**apply** (*case-tac v, simp-all*)  
**apply** (*rename-tac q*)  
**apply** (*erule closureL.induct*)  
**apply** (*rule closureL-basic*)  
**apply** (*rule closureL-step, simp*)  
**apply** (*simp add: descendants-def*)  
**apply** (*subgoal-tac qa  $\neq$  q, simp*)  
**prefer** 2 **apply** *blast*  
**apply** (*subgoal-tac qa  $\neq$  p, simp*)  
**by** *blast*

**lemma** *closureL-reuse-closureV*:

$\llbracket E1 \ x = \text{Some } (\text{Loc } p); h \ p = \text{Some } (j, C, vn);$   
 $\text{fresh } q \ h \rrbracket$   
 $\implies \text{closureL } q \ (h(p := \text{None})(q \mapsto (j, C, vn)), k) =$   
 $(\bigcup_{i < \text{length } vn} \text{closureV } (vn \ ! \ i) \ (h, k)) - \{p\} \cup \{q\}$   
**apply** (*frule-tac k=k in fresh-notin-closureL*)  
**apply** (*frule-tac k=k in no-cycles*)

```

apply (rule equalityI)
apply (rule subsetI)
apply (subst (asm) closureV-equals-closureL,force)
apply (erule-tac x=p in ballE)
  prefer 2 apply force
apply (subst (asm) closureV-equals-closureL,force,simp)
apply (erule disjE,simp)
apply (rule disjI2)
apply (elim bexE)
apply (rule-tac x=i in bexI)
  prefer 2 apply simp
apply (elim conjE)
apply (erule-tac x=i in ballE)
  prefer 2 apply simp
apply (erule-tac x=i in allE,simp)
apply (subst closureV-equals-reuse [where p=p],assumption+)
apply simp
apply (rule conjI)
  apply (rule closureL-basic)
apply (rule subsetI)
apply (subst closureV-equals-closureL,force)
apply (erule-tac x=p in ballE)
  prefer 2 apply force
apply (subst (asm) closureV-equals-closureL,force,simp)
apply (rule disjI2)
apply (elim bexE)
apply (rule-tac x=i in bexI)
  prefer 2 apply simp
apply (elim conjE)
apply (erule-tac x=i in ballE)
  prefer 2 apply simp
apply (erule-tac x=i in allE,simp)
by (subst (asm) closureV-equals-reuse [where p=p],assumption+)

```

**lemma** P9-REUSE:

```

  [| E1 x = Some (Loc p); h p = Some c; fresh q h;
    closed (E1, E2) {x} (h, k) |]
  ==> closed-f (Loc q) (h(p := None)(q ↦ c), k)
apply (case-tac c)
apply (case-tac b)
apply simp
apply (rename-tac j b C vn)
apply (simp add: closed-def)
apply (simp add: live-def)
apply (simp add: closureLS-def)
apply (simp add: closure-def)
apply (simp add: closed-f-def)
apply (simp add: domHeap-def)

```

**apply** (*subst closureL-reuse-closureV,assumption+*)  
**apply** (*subst (asm) closureV-equals-closureL,force*)  
**by** *blast*

Lemma for COPY

**axioms** *P9-COPY*:

$\llbracket E1\ x = \text{Some}\ (\text{Loc}\ p);$   
 $\text{copy}\ (h, k)\ p\ j = ((h', k), p'); \text{def-copy}\ p\ (h, k);$   
 $\text{closed}\ (E1, E2)\ \{x\}\ (h, k)\ \rrbracket$   
 $\implies \text{closed-f}\ (\text{Loc}\ p')\ (h', k)$

Lemmas for LET1 and LET2

**lemma** *P9-LET*:

$\llbracket \text{closed-f}\ v1\ (h', k'); \text{closed-f}\ v\ (hh, kk) \rrbracket$   
 $\implies \text{closed-f}\ v\ (hh, kk)$

**by** *simp*

Lemmas for LET1C and LET2C

**axioms** *none-notequal-p*:

*the None  $\neq$  Loc p*

**lemma** *closed-e1-closureV-subseteq-dom-h*:

$\llbracket \text{closed}\ (E1, E2)\ (\text{set}\ (\text{map}\ \text{atom2var}\ as))\ (h, k);$   
 $\forall a \in \text{set}\ as. \text{atom}\ a\ \rrbracket$   
 $\implies (\bigcup_{i < \text{length}\ as} \text{closureV}\ (\text{map}\ (\text{atom2val}\ E1)\ as\ !\ i)\ (h, k)) \subseteq$   
 $\text{dom}\ h$

**apply** (*simp add: closed-def*)

**apply** (*simp add: live-def*)

**apply** (*simp add: closureLS-def*)

**apply** (*simp add: closure-def*)

**apply** (*simp add: closureV-def*)

**apply** (*rule subsetI*)

**apply** (*erule UN-E*)

**apply** (*erule-tac x=as!i in ballE*)

**prefer** 2 **apply** *simp*

**apply** (*case-tac as!i, simp-all*)

**apply** (*case-tac atom2val E1 (as!i), simp-all*)

**apply** (*subgoal-tac as!i  $\in$  set as*)

**prefer** 2 **apply** (*subst set-conv-nth, force*)

**apply** (*subgoal-tac*

*(case E1 (atom2var (as!i)) of None  $\Rightarrow$  {} | Some (Loc p)  $\Rightarrow$  closureL p (h, k) |*  
*Some -  $\Rightarrow$  {}))  $\subseteq$*

*domHeap (h, k))*

**prefer** 2 **apply** *blast*

**apply** (*simp add: domHeap-def*)

**apply** (*subgoal-tac*

*x  $\in$  (case E1 list of None  $\Rightarrow$  {} | Some (Loc p)  $\Rightarrow$  closureL p (h, k) | Some -  $\Rightarrow$*   
*{})*)

**apply** *blast*  
**apply** (*case-tac E1 list,simp-all*)  
**by** (*insert none-notequal-p,force*)

**lemma** *closureV-upt-subset-closureV*:

*fresh p h*  
 $\implies (\bigcup_{i < \text{length} (\text{map} (\text{atom2val } E1) \text{ as})} \text{closureV} (\text{map} (\text{atom2val } E1) \text{ as} ! i)$   
 $(h(p \mapsto (j, C, \text{map} (\text{atom2val } E1) \text{ as})), k)) \subseteq$   
 $(\bigcup_{i < \text{length} (\text{map} (\text{atom2val } E1) \text{ as})} \text{closureV} (\text{map} (\text{atom2val } E1) \text{ as} ! i)$   
 $(h, k)) \cup \{p\}$

**apply** (*rule subsetI*)  
**apply** (*erule UN-E*)  
**apply** (*subgoal-tac*  
 $(h(p \mapsto (j, C, \text{map} (\text{atom2val } E1) \text{ as}))) p =$   
 $\text{Some } (j, C, \text{map} (\text{atom2val } E1) \text{ as}))$   
**prefer 2 apply force**  
**apply** (*frule-tac k=k and h=(h(p \mapsto (j, C, \text{map} (\text{atom2val } E1) \text{ as}))) in no-cycles*)  
**apply** (*simp add: closureV-def*)  
**apply** (*erule-tac x=i in allE,simp*)  
**apply** (*case-tac atom2val E1 (as ! i),simp-all*)  
**apply** (*rule disjI2*)  
**apply** (*rule-tac x=i in bexI,simp*)  
**prefer 2 apply simp**  
**apply** (*erule closureL.induct*)  
**apply** (*rule closureL-basic*)  
**apply** (*rule closureL-step,simp*)  
**apply** (*simp add: descendants-def*)  
**by** (*case-tac q = p,simp-all*)

**lemma** *P9-LETC-e1*:

$\llbracket \forall a \in \text{set as. atom } a; \text{ fresh } p h;$   
 $\text{closed } (E1, E2) (\text{set } (\text{map atom2var as})) (h, k) \rrbracket$   
 $\implies \text{closed-f } (\text{Loc } p) (h(p \mapsto (j, C, \text{map} (\text{atom2val } E1) \text{ as})), k)$   
**apply** (*drule closed-e1-closureV-subseteq-dom-h,assumption+*)  
**apply** (*simp add: closed-f-def*)  
**apply** (*subst closureV-equals-closureL*)  
**apply force**  
**apply** (*subgoal-tac*  
 $(\bigcup_{i < \text{length} (\text{map} (\text{atom2val } E1) \text{ as})} \text{closureV} (\text{map} (\text{atom2val } E1) \text{ as} ! i) (h(p$   
 $\mapsto (j, C, \text{map} (\text{atom2val } E1) \text{ as})), k)) \subseteq$   
 $(\bigcup_{i < \text{length} (\text{map} (\text{atom2val } E1) \text{ as})} \text{closureV} (\text{map} (\text{atom2val } E1) \text{ as} ! i)$   
 $(h, k)) \cup \{p\}$ )  
**apply** (*simp add: domHeap-def*)  
**apply blast**  
**by** (*rule closureV-upt-subset-closureV,assumption*)



Lemmas for APP

**axioms** *extend-heaps-P9*:

$(h', \text{Suc } k) \sqsubseteq (h' \mid \{p \in \text{dom } h'. \text{fst } (\text{the } (h' p)) \leq k\}, k)$

**lemma** *P9-APP-1*:

$\llbracket x \in \text{domHeap } (h', \text{Suc } k); h x = h' x \rrbracket$   
 $\implies x \in \text{domHeap } (h, k)$

**apply** (*simp only: domHeap-def*)

**by** (*simp add: dom-def*)

**axioms** *Lemma4-consistent-v*:

$(h, k) \sqsubseteq (h', k')$   
 $\implies \forall t \eta p.$   
 $\quad \text{consistent-v } t \eta (\text{Loc } p) h$   
 $\quad \longrightarrow \text{consistent-v } t \eta (\text{Loc } p) h'$

**axioms** *consistent-v-identityClosure*:

$\text{consistent-v } t \eta (\text{Loc } p) h$   
 $\longrightarrow \text{consistent-v } t \eta (\text{Loc } p) h'$   
 $\implies \text{closureL } p (h, k) = \text{closureL } p (h', k') \wedge (\forall x \in \text{closureL } p (h, k). h x = h' x)$

**lemma** *P9-APP*:

$\llbracket hh = h' \mid \{p \in \text{dom } h'. \text{fst } (\text{the } (h' p)) \leq k\};$   
 $\quad \text{closed-f } v (h', \text{Suc } k) \rrbracket$   
 $\implies \text{closed-f } v (hh, k)$

**apply** (*simp add: closed-f-def*)

**apply** (*case-tac v, simp-all*)

**apply** (*rename-tac q*)

**apply** (*subgoal-tac*

$(h', \text{Suc } k) \sqsubseteq$   
 $(h' \mid \{p \in \text{dom } h'. \text{fst } (\text{the } (h' p)) \leq k\}, k))$

**prefer** 2 **apply** (*rule extend-heaps-P9*)

**apply** (*frule Lemma4-consistent-v*)

**apply** (*erule-tac x=t in allE*)

**apply** (*erule-tac x=η in allE*)

**apply** (*erule-tac x=q in allE*)

**apply** (*drule consistent-v-identityClosure [where k=Suc k and k'=k]*)

**apply** (*elim conjE*)

**apply** (*rule subsetI*)

**apply** (*subgoal-tac x ∈ domHeap (h', Suc k)*)

**prefer** 2 **apply** *blast*

**apply** (*erule-tac x=x in ballE*)

**apply** (*frule P9-APP-1 [where h=h' | {p ∈ dom h'. fst (the (h' p)) ≤ k}]*)

**apply** (*rule sym, assumption+*)

**by** *blast*

**axioms** *P9-APP*:  
 $\llbracket hh = h' \mid \{p \in \text{dom } h'. \text{fst } (the \ (h' \ p)) \leq k\} \rrbracket$   
 $\implies \text{closed-f } v \ (hh, k)$

Lemmas for APP-PRIMOP

**lemma** *P9-APP-PRIMOP*:  
 $\text{closed-f } (execOp \ oper \ (atom2val \ E1 \ a1) \ (atom2val \ E1 \ a2)) \ (h, k)$   
**apply** (*simp only: closed-f-def*)  
**by** (*case-tac oper, simp-all*)

**end**

## 21 Derived Assertions

**theory** *SafeDAssDepth* **imports** *SafeDAssBasic SafeDepthSemantics*  
*SafeDAss-P2 SafeDAss-P3 SafeDAss-P1*  
*SafeDAss-P5-P6 SafeDAss-P4 SafeDAss-P7 SafeDAss-P8*  
*SafeDAss-P9*

**begin**

**declare** *dom-fun-upd* [*simp del*]

**lemma** *SafeDADepth-LitInt*:  $ConstE \ (LitN \ i) \ a :_f, n \ \{\} \ , \ empty \}$   
**apply** (*simp only: SafeDAssDepth-def*)  
**apply** (*rule conjI, simp*)  
**apply** (*intro allI, rule impI*)  
**apply** (*elim conjE*)  
**apply** (*frule impSemBoundRA [where td=td]*)  
**apply** (*elim exE*)  
**apply** (*erule SafeRASem.cases*) **apply** (*simp-all*)  
**apply** (*simp add: closed-f-def*)  
**apply** (*simp add: shareRec-def*)  
**by** (*rule ballI, simp add: identityClosure-def*)

**lemma** *SafeDADepth-LitBool*:  $ConstE \ (LitB \ b) \ a :_f, n \ \{\} \ , \ empty \}$   
**apply** (*simp only: SafeDAssDepth-def*)  
**apply** (*rule conjI, simp*)  
**apply** (*intro allI, rule impI*)  
**apply** (*elim conjE*)  
**apply** (*frule impSemBoundRA [where td=td]*)

```

apply (elim exE)
apply (erule SafeRASem.cases,simp-all)
apply (simp add: closed-f-def)
apply (simp add: shareRec-def)
by (rule ballI, simp add: identityClosure-def)

```

```

lemma SafeDADepth-Var1:
   $\llbracket \Gamma \ x = \text{Some } s' \rrbracket$ 
   $\implies \text{VarE } x \ a :_f, \ n \ \{ \{x\} \}, \Gamma \rrbracket$ 
apply (simp add: SafeDAssDepth-def)
apply (rule conjI, simp add: dom-def)
apply (intro allI, rule impI)
apply (erule conjE)
apply (frule impSemBoundRA [where td=td])
apply (elim exE)
apply (erule SafeRASem.cases,simp-all)
apply (rule conjI, simp add: shareRec-def)
apply (rule ballI) apply (simp add: identityClosure-def)
apply (rule impI)
apply (elim conjE)
apply (simp add: closed-def add: live-def add: closureLS-def add: closure-def)
by (simp add: closed-f-def)

```

```

declare copy.simps [simp del]

```

```

lemma SafeDADepth-Var2:
   $\llbracket x \in \text{dom } \Gamma; \text{wellFormed } \{x\} \ \Gamma \ (\text{CopyE } x \ r \ d) \rrbracket$ 
   $\implies \text{CopyE } x \ r \ d :_f, \ n \ \{ \{x\} \}, \Gamma \rrbracket$ 
apply (simp only: SafeDAssDepth-def)
apply (rule conjI, simp)
apply (rule conjI, simp add: dom-def)
apply (intro allI, rule impI)
apply (elim conjE)
apply (frule impSemBoundRA [where td=td])
apply (elim exE)
apply (frule P1-COPY)
apply (elim exE, elim conjE)
apply (rule conjI)
apply simp
apply (rule P5-P6-COPY,assumption+)
apply (rule impI, elim conjE)
by (simp,rule P9-COPY,assumption+)

```

**lemma** *SafeDADepth-Var3*:  

$$\llbracket \Gamma \ x = \text{Some } d''; \text{wellFormed } \{x\} \ \Gamma \ (\text{ReuseE } x \ a) \rrbracket$$

$$\implies \text{ReuseE } x \ a :_f, n \llbracket \{x\}, \Gamma \rrbracket$$
**apply** (*simp only: SafeDAssDepth-def*)  
**apply** (*rule conjI, simp*)  
**apply** (*rule conjI, simp add: dom-def*)  
**apply** (*intro allI, rule impI*)  
**apply** (*elim conjE*)  
**apply** (*frule impSemBoundRA [where td=td]*)  
**apply** (*elim exE*)  
**apply** (*frule P1-REUSE*)  
**apply** (*elim exE, elim conjE*)  
**apply** (*rule conjI*)  
**apply** *simp*  
**apply** (*rule P5-P6-REUSE, assumption+*)  
**apply** (*rule impI, elim conjE*)  
**by** (*simp, rule P9-REUSE, assumption+*)

**lemma** *SafeDADepth-APP-PRIMOP*:  

$$\llbracket \text{atom } a1; \text{atom } a2; \text{primops } g = \text{Some } \text{oper};$$

$$L = \{\text{atom2var } a1, \text{atom2var } a2\};$$

$$\Gamma 0 = [\text{atom2var } a1 \mapsto s'', \text{atom2var } a2 \mapsto s''];$$

$$\Gamma 0 \subseteq_m \Gamma \rrbracket$$

$$\implies \text{AppE } g \ [a1, a2] \llbracket a :_f, n \llbracket L, \Gamma \rrbracket$$
**apply** (*simp only: SafeDAssDepth-def*)

**apply** (*rule conjI*)  
**apply** (*rule P4-APP-PRIMOP, assumption+*)

**apply** (*rule conjI*)  
**apply** (*rule P3-APP-PRIMOP, assumption+*)

**apply** (*rule allI*)  
**apply** (*rule impI*)  
**apply** (*elim conjE*)

**apply** (*simp add: SafeBoundSem-def*)

**apply** (*elim exE, elim conjE*)  
**apply** (*erule SafeDepthSem.cases, simp-all*)

**apply** (*rule conjI*)  
**apply** (*rule P5-P6-APP-PRIMOP, force, assumption*)

**apply** (*rule impI*)  
**by** (*rule P9-APP-PRIMOP*)

**lemma** *SafeDADepth-LET1*:  
 $\llbracket e1 : f, n \rrbracket L1, \Gamma1 \rrbracket$ ;  
 $e2 : f, n \rrbracket L2, \Gamma2' \rrbracket$ ;  
 $\Gamma2' = \text{disjointUnionEnv } \Gamma2 \ (\text{empty}(x1 \mapsto s''))$ ;  
 $\text{def-disjointUnionEnv } \Gamma2 \ (\text{empty}(x1 \mapsto s''))$ ;  
 $\text{def-pp } \Gamma1 \ \Gamma2 \ L2$ ;  
 $x1 \notin L1$ ;  
 $L = L1 \cup (L2 - \{x1\})$ ;  
 $\Gamma = \text{pp } \Gamma1 \ \Gamma2 \ L2$ ;  
 $\forall C \text{ as } r \ a'. \ e1 \neq \text{ConstrE } C \text{ as } r \ a''$   
 $\implies \text{Let } x1 = e1 \text{ In } e2 \ a : f, n \rrbracket L, \Gamma \rrbracket$   
**apply** (*simp (no-asm) only: SafeDAssDepth-def*)

**apply** (*simp only: SafeDAssDepth-def*)  
**apply** (*elim conjE*)

**apply** (*rule conjI*)  
**apply** (*erule P4-LET, assumption+*)

**apply** (*rule conjI*)  
**apply** (*simp, rule conjI*)  
**apply** (*erule P3-LET-e1*)  
**apply** (*erule P3-LET-e2, assumption+*)

**apply** (*rule allI*)  
**apply** (*rule impI*)  
**apply** (*elim conjE*)  
**apply** (*erule P1-f-n-LET, simp*)  
**apply** (*elim exE, elim conjE*)

**apply** (*erule-tac x=E1 in allE*)  
**apply** (*erule-tac x=E1 (x1  $\mapsto$  v1) in allE*)

```

apply (erule-tac  $x=E2$  in  $allE$ )
apply (erule-tac  $x=E2$  in  $allE$ )
apply (erule-tac  $x=h$  in  $allE$ )
apply (rotate-tac 13)
apply (erule-tac  $x=h'$  in  $allE$ )
apply (erule-tac  $x=k$  in  $allE$ )
apply (erule-tac  $x=k$  in  $allE$ )
apply (erule-tac  $x=h'$  in  $allE$ )
apply (erule-tac  $x=hh$  in  $allE$ )
apply (erule-tac  $x=v1$  in  $allE$ )
apply (erule-tac  $x=v$  in  $allE$ )

```

```

apply (frule-tac  $r=v1$  in  $P2-LET$ , assumption+)
apply (elim conjE)

```

```

apply (drule mp,simp)
apply (drule mp,simp)
apply (elim conjE)

```

```

apply (rule conjI)
apply (erule P5-P6-LET, assumption+,simp)

```

```

apply (rule impI)
apply (elim conjE)

```

```

apply (frule P8-LET-e1,simp)

```

```

apply (frule P7-LET-e1, assumption+, simp)

```

```

apply (frule P8-LET-e2, assumption+, simp)

```

```

apply (frule P7-LET1-e2, assumption+)

```

```

by (rule P9-LET, assumption+, simp)

```

```

declare atom.simps [simp del]

```

```

lemma SafeDADepth-LET1C:
  [|  $L1 = set (map\ atom2var\ as)$ ;  $\forall a \in set\ as.\ atom\ a$ ;

```

```

 $\Gamma 1 = \text{map-of } (\text{zip } (\text{map } \text{atom2var } as) (\text{replicate } (\text{length } as) s'')); \\
x1 \notin L1; \\
e2 :_f, n \Vdash L2, \text{disjointUnionEnv } \Gamma 2 (\text{empty}(x1 \mapsto s'')) \Vdash; \\
\text{def-disjointUnionEnv } \Gamma 2 (\text{empty}(x1 \mapsto s'')); \\
\text{def-pp } \Gamma 1 \Gamma 2 L2; \\
L = L1 \cup (L2 - \{x1\}); \\
\Gamma = \text{pp } \Gamma 1 \Gamma 2 L2 \parallel \\
\Rightarrow \text{Let } x1 = \text{ConstrE } C \text{ as } r \text{ a' In } e2 \text{ a} :_f, n \Vdash L, \Gamma \Vdash \\
\text{apply } (\text{simp only: SafeDAssDepth-def}) \\
\text{apply } (\text{elim conjE})$ 
```

```

apply (frule set-atom2var-as-subeteq- $\Gamma 1$ )

```

```

apply (rule conjI)
apply (frule fvs-as-subeteq-L1)
apply (rule P4-LET, simp, assumption+)

```

```

apply (rule conjI)
apply (rule P3-LET, assumption+)

```

```

apply (rule allI)+
apply (rule impI)
apply (elim conjE)

```

```

apply (frule P1-f-n-LETC, simp)
apply (elim exE, elim conjE)

```

```

apply (erule-tac x=E1 (x1  $\mapsto$  Loc p) in allE)
apply (erule-tac x=E2 in allE)
apply (erule-tac x=h (p  $\mapsto$  (j, C, map (atom2val E1) as)) in allE)
apply (erule-tac x=k in allE)
apply (erule-tac x=hh in allE)
apply (erule-tac x=v in allE)

```

```

apply (drule mp, simp)

```

```

apply (rule P2-LET-e2, assumption+)

```

```

apply (elim conjE)

```

```

apply (rule conjI)
apply (frule P5-P6-f-n-LETC-e1,assumption+,simp,assumption+,simp)
apply (frule P2-LET-e1)
apply (rule P5-P6-LET,assumption+)

```

```

apply (rule impI,elim conjE)

```

```

apply (frule P5-P6-f-n-LETC-e1 [where ?L1.0=L1 and ?Γ1.0=Γ1])
apply (assumption+,simp,assumption+,simp,assumption+)
apply (frule P8-LET-e1)
apply (frule P9-LETC-e1,assumption+,simp)
apply (frule P2-LET-e2,assumption+)
apply (frule P8-LET-e2)
apply (assumption+,force,assumption+)
apply simp

```

```

apply (frule P5-P6-f-n-LETC-e1,assumption+,simp,assumption+,simp)
apply (frule P2-LET-e1)
apply (frule P7-LET1-e2,assumption+)

```

```

apply (drule mp)
apply force

```

```

by simp

```

**lemma** *SafeDADepth-LET2*:

```

  ⌊ ∀ C as r a'. e1 ≠ ConstrE C as r a';
    e1 :f , n ⌊ L1 , Γ1 ⌋;
    e2 :f , n ⌊ L2, disjointUnionEnv Γ2 (empty(x1↦d'')) ⌋;
    def-disjointUnionEnv Γ2 (empty(x1↦d''));
    def-pp Γ1 Γ2 L2;
    x1 ∉ L1 ⌋
  ⇒ Let x1 = e1 In e2 a :f , n ⌊ L1 ∪ (L2-⌊x1⌋) , pp Γ1 Γ2 L2 ⌋
apply (simp (no-asm) only: SafeDAssDepth-def)

```

```

apply (simp only: SafeDAssDepth-def)
apply (elim conjE)

```



**apply** (*rule conjI*)  
**apply** (*erule P4-LET, assumption+*)

**apply** (*rule conjI*)  
**apply** (*simp, rule conjI*)  
**apply** (*erule P3-LET-e1*)  
**apply** (*erule P3-LET-e2, assumption+*)

**apply** (*rule allI*)  
**apply** (*rule impI*)  
**apply** (*elim conjE*)  
**apply** (*frule P1-f-n-LET, simp*)  
**apply** (*elim exE, elim conjE*)

**apply** (*erule-tac x=E1 in allE*)  
**apply** (*erule-tac x=E1 (x1  $\mapsto$  v1) in allE*)  
**apply** (*erule-tac x=E2 in allE*)  
**apply** (*erule-tac x=E2 in allE*)  
**apply** (*erule-tac x=h in allE*)  
**apply** (*erule-tac x=h' in allE*)  
**apply** (*erule-tac x=k in allE*)  
**apply** (*erule-tac x=k in allE*)  
**apply** (*erule-tac x=h' in allE*)  
**apply** (*erule-tac x=hh in allE*)  
**apply** (*erule-tac x=v1 in allE*)  
**apply** (*erule-tac x=v in allE*)

**apply** (*frule-tac r=v1 in P2-LET, assumption+*)  
**apply** (*elim conjE*)

**apply** (*drule mp, simp*)  
**apply** (*drule mp, simp*)  
**apply** (*elim conjE*)

**apply** (*rule conjI*)  
**apply** (*erule P5-P6-LET, assumption+*)

**apply** (*rule impI*)  
**apply** (*elim conjE*)

**apply** (*frule P8-LET-e1, simp*)

**apply** (*frule P7-LET-e1, assumption+, simp*)

**apply** (*frule* *P8-LET-e2*, *assumption+*, *simp*)

**apply** (*frule* *P7-LET2-e2*, *assumption+*)

**by** (*rule* *P9-LET*, *assumption+*, *simp*)

**lemma** *SafeDADepth-LET2C*:

$\llbracket L1 = \text{set } (\text{map } \text{atom2var } as); \forall a \in \text{set } as. \text{atom } a;$   
 $\Gamma 1 = \text{map-of } (\text{zip } (\text{map } \text{atom2var } as) (\text{replicate } (\text{length } as) s''));$   
 $x1 \notin L1;$   
 $e2 :_f, n \llbracket L2, \text{disjointUnionEnv } \Gamma 2 (\text{empty}(x1 \mapsto d'')) \rrbracket;$   
 $\text{def-disjointUnionEnv } \Gamma 2 (\text{empty}(x1 \mapsto d''));$   
 $\text{def-pp } \Gamma 1 \Gamma 2 L2;$   
 $L = L1 \cup (L2 - \{x1\});$   
 $\Gamma = \text{pp } \Gamma 1 \Gamma 2 L2 \rrbracket$   
 $\implies \text{Let } x1 = \text{ConstrE } C \text{ as } r \text{ } a' \text{ In } e2 \text{ } a :_f, n \llbracket L, \Gamma \rrbracket$   
**apply** (*simp only*: *SafeDAAssDepth-def*)  
**apply** (*elim conjE*)

**apply** (*frule* *set-atom2var-as-subeteq- $\Gamma 1$* )

**apply** (*rule conjI*)

**apply** (*frule* *fvs-as-subseteq-L1*)

**apply** (*rule* *P4-LET*, *simp*, *assumption+*)

**apply** (*rule conjI*)

**apply** (*rule* *P3-LET*, *assumption+*)

**apply** (*rule allI*)**+**

**apply** (*rule impI*)

**apply** (*elim conjE*)

**apply** (*frule* *P1-f-n-LETC*, *simp*)

**apply** (*elim exE*, *elim conjE*)

**apply** (*erule-tac*  $x = E1(x1 \mapsto \text{Loc } p)$  **in** *allE*)

```

apply (erule-tac  $x=E2$  in allE)
apply (erule-tac  $x=h(p \mapsto (j, C, \text{map } (\text{atom2val } E1) \text{ as}))$  in allE)
apply (erule-tac  $x=k$  in allE)
apply (erule-tac  $x=hh$  in allE)
apply (erule-tac  $x=v$  in allE)

apply (drule mp,simp)

apply (rule P2-LET-e2,assumption+)

apply (elim conjE)

apply (rule conjI)
apply (frule P5-P6-f-n-LETC-e1,assumption+,simp,assumption+,simp)
apply (frule P2-LET-e1)
apply (rule P5-P6-LET,assumption+)

apply (rule impI,elim conjE)

apply (frule P5-P6-f-n-LETC-e1 [where ?L1.0=L1 and ?Γ1.0=Γ1])
apply (assumption+,simp,assumption+,simp,assumption+)
apply (frule P8-LET-e1)
apply (frule P9-LETC-e1,assumption+,simp)
apply (frule P2-LET-e2,assumption+)
apply (frule P8-LET-e2)
apply (assumption+,force,assumption+)
apply simp

apply (frule P5-P6-f-n-LETC-e1,assumption+,simp,assumption+,simp)
apply (frule P2-LET-e1)
apply (frule P7-LET2-e2,assumption+)

apply (drule mp)
apply force

by simp

```

**declare** *fv-fvs-fvs'-fvAlts-fvTup-fvAlts'-fvTup'.simps* [*simp del*]

**lemma** *SafeDADepth-CASE*:

[[ *def-nonDisjointUnionEnvList* (*map snd assert*);  
   *length* (*map snd assert*) > 0; *length assert* = *length alts*;  
    $\forall i < \text{length alts}. \forall j < \text{length alts}. i \neq j \longrightarrow (\text{fst } (\text{assert } ! i) \cap \text{set } (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! j)))) = \{\}$ ;  
    $\forall i < \text{length alts}. \forall x \in \text{set } (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! i)))) . \text{snd } (\text{assert } ! i) x \neq \text{Some } d''$ ;  
    $\forall i < \text{length assert}. \text{snd } (\text{alts } ! i) :_f, n \Vdash \text{fst } (\text{assert } ! i), \text{snd } (\text{assert } ! i) \Vdash$ ;  
    $\forall i < \text{length alts}. x \in \text{fst } (\text{assert } ! i) \wedge x \notin \text{set } (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! i))))$ ;  
    $x \in \text{dom } \Gamma$ ;  
   *wellFormedDepth* *f n L*  $\Gamma$  (*Case* (*VarE* *x a*) *Of alts a'*);  
    $L = (\bigcup i < \text{length assert}. \text{fst } (\text{assert } ! i) - \text{set } (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! i))))$   
 $\cup \{x\}$ ;  
    $\Gamma = \text{nonDisjointUnionEnvList } (\text{map snd assert})$ ]]  
 $\implies \text{Case } (\text{VarE } x a) \text{ Of alts } a' :_f, n \Vdash L, \Gamma \Vdash$   
**apply** (*simp* (*no-asm*) *only: SafeDAssDepth-def*)  
**apply** (*simp only: SafeDAssDepth-def*)

**apply** (*rule conjI*)  
**apply** (*rule P4-CASE,simp,simp,simp*)

**apply** (*rule conjI*)  
**apply** (*rule P3-CASE,simp,simp,simp,simp*)

**apply** (*rule allI*)  
**apply** (*rule impI*)

**apply** (*elim conjE*)  
**apply** (*case-tac E1 x*)

**apply** (*subgoal-tac*  $x \in \text{dom } E1$ )  
**prefer** 2 **apply** *force*  
**apply** (*simp add: dom-def*)

**apply** (*case-tac aa*)

**apply** (*rename-tac p*)

**apply** (*subgoal-tac*  $0 < \text{length assert}$ )  
**prefer** 2 **apply** *simp*  
**apply** (*frule P3-CASE,simp,simp,simp*)

**apply** (*frule P4-CASE,simp,simp*)

**apply** (*subgoal-tac*

$\exists j \ C \ vs. \ h \ p = \text{Some } (j, C, vs) \wedge$   
 $(\exists i < \text{length } \text{alts}. \text{def-extend } E1 \ (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! \ i)))) \ vs$

$\wedge (\text{extend } E1 \ (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! \ i)))) \ vs, E2) \vdash h, k, \text{snd } (\text{alts } ! \ i)$

$\Downarrow(f, n) \ hh, k, v)$

**prefer** 2 **apply** (*rule P1-f-n-CASE*)

**apply** (*simp,simp*)

**apply** (*subgoal-tac*

$\forall i < \text{length } \text{alts}. \text{fv } (\text{snd } (\text{alts } ! \ i)) \subseteq \text{fst } (\text{assert } ! \ i))$

**prefer** 2 **apply** *clarsimp*

**apply** (*subgoal-tac*

$\forall i < \text{length } \text{alts}. \text{fst } (\text{assert } ! \ i) \subseteq \text{dom } (\text{snd } (\text{assert } ! \ i)))$

**prefer** 2 **apply** *clarsimp*

**apply** (*elim exE,elim conjE*)

**apply** (*elim exE, elim conjE*)

**apply** (*rotate-tac 5*)

**apply** (*erule-tac x=i in allE*)

**apply** (*drule mp, simp*)

**apply** (*elim conjE*)

**apply** (*erule-tac x=extend E1 (snd (extractP (fst (alts ! i)))) vs in allE*)

**apply** (*erule-tac x=E2 in allE*)

**apply** (*erule-tac x=h in allE*)

**apply** (*rotate-tac 25*)

**apply** (*erule-tac x=k in allE*)

**apply** (*rotate-tac 25*)

**apply** (*erule-tac x=hh in allE*)

**apply** (*erule-tac x=v in allE*)

**apply** (*drule mp*)

**apply** (*rule conjI,simp*)

**apply** (*rule P2-CASE,assumption+,simp*)

**apply** (*rule conjI*) **apply** *simp*

**apply** (*rule P5-P6-f-n-CASE*)

**apply** (*assumption+,simp,simp,assumption+,simp*)

**apply** (*rule impI*)

**apply** (*elim conjE*)

**apply** (*drule mp*)

**apply** (*rule conjI*) **apply** *simp*

**apply** (*rule P8-CASE*)

**apply** (*assumption+*,*simp*,*simp*,*simp*,*simp*,*force*,*simp*,*assumption+*,*simp*,*assumption+*)

**apply** (*frule P2-CASE*,*assumption+*,*simp*)

**apply** (*simp add: def-extend-def*)

**apply** (*elim conjE*)

**apply** (*rule P7-CASE*)

**apply** (*assumption+*,*simp*,*simp*,*assumption+*,*simp*,*simp*,*simp*,*simp*)

**apply** *simp*

**apply** (*subgoal-tac*

( $\exists i < \text{length } \text{alts}.$

$(E1, E2) \vdash h, k, \text{snd } (\text{alts } ! i) \Downarrow (f, n) \text{ } hh, k, v$

$\wedge \text{fst } (\text{alts } ! i) = \text{ConstP } (\text{LitN int}))$ )

**prefer** 2 **apply** (*rule P1-f-n-CASE-1-1*,*simp*,*assumption+*)

**apply** (*elim exE*, *elim conjE*)+

**apply** (*subgoal-tac* 0 < *length assert*)

**prefer** 2 **apply** *simp*

**apply** (*frule P3-CASE*,*simp*,*simp*,*simp*)

**apply** (*frule P4-CASE*,*simp*,*simp*)

**apply** (*rotate-tac* 5)

**apply** (*erule-tac* *x=i* **in** *allE*)

**apply** (*drule mp*, *simp*)

**apply** (*elim conjE*)

**apply** (*erule-tac* *x=E1* **in** *allE*)

**apply** (*erule-tac* *x=E2* **in** *allE*)

**apply** (*erule-tac* *x=h* **in** *allE*)

**apply** (*rotate-tac* 22)

**apply** (*erule-tac* *x=k* **in** *allE*)

**apply** (*rotate-tac* 22)

**apply** (*erule-tac* *x=hh* **in** *allE*)

**apply** (*erule-tac* *x=v* **in** *allE*)

**apply** (*drule mp*)

**apply** (*rule conjI*,*simp*)

**apply** (rule *P2-CASE-1-1*, assumption+,simp)

**apply** (rule *conjI*) **apply** simp  
**apply** (rule *P5-P6-f-n-CASE-1-1*)  
**apply** (assumption+,simp,assumption+,simp)  
**apply** (rule *impI*)

**apply** (elim *conjE*)  
**apply** (drule *mp*)

**apply** (rule *conjI*)  
**apply** (rule *P8-CASE-1-1*)  
**apply** (assumption+,simp,assumption+,simp)

**apply** (frule *P2-CASE-1-1*,assumption+,simp)  
**apply** (rule *P7-CASEL*,assumption+,force)  
**apply** (simp,simp,simp,simp,simp,simp,simp,simp)

**apply** simp

**apply** (subgoal-tac  
( $\exists i < \text{length } \text{alts}.$   
    ( $E1, E2 \vdash h, k, \text{snd } (\text{alts } ! i) \Downarrow (f, n) \text{ hh}, k, v$   
     $\wedge \text{fst } (\text{alts } ! i) = \text{ConstP } (\text{LitB } \text{bool}))$ )  
**prefer** 2 **apply** (rule *P1-f-n-CASE-1-2*,simp,assumption+)  
**apply** (elim *exE*, elim *conjE*)+

**apply** (subgoal-tac 0 < length assert)  
**prefer** 2 **apply** simp  
**apply** (frule *P3-CASE*,simp,simp,simp)  
**apply** (frule *P4-CASE*,simp,simp)

**apply** (rotate-tac 5)  
**apply** (erule-tac  $x=i$  in *allE*)  
**apply** (drule *mp*, simp)  
**apply** (elim *conjE*)  
**apply** (erule-tac  $x=E1$  in *allE*)  
**apply** (erule-tac  $x=E2$  in *allE*)  
**apply** (erule-tac  $x=h$  in *allE*)  
**apply** (rotate-tac 22)

```

apply (erule-tac x=k in allE)
apply (rotate-tac 22)
apply (erule-tac x=hh in allE)
apply (rotate-tac 22)
apply (erule-tac x=v in allE)

```

```

apply (drule mp)
apply (rule conjI,simp)

```

```

apply (rule P2-CASE-1-1,assumption+,simp)

```

```

apply (rule conjI) apply simp
apply (rule P5-P6-f-n-CASE-1-2)
apply (assumption+,simp,assumption+,simp)
apply (rule impI)

```

```

apply (elim conjE)
apply (drule mp)

```

```

apply (rule conjI)
apply (rule P8-CASE-1-2)
apply (assumption+,simp,assumption+,simp)

```

```

apply (erule P2-CASE-1-1,assumption+,simp)
apply (rule P7-CASEL,assumption+,force)
apply (simp,simp,simp,simp,simp,simp,simp,simp,simp)

```

```

by simp

```

**lemma** *SafeDADepth-CASED*:

```

 $\llbracket \text{length} (\text{map } \text{snd } \text{assert}) > 0; \text{length } \text{assert} = \text{length } \text{alts};$ 
 $\forall i < \text{length } \text{alts}. \forall j. \forall x \in \text{set } (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! i)))). \text{snd } (\text{assert } ! i)$ 
 $x = \text{Some } d'' \longrightarrow j \in \text{RecPos } Ci;$ 
 $\forall i < \text{length } \text{assert}. \text{snd } (\text{alts } ! i) :_f, n \Downarrow \text{fst } (\text{assert} ! i), \text{snd } (\text{assert} ! i) \Downarrow;$ 
 $\forall z \in \text{dom } \Gamma. \Gamma z \neq \text{Some } s'' \longrightarrow (\forall i < \text{length } \text{alts}. z \notin \text{fst } (\text{assert } ! i));$ 
def-nonDisjointUnionEnvList
  (map ( $\lambda(Li, \Gamma i). \text{restrict-neg-map } \Gamma i (\text{insert } x (\text{set } Li))$ ))
  (zip (map (snd  $\circ \text{extractP} \circ \text{fst}$ ) alts) (map snd assert)));
def-disjointUnionEnv

```



$(nonDisjointUnionEnvList ((map (\lambda(Li,\Gamma i). restrict-neg-map \Gamma i (set$   
 $Li \cup \{x\}))))$   
 $(zip (map (snd o extractP o fst) alts) (map snd assert))))$   
 $[x \mapsto d''];$   
 $\forall i < length\ alts. \forall j < length\ alts. i \neq j \longrightarrow (fst (assert ! i) \cap set (snd$   
 $(extractP (fst (alts ! j))))) = \{\};$   
 $L = (\bigcup i < length\ assert. fst (assert!i) - set (snd (extractP (fst (alts ! i)))))$   
 $\cup \{x\};$   
 $wellFormedDepth\ f\ n\ L\ \Gamma\ (CaseD\ (VarE\ x\ a)\ Of\ alts\ a');$   
 $\Gamma = disjointUnionEnv$   
 $(nonDisjointUnionEnvList ((map (\lambda(Li,\Gamma i). restrict-neg-map \Gamma i (set$   
 $Li \cup \{x\}))))$   
 $(zip (map (snd o extractP o fst) alts) (map snd assert))))$   
 $(empty(x \mapsto d'')) \parallel$   
 $\implies CaseD\ (VarE\ x\ a)\ Of\ alts\ a' :_f, n \parallel L, \Gamma \parallel$   
**apply** (*simp* (*no-asm*) *only: SafeDAssDepth-def*)  
**apply** (*simp only: SafeDAssDepth-def*)

**apply** (*rule conjI, simp*)  
**apply** (*rule P4-CASED, simp, simp*)

**apply** (*rule conjI, simp, rule conjI*)  
**apply** (*rule P3-1-CASED, simp, simp, simp*)  
**apply** (*rule P3-2-CASED, simp, simp, simp, simp, simp*)

**apply** (*rule allI*)  
**apply** (*rule impI*)  
**apply** (*elim conjE*)

**apply** (*subgoal-tac 0 < length assert*)  
**prefer 2 apply simp**  
**apply** (*frule P3-1-CASED [where x=x], simp, simp*)  
**apply** (*frule P3-2-CASED [where x=x], simp, simp, simp, simp*)  
**apply** (*subgoal-tac*  
 $\forall i < length\ alts. fv (snd (alts ! i)) \subseteq fst (assert ! i))$   
**prefer 2 apply clarsimp**  
**apply** (*subgoal-tac*  
 $\forall i < length\ alts. fst (assert ! i) \subseteq dom (snd (assert ! i))$ )  
**prefer 2 apply clarsimp**  
**apply** (*frule P4-CASED, simp*)

**apply** (*subgoal-tac*  
 $\exists p\ j\ C\ vs. E1\ x = Some\ (Loc\ p) \wedge h\ p = Some\ (j, C, vs) \wedge$   
 $(\exists i < length\ alts.$   
 $def-extend\ E1\ (snd\ (extractP\ (fst\ (alts\ !\ i))))\ vs$   
 $\wedge (extend\ E1\ (snd\ (extractP\ (fst\ (alts\ !\ i))))\ vs, E2) \vdash$

```

      h(p := None), k, snd (alts ! i) ↓(f,n) hh,k , v))
prefer 2 apply (rule P1-f-n-CASED,simp)

apply (elim exE,elim conjE)
apply (elim exE, elim conjE)
apply (rotate-tac 3)
apply (erule-tac x=i in allE)
apply (drule mp, simp)
apply (elim conjE)
apply (erule-tac x=extend E1 (snd (extractP (fst (alts ! i)))) vs in allE)
apply (erule-tac x=E2 in allE)
apply (erule-tac x=h(p:=None) in allE)
apply (rotate-tac 25)
apply (erule-tac x=k in allE)
apply (rotate-tac 25)
apply (erule-tac x=hh in allE)
apply (erule-tac x=v in allE)


apply (drule mp)
apply (rule conjI,simp)


apply (frule-tac vs=vs and ?E1.0=E1 in P2-CASED)
apply (assumption+,simp,clarsimp,force,simp)
apply (simp add: def-extend-def,force,assumption+,simp,simp)
apply (simp add: def-extend-def,simp,clarsimp)


apply (rule conjI,simp)
apply (rule P5-P6-f-n-CASED)
apply (assumption+,simp,simp,assumption+,force)
apply (rule impI)


apply (elim conjE)
apply (drule mp)


apply (rule conjI)
apply (simp,frule P5-P6-f-n-CASED)
apply (simp,assumption+)
apply (simp only: def-extend-def, elim conjE)
apply (rule P8-CASED)
apply (assumption+,simp,assumption+,simp,simp,assumption+)


apply (frule-tac vs=vs and ?E1.0=E1 in P2-CASED)
apply (assumption+,simp,clarsimp,force,simp)

```

```

apply (simp add: def-extend-def,force,assumption+,simp,simp)
apply (simp add: def-extend-def) apply simp apply simp
apply (frule P5-P6-f-n-CASED)
apply (simp,simp,assumption+)

```

```

apply (simp add: def-extend-def)
apply (elim conjE)
apply (rule P7-CASED)
apply (simp,assumption+,simp,simp,simp,assumption+,simp)
apply (assumption+,simp,assumption+,simp)

```

**by** simp

```

declare nonDisjointUnionSafeEnvList.simps [simp del]
declare fv-fvs-fvs'-fvAlts-fvTup-fvAlts'-fvTup'.simps [simp add]
declare atom.simps [simp del]

```

```

lemma lemma-19-aux [rule-format]:
   $\models \Sigma m$ 
   $\longrightarrow \Sigma m \ g = \text{Some } ms$ 
   $\longrightarrow (\text{bodyAPP } \Sigma f \ g) : \{ \text{set } (\text{varsAPP } \Sigma f \ g) , [\text{varsAPP } \Sigma f \ g \mapsto ms] \}$ 
apply (rule impI)
apply (erule ValidGlobalMarkEnv.induct,simp-all)
apply (rule impI)+
by simp

```

```

lemma equiv-SafeDAss-all-n-SafeDAssDepth:
   $e : \{ L , \Gamma \} \Longrightarrow \forall n. \text{SafeDAssDepth } e \ f \ n \ L \ \Gamma$ 
apply (simp only: SafeDAss-def)
apply (simp only: SafeDAssDepth-def)
apply clarsimp
apply (simp only: SafeBoundSem-def)
apply (simp add: Let-def)
apply (elim exE)
apply (elim conjE)
apply (erule-tac x=E1 in allE)
apply (erule-tac x=E2 in allE)

```

```

apply (erule-tac x=h in allE)
apply (erule-tac x=k in allE)
apply (erule-tac x=td in allE)
apply (erule-tac x=hh in allE)
apply (erule-tac x=v in allE)
apply (erule-tac td=td in eqSemDepthRA)
apply (elim exE)
apply (erule mp,force)
by simp

```

```

lemma map-le-x-in-dom-m2:
   $\llbracket m1 \subseteq_m m2; x \in \text{dom } m1; m1 \ x \neq \text{Some } y \rrbracket$ 
   $\implies x \in \text{dom } m2 \wedge m2 \ x \neq \text{Some } y$ 
apply (rule conjI)
apply (simp add: map-le-def,force)
by (simp add: map-le-def)

```

```

lemma shareRec-map-le-Γ:
   $\llbracket [xs \mapsto] ms \subseteq_m \Gamma g; \text{shareRec } (\text{set } xs) [xs \mapsto] ms \ (E1, E2) \ (h, k) \ (hh, k);$ 
   $\text{length } xs = \text{length } ms \rrbracket$ 
   $\implies \text{shareRec } (\text{set } xs) \ \Gamma g \ (E1, E2) \ (h, k) \ (hh, k)$ 
apply (simp add: shareRec-def)
apply (rule conjI)

```

```

apply (rule ballI,rule impI)
apply (elim conjE)
apply (erule-tac x=x in ballE)
prefer 2 apply simp
apply (elim bexE, elim conjE)
apply (erule mp)
apply (erule-tac x=z in bexI,simp)
apply (simp add: map-le-def,assumption+)
apply (elim conjE)
apply (simp add: map-le-def)
apply (subgoal-tac x ∈ dom [xs ↦] ms)
prefer 2 apply simp
apply (erule-tac x=x in ballE,force,simp)

```

```

apply (rule ballI, rule impI)
apply (elim conjE)
apply (erule-tac x=x in ballE)+
apply simp
apply (elim conjE)
apply (rule map-le-x-in-dom-m2,assumption+,simp,assumption)

```

**apply** *simp*  
**by** *simp*

**lemma** *RSet-subseteq-RSet-map-le-Γ*:

$\llbracket [xs \mapsto] ms \rrbracket \subseteq_m \Gamma g; \text{length } xs = \text{length } ms \rrbracket$   
 $\implies RSet \ (set \ xs) \ [xs \mapsto] \ ms] \ (E1, E2) \ (h, k) \subseteq RSet \ (set \ xs) \ \Gamma g \ (E1, E2)$   
 $(h, k)$   
**apply** (*rule subsetI*)  
**apply** (*simp add: RSet-def*)  
**apply** (*elim conjE, elim bexE, elim conjE*)  
**apply** (*rule-tac x=z in bexI*)  
**apply** (*simp add: map-le-def*)  
**by** *simp*

**lemma** *SSet-subseteq-SSet-map-le-Γ*:

$\llbracket [xs \mapsto] ms \rrbracket \subseteq_m \Gamma g; \text{length } xs = \text{length } ms \rrbracket$   
 $\implies SSet \ (set \ xs) \ [xs \mapsto] \ ms] \ (E1, E2) \ (h, k) \subseteq SSet \ (set \ xs) \ \Gamma g \ (E1, E2) \ (h,$   
 $k)$   
**apply** (*rule subsetI*)  
**apply** (*simp add: SSet-def*)  
**apply** (*simp add: Let-def*)  
**apply** (*elim exE, elim conjE*)  
**apply** (*rule-tac x=xa in exI*)  
**by** (*simp add: map-le-def*)

**lemma** *SSet-RSet-map-le-Γ*:

$\llbracket [xs \mapsto] ms \rrbracket \subseteq_m \Gamma g; \text{length } xs = \text{length } ms;$   
 $SSet \ (set \ xs) \ \Gamma g \ (E1, E2) \ (h, k) \cap RSet \ (set \ xs) \ \Gamma g \ (E1, E2) \ (h, k) = \{\}$   
 $\implies SSet \ (set \ xs) \ [xs \mapsto] \ ms] \ (E1, E2) \ (h, k) \cap RSet \ (set \ xs) \ [xs \mapsto] \ ms] \ (E1,$   
 $E2) \ (h, k) = \{\}$   
**apply** (*frule SSet-subseteq-SSet-map-le-Γ, assumption+*)  
**apply** (*frule RSet-subseteq-RSet-map-le-Γ, assumption+*)  
**by** *blast*

**lemma** *SafeDAssDepth-map-le-Γ*:

$\llbracket Lg = set \ xs; [xs \mapsto] ms \rrbracket \subseteq_m \Gamma g; \Sigma f \ g = Some \ (xs, rs, eg); \text{length } xs = \text{length}$   
 $ms;$   
 $bodyAPP \ \Sigma f \ g :_f, \ n \ \{\ set \ (varsAPP \ \Sigma f \ g) , [varsAPP \ \Sigma f \ g \mapsto] \ ms] \ \}$   
 $\implies eg :_f, \ n \ \{\ Lg, \ \Gamma g \}$   
**apply** (*simp add: bodyAPP-def*)  
**apply** (*simp add: varsAPP-def*)  
**apply** (*simp add: SafeDAssDepth-def*)  
**apply** (*subgoal-tac set xs  $\subseteq$  dom [xs  $\mapsto$ ] ms]*)  
**prefer** 2 **apply** *simp*  
**apply** (*rule conjI*)  
**apply** (*frule map-le-implies-dom-le, simp*)  
**apply** (*rule allI*)**+**

**apply** (*rule impI*, *elim conjE*)  
**apply** (*erule-tac x=E1 in allE*)  
**apply** (*erule-tac x=E2 in allE*)  
**apply** (*erule-tac x=h in allE*)  
**apply** (*erule-tac x=k in allE*)  
**apply** (*erule-tac x=hh in allE*)  
**apply** (*erule-tac x=v in allE*)  
**apply** (*drule mp*)  
**apply** (*rule conjI,simp*)  
**apply** (*frule map-le-implies-dom-le,simp*)  
**apply** (*elim conjE*)  
**apply** (*rule conjI*)  
**apply** (*rule shareRec-map-le-Γ,assumption+*)  
**apply** (*rule impI*)+  
**apply** (*drule mp*)  
**apply** (*rule conjI,simp*)  
**apply** (*rule SSet-RSet-map-le-Γ, assumption+,simp*)  
**by** *simp*

**lemma** *SafeDAss-map-le-Γ*:

$\llbracket Lg = \text{set } xs; [xs \mapsto] ms \rrbracket \subseteq_m \Gamma g; \Sigma f g = \text{Some } (xs, rs, eg); \text{length } xs = \text{length } ms;$   
 $\text{bodyAPP } \Sigma f g : \llbracket \text{set } (varsAPP \Sigma f g), [varsAPP \Sigma f g \mapsto] ms \rrbracket$   
 $\implies eg :_f, n \llbracket Lg, \Gamma g \rrbracket$   
**apply** (*frule-tac f=f in equiv-SafeDAss-all-n-SafeDAssDepth*)  
**apply** (*erule-tac x=n in allE*)  
**by** (*rule SafeDAssDepth-map-le-Γ,simp-all*)

**lemma** *lemma-19 [rule-format]*:

$\models_f, n \Sigma m$   
 $\longrightarrow \Sigma f g = \text{Some } (xs, rs, eg)$   
 $\longrightarrow \Sigma m g = \text{Some } ms$   
 $\longrightarrow g \neq f$   
 $\longrightarrow \text{length } xs = \text{length } ms$   
 $\longrightarrow Lg = \text{set } xs$   
 $\longrightarrow [xs \mapsto] ms \rrbracket \subseteq_m \Gamma g$   
 $\longrightarrow eg :_f, n \llbracket Lg, \Gamma g \rrbracket$   
**apply** (*rule impI*)  
**apply** (*erule ValidGlobalMarkEnvDepth.induct*)

**apply** (*rule impI*)+  
**apply** (*frule lemma-19-aux,force*)  
**apply** (*rule SafeDAss-map-le-Γ,assumption+*)

**apply** (*rule impI*)+

```

apply (frule lemma-19-aux,force)
apply (rule SafeDAss-map-le-Γ,assumption+)

```

```

apply (rule impI)+
apply (frule lemma-19-aux,force)
apply (rule SafeDAss-map-le-Γ,assumption+)

```

```

apply (case-tac ga=g,simp-all)
apply (rule impI)+
apply (rule SafeDAss-map-le-Γ,simp-all)
done

```

```

lemma lemma-20 [rule-format]:
   $\models_f, n \Sigma m$ 
 $\longrightarrow \Sigma f f = \text{Some } (xs, rs, ef)$ 
 $\longrightarrow \Sigma m f = \text{Some } ms$ 
 $\longrightarrow \text{length } xs = \text{length } ms$ 
 $\longrightarrow Lf = \text{set } xs$ 
 $\longrightarrow [xs \mapsto] ms \subseteq_m \Gamma f$ 
 $\longrightarrow n = \text{Suc } n'$ 
 $\longrightarrow ef : f, n' \Vdash Lf, \Gamma f \Vdash$ 
apply (rule impI)
apply (erule ValidGlobalMarkEnvDepth.induct)

```

```

apply (rule impI)+
apply (frule lemma-19-aux,force)
apply (rule SafeDAss-map-le-Γ,assumption+)

```

```

apply simp

```

```

apply (rule impI)+
apply simp
apply (rule SafeDAssDepth-map-le-Γ)
apply (simp,simp,simp,simp,simp)

```

```

apply simp
done

```

```

lemma map-upds-equals-map-of-distinct-xs:

```

$\llbracket \text{distinct } xs; \text{length } xs = \text{length } ms \rrbracket$   
 $\implies [xs \mapsto ms] = \text{map-of } (\text{zip } xs \ ms)$   
**by** (*induct xs ms rule: list-induct2',simp-all*)

**lemma** *SafeDADepth-APP:*

$\llbracket \text{length } xs = \text{length } ms; \text{primops } g = \text{None};$   
 $\forall a \in \text{set } as. \text{atom } a; \text{length } as = \text{length } ms;$   
 $\Sigma f \ g = \text{Some } (xs, rs, ef);$   
 $L = \text{fvs}' \ as;$   
 $\Gamma 0 = \text{nonDisjointUnionSafeEnvList } (\text{maps-of } (\text{zip } (\text{map } \text{atom2var } as) \ ms));$   
 $\text{def-nonDisjointUnionSafeEnvList } (\text{maps-of } (\text{zip } (\text{map } \text{atom2var } as) \ ms));$   
 $\Gamma 0 \subseteq_m \Gamma;$   
 $\Sigma m \ g = \text{Some } ms;$   
 $\text{wellFormedDepth } f \ n \ L \ \Gamma \ (\text{AppE } g \ as \ rs' \ a);$   
 $\models_f, n \ \Sigma m \rrbracket$   
 $\implies \text{AppE } g \ as \ rs' \ a :_f, n \ \llbracket L, \Gamma \rrbracket$   
**apply** (*case-tac g≠f*)

**apply** (*frule-tac  $\Gamma g = [xs \mapsto ms]$  in lemma-19*)  
**apply** (*assumption+,simp,simp add: map-le-def*)

**apply** (*simp only: SafeDAssDepth-def*)  
**apply** (*elim conjE*)

**apply** (*rule conjI,simp*)

**apply** (*rule conjI*)  
**apply** (*rule P3-APP,simp,simp,simp*)

**apply** (*rule allI*)  
**apply** (*rule impI*)  
**apply** (*elim conjE*)

**apply** (*frule P1-f-n-APP-2,simp,force,simp,force*)  
**apply** (*elim exE*)  
**apply** (*erule-tac  $x = \text{map-of } (\text{zip } xs \ (\text{map } (\text{atom2val } E1) \ as))$  in allE*)  
**apply** (*erule-tac  $x = \text{map-of } (\text{zip } rs \ (\text{map } (\text{the } \circ E2) \ rs'))$  (self  $\mapsto$   
 $\text{Suc } k$ ) in allE*)  
**apply** (*erule-tac  $x = h$  in allE*)  
**apply** (*erule-tac  $x = \text{Suc } k$  in allE*)



```

apply (erule-tac x=h' in allE)
apply (erule-tac x=v in allE)
apply (elim conjE)
apply (drule mp)
apply (rule conjI,simp)

```

```

apply (simp)

```

```

apply (elim conjE)

```

```

apply (rule conjI)
apply (frule P3-APP,simp,simp)
apply (rule P5-P6-f-n-APP-2)
apply (simp,assumption+,simp,assumption+)

```

```

apply (rule impI)
apply (elim conjE)

```

```

apply (subst (asm) map-upds-equals-map-of-distinct-xs,assumption+)+
apply (frule P3-APP,simp,simp)
apply (subgoal-tac length xs=length as)
  prefer 2 apply simp
apply (frule P7-APP-ef,assumption+)

```

```

apply (frule P3-APP,simp,simp)
apply (frule P8-APP-ef,assumption+)

```

```

apply (drule mp)
apply (rule conjI,simp)
apply simp

```

```

apply (rule P9-APP,assumption+)

```

```

apply simp
apply (case-tac n)

```

```

  apply (simp only: SafeDAssDepth-def)

```

```

apply (rule conjI,simp)

apply (rule conjI)
apply (rule P3-APP,simp,simp,simp)

apply (rule allI)+
apply (rule impI)
apply (elim conjE)
apply (frule P1-f-n-APP,assumption+,simp)

apply (frule-tac  $\Gamma f=[xs \mapsto] ms$  in lemma-20)
apply (assumption+,simp,simp,simp)
apply simp

apply (simp only: SafeDAssDepth-def)
apply (elim conjE)

apply (rule conjI,simp)

apply (rule conjI)
apply (rule P3-APP,simp,simp,simp)

apply (rule allI)+
apply (rule impI)
apply (elim conjE)

apply (frule P1-f-n-ge-0-APP,simp,force)
apply (elim exE)
apply (erule-tac x=map-of (zip xs (map (atom2val E1) as)) in allE)
apply (erule-tac x=map-of (zip rs (map (the  $\circ$  E2) rs'))(self  $\mapsto$ 
  Suc k) in allE)
apply (erule-tac x=h in allE)
apply (erule-tac x=Suc k in allE)
apply (erule-tac x=h' in allE)
apply (erule-tac x=v in allE)
apply (elim conjE)
apply (drule mp)
apply (rule conjI,simp)

apply (simp)

```

```
apply (elim conjE)
```

```
apply (rule conjI)
apply (frule P3-APP,simp,simp)
apply (rule P5-P6-f-n-APP,assumption+)
```

```
apply (rule impI)
apply (elim conjE)
```

```
apply (subst (asm) map-upds-equals-map-of-distinct-xs,assumption+)+
apply (frule P3-APP,simp,simp)
apply (frule P7-APP-ef,assumption+)
```

```
apply (frule P3-APP,simp,simp)
apply (frule P8-APP-ef,assumption+)
```

```
apply (drule mp)
apply (rule conjI,simp)
apply simp
```

```
apply (rule P9-APP,assumption+)
done
```

```
end
```

## 22 Proof rules for explicit deallocation

```
theory ProofRules
imports SafeDAssDepth
begin
```

```
consts RecPos :: string  $\Rightarrow$  nat set
```

```
inductive
```

```
ProofRulesED :: [unit Exp, MarkEnv, string, string set, TypeEnvironment]  $\Rightarrow$ 
bool
```

```
( -, -  $\vdash$  - ' ( -, - ' ) [71,71,71,71,71] 70)
```

```
where
```

$litInt : ConstE (LitN i) a, \Sigma m \vdash_f (\{\}, empty)$   
 $| litBool : ConstE (LitB b) a, \Sigma m \vdash_f (\{\}, empty)$   
 $| var1 : \llbracket \Gamma x = Some\ s' \rrbracket$   
 $\quad \implies VarE\ x\ a, \Sigma m \vdash_f (\{x\}, \Gamma)$   
 $| var2 : \llbracket x \in dom\ \Gamma; wellFormed\ \{x\}\ \Gamma\ (CopyE\ x\ r\ d) \rrbracket$   
 $\quad \implies CopyE\ x\ r\ d, \Sigma m \vdash_f (\{x\}, \Gamma)$   
 $| var3 : \llbracket \Gamma x = Some\ d''; wellFormed\ \{x\}\ \Gamma\ (ReuseE\ x\ a) \rrbracket$   
 $\quad \implies ReuseE\ x\ a, \Sigma m \vdash_f (\{x\}, \Gamma)$   
 $| let1 : \llbracket e1, \Sigma m \vdash_f (L1, \Gamma1);$   
 $\quad e2, \Sigma m \vdash_f (L2, \Gamma2');$   
 $\quad \Gamma2' = disjointUnionEnv\ \Gamma2\ (empty(x1 \mapsto s''));$   
 $\quad def-disjointUnionEnv\ \Gamma2\ (empty(x1 \mapsto s''));$   
 $\quad def-pp\ \Gamma1\ \Gamma2\ L2;$   
 $\quad x1 \notin L1;$   
 $\quad L = L1 \cup (L2 - \{x1\});$   
 $\quad \Gamma = pp\ \Gamma1\ \Gamma2\ L2;$   
 $\quad \forall\ C\ as\ r\ a'.\ e1 \neq ConstrE\ C\ as\ r\ a' \rrbracket$   
 $\quad \implies Let\ x1 = e1\ In\ e2\ a, \Sigma m \vdash_f (L, \Gamma)$   
 $| let1c : \llbracket L1 = set\ (map\ atom2var\ as); \forall a \in set\ as.\ atom\ a;$   
 $\quad \Gamma1 = map-of\ (zip\ (map\ atom2var\ as)\ (replicate\ (length\ as)\ s''));$   
 $\quad x1 \notin L1;$   
 $\quad e2, \Sigma m \vdash_f (L2, disjointUnionEnv\ \Gamma2\ (empty(x1 \mapsto s'')));$   
 $\quad def-disjointUnionEnv\ \Gamma2\ (empty(x1 \mapsto s''));$   
 $\quad def-pp\ \Gamma1\ \Gamma2\ L2;$   
 $\quad L = L1 \cup (L2 - \{x1\});$   
 $\quad \Gamma = pp\ \Gamma1\ \Gamma2\ L2 \rrbracket$   
 $\quad \implies Let\ x1 = ConstrE\ C\ as\ r\ a'\ In\ e2\ a, \Sigma m \vdash_f (L, \Gamma)$   
 $| let2 : \llbracket \forall\ C\ as\ r\ a'.\ e1 \neq ConstrE\ C\ as\ r\ a';$   
 $\quad e1, \Sigma m \vdash_f (L1, \Gamma1);$   
 $\quad e2, \Sigma m \vdash_f (L2, disjointUnionEnv\ \Gamma2\ (empty(x1 \mapsto d'')));$   
 $\quad def-disjointUnionEnv\ \Gamma2\ (empty(x1 \mapsto d''));$   
 $\quad def-pp\ \Gamma1\ \Gamma2\ L2;$   
 $\quad x1 \notin L1 \rrbracket$   
 $\quad \implies Let\ x1 = e1\ In\ e2\ a, \Sigma m \vdash_f ((L1 \cup (L2 - \{x1\})), (pp\ \Gamma1\ \Gamma2\ L2))$   
 $| let2c : \llbracket L1 = set\ (map\ atom2var\ as); \forall a \in set\ as.\ atom\ a;$   
 $\quad \Gamma1 = map-of\ (zip\ (map\ atom2var\ as)\ (replicate\ (length\ as)\ s''));$   
 $\quad x1 \notin L1;$   
 $\quad e2, \Sigma m \vdash_f (L2, disjointUnionEnv\ \Gamma2\ (empty(x1 \mapsto d'')));$   
 $\quad def-disjointUnionEnv\ \Gamma2\ (empty(x1 \mapsto d''));$   
 $\quad def-pp\ \Gamma1\ \Gamma2\ L2;$   
 $\quad L = L1 \cup (L2 - \{x1\});$

$$\begin{aligned}
& \Gamma = pp \ \Gamma 1 \ \Gamma 2 \ L 2 \ ] \\
& \implies Let \ x1 = ConstrE \ C \ as \ r \ a' \ In \ e2 \ a, \ \Sigma m \vdash_f ( \ L, \ \Gamma \ ) \\
| \ case1 : [ & def-nonDisjointUnionEnvList (map snd assert); \\
& length (map snd assert) > 0; length assert = length alts; \\
& \forall i < length alts. \forall j < length alts. i \neq j \longrightarrow (fst (assert ! i) \cap set \\
& (snd (extractP (fst (alts ! j))))) = \{\}; \\
& \forall i < length alts. \forall x \in set (snd (extractP (fst (alts ! i)))). snd (assert \\
& ! i) \ x \neq Some \ d''; \\
& \forall i < length assert. snd (alts ! i), \Sigma m \vdash_f (fst (assert!i), snd (assert!i)); \\
& \forall i < length alts. x \in fst (assert ! i) \wedge x \notin set (snd (extractP (fst \\
& (alts ! i)))); \\
& x \in dom \ \Gamma; \\
& wellFormed \ L \ \Gamma \ (Case \ (VarE \ x \ a) \ Of \ alts \ a'); \\
& L = (\bigcup i < length assert. fst (assert!i) - set (snd (extractP (fst (alts \\
& ! i))))) \cup \{x\}; \\
& \Gamma = nonDisjointUnionEnvList (map snd assert)] \\
& \implies Case \ (VarE \ x \ a) \ Of \ alts \ a', \ \Sigma m \vdash_f ( \ L, \ \Gamma \ ) \\
| \ case2 : [ & length (map snd assert) > 0; length assert = length alts; \\
& \forall i < length alts. \forall j. \forall x \in set (snd (extractP (fst (alts ! i)))). snd \\
& (assert ! i) \ x = Some \ d'' \longrightarrow j \in RecPos \ Ci; \\
& \forall i < length assert. snd (alts ! i), \Sigma m \vdash_f (fst (assert!i), snd (assert!i) \\
& ); \\
& \forall z \in dom \ \Gamma. \Gamma \ z \neq Some \ s'' \longrightarrow (\forall i < length alts. z \notin fst (assert ! \\
& i)); \\
& def-nonDisjointUnionEnvList \\
& (map (\lambda(Li, \Gamma i). restrict-neg-map \Gamma i (insert x (set Li))) \\
& (zip (map (snd \circ extractP \circ fst) alts) (map snd assert))); \\
& def-disjointUnionEnv \\
& (nonDisjointUnionEnvList ((map (\lambda(Li, \Gamma i). restrict-neg-map \Gamma i (set \\
& Li \cup \{x\}))) \\
& (zip (map (snd \circ extractP \circ fst) alts) (map snd assert)))) \\
& [x \mapsto d'']; \\
& \forall i < length alts. \forall j < length alts. i \neq j \longrightarrow (fst (assert ! i) \cap set \\
& (snd (extractP (fst (alts ! j))))) = \{\}; \\
& L = (\bigcup i < length assert. fst (assert!i) - set (snd (extractP (fst (alts \\
& ! i))))) \cup \{x\}; \\
& wellFormed \ L \ \Gamma \ (CaseD \ (VarE \ x \ a) \ Of \ alts \ a'); \\
& \Gamma = disjointUnionEnv \\
& (nonDisjointUnionEnvList ((map (\lambda(Li, \Gamma i). restrict-neg-map \Gamma i (set \\
& Li \cup \{x\}))) \\
& (zip (map (snd \circ extractP \circ fst) alts) (map snd assert)))) \\
& (empty(x \mapsto d'')) ] \\
& \implies CaseD \ (VarE \ x \ a) \ Of \ alts \ a', \ \Sigma m \vdash_f ( \ L, \ \Gamma \ ) \\
| \ app-primop : [ & atom \ a1; atom \ a2; primops \ f = Some \ oper; \\
& L = \{atom2var \ a1, atom2var \ a2\}; \\
& \Gamma 0 = [atom2var \ a1 \mapsto s'', atom2var \ a2 \mapsto s''];
\end{aligned}$$

$$\begin{aligned}
& \Gamma 0 \subseteq_m \Gamma \parallel \\
& \implies AppE f [a1, a2] \parallel a, \Sigma m \vdash_f (L, \Gamma)
\end{aligned}$$

| *app* :  $\parallel$  *length xs = length ms; primops g = None;*  
 $\forall a \in \text{set } as. \text{atom } a; \text{length } as = \text{length } ms;$   
 $\Sigma f g = \text{Some } (xs, rs, ef);$   
 $L = fvs' as;$   
 $\Sigma m g = \text{Some } ms;$   
 $\Gamma 0 = \text{nonDisjointUnionSafeEnvList } (\text{maps-of } (\text{zip } (\text{map } \text{atom2var}$   
*as*) *ms*));

$$\begin{aligned}
& \text{def-nonDisjointUnionSafeEnvList } (\text{maps-of } (\text{zip } (\text{map } \text{atom2var } as) \\
& ms)); \\
& \Gamma 0 \subseteq_m \Gamma; \\
& \text{wellFormed } L \Gamma (AppE g as rs' a) \parallel \\
& \implies AppE g as rs' a, \Sigma m \vdash_f (L, \Gamma)
\end{aligned}$$

| *rec* :  $\parallel$   $\Sigma f f = \text{Some } (xs, rs, ef);$   
 $f \notin \text{dom } \Sigma m;$   
 $Lf = \text{set } xs;$   
 $\Gamma f = \text{empty } (xs \mapsto ms);$   
 $ef, \Sigma m(f \mapsto ms) \vdash_f (Lf, \Gamma f) \parallel$   
 $\implies ef, \Sigma m \vdash_f (Lf, \Gamma f)$

**lemma** *equiv-all-n-SafeDAssDepth-SafeDAss:*  
 $\forall n. \text{SafeDAssDepth } e f n L \Gamma \implies e : \{ L, \Gamma \}$   
**apply** (*simp only: SafeDAss-def*)  
**apply** (*simp only: SafeDAssDepth-def*)  
**apply** (*rule conjI, simp*) +  
**apply** (*rule allI*) +  
**apply** (*rule impI*)  
**apply** (*elim conjE*)  
**apply** (*frule-tac f=f in eqSemRADepth*)  
**apply** (*simp only: SafeBoundSem-def*)  
**apply** (*elim exE*)  
**apply** (*rename-tac n*)  
**apply** (*erule-tac x=n in allE*)  
**apply** (*elim conjE*)  
**apply** (*erule-tac x=E1 in allE*)  
**apply** (*erule-tac x=E2 in allE*)  
**apply** (*erule-tac x=h in allE*)  
**apply** (*erule-tac x=k in allE*)  
**apply** (*erule-tac x=hh in allE*)  
**apply** (*erule-tac x=v in allE*)  
**apply** (*simp add: Let-def*)

**apply** (*drule mp,force*)  
**by** *simp*

**lemma** *equiv-SafeDAss-all-n-SafeDAssDepth*:  
 $e : \llbracket L, \Gamma \rrbracket \implies \forall n. \text{SafeDAssDepth } e \text{ } f \text{ } n \text{ } L \text{ } \Gamma$   
**apply** (*simp only: SafeDAss-def*)  
**apply** (*simp only: SafeDAssDepth-def*)  
**apply** *clarsimp*  
**apply** (*simp only: SafeBoundSem-def*)  
**apply** (*simp add: Let-def*)  
**apply** (*elim exE*)  
**apply** (*elim conjE*)  
**apply** (*erule-tac x=E1 in allE*)  
**apply** (*erule-tac x=E2 in allE*)  
**apply** (*erule-tac x=h in allE*)  
**apply** (*erule-tac x=k in allE*)  
**apply** (*erule-tac x=td in allE*)  
**apply** (*erule-tac x=hh in allE*)  
**apply** (*erule-tac x=v in allE*)  
**apply** (*erule-tac td=td in eqSemDepthRA*)  
**apply** (*elim exE*)  
**apply** (*drule mp,force*)  
**by** *simp*

**lemma** *lemma-5*:  
 $\forall n. \text{SafeDAssDepth } e \text{ } f \text{ } n \text{ } L \text{ } \Gamma \equiv e : \llbracket L, \Gamma \rrbracket$   
**apply** (*rule eq-reflection*)  
**apply** (*rule iffI*)

**apply** (*rule equiv-all-n-SafeDAssDepth-SafeDAss,force*)

**by** (*rule equiv-SafeDAss-all-n-SafeDAssDepth,force*)

**declare** *fun-upd-apply* [*simp del*]

**lemma** *imp-ValidGlobalMarkEnv-all-n-ValidGlobalMarkEnvDepth*:  
 $\text{ValidGlobalMarkEnv } \Sigma \implies \forall n. \models_{f,n} \Sigma$   
**apply** (*erule ValidGlobalMarkEnv.induct,simp-all*)  
**apply** (*rule allI*)  
**apply** (*rule ValidGlobalMarkEnvDepth.base*)  
**apply** (*rule ValidGlobalMarkEnv.base*)

```

apply simp
apply (rule allI)
apply (case-tac fa=f, simp)
apply (induct-tac n)
  apply (rule ValidGlobalMarkEnvDepth.depth0, simp, simp)
apply (rule ValidGlobalMarkEnvDepth.step)
apply (simp, simp, simp, simp, simp)
apply (frule-tac f=f in equiv-SafeDAss-all-n-SafeDAssDepth, simp)
apply (rule ValidGlobalMarkEnvDepth.g)
apply (simp, simp, simp, simp, simp, simp)
done

```

```

lemma imp-ValidDepth-n-Sigma-Valid-Sigma [rule-format]:
   $\models_f, n \Sigma m$ 
   $\longrightarrow f \notin \text{dom } \Sigma m$ 
   $\longrightarrow \text{ValidGlobalMarkEnv } \Sigma m$ 
apply (rule impI)
apply (erule ValidGlobalMarkEnvDepth.induct) apply (simp-all)
  apply (simp add: fun-upd-apply add: dom-def)
  apply (simp add: fun-upd-apply add: dom-def)
apply (rule impI)
apply (erule mp)
  apply (simp add: fun-upd-apply add: dom-def)
by (rule ValidGlobalMarkEnv.step, simp-all)

```

```

lemma imp-f-notin-Sigma-ValidDepth-n-Sigma-Valid-Sigma:
   $\llbracket f \notin \text{dom } \Sigma m; \forall n. \models_f, n \Sigma m \rrbracket$ 
   $\implies \text{ValidGlobalMarkEnv } \Sigma m$ 
apply (erule-tac x=n in allE)
by (rule imp-ValidDepth-n-Sigma-Valid-Sigma, assumption+)

```

```

lemma Theorem-4-aux [rule-format]:
   $\models_f, n \Sigma m$ 
   $\longrightarrow n = \text{Suc } n'$ 
   $\longrightarrow f \in \text{dom } \Sigma m$ 
   $\longrightarrow (\text{bodyAPP } \Sigma f f) :_f, n' \Vdash \text{set } (\text{varsAPP } \Sigma f f), [(\text{varsAPP } \Sigma f f) \mapsto] \text{ the } (\Sigma m f) \rrbracket$ 
apply (rule impI)
apply (erule ValidGlobalMarkEnvDepth.induct, simp-all)
apply (rule impI) +
apply (subgoal-tac the (( $\Sigma m(f \mapsto ms)$ ) f) = ms, simp)
apply (simp add: fun-upd-apply add: dom-def)
apply (rule impI) +

```



**apply** (*case-tac*  $g=f, \text{simp-all}$ )  
**apply** (*subgoal-tac*  $f \in \text{dom } \Sigma m, \text{simp}$ )  
**prefer** 2 **apply** (*simp*  $\text{add: fun-upd-apply add: dom-def}$ )  
**apply** (*subgoal-tac*  $\text{the } ((\Sigma m(g \mapsto ms)) f) = \text{the } (\Sigma m f), \text{simp}$ )  
**by** (*simp*  $\text{add: fun-upd-apply add: dom-def}$ )

**lemma** *Theorem-4:*

$\llbracket \forall n > 0. \models_f, n \Sigma m; f \in \text{dom } \Sigma m \rrbracket$   
 $\implies \forall n. (\text{bodyAPP } \Sigma f f) : \llbracket \text{set } (\text{varsAPP } \Sigma f f) , [(\text{varsAPP } \Sigma f f) [\mapsto]$   
 $\text{the } (\Sigma m f)] \rrbracket$   
**apply** (*rule allI*)  
**apply** (*rule-tac*  $n=\text{Suc } n$  **in** *Theorem-4-aux*)  
**by** *simp-all*

**lemma** *Theorem-5-aux [rule-format]:*

$\models_f, n \Sigma m$   
 $\longrightarrow n = \text{Suc } n'$   
 $\longrightarrow f \in \text{dom } \Sigma m$   
 $\longrightarrow \text{bodyAPP } \Sigma f f : \llbracket \text{set } (\text{varsAPP } \Sigma f f) , [\text{varsAPP } \Sigma f f [\mapsto] \text{the } (\Sigma m f)] \rrbracket$   
 $\longrightarrow \models \Sigma m$   
**apply** (*rule impI*)  
**apply** (*erule ValidGlobalMarkEnvDepth.induct, simp-all*)  
**apply** (*rule impI*) +  
**apply** (*rule ValidGlobalMarkEnv.step*)  
**apply** (*simp, simp, simp, simp, simp*)  
**apply** (*subgoal-tac*  $\text{the } ((\Sigma m(f \mapsto ms)) f) = ms, \text{simp}$ )  
**apply** (*simp*  $\text{add: fun-upd-apply add: dom-def}$ )  
**apply** (*rule impI*) +  
**apply** (*case-tac*  $g=f, \text{simp-all}$ )  
**apply** (*rule ValidGlobalMarkEnv.step, simp-all*)  
**apply** (*subgoal-tac*  $f \in \text{dom } \Sigma m, \text{simp}$ )  
**prefer** 2 **apply** (*simp*  $\text{add: fun-upd-apply add: dom-def}$ )  
**apply** (*subgoal-tac*  $\text{the } ((\Sigma m(g \mapsto ms)) f) = \text{the } (\Sigma m f), \text{simp}$ )  
**by** (*simp*  $\text{add: fun-upd-apply add: dom-def}$ )

**lemma** *Theorem-5:*

$\llbracket \forall n > 0. \models_f, n \Sigma m; f \in \text{dom } \Sigma m;$   
 $\text{bodyAPP } \Sigma f f : \llbracket \text{set } (\text{varsAPP } \Sigma f f) , [\text{varsAPP } \Sigma f f [\mapsto] \text{the } (\Sigma m f)] \rrbracket \rrbracket$   
 $\implies \models \Sigma m$

**apply** (*rule-tac*  $n = \text{Suc } n$  **in** *Theorem-5-aux*)  
**by** *simp-all*

**lemma** *imp-f-in-Sigma-ValidDepth-n-Sigma-Valid-Sigma*:  
 $\llbracket \forall n. \models_f, n \Sigma m; f \in \text{dom } \Sigma m \rrbracket$   
 $\implies \text{ValidGlobalMarkEnv } \Sigma m$   
**apply** (*subgoal-tac*  $\models_f, n \Sigma m$ )  
**prefer** 2 **apply** *simp*  
**apply** (*subgoal-tac*  $\models_f, 0 \Sigma m \wedge (\forall n > 0. \models_f, n \Sigma m), \text{elim conjE}$ )  
**prefer** 2 **apply** *simp*  
**apply** (*frule* *Theorem-4, assumption+*)  
**apply** (*frule* *Theorem-5, assumption+*)  
**by** (*rule equiv-all-n-SafeDAssDepth-SafeDAss, simp, simp*)

**lemma** *imp-all-n-ValidGlobalMarkEnvDepth-ValidGlobalMarkEnv*:  
 $\llbracket \forall n. \models_{f,n} \Sigma \rrbracket \implies \text{ValidGlobalMarkEnv } \Sigma$   
**apply** (*case-tac*  $f \notin \text{dom } \Sigma, \text{simp-all}$ )  
**apply** (*rule imp-f-notin-Sigma-ValidDepth-n-Sigma-Valid-Sigma, assumption+*)  
**by** (*rule imp-f-in-Sigma-ValidDepth-n-Sigma-Valid-Sigma, assumption+*)

**lemma** *lemma-6*:  
 $\forall n. \models_{f,n} \Sigma m \equiv \text{ValidGlobalMarkEnv } \Sigma m$   
**apply** (*rule eq-reflection*)  
**apply** (*rule iffI*)  
  
**apply** (*rule-tac*  $f = f$  **in** *imp-all-n-ValidGlobalMarkEnvDepth-ValidGlobalMarkEnv, force*)  
  
**by** (*rule imp-ValidGlobalMarkEnv-all-n-ValidGlobalMarkEnvDepth, force*)

**lemma** *lemma-7*:  
 $\llbracket \forall n. e, \Sigma m :_{f,n} \{ L, \Gamma \} \rrbracket$   
 $\implies \text{SafeDAssCntxt } e \Sigma m \ L \ \Gamma$   
**apply** (*simp only: SafeDAssDepthCntxt-def*)

**apply** (*subgoal-tac* ( $\forall n. \models_f, n \Sigma m \longrightarrow (\forall n. e :_f, n \llbracket L, \Gamma \rrbracket)$ ))  
**apply** (*erule thin-rl*)  
**apply** (*subst* (*asm*) *lemma-5*)  
**apply** (*subst* (*asm*) *lemma-6*)  
**apply** (*simp add: SafeDAssCntxt-def*)  
**by** *force*

**lemma** *lemma-8-REC* [*rule-format*]:  
 $(\forall n. (ValidGlobalMarkEnvDepth\ f\ n\ (\Sigma m(f \mapsto ms))) \longrightarrow (bodyAPP\ \Sigma f\ f) :_{f,n} \llbracket set\ (varsAPP\ \Sigma f\ f) , [(varsAPP\ \Sigma f\ f) \mapsto] ms \rrbracket)$   
 $\longrightarrow f \notin dom\ \Sigma m$   
 $\longrightarrow ValidGlobalMarkEnvDepth\ f\ n\ \Sigma m$   
 $\longrightarrow (bodyAPP\ \Sigma f\ f) :_{f,n} \llbracket set\ (varsAPP\ \Sigma f\ f) , [(varsAPP\ \Sigma f\ f) \mapsto] ms \rrbracket$   
**apply** (*rule impI*)  
**apply** (*induct-tac* *n*)

**apply** (*rule impI*) +  
**apply** (*erule-tac*  $x=0$  **in** *allE*)  
**apply** (*frule imp-ValidDepth-n-Sigma-Valid-Sigma,assumption+*)  
**apply** (*subgoal-tac*  $\models_f, 0 \Sigma m(f \mapsto ms), simp$ )  
**apply** (*rule ValidGlobalMarkEnvDepth.depth0,assumption+*)

**apply** (*erule-tac*  $x=Suc\ n$  **in** *allE*)  
**apply** (*rule impI*) +  
**apply** (*frule imp-ValidDepth-n-Sigma-Valid-Sigma,assumption+*)  
**apply** (*subgoal-tac*  $\models_f, n \Sigma m, simp$ )  
**apply** (*subgoal-tac*  $\models_f, Suc\ n \Sigma m(f \mapsto ms), simp$ )  
**apply** (*rule ValidGlobalMarkEnvDepth.step, simp-all*)  
**by** (*rule ValidGlobalMarkEnvDepth.base, assumption+*)

**lemma** *lemma-8*:  
 $e, \Sigma m \vdash_f (L, \Gamma)$   
 $\implies \forall n. e, \Sigma m :_{f,n} \llbracket L, \Gamma \rrbracket$   
**apply** (*erule ProofRulesED.induct*)

**apply** (*simp only: SafeDAssDepthCntxt-def*)  
**apply** (*rule allI, rule impI*)  
**apply** (*rule SafeDADepth-LitInt*)

**apply** (*simp only: SafeDAssDepthCntxt-def*)

```

apply (rule allI, rule impI)
apply (rule SafeDADepth-LitBool)

```

```

apply (simp only: SafeDAssDepthCntxt-def)
apply (rule allI, rule impI)
apply (rule SafeDADepth-Var1,force)

```

```

apply (simp only: SafeDAssDepthCntxt-def)
apply (rule allI, rule impI)
apply (rule SafeDADepth-Var2,force,force)

```

```

apply (simp only: SafeDAssDepthCntxt-def)
apply (rule allI, rule impI)
apply (rule SafeDADepth-Var3,force,force)

```

```

apply (rule allI)
apply (simp only: SafeDAssDepthCntxt-def)
apply (rule impI)
apply (erule-tac x=n in allE)+
apply (drule mp, simp)+
apply (rule SafeDADepth-LET1)
apply (assumption+,simp,assumption+,simp,simp,assumption+)

```

```

apply (rule allI)
apply (simp only: SafeDAssDepthCntxt-def)
apply (rule impI)
apply (erule-tac x=n in allE)+
apply (drule mp, simp)+
apply (rule SafeDADepth-LET1C)
apply (assumption+,simp,assumption+,simp,simp,simp)

```

```

apply (rule allI)
apply (simp only: SafeDAssDepthCntxt-def)
apply (rule impI)
apply (erule-tac x=n in allE)+
apply (drule mp, simp)+
apply (rule SafeDADepth-LET2)
apply assumption+

```

```

apply (rule allI)
apply (simp only: SafeDAssDepthCntxt-def)
apply (rule impI)

```

```

apply (erule-tac  $x=n$  in allE) +
apply (drule mp, simp) +
apply (rule SafeDADepth-LET2C)
apply (assumption +, simp, assumption +, simp, simp, simp)

```

```

apply (rule allI)
apply (simp only: SafeDAssDepthCntxt-def)
apply (rule impI)
apply (subgoal-tac
   $\forall i < \text{length } \text{alts}. \text{snd } (\text{alts } ! i) : f, n \parallel \text{fst } (\text{assert } ! i), \text{snd } (\text{assert } ! i) \parallel$ )
prefer 2 apply force
apply (frule-tac  $f=f$  and  $n=n$  in imp-wellFormed-wellFormedDepth)
apply (rule SafeDADepth-CASE)
apply (assumption +, simp, assumption +, simp, simp)

```

```

apply (rule allI)
apply (simp only: SafeDAssDepthCntxt-def)
apply (rule impI)
apply (subgoal-tac
   $\forall i < \text{length } \text{alts}. \text{snd } (\text{alts } ! i) : f, n \parallel \text{fst } (\text{assert } ! i), \text{snd } (\text{assert } ! i) \parallel$ )
prefer 2 apply force
apply (frule-tac  $f=f$  and  $n=n$  in imp-wellFormed-wellFormedDepth)
apply (rule SafeDADepth-CASED)
apply (assumption +, simp, assumption +, simp, simp, simp)

```

```

apply (rule allI)
apply (simp (no-asm) only: SafeDAssDepthCntxt-def)
apply (rule impI)
apply (rule SafeDADepth-APP-PRIMOP)
apply (assumption +)

```

```

apply (rule allI)
apply (simp only: SafeDAssDepthCntxt-def)
apply (rule impI)
apply (frule-tac  $f=f$  and  $n=n$  in imp-wellFormed-wellFormedDepth)
apply (rule SafeDADepth-APP)
apply (assumption +, simp, assumption +, simp, assumption +)

```

```

apply (simp only: SafeDAssDepthCntxt-def)
apply (rule allI)
apply (rule impI)
apply (subgoal-tac
   $ef = (\text{bodyAPP } \Sigma f f) \wedge$ 
   $xs = (\text{varsAPP } \Sigma f f), \text{simp}$ )

```

```

apply (rule-tac  $\Sigma m = \Sigma m$  in lemma-8-REC)
apply (simp,simp,simp)
apply (simp add: bodyAPP-def add: varsAPP-def)
done

```

```

lemma lemma-2:
   $e, \Sigma m \vdash_f (L, \Gamma)$ 
 $\implies \text{SafeDAssCntxt } e \ \Sigma m \ L \ \Gamma$ 
apply (rule lemma-7)
by (rule lemma-8,assumption)

```

**end**

## 23 Region deallocation

```

theory SafeRegionDepth imports SafeRegion-definitions
                                SafeDAssBasic
                                BasicFacts
                                SafeDAss-P1

```

**begin**

```

declare consistent.simps [simp del]
declare argP.simps [simp del]
declare wellT.simps [simp del]
declare SafeRegionDAss.simps [simp del]

```

```

lemma map-add-fst-Some:
   $\llbracket \vartheta \ r = \text{Some } y; \text{ dom } \vartheta \cap \text{dom } \vartheta' = \{\} \rrbracket$ 
 $\implies (\vartheta \ ++ \ \vartheta') \ r = \text{Some } y$ 
by (subst map-add-Some-iff,force)

```

```

lemma map-of-zip-is-Some:
  assumes length xs = length ys

```

**shows**  $x \in \text{set } xs \iff (\exists y. \text{map-of } (\text{zip } xs \text{ } ys) \ x = \text{Some } y)$   
**using** *assms* **by** (induct rule: list-induct2) simp-all

**lemma** *map-of-zip-twice-is-Some*:

$\llbracket x \in \text{set } xs; \text{length } xs = \text{length } vs; \text{distinct } xs; \text{length } xs = \text{length } zs \rrbracket$   
 $\implies \exists i < \text{length } vs. (\text{map-of } (\text{zip } xs \text{ } vs)) \ x = \text{Some } (vs!i) \wedge (\text{map-of } (\text{zip } xs \text{ } zs)) \ x = \text{Some } (zs!i)$   
**apply** (frule-tac  $x=x$  **in** *map-of-zip-is-Some,simp*)  
**apply** (elim exE)  
**apply** (subgoal-tac  
 $\text{set } (\text{zip } xs \text{ } vs) =$   
 $\{(xs!i, vs!i) \mid i. i < \min (\text{length } xs) (\text{length } vs)\}$ )  
**prefer** 2 **apply** (rule set-zip)  
**apply** (subgoal-tac  
 $\text{set } (\text{zip } xs \text{ } zs) =$   
 $\{(xs!i, zs!i) \mid i. i < \min (\text{length } xs) (\text{length } zs)\}$ )  
**prefer** 2 **apply** (rule set-zip)  
**by** auto

**lemma** *dom-map-f-comp*:

$\text{dom } (f \circ_f g) = \text{dom } g$   
**apply** (simp add: map-f-comp-def,auto)  
**by** (case-tac  $g \ x$ ,simp,simp)

**lemma** *dom-map-comp*:

$\forall x \in \text{dom } g. (\exists y. f (\text{the } (g \ x)) = \text{Some } y)$   
 $\implies \text{dom } (f \circ_m g) = \text{dom } g$   
**apply** (simp add: map-comp-def,auto)  
**apply** (case-tac  $g \ x$ ,simp,simp)  
**by** force

**lemma** *map-comp-map-of-zip*:

$\llbracket x \in \text{set } xs; \text{length } xs = \text{length } ys; \text{distinct } xs; \forall i < \text{length } ys. \exists z. m \ (ys!i) = \text{Some } z \rrbracket$   
 $\implies (m \circ_m \text{map-of } (\text{zip } xs \text{ } ys)) \ x = \text{map-of } (\text{zip } xs \text{ } (\text{map } (\text{the } \circ m) \text{ } ys)) \ x$   
**apply** (frule-tac  $x=x$  **in** *map-of-zip-is-Some,clarsimp*)  
**apply** (insert set-zip [where  $xs=xs$  and  $ys=ys$ ])  
**apply** (insert set-zip [where  $xs=xs$  and  $ys=\text{map } (\text{the } \circ m) \text{ } ys$ ])  
**apply** clarsimp  
**apply** (erule-tac  $x=i$  **in** *allE,simp*)  
**apply** (elim exE,simp)

**by** *force*

**lemma** *map-f-comp-map-of-zip*:

$\llbracket x \in \text{set } xs; \text{length } xs = \text{length } ys; \text{distinct } xs \rrbracket$   
 $\implies (f \circ_f \text{map-of } (\text{zip } xs \ ys)) \ x = \text{map-of } (\text{zip } xs \ (\text{map } f \ ys)) \ x$   
**apply** (*frule-tac*  $x=x$  **in** *map-of-zip-is-Some,clarsimp*)  
**apply** (*simp add: map-f-comp-def*)  
**apply** (*insert set-zip* [**where**  $xs=xs$  **and**  $ys=ys$ ])  
**apply** (*insert set-zip* [**where**  $xs=xs$  **and**  $ys=\text{map } f \ ys$ ])  
**by** *force*

**lemma** *map-of-zip-is-SomeI*:

$\llbracket x \in \text{set } xs; \text{length } xs = \text{length } vs; \text{distinct } xs \rrbracket$   
 $\implies \exists \ i < \text{length } vs. \ xs!i = x \wedge (\text{map-of } (\text{zip } xs \ vs)) \ x = \text{Some } (vs!i)$   
**apply** (*frule-tac*  $x=x$  **in** *map-of-zip-is-Some,simp*)  
**apply** (*subgoal-tac*  
 $\text{set } (\text{zip } xs \ vs) =$   
 $\{(xs!i, \ vs!i) \mid i. \ i < \min (\text{length } xs) (\text{length } vs)\}$ )  
**prefer** 2 **apply** (*rule set-zip*)  
**by** *auto*

**lemma** *map-mu-self*:

$\varrho\text{self} \notin \text{set } \varrho s$   
 $\implies \text{map } (\text{the} \circ \mu 2 (\varrho\text{self} \mapsto \varrho\text{self})) \ \varrho s = \text{map } (\text{the} \circ \mu 2) \ \varrho s$   
**by** *simp*

**lemma** *map-last*:

$xs \neq []$   
 $\implies \text{last } (\text{map } f \ xs) = f \ (\text{last } xs)$   
**by** (*induct xs,simp,simp*)

**lemma** *as-in-E1*:

$\llbracket fvs' \ as \subseteq \text{dom } E1; \ as \ ! \ i = \text{VarE } x \ a; \ i < \text{length } as \rrbracket$   
 $\implies x \in \text{dom } E1$   
**apply** (*induct as arbitrary: i,simp,clarsimp*)  
**by** (*case-tac i,force,force*)



**lemma** *rr-in-E2*:  
 $\llbracket \text{set } rr \subseteq \text{dom } E2 \rrbracket$   
 $\implies \forall i < \text{length } rr. \exists n. E2 (rr!i) = \text{Some } n$   
**apply** (*subgoal-tac*  $\text{set } rr = \{rr!i \mid i. i < \text{length } rr\}, \text{force}$ )  
**by** (*rule set-conv-nth*)

**lemma**  *$\eta$ - $\eta$ -ren*:  
 $\llbracket \eta x = \text{Some } r; x \neq \text{oself}; \text{ofake} \notin \text{dom } \eta \rrbracket$   
 $\implies \eta\text{-ren } \eta x = \text{Some } r$   
**apply** (*simp add:  $\eta$ -ren-def add: dom-def*)  
**by** *clarsimp*

**lemma** *SafeDARegionDepth-LitInt*:  
 $(\text{ConstE } (\text{LitN } i) a) :_f, n \{ (\vartheta 1, \vartheta 2), (\text{ConstrT } \text{intType } [] []) \}$   
**apply** (*unfold SafeRegionDAssDepth.simps*)  
**apply** (*intro allI, rule impI, elim conjE*)  
**apply** (*frule impSemBoundRA [where td=td]*)  
**apply** (*elim exE*)  
**apply** (*erule SafeRASem.cases, simp-all*)  
**by** (*rule consistent-v.primitiveI*)

**lemma** *SafeDARegionDepth-LitBool*:  
 $\text{ConstE } (\text{LitB } b) a :_f, n \{ (\vartheta 1, \vartheta 2), (\text{ConstrT } \text{boolType } [] []) \}$   
**apply** (*unfold SafeRegionDAssDepth.simps*)  
**apply** (*intro allI, rule impI, elim conjE*)  
**apply** (*frule impSemBoundRA [where td=td]*)  
**apply** (*elim exE*)  
**apply** (*erule SafeRASem.cases, simp-all*)  
**by** (*rule consistent-v.primitiveB*)

**lemma** *SafeDARegionDepth-Var1*:  
 $\llbracket \vartheta 1 x = \text{Some } t \rrbracket$   
 $\implies \text{VarE } x a :_f, n \{ (\vartheta 1, \vartheta 2), t \}$   
**apply** (*unfold SafeRegionDAssDepth.simps*)  
**apply** (*intro allI, rule impI, elim conjE*)

```

apply (frule impSemBoundRA [where td=td])
apply (elim exE)
apply (erule SafeRASem.cases,simp-all,clarsimp)
apply (simp add: consistent.simps)
apply (elim conjE)
apply (erule-tac x=x in ballE)
  apply clarsimp
by (simp add: dom-def)

```

```

lemma  $\varrho$ -notin-regions-mu-ext-copy-monotone:
  ( $\varrho \notin \text{regions } t \longrightarrow \text{mu-ext } (\mu 1, \mu 2(\varrho \mapsto \varrho')) t = \text{mu-ext } (\mu 1, \mu 2) t$ )  $\wedge$ 
  ( $\varrho \notin \text{regions}' tm \longrightarrow \text{mu-exts } (\mu 1, \mu 2(\varrho \mapsto \varrho')) tm = \text{mu-exts } (\mu 1, \mu 2) tm$ )
apply (induct-tac t and tm)
by clarsimp+

```

```

lemma map- $\varrho$ s-equals-butlast-copy:
  [ $\varrho s \neq []$ ; last  $\varrho s = \varrho$ ; distinct  $\varrho s$ ]
   $\implies \text{map } (\text{the} \circ \mu 2(\varrho \mapsto \varrho')) \varrho s =$ 
    butlast (map (the  $\circ \mu 2$ )  $\varrho s$ ) @ [ $\varrho'$ ]
by (induct  $\varrho s$ ,simp,clarsimp)

```

```

lemma copy'-Loc-p:
  [ $h p = \text{Some } (j', C, vn)$ ;
  copy'-dom (j, h, Loc p, True);
   $h' = \text{fst } (\text{mapAccumL } (\text{copy}' j) h (\text{zip } vn (\text{recursiveArgs } C)))$ ;
   $p' = \text{getFresh } h'$ ;
   $ps' = (\text{snd } (\text{mapAccumL } (\text{copy}' j) h (\text{zip } vn (\text{recursiveArgs } C))))$ ]
   $\implies \text{copy}' j h (\text{Loc } p, \text{True}) = (h'(p' \mapsto (j, C, ps')), \text{Val.Loc } p')$ 
apply (simp add: copy'.psimps(2))
apply (simp only: Let-def)
apply (case-tac (mapAccumL (copy' j) h (zip vn (recursiveArgs C))))
by simp

```

```

lemma copy-h'-p':
  [ $\text{copy } (h, k) p j = ((h', k), p')$ ; copy'-dom (j, h, Loc p, True);
   $h p = \text{Some } (j', C, vn)$ ]
   $\implies h' p' = \text{Some } (j, C, \text{snd } (\text{mapAccumL } (\text{copy}' j) h (\text{zip } vn (\text{recursiveArgs } C))))$ 
apply (simp only: copy.simps)
apply (subgoal-tac
  copy' j h (Loc p, True) =
    ((fst (mapAccumL (copy' j) h (zip vn (recursiveArgs C))))
     (getFresh (fst (mapAccumL (copy' j) h (zip vn (recursiveArgs C))))))  $\mapsto$ 
    (j, C, snd (mapAccumL (copy' j) h (zip vn (recursiveArgs C))))),
    Val.Loc (getFresh (fst (mapAccumL (copy' j) h (zip vn (recursiveArgs

```

```

C))))))
apply clarsimp
apply (rule copy'-Loc-p)
by simp-all

```

```

lemma snd-mapAccumL-x-xs:
  snd (mapAccumL f s (x#xs)) = (snd (f s x)) # snd (mapAccumL f (fst (f s x))
xs)
apply simp
apply (case-tac f s x, simp)
apply (case-tac (mapAccumL f a xs))
by simp

```

```

lemma length-snd-mapAccumL:
   $\forall s. \text{length } (\text{snd } (\text{mapAccumL } f s xs)) = \text{length } xs$ 
apply (induct xs)
apply simp
by (subst snd-mapAccumL-x-xs, simp)

```

```

lemma mapAccumL-snd-length:
  length xs = length ys
 $\implies \text{length } (\text{snd } (\text{mapAccumL } f s ys)) = \text{length } xs$ 
by (insert length-snd-mapAccumL, force)

```

```

lemma length-tn'-equals-length-recursiveArgs:
   $\llbracket \text{length } vn = \text{length } tn';$ 
    coherentC C;
    constructorSignature C = Some (tn',  $\varrho'$ , TypeExpression.ConstrT T tm'  $\varrho s'$ )
   $\rrbracket$ 
 $\implies \text{length } tn' = \text{length } (\text{zip } vn (\text{recursiveArgs } C))$ 
apply (simp add: coherentC-def)
apply (case-tac the (ConstructorTable C), simp)
apply (case-tac b, simp)
by (simp add: recursiveArgs-def)

```

```

lemma SafeDARegion-Var2-length-equals:
   $\llbracket \text{length } vn = \text{length } tn';$ 
    coherentC C;
    constructorSignature C = Some (tn',  $\varrho'$ , TypeExpression.ConstrT T tm'  $\varrho s'$ )
   $\rrbracket$ 
 $\implies \text{length } (\text{snd } (\text{mapAccumL } (\text{copy}' j) h (\text{zip } vn (\text{recursiveArgs } C)))) = \text{length } tn'$ 
apply (subgoal-tac length tn' = length (zip vn (recursiveArgs C)))

```

**apply** (*rule mapAccumL-snd-length,assumption*)  
**by** (*rule length-tn'-equals-length-recursiveArgs,assumption+*)

**declare** *copy.simps* [*simp del*]

**lemma** *SafeDARegionDepth-Var2*:  
 $\llbracket \vartheta 1 \ x = \text{Some} \ (\text{ConstrT} \ T \ ti \ \varrho l);$   
 $\vartheta 2 \ r = \text{Some} \ \varrho';$   
 $\text{coherent constructorSignature } Tc \rrbracket$   
 $\implies \text{CopyE } x \ r \ a :_f, _n \llbracket (\vartheta 1, \vartheta 2), \text{ConstrT } T \ ti \ ((\text{butlast } \varrho l) @ [\varrho']) \rrbracket$   
**apply** (*unfold SafeRegionDAssDepth.simps*)  
**apply** (*intro allI, rule impI, elim conjE*)  
**apply** (*frule impSemBoundRA [where td=td]*)  
**apply** (*elim exE*)  
**apply** (*erule SafeRASem.cases,simp-all*)  
**apply** (*simp add: consistent.simps*)  
**apply** (*elim conjE*)  
**apply** (*erule-tac x=x in ballE*)  
**prefer** 2 **apply** (*simp add: dom-def*)  
**apply** (*elim exE, elim conjE*)  
**apply** (*elim exE, elim conjE*)  
**apply** *clarsimp*  
**apply** (*erule consistent-v.cases,simp-all*)

**apply** (*simp add: def-copy*)  
**apply** (*erule-tac x=pa in ballE,simp*)  
**apply** (*subgoal-tac pa ∈ closureL pa (h,ka),simp*)  
**apply** (*rule closureL-basic*)

**apply** (*rule consistent-v.algebraic*)

**apply** (*frule dom-copy'*)  
**apply** (*frule copy-h'-p',force,force,force*)

**apply** *simp*

**apply** (*erule-tac x=r and A=dom E2a in ballE*)  
**prefer** 2 **apply** *force*  
**apply** (*elim exE, elim conjE*)+  
**apply** (*simp add: dom-def*)

**apply** (*erule-tac x=r and A=dom E2a in ballE*)  
**prefer** 2 **apply** *force*  
**apply** (*elim exE, elim conjE*)+  
**apply** *simp*

**apply** *force*

**apply** *force*

**apply** (*simp only: coherent-def*)  
**apply** (*elim conjE*)  
**apply** (*rule SafeDARegion-Var2-length-equals,assumption+*)  
**apply** (*erule-tac x=C and A=dom constructorSignature in ballE*)  
**apply** (*simp,force,force*)

**apply** *clarsimp*  
**apply** (*rule-tac x= $\mu 1$  in exI*)  
**apply** (*rule-tac x= $\mu 2$ (last  $qs' \mapsto q'$ ) in exI*)  
**apply** (*drule-tac s=the ( $\mu 2$  (last  $qs'$ )) in sym*)  
**apply** *simp*  
**apply** (*simp add: wellT.simps*)  
**apply** (*elim conjE*)  
**apply** (*subgoal-tac  $\exists \varrho. \text{last } \varrho s' = \varrho$* )  
**prefer** 2 **apply** *simp*  
**apply** (*erule exE*)  
**apply** *simp*  
**apply** (*rule conjI*)  
**apply** (*subgoal-tac*  
   ( $\varrho \notin \text{regions } t$   
      $\longrightarrow \text{mu-ext } (\mu 1, \mu 2(\varrho \mapsto q')) \ t = \text{mu-ext } (\mu 1, \mu 2) \ t \wedge$   
   ( $\varrho \notin \text{regions}' \ tm'$   
      $\longrightarrow \text{mu-exts } (\mu 1, \mu 2(\varrho \mapsto q')) \ tm' = \text{mu-exts } (\mu 1, \mu 2) \ tm'$ ))  
**apply** *simp*  
**apply** (*rule  $\varrho$ -notin-regions-mu-ext-copy-monotone*)  
**apply** (*rule conjI*)  
**apply** (*rule map-qs-equals-butlast-copy,assumption+*)  
**by** (*rule SafeDARegion-Var2-2, assumption+*)

**lemma** *case-Recursive:*

$\llbracket h \ q = \text{Some } (j, C, vn); i < \text{length } vn; vn \ ! \ i = \text{Loc } p;$   
 $\text{snd } (\text{snd } (\text{the } (\text{ConstructorTable } C)))!i = \text{Recursive};$   
 $\text{length } (\text{snd } (\text{snd } (\text{the } (\text{ConstructorTable } C)))) = \text{length } vn \rrbracket$   
 $\implies (\text{Recursive}, \text{Loc } p) \in$   
 $\{x \in \text{set } (\text{zip } (\text{getArgType } (\text{getConstructorCell } (C, vn)))$   
    $(\text{getValuesCell } (C, vn))). \text{isNonBasicValue } (\text{fst } x)\}$   
**apply** (*subst set-zip*)  
**apply** (*simp add: getConstructorCell-def*)  
**apply** (*simp add: getArgType-def*)  
**apply** (*simp add: getValuesCell-def*)  
**apply** (*simp add: isNonBasicValue-def*)  
**by** *force*

**lemma** *case-NonRecursive*:

$\llbracket h \ q = \text{Some } (j, C, vn); i < \text{length } vn; vn ! i = \text{Loc } p;$   
 $\text{snd } (\text{snd } (\text{the } (\text{ConstructorTable } C))) ! i = \text{NonRecursive};$   
 $\text{length } (\text{snd } (\text{snd } (\text{the } (\text{ConstructorTable } C)))) = \text{length } vn \rrbracket$   
 $\implies (\text{NonRecursive}, \text{Loc } p) \in$   
 $\{x \in \text{set } (\text{zip } (\text{getArgType } (\text{getConstructorCell } (C, vn))) (\text{getValuesCell } (C,$   
 $vn)))) . \text{isNonBasicValue } (\text{fst } x)\}$   
**apply** (*subst set-zip*)  
**apply** (*simp add: getConstructorCell-def*)  
**apply** (*simp add: getArgType-def*)  
**apply** (*simp add: getValuesCell-def*)  
**apply** (*simp add: isNonBasicValue-def*)  
**by** *force*

**lemma** *p-in-closureL-Recursive*:

$\llbracket h \ q = \text{Some } (j, C, vn); i < \text{length } vn; vn ! i = \text{Loc } p;$   
 $\text{length } (\text{snd } (\text{snd } (\text{the } (\text{ConstructorTable } C)))) = \text{length } vn;$   
 $\text{snd } (\text{snd } (\text{the } (\text{ConstructorTable } C))) ! i = \text{Recursive} \rrbracket$   
 $\implies p \in \text{closureL } q \ (h, k)$   
**apply** (*rule-tac q=q in closureL-step*)  
**apply** (*rule closureL-basic*)  
**apply** (*simp add: descendants-def*)  
**apply** (*simp add: getNonBasicValuesCell-def*)  
**apply** (*frule case-Recursive, assumption+*)  
**by** *force*

**lemma** *p-in-closureL-NonRecursive*:

$\llbracket h \ q = \text{Some } (j, C, vn); i < \text{length } vn; vn ! i = \text{Loc } p;$   
 $\text{length } (\text{snd } (\text{snd } (\text{the } (\text{ConstructorTable } C)))) = \text{length } vn;$   
 $\text{snd } (\text{snd } (\text{the } (\text{ConstructorTable } C))) ! i = \text{NonRecursive} \rrbracket$   
 $\implies p \in \text{closureL } q \ (h, k)$   
**apply** (*rule-tac q=q in closureL-step*)  
**apply** (*rule closureL-basic*)  
**apply** (*simp add: descendants-def*)  
**apply** (*simp add: getNonBasicValuesCell-def*)  
**apply** (*frule case-NonRecursive, assumption+*)  
**by** *force*

**lemma** *closureV-vn-closureV-q*:

$\llbracket h \ q = \text{Some } (j, C, vn); \text{coherentC } C;$   
 $\text{constructorSignature } C = \text{Some } (tn', \varrho', \text{TypeExpression.ConstrT } T \ tm' \ \varrho s^{\wedge});$   
 $\text{length } tn' = \text{length } vn \rrbracket$   
 $\implies (\bigcup i < \text{length } vn. \text{closureV } (vn ! i) \ (h, k)) \subseteq$   
 $\text{closureV } (\text{Loc } q) \ (h, k)$

```

apply (rule subsetI)
apply (simp add: closureV-def)
apply (elim bexE)
apply (case-tac vn!i, simp-all)
apply (erule closureL.induct)
  prefer 2 apply (rule closureL-step, assumption+)
apply (rename-tac p)
apply (simp only: coherentC-def)
apply (case-tac the (ConstructorTable C))
apply (case-tac b)
apply (rename-tac nargs b n largs)
apply (case-tac the (constructorSignature C))
apply (case-tac ba)
apply (rename-tac ts ba gl t)
apply (simp add: Let-def)
apply (elim conjE)
apply (erule-tac x=i in allE, simp)
apply (elim conjE)
apply (case-tac ts!i)

apply simp
apply (rule p-in-closureL-NonRecursive)
apply (assumption+, simp, assumption+, simp, simp)

apply (case-tac ts!i = TypeExpression.ConstrT intType [] [])

apply clarsimp

apply (case-tac ts!i = TypeExpression.ConstrT boolType [] [])
apply clarsimp

apply (case-tac ts!i ≠ ConstrT T tm' qs')
apply simp
apply (rule p-in-closureL-NonRecursive)
apply (assumption+, simp, assumption+, simp, simp)

apply simp
apply (rule p-in-closureL-Recursive)
apply (assumption+, simp, assumption+, simp, simp)
done

lemma p-notin-closureV-q-notin-closureV-vn:
  [| p ∉ closureV (Loc q) (h, k);
    h q = Some (j, C, vn);
    constructorSignature C = Some (tn', q', TypeExpression.ConstrT T tm' qs');
    length tn' = length vn;
    coherentC C |]
  ⇒ ∀ i < length vn. p ∉ closureV (vn!i) (h, k)
apply (subgoal-tac

```

$(\bigcup i < \text{length } vn. \text{closure}V (vn!i) (h,k)) \subseteq$   
 $\text{closure}V (\text{Loc } q) (h, k))$   
**apply** *blast*  
**by** (*rule closureV-vn-closureV-q,force+*)

**lemma** *SafeDARegion-Var3-1* [*rule-format*]:  
*consistent-v t η v h*  
 $\longrightarrow \text{fresh } q \ h$   
 $\longrightarrow h \ p = \text{Some } (j, C, vn)$   
 $\longrightarrow v \in \text{rangeHeap } h$   
 $\longrightarrow p \notin \text{closure}V \ v \ (h, k)$   
 $\longrightarrow \text{coherent constructorSignature } Tc$   
 $\longrightarrow \text{consistent-v } t \ \eta \ v \ (h(p := \text{None})(q \mapsto c))$   
**apply** (*rule impI*)  
**apply** (*erule consistent-v.induct,simp-all*)

**apply** (*clarsimp, rule consistent-v.primitiveI*)

**apply** (*clarsimp, rule consistent-v.primitiveB*)

**apply** (*clarsimp, rule consistent-v.variable*)

**apply** (*clarsimp, rule consistent-v.algebraic-None*)  
**apply** (*subgoal-tac pa ≠ p,simp*)  
**prefer** 2 **apply** *force*  
**apply** (*subgoal-tac pa ≠ q,clarsimp*)  
**apply** (*simp add: fresh-def, clarsimp*)

**apply** (*elim exE, elim conjE*)  
**apply** (*rule impI*)+  
**apply** (*rule consistent-v.algebraic*)

**apply** (*subgoal-tac pa ≠ q*)  
**prefer** 2 **apply** (*simp add: fresh-def,elim conjE, simp add: dom-def, force*)  
**apply** (*case-tac pa ≠ p,force*)  
**apply** (*simp add: closureV-def*)  
**apply** (*subgoal-tac p ∈ closureL p (h, k)*)  
**apply** *simp*  
**apply** (*rule closureL.closureL-basic*)

**apply** *force*

**apply** *force*

**apply** *force*

**apply** *force*



```

apply force

apply force

apply (rule-tac  $x=\mu 1$  in  $exI$ )
apply (rule-tac  $x=\mu 2$  in  $exI$ )
apply (rule conjI)
  apply force
apply (rule allI,rule impI)
apply (erule-tac  $x=i$  in  $allE,simp$ )
apply (elim conjE)
apply (simp add: coherent-def)
apply (elim conjE)
apply (erule-tac  $x=Ca$  in  $ballE$ )
apply (frule p-notin-closureV-q-notin-closureV-vn)
apply (assumption+,simp+)
apply (simp add: rangeHeap-def,clarsimp)
apply force
apply force
done

lemma SafeDARegionDepth-Var3:
  [  $\vartheta 1\ x = \text{Some } t; \text{coherent constructorSignature } Tc$  ]
   $\implies \text{ReuseE } x\ a :_f, n\ \{ (\vartheta 1, \vartheta 2), t \}$ 
apply (unfold SafeRegionDAssDepth.simps)
apply (intro allI, rule impI, elim conjE)
apply (frule impSemBoundRA [where td=td])
apply (elim exE)
apply (erule SafeRASem.cases,simp-all,clarsimp)
apply (case-tac t,simp-all)
  apply (rule consistent-v.variable)
apply (simp add: consistent.simps)
apply (elim conjE)
apply (erule-tac  $x=x$  in  $ballE$ )
  prefer 2 apply (simp add: dom-def)
apply (elim exE, elim conjE)
apply (elim exE, elim conjE)
apply clarsimp
apply (erule consistent-v.cases,simp-all)

apply (simp add: dom-def)

apply (elim exE, elim conjE)
apply (rule consistent-v.algebraic)

apply force

```

**apply** *simp*

**apply** (*erule-tac*  $x=r$  **and**  $A=\text{dom } E2a$  **in**  $\text{ball}E$ )  
**prefer** 2 **apply** *force*  
**apply** (*elim*  $\text{ex}E$ , *elim*  $\text{conj}E$ ) +  
**apply** (*simp* *add: dom-def*)

**apply** (*erule-tac*  $x=r$  **and**  $A=\text{dom } E2a$  **in**  $\text{ball}E$ )  
**prefer** 2 **apply** *force*  
**apply** (*elim*  $\text{ex}E$ , *elim*  $\text{conj}E$ ) +  
**apply** *simp*

**apply** *force*

**apply** *force*

**apply** *force*

**apply** (*rule-tac*  $x=\mu 1$  **in**  $\text{ex}I$ )  
**apply** (*rule-tac*  $x=\mu 2$  **in**  $\text{ex}I$ )  
**apply** (*rule*  $\text{conj}I$ )  
**apply** *simp*  
**apply** (*rule*  $\text{all}I$ , *rule*  $\text{imp}I$ )  
**apply** (*erule-tac*  $x=i$  **in**  $\text{all}E$ , *simp*)  
**apply** (*frule-tac*  $k=k$  **in** *no-cycles*)  
**apply** (*erule-tac*  $x=i$  **in**  $\text{all}E$ , *simp*)  
**apply** (*frule* *SafeDARegion-Var3-1*)  
**apply** (*assumption* +, *simp* *add: rangeHeap-def, force, assumption* +)  
**done**

**lemma** *SafeDARegion-LET1-P1'*:

$\llbracket (E1, E2) \vdash h, k, td, \text{Let } x1 = e1 \text{ In } e2 \text{ a} \Downarrow h', k, v, r;$   
 $x1 \notin \text{fv } e1;$   
 $\text{fv } e1 \cup \text{fv } e2 - \{x1\} \subseteq \text{dom } E1 \rrbracket$   
 $\implies \text{fv } e1 \subseteq \text{dom } E1 \wedge \text{fv } e2 \subseteq \text{insert } x1 (\text{dom } E1)$   
**apply** (*rule*  $\text{conj}I$ )  
**apply** *blast*  
**by** *blast*

**lemma** *consistent-v-notin-dom-h'* [*rule-format*]:

*consistent-v*  $t \eta v h$   
 $\longrightarrow v \notin \text{domLoc } h'$

$\longrightarrow (\forall p \in \text{dom } h. p \notin \text{dom } h' \vee h p = h' p)$   
 $\longrightarrow \text{consistent-}v \ t \ \eta \ v \ h'$   
**apply** (rule *impI*)  
**apply** (erule *consistent-v.induct*)

**apply** (rule *impI*)+  
**apply** (rule *consistent-v.primitiveI*)

**apply** (rule *impI*)+  
**apply** (rule *consistent-v.primitiveB*)

**apply** (rule *impI*)+  
**apply** (rule *consistent-v.variable*)

**apply** (rule *impI*)+  
**apply** (rule *consistent-v.algebraic-None*)  
**apply** (simp add: *domLoc-def*)

**apply** (rule *impI*)+  
**apply** (elim *exE*)  
**apply** (erule-tac  $x=p$  **in** *ballE*)  
**apply** (erule *disjE*)

**apply** (rule *consistent-v.algebraic-None,simp*)

**apply** (rule *consistent-v.algebraic*)  
**apply** (force,force,force,force,force,force,force)  
**apply** (rule-tac  $x=\mu 1$  **in** *exI*)  
**apply** (rule-tac  $x=\mu 2$  **in** *exI*)  
**apply** (simp add: *domLoc-def*)  
**apply** force  
**apply** (simp add: *dom-def*)  
**done**

**lemma** *consistent-v-Loc-p-in-dom-h* [rule-format]:  
 $\text{consistent-}v \ t \ \eta \ (\text{Loc } p) \ h$   
 $\longrightarrow (E1, E2) \vdash h, k, td, e \Downarrow h', k, v, r$   
 $\longrightarrow (\text{Loc } p) \in \text{domLoc } h$   
 $\longrightarrow (\forall p \in \text{dom } h. p \notin \text{dom } h' \vee h p = h' p)$   
 $\longrightarrow \text{consistent-}v \ t \ \eta \ (\text{Loc } p) \ h'$   
**apply** (rule *impI*)  
**apply** (erule *consistent-v.induct*)

**apply** (simp add: *domLoc-def*)

**apply** (simp add: *domLoc-def*)

**apply** (clarsimp,rule *consistent-v.variable*)

```

apply (simp add: domLoc-def)

apply (rule impI)+
apply (case-tac  $p \notin \text{dom } h'$ )
apply (rule consistent-v.algebraic-None,simp)

apply (elim exE, elim conjE,simp,elim conjE)
apply (rule consistent-v.algebraic)
apply (simp add: domLoc-def)
apply (force,force,force,force,force,force,force)
apply (rule-tac  $x=\mu 1$  in exI)
apply (rule-tac  $x=\mu 2$  in exI)
apply (rule conjI, force)
apply clarsimp
apply (erule-tac  $x=i$  in allE,simp)+
apply (simp add: domLoc-def)
apply (case-tac  $(\exists p. p \in \text{dom } h \wedge \text{vn } ! i = \text{Loc } p),\text{simp})$ 
apply (rule consistent-v-notin-dom-h',simp)

apply (erule-tac  $x=p$  in ballE)
prefer 2 apply (simp add: dom-def)
apply (drule mp)
apply (simp add: dom-def)
apply (frule-tac  $v'=\text{vn}!i$  in semantic-no-capture-h)
apply (simp add: rangeHeap-def add: domLoc-def)
apply force
apply simp
apply simp
done

lemma monotone-consistent-Loc-p:
   $\llbracket (E1, E2) \vdash h, k, td, e \Downarrow h', k, v, r; \quad$ 
 $\vartheta 1 x = \text{Some } t; E1 x = \text{Some } (\text{Loc } p); \quad$ 
 $\text{consistent-v } t \eta (\text{Loc } p) h \rrbracket$ 
 $\implies \text{consistent-v } t \eta (\text{Loc } p) h'$ 
apply (case-tac  $p \in \text{dom } h$ )
apply (frule semantic-extend-pointers)
apply (rule consistent-v-Loc-p-in-dom-h,assumption+)
apply (simp add: domLoc-def, simp)
apply (frule semantic-no-capture-E1,assumption+)
by (rule consistent-v.algebraic-None,simp)

lemma monotone-consistent-v:
   $\llbracket (E1, E2) \vdash h, k, td, e \Downarrow h', k, v, r; \quad$ 
 $\vartheta 1 x = \text{Some } t; E1 x = \text{Some } v'; \text{consistent-v } t \eta v' h \rrbracket$ 
 $\implies \text{consistent-v } t \eta v' h'$ 
apply (case-tac  $v',\text{simp-all})$ 

```

```

apply (rule monotone-consistent-Loc-p,assumption+)

apply (erule consistent-v.cases,simp-all)
apply (rule consistent-v.primitiveI)
apply (rule consistent-v.variable)

apply (erule consistent-v.cases,simp-all)
apply (rule consistent-v.primitiveB)
apply (rule consistent-v.variable)
done

```

```

lemma monotone-consistent:
  
$$\begin{aligned} & \llbracket (E1, E2) \vdash h, k, td, e \Downarrow h', k, v, r; \\ & \quad \text{consistent } (\vartheta 1, \vartheta 2) \eta (E1, E2) h \rrbracket \\ & \implies \text{consistent } (\vartheta 1, \vartheta 2) \eta (E1, E2) h' \end{aligned}$$

apply (simp add: consistent.simps)
apply (rule ballI)
apply (elim conjE)
apply (erule-tac x=x in ballE)
  prefer 2 apply simp
apply (elim exE, elim conjE)
apply (elim exE, elim conjE)
apply (rule-tac x=t in exI)
  apply (rule conjI,assumption)
apply (rule-tac x=va in exI)
  apply (rule conjI,assumption)
by (rule monotone-consistent-v,assumption+)

```

```

lemma SafeDARegion-LET1-P5:
  
$$\begin{aligned} & \llbracket (E1, E2) \vdash h, k, td, e1 \Downarrow h', k, v1, r; x1 \notin \text{dom } \vartheta 1; \\ & \quad \text{consistent } (\vartheta 1, \vartheta 2) \eta (E1, E2) h; \text{consistent-v } t1 \eta v1 h' \rrbracket \\ & \implies \text{consistent } (\vartheta 1(x1 \mapsto t1), \vartheta 2) \eta (E1(x1 \mapsto v1), E2) h' \end{aligned}$$

apply (drule-tac h'=h' in monotone-consistent, simp)
apply (unfold consistent.simps)
apply (rule conjI)
apply simp
apply (rule conjI)
apply (rule ballI)
apply (elim conjE)
apply (erule-tac x=r and A=dom E2 in ballE)
  prefer 2 apply simp
apply (elim exE, elim conjE)
apply (rule-tac x=r' in exI)
apply (rule-tac x=r'' in exI)
apply (rule conjI)

```

**apply** *clarsimp*  
**apply** *clarsimp*  
**by** *clarsimp*

**lemma** *SafeDARegionDepth-LET1*:

$\llbracket \forall C \text{ as } r \text{ a}'. e1 \neq \text{ConstrE } C \text{ as } r \text{ a}';$   
 $x1 \notin \text{dom } \vartheta 1; x1 \notin \text{fv } e1;$   
 $e1 :_f, n \llbracket (\vartheta 1, \vartheta 2), t1 \rrbracket ;$   
 $e2 :_f, n \llbracket (\vartheta 1(x1 \mapsto t1), \vartheta 2), t2 \rrbracket \rrbracket$   
 $\implies \text{Let } x1 = e1 \text{ In } e2 \text{ a} :_f, n \llbracket (\vartheta 1, \vartheta 2), t2 \rrbracket$

**apply** (*unfold SafeRegionDAssDepth.simps*)  
**apply** (*intro allI, rule impI*)  
**apply** (*elim conjE*)  
**apply** (*frule impSemBoundRA [where e=Let x1 = e1 In e2 a and td=td]*)  
**apply** (*elim exE*)  
**apply** (*frule P1-f-n-LET, assumption*)  
**apply** (*elim exE*)  
**apply** (*erule-tac x=C in allE*)  
**apply** (*erule-tac x=E1 in allE*)  
**apply** (*erule-tac x=E2 in allE*)  
**apply** (*erule-tac x=h in allE*)  
**apply** (*erule-tac x=k in allE*)  
**apply** (*erule-tac x=h'a in allE*)  
**apply** (*erule-tac x=v1 in allE*)  
**apply** (*erule-tac x= $\eta$  in allE*)

**apply** (*erule-tac x=E1( $x1 \mapsto v1$ ) in allE*)  
**apply** (*erule-tac x=E2 in allE*)  
**apply** (*erule-tac x=h'a in allE*)  
**apply** (*erule-tac x=k in allE*)  
**apply** (*erule-tac x=h' in allE*)  
**apply** (*erule-tac x=v in allE*)  
**apply** (*erule-tac x= $\eta$  in allE*)  
**apply** (*frule SafeDARegion-LET1-P1', assumption, simp*)  
**apply** (*elim conjE*)

**apply** (*drule mp*)  
**apply** (*rule conjI, simp*)  
**apply** (*rule conjI, simp*)  
**apply** (*rule conjI, simp*)  
**apply** (*rule conjI, simp*)  
**apply** (*rule conjI, simp*)  
**apply** (*rule conjI, simp*)  
**apply** (*simp*)

**apply** (*drule mp*)  
**apply** (*rule conjI, simp*)

```

apply (rule conjI,simp)
apply (rule conjI,simp)
apply (rule conjI,simp,blast)
apply (rule conjI,simp)
apply (rule conjI,simp)
apply (frule-tac e=e1 and td=td in impSemBoundRA)
apply (elim exE)
by (rule SafeDARegion-LET1-P5,force,assumption+)

```

```

lemma fvs-prop2:
   $\llbracket i < \text{length } as; as ! i = \text{VarE } x \ a \rrbracket$ 
   $\implies x \in \text{fvs } as$ 
by (induct as i rule: list-induct3, simp-all)

```

```

lemma mu-last:
   $\varrho s \neq []$ 
   $\implies (\text{the } (\mu 2 (\text{last } \varrho s))) = \text{last } (\text{map } (\text{the } \circ \mu 2) \varrho s)$ 
by (induct  $\varrho s$ ,simp,clarsimp)

```

```

lemma extend-heaps-upd:
  fresh p h
   $\implies \text{extend-heaps } (h,k) (h(p \mapsto (j, C, \text{map } (\text{atom2val } E1) as)),k)$ 
apply (simp add: extend-heaps.simps)
apply (rule ballI)
apply (subgoal-tac p  $\notin \text{dom } h$ )
apply (subgoal-tac pa  $\neq p$ ,simp)
  apply (frule fresh-notin-closureL)
  apply force
apply (simp add: dom-def,force)
by (simp add: fresh-def)

```

```

lemma not-in-set-conv-nth:
   $x \notin \text{set } xs$ 
   $\implies \forall i < \text{length } xs. xs!i \neq x$ 
apply (case-tac xs = [],simp-all)
apply (rule allI, rule impI)
apply (induct xs arbitrary: i,simp-all)
apply (case-tac i, simp-all)
apply force

```

by force

```

lemma consistent-v-fresh-p-Loc-q [rule-format]:
  consistent-v t  $\eta$  (Loc q) h
   $\longrightarrow$  (Loc q)  $\neq$  (Loc p)
   $\longrightarrow$  fresh p h
   $\longrightarrow$  consistent-v t  $\eta$  (Loc q) (h(p  $\mapsto$  (j, C, map (atom2val E1) as)))
apply (rule impI)
apply (erule consistent-v.induct,simp-all)

apply (clarsimp,rule consistent-v.primitiveI)

apply (clarsimp,rule consistent-v.primitiveB)

apply (clarsimp,rule consistent-v.variable)

apply (clarsimp,rule consistent-v.algebraic-None)
apply (simp add: fresh-def, simp add: dom-def)

apply clarsimp
apply (rule consistent-v.algebraic)
apply (subgoal-tac p  $\neq$  pa,force)
apply (simp add: fresh-def)
apply (force,force,force,force,force,force)
apply (rule-tac x= $\mu$ 1 in exI)
apply (rule-tac x= $\mu$ 2 in exI)
apply (rule conjI, force)
apply (simp add: fresh-def)
apply (elim conjE)
apply (simp add: rangeHeap-def)
apply (rotate-tac 9)
apply (erule-tac x=pa in allE)
apply (rotate-tac 9)
apply (erule-tac x=ja in allE)
apply (erule-tac x=Ca in allE)
apply (erule-tac x=vn in allE)
apply simp
apply (frule not-in-set-conv-nth)
apply simp
done

```

```

lemma monotone-consistent-v-fresh:
   $\llbracket (E1, E2) \vdash h, k, td, e \Downarrow h', k, v, r ;$ 
   $\vartheta 1\ x = \text{Some } t; E1\ x = \text{Some } v'; \text{fresh } p\ h;$ 
   $\text{consistent-v } t\ \eta\ v'\ h \rrbracket$ 
   $\implies \text{consistent-v } t\ \eta\ v'\ (h(p \mapsto (j, C, \text{map } (\text{atom2val } E1)\ as)))$ 
apply (case-tac v',simp-all)

```



```

apply (rename-tac q)
apply (frule semantic-no-capture-E1-fresh,simp)
apply (erule-tac x=x in ballE)
  prefer 2 apply (simp add: dom-def)
apply (erule-tac x=q in allE, simp)
apply (rule consistent-v-fresh-p-Loc-q,assumption+,force,simp)

```

```

apply (erule consistent-v.cases,simp-all)
apply (rule consistent-v.primitiveI)
apply (rule consistent-v.variable)

```

```

apply (erule consistent-v.cases,simp-all)
apply (rule consistent-v.primitiveB)
apply (rule consistent-v.variable)
done

```

**lemma** *SafeDARegion-LETC-P5*:

```


$$\begin{aligned}
& \llbracket (E1, E2) \vdash h, k, td, \text{Let } x1 = \text{ConstrE } C \text{ as } r () \text{ In } e2 () \Downarrow h', k, v, ra; \\
& \quad E2 \text{ r} = \text{Some } j; x1 \notin \text{fvs as}; \text{fvs as} \subseteq \text{dom } E1; x1 \notin \text{dom } \vartheta 1; \\
& \quad \text{constructorSignature } C = \text{Some } (ti, \varrho, \text{TypeExpression.ConstrT } T \text{ tn } \varrho s); \\
& \quad t' = \text{mu-ext } (\mu 1, \mu 2) (\text{TypeExpression.ConstrT } T \text{ tn } \varrho s); \\
& \quad \text{argP } (\text{map } (\text{mu-ext } (\mu 1, \mu 2)) \text{ ti}) (\text{the } (\mu 2 \varrho)) \text{ as } r (\vartheta 1, \vartheta 2); \\
& \quad \text{consistent } (\vartheta 1, \vartheta 2) \eta (E1, E2) \text{ h}; \text{fresh } p \text{ h}; \\
& \quad \text{wellT } ti \varrho (\text{TypeExpression.ConstrT } T \text{ tn } \varrho s) \rrbracket \\
& \implies \text{consistent } (\vartheta 1(x1 \mapsto t'), \vartheta 2) \eta (E1(x1 \mapsto \text{Loc } p), E2) (h(p \mapsto (j, C, \text{map} \\
& (\text{atom2val } E1) \text{ as})))
\end{aligned}$$

apply (unfold consistent.simps)
apply (elim conjE)
apply (rule conjI)

```

```

apply (rule ballI,simp)
apply (rule conjI)

```

```

apply (rule impI,simp)
apply (simp add: wellT.simps)
apply (simp add: argP.simps)
apply (elim conjE)
apply (erule-tac x=r and A=dom E2 in ballE)
  prefer 2 apply (simp add: dom-def)
apply (elim exE, elim conjE)+
apply (subgoal-tac (the (\mu 2 (last \varrho s))) = last (map (the \circ \mu 2) \varrho s))
  prefer 2 apply (rule mu-last,simp)
apply (rule consistent-v.algebraic)

```

```

apply force

```

```

apply force

apply (simp add: dom-def)

apply force

apply force

  apply (simp add: wellT.simps)

  apply force

apply (rule-tac x= $\mu$ 1 in exI)
apply (rule-tac x= $\mu$ 2 in exI)
apply clarsimp
apply (erule-tac x=i in allE, simp)+
apply (erule disjE)

  apply (erule exE)+
  apply simp
  apply (rule consistent-v.primitiveI)
apply (erule disjE)

  apply (erule exE)+
  apply simp
  apply (rule consistent-v.primitiveB)

apply (erule exE)+
apply simp
apply (erule-tac x=x in ballE)
  prefer 2 apply (frule fvs-prop2, assumption+, force)
apply (elim exE, elim conjE)+
apply simp
apply (rule monotone-consistent-v-fresh, assumption+)

apply (rule impI)
apply (erule-tac x=x in ballE)
  prefer 2 apply simp
apply simp
apply (elim exE, elim conjE)+
apply (rule-tac x=t in exI)
apply (rule conjI, assumption)
apply (rule-tac x=va in exI)
apply (rule conjI, assumption)
apply (rule monotone-consistent-v-fresh, assumption+)

apply (rule conjI)

```

```

apply (rule ballI)
apply (erule-tac x=rb and A=dom E2 in ballE)
  prefer 2 apply simp
apply (elim exE)
apply (rule-tac x=r' in exI)
apply (rule-tac x=r'' in exI)
apply (rule conjI)
  apply clarsimp
apply clarsimp
by clarsimp

```

**lemma** *SafeDARegionDepth-LETC*:

```

   $\llbracket x1 \notin \text{fvs } as; x1 \notin \text{dom } \vartheta1;$ 
   $\text{constructorSignature } C = \text{Some } (ti, \varrho, t);$ 
   $t = \text{ConstrT } T \text{ tn } \varrho s;$ 
   $t' = \text{mu-ext } (\mu1, \mu2) \text{ } t;$ 
   $\text{argP } (\text{map } (\text{mu-ext } (\mu1, \mu2)) \text{ } ti) ((\text{the } \circ \mu2) \text{ } \varrho) \text{ as } r \text{ } (\vartheta1, \vartheta2);$ 
   $\text{wellT } ti \text{ } \varrho \text{ } t;$ 
   $e2 :_f, n \llbracket (\vartheta1(x1 \mapsto t'), \vartheta2), t'' \rrbracket$ 
   $\implies \text{Let } x1 = \text{ConstrE } C \text{ as } r \text{ } a' \text{ In } e2 \text{ } a :_f, n \llbracket (\vartheta1, \vartheta2), t'' \rrbracket$ 
apply (unfold SafeRegionDAssDepth.simps)
apply (intro allI, rule impI)
apply (elim conjE)
apply (frule impSemBoundRA [where e=Let x1 = ConstrE C as r a' In e2 a
  and td=td])
apply (elim exE)
apply (frule P1-f-n-LETC)
apply (erule exE)+
apply (elim conjE)

```

```

apply (erule-tac x=E1(x1  $\mapsto$  Loc p) in allE)
apply (erule-tac x=E2 in allE)
apply (erule-tac x=h(p  $\mapsto$  (j, C, map (atom2val E1) as)) in allE)
apply (erule-tac x=k in allE)
apply (erule-tac x=h' in allE)
apply (erule-tac x=v in allE)
apply (erule-tac x= $\eta$  in allE)
apply (simp)
apply (elim conjE)
apply (drule mp)
apply (rule conjI,blast)
apply (rule conjI,blast)
apply (subgoal-tac fvs as  $\subseteq$  dom E1)
  prefer 2 apply blast

```

**apply** (*rule-tac* ? $\mu 1.0=\mu 1$  **and** ? $\mu 2.0=\mu 2$  **in** *SafeDARegion-LETC-P5*)  
**by** (*assumption+*, *simp*, *assumption+*)

**lemma** *P1-f-n-CASE*:

$\llbracket E1\ x = \text{Some}\ (\text{Val.Loc}\ p);$   
 $(E1, E2) \vdash h, k, \text{Case}\ \text{VarE}\ x\ a\ \text{Of}\ \text{alts}\ a' \Downarrow (f, n)\ hh, k, v \rrbracket$   
 $\implies \exists j\ C\ \text{vs.}\ h\ p = \text{Some}\ (j, C, \text{vs}) \wedge$   
 $(\exists i < \text{length}\ \text{alts}.$   
 $((\text{extend}\ E1\ (\text{snd}\ (\text{extractP}\ (\text{fst}\ (\text{alts}\ !\ i))))\ \text{vs}, E2)$   
 $\vdash h, k, \text{snd}\ (\text{alts}\ !\ i) \Downarrow (f, n)\ hh, k, v$   
 $\wedge \text{def-extend}\ E1\ (\text{snd}\ (\text{extractP}\ (\text{fst}\ (\text{alts}\ !\ i))))\ \text{vs}) \wedge$   
 $(\exists\ \text{pati}\ ei\ ps\ ms.$   
 $\text{alts}\ !\ i = (\text{pati}, ei) \wedge$   
 $\text{pati} = \text{ConstrP}\ C\ ps\ ms))$   
**apply** (*simp add: SafeBoundSem-def*)  
**apply** (*elim exE, elim conjE*)  
**apply** (*erule SafeDepthSem.cases, simp-all*)  
**by** *force*

**lemma** *fvTup-subseteq-fvAlts*:

$\forall i < \text{length}\ \text{alts}. \text{fvTup}\ (\text{alts}!i) \subseteq \text{fvAlts}\ \text{alts}$   
**apply** (*rule allI, rule impI*)  
**apply** (*rule subsetI*)  
**apply** (*induct alts arbitrary: i*)  
**apply** *simp*  
**apply** (*case-tac i*)  
**apply** *simp*  
**by** *clarsimp*

**lemma** *SafeRegion-CASE-fv-P1'* [*rule-format*]:

$\text{length}\ \text{alts} > 0$   
 $\longrightarrow \text{fv}\ (\text{Case}\ \text{VarE}\ x\ a\ \text{Of}\ \text{alts}\ a') \subseteq \text{dom}\ E1$   
 $\longrightarrow (\forall i < \text{length}\ \text{alts}. \forall\ \text{vs}.$   
 $\text{def-extend}\ E1\ (\text{snd}\ (\text{extractP}\ (\text{fst}\ (\text{alts}\ !\ i))))\ \text{vs}$   
 $\longrightarrow \text{fv}\ (\text{snd}\ (\text{alts}\ !\ i)) \subseteq \text{dom}\ (\text{extend}\ E1\ (\text{snd}\ (\text{extractP}\ (\text{fst}\ (\text{alts}\ !\ i))))\ \text{vs}))$   
**apply** (*induct alts arbitrary: i, simp-all*)  
**apply** (*rule impI*)  
**apply** (*elim conjE*)  
**apply** (*rule allI*)  
**apply** (*rule impI*)  
**apply** (*case-tac alts = []*, *simp-all*)

```

apply (rule allI, rule impI)
apply (simp add: def-extend-def)
apply (case-tac a, simp-all)
apply (elim conjE)
apply (simp add: extend-def)
apply blast
apply (rule allI, rule impI)
apply (case-tac i, simp-all)
apply (case-tac a, simp-all)
apply (simp add: def-extend-def)
apply (erule-tac x=0 in allE, simp)
apply (erule-tac x=vs in allE)
apply (simp add: extend-def)
by blast

```

```

lemma SafeRegion-CASEL-LitN-fv-P1' [rule-format]:
  length alts > 0
   $\longrightarrow$  fv (Case VarE x a Of alts a')  $\subseteq$  dom E1
   $\longrightarrow$  ( $\forall$  i < length alts.
    fst (alts ! i) = ConstP (LitN n)
     $\longrightarrow$  fv (snd (alts ! i))  $\subseteq$  dom E1)
apply (induct alts arbitrary: i, simp-all)
apply (rule impI)+
apply (elim conjE)
apply (rule allI)
apply (rule impI)+
apply (case-tac alts = [], simp-all)
  apply (case-tac a, simp-all)
apply (case-tac i, simp-all)
by (case-tac a, simp-all)

```

```

lemma SafeRegion-CASEL-LitB-fv-P1' [rule-format]:
  length alts > 0
   $\longrightarrow$  fv (Case VarE x a Of alts a')  $\subseteq$  dom E1
   $\longrightarrow$  ( $\forall$  i < length alts.
    fst (alts ! i) = ConstP (LitB b)
     $\longrightarrow$  fv (snd (alts ! i))  $\subseteq$  dom E1)
apply (induct alts arbitrary: i, simp-all)
apply (rule impI)+
apply (elim conjE)
apply (rule allI)
apply (rule impI)+
apply (case-tac alts = [], simp-all)
  apply (case-tac a, simp-all)
apply (case-tac i, simp-all)

```

**by** (*case-tac a, simp-all*)

**lemma** *SafeRegion-CASE-fvReg-P1'*:

$\llbracket \text{fvReg } (\text{Case VarE } x \text{ a Of alts } a') \subseteq \text{dom } E2; i < \text{length alts} \rrbracket$

$\implies \text{fvReg } (\text{snd } (\text{alts } ! i)) \subseteq \text{dom } E2$

**apply** (*induct alts arbitrary: i, simp-all*)

**apply** (*case-tac i, simp*)

**by** (*case-tac a, simp-all*)

**lemma** *SafeRegion-CASE-E1-P2*:

$\llbracket \text{dom } E1 \subseteq \text{dom } \vartheta 1;$

$i < \text{length alts};$

$\forall i < \text{length alts}.$

$\text{constructorSignature } (\text{fst } (\text{extractP } (\text{fst } (\text{alts } ! i)))) = \text{Some } (ti, \varrho, t) \wedge$

$t = \text{TypeExpression.ConstrT } T \text{ tn } \varrho s \wedge$

$\text{length } (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! i)))) = \text{length } ti \wedge$

$\text{wellT } ti \ \varrho \ t \wedge$

$\text{fst } (\text{assert } ! i) = \vartheta 1 ++ \text{map-of } (\text{zip } (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! i))))$

$(\text{map } (\text{mu-ext } \mu) \text{ ti})) \wedge$

$\text{dom } \vartheta 1 \cap \text{dom } (\text{map-of } (\text{zip } (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! i))))$

$(\text{map } (\text{mu-ext } \mu) \text{ ti})) = \{\}$   $\wedge$

$\vartheta 1 \ x = \text{Some } (\text{mu-ext } \mu \ t) \wedge \text{snd } (\text{assert } ! i) = \vartheta 2;$

$\text{def-extend } E1 \ (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! i)))) \text{ vs};$

$\forall i < \text{length alts}. x \notin \text{set } (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! i)))) \rrbracket$

$\implies \text{dom } (\text{extend } E1 \ (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! i)))) \text{ vs} \subseteq \text{dom } (\text{fst } (\text{assert } ! i))$

**apply** (*simp only: def-extend-def*)

**apply** (*rule subsetI*)

**apply** (*erule-tac x=i in allE*) $+$

**apply** (*simp del: dom-map-add*)

**apply** (*elim conjE*)

**apply** (*frule-tac E=E1 and vs=vs in extend-monotone*)

**apply** (*simp only: extend-def*)

**apply** (*subst dom-map-add, simp*)

**apply** (*erule disjE*)

**apply** *simp*

**apply** (*rule disjI2*)

**by** *blast*

**lemma** *SafeRegion-CASEL-E1-P2*:

$\llbracket \text{dom } E1 \subseteq \text{dom } \vartheta 1;$

$i < \text{length alts};$

$\forall i < \text{length alts}.$

$\text{constructorSignature } (\text{fst } (\text{extractP } (\text{fst } (\text{alts } ! i)))) = \text{Some } (ti, \varrho, t) \wedge$

$t = \text{TypeExpression.ConstrT } T \text{ tn } \varrho s \wedge$   
 $\text{length } (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! i)))) = \text{length } ti \wedge$   
 $\text{wellT } ti \varrho t \wedge$   
 $\text{fst } (\text{assert } ! i) = \vartheta 1 ++ \text{map-of } (\text{zip } (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! i))))$   
 $\quad (\text{map } (\text{mu-ext } \mu) \text{ ti})) \wedge$   
 $\text{dom } \vartheta 1 \cap \text{dom } (\text{map-of } (\text{zip } (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! i))))$   
 $\quad (\text{map } (\text{mu-ext } \mu) \text{ ti})) = \{\} \wedge$   
 $\vartheta 1 \text{ x} = \text{Some } (\text{mu-ext } \mu \text{ t}) \wedge \text{snd } (\text{assert } ! i) = \vartheta 2;$   
 $\forall i < \text{length } \text{alts}. x \notin \text{set } (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! i)))) \parallel$   
 $\implies \text{dom } E1 \subseteq \text{dom } (\text{fst } (\text{assert } ! i))$   
**apply** (*rule subsetI*)  
**apply** (*erule-tac x=i in allE*)  
**apply** (*simp del: dom-map-add*)  
**apply** (*elim conjE*)  
**apply** (*simp only: extend-def*)  
**apply** (*subst dom-map-add, simp*)  
**apply** (*rule disjI2*)  
**by** *blast*

**lemma** *in-set-conv-nth-2*:  $(x \in \text{set } xs) \implies (\exists i < \text{length } xs. xs[i] = x)$   
**by** (*auto simp: set-conv-nth*)

**lemma** *x-in-variables-same-μ*:  
 $(x \in \text{variables } t \wedge \text{mu-ext } \mu' t = \text{mu-ext } \mu t \longrightarrow \text{the } (\text{fst } \mu x) = \text{the } (\text{fst } \mu' x))$   
 $\wedge$   
 $(x \in \text{variables}' tm \wedge \text{mu-exts } \mu' tm = \text{mu-exts } \mu tm \longrightarrow \text{the } (\text{fst } \mu x) = \text{the } (\text{fst } \mu' x))$   
**apply** (*induct-tac t and tm, simp-all*)  
**by** *clarsimp*

**lemma** *x-in-regions-same-μ*:  
 $(x \in \text{regions } t \cup \text{set } \varrho s \wedge (\forall x \in \text{set } \varrho s. \text{the } (\text{snd } \mu' x) = \text{the } (\text{snd } \mu x))$   
 $\quad \wedge \text{mu-ext } \mu' t = \text{mu-ext } \mu t$   
 $\longrightarrow \text{the } (\text{snd } \mu x) = \text{the } (\text{snd } \mu' x)) \wedge$   
 $(x \in \text{regions}' tm \cup \text{set } \varrho s \wedge (\forall x \in \text{set } \varrho s. \text{the } (\text{snd } \mu' x) = \text{the } (\text{snd } \mu x))$   
 $\quad \wedge \text{mu-exts } \mu' tm = \text{mu-exts } \mu tm$   
 $\longrightarrow \text{the } (\text{snd } \mu x) = \text{the } (\text{snd } \mu' x))$   
**apply** (*induct-tac t and tm, simp-all*)  
**by** *clarsimp*

**lemma** *regions-variables-same-μ*:  
 $(\text{regions } t \subseteq \text{regions}' tm' \cup \text{set } \varrho s \wedge (\forall x \in \text{set } \varrho s. \text{the } (\text{snd } \mu' x) = \text{the } (\text{snd } \mu x))$   
 $\quad \wedge$

$variables\ t \subseteq variables'\ tm' \wedge mu-exts\ \mu' tm' = mu-exts\ \mu tm'$   
 $\longrightarrow mu-ext\ \mu t = mu-ext\ \mu' t) \wedge$   
 $(regions'\ tm \subseteq regions'\ tm' \cup set\ \varrho s \wedge (\forall x \in set\ \varrho s. the\ (snd\ \mu' x) = the\ (snd\ \mu x)) \wedge$   
 $variables'\ tm \subseteq variables'\ tm' \wedge mu-exts\ \mu' tm' = mu-exts\ \mu tm'$   
 $\longrightarrow mu-exts\ \mu tm = mu-exts\ \mu' tm)$   
**apply** (induct-tac t **and** tm,simp-all)  
**apply** (rename-tac x)  
**apply** (subgoal-tac  
 $(x \in variables\ t \wedge mu-ext\ \mu' t = mu-ext\ \mu t \longrightarrow the\ (fst\ \mu x) = the\ (fst\ \mu' x))$   
 $\wedge$   
 $(x \in variables'\ tm' \wedge mu-exts\ \mu' tm' = mu-exts\ \mu tm' \longrightarrow the\ (fst\ \mu x) = the\ (fst\ \mu' x)))$   
**prefer** 2 **apply** (rule x-in-variables-same- $\mu$ )  
**apply** simp  
**apply** clarsimp  
**apply** (subgoal-tac  
 $(x \in regions\ t \cup set\ \varrho s \wedge (\forall x \in set\ \varrho s. the\ (snd\ \mu' x) = the\ (snd\ \mu x))$   
 $\wedge mu-ext\ \mu' t = mu-ext\ \mu t$   
 $\longrightarrow the\ (snd\ \mu x) = the\ (snd\ \mu' x)) \wedge$   
 $(x \in regions'\ tm' \cup set\ \varrho s \wedge (\forall x \in set\ \varrho s. the\ (snd\ \mu' x) = the\ (snd\ \mu x))$   
 $\wedge mu-exts\ \mu' tm' = mu-exts\ \mu tm'$   
 $\longrightarrow the\ (snd\ \mu x) = the\ (snd\ \mu' x)))$   
**prefer** 2 **apply** (rule x-in-regions-same- $\mu$ )  
**by** force

**lemma** same- $\mu$ :

$\llbracket constructorSignature\ C = Some\ (tn,\ \varrho,\ ConstrT\ T\ tm\ \varrho s);$   
 $wellT\ tn\ \varrho\ (TypeExpression.ConstrT\ T\ tm\ \varrho s);$   
 $mu-ext\ \mu\ (ConstrT\ T\ tm\ \varrho s) = t';$   
 $mu-ext\ \mu'\ (ConstrT\ T\ tm\ \varrho s) = t' \rrbracket$   
 $\implies map\ (mu-ext\ \mu)\ tn = map\ (mu-ext\ \mu')\ tn$   
**apply** (simp only: wellT.simps)  
**apply** (elim conjE)  
**apply** (induct-tac tm, simp-all)  
**apply** (rule ballI)  
**apply** (drule in-set-conv-nth-2)  
**apply** (elim exE, elim conjE)  
**apply** (erule-tac x=i in allE, simp)  
**apply** (elim conjE, clarsimp)  
**apply** (subgoal-tac  
 $(regions\ (tn\ !\ i) \subseteq regions'\ tm \cup set\ \varrho s \wedge (\forall x \in set\ \varrho s. the\ (snd\ \mu' x) =$   
 $the\ (snd\ \mu x)) \wedge$   
 $variables\ (tn\ !\ i) \subseteq variables'\ tm \wedge mu-exts\ \mu' tm = mu-exts\ \mu tm$   
 $\longrightarrow mu-ext\ \mu\ (tn\ !\ i) = mu-ext\ \mu'\ (tn\ !\ i)) \wedge$   
 $(regions'\ tm' \subseteq regions'\ tm \cup set\ \varrho s \wedge (\forall x \in set\ \varrho s. the\ (snd\ \mu' x) =$   
 $the\ (snd\ \mu x)) \wedge$   
 $variables'\ tm' \subseteq variables'\ tm \wedge mu-exts\ \mu' tm = mu-exts\ \mu tm$



$\longrightarrow \text{mu-exts } \mu \text{ } tm' = \text{mu-exts } \mu' \text{ } tm')$   
**prefer 2 apply** (rule regions-variables-same- $\mu$ )  
**by force**

**lemma** *SafeRegion-f-n-CASE-P4*:

$\llbracket \text{length } \text{assert} = \text{length } \text{alts}; \text{length } \text{alts} > 0;$   
 $\forall i < \text{length } \text{alts}.$   
 $\text{constructorSignature } (\text{fst } (\text{extractP } (\text{fst } (\text{alts } ! \ i)))) = \text{Some } (ti, \varrho, t) \wedge$   
 $t = \text{ConstrT } T \text{ } tn \ \varrho s \wedge$   
 $\text{length } (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! \ i)))) = \text{length } ti \wedge$   
 $\text{wellT } ti \ \varrho \ t \wedge$   
 $\text{fst } (\text{assert } ! \ i) = \vartheta 1 \ ++ \ (\text{map-of } (\text{zip } (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! \ i))))$   
 $\quad (\text{map } (\text{mu-ext } \mu) \text{ } ti))) \wedge$   
 $\text{dom } \vartheta 1 \cap \text{dom } (\text{map-of } (\text{zip } (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! \ i))))$   
 $\quad (\text{map } (\text{mu-ext } \mu) \text{ } ti))) = \{\}$   $\wedge$   
 $\vartheta 1 \ x = \text{Some } (\text{mu-ext } \mu \ t) \wedge$   
 $\text{snd } (\text{assert } ! \ i) = \vartheta 2;$   
 $\text{consistent } (\vartheta 1, \vartheta 2) \ \eta \ (E1, E2) \ h; E1 \ x = \text{Some } (\text{Loc } p); h \ p = \text{Some}(j, C, vs);$

$i < \text{length } \text{alts};$   
 $(\text{extend } E1 \ (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! \ i)))) \text{ } vs, E2)$   
 $\vdash h, k, \text{snd } (\text{alts } ! \ i) \Downarrow (f, n) \ h', k, v;$   
 $\text{def-extend } E1 \ (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! \ i)))) \text{ } vs;$   
 $\text{alts } ! \ i = (\text{pati}, ei);$   
 $\text{pati} = \text{ConstrP } C \text{ } ps \ ms \rrbracket$   
 $\implies \text{consistent } (\text{fst } (\text{assert } ! \ i), \text{snd } (\text{assert } ! \ i)) \ \eta$   
 $\quad (\text{extend } E1 \ (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! \ i)))) \text{ } vs, E2) \ h$

**apply** (erule-tac  $x=i$  **in**  $\text{allE}, \text{simp}$ )  
**apply** (elim conjE)  
**apply** clarsimp  
**apply** (simp add: Let-def)  
**apply** (simp add: consistent.simps)

**apply** (rule ballI)  
**apply** (simp add: extend-def)  
**apply** (erule disjE)

**apply** (elim conjE)  
**apply** (erule-tac  $x=x$  **in** ballE)  
**prefer 2 apply** (simp add: dom-def)  
**apply** (elim exE, elim conjE)  
**apply** (elim exE, elim conjE)  
**apply** simp  
**apply** (erule consistent-v.cases)

```

apply simp

apply simp

apply simp

apply (simp add: dom-def)

apply (case-tac  $\mu$ , simp)
apply (elim exE, elim conjE)
apply (subgoal-tac  $qs' \neq []$ )
  prefer 2 apply (simp add: wellT.simps)
apply (frule-tac  $? \mu 2.0 = \mu 2$  in mu-last, simp)
apply (case-tac  $vs = []$ , simp)
apply (unfold def-extend-def)
apply (elim conjE)
apply (frule-tac  $\mu = \mu$  and  $\mu' = (\mu 1, \mu 2)$  in same- $\mu$ , simp, simp, simp)
apply (subgoal-tac  $xa \in \text{set } (\text{map pat2var ps})$ )
  prefer 2 apply clarsimp
apply (frule-tac  $x = xa$  and  $vs = (\text{map } (\text{mu-ext } \mu) \text{ ti})$  and  $zs = vs$  in map-of-zip-twice-is-Some)

apply (simp, simp, simp, simp)
apply (elim exE, elim conjE)
apply (erule-tac  $x = ia$  in allE)
apply (drule mp, simp)
apply (rule-tac  $x = (\text{mu-ext } \mu (\text{ti} \text{ ! } ia))$  in exI, simp)

apply (elim conjE)
apply (erule-tac  $x = xa$  in ballE)
  prefer 2 apply simp
apply (elim exE, elim conjE)
apply (elim exE, elim conjE)
apply (rule-tac  $x = t$  in exI)
apply (rule conjI)
  apply (rule map-add-fst-Some, assumption+)
apply (rule-tac  $x = va$  in exI)
apply (rule conjI)
  apply (simp add: def-extend-def)
apply (subgoal-tac  $E1 \text{ xa} = (\text{extend } E1 (\text{map pat2var ps}) \text{ vs}) \text{ xa}$ )
  apply (simp add: extend-def)
apply (rule extend-monotone, force)
by assumption+

```

**lemma** *SafeRegion-CASEL-LitB-P4*:  
 $\llbracket \text{length assert} = \text{length alts}; \text{length alts} > 0; \forall i < \text{length alts}. \dots \rrbracket$

$constructorSignature (fst (extractP (fst (alts ! i)))) = Some (ti, \varrho, t) \wedge$   
 $t = ConstrT\ T\ tn\ \varrho s \wedge$   
 $length (snd (extractP (fst (alts ! i)))) = length\ ti \wedge$   
 $wellT\ ti\ \varrho\ t \wedge$   
 $fst (assert ! i) = \vartheta 1 ++ (map-of (zip (snd (extractP (fst (alts ! i))))$   
 $\quad (map (mu-ext\ \mu)\ ti))) \wedge$   
 $dom\ \vartheta 1 \cap dom (map-of (zip (snd (extractP (fst (alts ! i))))$   
 $\quad (map (mu-ext\ \mu)\ ti))) = \{\}$   
 $\vartheta 1\ x = Some (mu-ext\ \mu\ t) \wedge$   
 $snd (assert ! i) = \vartheta 2;$   
 $consistent (\vartheta 1, \vartheta 2)\ \eta\ (E1, E2)\ h; E1\ x = Some (BoolT\ b);$   
 $i < length\ alts;$   
 $fst (alts ! i) = ConstP (LitB\ b)]$   
 $\implies consistent (fst (assert ! i), snd (assert!i))\ \eta\ (E1, E2)\ h$   
**apply** (erule-tac  $x=i$  **in**  $allE, simp$ )+  
**done**

**lemma** *SafeRegion-CASEL-LitN-P4*:

$\llbracket length\ assert = length\ alts; length\ alts > 0;$   
 $\forall i < length\ alts.$   
 $constructorSignature (fst (extractP (fst (alts ! i)))) = Some (ti, \varrho, t) \wedge$   
 $t = ConstrT\ T\ tn\ \varrho s \wedge$   
 $length (snd (extractP (fst (alts ! i)))) = length\ ti \wedge$   
 $wellT\ ti\ \varrho\ t \wedge$   
 $fst (assert ! i) = \vartheta 1 ++ (map-of (zip (snd (extractP (fst (alts ! i))))$   
 $\quad (map (mu-ext\ \mu)\ ti))) \wedge$   
 $dom\ \vartheta 1 \cap dom (map-of (zip (snd (extractP (fst (alts ! i))))$   
 $\quad (map (mu-ext\ \mu)\ ti))) = \{\}$   
 $\vartheta 1\ x = Some (mu-ext\ \mu\ t) \wedge$   
 $snd (assert ! i) = \vartheta 2;$   
 $consistent (\vartheta 1, \vartheta 2)\ \eta\ (E1, E2)\ h; E1\ x = Some (IntT\ n);$   
 $i < length\ alts;$   
 $fst (alts ! i) = ConstP (LitN\ n)]$   
 $\implies consistent (fst (assert ! i), snd (assert!i))\ \eta\ (E1, E2)\ h$   
**apply** (erule-tac  $x=i$  **in**  $allE, simp$ )+  
**done**

**lemma** *SafeDARegionDepth-CASE*:

$\llbracket length\ assert = length\ alts; length\ alts > 0;$   
 $\forall i < length\ alts. constructorSignature (fst (extractP (fst (alts ! i))))$   
 $\quad = Some (ti, \varrho, t) \wedge$   
 $\quad t = ConstrT\ T\ tn\ \varrho s \wedge$   
 $\quad length (snd (extractP (fst (alts ! i)))) = length\ ti \wedge$   
 $\quad wellT\ ti\ \varrho\ t \wedge$   
 $fst (assert ! i) = \vartheta 1 ++ (map-of (zip (snd (extractP (fst (alts ! i))))$

$$\begin{aligned}
& (\text{map } (\mu\text{-ext } \mu) \text{ ti})) \wedge \\
& \text{dom } \vartheta 1 \cap \text{dom } (\text{map-of } (\text{zip } (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! i)))) \\
& \quad (\text{map } (\mu\text{-ext } \mu) \text{ ti})) = \{\} \wedge \\
& \vartheta 1 \text{ } x = \text{Some } (\mu\text{-ext } \mu \text{ } t) \wedge \\
& \text{snd } (\text{assert } ! i) = \vartheta 2; \\
& \forall i < \text{length alts. } \text{snd } (\text{alts } ! i) :_f, n \llbracket (\text{fst } (\text{assert!}i), \\
& \quad \text{snd } (\text{assert!}i)), t' \rrbracket; \\
& \forall i < \text{length alts. } x \notin \text{set } (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! i)))) \rrbracket \\
\implies & \text{Case } (\text{VarE } x \text{ } a) \text{ Of alts } a' :_f, n \llbracket (\vartheta 1, \vartheta 2), t' \rrbracket \\
\text{apply } & (\text{unfold SafeRegionDAssDepth.simps}) \\
\text{apply } & (\text{intro allI, rule impI}) \\
\text{apply } & (\text{elim conjE}) \\
\text{apply } & (\text{case-tac E1 } x)
\end{aligned}$$

**apply** (simp add: dom-def)

**apply** (case-tac aa)

**apply** (rename-tac p)

**apply** (frule impSemBoundRA [where e=Case VarE x a Of alts a' and td=td])

**apply** (elim exE)

**apply** (subgoal-tac

$$\begin{aligned}
& \exists j \text{ } C \text{ vs. } h \text{ } p = \text{Some } (j, C, \text{vs}) \wedge \\
& (\exists i < \text{length alts.} \\
& \quad ((\text{extend E1 } (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! i)))) \text{ vs, E2}) \\
& \quad \vdash h, k, \text{snd } (\text{alts } ! i) \Downarrow (f, n) h', k, v \\
& \quad \wedge \text{def-extend E1 } (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! i)))) \text{ vs}) \wedge \\
& \quad (\exists \text{ pati ei ps ms.} \\
& \quad \quad \text{alts } ! i = (\text{pati, ei}) \wedge \\
& \quad \quad \text{pati} = \text{ConstrP } C \text{ ps ms}))) \\
\text{prefer } & 2 \text{ apply (rule P1-f-n-CASE) apply simp apply simp} \\
\text{apply } & (\text{elim exE, elim conjE}) \\
\text{apply } & (\text{elim exE, elim conjE}) + \\
\text{apply } & (\text{rotate-tac } \mathcal{I}) \\
\text{apply } & (\text{erule-tac } x=i \text{ in allE}) \\
\text{apply } & (\text{drule mp, simp})
\end{aligned}$$

**apply** (erule-tac x=extend E1 (snd (extractP (fst (alts ! i)))) vs in allE)

**apply** (erule-tac x=E2 in allE)

**apply** (erule-tac x=h in allE)

**apply** (rotate-tac 20)

**apply** (erule-tac x=k in allE)

**apply** (erule-tac x=h' in allE)

**apply** (erule-tac x=v in allE)

**apply** (erule-tac x= $\eta$  in allE)

**apply** (*drule mp*)

**apply** (*rule conjI,simp*)

**apply** (*rule conjI*)  
**apply** (*rule SafeRegion-CASE-fv-P1',assumption+*)

**apply** (*rule conjI*)  
**apply** (*rule SafeRegion-CASE-fvReg-P1',assumption+*)

**apply** (*rule conjI*)  
**apply** (*rule SafeRegion-CASE-E1-P2,assumption+*)

**apply** (*rule conjI*)  
**apply** *clarsimp*

**apply** (*rule conjI*)  
**apply** *assumption*

**apply** (*rule SafeRegion-f-n-CASE-P4*)  
**apply** (*assumption+,simp,assumption+*)

**apply** (*frule impSemBoundRA [where e=Case VarE x a Of alts a' and td=td]*)  
**apply** (*elim exE*)

**apply** (*subgoal-tac*  
( $\exists i < \text{length } \text{alts}.$   
    ( $E1, E2 \vdash h, k, \text{snd } (\text{alts } ! i) \Downarrow (f, n) h', k, v \wedge$   
         $\text{fst } (\text{alts } ! i) = \text{ConstP } (\text{LitN } \text{int})$ )))  
**prefer** 2 **apply** (*rule P1-f-n-CASE-1-1,simp,simp*)  
**apply** (*elim exE,elim conjE*)

**apply** (*rotate-tac 3*)  
**apply** (*erule-tac x=i in allE*)  
**apply** (*drule mp, simp*)

**apply** (*erule-tac x=E1 in allE*)  
**apply** (*erule-tac x=E2 in allE*)  
**apply** (*erule-tac x=h in allE*)

**apply** (*rotate-tac 17*)  
**apply** (*erule-tac x=k in allE*)  
**apply** (*erule-tac x=h' in allE*)  
**apply** (*erule-tac x=v in allE*)  
**apply** (*erule-tac x= $\eta$  in allE*)

**apply** (*drule mp*)

**apply** (*rule conjI,simp*)

**apply** (*rule conjI*)  
**apply** (*rule SafeRegion-CASEL-LitN-fv-P1',assumption+*)

**apply** (*rule conjI*)  
**apply** (*rule SafeRegion-CASE-fvReg-P1',assumption+*)

**apply** (*rule conjI*)  
**apply** (*rule SafeRegion-CASEL-E1-P2,assumption+*)

**apply** (*rule conjI*)  
**apply** *clarsimp*

**apply** (*rule conjI*)  
**apply** *assumption*

**apply** *simp*

**apply** *simp*

**apply** (*frule impSemBoundRA [where e=Case VarE x a Of alts a' and td=td]*)  
**apply** (*elim exE*)

**apply** (*subgoal-tac*  
 $(\exists i < \text{length alts}.$   
 $(E1,E2) \vdash h,k, \text{snd} (\text{alts} ! i) \Downarrow(f,n) h',k,v$   
 $\wedge \text{fst} (\text{alts} ! i) = \text{ConstP} (\text{LitB bool}))$ )  
**prefer 2 apply** (*rule P1-f-n-CASE-1-2*) **apply** *simp* **apply** *force*  
**apply** (*elim exE,elim conjE*)

**apply** (*rotate-tac* 3)  
**apply** (*erule-tac*  $x=i$  **in** *allE*)  
**apply** (*drule* *mp*, *simp*)

**apply** (*erule-tac*  $x=E1$  **in** *allE*)  
**apply** (*erule-tac*  $x=E2$  **in** *allE*)  
**apply** (*erule-tac*  $x=h$  **in** *allE*)  
**apply** (*rotate-tac* 17)  
**apply** (*erule-tac*  $x=k$  **in** *allE*)  
**apply** (*erule-tac*  $x=h'$  **in** *allE*)  
**apply** (*erule-tac*  $x=v$  **in** *allE*)  
**apply** (*erule-tac*  $x=\eta$  **in** *allE*)

**apply** (*drule* *mp*)

**apply** (*rule* *conjI*,*simp*)

**apply** (*rule* *conjI*)  
**apply** (*rule* *SafeRegion-CASEL-LitB-fv-P1'*,*assumption*+)

**apply** (*rule* *conjI*)  
**apply** (*rule* *SafeRegion-CASE-fvReg-P1'*,*assumption*+)

**apply** (*rule* *conjI*)  
**apply** (*rule* *SafeRegion-CASEL-E1-P2*,*assumption*+)

**apply** (*rule* *conjI*)  
**apply** *clarsimp*

**apply** (*rule* *conjI*)  
**apply** *assumption*

**apply** *simp*

**by** *simp*

**lemma** *SafeRegion-f-n-CASED-P1*:

$\llbracket (E1, E2) \vdash h, k, \text{CaseD } (\text{VarE } x \ a) \text{ Of } \text{alts } a \Downarrow (f, n) \ h', k, v \rrbracket$   
 $\implies \exists \ p \ j \ C \ \text{vs. } E1 \ x = \text{Some } (\text{Loc } p) \wedge h \ p = \text{Some } (j, C, \text{vs}) \wedge$   
 $(\exists \ i < \text{length } \text{alts}.$   
 $(\text{extend } E1 \ (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! \ i)))) \ \text{vs}, E2) \vdash h(p := \text{None}) , k ,$   
 $\text{snd } (\text{alts } ! \ i) \Downarrow (f, n) \ h' , k , v$   
 $\wedge \text{def-extend } E1 \ (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! \ i)))) \ \text{vs} \wedge$   
 $(\exists \ \text{pati } ei \ ps \ ms.$   
 $\text{alts } ! \ i = (\text{pati}, ei) \wedge$   
 $\text{pati} = \text{ConstrP } C \ ps \ ms))$   
**apply** (*simp add: SafeBoundSem-def*)  
**apply** (*elim exE, elim conjE*)  
**apply** (*erule SafeDepthSem.cases, simp-all*)  
**by** *force*

**lemma** *fvTup'-subsesteq-fvAlts'*:

$i < \text{length } \text{alts}$   
 $\implies \text{fvTup}'(\text{alts}!i) \subseteq \text{fvAlts}' \ \text{alts}$   
**apply** (*rule subsetI*)  
**apply** (*induct alts arbitrary: i*)  
**apply** *simp*  
**apply** (*case-tac i*)  
**apply** *simp*  
**by** *clarsimp*

**lemma** *SafeRegion-CASED-fv-P1'* [*rule-format*]:

$\text{length } \text{alts} > 0$   
 $\longrightarrow \text{fv } (\text{CaseD } \text{VarE } x \ a \text{ Of } \text{alts } a') \subseteq \text{dom } E1$   
 $\longrightarrow (\forall \ i < \text{length } \text{alts}. \forall \ \text{vs}.$   
 $\text{def-extend } E1 \ (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! \ i)))) \ \text{vs}$   
 $\longrightarrow \text{fv } (\text{snd } (\text{alts } ! \ i)) \subseteq \text{dom } (\text{extend } E1 \ (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! \ i)))) \ \text{vs}))$   
**apply** (*induct alts arbitrary: i, simp-all*)  
**apply** (*rule impI*)  
**apply** (*elim conjE*)  
**apply** (*rule allI*)  
**apply** (*rule impI*)  
**apply** (*case-tac alts = [], simp-all*)  
**apply** (*rule allI, rule impI*)  
**apply** (*simp add: def-extend-def*)  
**apply** (*case-tac a, simp-all*)  
**apply** (*elim conjE*)  
**apply** (*simp add: extend-def*)  
**apply** *blast*  
**apply** (*rule allI, rule impI*)  
**apply** (*case-tac i, simp-all*)



```

apply (case-tac a, simp-all)
apply (simp add: def-extend-def)
apply (erule-tac x=0 in allE, simp)
apply (erule-tac x=vs in allE)
apply (simp add: extend-def)
by blast

```

```

lemma SafeRegion-CASED-fvReg-P1':
  [| fvReg (CaseD VarE x a Of alts a')  $\subseteq$  dom E2; i < length alts |]
   $\implies$  fvReg (snd (alts ! i))  $\subseteq$  dom E2
apply (induct alts arbitrary: i, simp-all)
apply (case-tac i, simp)
by (case-tac a, simp-all)

```

```

lemma consistent-v-p-none [rule-format]:
  consistent-v t  $\eta$  v h
   $\longrightarrow$  p  $\notin$  closureV v (h,k)
   $\longrightarrow$  coherent constructorSignature Tc
   $\longrightarrow$  consistent-v t  $\eta$  v (h(p := None))
apply (rule impI)
apply (erule consistent-v.induct, simp-all)

```

```

apply (clarsimp, rule consistent-v.primitiveI)

```

```

apply (clarsimp, rule consistent-v.primitiveB)

```

```

apply (clarsimp, rule consistent-v.variable)

```

```

apply clarsimp
apply (rule consistent-v.algebraic-None)
apply (simp add: dom-def)

```

```

apply (elim exE, elim conjE)
apply (rule impI)+
apply (rule consistent-v.algebraic)

```

```

apply (case-tac pa  $\neq$  p, force)
apply (simp add: closureV-def)
apply (subgoal-tac p  $\in$  closureL p (h, k))
apply simp
apply (rule closureL.closureL-basic)

```

```

apply force

```

```

apply force

```

```

apply force

apply force

apply force

apply force

apply (rule-tac  $x=\mu 1$  in exI)
apply (rule-tac  $x=\mu 2$  in exI)
apply (rule conjI)
  apply force
apply (rule allI,rule impI)
apply (erule-tac  $x=i$  in allE,simp)
apply (elim conjE)
apply (simp add: coherent-def)
apply (elim conjE)
apply (erule-tac  $x=C$  in ballE)
apply (frule p-notin-closure V-q-notin-closure V-vn)
apply (assumption+,simp+)
apply force
done

```

**lemma** *consistent-v-p-none-x-in-dom-E1:*

```

  consistent-v t  $\eta$  v h
   $\implies$  consistent-v t  $\eta$  v ( $h(p := \text{None})$ )
apply (case-tac v,simp-all)

```

```

apply (rename-tac q)
apply (case-tac q = p)

```

```

apply (rule consistent-v.algebraic-None)
apply (simp add: dom-def)

```

```

apply (erule consistent-v.induct)

```

```

  apply (clarsimp, rule consistent-v.primitiveI)

```

```

  apply (clarsimp, rule consistent-v.primitiveB)

```

```

  apply (clarsimp, rule consistent-v.variable)

```

```

apply (case-tac pa=p)
apply (rule consistent-v.algebraic-None,force)
apply (rule consistent-v.algebraic-None,force)

```

```

apply (elim exE, elim conjE)
apply (case-tac pa = p)
apply (rule consistent-v.algebraic-None,force)
apply (rule consistent-v.algebraic)

apply force

apply force

apply force

apply force

apply force

apply force

apply (rule-tac x=μ1 in exI)
apply (rule-tac x=μ2 in exI)
apply (rule conjI)
  apply force
apply force

apply (erule consistent-v.cases,simp-all)
apply (rule consistent-v.primitiveI)
apply (rule consistent-v.variable)

apply (erule consistent-v.cases,simp-all)
apply (rule consistent-v.primitiveB)
apply (rule consistent-v.variable)
done

lemma SafeRegion-f-n-CASED-P4:
   $\llbracket \text{length } \text{assert} = \text{length } \text{alts}; \text{length } \text{alts} > 0; \\
  \text{coherent constructorSignature } Tc; \\
  \forall i < \text{length } \text{alts}. \\
  \text{constructorSignature } (\text{fst } (\text{extractP } (\text{fst } (\text{alts } ! i)))) = \text{Some } (ti, \varrho, t) \wedge \\
  t = \text{ConstrT } T \text{ tn } \varrho s \wedge \\
  \text{length } (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! i)))) = \text{length } ti \wedge \\
  \text{wellT } ti \ \varrho \ t \wedge \\
  \text{fst } (\text{assert } ! i) = \vartheta 1 \ ++ \ (\text{map-of } (\text{zip } (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! i)))) \\
  (\text{map } (\text{mu-ext } \mu) \ ti))) \wedge \\
  \text{dom } \vartheta 1 \cap \text{dom } (\text{map-of } (\text{zip } (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! i))))$ 

```

$(\text{map } (\text{mu-ext } \mu) \text{ } ti))) = \{\} \wedge$   
 $\vartheta 1 \ x = \text{Some } (\text{mu-ext } \mu \ t) \wedge$   
 $\text{snd } (\text{assert } ! \ i) = \vartheta 2;$   
 $\text{consistent } (\vartheta 1, \vartheta 2) \ \eta \ (E1, E2) \ h; E1 \ x = \text{Some } (\text{Loc } p); h \ p = \text{Some}(j, C, vs);$

$i < \text{length } \text{alts};$   
 $(\text{extend } E1 \ (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! \ i)))) \ vs, E2) \vdash h$   
 $(p := \text{None}) , k , \text{snd } (\text{alts } ! \ i) \Downarrow (f, n) \ h' , k , v;$   
 $\text{def-extend } E1 \ (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! \ i)))) \ vs;$   
 $\text{alts } ! \ i = (\text{pati}, \text{ei});$   
 $\text{pati} = \text{ConstrP } C \ ps \ ms \parallel$   
 $\implies \text{consistent } (\text{fst } (\text{assert } ! \ i), \text{snd } (\text{assert } ! \ i)) \ \eta$   
 $(\text{extend } E1 \ (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! \ i)))) \ vs, E2) \ (h(p := \text{None}))$

**apply** (*erule-tac*  $x=i$  **in** *allE, simp*) +  
**apply** (*elim conjE*)  
**apply** *clarsimp*  
**apply** (*simp add: Let-def*)  
**apply** (*simp add: consistent.simps*)

**apply** (*rule ballI*)  
**apply** (*simp add: extend-def*)  
**apply** (*erule disjE*)

**apply** (*elim conjE*)  
**apply** (*erule-tac*  $x=x$  **in** *ballE*)  
**prefer** 2 **apply** (*simp add: dom-def*)  
**apply** (*elim exE, elim conjE*)  
**apply** (*elim exE, elim conjE*)  
**apply** *simp*  
**apply** (*erule consistent-v.cases*)

**apply** *simp*

**apply** *simp*

**apply** *simp*

**apply** (*simp add: dom-def*)

**apply** (*case-tac*  $\mu, \text{simp}$ )  
**apply** (*elim exE, elim conjE*)  
**apply** (*subgoal-tac*  $\varrho s' \neq []$ )  
**prefer** 2 **apply** (*simp add: wellT.simps*)  
**apply** (*frule-tac*  $? \mu 2.0 = \mu 2$  **in** *mu-last, simp*)  
**apply** (*case-tac*  $vs = [], \text{simp}$ )  
**apply** (*unfold def-extend-def*)  
**apply** (*elim conjE*)

```

apply (frule-tac  $\mu=\mu$  and  $\mu'=(\mu 1, \mu 2)$  in same- $\mu$ ,simp,simp,simp)
apply (subgoal-tac  $xa \in \text{set}$  (map pat2var ps))
  prefer 2 apply clarsimp
apply (frule-tac  $x=xa$  and  $vs=(\text{map } (\mu\text{-ext } \mu) \text{ ti})$  and  $zs=vs$  in map-of-zip-twice-is-Some)

apply (simp,simp,simp,simp)
apply (elim exE,elim conjE)
apply (erule-tac  $x=ia$  in allE)
apply (drule mp, simp)
apply (rule-tac  $x=(\mu\text{-ext } \mu (\text{ti} ! ia))$  in exI,simp)
apply (frule no-cycles)
apply (rule consistent-v-p-none)
apply (assumption+,force,assumption)

```

```

apply (elim conjE)
apply (erule-tac  $x=xa$  in ballE)
  prefer 2 apply simp
apply (elim exE,elim conjE)
apply (elim exE,elim conjE)
apply (rule-tac  $x=t$  in exI)
apply (rule conjI)
  apply (rule map-add-fst-Some,assumption+)
apply (rule-tac  $x=va$  in exI)
apply (rule conjI)
  apply (simp add: def-extend-def)
apply (subgoal-tac  $E1 \text{ xa} = (\text{extend } E1 (\text{map pat2var ps}) \text{ vs}) \text{ xa}$ )
  apply (simp add: extend-def)
  apply (rule extend-monotone,force)
by (rule consistent-v-p-none-x-in-dom-E1,simp)

```

**lemma** *SafeDARegionDepth-CASED*:

```

   $\llbracket \text{length assert} = \text{length alts}; \text{length alts} > 0;$ 
  coherent constructorSignature Tc;
   $\forall i < \text{length alts}. \text{constructorSignature} (\text{fst } (\text{extractP } (\text{fst } (\text{alts} ! i))))$ 
     $= \text{Some } (ti, q, t) \wedge$ 
     $t = \text{ConstrT } T \text{ tn } qs \wedge$ 
     $\text{length } (\text{snd } (\text{extractP } (\text{fst } (\text{alts} ! i)))) = \text{length } ti \wedge$ 
     $\text{wellT } ti \text{ } q \text{ } t \wedge$ 
     $\text{fst } (\text{assert} ! i) = \vartheta 1 ++ (\text{map-of } (\text{zip } (\text{snd } (\text{extractP } (\text{fst } (\text{alts} ! i))))$ 
       $(\text{map } (\mu\text{-ext } \mu) \text{ ti}))) \wedge$ 
     $\text{dom } \vartheta 1 \cap \text{dom } (\text{map-of } (\text{zip } (\text{snd } (\text{extractP } (\text{fst } (\text{alts} ! i))))$ 
       $(\text{map } (\mu\text{-ext } \mu) \text{ ti}))) = \{\}$   $\wedge$ 
     $\vartheta 1 \text{ x} = \text{Some } (\mu\text{-ext } \mu \text{ t}) \wedge$ 

```

$\text{snd } (\text{assert } ! i) = \vartheta 2;$   
 $\forall i < \text{length } \text{alts}. \text{snd } (\text{alts } ! i) :_f, n$   
 $\quad \llbracket (\text{fst } (\text{assert}!i), \text{snd } (\text{assert}!i)), t' \rrbracket;$   
 $\forall i < \text{length } \text{alts}. x \notin \text{set } (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! i)))) \rrbracket$   
 $\implies \text{CaseD } (\text{VarE } x a) \text{ Of alts } a' :_f, n \llbracket (\vartheta 1, \vartheta 2), t' \rrbracket$   
**apply** (*unfold SafeRegionDAssDepth.simps*)  
**apply** (*intro allI, rule impI*)  
**apply** (*elim conjE*)

**apply** (*frule impSemBoundRA [where e=CaseD VarE x a Of alts a' and td=td]*)  
**apply** (*elim exE*)

**apply** (*subgoal-tac*)  
 $\exists p \ j \ C \text{ vs. } E1 \ x = \text{Some } (\text{Loc } p) \wedge h \ p = \text{Some } (j, C, \text{vs}) \wedge$   
 $(\exists i < \text{length } \text{alts}.$   
 $\quad (\text{extend } E1 \ (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! i)))) \text{ vs}, E2)$   
 $\quad \vdash h(p := \text{None}) , k , \text{snd } (\text{alts } ! i) \Downarrow (f, n) \ h' , k , v$   
 $\quad \wedge \text{def-extend } E1 \ (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! i)))) \text{ vs} \wedge$   
 $(\exists \text{ pati } \text{ ei } \text{ ps } \text{ ms}.$   
 $\quad \text{alts } ! i = (\text{pati}, \text{ei}) \wedge$   
 $\quad \text{pati} = \text{ConstrP } C \text{ ps } \text{ ms}))$   
**prefer 2 apply** (*rule SafeRegion-f-n-CASED-P1,simp*)  
**apply** (*elim exE,elim conjE*)+

**apply** (*rotate-tac 4*)  
**apply** (*erule-tac x=i in allE*)  
**apply** (*drule mp, simp*)

**apply** (*erule-tac x=extend E1 (snd (extractP (fst (alts ! i)))) vs in allE*)  
**apply** (*erule-tac x=E2 in allE*)  
**apply** (*erule-tac x=h(p:=None) in allE*)  
**apply** (*rotate-tac 20*)  
**apply** (*erule-tac x=k in allE*)  
**apply** (*erule-tac x=h' in allE*)  
**apply** (*erule-tac x=v in allE*)  
**apply** (*erule-tac x=η in allE*)

**apply** (*drule mp*)

**apply** (*rule conjI,simp*)

**apply** (*rule conjI*)  
**apply** (*rule SafeRegion-CASED-fv-P1',assumption+*)

**apply** (*rule conjI*)

**apply** (*rule SafeRegion-CASED-fvReg-P1',assumption+*)

**apply** (*rule conjI*)  
**apply** (*rule SafeRegion-CASE-E1-P2*  
 [where  $ti=ti$  and  $\varrho=\varrho$  and  $t=t$  and  $T=T$  and  $tn=tn$  and  
 $\varrho s=\varrho s$  and  $\mu=\mu$  and  $x=x$  and  $?v2.0=v2$ ])  
**apply** (*simp,simp,force,simp,simp*)

**apply** (*rule conjI, clarsimp*)

**apply** (*rule conjI*)  
**apply** *assumption*

**apply** (*rule SafeRegion-f-n-CASED-P4*  
 [where  $ti=ti$  and  $\varrho=\varrho$  and  $t=t$  and  $T=T$  and  $tn=tn$  and  
 $\varrho s=\varrho s$  and  $\mu=\mu$  and  $x=x$  and  $?v1.0=v1$  and  $?v2.0=v2$ ])  
**by** *assumption+*

**lemma** *SafeDARegion-APP-E1-P2*:  
 [  $\text{length } xs = \text{length } as; \text{length } xs = \text{length } ti; \text{distinct } xs$  ]  
 $\implies \text{dom } (\text{map-of } (\text{zip } xs (\text{map } (\text{atom2val } E1) as))) \subseteq$   
 $\text{dom } (\text{mu-ext } (\text{fst } (\mu\text{-ren } (\mu1, \mu2)), \text{snd } (\mu\text{-ren } (\mu1, \mu2)))(\varrho\text{self} \mapsto \varrho\text{self}))$   
 $\circ_f \text{map-of } (\text{zip } xs ti)$   
**by** (*simp, subst dom-map-f-comp, simp*)

**lemma** *SafeDARegion-APP-E2-P2*:  
 [  $\forall i < \text{length } \varrho s. \exists t. \mu2 (\varrho s!i) = \text{Some } t; \text{distinct } rs;$   
 $\text{length } rs = \text{length } rr; \text{length } rs = \text{length } \varrho s; \mu\text{-ren-dom } (\mu1, \mu2)$  ]  
 $\implies \text{dom } (\text{map-of } (\text{zip } rs (\text{map } (\text{the } \circ E2) rr))(self \mapsto \text{Suc } k)) \subseteq$   
 $\text{dom } (\text{snd } (\mu\text{-ren } (\mu1, \mu2)))(\varrho\text{self} \mapsto \varrho\text{self})$   
 $\circ_m \text{map-of } (\text{zip } rs \varrho s)(self \mapsto \varrho\text{self})$   
**apply** *simp*  
**apply** (*rule conjI*)  
**apply** *force*  
**apply** (*rule subsetI*)  
**apply** (*subgoal-tac*  
 $\text{dom } ((\lambda \varrho. \text{Some } (\varrho\text{-ren } (\text{the } (\mu2 \varrho)))))(\varrho\text{self} \mapsto \varrho\text{self})$ )

```

      ◦m map-of (zip rs ρs)(self ↦ ρself))
    = dom (map-of (zip rs ρs)(self ↦ ρself))
  apply simp
  by (rule dom-map-comp,simp)

```

**lemma** *SafeDARegion-APP-P3*:

```

  admissible η k
  ⇒ admissible (η-ren η ++ [ρself ↦ Suc k]) (Suc k)
  apply (simp add: admissible-def)
  apply (simp add: η-ren-def)
  apply (rule conjI)
  apply (rule impI)
  apply (elim conjE)
  apply (erule-tac x=ρself in ballE)
  prefer 2 apply simp
  apply clarsimp
  apply clarsimp
  apply (rule conjI)
  apply (rule impI)+
  apply (split split-if-asm,simp,simp)
  apply (erule-tac x=ρself in ballE)
  prefer 2 apply (simp add: dom-def)
  apply (simp, split split-if-asm,simp,simp)
  apply (rule impI)
  apply (erule-tac x=ρ in ballE)
  prefer 2 apply (simp add: dom-def)
  by simp

```

**lemma** *μ-ren-extend-ρself*:

```

  (ρself ∉ regions t
    → mu-ext (λx. Some (t-ren (the (μ1 x))), (λρ. Some (ρ-ren (the (μ2 ρ)))))
      (ρself ↦ ρself)) t =
    mu-ext (λx. Some (t-ren (the (μ1 x))), λρ. Some (ρ-ren (the (μ2 ρ))))) t ∧
  (ρself ∉ regions' ts
    → mu-exts (λx. Some (t-ren (the (μ1 x))), (λρ. Some (ρ-ren (the (μ2 ρ)))))
      (ρself ↦ ρself)) ts =
    mu-exts (λx. Some (t-ren (the (μ1 x))), λρ. Some (ρ-ren (the (μ2 ρ))))) ts)
  by (induct-tac t and ts,simp-all)

```



**lemma** *map-ρ-ren:*

$\rho\text{self} \notin \text{set } xs \implies xs = \text{map } \rho\text{-ren } xs$   
**apply** (*induct xs,simp-all*)  
**apply** (*simp add: ρ-ren-def*)  
**by** *force*

**lemma** *t-equals-t-ren:*

$(\rho\text{self} \notin \text{regions } t \wedge \rho\text{self} \notin \text{variables } t \longrightarrow t = t\text{-ren } t) \wedge$   
 $(\rho\text{self} \notin \text{regions}' tm \wedge \rho\text{self} \notin \text{variables}' tm \longrightarrow tm = t\text{-rens } tm)$   
**apply** (*induct-tac t and tm,simp-all*)  
**apply** (*rule impI, elim conjE*)  
**apply** *clarsimp*  
**by** (*rule map-ρ-ren, assumption+*)

**lemma** *ρ-equals-ρ-ren:*

$\llbracket \rho\text{self} \notin (\text{the } \circ \mu 2) \text{ ' set } \rho s; \rho \in \text{set } \rho s \rrbracket$   
 $\implies \text{the } (\mu 2 \ \rho) = \rho\text{-ren } (\text{the } (\mu 2 \ \rho))$   
**apply** (*induct ρs, simp-all*)  
**apply** (*simp add: ρ-ren-def*)  
**by** *force*

**lemma** *μ-ren-extend-t-ren:*

$(\rho\text{self} \notin \text{regions } (\mu\text{-ext } (\mu 1, \mu 2) t) \wedge \rho\text{self} \notin \text{variables } (\mu\text{-ext } (\mu 1, \mu 2) t)$   
 $\longrightarrow \mu\text{-ext } (\mu 1, \mu 2) t =$   
 $\mu\text{-ext } (\lambda x. \text{Some } (t\text{-ren } (\text{the } (\mu 1 x))), \lambda \rho. \text{Some } (\rho\text{-ren } (\text{the } (\mu 2 \ \rho)))) t) \wedge$   
 $(\rho\text{self} \notin \text{regions}' (\mu\text{-exts } (\mu 1, \mu 2) ts) \wedge \rho\text{self} \notin \text{variables}' (\mu\text{-exts } (\mu 1, \mu 2)$   
 $ts) \longrightarrow \mu\text{-exts } (\mu 1, \mu 2) ts =$   
 $\mu\text{-exts } (\lambda x. \text{Some } (t\text{-ren } (\text{the } (\mu 1 x))), \lambda \rho. \text{Some } (\rho\text{-ren } (\text{the } (\mu 2 \ \rho)))) ts)$   
**apply** (*induct-tac t and ts,simp-all*)  
**apply** (*subgoal-tac*  
 $(\rho\text{self} \notin \text{regions } (\text{the } (\mu 1 \text{ list})) \wedge \rho\text{self} \notin \text{variables } (\text{the } (\mu 1 \text{ list}))$   
 $\longrightarrow (\text{the } (\mu 1 \text{ list})) = t\text{-ren } (\text{the } (\mu 1 \text{ list}))) \wedge$   
 $(\rho\text{self} \notin \text{regions}' tm \wedge \rho\text{self} \notin \text{variables}' tm \longrightarrow tm = t\text{-rens } tm), \text{simp})$   
**apply** (*rule t-equals-t-ren*)  
**apply** *clarsimp*  
**by** (*rule ρ-equals-ρ-ren, assumption+*)

**lemma** *μ-ext-args-xs:*

$\llbracket \text{distinct } xs; \text{length } xs = \text{length } ti; \text{length } as = \text{length } ti;$   
 $x \in \text{set } xs;$   
 $\text{dom } (\text{map-of } (\text{zip } xs (\text{map } (\text{atom2val } E1) as))) = \text{set } xs;$   
 $\mu\text{-ext } (\mu 1, \mu 2) (ti ! i) = t;$   
 $i < \text{length } as;$   
 $\rho\text{self} \notin \text{regions } t; \rho\text{self} \notin \text{variables } t; \rho\text{self} \notin \text{regions } (ti ! i);$   
 $xs ! i = x \rrbracket$   
 $\implies (\mu\text{-ext } (\lambda x. \text{Some } (t\text{-ren } (\text{the } (\mu 1 x))),$

$(\lambda \varrho. \text{Some } (\varrho\text{-ren } (\text{the } (\mu 2 \ \varrho))))(\varrho\text{self} \mapsto \varrho\text{self})) \circ_f$   
 $\text{map-of } (\text{zip } xs \ ti)) \ x = \text{Some } t$   
**apply** (*drule-tac*  $t=t$  **in** *sym,simp*)  
**apply** (*subst map-f-comp-map-of-zip, assumption+*)  
**apply** (*drule-tac*  $t=x$  **in** *sym,simp*)  
**apply** (*insert*  
 $\text{set-zip } [\text{where } xs=xs \text{ and}$   
 $ys=(\text{map } (\mu\text{-ext } (\lambda x. \text{Some } (t\text{-ren } (\text{the } (\mu 1 \ x)))),$   
 $(\lambda \varrho. \text{Some } (\varrho\text{-ren } (\text{the } (\mu 2 \ \varrho))))(\varrho\text{self} \mapsto \varrho\text{self})) \ ti)])$   
**apply** (*simp, rule-tac*  $x=i$  **in** *exI,simp*)  
**apply** (*subgoal-tac*  
 $(\varrho\text{self} \notin \text{regions } (ti!i)$   
 $\longrightarrow \mu\text{-ext } (\lambda x. \text{Some } (t\text{-ren } (\text{the } (\mu 1 \ x))),$   
 $(\lambda \varrho. \text{Some } (\varrho\text{-ren } (\text{the } (\mu 2 \ \varrho))))(\varrho\text{self} \mapsto \varrho\text{self})) \ (ti!i) =$   
 $\mu\text{-ext } (\lambda x. \text{Some } (t\text{-ren } (\text{the } (\mu 1 \ x))), \lambda \varrho. \text{Some } (\varrho\text{-ren } (\text{the } (\mu 2 \ \varrho)))) \ (ti!i))$   
 $\wedge$   
 $(\varrho\text{self} \notin \text{regions}' \ ts$   
 $\longrightarrow \mu\text{-exts } (\lambda x. \text{Some } (t\text{-ren } (\text{the } (\mu 1 \ x))),$   
 $(\lambda \varrho. \text{Some } (\varrho\text{-ren } (\text{the } (\mu 2 \ \varrho))))(\varrho\text{self} \mapsto \varrho\text{self})) \ ts =$   
 $\mu\text{-exts } (\lambda x. \text{Some } (t\text{-ren } (\text{the } (\mu 1 \ x))), \lambda \varrho. \text{Some } (\varrho\text{-ren } (\text{the } (\mu 2 \ \varrho)))) \ ts))$   
**prefer** 2 **apply** (*rule*  $\mu\text{-ren-extend-}\varrho\text{self}$ )  
**apply** *simp*  
**apply** (*subgoal-tac*  
 $(\varrho\text{self} \notin \text{regions } (\mu\text{-ext } (\mu 1, \mu 2) \ (ti!i)) \wedge \varrho\text{self} \notin \text{variables } (\mu\text{-ext } (\mu 1, \mu 2)$   
 $(ti!i))$   
 $\longrightarrow \mu\text{-ext } (\mu 1, \mu 2) \ (ti!i) =$   
 $\mu\text{-ext } (\lambda x. \text{Some } (t\text{-ren } (\text{the } (\mu 1 \ x))), \lambda \varrho. \text{Some } (\varrho\text{-ren } (\text{the } (\mu 2 \ \varrho)))) \ (ti!i))$   
 $\wedge$   
 $(\varrho\text{self} \notin \text{regions}' \ (\mu\text{-exts } (\mu 1, \mu 2) \ ts) \wedge \varrho\text{self} \notin \text{variables}' \ (\mu\text{-exts } (\mu 1, \mu 2)$   
 $ts)$   
 $\longrightarrow \mu\text{-exts } (\mu 1, \mu 2) \ ts =$   
 $\mu\text{-exts } (\lambda x. \text{Some } (t\text{-ren } (\text{the } (\mu 1 \ x))), \lambda \varrho. \text{Some } (\varrho\text{-ren } (\text{the } (\mu 2 \ \varrho)))) \ ts))$   
**prefer** 2 **apply** (*rule*  $\mu\text{-ren-extend-t-ren}$ )  
**by** *simp*

**lemma** *map-comp-map-of-zip- $\mu$ :*

$\llbracket \text{distinct } rs; \text{length } \varrho s = \text{length } rr; \text{length } rs = \text{length } rr; \ rs \ ! \ i = r;$   
 $rs \ ! \ i \neq \text{self};$   
 $\forall \ i < \text{length } \varrho s. \exists \ t. (\text{snd } (\mu\text{-ren } (\mu 1, \mu 2))) \ (\varrho s!i) = \text{Some } t;$   
 $\varrho\text{self} \notin \text{set } \varrho s; \ i < \text{length } rr;$   
 $(rs \ ! \ i, n) \in \text{set } (\text{zip } rs \ (\text{map } (\text{the } \circ E2) \ rr)) \rrbracket$   
 $\implies (\text{snd } (\mu\text{-ren } (\mu 1, \mu 2))) (\varrho\text{self} \mapsto \varrho\text{self})$   
 $\circ_m \text{map-of } (\text{zip } rs \ \varrho s) (\text{self} \mapsto \varrho\text{self})) \ r =$   
 $\text{Some } (\text{the } (\text{snd } (\mu\text{-ren } (\mu 1, \mu 2))) \ (\varrho s \ ! \ i)))$   
**apply** (*drule-tac*  $t=r$  **in** *sym*)  
**apply** (*subgoal-tac*  
 $(\text{snd } (\mu\text{-ren } (\mu 1, \mu 2))) (\varrho\text{self} \mapsto \varrho\text{self}) \circ_m$

$$\begin{aligned} & \text{map-of } (\text{zip } rs \ \varrho s)(self \mapsto \varrho self)) \ (rs \ ! \ i) = \\ & (\text{snd } (\mu\text{-ren } (\mu 1, \mu 2))(\varrho self \mapsto \varrho self) \circ_m \\ & \quad \text{map-of } (\text{zip } rs \ \varrho s)) \ (rs \ ! \ i), \text{simp}) \\ & \text{prefer 2 apply (simp add: map-comp-def)} \\ & \text{apply (subst map-comp-map-of-zip,simp,simp,simp,simp)} \\ & \text{apply (subst map-mu-self,simp)} \\ & \text{apply (insert set-zip [where xs=rs and ys=(map (the \circ snd (\mu\text{-ren } (\mu 1, \mu 2)))} \\ & \quad \varrho s)])} \\ & \text{apply (simp,rule-tac x=i in exI)} \\ & \text{by simp} \end{aligned}$$

**lemma** *consistent- $\eta$ -ren- $\varrho s$ :*

$$\begin{aligned} & \llbracket i < \text{length } \varrho s; \eta \ (\text{the } (\mu 2 \ (\varrho s \ ! \ i))) = \text{Some } n; \\ & \quad \varrho fake \notin \text{dom } \eta; \varrho fake \notin \text{ran } \mu 2; \mu\text{-ren-dom } (\mu 1, \mu 2); \\ & \quad \forall i < \text{length } \varrho s. \exists t. \mu 2 \ (\varrho s!i) = \text{Some } t \rrbracket \\ & \implies (\eta\text{-ren } \eta \ ++ \ [\varrho self \mapsto \text{Suc } k]) \ (\text{the } (\text{snd } (\mu\text{-ren } (\mu 1, \mu 2)) \ (\varrho s \ ! \ i))) = \text{Some} \\ & \quad n \\ & \text{apply simp} \\ & \text{apply (rule conjI)} \\ & \text{apply (simp add: \varrho-ren-def)} \\ & \text{apply (simp add: \varrho self-def add: \varrho fake-def)} \\ & \text{apply (rule impI)} \\ & \text{apply (simp add: \varrho-ren-def)} \\ & \text{apply (rule conjI,rule impI)} \\ & \text{apply (simp add: \eta-ren-def)} \\ & \text{apply (rule impI,force)} \\ & \text{apply (simp add: \eta-ren-def)} \\ & \text{by force} \end{aligned}$$

**lemma** *t-ren-mu-ext:*

$$\begin{aligned} & t\text{-ren } (\mu\text{-ext } (\mu 1, \mu 2) \ t) = \\ & \quad \mu\text{-ext } (\lambda x. \text{Some } (t\text{-ren } (\text{the } (\mu 1 \ x))), \lambda a. \text{Some } (\varrho\text{-ren } (\text{the } (\mu 2 \ a)))) \ t \wedge \\ & \quad t\text{-rens } (\mu\text{-exts } (\mu 1, \mu 2) \ ts) = \\ & \quad \mu\text{-exts } (\lambda x. \text{Some } (t\text{-ren } (\text{the } (\mu 1 \ x))), \lambda a. \text{Some } (\varrho\text{-ren } (\text{the } (\mu 2 \ a)))) \ ts \\ & \text{apply (induct-tac t and ts,simp-all)} \\ & \text{by (induct-tac list3,simp-all)} \end{aligned}$$

**lemma** *mu-exts-ren:*

$$\begin{aligned} & \llbracket \mu\text{-exts } (\mu 1, \mu 2) \ tm' = tm; \text{map } (\text{the } \circ \mu 2) \ \varrho s' = \varrho s \rrbracket \\ & \implies \mu\text{-exts } (\lambda x. \text{Some } (t\text{-ren } (\text{the } (\mu 1 \ x))), \\ & \quad \lambda a. \text{Some } (\varrho\text{-ren } (\text{the } (\mu 2 \ a)))) \ tm' = t\text{-rens } tm \\ & \quad \wedge \text{map } (\text{the } \circ (\lambda a. \text{Some } (\varrho\text{-ren } (\text{the } (\mu 2 \ a))))) \ \varrho s' = \text{map } \varrho\text{-ren } \varrho s \\ & \text{apply clarsimp} \\ & \text{apply (rule conjI)} \\ & \text{apply (subgoal-tac} \\ & \quad t\text{-ren } (\mu\text{-ext } (\mu 1, \mu 2) \ t) = \end{aligned}$$

$\mu\text{-ext } (\lambda x. \text{Some } (t\text{-ren } (\text{the } (\mu 1 \ x)))) , \lambda a. \text{Some } (\varrho\text{-ren } (\text{the } (\mu 2 \ a)))) \ t \wedge$   
 $t\text{-rens } (\mu\text{-exts } (\mu 1, \mu 2) \ tm') =$   
 $\mu\text{-exts } (\lambda x. \text{Some } (t\text{-ren } (\text{the } (\mu 1 \ x)))) , \lambda a. \text{Some } (\varrho\text{-ren } (\text{the } (\mu 2 \ a)))) \ tm'$   
**prefer 2 apply** (rule *t-ren-mu-ext*)  
**apply** *clarsimp*  
**by** (*induct-tac*  $\varrho s', \text{simp-all}$ )

**lemma**  *$\eta\text{-ren-}\varrho\text{-ren-}\varrho l\text{-SomeI}$* :  
 $\llbracket \varrho l = \text{last } \varrho s; \text{last } \varrho s \in \text{dom } \eta; \eta (\text{last } \varrho s) = \text{Some } j;$   
 $\eta' = \eta\text{-ren } \eta(\varrho\text{self} \mapsto \text{Suc } k);$   
 $\varrho s \neq [];$   
 $\varrho\text{fake} \notin \text{dom } \eta \rrbracket$   
 $\implies (\eta\text{-ren } \eta(\varrho\text{self} \mapsto \text{Suc } k)) (\varrho\text{-ren } (\text{last } \varrho s)) = \text{Some } j$   
**apply** (*simp add:  $\varrho\text{-ren-def}$* )  
**apply** (rule *conjI*)  
**apply** (rule *impI*)  
**apply** (*simp add:  $\eta\text{-ren-def}$* )  
**apply** *clarsimp*  
**apply** (rule *impI*)  
**apply** (*simp add:  $\eta\text{-ren-def}$* )  
**by** *clarsimp*

**lemma**  *$\text{last-map-not-}\varrho\text{self}$* :  
 $\llbracket \varrho\text{self} \notin (\text{the } \circ \mu 2) \text{ ' set } \varrho s';$   
 $\varrho s' \neq [] \rrbracket$   
 $\implies \text{last } (\text{map } (\text{the } \circ \mu 2) \ \varrho s') \neq \varrho\text{self}$   
**by** (*induct*  $\varrho s', \text{simp-all, force}$ )

**lemma**  *$\text{length-}\varrho s$* :  
 $\llbracket \text{length } \varrho s' > 0;$   
 $(\text{the } (\mu 2 \ \varrho'), \mu\text{-ext } (\mu 1, \mu 2) (\text{TypeExpression.ConstrT } T \ tm' \ \varrho s')) =$   
 $(\varrho l, \text{TypeExpression.ConstrT } T \ tm \ \varrho s) \rrbracket$   
 $\implies \varrho s \neq []$   
**apply** (*simp, elim conjE*)  
**by** (*induct*  $\varrho s', \text{simp, clarsimp}$ )

**lemma**  *$\text{consistent-t-consistent-t-ren [rule-format]}$* :  
 $\text{consistent-v } t \ \eta \ v \ h$   
 $\longrightarrow \varrho\text{fake} \notin \text{dom } \eta$   
 $\longrightarrow \eta' = (\eta\text{-ren } \eta(\varrho\text{self} \mapsto \text{Suc } k))$   
 $\longrightarrow \text{consistent-v } (t\text{-ren } t) \ \eta' \ v \ h$   
**apply** (rule *impI*)  
**apply** (*erule consistent-v.induct*)  
  
**apply** (rule *impI*)+

**apply** *simp*  
**apply** (*rule consistent-v.primitiveI*)

**apply** (*rule impI*)+  
**apply** *simp*  
**apply** (*rule consistent-v.primitiveB*)

**apply** (*rule impI*)+  
**apply** *simp*  
**apply** (*rule consistent-v.variable*)

**apply** *clarsimp*  
**apply** (*rule consistent-v.algebraic-None*)  
**apply** (*simp add: dom-def*)

**apply** (*rule impI*)+  
**apply** (*elim exE, elim conjE*)  
**apply** (*simp only: t-ren-t-rent.simps(2)*)  
**apply** (*rule consistent-v.algebraic*)

**apply** *force*

**apply** *force*

**apply** (*simp only: wellT.simps*)  
**apply** (*elim conjE*)  
**apply** (*frule length-qs,assumption+*)  
**apply** (*subst map-last,assumption+*)  
**apply** (*frule  $\eta$ -ren- $\varrho$ -ren- $\varrho$ l-SomeI, assumption+*)  
**apply** (*simp add: dom-def*)

**apply** (*simp only: wellT.simps*)  
**apply** (*elim conjE*)  
**apply** (*frule length-qs,assumption+*)  
**apply** (*subst map-last,assumption+*)  
**apply** (*frule  $\eta$ -ren- $\varrho$ -ren- $\varrho$ l-SomeI*)  
**apply** (*assumption,assumption,assumption,assumption,assumption,assumption*)

**apply** *force*

**apply** *force*

**apply** *force*

**apply** (*rule-tac x=( $\lambda x$ . Some (t-ren (the ( $\mu 1$  x))))*) **in** *exI*)  
**apply** (*rule-tac x=( $\lambda a$ . Some ( $\varrho$ -ren (the ( $\mu 2$  a))))*) **in** *exI*)  
**apply** (*rule conjI*)  
**apply** (*simp only: wellT.simps*)  
**apply** (*elim conjE*)

```

apply (frule length-qs,assumption+)
apply simp
apply (rule conjI)
apply (subst map-last,assumption+,simp)
apply (elim conjE)
apply (rule mu-exts-ren,assumption+)

apply (rule allI, rule impI)
apply (erule-tac x=i in allE)
apply (drule mp,simp)
apply (elim conjE)

apply (drule mp,simp)
apply (drule mp,simp)
apply simp
apply (subgoal-tac
  (t-ren (mu-ext ( $\mu 1$ ,  $\mu 2$ ) (tn' ! i))) =
  (mu-ext ( $\lambda x$ . Some (t-ren (the ( $\mu 1$  x))),
     $\lambda a$ . Some ( $\varrho$ -ren (the ( $\mu 2$  a)))) (tn' ! i))  $\wedge$ 
  t-rems (mu-exts ( $\mu 1$ ,  $\mu 2$ ) ts) =
  mu-exts ( $\lambda x$ . Some (t-ren (the ( $\mu 1$  x))),  $\lambda a$ . Some ( $\varrho$ -ren (the ( $\mu 2$  a)))) ts)
  prefer 2 apply (rule t-ren-mu-ext)
apply simp
done

lemma mu-ext-args-xs-ti:
  [| distinct xs; length xs = length ti; length as = length ti;
    x  $\in$  set xs;
    dom (map-of (zip xs (map (atom2val E1) as))) = set xs;
    mu-ext ( $\mu 1$ , $\mu 2$ ) (ti ! i) = t;
    i < length as;  $\varrho$ self  $\notin$  regions (ti ! i);
    xs ! i = x |]
   $\implies$  (mu-ext ( $\lambda x$ . Some (t-ren (the ( $\mu 1$  x))),
    ( $\lambda \varrho$ . Some ( $\varrho$ -ren (the ( $\mu 2$   $\varrho$ ))))( $\varrho$ self  $\mapsto$   $\varrho$ self))  $\circ_f$ 
    map-of (zip xs ti)) x = Some (mu-ext ( $\lambda x$ . Some (t-ren (the ( $\mu 1$  x))),
       $\lambda \varrho$ . Some ( $\varrho$ -ren (the ( $\mu 2$   $\varrho$ )))) (ti ! i))

apply (drule-tac t=t in sym,simp)
apply (subst map-f-comp-map-of-zip, assumption+)
apply (drule-tac t=x in sym,simp)
apply (insert
  set-zip [where xs=xs and
    ys=(map (mu-ext ( $\lambda x$ . Some (t-ren (the ( $\mu 1$  x))),
      ( $\lambda \varrho$ . Some ( $\varrho$ -ren (the ( $\mu 2$   $\varrho$ ))))( $\varrho$ self  $\mapsto$   $\varrho$ self))) ti)]

apply (simp, rule-tac x=i in exI,simp)
apply (subgoal-tac
  ( $\varrho$ self  $\notin$  regions (ti!i)
   $\longrightarrow$  mu-ext ( $\lambda x$ . Some (t-ren (the ( $\mu 1$  x))),
    ( $\lambda \varrho$ . Some ( $\varrho$ -ren (the ( $\mu 2$   $\varrho$ ))))( $\varrho$ self  $\mapsto$   $\varrho$ self)) (ti!i) =
    mu-ext ( $\lambda x$ . Some (t-ren (the ( $\mu 1$  x))),

```

$\lambda \varrho. \text{Some } (\varrho\text{-ren } (\text{the } (\mu 2 \ \varrho))) (ti!i)) \wedge (\varrho\text{self} \notin \text{regions}' \ ts)$   
 $\longrightarrow \text{mu-exts } (\lambda x. \text{Some } (t\text{-ren } (\text{the } (\mu 1 \ x))),$   
 $\quad (\lambda \varrho. \text{Some } (\varrho\text{-ren } (\text{the } (\mu 2 \ \varrho)))) (\varrho\text{self} \mapsto \varrho\text{self})) \ ts =$   
 $\text{mu-exts } (\lambda x. \text{Some } (t\text{-ren } (\text{the } (\mu 1 \ x))),$   
 $\quad \lambda \varrho. \text{Some } (\varrho\text{-ren } (\text{the } (\mu 2 \ \varrho)))) \ ts))$   
**apply** *simp*  
**by** (rule  $\mu\text{-ren-extend-}\varrho\text{self}$ )

**lemma** *SafeDARegion-APP-P4*:

$\llbracket \text{distinct } xs; \text{length } xs = \text{length } ti; \text{length } xs = \text{length } as;$   
 $\text{distinct } rs; \text{length } rs = \text{length } rr; \text{length } rs = \text{length } \varrho s;$   
 $\varrho\text{self} \notin \text{regions } tg \cup (\bigcup \text{set } (\text{map } \text{regions } ti)) \cup \text{set } \varrho s;$   
 $\forall \ i < \text{length } \varrho s. \exists \ t. \mu 2 \ (\varrho s!i) = \text{Some } t; \varrho\text{fake} \notin \text{ran } \mu 2;$   
 $\mu\text{-ren-dom } (\mu 1, \mu 2);$   
 $\text{set } rr \subseteq \text{dom } E2; \text{fvs}' \ as \subseteq \text{dom } E1;$   
 $\text{consistent } (\vartheta 1, \vartheta 2) \ \eta \ (E1, E2) \ h;$   
 $\text{argP-app } (\text{map } (\text{mu-ext } (\mu 1, \mu 2)) \ ti) \ (\text{map } (\text{the} \circ \mu 2) \ \varrho s) \ as \ rr \ (\vartheta 1, \vartheta 2) \rrbracket$   
 $\implies \text{consistent}$   
 $(\text{mu-ext } (\text{fst } (\mu\text{-ren } (\mu 1, \mu 2)), \text{snd } (\mu\text{-ren } (\mu 1, \mu 2)) (\varrho\text{self} \mapsto \varrho\text{self})))$   
 $\quad \circ_f \text{map-of } (\text{zip } xs \ ti),$   
 $\text{snd } (\mu\text{-ren } (\mu 1, \mu 2)) (\varrho\text{self} \mapsto \varrho\text{self})$   
 $\quad \circ_m \text{map-of } (\text{zip } rs \ \varrho s) (\text{self} \mapsto \varrho\text{self}))$   
 $(\eta\text{-ren } \eta \ ++ \ [\varrho\text{self} \mapsto \text{Suc } k]) \ (\text{map-of } (\text{zip } xs \ (\text{map } (\text{atom2val } E1) \ as)),$   
 $\text{map-of } (\text{zip } rs \ (\text{map } (\text{the} \circ E2) \ rr)) (\text{self} \mapsto \text{Suc } k)) \ h$   
**apply** (subgoal-tac  $\varrho\text{fake} \notin \text{dom } \eta$ )  
**prefer** 2 **apply** (rule  $\varrho\text{fake-not-in-dom-}\eta$ )  
**apply** (unfold *consistent.simps*)  
**apply** (elim *conjE*)  
**apply** (rule *conjI*)

**apply** *simp*  
**apply** (rule *ballI*)  
**apply** (subgoal-tac  
 $\exists \ i < \text{length } as. \ xs!i = x \wedge$   
 $(\text{map-of } (\text{zip } xs \ (\text{map } (\text{atom2val } E1) \ as))) \ x = \text{Some } (\text{atom2val } E1 \ (as!i)))$   
**prefer** 2 **apply** (frule-tac  $vs = \text{map } (\text{atom2val } E1) \ as$   
**in**  $\text{map-of-zip-is-SomeI, simp, simp, force}$ )  
**apply** (elim *exE*, elim *conjE*)  
**apply** (simp add: *argP-app.simps*)  
**apply** (elim *conjE*)  
**apply** (erule-tac  $x=i$  **in** *allE, simp*) +  
**apply** (erule *disjE*)  
  
**apply** (elim *exE, simp*)  
**apply** (rule-tac  $x = (\text{ConstrT } \text{intType } [] [])$  **in** *exI*)  
**apply** (rule *conjI*)

```

    apply (rule-tac i=i in mu-ext-args-xs)
    apply (assumption+,simp,assumption+,simp,force,simp,simp,assumption+)
    apply (rule-tac x=IntT c in exI,simp)
    apply (rule consistent-v.primitiveI)
    apply (erule disjE)

    apply (elim exE,simp)
    apply (rule-tac x=(ConstrT boolType [] []) in exI)
    apply (rule conjI)
    apply (rule-tac i=i in mu-ext-args-xs)
    apply (assumption+,simp,assumption+,simp,force,simp,simp,assumption+)
    apply (rule-tac x=BoolT b in exI,simp)
    apply (rule consistent-v.primitiveB)

    apply (elim exE)
    apply (simp add: argP-aux.simps)
    apply (rule-tac x=mu-ext ( $\lambda x. \text{Some } (t\text{-ren } (the (\mu 1 x)))$ ),
      ( $\lambda \varrho. \text{Some } (\varrho\text{-ren } (the (\mu 2 \varrho))))$ ) (ti ! i) in exI)
    apply (rule conjI)
    apply (rule-tac i=i in mu-ext-args-xs-ti,assumption+,simp,simp,simp,force,simp)
    apply (rule-tac x=the (E1 xa) in exI,simp)
    apply (erule-tac x=xa in ballE)
    apply (elim exE, elim conjE)+
    apply simp
    apply (subgoal-tac
      t-ren (mu-ext ( $\mu 1, \mu 2$ ) (ti ! i)) =
      mu-ext ( $\lambda x. \text{Some } (t\text{-ren } (the (\mu 1 x)))$ ),  $\lambda a. \text{Some } (\varrho\text{-ren } (the (\mu 2 a))))$ ) (ti !
i)  $\wedge$ 
      t-rems (mu-exts ( $\mu 1, \mu 2$ ) ts) =
      mu-exts ( $\lambda x. \text{Some } (t\text{-ren } (the (\mu 1 x)))$ ),  $\lambda a. \text{Some } (\varrho\text{-ren } (the (\mu 2 a))))$ ) ts)
    prefer 2 apply (rule t-ren-mu-ext)
    apply (elim conjE)
    apply (drule-tac s=t-ren (mu-ext ( $\mu 1, \mu 2$ ) (ti ! i)) in sym)
    apply simp
    apply (rule consistent-t-consistent-t-ren, assumption+,simp)

    apply (frule as-in-E1,simp,simp,simp)

    apply (rule conjI)
    apply (rule ballI)
    apply (case-tac r = self)

    apply clarsimp

    apply (subgoal-tac r  $\in$  set rs)

```



```

prefer 2 apply simp
apply (subgoal-tac
   $\exists i < \text{length } rr. rs!i = r \wedge (\text{map-of } (\text{zip } rs (\text{map } (\text{the} \circ E2) rr))) r =$ 
     $\text{Some } ((\text{the} \circ E2) (rr!i))$ )
prefer 2 apply (frule-tac vs=map (the ∘ E2) rr
  in map-of-zip-is-SomeI,simp,simp,force)
apply (simp only: argP-app.simps)
apply (elim exE,elim conjE)
apply (subgoal-tac  $\forall i < \text{length } rr. \exists n. E2 (rr!i) = \text{Some } n$ )
  prefer 2 apply (rule rr-in-E2,assumption)
apply (rotate-tac 28)
apply (erule-tac x=i in allE) apply (drule mp) apply simp
apply (erule-tac x=rr!i and A=dom E2 in ballE)
  prefer 2 apply (simp add: dom-def)
apply (elim exE, elim conjE)+
apply (rule-tac x=the ((snd (μ-ren (μ1,μ2))) (ρs!i)) in exI)
apply (rule-tac x=n in exI)
apply (rule conjI)
apply (drule-tac s=length ρs in sym)
apply (rule-tac ρs=ρs and rr=rr in map-comp-map-of-zip-μ)
apply (assumption+,simp,simp,assumption+,simp)
apply (simp,simp,simp,simp)
apply (rule conjI)
apply (rule consistent-η-ren-ρs)
apply (simp,simp,simp,simp,simp)
apply simp
apply simp
apply (rule conjI)
apply simp
by simp

```

**lemma** *regions-induct-t-var:*

```

  ( $x \in \text{regions } (\text{the } (\text{fst } \mu \varrho)) \wedge \varrho \in \text{variables } t \longrightarrow x \in \text{regions } (\text{mu-ext } \mu t)) \wedge$ 
  ( $x \in \text{regions } (\text{the } (\text{fst } \mu \varrho)) \wedge \varrho \in \text{variables}' tm \longrightarrow x \in \text{regions}' (\text{mu-exts } \mu$ 
     $tm))$ )
apply (induct-tac t and tm)
by clarsimp+

```

**lemma** *regions-induct-constr-ρ:*

```

  ( $\varrho \in \text{regions } t \longrightarrow \text{the } (\text{snd } \mu \varrho) \in \text{regions } (\text{mu-ext } \mu t)) \wedge$ 
  ( $\varrho \in \text{regions}' tm \longrightarrow \text{the } (\text{snd } \mu \varrho) \in \text{regions}' (\text{mu-exts } \mu tm)$ )
apply (induct-tac t and tm)
by clarsimp+

```

**lemma** *regions-induct-constr-qs* [rule-format]:  
 $set\ \varrho s \subseteq regions\ t' \longrightarrow$   
 $(the \circ snd\ \mu) \text{ ' } set\ \varrho s \subseteq regions\ (mu-ext\ \mu\ t')$   
**apply** (*induct qs,simp-all*)  
**apply** *clarsimp*  
**apply** (*subgoal-tac*  
 $(a \in regions\ t' \longrightarrow the\ (snd\ \mu\ a) \in regions\ (mu-ext\ \mu\ t')) \wedge$   
 $(a \in regions'\ tm \longrightarrow the\ (snd\ \mu\ a) \in regions'\ (mu-exts\ \mu\ tm)))$ )  
**apply** *simp*  
**by** (*rule regions-induct-constr-qs*)

**lemma** *regions-regions-mu-ext*:  
 $(regions\ t \subseteq regions\ t' \wedge variables\ t \subseteq variables\ t'$   
 $\longrightarrow regions\ (mu-ext\ \mu\ t) \subseteq regions\ (mu-ext\ \mu\ t')) \wedge$   
 $(regions'\ tm \subseteq regions\ t' \wedge variables'\ tm \subseteq variables\ t'$   
 $\longrightarrow regions'\ (mu-exts\ \mu\ tm) \subseteq regions\ (mu-ext\ \mu\ t'))$   
**apply** (*induct-tac t and tm*)

**apply** *clarsimp*  
**apply** (*rename-tac q x*)  
**apply** (*subgoal-tac*  
 $(x \in regions\ (the\ (fst\ \mu\ \varrho)) \wedge \varrho \in variables\ t' \longrightarrow x \in regions\ (mu-ext\ \mu\ t')) \wedge$   
 $(x \in regions\ (the\ (fst\ \mu\ \varrho)) \wedge \varrho \in variables'\ tm \longrightarrow x \in regions'\ (mu-exts\ \mu\ tm)))$ )  
**apply** *simp*  
**apply** (*rule regions-induct-t-var*)

**apply** (*rule impI*)  
**apply** (*elim conjE*)  
**apply** *simp*  
**apply** (*rule regions-induct-constr-qs,simp*)

**apply** *clarsimp*

**by** *clarsimp*

**lemma** *wellT-regions*:  
 $\llbracket wellT\ tn'\ (last\ \varrho s')\ (TypeExpression.ConstrT\ T\ tm'\ \varrho s');$   
 $mu-ext\ (\mu 1, \mu 2)\ (TypeExpression.ConstrT\ T\ tm'\ \varrho s') = TypeExpression.ConstrT$   
 $T\ tm\ \varrho s;$   
 $\varrho self \notin regions\ (TypeExpression.ConstrT\ T\ tm\ \varrho s) \rrbracket$

$\implies \forall i < \text{length } tn'. \varrho_{\text{self}} \notin \text{regions } (\text{map } (\text{mu-ext } (\mu 1, \mu 2)) \text{ } tn' ! i)$   
**apply** (*unfold wellT.simps*)  
**apply** (*elim conjE*)  
**apply** (*rule allI, rule impI*)  
**apply** (*erule-tac x=i in allE*)  
**apply** (*drule mp, simp*)  
**apply** (*drule sym*)  
**apply** (*subgoal-tac*)  
 $(\text{regions } (tn' ! i) \subseteq \text{regions } (\text{TypeExpression.ConstrT } T \text{ } tm' \varrho s') \wedge$   
 $\text{variables } (tn' ! i) \subseteq \text{variables } (\text{TypeExpression.ConstrT } T \text{ } tm' \varrho s'))$   
 $\longrightarrow \text{regions } (\text{mu-ext } (\mu 1, \mu 2) \text{ } (tn' ! i)) \subseteq$   
 $\text{regions } (\text{mu-ext } (\mu 1, \mu 2) \text{ } (\text{TypeExpression.ConstrT } T \text{ } tm' \varrho s')) \wedge$   
 $(\text{regions}' \text{ } tm'' \subseteq \text{regions } (\text{TypeExpression.ConstrT } T \text{ } tm' \varrho s') \wedge$   
 $\text{variables}' \text{ } tm'' \subseteq \text{variables } (\text{TypeExpression.ConstrT } T \text{ } tm' \varrho s'))$   
 $\longrightarrow \text{regions}' (\text{mu-exts } (\mu 1, \mu 2) \text{ } tm'') \subseteq$   
 $\text{regions } (\text{mu-ext } (\mu 1, \mu 2) \text{ } (\text{TypeExpression.ConstrT } T \text{ } tm' \varrho s')))$   
**apply** (*simp del: regions-regions'.simps(2) del: mu-ext-mu-exts.simps(2)*)  
**apply** (*elim conjE*)  
**apply** *blast*  
**by** (*rule regions-regions-mu-ext*)

**lemma** *last-notin-set*:  
 $\llbracket \varrho s \neq []; \varrho \notin \text{set } \varrho s \rrbracket$   
 $\implies \text{last } \varrho s \neq \varrho$   
**by** (*induct  $\varrho s$ , simp, clarsimp*)

**lemma**  *$\eta$ - $\varrho$ -ren-inv- $\varrho$ -Some- $j$* :  
 $\llbracket (\eta\text{-ren } \eta(\varrho_{\text{self}} \mapsto \text{Suc } k)) \text{ } \varrho = \text{Some } j;$   
 $\varrho_{\text{fake}} \notin \text{dom } \eta;$   
 $\varrho \neq \varrho_{\text{self}} \rrbracket$   
 $\implies \eta (\varrho\text{-ren-inv } \varrho) = \text{Some } j$   
**apply** (*simp add:  $\eta$ -ren-def*)  
**apply** (*split split-if-asm*)  
**apply** (*simp add:  $\varrho$ -ren-inv-def*)  
**apply** *force*  
**apply** (*simp add:  $\varrho$ -ren-inv-def*)  
**by** *force*

**lemma**  *$t$ -ren-in-mu-ext-inv*:  
 $t\text{-ren-inv } (\text{mu-ext } (\mu 1, \mu 2) \text{ } t) =$   
 $\text{mu-ext } (\lambda x. \text{Some } (t\text{-ren-inv } (\text{the } (\mu 1 \text{ } x))), \lambda a. \text{Some } (\varrho\text{-ren-inv } (\text{the } (\mu 2 \text{ } a))))$   
 $t \wedge$   
 $t\text{-ren-invs } (\text{mu-exts } (\mu 1, \mu 2) \text{ } ts) =$

$mu-exts (\lambda x. Some (t-ren-inv (the (\mu 1 x))), \lambda a. Some (\varrho-ren-inv (the (\mu 2 a))))$   
 $ts$   
**apply** (*induct-tac*  $t$  **and**  $ts, simp-all$ )  
**by** (*induct-tac*  $list3, simp-all$ )

**lemma** *mu-exts-ren-inv*:

$\llbracket mu-exts (\mu 1, \mu 2) tm' = tm; map (the \circ \mu 2) \varrho s' = \varrho s \rrbracket$   
 $\implies mu-exts (\lambda x. Some (t-ren-inv (the (\mu 1 x))),$   
 $\quad \lambda a. Some (\varrho-ren-inv (the (\mu 2 a)))) tm' = t-ren-invs tm$   
 $\wedge map (the \circ (\lambda a. Some (\varrho-ren-inv (the (\mu 2 a)))) \varrho s' = map \varrho-ren-inv \varrho s$   
**apply** *clarsimp*  
**apply** (*rule conjI*)  
**apply** (*subgoal-tac*)  
 $t-ren-inv (mu-ext (\mu 1, \mu 2) t) =$   
 $mu-ext (\lambda x. Some (t-ren-inv (the (\mu 1 x))), \lambda a. Some (\varrho-ren-inv (the (\mu 2 a))))$   
 $t \wedge$   
 $t-ren-invs (mu-exts (\mu 1, \mu 2) tm') =$   
 $mu-exts (\lambda x. Some (t-ren-inv (the (\mu 1 x))), \lambda a. Some (\varrho-ren-inv (the (\mu 2 a))))$   
 $tm')$   
**prefer** 2 **apply** (*rule t-ren-in-mu-ext-inv*)  
**apply** *clarsimp*  
**by** (*induct-tac*  $\varrho s', simp-all$ )

**lemma** *consistent-t-consistent-t-ren-inv* [*rule-format*]:

$consistent-v t' \eta' v h-e$   
 $\longrightarrow \varrho self \notin regions t'$   
 $\longrightarrow \eta' = (\eta-ren \ \eta(\varrho self \mapsto Suc k))$   
 $\longrightarrow consistent-v (t-ren-inv t') \eta v h-e$   
**apply** (*rule impI*)  
**apply** (*erule consistent-v.induct*)

**apply** (*rule impI*)  
**apply** *simp*  
**apply** (*rule consistent-v.primitiveI*)

**apply** (*rule impI*)  
**apply** *simp*  
**apply** (*rule consistent-v.primitiveB*)

**apply** (*rule impI*)  
**apply** *simp*  
**apply** (*rule consistent-v.variable*)

**apply** *clarsimp*  
**apply** (*rule consistent-v.algebraic-None*)  
**apply** (*simp add: dom-def*)

**apply** (*rule impI*)  
**apply** *simp*

```

apply (elim exE, elim conjE)
apply (simp only: t-ren-inv-t-ren-invs.simps(2))
apply (rule consistent-v.algebraic)

  apply force

  apply force

apply (simp only: wellT.simps)
apply (elim conjE)
apply (frule length-qs,assumption+)
apply (subst map-last,assumption+)
apply (subgoal-tac last qs ≠ qself)
apply (subgoal-tac qfake ∉ dom η)
  prefer 2 apply (rule qfake-not-in-dom-η)
apply (frule η-q-ren-inv-q-Some-j,assumption+)
apply (simp add: dom-def)
apply (simp, elim conjE)
apply (rule last-notin-set, assumption+)

apply (simp only: wellT.simps)
apply (elim conjE)
apply (frule length-qs,assumption+)
apply (subst map-last,assumption+)
apply (subgoal-tac last qs ≠ qself)
apply (subgoal-tac qfake ∉ dom η)
  prefer 2 apply (rule qfake-not-in-dom-η)
apply (frule η-q-ren-inv-q-Some-j,assumption+)
apply (simp, elim conjE)
apply (rule last-notin-set, assumption, assumption)

  apply force

  apply force

  apply force

apply (rule-tac x=(λx. Some (t-ren-inv (the (μ1 x)))) in exI)
apply (rule-tac x=(λa. Some (q-ren-inv (the (μ2 a)))) in exI)
apply (rule conjI)
apply (simp only: wellT.simps)
apply (elim conjE)
apply (frule length-qs,assumption+)
apply simp
apply (rule conjI)
apply (subst map-last,assumption+,simp)
apply (elim conjE)
apply (rule mu-exts-ren-inv,assumption+)

```

```

apply (rule allI, rule impI)
apply (erule-tac x=i in allE)
apply (drule mp,simp)
apply (elim conjE)

apply (frule wellT-regions,force,simp)
apply (drule mp,simp)
apply (drule mp,simp)
apply simp
apply (subgoal-tac
  (t-ren-inv (mu-ext (μ1, μ2) (tn' ! i))) =
  (mu-ext (λx. Some (t-ren-inv (the (μ1 x))),
    λa. Some (ρ-ren-inv (the (μ2 a)))) (tn' ! i)) ∧
  t-ren-invs (mu-exts (μ1, μ2) ts) =
  mu-exts (λx. Some (t-ren-inv (the (μ1 x))),
    λa. Some (ρ-ren-inv (the (μ2 a)))) ts)
prefer 2 apply (rule t-ren-in-mu-ext-inv)
apply simp
done

lemma mu-ext-ρself-mu-ext [rule-format]:
  (ρself ∉ regions t
    ⟶ (mu-ext (μ1,μ2(ρself ↦ ρself)) t) = (mu-ext (μ1,μ2) t)) ∧
  (ρself ∉ regions' tm
    ⟶ (mu-exts (μ1,μ2(ρself ↦ ρself)) tm) = (mu-exts (μ1,μ2) tm))
by (induct-tac t and tm,simp-all)

lemma ρself-notin-regions-t-ren [rule-format]:
  ρself ∉ regions (t-ren t) ∧
  ρself ∉ regions' (t-rems tm)
apply (induct-tac t and tm, simp-all)
apply (induct-tac list3,simp-all)
apply (simp add: ρ-ren-def)
by (simp add: ρself-def add: ρfake-def)

lemma ρself-notin-regions-mu-ren:
  (ρself ∉ regions (mu-ext (λx. Some (t-ren (the (μ1 x))),
    λa. Some (ρ-ren (the (μ2 a)))) t)) ∧
  (ρself ∉ regions' (mu-exts (λx. Some (t-ren (the (μ1 x))),
    (λρ. Some (ρ-ren (the (μ2 ρ))))) tm))
apply (induct-tac t and tm,simp-all)
apply (subst ρself-notin-regions-t-ren,simp)
apply (induct-tac list3,simp-all)
apply (simp add: ρ-ren-def)
by (simp add: ρself-def add: ρfake-def)

lemma t-ren-inv-t-ren:
  (notFake t ⟶ t-ren-inv (t-ren t) = t) ∧

```

$(\text{notFakes } tm \longrightarrow t\text{-ren-inv} (t\text{-rens } tm) = tm)$   
**apply** (*induct-tac* *t* **and** *tm,simp-all*)  
**apply** (*induct-tac* *list3,simp-all*)  
**apply** *clarsimp*  
**by** (*simp add:  $\varrho$ -ren-def add:  $\varrho$ -ren-inv-def*)

**lemma** *mu-ext-def-ConstrT*:  
 $\text{mu-ext-def } (\mu1, \mu2) (\text{ConstrT } T \text{ } tm \ \varrho s)$   
 $\implies \text{mu-exts-def } (\mu1, \mu2) \text{ } tm$   
**apply** (*simp add: mu-ext-def-def*)  
**apply** (*induct tm,simp-all*)  
**apply** (*case-tac a,simp-all*)  
**apply** (*simp add: mu-ext-def-def*)  
**by** (*simp add: mu-ext-def-def*)

**lemma** *mu-ext-def- $\varrho s$  [rule-format]*:  
 $\text{mu-ext-def } (\mu1, \mu2) (\text{ConstrT } T \text{ } tm \ \varrho s) \longrightarrow$   
 $\text{map } \varrho\text{-ren-inv } (\text{map } (\text{the} \circ (\lambda \varrho. \text{Some } (\varrho\text{-ren } (\text{the } (\mu2 \ \varrho))))) \ \varrho s)$   
 $= \text{map } (\text{the} \circ \mu2) \ \varrho s$   
**apply** (*induct-tac  $\varrho s$ , simp-all*)  
**apply** (*rule impI*)  
**apply** (*rule conjI*)  
**apply** (*simp add: mu-ext-def-def*)  
**apply** (*simp add:  $\varrho$ -ren-def add:  $\varrho$ -ren-inv-def*)  
**by** (*simp add: mu-ext-def-def*)

**lemma** *t-ren-inv-t-ren-t*:  
 $(\text{mu-ext-def } (\mu1, \mu2) \text{ } t$   
 $\longrightarrow (\text{t-ren-inv } (\text{mu-ext } (\lambda x. \text{Some } (t\text{-ren } (\text{the } (\mu1 \ x))),$   
 $\quad (\lambda \varrho. \text{Some } (\varrho\text{-ren } (\text{the } (\mu2 \ \varrho))))) \text{ } t)) =$   
 $(\text{mu-ext } (\mu1, \mu2) \text{ } t)) \wedge$   
 $(\text{mu-exts-def } (\mu1, \mu2) \text{ } tm$   
 $\longrightarrow (\text{t-ren-inv} (\text{mu-exts } (\lambda x. \text{Some } (t\text{-ren } (\text{the } (\mu1 \ x))),$   
 $\quad (\lambda \varrho. \text{Some } (\varrho\text{-ren } (\text{the } (\mu2 \ \varrho))))) \text{ } tm)) =$   
 $(\text{mu-exts } (\mu1, \mu2) \text{ } tm))$   
**apply** (*induct-tac t and tm,simp-all*)  
**apply** (*rule impI*)  
**apply** (*subst t-ren-inv-t-ren*)  
**apply** (*simp add: mu-ext-def-def*)  
**apply** *simp*  
**apply** *clarsimp*  
**apply** (*frule mu-ext-def-ConstrT,simp*)  
**by** (*rule mu-ext-def- $\varrho s$ ,simp*)

**lemma** *consistent-t-ren-inv-consistent-t*:  
 $\llbracket \varrho \text{self} \notin \text{regions } tg; \text{mu-ext-def } (\mu1, \mu2) \text{ } tg;$

$$\text{consistent-}v \text{ (} t\text{-ren-inv (mu-ext (\lambda x. Some (t-ren (the (\mu 1 x))),$$
  

$$(\lambda \varrho. Some (\varrho\text{-ren (the (\mu 2 \varrho))))(\varrho\text{self} \mapsto \varrho\text{self})) tg)) } \eta \text{ } v \text{ } h\text{-}e \rrbracket$$
  

$$\implies \text{consistent-}v \text{ (mu-ext (\mu 1, \mu 2) tg) } \eta \text{ } v \text{ } h\text{-}e$$
  
**apply** (insert mu-ext- $\varrho$ self-mu-ext)  
**by** (insert t-ren-inv-t-ren-t,simp-all)

**lemma** *restricted-h-equals-h*:  

$$\llbracket h \text{ } p = \text{Some } (j, C, vn);$$
  

$$j < \text{Suc } k \rrbracket$$
  

$$\implies (h \mid \{p \in \text{dom } h. \text{fst (the (h } p)) \leq k\}) \text{ } p = h \text{ } p$$
  
**apply** (subgoal-tac  $p \in \text{dom } h, \text{simp}$ )  
**by** (simp add: dom-def)

**lemma** *j-le-Suc-k*:  

$$\llbracket \eta \text{ } \varrho l = \text{Some } j; \text{admissible } \eta \text{ } k \rrbracket$$
  

$$\implies j < \text{Suc } k$$
  
**apply** (simp add: admissible-def)  
**apply** (elim conjE)  
**apply** (erule-tac  $x = \varrho l$  in ballE)  
**apply** force  
**by** (simp add: dom-def)

**lemma** *SafeDARegion-APP-P5* [rule-format]:  

$$\text{consistent-}v \text{ } t \text{ } \eta \text{ } v \text{ } h\text{-}e$$
  

$$\longrightarrow \text{admissible } \eta \text{ } k$$
  

$$\longrightarrow \text{consistent-}v \text{ } t \text{ } \eta \text{ } v \text{ } (h\text{-}e \mid \{p \in \text{dom } h\text{-}e. \text{fst (the (h-e } p)) \leq k\})$$
  
**apply** (rule impI)  
**apply** (erule consistent-v.induct)

**apply** (rule impI)+  
**apply** (rule consistent-v.primitiveI)

**apply** (rule impI)+  
**apply** (rule consistent-v.primitiveB)

**apply** (rule impI)+  
**apply** (rule consistent-v.variable)

**apply** clarsimp  
**apply** (rule consistent-v.algebraic-None)  
**apply** (simp add: dom-def)

**apply** (rule impI)+  
**apply** (elim exE, elim conjE)



```

apply (rule consistent-v.algebraic)

apply (frule j-le-Suc-k,assumption)
apply (frule-tac h=h in restricted-h-equals-h,assumption)
apply force

apply force

apply force

apply force

apply force

apply force

apply (rule-tac x= $\mu$ 1 in exI)
apply (rule-tac x= $\mu$ 2 in exI)
apply simp
done

declare dom-fun-upd [simp del]
declare fun-upd-apply [simp del]
declare restrict-map-def [simp del]
declare map-upds-def [simp del]

lemma lemma-19-aux [rule-format]:
   $\models \Sigma t$ 
   $\longrightarrow \Sigma t \ g = \text{Some } (ti, qs, tg)$ 
   $\longrightarrow \Sigma f \ g = \text{Some } (xs, rs, eg)$ 
   $\longrightarrow (bodyAPP \ \Sigma f \ g): \{ (map-of \ (zip \ (varsAPP \ \Sigma f \ g) \ (typesArgAPP \ \Sigma t \ g)),$ 
    map-of (zip (regionsAPP  $\Sigma f \ g$ ) (regionsArgAPP  $\Sigma t \ g$ )) ++ [self  $\mapsto$  qself]),
    (typeResAPP  $\Sigma t \ g$ ) $\}$ 

apply (rule impI)
apply (erule ValidGlobalRegionEnv.induct)
apply simp
apply (rule impI)+
apply (case-tac g=f)

apply (simp add: typeResAPP-def regionsArgAPP-def typesArgAPP-def)
apply (subst (asm) fun-upd-apply, simp)

```

**apply** (*simp add: typeResAPP-def regionsArgAPP-def typesArgAPP-def*)  
**by** (*subst (asm) fun-upd-apply, simp*)

**lemma** *equiv-SafeRegionDAss-all-n-SafeRegionDAssDepth*:

$e : \llbracket \vartheta, t \rrbracket \implies \forall n. \text{SafeRegionDAssDepth } e \text{ } f \text{ } n \text{ } \vartheta \text{ } t$

**apply** (*case-tac  $\vartheta$* )  
**apply** (*simp only: SafeRegionDAss.simps*)  
**apply** (*simp only: SafeRegionDAssDepth.simps*)  
**apply** *clarsimp*  
**apply** (*simp only: SafeBoundSem-def*)  
**apply** (*simp add: Let-def*)  
**apply** (*elim exE*)  
**apply** (*elim conjE*)  
**apply** (*frule-tac  $td=td$  in eqSemDepthRA*)  
**apply** (*elim exE*)  
**apply** (*case-tac  $x, \text{case-tac } ba$* )  
**apply** (*erule-tac  $x=E1$  in allE*)  
**apply** (*erule-tac  $x=E2$  in allE*)  
**apply** (*erule-tac  $x=h$  in allE*)  
**apply** (*erule-tac  $x=k$  in allE*)  
**apply** (*erule-tac  $x=td$  in allE*)  
**apply** (*erule-tac  $x=h'$  in allE*)  
**apply** (*erule-tac  $x=v$  in allE*)  
**apply** (*erule-tac  $x=aa$  in allE*)  
**apply** (*erule-tac  $x=ab$  in allE*)  
**apply** (*erule-tac  $x=bb$  in allE*)  
**apply** (*erule-tac  $x=\eta$  in allE*)  
**apply** (*drule mp,force*)  
**by** *simp*

**declare** *SafeRegionDAssDepth.simps* [*simp del*]

**lemma** *lemma-19* [*rule-format*]:

*ValidGlobalRegionEnvDepth*  $f \text{ } n \text{ } \Sigma t$   
 $\longrightarrow \Sigma f \text{ } g = \text{Some } (xs, rs, eg)$   
 $\longrightarrow \Sigma t \text{ } g = \text{Some } (ti, qs, tg)$   
 $\longrightarrow g \neq f$   
 $\longrightarrow \vartheta 1 = \text{map-of } (\text{zip } xs \text{ } ti)$   
 $\longrightarrow \vartheta 2 = \text{map-of } (\text{zip } rs \text{ } qs) ++ [\text{self} \mapsto q\text{self}]$   
 $\longrightarrow \text{SafeRegionDAssDepth } eg \text{ } f \text{ } n \text{ } (\vartheta 1, \vartheta 2) \text{ } tg$   
**apply** (*rule impI*)  
**apply** (*erule ValidGlobalRegionEnvDepth.induct*)

**apply** (*rule impI*)  
**apply** (*frule lemma-19-aux,force,force*)

```

apply (frule equiv-SafeRegionDAss-all-n-SafeRegionDAssDepth)
apply (simp add: typeResAPP-def regionsArgAPP-def typesArgAPP-def)
apply (simp add: bodyAPP-def varsAPP-def regionsAPP-def)
apply force

```

```

apply (rule impI)+
apply (frule lemma-19-aux, simp add: fun-upd-apply, force)
apply (frule equiv-SafeRegionDAss-all-n-SafeRegionDAssDepth)
apply (subst (asm) fun-upd-apply, simp)
apply (simp add: typeResAPP-def regionsArgAPP-def typesArgAPP-def)
apply (simp add: bodyAPP-def varsAPP-def regionsAPP-def)
apply force

```

```

apply (rule impI)+
apply (frule lemma-19-aux, simp add: fun-upd-apply, force)
apply (frule equiv-SafeRegionDAss-all-n-SafeRegionDAssDepth)
apply (subst (asm) fun-upd-apply, simp)
apply (simp add: typeResAPP-def regionsArgAPP-def typesArgAPP-def)
apply (simp add: bodyAPP-def varsAPP-def regionsAPP-def)
apply force

```

```

apply (case-tac ga=g,simp-all)
apply (rule impI)+
apply (subst (asm) fun-upd-apply, simp)
apply (simp add: bodyAPP-def varsAPP-def regionsAPP-def)
apply (frule equiv-SafeRegionDAss-all-n-SafeRegionDAssDepth)
apply force
apply (rule impI)+
apply (drule mp,force)
apply (drule mp)
apply (subst (asm) fun-upd-apply, simp)
apply (drule mp, simp)
apply simp
done

```

```

lemma lemma-20 [rule-format]:
  ValidGlobalRegionEnvDepth f n  $\Sigma t$ 
   $\longrightarrow \Sigma f f = \text{Some } (xs,rs,ef)$ 
   $\longrightarrow \Sigma t f = \text{Some } (ti,qs,tf)$ 
   $\longrightarrow \vartheta 1 = \text{map-of } (\text{zip } xs \ ti)$ 
   $\longrightarrow \vartheta 2 = \text{map-of } (\text{zip } rs \ qs) \ ++ \ [self \mapsto qself]$ 
   $\longrightarrow n = \text{Suc } n'$ 

```

$\longrightarrow \text{SafeRegionDAssDepth } ef \ f \ n' \ (\vartheta 1, \vartheta 2) \ tf$   
**apply** (rule impI)  
**apply** (erule ValidGlobalRegionEnvDepth.induct)

**apply** (rule impI)+  
**apply** (frule lemma-19-aux, simp add: fun-upd-apply, force)  
**apply** (frule equiv-SafeRegionDAss-all-n-SafeRegionDAssDepth)  
**apply** (subst (asm) fun-upd-apply, simp)  
**apply** (simp add: typeResAPP-def regionsArgAPP-def typesArgAPP-def)  
**apply** (simp add: bodyAPP-def varsAPP-def regionsAPP-def)  
**apply** force

**apply** simp

**apply** (rule impI)+  
**apply** simp  
**apply** (subst (asm) fun-upd-apply, simp)  
**apply** (simp add: bodyAPP-def varsAPP-def regionsAPP-def)

**apply** (rule impI)+  
**apply** simp  
**apply** (subst (asm) fun-upd-apply, simp)  
**done**

**lemma** SafeDARegionDepth-APP:

$\llbracket \Sigma t \ g = \text{Some } (ti, qs, tg); \text{primops } g = \text{None};$   
 $qs \text{self} \notin \text{regions } tg \cup (\bigcup \text{set } (\text{map } \text{regions } ti)) \cup \text{set } qs;$   
 $\text{length } as = \text{length } ti; \text{length } qs = \text{length } rr;$   
 $\forall i < \text{length } qs. \exists t. \mu 2 \ (qs \ ! \ i) = \text{Some } t;$   
 $qfake \notin \text{ran } \mu 2; \mu\text{-ren-dom } (\mu 1, \mu 2);$   
 $t = \mu\text{-ext } (\mu 1, \mu 2) \ tg; \mu\text{-ext-def } (\mu 1, \mu 2) \ tg;$   
 $\Sigma f \ g = \text{Some } (xs, rs, e); \text{fv } e \subseteq \text{set } xs; \text{fvReg } e \subseteq \text{set } rs \cup \{\text{self}\};$   
 $\text{argP-app } (\text{map } (\mu\text{-ext } (\mu 1, \mu 2)) \ ti) \ (\text{map } (\text{the } \circ \mu 2) \ qs) \ as \ rr \ (\vartheta 1, \vartheta 2);$   
 $\models_f, n \ \Sigma t \rrbracket$   
 $\implies \text{AppE } g \ as \ rr \ a \ :_f, n \ \llbracket (\vartheta 1, \vartheta 2), t \rrbracket$

**apply** (case-tac g≠f)

**apply** (unfold SafeRegionDAssDepth.simps)  
**apply** (intro allI, rule impI)  
**apply** (elim conjE)

**apply** (frule lemma-19)  
**apply** (force, force, assumption+, simp, simp)

**apply** (*frule-tac* ? $\mu 1.0 = \text{fst}(\mu\text{-ren } (\mu 1, \mu 2))$  **and**  
           ? $\mu 2.0 = \text{snd}(\mu\text{-ren } (\mu 1, \mu 2))$  ( $\varrho\text{self} \mapsto \varrho\text{self}$ ) **in** *Regions-Lemma-5-Depth*)

**apply** (*frule* *P1-f-n-APP-2*, *simp*, *force*, *simp*, *force*)  
**apply** (*elim* *exE*, *elim* *conjE*)  
**apply** (*unfold* *SafeRegionDAssDepth.simps*)  
**apply** (*rotate-tac* 24)  
**apply** (*erule-tac*  $x = (\text{map-of } (\text{zip } xs \ (\text{map } (\text{atom2val } E1) \ as)))$  **in** *allE*)  
**apply** (*fold* *SafeRegionDAssDepth.simps*)  
**apply** (*erule-tac*  $x = (\text{map-of } (\text{zip } rs \ (\text{map } (\text{the} \circ E2) \ rr)))$  ( $\text{self} \mapsto \text{Suc } k$ ) **in** *allE*)  
**apply** (*erule-tac*  $x = h$  **in** *allE*)  
**apply** (*rotate-tac* 24)  
**apply** (*erule-tac*  $x = \text{Suc } k$  **in** *allE*)  
**apply** (*rotate-tac* 24)  
**apply** (*erule-tac*  $x = h'a$  **in** *allE*)  
**apply** (*erule-tac*  $x = v$  **in** *allE*)  
**apply** (*rotate-tac* 24)  
**apply** (*erule-tac*  $x = (\eta\text{-ren } \eta) ++ [\varrho\text{self} \mapsto \text{Suc } k]$  **in** *allE*)

**apply** (*drule* *mp*)

**apply** (*rule* *conjI*, *simp*)

**apply** (*rule* *conjI*, *simp*)

**apply** (*rule* *conjI*)  
**apply** (*simp* *add: dom-fun-upd*)

**apply** (*rule* *conjI*)  
**apply** (*frule* *SafeDARegion-APP-E1-P2*, *simp*, *simp*, *simp*)

**apply** (*rule* *conjI*)  
**apply** (*frule* *SafeDARegion-APP-E2-P2*, *simp*, *assumption+*, *simp*, *assumption*, *simp*)

**apply** (*rule* *conjI*)  
**apply** (*rule* *SafeDARegion-APP-P3*, *assumption*)

**apply** (*frule-tac*  $xs = xs$  **and**  $ti = ti$  **and**  $as = as$  **and**  $rs = rs$  **and**  $\varrho s = \varrho s$  **in** *SafeDARegion-APP-P4*)

**apply** (*simp*, *assumption+*, *simp*, *force*, *assumption+*, *simp*, *simp*, *simp*, *assumption+*)

**apply** *simp*

**apply** (*drule consistent-t-consistent-t-ren-inv*)  
**apply** (*subst mu-ext-qself-mu-ext, simp*)  
**apply** (*subst qself-notin-regions-mu-ren, simp*)  
**apply** *simp*  
**apply** (*elim conjE*)  
**apply** (*frule consistent-t-ren-inv-consistent-t, assumption+*)  
**apply** (*rule SafeDARegion-APP-P5, assumption+*)

**apply** *simp*  
**apply** (*case-tac n*)

**apply** (*simp only: SafeRegionDAssDepth.simps*)  
**apply** (*rule allI*)  
**apply** (*rule impI*)  
**apply** (*elim conjE*)  
**apply** (*frule P1-f-n-APP, assumption+, simp*)

**apply** (*unfold SafeRegionDAssDepth.simps*)  
**apply** (*intro allI, rule impI*)  
**apply** (*elim conjE*)

**apply** (*frule lemma-20*)  
**apply** (*force, force, simp, simp, simp*)

**apply** (*frule-tac ?μ1.0=fst(μ-ren (μ1,μ2)) and*  
*?μ2.0=snd (μ-ren (μ1,μ2))(qself ↦ qself) in Regions-Lemma-5-Depth*)  
**apply** (*subgoal-tac (E1, E2) ⊢ h, k, AppE f as rr () ↓ (f, Suc nat) h', k, v*)  
**prefer** 2 **apply** *simp*  
**apply** (*frule P1-f-n-ge-0-APP, simp, force*)  
**apply** (*elim exE, elim conjE*)  
**apply** (*unfold SafeRegionDAssDepth.simps*)  
**apply** (*rotate-tac 27*)  
**apply** (*erule-tac x=(map-of (zip xs (map (atom2val E1) as))) in allE*)  
**apply** (*fold SafeRegionDAssDepth.simps*)  
**apply** (*erule-tac x=(map-of (zip rs (map (the ∘ E2) rr))(self ↦ Suc k)) in allE*)  
**apply** (*erule-tac x=h in allE*)  
**apply** (*rotate-tac 27*)  
**apply** (*erule-tac x=Suc k in allE*)  
**apply** (*rotate-tac 27*)  
**apply** (*erule-tac x=h'a in allE*)  
**apply** (*erule-tac x=v in allE*)

```

apply (rotate-tac 27)
apply (erule-tac  $x=(\eta\text{-ren } \eta) ++ [\varrho\text{self} \mapsto \text{Suc } k]$  in allE)

apply (drule mp)

apply (rule conjI,simp)

apply (rule conjI,simp)

apply (rule conjI)
apply (simp add: dom-fun-upd)

apply (rule conjI)
apply (frule SafeDARegion-APP-E1-P2,simp,simp,simp)

apply (rule conjI)
apply (rule SafeDARegion-APP-E2-P2)
apply (simp,assumption+,simp,assumption+)

apply (rule conjI)
apply (rule SafeDARegion-APP-P3, assumption)

apply (rule-tac  $xs=xs$  and  $ti=ti$  and  $as=as$  and  $rs=rs$  and  $\varrho s=\varrho s$  in SafeDARegion-APP-P4)

apply (simp,simp,assumption+,simp,simp,simp,assumption+,simp,simp,assumption+)

apply simp
apply (drule consistent-t-consistent-t-ren-inv)
  apply (subst mu-ext- $\varrho$ self-mu-ext,simp)
  apply (subst  $\varrho$ self-notin-regions-mu-ren,simp)
  apply simp
apply (frule consistent-t-ren-inv-consistent-t,assumption+)
by (rule SafeDARegion-APP-P5,assumption+)

end

```

## 24 Proof rules for region deallocation

```

theory ProofRulesRegions
imports SafeRegionDepth

```

**begin**

**inductive**

*ProofRulesREG* :: [*unit Exp*, *RegionEnv*, *string*, *ThetaMapping*, *TypeExpression*]  
 $\Rightarrow \text{bool}$

( $\_$ ,  $\_$ ,  $\vdash$ ,  $\_$   $\rightsquigarrow$   $\_$  [*71,71,71,71,71*] *70*)

**where**

*litInt* : *ConstE* (*LitN* *i*) *a*,  $\Sigma t \vdash_f (\vartheta 1, \vartheta 2) \rightsquigarrow (\text{ConstrT intType } [] [])$

| *litBool*: *ConstE* (*LitB* *b*) *a*,  $\Sigma t \vdash_f (\vartheta 1, \vartheta 2) \rightsquigarrow (\text{ConstrT boolType } [] [])$

| *var1* :  $\llbracket \vartheta 1 \ x = \text{Some } t \rrbracket$   
 $\implies \text{VarE } x \ a, \Sigma t \vdash_f (\vartheta 1, \vartheta 2) \rightsquigarrow t$

| *var2* :  $\llbracket \vartheta 1 \ x = \text{Some } (\text{ConstrT } T \ ti \ \varrho l);$   
 $\vartheta 2 \ r = \text{Some } \varrho';$   
*coherent constructorSignature* *Tc*  $\rrbracket$   
 $\implies \text{CopyE } x \ r \ d, \Sigma t \vdash_f (\vartheta 1, \vartheta 2) \rightsquigarrow \text{ConstrT } T \ ti \ ((\text{butlast } \varrho l) @ [\varrho'])$

| *var3* :  $\llbracket \vartheta 1 \ x = \text{Some } t; \text{coherent constructorSignature } Tc \rrbracket$   
 $\implies \text{ReuseE } x \ a, \Sigma t \vdash_f (\vartheta 1, \vartheta 2) \rightsquigarrow t$

| *let1* :  $\llbracket \forall \ C \ \text{as } r \ a'. \ e1 \neq \text{ConstrE } C \ \text{as } r \ a';$   
 $x1 \notin \text{dom } \vartheta 1; x1 \notin \text{fv } e1;$   
 $e1, \Sigma t \vdash_f (\vartheta 1, \vartheta 2) \rightsquigarrow t1;$   
 $e2, \Sigma t \vdash_f (\vartheta 1(x1 \mapsto t1), \vartheta 2) \rightsquigarrow t2 \rrbracket$   
 $\implies \text{Let } x1 = e1 \ \text{In } e2 \ a, \Sigma t \vdash_f (\vartheta 1, \vartheta 2) \rightsquigarrow t2$

| *letc* :  $\llbracket x1 \notin \text{fvs } as; x1 \notin \text{dom } \vartheta 1;$   
*constructorSignature* *C* = *Some* (*ti*,  $\varrho$ , *t*);  
 $t = \text{ConstrT } T \ tn \ \varrho s;$   
 $t' = \text{mu-ext } (\mu 1, \mu 2) \ t;$   
 $\text{argP } (\text{map } (\text{mu-ext } (\mu 1, \mu 2)) \ ti) \ ((\text{the } \circ \mu 2) \ \varrho) \ \text{as } r \ (\vartheta 1, \vartheta 2);$   
 $\text{wellT } ti \ \varrho \ t;$   
 $e2, \Sigma t \vdash_f (\vartheta 1(x1 \mapsto t'), \vartheta 2) \rightsquigarrow t' \rrbracket$   
 $\implies \text{Let } x1 = \text{ConstrE } C \ \text{as } r \ a' \ \text{In } e2 \ a, \Sigma t \vdash_f (\vartheta 1, \vartheta 2) \rightsquigarrow t''$

| *case1* :  $\llbracket \text{length } \text{assert} = \text{length } \text{alts}; \text{length } \text{alts} > 0;$   
 $\forall \ i < \text{length } \text{alts}. \text{constructorSignature } (\text{fst } (\text{extractP } (\text{fst } (\text{alts } ! \ i))))$   
 $= \text{Some } (ti, \varrho, t) \wedge$   
 $t = \text{ConstrT } T \ tn \ \varrho s \wedge$   
 $\text{length } (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! \ i)))) = \text{length } ti \wedge$   
 $\text{wellT } ti \ \varrho \ t \wedge$   
 $\text{fst } (\text{assert } ! \ i) = \vartheta 1 \ ++ \ (\text{map-of } (\text{zip } (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! \ i))))$   
 $\quad (\text{map } (\text{mu-ext } \mu) \ ti))) \wedge$   
 $\text{dom } \vartheta 1 \cap \text{dom } (\text{map-of } (\text{zip } (\text{snd } (\text{extractP } (\text{fst } (\text{alts } ! \ i))))$   
 $\quad (\text{map } (\text{mu-ext } \mu) \ ti))) = \{\}$   $\rrbracket$



$$\begin{aligned}
& \vartheta 1 \ x = \text{Some} \ (\mu\text{-ext} \ \mu \ t) \ \wedge \\
& \text{snd} \ (\text{assert} \ ! \ i) = \vartheta 2; \\
& \forall \ i < \text{length} \ \text{alts}. \ \text{snd} \ (\text{alts} \ ! \ i), \ \Sigma t \\
& \quad \vdash_f \ (\text{fst} \ (\text{assert} \ ! \ i), \ \text{snd} \ (\text{assert} \ ! \ i)) \rightsquigarrow t'; \\
& \forall \ i < \text{length} \ \text{alts}. \ x \notin \text{set} \ (\text{snd} \ (\text{extractP} \ (\text{fst} \ (\text{alts} \ ! \ i)))) \\
& \implies \text{Case} \ (\text{VarE} \ x \ a) \ \text{Of} \ \text{alts} \ a', \ \Sigma t \vdash_f \ (\vartheta 1, \vartheta 2) \rightsquigarrow t'
\end{aligned}$$
  

$$\begin{aligned}
| \ \text{case2} \ : \ & \llbracket \text{length} \ \text{assert} = \text{length} \ \text{alts}; \ \text{length} \ \text{alts} > 0; \\
& \text{coherent} \ \text{constructorSignature} \ Tc; \\
& \forall \ i < \text{length} \ \text{alts}. \ \text{constructorSignature} \\
& \quad (\text{fst} \ (\text{extractP} \ (\text{fst} \ (\text{alts} \ ! \ i)))) = \text{Some} \ (ti, \varrho, t) \wedge \\
& \quad t = \text{ConstrT} \ T \ tn \ \varrho s \wedge \\
& \quad \text{length} \ (\text{snd} \ (\text{extractP} \ (\text{fst} \ (\text{alts} \ ! \ i)))) = \text{length} \ ti \wedge \\
& \quad \text{wellT} \ ti \ \varrho \ t \wedge \\
& \text{fst} \ (\text{assert} \ ! \ i) = \vartheta 1 \ ++ \ (\text{map-of} \ (\text{zip} \ (\text{snd} \ (\text{extractP} \ (\text{fst} \ (\text{alts} \ ! \ i)))) \\
& \quad \quad \quad (\text{map} \ (\mu\text{-ext} \ \mu) \ ti))) \wedge \\
& \quad \text{dom} \ \vartheta 1 \cap \text{dom} \ (\text{map-of} \ (\text{zip} \ (\text{snd} \ (\text{extractP} \ (\text{fst} \ (\text{alts} \ ! \ i)))) \\
& \quad \quad \quad (\text{map} \ (\mu\text{-ext} \ \mu) \ ti))) = \{\} \wedge \\
& \quad \vartheta 1 \ x = \text{Some} \ (\mu\text{-ext} \ \mu \ t) \ \wedge \\
& \quad \text{snd} \ (\text{assert} \ ! \ i) = \vartheta 2; \\
& \forall \ i < \text{length} \ \text{alts}. \ \text{snd} \ (\text{alts} \ ! \ i), \ \Sigma t \\
& \quad \vdash_f \ (\text{fst} \ (\text{assert} \ ! \ i), \ \text{snd} \ (\text{assert} \ ! \ i)) \rightsquigarrow t'; \\
& \forall \ i < \text{length} \ \text{alts}. \ x \notin \text{set} \ (\text{snd} \ (\text{extractP} \ (\text{fst} \ (\text{alts} \ ! \ i)))) \\
& \implies \text{CaseD} \ (\text{VarE} \ x \ a) \ \text{Of} \ \text{alts} \ a', \ \Sigma t \vdash_f \ (\vartheta 1, \vartheta 2) \rightsquigarrow t'
\end{aligned}$$
  

$$\begin{aligned}
| \ \text{app} \ : \ & \llbracket \Sigma t \ g = \text{Some} \ (ti, \varrho s, tg); \ \text{primops} \ g = \text{None}; \\
& \varrho \text{self} \notin \text{regions} \ tg \cup (\bigcup \ \text{set} \ (\text{map} \ \text{regions} \ ti)) \cup \text{set} \ \varrho s; \\
& \text{length} \ as = \text{length} \ ti; \ \text{length} \ \varrho s = \text{length} \ rr; \\
& \forall \ i < \text{length} \ \varrho s. \ \exists \ t. \ \mu 2 \ (\varrho s \ ! \ i) = \text{Some} \ t; \\
& \varrho \text{fake} \notin \text{ran} \ \mu 2; \ \mu\text{-ren-dom} \ (\mu 1, \mu 2); \\
& \quad t = \mu\text{-ext} \ (\mu 1, \mu 2) \ tg; \ \mu\text{-ext-def} \ (\mu 1, \mu 2) \ tg; \\
& \quad \Sigma f \ g = \text{Some} \ (xs, rs, e); \ \text{fv} \ e \subseteq \text{set} \ xs; \ \text{fvReg} \ e \subseteq \text{set} \ rs \cup \{\text{self}\}; \\
& \quad \text{argP-app} \ (\text{map} \ (\mu\text{-ext} \ (\mu 1, \mu 2)) \ ti) \ (\text{map} \ (\text{the} \circ \mu 2) \ \varrho s) \ as \ rr \\
& \quad (\vartheta 1, \vartheta 2) \rrbracket \\
& \implies \text{AppE} \ g \ as \ rr \ a, \ \Sigma t \vdash_f \ (\vartheta 1, \vartheta 2) \rightsquigarrow t
\end{aligned}$$
  

$$\begin{aligned}
| \ \text{rec} \ : \ & \llbracket \Sigma f \ f = \text{Some} \ (xs, rs, ef); \\
& \quad f \notin \text{dom} \ \Sigma t; \\
& \quad \vartheta 1 f = \text{map-of} \ (\text{zip} \ xs \ ti); \\
& \quad \vartheta 2 f = \text{map-of} \ (\text{zip} \ rs \ \varrho s) \ ++ \ [\text{self} \mapsto \varrho \text{self}]; \\
& \quad ef, \ \Sigma t (f \mapsto (ti, \varrho s, tf)) \vdash_f \ (\vartheta 1 f, \vartheta 2 f) \rightsquigarrow tf \rrbracket \\
& \implies ef, \ \Sigma t \vdash_f \ (\vartheta 1 f, \vartheta 2 f) \rightsquigarrow tf
\end{aligned}$$

**declare** *SafeRegionDAss.simps* [*simp del*]  
**declare** *SafeRegionDAssDepth.simps* [*simp del*]

**lemma** *equiv-all-n-SafeRegionDAssDepth-SafeRegionDAss:*  
 $\forall n. \text{SafeRegionDAssDepth } e \text{ f } n \ \vartheta \ t \implies e : \llbracket \vartheta, t \rrbracket$   
**apply** (*case-tac*  $\vartheta$ )  
**apply** (*simp only: SafeRegionDAss.simps*)  
**apply** (*simp only: SafeRegionDAssDepth.simps*)  
**apply** (*rule allI*)  
**apply** (*rule impI*)  
**apply** (*elim conjE*)  
**apply** (*frule-tac*  $f=f$  **in** *eqSemRADepth*)  
**apply** (*simp only: SafeBoundSem-def*)  
**apply** (*elim exE*)  
**apply** (*rename-tac*  $n$ )  
**apply** (*erule-tac*  $x=n$  **in** *allE*)  
**apply** (*erule-tac*  $x=E1$  **in** *allE*)  
**apply** (*erule-tac*  $x=E2$  **in** *allE*)  
**apply** (*erule-tac*  $x=h$  **in** *allE*)  
**apply** (*erule-tac*  $x=k$  **in** *allE*)  
**apply** (*erule-tac*  $x=h'$  **in** *allE*)  
**apply** (*erule-tac*  $x=v$  **in** *allE*)  
**apply** (*erule-tac*  $x=\eta$  **in** *allE*)  
  
**apply** (*simp add: Let-def*)  
**apply** (*drule mp,force*)  
**by** *simp*

**lemma** *equiv-SafeRegionDAss-all-n-SafeRegionDAssDepth:*  
 $e : \llbracket \vartheta, t \rrbracket \implies \forall n. \text{SafeRegionDAssDepth } e \text{ f } n \ \vartheta \ t$   
**apply** (*case-tac*  $\vartheta$ )  
**apply** (*simp only: SafeRegionDAss.simps*)  
**apply** (*simp only: SafeRegionDAssDepth.simps*)  
**apply** *clarsimp*  
**apply** (*simp only: SafeBoundSem-def*)  
**apply** (*simp add: Let-def*)  
**apply** (*elim exE*)  
**apply** (*elim conjE*)  
**apply** (*frule-tac*  $td=td$  **in** *eqSemDepthRA*)  
**apply** (*elim exE*)  
**apply** (*case-tac*  $x, \text{case-tac } ba$ )  
**apply** (*erule-tac*  $x=E1$  **in** *allE*)  
**apply** (*erule-tac*  $x=E2$  **in** *allE*)  
**apply** (*erule-tac*  $x=h$  **in** *allE*)  
**apply** (*erule-tac*  $x=k$  **in** *allE*)  
**apply** (*erule-tac*  $x=td$  **in** *allE*)  
**apply** (*erule-tac*  $x=h'$  **in** *allE*)  
**apply** (*erule-tac*  $x=v$  **in** *allE*)  
**apply** (*erule-tac*  $x=aa$  **in** *allE*)

**apply** (*erule-tac*  $x=ab$  **in**  $allE$ )  
**apply** (*erule-tac*  $x=bb$  **in**  $allE$ )  
**apply** (*erule-tac*  $x=\eta$  **in**  $allE$ )  
**apply** (*drule*  $mp,force$ )  
**by** *simp*

**lemma** *lemma-5*:

$\forall n. SafeRegionDAssDepth\ e\ f\ n\ \vartheta\ t \equiv e : \{\vartheta, t\}$   
**apply** (*rule* *eq-reflection*)  
**apply** (*rule* *iffI*)

**apply** (*rule* *equiv-all-n-SafeRegionDAssDepth-SafeRegionDAss,force*)

**by** (*rule* *equiv-SafeRegionDAss-all-n-SafeRegionDAssDepth,force*)

**declare** *fun-upd-apply* [*simp del*]

**lemma** *imp-ValidGlobalRegionEnv-all-n-ValidGlobalRegionEnvDepth*:

$ValidGlobalRegionEnv\ \Sigma t \implies \forall n. \models_{f,n} \Sigma t$   
**apply** (*erule* *ValidGlobalRegionEnv.induct,simp-all*)  
**apply** (*rule* *allI*)  
**apply** (*rule* *ValidGlobalRegionEnvDepth.base*)  
**apply** (*rule* *ValidGlobalRegionEnv.base*)  
**apply** *simp*  
**apply** (*rule* *allI*)  
**apply** (*case-tac*  $fa=f,simp$ )  
**apply** (*induct-tac*  $n$ )  
**apply** (*rule* *ValidGlobalRegionEnvDepth.depth0,simp,simp*)  
**apply** (*rule* *ValidGlobalRegionEnvDepth.step*)  
**apply** (*simp,simp,simp,simp,simp*)  
**apply** (*frule-tac*  $f=f$  **in** *equiv-SafeRegionDAss-all-n-SafeRegionDAssDepth,simp*)  
  
**apply** (*rule* *ValidGlobalRegionEnvDepth.g*)  
**apply** (*simp,simp,simp,simp,simp*)  
**by** (*frule-tac*  $f=fa$  **in** *equiv-SafeRegionDAss-all-n-SafeRegionDAssDepth,simp*)

**lemma** *imp-ValidRegionDepth-n-SigmaRegion-Valid-Sigma* [*rule-format*]:

$\models_{f,n} \Sigma t$   
 $\longrightarrow f \notin dom\ \Sigma t$   
 $\longrightarrow ValidGlobalRegionEnv\ \Sigma t$   
**apply** (*rule* *impI*)  
**apply** (*erule* *ValidGlobalRegionEnvDepth.induct,simp-all*)  
**apply** (*simp add: fun-upd-apply add: dom-def*)  
**apply** (*simp add: fun-upd-apply add: dom-def*)

**apply** (*rule impI*)  
**apply** (*drule mp*)  
**apply** (*simp add: fun-upd-apply add: dom-def*)  
**by** (*rule ValidGlobalRegionEnv.step, simp-all*)

**lemma** *imp-f-notin-SigmaRegion-ValidDepth-n-SigmaRegion-Valid-Sigma*:  
 $\llbracket f \notin \text{dom } \Sigma t; \forall n. \models_f, n \Sigma t \rrbracket$   
 $\implies \text{ValidGlobalRegionEnv } \Sigma t$   
**apply** (*erule-tac x=n in allE*)  
**by** (*rule imp-ValidRegionDepth-n-SigmaRegion-Valid-Sigma, assumption+*)

**lemma** *Theorem-4-aux* [*rule-format*]:  
 $\text{ValidGlobalRegionEnvDepth } f \ n \ \Sigma t$   
 $\longrightarrow n = \text{Suc } n'$   
 $\longrightarrow f \in \text{dom } \Sigma t$   
 $\longrightarrow (\text{bodyAPP } \Sigma f f) :_f, n' \Vdash (\text{map-of } (\text{zip } (\text{varsAPP } \Sigma f f) (\text{typesArgAPP } \Sigma t f)), \text{map-of } (\text{zip } (\text{regionsAPP } \Sigma f f) (\text{regionsArgAPP } \Sigma t f)) ++ [\text{self} \mapsto \varrho \text{self}]),$   
 $(\text{typeResAPP } \Sigma t f) \Vdash$   
**apply** (*rule impI*)  
**apply** (*erule ValidGlobalRegionEnvDepth.induct, simp-all*)  
**apply** (*rule impI*)  
**apply** (*subgoal-tac typesArgAPP* ( $\Sigma t(f \mapsto (ti, \varrho s, tf))$ )  $f = ti, \text{simp}$ )  
**apply** (*subgoal-tac regionsArgAPP* ( $\Sigma t(f \mapsto (ti, \varrho s, tf))$ )  $f = \varrho s, \text{simp}$ )  
**apply** (*subgoal-tac typeResAPP* ( $\Sigma t(f \mapsto (ti, \varrho s, tf))$ )  $f = tf, \text{simp}$ )  
**apply** (*unfold typeResAPP-def regionsArgAPP-def typesArgAPP-def*)  
**by** (*simp add: fun-upd-apply add: dom-def*)

**lemma** *Theorem-4*:  
 $\llbracket \forall n > 0. \text{ValidGlobalRegionEnvDepth } f \ n \ \Sigma t; f \in \text{dom } \Sigma t \rrbracket$   
 $\implies \forall n. (\text{bodyAPP } \Sigma f f) :_f, n \Vdash (\text{map-of } (\text{zip } (\text{varsAPP } \Sigma f f) (\text{typesArgAPP } \Sigma t f)),$   
 $\text{map-of } (\text{zip } (\text{regionsAPP } \Sigma f f) (\text{regionsArgAPP } \Sigma t f)) ++ [\text{self} \mapsto \varrho \text{self}]),$   
 $(\text{typeResAPP } \Sigma t f) \Vdash$   
**apply** (*rule allI*)  
**apply** (*rule-tac n=Suc n in Theorem-4-aux*)  
**by** *simp-all*

**lemma** *Theorem-5-aux* [*rule-format*]:  
 $\models_f, n \Sigma t$   
 $\longrightarrow n = \text{Suc } n'$   
 $\longrightarrow f \in \text{dom } \Sigma t$   
 $\longrightarrow (\text{bodyAPP } \Sigma f f) : \Vdash (\text{map-of } (\text{zip } (\text{varsAPP } \Sigma f f) (\text{typesArgAPP } \Sigma t f)),$

$$\begin{aligned} & \text{map-of } (\text{zip } (\text{regionsAPP } \Sigma f f) (\text{regionsArgAPP } \Sigma t f)) ++ \\ & [\text{self} \mapsto \varrho \text{self}], \\ & \text{map-of } (\text{zip } (\text{regionsAPP } \Sigma f f) (\text{regionsArgAPP } \Sigma t f)) ++ \\ & (\text{typeResAPP } \Sigma t f) \} \\ & \longrightarrow \models \Sigma t \\ & \text{apply (rule impI)} \\ & \text{apply (erule ValidGlobalRegionEnvDepth.induct,simp-all)} \\ & \text{apply (rule impI)+} \\ & \text{apply (rule ValidGlobalRegionEnv.step)} \\ & \text{apply (simp,simp,simp,simp,simp)} \\ & \text{apply (subgoal-tac typesArgAPP } (\Sigma t(f \mapsto (ti, \varrho s, tf))) f = ti, \text{simp})} \\ & \text{apply (subgoal-tac regionsArgAPP } (\Sigma t(f \mapsto (ti, \varrho s, tf))) f = \varrho s, \text{simp})} \\ & \text{apply (subgoal-tac typeResAPP } (\Sigma t(f \mapsto (ti, \varrho s, tf))) f = tf, \text{simp})} \\ & \text{apply (simp add: typeResAPP-def add: fun-upd-apply add: dom-def)} \\ & \text{apply (simp add: regionsArgAPP-def add: fun-upd-apply add: dom-def)} \\ & \text{apply (simp add: typesArgAPP-def add: fun-upd-apply add: dom-def)} \\ & \text{apply (rule impI)+} \\ & \text{apply (case-tac g=f,simp-all)} \\ & \text{apply (rule ValidGlobalRegionEnv.step,simp-all)} \\ & \text{apply (subgoal-tac } f \in \text{dom } \Sigma t, \text{simp})} \\ & \text{prefer 2 apply (simp add: fun-upd-apply add: dom-def)} \\ & \text{apply (subgoal-tac typesArgAPP } (\Sigma t(g \mapsto (ti, \varrho s, tf))) f = \text{typesArgAPP } \Sigma t \\ & f, \text{simp})} \\ & \text{apply (subgoal-tac regionsArgAPP } (\Sigma t(g \mapsto (ti, \varrho s, tf))) f = \text{regionsArgAPP } \Sigma t \\ & f, \text{simp})} \\ & \text{apply (subgoal-tac typeResAPP } (\Sigma t(g \mapsto (ti, \varrho s, tf))) f = \text{typeResAPP } \Sigma t f, \text{simp})} \\ & \text{apply (unfold typeResAPP-def regionsArgAPP-def typesArgAPP-def)} \\ & \text{by (simp add: fun-upd-apply add: dom-def)+} \end{aligned}$$

**lemma Theorem-5:**

$$\begin{aligned} & \llbracket \forall n > 0. \models_f, n \Sigma t; f \in \text{dom } \Sigma t; \\ & \quad (\text{bodyAPP } \Sigma f f) : \{ \text{map-of } (\text{zip } (\text{varsAPP } \Sigma f f) (\text{typesArgAPP } \Sigma t f)), \\ & \quad \quad \text{map-of } (\text{zip } (\text{regionsAPP } \Sigma f f) (\text{regionsArgAPP } \Sigma t f)) ++ \\ & \quad \quad (\text{typeResAPP } \Sigma t f) \} \rrbracket \\ & \implies \models \Sigma t \\ & \text{apply (rule-tac } n = \text{Suc } n \text{ in Theorem-5-aux)} \\ & \text{by simp-all} \end{aligned}$$

**lemma imp-f-in-SigmaRegion-ValidDepth-n-SigmaRegion-Valid-Sigma:**

$$\begin{aligned} & \llbracket \forall n. \models_f, n \Sigma t; f \in \text{dom } \Sigma t \rrbracket \\ & \implies \text{ValidGlobalRegionEnv } \Sigma t \\ & \text{apply (subgoal-tac } \models_f, n \Sigma t) \\ & \text{prefer 2 apply simp} \\ & \text{apply (subgoal-tac } \models_f, 0 \Sigma t \wedge (\forall n > 0. \models_f, n \Sigma t), \text{elim conjE}) \\ & \text{prefer 2 apply simp} \\ & \text{apply (frule Theorem-4, assumption+)} \end{aligned}$$

**apply** (*frule Theorem-5,assumption+*)  
**by** (*rule equiv-all-n-SafeRegionDAssDepth-SafeRegionDAss,simp,simp*)

**lemma** *imp-all-n-ValidGlobalRegionEnvDepth-ValidGlobalRegionEnv*:  
 $\llbracket \forall n. \models_{f,n} \Sigma t \rrbracket \implies \text{ValidGlobalRegionEnv } \Sigma t$   
**apply** (*case-tac f \notin dom \Sigma t,simp-all*)  
**apply** (*rule imp-f-notin-SigmaRegion-ValidDepth-n-SigmaRegion-Valid-Sigma,assumption+*)  
**by** (*rule imp-f-in-SigmaRegion-ValidDepth-n-SigmaRegion-Valid-Sigma,assumption+*)

**lemma** *lemma-6*:  
 $\forall n. \models_{f,n} \Sigma t \equiv \text{ValidGlobalRegionEnv } \Sigma t$   
**apply** (*rule eq-reflection*)  
**apply** (*rule iffI*)

**apply** (*rule-tac f=f in imp-all-n-ValidGlobalRegionEnvDepth-ValidGlobalRegionEnv,force*)  
**by** (*rule imp-ValidGlobalRegionEnv-all-n-ValidGlobalRegionEnvDepth,force*)

**lemma** *lemma-7*:  
 $\llbracket \forall n. e, \Sigma t :_{f,n} \{ \vartheta, t \} \rrbracket \implies \text{SafeRegionDAssCntxt } e \Sigma t \vartheta t$   
**apply** (*simp only: SafeRegionDAssDepthCntxt-def*)  
**apply** (*subgoal-tac (\forall n. \models\_{f,n} \Sigma t) \longrightarrow (\forall n. e :\_{f,n} \{ \vartheta, t \})*)  
**apply** (*erule thin-rl*)  
**apply** (*subst (asm) lemma-5*)  
**apply** (*subst (asm) lemma-6*)  
**apply** (*simp add: SafeRegionDAssCntxt-def*)  
**by** *force*

**lemma** *lemma-8-REC [rule-format]*:  
 $(\forall n. (\text{ValidGlobalRegionEnvDepth } f \ n (\Sigma t(f \mapsto (ti,qs,tf))))$   
 $\longrightarrow (\text{bodyAPP } \Sigma f f) :_{f,n} \{ (\text{map-of } (\text{zip } (\text{varsAPP } \Sigma f f) \ ti), \text{map-of } (\text{zip}$   
 $(\text{regionsAPP } \Sigma f f) \ qs)(\text{self} \mapsto \text{qself}), \text{tf} \} )$   
 $\longrightarrow f \notin \text{dom } \Sigma t$   
 $\longrightarrow \text{ValidGlobalRegionEnvDepth } f \ n \Sigma t$   
 $\longrightarrow (\text{bodyAPP } \Sigma f f) :_{f,n} \{ (\text{map-of } (\text{zip } (\text{varsAPP } \Sigma f f) \ ti), \text{map-of } (\text{zip}$

```

(regionsAPP  $\Sigma f$  f) qs)(self  $\mapsto$  qsself)), tf }
apply (rule impI)
apply (induct-tac n)

apply (rule impI)+
apply (erule-tac x=0 in allE)
apply (frule imp-ValidRegionDepth-n-SigmaRegion-Valid-Sigma,assumption+)
apply (subgoal-tac  $\models_f$  , 0  $\Sigma t(f \mapsto (ti,qs,tf))$ ),simp)
apply (rule ValidGlobalRegionEnvDepth.depth0,assumption+)

apply (erule-tac x=Suc n in allE)
apply (rule impI)+
apply (frule imp-ValidRegionDepth-n-SigmaRegion-Valid-Sigma,assumption+)
apply (subgoal-tac  $\models_f$  , n  $\Sigma t$ ,simp)
apply (subgoal-tac  $\models_f$  , Suc n  $\Sigma t(f \mapsto (ti,qs,tf))$ ),simp)
apply (rule ValidGlobalRegionEnvDepth.step,simp-all)
by (rule ValidGlobalRegionEnvDepth.base,assumption+)

lemma lemma-8:
  e,  $\Sigma t \vdash_f \vartheta \leadsto t$ 
   $\implies \forall n. e, \Sigma t :_{f,n} \Vdash \vartheta, t \Vdash$ 
apply (erule ProofRulesREG.induct)

apply (simp only: SafeRegionDAssDepthCntxt-def)
apply (rule allI, rule impI)
apply (rule SafeDARegionDepth-LitInt)

apply (simp only: SafeRegionDAssDepthCntxt-def)
apply (rule allI, rule impI)
apply (rule SafeDARegionDepth-LitBool)

apply (simp only: SafeRegionDAssDepthCntxt-def)
apply (rule allI, rule impI)
apply (rule SafeDARegionDepth-Var1,force)

apply (simp only: SafeRegionDAssDepthCntxt-def)
apply (rule allI, rule impI)
apply (rule SafeDARegionDepth-Var2,force,force,force)

apply (simp only: SafeRegionDAssDepthCntxt-def)
apply (rule allI, rule impI)
apply (rule SafeDARegionDepth-Var3,force,force)

```

```

apply (rule allI)
apply (simp only: SafeRegionDAssDepthCntxt-def)
apply (rule impI)
apply (erule-tac x=n in allE)+
apply (drule mp, simp)+
apply (rule SafeDARegionDepth-LET1)
apply assumption+

```

```

apply (rule allI)
apply (simp only: SafeRegionDAssDepthCntxt-def)
apply (rule impI)
apply (erule-tac x=n in allE)+
apply (drule mp, simp)+
apply (rule SafeDARegionDepth-LETC)
apply (assumption+,force,assumption+,simp)

```

```

apply (rule allI)
apply (simp only: SafeRegionDAssDepthCntxt-def)
apply (rule impI)
apply (subgoal-tac
   $\forall i < \text{length } \text{alts}. \text{snd } (\text{alts } ! i) : f, n \Vdash (\text{fst } (\text{assert } ! i), \text{snd } (\text{assert } ! i)) , t' \Vdash$ )
prefer 2 apply force
apply (rule SafeDARegionDepth-CASE)
apply assumption+

```

```

apply (rule allI)
apply (simp only: SafeRegionDAssDepthCntxt-def)
apply (rule impI)
apply (subgoal-tac
   $\forall i < \text{length } \text{alts}. \text{snd } (\text{alts } ! i) : f, n \Vdash (\text{fst } (\text{assert } ! i), \text{snd } (\text{assert } ! i)) , t' \Vdash$ )
prefer 2 apply force
apply (rule SafeDARegionDepth-CASED)
apply assumption+

```

```

apply (rule allI)
apply (simp only: SafeRegionDAssDepthCntxt-def)
apply (rule impI)
apply (rule SafeDARegionDepth-APP)
apply (assumption+,simp,assumption+,simp,assumption+,simp)

```

```

apply (simp only: SafeRegionDAssDepthCntxt-def)
apply (rule allI,rule impI)

```



```

apply (subgoal-tac
   $ef = (bodyAPP \Sigma f f) \wedge$ 
   $xs = (varsAPP \Sigma f f) \wedge$ 
   $rs = (regionsAPP \Sigma f f), simp$ )
apply (rule lemma-8-REC,force,force,force)
by (simp add: bodyAPP-def add: varsAPP-def add: regionsAPP-def)

```

```

lemma lemma-2:
   $e, \Sigma t \vdash_f \vartheta \rightsquigarrow t$ 
   $\implies e, \Sigma t : \llbracket \vartheta, t \rrbracket$ 
apply (rule lemma-7)
by (rule lemma-8,assumption)

```

```

end

```