Isabelle/HOL theories for Certification of the CoreSafe to SafeImp translation

By Javier de Dios and Ricardo Peña

March, 2009

Contents

1 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*

— Destructions of datatype val
consts *the-IntT* :: *Val* ⇒ *int*
primrec
the-IntT (*IntT i*) = *i*

consts *the-BoolT* :: *Val* ⇒ *bool*
primrec
the-BoolT (*BoolT b*) = *b*

— check if is constant bool
constdefs *isBool* :: *Val* ⇒ *bool*
isBool v ≡ (*case v of* (*BoolT* -) ⇒ *True*
| - ⇒ *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 $0 \ldots k$.

types
Cell = *Constructor* × *Val list*
Region = *nat*
HeapMap = *Location* ⇀ (*Region* × *Cell*)
Heap = *HeapMap* × *nat*

consts
restrictToRegion :: *Heap* => *Region* => *Heap* (**infix** ↓ *110*)

primrec
(*h*,*k*) ↓ *k0* = (*let A* = { *p* . *p* ∈ *dom h* & *fst* (*the* (*h p*)) <= *k0*}
in (*h* |‘ *A*,*k0*))

constdefs
fresh :: *Location* ⇒ *HeapMap* ⇒ *bool*
fresh p h ≡ *p* ∉ *dom h*

constdefs
getFresh :: *HeapMap* ⇒ *Location*
getFresh h ≡ *SOME b*. *fresh b h*

'copy' is a runtime support function copying the recursive part of a data structure.

consts
copy :: [*Heap*,*Region*,*Location*] => *Heap* × *Location*

constdefs
self :: *string* — this identifies the topmost region referenced in a function body
self ≡ ″*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* | *BoolArg* | *NonRecursive* | *Recursive*

types
ConstructorTableType = (*Constructor* × (*nat* × *nat* × *ArgType list*)) *list*
ConstructorTableFun = *Constructor* ⇀ (*nat* × *nat* × *ArgType list*)

This is the constructor table of the Safe expressions semantics. It is assumed to be a constant which somebody else will provide. It is used in the semantic function 'copy'

consts

$ConstructorTable :: ConstructorTableFun$

constdefs $getConstructorCell :: Cell \Rightarrow Constructor$
$getConstructorCell\ c \equiv fst\ c$

constdefs $getValuesCell :: Cell \Rightarrow Val\ list$
$getValuesCell\ c \equiv snd\ c$

constdefs $getCell :: Heap \Rightarrow Location \Rightarrow Cell$
$getCell\ h\ l \equiv snd\ (the\ ((fst\ h)\ l))$

constdefs $getRegion :: Heap \Rightarrow Location \Rightarrow Region$
$getRegion\ h\ l \equiv fst\ (the\ ((fst\ h)\ l))$

constdefs $domHeap :: Heap \Rightarrow Location\ set$
$domHeap\ h \equiv dom\ (fst\ h)$

constdefs $isNonBasicValue :: ArgType \Rightarrow bool$
$isNonBasicValue\ a == (a = NonRecursive) \vee (a = Recursive)$

constdefs $isRecursive :: ArgType \Rightarrow bool$
$isRecursive\ a == (a = Recursive)$

consts
$theLocation :: Val \Rightarrow Location$
primrec $theLocation\ (Loc\ l) = l$

constdefs
$getArgType\ C \equiv snd\ (snd\ (the\ (ConstructorTable\ C)))$

constdefs $getRecursiveValuesCell :: Cell \Rightarrow Location\ set$
$getRecursiveValuesCell\ c == set\ (map\ (theLocation\ o\ snd)$
$(filter\ (isRecursive\ o\ fst)\ (zip\ (getArgType\ (getConstructorCell$
$c))\ (getValuesCell\ c))))$

constdefs $recDescendants :: Location \Rightarrow Heap \Rightarrow Location\ set$
$recDescendants\ l\ h \equiv case\ ((fst\ h)\ l)\ of\ Some\ c \Rightarrow getRecursiveValuesCell\ (snd\ c)$
$\mid None \Rightarrow \{\}$

constdefs $getNonBasicValuesCell :: Cell \Rightarrow Location\ set$
$getNonBasicValuesCell\ c == set\ (map\ (theLocation\ o\ snd)$
$(filter\ (isNonBasicValue\ o\ fst)\ (zip\ (getArgType$
$(getConstructorCell\ c))\ (getValuesCell\ c))))$

constdefs $descendants :: Location \Rightarrow Heap \Rightarrow Location\ set$
$descendants\ l\ h \equiv case\ ((fst\ h)\ l)\ of\ Some\ c \Rightarrow getNonBasicValuesCell\ (snd\ c)$

```
| None ⇒ {}
```

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

2 Useful functions 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))


thm mapAccumL.induct


primrec
  unzip3 []         = ([],[],[])
  unzip3 (tup#tups) = (let (xs,ys,zs) = unzip3 tups;
                           (x,y,z)    = tup
                       in  (x#xs,y#ys,z#zs))
primrec
  unzip []         = ([],[])
  unzip (tup#tups) = (let (xs,ys) = unzip tups;
                          (x,y)   = tup
                      in  (x#xs,y#ys))

primrec
  zipWith f (x#xs) yy = (case yy of
```

```
                    []    => []
                  | y#ys => f x y # zipWith f xs ys)
  zipWith f []      yy = []
```

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

3 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 expresion LetE and in the definition of 'a Der, in order to avoid unnecessary mutual recursion between types. First, we define the key elements of CoreSafe 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
```

3.1 Free Variables

```
fun pat2var :: 'a Patron => string
where
```

```
  pat2var (VarP x -) = x


fun extractP :: 'a Patron ⇒ (string × string list)
where
  extractP (ConstrP C ps a) = (
            let xs = map pat2var ps
            in (C,xs))

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
varProg (CaseDE  exp alts a)    = varProg exp ∪ varProgAlts' alts

varProgs []         = {}
varProgs (exp#exps) = varProg exp ∪ varProgs exps
```

$varProgs'\ []\ = \{\}$
$varProgs'\ (exp\#exps) = varProg\ exp \cup varProgs'\ exps$

$varProgAlts\ []\ = \{\}$
$varProgAlts\ (alt\#alts) = varProgTup\ alt \cup varProgAlts\ alts$

$varProgAlts'\ []\ = \{\}$
$varProgAlts'\ (alt\#alts) = varProgTup'\ alt \cup varProgAlts'\ alts$

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

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

consts $fv\ :: 'a\ Exp \Rightarrow string\ set$
$fvs\ :: 'a\ Exp\ list \Rightarrow string\ set$
$fvs'\ :: 'a\ Exp\ list \Rightarrow string\ set$
$fvAlts\ :: ('a\ Patron \times 'a\ Exp)\ list \Rightarrow string\ set$
$fvAlts'\ :: ('a\ Patron \times 'a\ Exp)\ list \Rightarrow string\ set$
$fvTup\ :: 'a\ Patron \times 'a\ Exp \Rightarrow string\ set$
$fvTup'\ :: 'a\ Patron \times 'a\ Exp \Rightarrow string\ set$

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

$fvs\ []\ = \{\}$
$fvs\ (exp\#exps) = fv\ exp \cup fvs\ exps$

$fvs'\ []\ = \{\}$
$fvs'\ (exp\#exps) = fv\ exp \cup fvs'\ exps$

$fvAlts\ []\ = \{\}$
$fvAlts\ (alt\#alts) = fvTup\ alt \cup fvAlts\ alts$

$fvAlts'\ []\ = \{\}$
$fvAlts'\ (alt\#alts) = fvTup'\ alt \cup fvAlts'\ alts$

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

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

consts *fvReg* :: *'a Exp ⇒ string set*
fvsReg :: *'a Exp list ⇒ string set*
fvsReg' :: *'a Exp list ⇒ string set*
fvAltsReg :: *('a Patron × 'a Exp) list ⇒ string set*
fvAltsReg' :: *('a Patron × 'a Exp) list ⇒ string set*
fvTupReg :: *'a Patron × 'a Exp ⇒ string set*
fvTupReg' :: *'a Patron × 'a Exp ⇒ 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 ∪ fvReg e2
fvReg (CaseE exp patexps a) = fvAltsReg patexps
fvReg (CaseDE exp patexps a) = fvAltsReg' patexps

fvAltsReg [] = {}
fvAltsReg (alt#alts) = fvTupReg alt ∪ fvAltsReg alts

fvAltsReg' [] = {}
fvAltsReg' (alt#alts) = fvTupReg' alt ∪ fvAltsReg' alts

fvTupReg (pat,e) = fvReg e − varProgPat pat

fvTupReg' (pat,e) = fvReg e − varProgPat pat

consts *boundVar* :: *'a Exp ⇒ string set*
boundVars :: *'a Exp list ⇒ string set*
boundVars' :: *'a Exp list ⇒ string set*
boundVarAlts :: *('a Patron × 'a Exp) list ⇒ string set*
boundVarAlts' :: *('a Patron × 'a Exp) list ⇒ string set*
boundVarTup :: *'a Patron × 'a Exp ⇒ string set*
boundVarTup' :: *'a Patron × 'a Exp ⇒ string set*

primrec
boundVar (ConstE Lit a) = {}
boundVar (ConstrE C exps rv a) = boundVars exps
boundVar (VarE x a) = {}
boundVar (CopyE x rv a) = {}
boundVar (ReuseE x a) = {}
boundVar (AppE fn exps rvs a) = {}
boundVar (LetE x1 e1 e2 a) = {x1}
boundVar (CaseE exp patexps a) = boundVarAlts patexps
boundVar (CaseDE exp patexps a) = boundVarAlts' patexps

$boundVars\ [] = \{\}$
$boundVars\ (exp\#exps) = boundVar\ exp \cup boundVars\ exps$

$boundVarAlts\ [] = \{\}$
$boundVarAlts\ (alt\#alts) = boundVarTup\ alt \cup boundVarAlts\ alts$

$boundVarAlts'\ [] = \{\}$
$boundVarAlts'\ (alt\#alts) = boundVarTup'\ alt \cup boundVarAlts'\ alts$

$boundVarTup\ (pat,e) = set\ (extractVar\ pat)$

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

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
$Environment = (ProgVar \rightharpoonup Val) \times (RegVar \rightharpoonup Region)$

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 Σ 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.

constdefs $recursiveArgs :: Constructor \Rightarrow bool\ list$
$recursiveArgs\ C \equiv\ (let$
$\quad (\text{-},\text{-},args) = the\ (ConstructorTable\ C)$
$\quad in\ map\ (\%a.\ a = Recursive)\ args)$

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

$\quad copy'\ j\ h\ (v,False) \quad = (h,v)$
$|\ copy'\ j\ h\ (Val.Loc\ p,True) = (let$
$\quad (k,C,ps) = the\ (h\ p);$
$\quad bs \quad = recursiveArgs\ C;$
$\quad pbs \quad = zip\ ps\ bs;$
$\quad (h',ps') = mapAccumL\ (copy'\ j)\ h\ pbs;$
$\quad p' \quad = getFresh\ h'$
$\quad in\ (h'(p' \mapsto (j,C,ps')), Val.Loc\ p'))$
$|\ copy'\ j\ h\ (IntT\ i,True) \quad = (h,IntT\ i)$
$|\ copy'\ j\ h\ (BoolT\ b,True) \quad = (h,BoolT\ b)$
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*

consts
Σ :: *FunName* ⇀ (*ProgVar list* × *RegVar list* × *′a Exp*)

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*)

Lemmas for extend function

lemma *extend-monotone*: *x* ∉ *set xs* ⟹ *E x* = *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* ∉ *set xs*,*simp*)
apply (*subgoal-tac extend E* (*xa* # *xs*) (*y* # *ys*) = (*extend E xs ys*)(*xa* ↦*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* ==> *P* (*x*#*xs*) (*Suc i*) |]
==> *P xs i*
by (*induct xs arbitrary*: *i*) (*case-tac x*, *auto*)+

lemma *extend-monotone-i*:

$i < length\ alts \longrightarrow$
$length\ alts > 0 \longrightarrow$
$x \notin set\ (snd\ (extractP\ (fst\ (alts\ !\ i)))) \longrightarrow$
$E\ x = extend\ E\ (snd\ (extractP\ (fst\ (alts\ !\ i))))\ vs\ x$
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*:
$[\![\ z \in dom\ (extend\ E\ xs\ vs);\ z \notin set\ xs;\ length\ xs = length\ vs\ ]\!] \Longrightarrow z \in dom\ E$
apply (*simp add*: *extend-def*)
apply (*erule disjE*)
apply (*simp add*: *dom-def*)
by *simp*

end

4 Primitive operators for SVM anf JVM

theory *BinOP*
imports *Main*
begin

Primitive operators

datatype *PrimOp* = *Add* | *Substract* | *Times* | *Divide* | *LessThan* | *LessEqual*
| *Equal* | *GreaterThan* | *GreaterEqual* | *NotEqual*

end

5 State of the SVM

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

5.1 Sizes Tables with next static information: (1) Number of cells, (2) Length of Region Stack and (3) maximum length of Stack S

types *ncell* = *nat*
sizeRegions = *nat*
sizeStackS = *nat*

types *SizesTables* = *ncell* × *sizeRegions* × *sizeStackS*

5.2 Stack

types
CodeLabel = *nat*
Continuation = *Region* × *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*

5.3 Code Store

— 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 firts 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 × CodeSequence × FunName) list
ContinuationMap = (FunName × CodeLabel × CodeLabel list) list
CodeStore = SVMCode × ContinuationMap
SafeImpProg = CodeStore × CodeLabel × ConstructorTableType × SizesTables

5.4 Runtime State

types
PC = CodeLabel × nat
SVMState = Heap × Region × PC × Stack

consts
incrPC :: PC => PC

primrec
incrPC (l,i) = (l,i+1)

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

constdefs
primops :: string ⇀ PrimOp

primops ≡ map-of [(''+'',*Add*),
(''−'',*Substract*),
(''∗'',*Times*),
(''%'',*Divide*),
(''<'',*LessThan*),
(''<='',*LessEqual*),
(''=='',*Equal*),
(''>'',*GreaterThan*),
(''>='',*GreaterEqual*)
]

— Define primitive operations
consts
execOp :: $[PrimOp, Val, Val] => Val$

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 Substract 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

6 Resource-Aware Operational semantics of Safe expressions

theory *SafeRASemantics*
imports *SafeExpr ../SafeImp/SVMState*
begin

types
Delta = (*Region* ⇀ *int*)
MinimumFreshCells = *int*
MinimumWords = *int*
Resources = *Delta* × *MinimumFreshCells* × *MinimumWords*

constdefs *sizeVal* :: [*HeapMap*, *Val*] ⇒ *int*
sizeVal h v ≡ (*case v of* (*Loc p*) ⇒ *int p*
| - ⇒ *0*)

constdefs *size* :: [*HeapMap*, *Location*] ⇒ *int* **where**
size h p ≡ (*case h p of*
Some (*j*,*C*,*vs*) ⇒ (*let rp* = *getRecursiveValuesCell* (*C*,*vs*)
in 1+ (∑ *i*∈*rp*. *sizeVal h* (*vs*!*i*))))

constdefs *balanceCells* :: *Delta* ⇒ *int* (‖ - ‖ [*71*] *70*)
balanceCells δ ≡ (∑ *n*∈*ran* δ. *n*)

constdefs *addDelta* :: *Delta* ⇒ *Delta* ⇒ *Delta* (**infix** ⊕ *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 $emptyDelta :: nat \Rightarrow nat \rightharpoonup int\ \ (\ []\text{-}\ [71]\ 70)$
$[]_k \equiv (\%i.\ if\ i \in \{0..k\}$
$then\ Some\ 0$
$else\ None)$

inductive
$SafeRASem :: [Environment, HeapMap, nat, nat, 'a\ Exp, HeapMap, nat, Val, Resources] \Rightarrow bool$
$(\ \text{-} \vdash \text{-}, \text{-}, \text{-}, \text{-} \Downarrow \text{-}, \text{-}, \text{-}, \text{-}\ [71,71,71,71,71,71,71,71]\ 70)$
where

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

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

| $var1\ \ :\ E1\ x = Some\ (Val.Loc\ p)$
$\Longrightarrow (E1,E2) \vdash h, k, td, (VarE\ x\ a) \Downarrow h, k, Val.Loc\ p,([]_k,0,1)$

| $var2\ \ :\ [\![E1\ x = Some\ (Val.Loc\ p);\ E2\ r = Some\ j;\ \ j <= k;\ m > 0;$
$SafeHeap.copy\ (h,k)\ j\ p = ((h',k),p');\ m = size\ h\ p]\!]$
$\Longrightarrow\ (E1,E2) \vdash h, k, td, (x\ @\ r\ a) \Downarrow h', k, Val.Loc\ p',([j \mapsto m],m,2)$

| $var3\ \ :\ [\![E1\ x = Some\ (Val.Loc\ p);\ h\ p = Some\ c;\ \ SafeHeap.fresh\ q\ h\ ]\!]$
$\Longrightarrow (E1,E2) \vdash h, k, td, (ReuseE\ x\ a) \Downarrow ((h(p{:=}None))(q \mapsto c)),k$
$,Val.Loc\ q,([]_k,0,1)$

| $let1\ \ :\ [\![\ \forall\ C\ as\ r\ a'.\ e1 \neq ConstrE\ C\ as\ r\ a';\ x1 \notin dom\ E1;$
$\forall\ h'\ v1.\ (E1,E2) \vdash h,k,0,e1 \Downarrow h',k,v1,(\delta 1,m1,s1) \wedge$
$(E1(x1 \mapsto v1),E2) \vdash h',k,(td+1),e2 \Downarrow h'',k,v2,(\delta 2,m2,s2)\ ]\!]$
$\Longrightarrow (E1,E2) \vdash h,k,\ td,\ \ Let\ x1 = e1\ In\ e2\ a \Downarrow h'',k,v2,(\delta 1 \oplus \delta 2,max\ m1\ (m2$
$+ \|\delta 1\|),max\ (s1+2)\ (s2+1))$

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

| *case1*: ⟦(∀ *p E1′ xs vs j C*.
i < *length alts*
∧ *E1 x* = *Some* (*Val.Loc p*)
∧ *h p* = *Some* (*j*,*C*,*vs*)
∧ *alts*!*i* = (*pati*, *ei*)
∧ *pati* = *ConstrP C ps ms*
∧ *xs* = *map pat2var ps*
∧ *E1′* = *extend E1 xs vs*
∧ *def-extend E1 xs vs*
∧ *nr* = *int* (*length vs*)
∧ (*E1′*,*E2*) ⊢ *h*,*k*, *nat* ((*int td*)+ *nr*), *ei* ⇓ *h′*,*k*,*v*,(δ,*m*,*s*))⟧
⟹ (*E1*,*E2*) ⊢ *h*,*k*, *td*, *Case* (*VarE x a*) *Of alts a′* ⇓ *h′*, *k*,*v*,(δ,*m*, (*s*+*nr*))

| *case1-1*: ⟦(∀ *n*.
i < *length alts*
∧ *E1 x* = *Some* (*IntT n*)
∧ *alts*!*i* = (*pati*, *ei*)
∧ *pati* = *ConstP* (*LitN n*)
∧ (*E1*,*E2*) ⊢ *h*,*k*, *td*, *ei* ⇓ *h′*,*k*,*v*,(δ,*m*,*s*))⟧
⟹ (*E1*,*E2*) ⊢ *h*,*k*, *td*, *Case* (*VarE x a*) *Of alts a′* ⇓ *h′*, *k*,*v*,(δ,*m*,*s*)

| *case1-2*: ⟦(∀ *b*.
i < *length alts*
∧ *E1 x* = *Some* (*BoolT b*)
∧ *alts*!*i* = (*pati*, *ei*)
∧ *pati* = *ConstP* (*LitB b*)
∧ (*E1*,*E2*) ⊢ *h*,*k*, *td*, *ei* ⇓ *h′*,*k*,*v*,(δ,*m*,*s*))⟧
⟹ (*E1*,*E2*) ⊢ *h*,*k*, *td*, *Case* (*VarE x a*) *Of alts a′* ⇓ *h′*, *k*,*v*,(δ,*m*,*s*)

| *case2*: ⟦(∀ *p E1′ xs vs j C*.
i < *length alts*
∧ *E1 x* = *Some* (*Val.Loc p*)
∧ *h p* = *Some* (*j*,*C*,*vs*)
∧ *alts*!*i* = (*pati*, *ei*)
∧ *pati* = *ConstrP C ps ms*
∧ *xs* = *map pat2var ps*
∧ *E1′* = *extend E1 xs vs*
∧ *def-extend E1 xs vs*
∧ *nr* = *int* (*length vs*)
∧ *j* <= *k*
∧ (*E1′*,*E2*) ⊢ *h*(*p* :=*None*),*k*, *nat* ((*int td*)+ *nr*), *ei* ⇓ *h′*,*k*,*v*,(δ,*m*,*s*))⟧

⟹ (*E1*,*E2*) ⊢ *h*,*k*, *td*, *CaseD* (*VarE x a*) *Of alts a′* ⇓ *h′*, *k*,*v*,(δ⊕(*empty*(*j*↦−*1*)),*max* *0* (*m* − *1*),*s*+*nr*)

| *app-primops*: ⟦ *primops f* = *Some oper*;
v1 = *atom2val E1 a1*;

$v2 = atom2val\ E1\ a2$;
$v = execOp\ oper\ v1\ v2$⟧
⟹ $(E1,E2) \vdash h,k,\ td,\ AppE\ f\ [a1,a2]\ []\ a \Downarrow\ h,\ k,v,([]_k,0,2)$
| *app*: ⟦ $\Sigma\ f = Some\ (xs,rs,e)$; $primops\ f = None$;
$E1' = map\text{-}of\ (zip\ xs\ (map\ (atom2val\ E1)\ as))$;
$n = length\ xs$;
$l = length\ rs$;
$E2' = (map\text{-}of\ (zip\ rs\ \ (map\ (the \circ E2)\ rr)))\ (self \mapsto Suc\ k)$;
$(E1',E2')\ \vdash h,\ (Suc\ k),\ (n+l),\ e \Downarrow h',\ (Suc\ k),v,(\delta,m,s)$⟧
⟹ $(E1,E2) \vdash\ h,k,\ td,\ AppE\ f\ as\ rr\ a \Downarrow\ h' \mid `\ \{p.\ p \in dom\ h'\ \&\ fst\ (the\ (h'\ p)) \leq k\},\ k,v,$
$(\delta(k+1:=None),m,max\ ((int\ n)+(int\ l))\ (s+\ (int\ n)+\ (int\ l)\ -\ int\ td))$

end

7 Semantics of the SVM instructions

theory *SVMSemantics*
imports *SVMState HaskellLib*
begin

- 'execSVMInst' executes a single SafeImp instruction in a state and gives another state.
- 'execSVM' executes the next SafeImp instruction in a SVM program.
- 'execOp' operates two basic values with an operator.

consts
execSVMInst :: [*SafeInstr*,*ConstructorTableFun*,*Heap*,*Region*,*PC*,*Stack*]
=> (*SVMState*,*SVMState*) *Either*

consts
item2Stack :: *Region* => *Stack* => *Item* => *StackObject*
item2Val :: *Stack* => *Item* => *Val*

Auxiliary functions for BUILDENV and BUILDCLS

fun *stackOffset* :: *Stack* ⇒ *nat* ⇒ *StackObject* (**infix** !+ *110*) **where**
(*so* # *S*) !+ *0* = *so*
| (*Val v* # *S*) !+ *n* = *S* !+ (*n* − *1*)
| (*Reg r* # *S*) !+ *n* = *S* !+ (*n* − *1*)
| (*Cont c* # *S*) !+ *n* = *S* !+ (*n* − *2*)

primrec
item2Stack k S (*ItemConst v*) = *Val v*
item2Stack k S (*ItemVar off*) = *S* !+*off*
item2Stack k S ItemRegSelf = *Reg k*

primrec
item2Val S (ItemConst v) = v
item2Val S (ItemVar off) = (case S !+ off of Val v => v)

— We use fun here for full pattern matching
constdefs
execSVM :: SafeImpProg ⇒ SVMState ⇒ (SVMState,SVMState) Either
execSVM prog state == (
case prog of ((code,cnt),q,ct,st) =>
case state of (h,k0,(l,i),S) =>
let codef = map-of code
in execSVMInst (fst(the (codef l))!i) (map-of ct) h k0 (l,i) S)

consts
execSVM-N :: SafeImpProg ⇒ SVMState ⇒ nat ⇒ SVMState

primrec
execSVM-N prog s 0 = s
execSVM-N prog s (Suc n) = (case execSVM prog s of
Right s' => execSVM-N prog s' n)

Function *execSVMInst* executes a single SVM instruction. If it is POPCONT and there is no continuation in the stack, it returns the current state with the constructor Left. Otherwise, it returns the next state with the constructor Right.

primrec
execSVMInst DECREGION ct h k0 pc S = Right (h ↓ k0,k0,incrPC pc,S)

execSVMInst POPCONT ct h k0 pc S = (case S of
v#[] => Left (h,k0,pc,S)
| v#Cont (k0,l)#S' => Right (h,k0,(l,0),v#S'))

execSVMInst (PUSHCONT p)ct h k0 pc S = Right (h,snd h,incrPC pc,Cont (k0,p)#S)

execSVMInst COPY ct h k0 pc S = (case S of
Val (Loc b)#Reg j#S' =>
if j ≤ snd h then
let pair = copy h j b in
case pair of (h',b') =>
Right (h',k0,incrPC pc,Val (Loc b')#S')
else arbitrary)

execSVMInst REUSE ct h k0 pc S = (case S of
Val (Loc b)#S' =>
case h of (hm,k) =>

```
                    let cell = the (hm b);
                        b'   = getFresh hm in
                    Right ((hm(b:=None)(b'↦ cell),k),
                           k0,incrPC pc,Val (Loc b')#S'))

execSVMInst (CALL p)       ct h k0 pc S = (case h of
                   (hm,k) => Right ((hm,k+1),k0,(p,0),S))

execSVMInst (PRIMOP oper) ct h k0 pc S = (case S of
                   Val v1# Val v2#S' =>
                     let v = execOp oper v1 v2 in
                     Right (h,k0,incrPC pc,Val v#S'))


execSVMInst (MATCH l ps)  ct h k0 pc S = (case S!+l of Val (Loc b) =>
                   case (fst h) b of
                     Some (j,C,vs) =>
                       case ct C of
                         Some (-,r,-) =>
                           Right (h,k0,(ps!r,0),(map Val vs) @ S))

execSVMInst (MATCHD l ps) ct h k0 pc S = (case S!+l of Val (Loc b) =>
                   case (fst h) b of
                     Some (j,C,vs) =>
                       case ct C of
                         Some (-,r,-) =>
                           let h' = ((fst h)(b:= None), snd h) in
                           Right (h',k0,(ps!r,0),(map Val vs) @ S))


execSVMInst (MATCHN l v m ps) ct h k0 pc S = (case S!+l of
                 Val (IntT i) =>
                   let r = (nat i)- v;
                       p = if r ≥ 0 ∧ r ≤ m - 1
                           then ps!r
                           else ps!m
                   in Right (h,k0,(p,0),S)
               | Val (BoolT b) =>
                   let p = if ¬ b then ps!0 else ps!1
                   in Right (h,k0,(p,0),S))

execSVMInst (BUILDENV is)  ct h k0 pc S =
                   (let bs = map (item2Stack (snd h) S) is
                    in Right (h,k0,incrPC pc,bs @ S))

execSVMInst (BUILDCLS C is i) ct h k0 pc S =
                   (case item2Stack (snd h) S i of
                     Reg j => case h of
                      (hm,k) =>
```

```
    let vs = map (item2Val S) is;
        b  = getFresh hm;
        h' = (hm (b ↦ (j,C,vs)),k)
    in Right (h',k0,incrPC pc,Val (Loc b)#S))
```

execSVMInst (SLIDE m n) ct h k0 pc S = Right (h,k0,incrPC pc,take m S @ drop (m+n) S)

We convert function *execSVM* into a reflexive transitive relation. Also, we define an inductive relation *execSVMBalanced* giving us the list of all the states reachable from an initial one so that the stack does not decrease more than a given amount n. The list is given in reverse order.

constdefs
execSVMAll :: [*SafeImpProg*,*SVMState*,*SVMState*] => *bool*
(-⊢ - −*svm*→ - [*61*,*61*,*61*]*60*)
P ⊢ *s* −*svm*→ *s'* ≡ (*s*,*s'*) ∈ {(*s*,*t*) . *execSVM P s = Right t*} ^*

fun
diffStack :: *SVMState* => *SVMState* => *nat* => *int*
where
diffStack (*h'*,*k0'*,*pc'*,*S'*) (*h*,*k0*,*pc*,*S*) *m* = (
let n = length S;
n' = length S'
in int m + (*int n'* − *int n*))

fun *instrSVM* :: *SafeImpProg* => *SVMState* => *SafeInstr*
where
instrSVM ((*code*,*cnt*),*q*,*ct*,*st*) (*h*,*k0*,(*l*,*i*),*S*) = *fst*(*the* (*map-of code l*))!*i*

inductive
execSVMBalanced :: [*SafeImpProg*,*SVMState*,*nat list*,*SVMState list*,*nat list*] => *bool*
(-⊢- , - −*svm*→ - , - [*61*,*61*,*61*,*61*,*61*]*60*)
where
init: *P* ⊢ *s*, *n*#*ns* −*svm*→ [*s*], *n*#*ns*
| *step*: ⟦ *P* ⊢ *s*, *n*#*ns* −*svm*→ *s'*#*ss*, *m*#*ms*;
execSVM P s' = Right s'';
m' = nat (*diffStack s'' s' m*);
m' ≥ *0*;
ms' = (if pushcont (*instrSVM P s'*) *then 0*#*m*#*ms*
else if popcont (*instrSVM P s'*) ∧ *ms=m''*#*ms''* *then* (*Suc m''*)#*ms''*
else m'#*ms*)⟧
⟹
P ⊢ *s*, *n*#*ns* −*svm*→ *s''*#*s'*#*ss*, *ms'*

lemma *popcont-neq-pushcont*:
popcont (*instrSVM P s*) ⟹ ¬*pushcont* (*instrSVM P s*)

by (*case-tac instrSVM P s*,*simp-all*)

lemma *execSVMBalanced-n-step-aux* [*rule-format*]:
$P \vdash s'', n''\#ns'' -svm\rightarrow sss\ , n'''\#ns''' \longrightarrow (\forall\ \ s'''\ ss''\ ss.\ sss = s'''\#ss'' \longrightarrow$

$P \vdash s', n'\#ns' -svm\rightarrow s''\ \#\ ss', n''\#ns'' \longrightarrow$
$ss = ss'' @ ss' \longrightarrow$
$P \vdash s', n'\#ns' -svm\rightarrow s'''\#\ ss, n'''\#ns''')$
apply (*rule impI*)

apply (*erule execSVMBalanced.induct*) **apply** *simp*
apply (*rule allI*)+
apply (*erule-tac x=s'a* **in** *allE*)
apply (*erule-tac x=ss* **in** *allE*)
apply (*erule-tac x=ss @ ss'* **in** *allE*)
apply (*rule impI*)+
apply *simp*

apply (*elim conjE*)

apply (*rule conjI*) **apply** (*rule impI*) **apply** (*rule conjI*) **apply** (*rule impI*)
apply *simp*

apply (*erule conjE*) **apply** (*frule popcont-neq-pushcont*) **apply** *simp*

apply (*rule impI*)
apply *simp*
apply (*drule-tac t=ss''* **in** *sym*) **apply** *simp*
apply (*rule execSVMBalanced.step*) **apply** *simp* **apply** *simp* **apply** *simp* **apply** *simp* **apply** *simp*
apply (*rule conjI*) **apply** *simp* **apply** *simp*

apply (*rule impI*) **apply** (*rule conjI*) **apply** (*rule impI*)
apply *simp*
apply (*drule-tac t=ss''* **in** *sym*) **apply** *simp*
apply (*rule execSVMBalanced.step*) **apply** *simp* **apply** *simp* **apply** *simp* **apply** *simp* **apply** *simp*

apply (*rule impI*)
apply *simp*
apply (*drule-tac t=ss''* **in** *sym*) **apply** *simp*
apply (*rule execSVMBalanced.step*) **apply** *simp* **apply** *simp* **apply** *simp* **apply** *simp*
apply (*split split-if-asm*) **apply** *simp* **apply** *simp*
apply (*rule impI*) **apply** (*elim conjE*)
apply (*rule conjI*) **apply** *simp* **apply** *auto*
done

lemma *execSVMBalanced-n-step*:
⟦$P \vdash s'$, $n'\#ns' -svm\rightarrow s''\ \#\ ss'$, $n''\#ns''$;
$P \vdash s''$, $n''\#ns'' -svm\rightarrow s'''\#\ ss''$, $n'''\#ns'''$;
$ss = ss'' @ ss'$⟧
⟹ $P \vdash s'$, $n'\#ns' -svm\rightarrow s'''\#\ ss$, $n'''\#ns'''$
apply (*rule execSVMBalanced-n-step-aux*)
apply *simp-all*
done

end

8 Translation from a CoreSafe program to a SafeImp program

theory *CoreSafeToSVM*
imports *SVMState ../CoreSafe/SafeExpr HaskellLib*
begin

FunMap keeps the correspondence between a function name and the label of its initial instruction sequence

types
FunMap = *FunName* ⇀ *CodeLabel*
TagMap = *Constructor* ⇀ *nat*

8.1 Compile-time environment

This part is devoted to compile-time environments. See the WFLP'08 paper.

types
MiniEnv = *string* ⇀ *nat*
Env = (*MiniEnv* × *nat* × *nat*) *list*

consts
envSearch' :: *Env* => *string* => *nat* => *nat*

fun *envPlusPlus* :: *Env* => *Env* (- ++ *120*) **where**
rho++ = (*case rho of*
(*delta*,*m*,*n*)#*rest* => (*empty*,*0*,*0*)#(*delta*,*m*+*2*,*2*)#*rest*)

fun *envAdd* :: *Env* => (*string* × *nat*) *list* => *Env* (**infix** + *110*) **where**
rho + *namNats* = (*case rho of* (*delta*,*m*,*n1*)#*rest* =>
let n2 = *length namNats*;
g = λ (*x*,*j*) . (*x*,*m*+*j*);
delta' = *map-of* (*map g namNats*)

```
                in  (delta ++ delta',m+n2,0)#rest)

fun envSearch :: Env => string => nat    (infix |> 110 ) where
  rho |> x = envSearch' rho x 0

primrec
  envSearch' (tup # rest) x l = (case tup of
              (delta,m,n) =>
                case delta x of
                   Some j => (l + m) - j
                 | None   => envSearch' rest x (l+m))

fun topDepth :: Env => nat where
  topDepth ((delta,m,0)#-) = m


fun decreasing :: nat => nat list where
  decreasing n = (if n = 0 then []
                           else n # decreasing (n - 1))
```

8.2 Translation functions

These are the main translation functions:

- *trProg* translates a complete CoreSafe program.
- *trF* translates a CoreSafe function definition. It receives a not used CodeLabel and returns another one not used. All the CodeLabels in between are used to translate this function. Also, it returns modified FunMap and ContinuationMap, including a bindings Name -¿ CodeLabel and Name -¿ CodeLabel list for this function.
- *trE* translates a CoreSafe expression. The CodeLabel returned is a non used one, the CodeLabel list contains the labels corresponding to continuations, the CodeSequence contains the instructions of the translation and the SVMCode the auxiliary code sequences created. *trAlts* and *trTup* are auxiliary functions needed in order to define *trE* as primitive recursive.

```
consts
  trE    :: [CodeLabel,FunMap,FunName,Env,'a Exp]=>
              CodeLabel× CodeLabel list × CodeSequence × SVMCode
  trAlts :: [CodeLabel,FunMap,FunName,Env,
              ('a Patron × 'a Exp) list] =>
              CodeLabel × (CodeLabel list × CodeLabel × SVMCode) list
  trAlts' :: [CodeLabel,FunMap,FunName,Env,
              ('a Patron × 'a Exp) list] =>
              CodeLabel × (CodeLabel list × CodeLabel × SVMCode) list
```

```
  trTup  :: [CodeLabel,FunMap,FunName,Env,
             'a Patron × 'a Exp] =>
             CodeLabel list × CodeLabel × SVMCode
  trTup' :: [CodeLabel,FunMap,FunName,Env,
             'a Patron × 'a Exp] =>
             CodeLabel list × CodeLabel × SVMCode

fun  lit2val :: Lit => Val where
  lit2val (LitN i) = IntT i
| lit2val (LitB b) = BoolT b


fun  atom2item :: Env => 'a Exp => Item where
  atom2item rho (ConstE c -) = ItemConst (lit2val c)
| atom2item rho (VarE x -)   = ItemVar (rho |> x)



fun  region2item :: Env => RegVar => Item where
  region2item rho r = (if r = self then ItemRegSelf
                                   else ItemVar (rho |> r))

constdefs  mainName :: string
  mainName ≡ ''PSafeMain''


fun trF    :: (CodeLabel × FunMap × ContinuationMap) => 'a Def =>
              (CodeLabel × FunMap × ContinuationMap) × SVMCode
where
  trF (p,funm,contm) (t,izq,Simple e []) =
          (let
             (f,pbs,rs)      = izq;
             (ps,bs)         = unzip pbs;
             xs              = map pat2var ps;
             topdepth        = length xs + length rs;
             varnats         = zip (append xs rs) (decreasing topdepth);
             rho             = [(empty,0,0)] + varnats;
             funm'           = funm (f↦ p);
             (q,ls,is,cs)    = trE (p+1) funm' f rho e
           in ((q, funm',(f,p,ls)#contm), append cs [(p,is,f)]))

constdefs
  type2argType :: ExpTipo => ArgType
  type2argType t == (case t of
                       Rec                => Recursive
                     | ConstrT name ts b rs => if name = intType then IntArg
                                          else if name = boolType then BoolArg
                                               else NonRecursive
                     | VarT a             => IntArg)
```

```
fun trAltData :: TagMap => ConstructorTableType => AltData => Construc-
torTableType
where
  trAltData tm ct (ConstrA C ts -) = (
          let nargs    = length ts;
              argtypes = map type2argType ts
          in (C, (nargs,the (tm C),argtypes))# ct)

fun extractCons :: AltData => Constructor
where
  extractCons (ConstrA C - -) = C

fun trData :: ConstructorTableType => DecData => ConstructorTableType
where
  trData ct (T,as,rs,alts) = (
          let cs     = map extractCons alts;
              n      = length cs;
              tagmap = map-of (zip (sort cs) [0..<n])
          in  foldl (trAltData tagmap) ct alts)

fun trProg :: 'a Prog => SafeImpProg
where
  trProg (datas,defs,e) = (
          let
           ctable                = foldl trData [] datas;
           ((p,funm,contm),codes) = mapAccumL trF (1,empty,[]) defs;
           (q,ls,is,cs)          = trE p funm mainName [(empty,0,0)] e;
           code                  = concat [concat codes, cs, [(q,is,mainName)]];
           contmap                 = (mainName,q,ls)#contm
          in ((code,contmap),q,ctable,(0,0,0))      )

fun normalForm :: Env => CodeSequence where
  normalForm rho = (let td = (topDepth rho)
                    in  if td ≠ 0 then  [SLIDE 1 td,DECREGION,POPCONT]
                                  else  [DECREGION,POPCONT])
```

The translation of a expression does a recursive traversal of the abstract syntax tree.

```
primrec

  trE p funm fname rho (ConstE c a) =
            (p,[],BUILDENV [ItemConst (lit2val c)] # normalForm rho,[])

  trE p funm fname rho (VarE x a)   =
            (p,[],BUILDENV [ItemVar (rho |> x)] # normalForm rho,[])

  trE p funm fname rho (CopyE x r a) =
            (p,[],BUILDENV [ItemVar (rho |> x),ItemVar (rho |> r)]
```

```
          # COPY # normalForm rho,[])

  trE p funm fname rho (ReuseE x a)   =
              (p,[],BUILDENV [ItemVar (rho |> x)] # REUSE # normalForm
rho,[])

  trE p funm fname rho (AppE f as rs a) = (
            case primops f of
              Some oper => case as of
               a1#a2#[]  =>
                 let i1 = atom2item rho a1;
                     i2 = atom2item rho a2
                 in (p,[],BUILDENV [i1,i2] # PRIMOP oper # normalForm rho,[])
              | None    =>
               let its1 = map (atom2item rho)   as;
                   its2 = map (region2item rho) rs;
                   is   = [BUILDENV (append its1 its2),
                           SLIDE (length as + length rs) (topDepth rho),
                           CALL (the (funm f))]
               in (p,[],is,[]))


  trE p funm fname rho (Let x1 = e1 In e2 a) = (
        let (q2,ls2,is2,cs2) = trE p funm fname (rho+[(x1,1)]) e2
        in case e1 of
            ConstrE C as r - =>
             let items = map (atom2item rho) as;
                 item  = region2item rho r
             in  (q2,ls2,BUILDCLS C items item # is2,cs2)
            | -          =>
             let (q1,ls1,is1,cs1) = trE (q2+1) funm fname (rho++) e1;
                 cs               = append cs2  ((q2,is2,fname) # cs1)
             in  (q1,q2#(append ls1  ls2),PUSHCONT q2 # is1,cs))


  trE p funm fname rho (Case e Of alts a) = (
            let  (q,rss)      = trAlts p funm fname rho alts;
                 (lss,qs,css) = unzip3 rss
            in case e of
                 VarE x - =>
                  let (pat1,e1) = hd alts
                  in case pat1 of
                    ConstrP - - -   => (q,concat lss,[MATCH (rho |> x) qs],concat
css)
                   | ConstP (LitN i) =>
                      let m = length alts - 1;
                          v = nat i
                      in (q,concat lss,[MATCHN (rho |> x) v m qs],concat css)
                   | ConstP (LitB b) => (q,concat lss,[MATCHN (rho |> x) 0 2
```

```
qs],concat css))

trE p funm fname rho (CaseD e Of alts a) = (
          let  (q,rss)       = trAlts' p funm fname rho alts;
               (lss,qs,css) = unzip3 rss
          in case e of
               VarE x - => (q,concat lss,[MATCHD (rho |> x) qs],concat css))


trAlts  p funm fname rho [] = (p,[])

trAlts  p funm fname rho (alt#alts) = (
          let (ls,q,cs')   = trTup p funm fname rho alt;
              (q',tups)    = trAlts (q+1) funm fname rho alts
          in  (q',(ls,q,cs')#tups))


trAlts' p funm fname rho [] = (p,[])

trAlts' p funm fname rho (alt#alts) = (
          let (ls,q,cs')   = trTup' p funm fname rho alt;
              (q',tups)    = trAlts' (q+1) funm fname rho alts
          in  (q',(ls,q,cs')#tups))


trTup   p funm fname rho (pat,e) = (case pat of
          ConstrP C ps - =>
           let xs           = map pat2var ps;
               rho'         = rho + zip xs (decreasing (length xs));
               (q,ls,is,cs) = trE p funm fname rho' e;
               cs'          = append cs [(q,is,fname)]
           in  (ls,q,cs')
        | ConstP - =>
           let (q,ls,is,cs) = trE p funm fname rho e;
               cs'          = append cs [(q,is,fname)]
           in  (ls,q,cs'))


trTup'  p funm fname rho (pat,e) = (case pat of
          ConstrP C ps - =>
           let xs           = map pat2var ps;
               rho'         = rho + zip xs (decreasing (length xs));
               (q,ls,is,cs) = trE p funm fname rho' e;
               cs'          = append cs [(q,is,fname)]
           in  (ls,q,cs')
        | -             =>
           let  (q,ls,is,cs) = trE p funm fname rho e;
               cs'          = append cs [(q,is,fname)]
           in  (ls,q,cs'))
```

8.3 Auxiliary functions for the correctness theorem

The following functions are useful for stating the correctness theorem of the CoreSafe translation.

— returns the set of function names called from an expression
consts
called :: *'a Exp => FunName set*
calledList :: *('a Patron × 'a Exp) list => FunName set*
calledList' :: *('a Patron × 'a Exp) list => FunName set*
calledTup :: *('a Patron × 'a Exp) => FunName set*
calledTup' :: *('a Patron × 'a Exp) => FunName set*

primrec
called (ConstE Lit a) = {}
called (ConstrE C exps rv a) = {}
called (VarE x a) = {}
called (CopyE x rv a) = {}
called (ReuseE x a) = {}
called (AppE f as rs a) = {*f*}
called (Let x1 = e1 In e2 a) = *called e1* ∪ *called e2*
called (Case e Of alts a) = *called e* ∪ *calledList alts*
called (CaseD e Of alts a) = *called e* ∪ *calledList' alts*

calledList [] = {}
calledList (alt#alts) = *calledTup alt* ∪ *calledList alts*

calledList' [] = {}
calledList' (alt#alts) = *calledTup' alt* ∪ *calledList' alts*

calledTup (p,e) = *called e*
calledTup' (p,e) = *called e*

consts
— returns the set of function names defined in a program
definedFuns :: *('a Def) list => FunName set*

primrec
definedFuns [] = {}
definedFuns (def#defs) = *(case def of (t,(f,pbs,rs),der) =>*
{f} ∪ *definedFuns defs)*

consts
— given a defined name and a program, it returns the function body
extractBody :: *FunName => ('a Def) list => 'a Exp*
primrec
extractBody f (def#defs) = *(case def of (t,(g,pbs,rs),Simple e []) =>*
if f = g then e else extractBody f defs)

— It inductively defines the set of function names reached from an expression

inductive-set
closureCalled :: *'a Exp* => (*'a Def*) *list* => *FunName set*
for *e* :: *'a Exp* **and** *ds* :: (*'a Def*) *list*
where
init : $f \in called\ e \Longrightarrow f \in closureCalled\ e\ ds$
| *step* : $[\![$
$f \in closureCalled\ e\ ds$;
$f \in definedFuns\ ds$;
$ef = extractBody\ f\ ds$;
$g \in called\ ef$
$]\!] \Longrightarrow$
$g \in closureCalled\ e\ ds$

end

9 Certification of the translation CoreSafe to SVM. Definitions

theory *CertifSafeToSVM-definitions*
imports *Main ../CoreSafe/SafeRASemantics SVMSemantics CoreSafeToSVM*

begin

9.1 Auxiliary functions

fun *domRho* :: *Env* ⇒ *char list set* **where**
domRho [] = {}
| *domRho* ((*delta*,*m*,*n*)#*rest*) = (*dom delta*) ∪ *domRho rest*

fun *identityEnvironment* :: *Environment* ⇒ *Env* ⇒ *Stack* ⇒ *bool* (- ⋈ *'*(-, -*'*) *120*) **where**
(*E1*,*E2*) ⋈ (*ϱ*,*S*) = (((*dom E1* ∪ *dom E2* − {*self*}) = *domRho ϱ*)
∧ (∀ *x* ∈ *dom E1*. *Val* (*the* (*E1 x*)) = *S*!+(*ϱ* |> *x*))
∧ (∀ *r* ∈ *dom E2* − {*self*}. *Reg* (*the* (*E2 r*)) = *S*!+(*ϱ* |> *r*)))

axioms *identityEnvironment-good*:
(*E1*,*E2*) ⋈ (*ϱ*,*S*)
⟹ (∀ *x*∈ *dom E1*. *S*!+(*ϱ* |> *x*) = *Val v*) ∧ (∀ *x*∈ *dom E2*. *S*!+(*ϱ* |> *x*) = *Reg r*)

fun *sizeST* :: *Stack* ⇒ *int* **where**
  sizeST [] = *0*
| *sizeST* (*Val v # S*) = *1* + *sizeST S*
| *sizeST* (*Reg r # S*) = *1* + *sizeST S*
| *sizeST* (*Cont c # S*) = *2* + *sizeST S*

fun *sizeH* :: *Heap* ⇒ *int* **where**
  sizeH (*h*,*k*) = *int* (*card* (*dom h*))

fun *sizeSVMStateST* :: *SVMState* ⇒ *int* **where**
  sizeSVMStateST (*h*,*r*,*pc*,*s*) = *sizeST s*

fun *sizeSVMStateH* :: *SVMState* ⇒ *int* **where**
  sizeSVMStateH (*h*,*r*,*pc*,*s*) = *sizeH h*

fun *foldl1* :: ('*a* ⇒ '*a* ⇒ '*a*) ⇒ '*a list* ⇒ '*a* **where**
foldl1 f (*x*#*xs*) = *foldl f x xs*

fun *maximum* :: *int list* ⇒ *int* **where**
  maximum xs = *foldl1 max xs*

fun *maxFreshCells* :: *SVMState list* ⇒ *int* **where**
  maxFreshCells ((*h0*,*r0*,*pc0*,*s0*)#*sts*) = *maximum* (*map* (%(*h*,*r*,*pc*,*s*). *sizeH h* − (*sizeH h0*)) ((*h0*,*r0*,*pc0*,*s0*)#*sts*))

fun *maxFreshWords* :: *SVMState list* ⇒ *int* **where**
  maxFreshWords ((*h0*,*r0*,*pc0*,*s0*)#*sts*) = *maximum* (*map* (%(*h*,*r*,*pc*,*s*). *sizeST s* − (*sizeST s0*)) ((*h0*,*r0*,*pc0*,*s0*)#*sts*))

constdefs *region* :: *Heap* ⇒ *Location* ⇒ *Region*
  region h l ≡ (*case h of*
                (*hm*,*k*) ⇒
                  case (*hm l*) *of*
                   Some (*r*,(*c*,*xs*)) ⇒ *r*)

constdefs *numCellsRegion* :: *Heap* ⇒ *nat* ⇒ *int*
  numCellsRegion h i ≡ *int* (*card* {*w* ∈ *dom* (*fst h*). *region h w* = *i*})

constdefs *diff* :: *nat* ⇒ *Heap* ⇒ *Heap* ⇒ *nat* ⇀ *int*
  diff k h h' ≡ (%*i*. *if i* ∈ {*0*..*k*}
                  then Some (*numCellsRegion h' i* − *numCellsRegion h i*)
                  else None)

constdefs *extends* :: ('*a* ⇀ '*b*) ⇒ ('*a* ⇀ '*b*) ⇒ *bool*
  extends f g ≡ ((*dom f* ⊆ *dom g*) ∧ (∀ *x* ∈ *dom f*. *f x* = *g x*))

constdefs *extendsL* :: $('a \times 'b)$ *list* $\Rightarrow$ $('a \times 'b)$ *list* $\Rightarrow$ *bool* (**infix** $\sqsubseteq$ *110*)
$xs \sqsubseteq ys == extends\ (map\text{-}of\ xs)\ (map\text{-}of\ ys)$

axioms *A7*:
SafeHeap.copy $(h,k)\ j\ p = ((h',k),p')$
$\Longrightarrow$ *extends h h'* $\wedge$
size h' p' $=$ *size h p* $\wedge$
sizeH (h',k) $-$ *sizeH* (h,k) $=$ *size h' p'* $\wedge$
$(\forall\ k.\ k >= j \longrightarrow diff\ k\ (h,k)\ (h',k) = empty(j \mapsto size\ h'\ p'))$

constdefs *disjointList* :: $('a \times 'b)$ *list* => $('a \times 'b)$ *list* => *bool*
disjointList n m $\equiv$ *dom* $(map\text{-}of\ n) \cap dom\ (map\text{-}of\ m) = \{\}$

fun *rho-good'* :: *Env* => *bool* **where**
rho-good' [] $=$ *True*
| *rho-good'* $((delta,m,n)\#rest) = ((\forall\ x \in dom\ delta.\ m - n >= the\ (delta\ x) \wedge$
$the\ (delta\ x) >= 1) \wedge rho\text{-}good'\ rest)$

fun *rho-good* :: *Env* => *bool* **where**
rho-good $((delta,m,n)\#rest) = (n = 0 \wedge rho\text{-}good'\ ((delta,m,n)\#rest))$

end

10 Certification of the translation CoreSafe to SVM. Basic properties

theory *basic-properties*
imports *Main ../CoreSafe/SafeRASemantics SVMSemantics CertifSafeToSVM-definitions*

begin

This set of axioms reflects basic properties of the Core-Safe expressions. The abstract syntax of Safe is general enough to admit both Full-Safe and Core-Safe programs, but the latter has severe restrictions which are needed in order to prove some of the theorems. These properties are guaranteed by the compiler and cannot be proved. The most important ones are:

- The arguments of function and constructor applications are atoms (i.e. either literals or variables)
- All the bound variables are distinct. Also, 'self' is a reserved identifier denoting the working region of the function at hand.
- The lengths of formal and actual argument lists are equal.

In a future, perhaps the valid Core-Safe expressions could be defined as an inductive set.

```
consts
  core :: 'a Exp => bool

fun atom :: 'a Exp => bool
where
    atom (ConstE c -) = True
|   atom (VarE   x -) = True
|   atom -            = False


axioms all-exp-core: core e

axioms x-not-self:
 core e
  ⟹ (∀ x∈ varProg e. x≠ self)

axioms app-distinct-boundVar:
 Σ f = Some (xs,rs,e)
  ⟹ core e
  ∧ distinct (xs @ rs)
  ∧ self ∉ set xs
  ∧ self ∉ set rs

axioms app-atoms:
 core (AppE f as rr a)
  ⟹ (∀ a ∈ set as. atom a)

axioms app-arguments-good:
 ⟦ core (AppE f as rr a);
  Σ f = Some (xs,rs,e)⟧
  ⟹ length xs = length as ∧ length rs = length rr

axioms let-atoms:
 core (Let x1 = ConstrE C as r a1 In e2 a2)
  ⟹ (∀ a ∈ set as. atom a) ∧ x1 ∉ fvs as
```

This group are obvious theorems of the translation, but tedious to prove. For the moment, we have concentrated on proving the theorems related to the semantics, both at the Core-Safe and at the SVM levels, which are more important for the correctness.

```
axioms codestore-extend:
  ⟦ ((p, funm, contm), codes) = mapAccumL trF (1, empty, []) defs;
      (q,ls,is,cs1) = trE p' fun funame rho e;
      cs = concat codes ⟧
   ⟹ (cs1 @ [(q, is, fname)]) ⊑ cs
```

axioms *codestore-let-disjoint*:
$[\![(q, ls, is, cs1) = trE\ p'\ funm\ fname\ rho\ (Let\ x1 = e1\ In\ e2\ a);$
$(q1, ls1, is1, cs1') = trE\ p1'\ funm\ fname\ rho1\ e1;$
$(q2, ls2, is2, cs2) = trE\ p'\ funm\ fname\ rho2\ e2;$
$cs1 = cs2\ @\ (q2, is2, fname)\ \#\ cs1';$
$drop\ j\ is' = is]\!]$
$\Longrightarrow disjointList\ cs1\ \ [(q, is', fname)]$
$\wedge\ disjointList\ (cs2\ @\ [(q2, is2, fname)])\ \ (cs1'\ @\ [(q1, is', fname)])$

axioms *codestore-let2-disjoint*:
$[\![(q, ls, is1, cs1) = trE\ p'\ funm\ fname\ rho\ e\ ]\!]$
$\Longrightarrow q \notin dom\ (map\text{-}of\ cs1)$

axioms *codestore-case-alts-disjoint*:
$[\![(q, ls, is, cs1) = trE\ p'\ funm\ fname\ rho\ e;$
$e = (Case\ VarE\ x\ a\ Of\ alts\ a') \vee\ \ e = (CaseD\ VarE\ x\ a\ Of\ alts\ a');$
$(q, rss) = trAlts\ p'\ funm\ fname\ rho\ alts;$
$(lss, qs, css) = unzip3\ rss;$
$i < length\ alts;$
$alts!i = (pati, ei);$
$pati = ConstrP\ C\ ps\ ms;$
$xs = map\ pat2var\ ps;$
$(qs\ !\ i, lss\ !i, isi, csi) = trE\ p'\ funm\ fname\ (rho + zip\ xs\ (decreasing\ (length\ xs)))\ ei]\!]$
$\Longrightarrow q \notin dom\ (map\text{-}of\ (concat\ css)) \wedge qs!i \notin dom\ (map\text{-}of\ (concat\ css))$

axioms *codestore-caseL-alts-disjoint*:
$[\![(q, ls, is, cs1) = trE\ p'\ funm\ fname\ rho\ (Case\ VarE\ x\ a\ Of\ alts\ a');$
$(q, rss) = trAlts\ p'\ funm\ fname\ rho\ alts;$
$(lss, qs, css) = unzip3\ rss;$
$i < length\ alts;$
$alts!i = (pati, ei);$
$(qs\ !\ i, lss\ !i, isi, csi) = trE\ p'\ funm\ fname\ rho\ ei]\!]$
$\Longrightarrow q \notin dom\ (map\text{-}of\ (concat\ css)) \wedge qs!i \notin dom\ (map\text{-}of\ (concat\ css))$

axioms *trE-let*:
$[\![(q, ls, is, cs1) = trE\ p'\ funm\ fname\ rho\ (Let\ x1 = e1\ In\ e2\ a);$
$e1 \neq ConstrE\ C\ as\ r\ a]\!]$
$\Longrightarrow$
$\exists\ q1\ ls1\ is1\ cs1'\ q2\ ls2\ is2\ cs2\ .$
$(q1,ls1,is1,cs1') = trE\ (q2+1)\ funm\ fname\ (rho{+}{+})\ e1$
$\wedge\ (q2,ls2,is2,cs2) = trE\ p'\ funm\ fname\ (rho{+}[(x1,1)])\ e2$
$\wedge\ cs1 = cs2\ @\ ((q2,is2,fname)\ \#\ cs1')$
$\wedge\ q{=}q1 \wedge ls = q2\#(ls1\ @\ ls2) \wedge is = PUSHCONT\ q2\ \#\ is1$

axioms *trE-let2*:
$[\![(q, ls, is, cs1) = trE\ p'\ funm\ fname\ rho\ (Let\ x1 = ConstrE\ C\ as\ r\ a'\ In\ e2\ a)]\!]$
$\Longrightarrow$

$\exists$ *q2 ls2 is2 cs2 items item .*
(q2,ls2,is2,cs2) = trE p′ funm fname (rho+[(x1,1)]) e2
$\wedge$ *items = map (atom2item rho) as*
$\wedge$ *item = region2item rho r*
$\wedge$ *q = q2*
$\wedge$ *ls = ls2*
$\wedge$ *is = BUILDCLS C items item # is2*
$\wedge$ *cs1 = cs2*

axioms *trE-case*:
$[\![$ *(q, ls, is, cs1) = trE p′ funm fname rho (Case VarE x a Of alts a′);*
i < length alts; alts!i = (pati, ei); pati = ConstrP C ps ms;
xs = map pat2var ps $]\!]$
$\Longrightarrow$
$\exists$ *rss lss qs css isi csi.*
(q,rss) = trAlts p′ funm fname rho alts
$\wedge$ *(lss,qs,css) = unzip3 rss*
$\wedge$ *ls = concat lss*
$\wedge$ *is = [MATCH (rho |> x) qs]*
$\wedge$ *cs1 = concat css*
$\wedge$ *subList (csi @ [(qs!i,isi,fname)]) cs1*
$\wedge$ *(qs ! i, lss !i, isi, csi) = trE p′ funm fname (rho + zip xs (decreasing (length xs))) ei*

axioms *trE-case-1-1*:
$[\![$ *(q, ls, is, cs1) = trE p′ funm fname rho (Case VarE x a Of alts a′);*
i < length alts; alts!i = (pati, ei); pati = ConstP (LitN n) $]\!]$
$\Longrightarrow$
$\exists$ *rss lss qs css v′ m′ isi csi.*
(q,rss) = trAlts p′ funm fname rho alts
$\wedge$ *(lss,qs,css) = unzip3 rss*
$\wedge$ *ls = concat lss*
$\wedge$ *is = [MATCHN (rho |> x) v′ m′ qs]*
$\wedge$ *cs1 = concat css*
$\wedge$ *subList (csi @ [(qs!i,isi,fname)]) cs1*
$\wedge$ *(qs ! i, lss !i, isi, csi) = trE p′ funm fname rho ei*

axioms *trE-case-1-2*:
$[\![$ *(q, ls, is, cs1) = trE p′ funm fname rho (Case VarE x a Of alts a′);*
i < length alts; alts!i = (pati, ei); pati = ConstP (LitB b) $]\!]$
$\Longrightarrow$
$\exists$ *rss lss qs css isi csi.*
(q,rss) = trAlts p′ funm fname rho alts
$\wedge$ *(lss,qs,css) = unzip3 rss*
$\wedge$ *ls = concat lss*
$\wedge$ *is = [MATCHN (rho |> x) 0 2 qs]*
$\wedge$ *cs1 = concat css*
$\wedge$ *subList (csi @ [(qs!i,isi,fname)]) cs1*
$\wedge$ *(qs ! i, lss !i, isi, csi) = trE p′ funm fname rho ei*

axioms *trE-cased*:
⟦ (*q*, *ls*, *is*, *cs1*) = *trE p′ funm fname rho* (*CaseD VarE x a Of alts a′*);
i < *length alts*; *alts*!*i* = (*pati*, *ei*); *pati* = *ConstrP C ps ms*;
xs = *map pat2var ps* ⟧
⟹
∃ *rss lss qs css isi csi*.
(*q*,*rss*) = *trAlts p′ funm fname rho alts*
∧ (*lss*,*qs*,*css*) = *unzip3 rss*
∧ *ls* = *concat lss*
∧ *is* = [*MATCHD* (*rho* |> *x*) *qs*]
∧ *cs1* = *concat css*
∧ *subList* (*csi* @ [(*qs*!*i*,*isi*,*fname*)]) *cs1*
∧ (*qs* ! *i*, *lss* !*i*, *isi*, *csi*) = *trE p′ funm fname* (*rho* + *zip xs* (*decreasing* (*length xs*))) *ei*

axioms *trE-app*:
⟦ Σ *f* = *Some* (*xs*, *rs*, *e*);
(*q*, *ls*, *is*, *cs1*) = *trE p′ funm fname rho* (*AppE f as rr a*) ⟧
⟹
∃ *qf lsf csf*.
(*qf*, *lsf*, *is*, *csf*) = *trE p′ funm fname* ([(*empty*,0,0)] + (*zip* (*xs* @ *rs*) (*decreasing* (*length xs* + *length rs*)))) *e*
∧ *xs* = *fst* (*the* (Σ *f*))
∧ *rs* = *fst* (*snd* (*the* (Σ *f*)))
∧ *e* = *snd* (*snd* (*the* (Σ *f*)))
∧ *qf* = *the* (*funm f*)

axioms *p-Fresh*:
((*p*,*funm*,*contm*),*cs*) = *mapAccumL trF* (*Suc* 0,*empty*,[]) *defs*
⟹ ∀ *q* ∈ *dom* (*map-of* (*concat cs*)). *q* < *p*

constdefs
goodPC :: *CodeLabel* × *nat* => *SVMCode* => *bool*
goodPC pc cs ≡ (*case pc of* (*p*,*j*) =>
case map-of cs p of Some (*is*,*f*) =>
j < *length is*)

axioms *goodPCtranslation*:
⟦ *P*⊢*s0* , *td*#*tds* −*svm*→ *s* # *ss*, *Suc* 0#*tds*;
P = ((*cs*,*cm*),*p*,*ct*,*sz*);
s0 = ((*h*,*k*),*k0*,*pc*,*S*);
s = ((*h′*,*k′*),*k0′*,*pc′*,*S′*)⟧
⟹
goodPC pc cs ∧ *goodPC pc′ cs*

axioms *rho-good*:
$(q,\ ls,\ is,\ cs1) = trE\ p'\ funm\ fname\ rho\ e$
$\Longrightarrow length\ rho > 0 \wedge rho\text{-}good\ rho$

This group are real axioms in the sense that either the properties must be guaranteed outside these scripts, or they depend on not controlable issues, as it is the case of fresh names generation

axioms *finite-dom-h* [*rule-format*]:
$(E1,E2) \vdash h\ ,\ k\ ,\ td\ ,\ e \Downarrow h'\ ,\ k\ ,\ v\ ,\ r$
$\longrightarrow finite\ (dom\ h)$
$\wedge\ dom\ E1 \cap dom\ E2 = \{\}$
$\wedge\ fvReg\ e \subseteq dom\ E2$
$\wedge\ fv\ e \subseteq dom\ E1$
$\wedge\ self \in dom\ E2$
$\wedge\ boundVar\ e \cap dom\ E2 = \{\}$

axioms *case-constructor-good*:
$[\![(E1',E2) \vdash h,k,\ nat\ ((int\ td)+\ nr),\ ei \Downarrow h',k,v,r;$
$E1' = extend\ E1\ xs\ vs;$
$E1\ x\ = Some\ (Val.Loc\ p);$
$h\ p\ = Some\ (j,C,vs);$
$alts!i = (pati,\ ei);$
$pati = ConstrP\ C\ ps\ ms]\!]$
$\Longrightarrow ct\ C = Some\ (g1,i,g2)$

axioms *cased-constructor-good*:
$[\![(E1',E2) \vdash h(p := None),k,\ nat\ ((int\ td)+\ nr),\ ei \Downarrow h',k,v,r;$
$E1\ x\ = Some\ (Val.Loc\ p);$
$E1' = extend\ E1\ xs\ vs;$
$h\ p\ = Some\ (j,C,vs);$
$alts!i = (pati,\ ei);$
$pati = ConstrP\ C\ ps\ ms]\!]$
$\Longrightarrow ct\ C = Some\ (g1,i,g2)$

axioms *SVM-Semantic-branch-Nat-consistent*:
$[\![\ is =\ [MATCHN\ (rho\ |>\ x)\ v'\ m'\ qs];$
$E1\ x = Some\ (IntT\ n);$
$i < length\ alts\ ]\!]$
$\Longrightarrow (nat\ n - v' >= 0 \wedge nat\ n - v' <= m' - 1 \longrightarrow nat\ n - v' = i) \wedge$
$(\neg\ (nat\ n - v' >= 0 \wedge nat\ n - v' <= m' - 1) \longrightarrow m' = i)$

axioms *SVM-Semantic-branch-Bool-consistent*:
$[\![\ is =\ [MATCHN\ (rho\ |>\ x)\ 0\ 2\ qs];$
$E1\ x = Some\ (BoolT\ b);$
$i < length\ alts\ ]\!]$

$\Longrightarrow (\neg\ b \longrightarrow qs\ !\ 0 = qs\ !\ i) \wedge (b \longrightarrow qs\ !\ Suc\ 0 = qs\ !\ i)$

axioms *getFresh-fresh*:
$SafeHeap.fresh\ q\ h \Longrightarrow getFresh\ h = q$

axioms *finite-dom-h'*:
$[\![(E1,E2) \vdash h\ ,\ k\ ,\ td\ ,\ e \Downarrow h'\ ,\ k\ ,\ v\ ,\ r;\ finite\ (dom\ h)]\!]$
$\Longrightarrow finite\ (dom\ h')$

axioms *app1-1*:
$\forall\ a \in set\ (map\ (item2Stack\ k\ S)\ (map\ (atom2item\ rho)\ as)).\ a = Val\ v$

axioms *app1-2*:
$\forall\ r \in set\ (map\ (item2Stack\ k\ S)\ (map\ (region2item\ rho)\ rr)).\ r = Reg\ r'$

lemma *closureCalled-app*:
$closureCalled\ e\ defs \subseteq closureCalled\ (AppE\ f\ as\ rs'\ a)\ defs$
apply (*rule subsetI*)
apply (*erule closureCalled.induct*)
apply (*rule closureCalled.init*) **apply** *simp* **defer**
apply (*simp*, *rule closureCalled.step*, *assumption+*,*simp*)
oops

axioms *closureCalled-app*:
$closureCalled\ e\ defs \subseteq closureCalled\ (AppE\ f\ as\ rs'\ a)\ defs$

axioms *j-length*:
$j < length\ (fst\ (the\ (map\text{-}of\ (concat\ codes)\ p')))$

axioms *j-length-2*:
$j < length\ is'$

axioms *lengthS-topDepth*:
$E \bowtie (rho,S) \Longrightarrow topDepth\ rho < length\ S$

axioms *resources-gre-0*:
$(E1,\ E2) \vdash h\ ,\ k\ ,\ td\ ,\ e \Downarrow h',\ k\ ,\ v\ ,\ r \Longrightarrow \exists\ \delta\ m\ s.\ r=(\delta\ ,m,s) \wedge m >= 0 \wedge s > 0$

axioms *addEnv-rho-good*:
$addEnv\text{-}good\ rho\ (zip\ xs\ (decreasing\ (length\ xs)))$

axioms *identityEnvironment-good*:
$(E1,E2) \bowtie (\varrho,S)$
$\Longrightarrow (\forall\, x{\in}\ dom\ E1.\ \ S!+(\varrho\ |{>}\ x) = Val\ v) \wedge (\forall\, x{\in}\ dom\ E2.\ \ S!+(\varrho\ |{>}\ x) = Reg\ r) \wedge dom\ E1 \cap dom\ E2 = \{\}$

axioms *continuations-bueno*:
$[\![(E1,\ E2) \bowtie (rho,S);\ td = topDepth\ rho]\!]$
$\Longrightarrow sizeST\ (drop\ td\ S) - sizeST\ S = -\ int\ td$

axioms *let1-3*:
$sizeH\ ((h'\ |`\ \{p \in (dom\ h').\ ((fst\ (the\ (h'\ p))) \leq k)\}),\ k) - (sizeH\ (h,\ k)) =$
$\|\ diff\ k\ (h,k)\ (h'\ |`\ \{p \in dom\ h'.\ fst\ (the\ (h'\ p)) \leq k\},\ k)\ \|$

axioms *xi2*: $S\ !+\ (rho\ |{>}\ (ai)) =$
$append\ (append\ (map\ (item2Stack\ k\ S)\ (map\ (atom2item\ rho)\ as))\ (map\ (item2Stack\ k\ S)\ (map\ (region2item\ rho)\ rr)))$
$(drop\ td\ S)\ !+\ i$

axioms *ri2*: $S\ !+\ (rho\ |{>}\ (rs\ !\ i)) =$
$append\ (append\ (map\ (item2Stack\ k\ S)\ (map\ (atom2item\ rho)\ as))\ (map\ (item2Stack\ k\ S)\ (map\ (region2item\ rho)\ rr)))$
$(drop\ td\ S)\ !+\ \ (length\ xs + i)$

axioms *xi3*: $([(empty,\ 0,\ 0)] + zip\ (xs\ @\ rs)\ (decreasing\ (length\ xs + length\ rs))) |{>}\ (xs!i) = i$

axioms *ri3*: $(length\ xs + i) = ([(empty,\ 0,\ 0)] + zip\ (xs\ @\ rs)\ (decreasing\ (length\ xs + length\ rs))) |{>}\ (rs!i)$

axioms *xiConstE*:
$Val\ (the\ (map\text{-}of\ (zip\ xs\ (map\ (atom2val\ E1)\ as))\ (xs\ !\ i))) =$
$(map\ (item2Stack\ k\ S)\ (map\ (atom2item\ rho)\ as)\ @\ map\ (item2Stack\ k\ S)\ (map\ (region2item\ rho)\ rr)\ @\ drop\ td\ S)\ !+$

(([(*empty*, *0*, *0*)] + *zip* (*xs* @ *rs*) (*decreasing* (*length xs* + *length rs*))) |>
(*xs* ! *i*))

axioms *good-app-region-arguments*:
∀ *r* ∈ *set rr*. *r* ∈ *dom E2* − {*self*}

end

11 Certification of the translation CoreSafe to SVM. execSVMBalanced property

theory *execSVMBalanced-lemmas*
imports *Main ../CoreSafe/SafeRASemantics SVMSemantics CoreSafeToSVM CertifSafeToSVM-definitions basic-properties*
begin

lemma *identityEnvironment-Val*:
⟦ (*E1*,*E2*) ⋈ (*rho*,*S*); *E1 x* = *Some v*; *length rho* > *0*⟧
⟹ *S*!+(*rho* |> *x*) = *Val v*
apply (*unfold identityEnvironment.simps*)
apply (*elim conjE*)
apply (*erule-tac x*=*x* **in** *ballE*)
apply (*case-tac* *S* !+(*rho* |> *x*) = *Val v*,*simp*,*simp*)
by (*simp add*: *dom-def*)

lemma *identityEnvironment-Reg*:
⟦ (*E1*,*E2*) ⋈ (*rho*,*S*); *E2 x* = *Some r*; *length rho* > *0*⟧
⟹ *S*!+(*rho* |> *x*) = *Reg r*
apply (*frule-tac r*=*r* **in** *identityEnvironment-good*, *elim conjE*)
apply (*unfold identityEnvironment.simps*)
apply (*elim conjE*)
apply (*erule-tac x*=*x* **and** *A*=*dom E2* **in** *ballE*)
apply (*case-tac* *S* !+ (*rho* |> *x*) = *Reg r*, *simp*, *simp*)
by (*simp add*: *dom-def*)

lemma *restrictToRegion-monotone*:
(*h'* |` {*p* ∈ *dom h'*. *fst* (*the* (*h'* *p*)) ≤ *k*}, *k*) ↓ *k* = (*h'* |` {*p* ∈ *dom h'*. *fst* (*the* (*h'* *p*)) ≤ *k*}, *k*)
apply (*unfold restrictToRegion.simps*, *unfold Let-def*, *auto*)
apply (*subgoal-tac*

$\{p \in dom\ h'.\ fst\ (the\ (h'\ p)) \leq k \wedge fst\ (the\ ((h'\ |`\ \{p \in dom\ h'.\ fst\ (the\ (h'\ p)) \leq k\})\ p)) \leq k\} =$
$\{p \in dom\ h'.\ fst\ (the\ (h'\ p)) \leq k\})$
apply (*simp*, *auto*)
apply (*subgoal-tac* $x \in dom\ h'$, *clarsimp*)
by (*simp add*: *dom-def*)

declare *envSearch.simps* [*simp del*]
declare *identityEnvironment.simps* [*simp del*]

lemma *execSVMBalanced-IntT*:
$[\![(q, ls, is, cs1) = trE\ p'\ funm\ fname\ rho\ (ConstE\ (LitN\ i)\ a);$
$cs = concat\ codes;$
$(cs1\ @\ [(q, is', fname)]) \sqsubseteq cs;$
$drop\ j\ is' = is;$
$P = ((cs, contm), p, ct, st);$
$td = topDepth\ rho;$
$S' = drop\ td\ S;$
$s0 = ((h, k), k0, (q, j), S);$
$E \bowtie (rho,S)]\!]$
$\Longrightarrow P \vdash s0\ ,\ td\#tds\ -svm\rightarrow ((h, k) \downarrow k0, k0, (q, Suc\ (Suc\ (Suc\ j))), Val\ (IntT\ i)\ \#\ S')\ \#$
$[((h, k), k0, (q, j + 2), Val\ (IntT\ i)\ \#\ drop\ (topDepth\ rho)\ S),$
$((h, k), k0, (q, j + 1), Val\ (IntT\ i)\ \#\ S),$
$((h, k), k0, (q, j), S)]\ ,\ Suc\ 0\#tds \wedge fst\ (the\ (map\text{-}of\ cs\ q))$
$!\ Suc\ (Suc\ (Suc\ j)) = POPCONT$
apply *simp*
apply (*subgoal-tac topDepth rho* $<$ *length S*)
prefer *2* **apply** (*rule lengthS-topDepth*,*assumption*)
apply (*subgoal-tac is'* $=$ *fst* (*the* (*map-of* (*concat codes*) *p'*)))
apply (*subgoal-tac j* $<$ *length* (*fst* (*the* (*map-of* (*concat codes*) *p'*))))
prefer *2* **apply** (*rule j-length*)
apply (*frule-tac i=j* **in** *nth-drop'*,*simp*)
apply (*subgoal-tac* (*Suc j*) $<$ *length* (*fst* (*the* (*map-of* (*concat codes*) *p'*))))
prefer *2* **apply** (*rule-tac j=Suc j* **in** *j-length*)
apply (*frule-tac i=Suc j* **in** *nth-drop'*,*simp*)
apply (*subgoal-tac* (*Suc* (*Suc j*)) $<$ *length* (*fst* (*the* (*map-of* (*concat codes*) *p'*))))

prefer *2* **apply** (*rule-tac j=*(*Suc* (*Suc j*)) **in** *j-length*)
apply (*frule-tac i=*(*Suc* (*Suc j*)) **in** *nth-drop'*,*simp*)
apply (*subgoal-tac* (*Suc* (*Suc* (*Suc j*))) $<$ *length* (*fst* (*the* (*map-of* (*concat codes*) *p'*))))
prefer *2* **apply** (*rule-tac j=*(*Suc* (*Suc* (*Suc j*))) **in** *j-length*)
apply (*frule-tac i=*(*Suc* (*Suc* (*Suc j*))) **in** *nth-drop'*,*simp*)
apply (*elim conjE*)
apply (*simp add*: *Let-def*, *split split-if-asm*, *simp*)

apply (*rule execSVMBalanced.step*, *simp-all*)
apply (*rule execSVMBalanced.step*, *simp-all*)
apply (*rule execSVMBalanced.step*, *simp-all*)
apply (*rule execSVMBalanced.init*)
apply (*unfold execSVM-def*)
apply (*simp add*: *Let-def*, *rule conjI*, *simp*, *simp*)
apply (*simp add*: *Let-def*, *rule conjI*, *simp*, *simp*)
apply (*simp*, *simp add*: *Let-def*)
apply (*rule conjI*, *simp*, *simp*)
apply (*simp add*: *extendsL-def add*: *extends-def*)
by *clarsimp*

lemma *execSVMBalanced-BoolT*:
⟦(*q*, *ls*, *is*, *cs1*) = *trE p′ funm fname rho* (*ConstE* (*LitB b*) *a*);
cs = *concat codes*;
(*cs1* @ [(*q*, *is′*, *fname*)]) ⊑ *cs*;
drop j is′ = *is*;
P = ((*cs*, *contm*), *p*, *ct*, *st*);
td = *topDepth rho*;
S′ = *drop td S*;
s0 = ((*h*, *k*), *k0*, (*q*, *j*), *S*);
E ⋈ (*rho*,*S*)⟧
⟹ *P*⊢*s0* , *td*#*tds* −*svm*→ ((*h*, *k*) ↓ *k0*, *k0*, (*q*, *Suc* (*Suc* (*Suc j*))), *Val* (*BoolT b*) # *S′*) #
[((*h*, *k*), *k0*, (*q*, *j* + *2*), *Val* (*BoolT b*) # *drop* (*topDepth rho*) *S*),
((*h*, *k*), *k0*, (*q*, *j* + *1*), *Val* (*BoolT b*) # *S*),
((*h*, *k*), *k0*, (*q*, *j*), *S*)] , *Suc 0*#*tds* ∧ *fst* (*the* (*map-of cs q*)) ! *Suc* (*Suc* (*Suc j*)) = *POPCONT*
apply *simp*
apply (*subgoal-tac topDepth rho* < *length S*)
prefer *2* **apply** (*rule lengthS-topDepth*,*assumption*)
apply (*subgoal-tac is′* = *fst* (*the* (*map-of* (*concat codes*) *p′*)))
apply (*subgoal-tac j* < *length* (*fst* (*the* (*map-of* (*concat codes*) *p′*))))
prefer *2* **apply** (*rule j-length*)
apply (*frule-tac i*=*j* **in** *nth-drop′*,*simp*)
apply (*subgoal-tac* (*Suc j*) < *length* (*fst* (*the* (*map-of* (*concat codes*) *p′*))))
prefer *2* **apply** (*rule-tac j*=*Suc j* **in** *j-length*)
apply (*frule-tac i*=*Suc j* **in** *nth-drop′*,*simp*)
apply (*subgoal-tac* (*Suc* (*Suc j*)) < *length* (*fst* (*the* (*map-of* (*concat codes*) *p′*))))

prefer *2* **apply** (*rule-tac j*=(*Suc* (*Suc j*)) **in** *j-length*)
apply (*frule-tac i*=(*Suc* (*Suc j*)) **in** *nth-drop′*,*simp*)
apply (*subgoal-tac* (*Suc* (*Suc* (*Suc j*))) < *length* (*fst* (*the* (*map-of* (*concat codes*) *p′*))))
prefer *2* **apply** (*rule-tac j*=(*Suc* (*Suc* (*Suc j*))) **in** *j-length*)
apply (*frule-tac i*=(*Suc* (*Suc* (*Suc j*))) **in** *nth-drop′*,*simp*)
apply (*elim conjE*)

apply (*simp add*: *Let-def*, *split split-if-asm*, *simp*)
apply (*rule execSVMBalanced.step*, *simp-all*)
apply (*rule execSVMBalanced.step*, *simp-all*)
apply (*rule execSVMBalanced.step*, *simp-all*)
apply (*rule execSVMBalanced.init*)
apply (*unfold execSVM-def*)
apply (*simp add*: *Let-def*, *rule conjI*, *simp*, *simp*)
apply (*simp add*: *Let-def*, *rule conjI*, *simp*, *simp*)
apply (*simp*, *simp add*: *Let-def*)
apply (*rule conjI*, *simp*, *simp*)
apply (*simp add*: *extendsL-def add*: *extends-def*)
by *clarsimp*

lemma *execSVMBalanced-VarE*:
⟦*E1 x* = *Some* (*Loc pa*);
(*q*, *ls*, *is*, *cs1*) = *trE p′ funm fname rho* (*VarE x a*);
cs = *concat codes*;
(*cs1* @ [(*q*, *is′*, *fname*)]) ⊑ *cs*;
drop j is′ = *is*;
P = ((*cs*, *contm*), *p*, *ct*, *st*);
td = *topDepth rho*;
S′ = *drop td S*;
(*E1*,*E2*) ⋈ (*rho*,*S*);
length rho > *0*;
topDepth rho < *length S*;
s0 = ((*h*, *k*), *k0*, (*q*, *j*), *S*)⟧
⟹ *P*⊢*s0* , *td*#*tds* −*svm*→ ((*h*, *k*) ↓ *k0*, *k0*, (*q*, *Suc* (*Suc* (*Suc j*))), *Val* (*Loc pa*) # *S′*) #
[((*h*, *k*), *k0*, (*q*, *j* + *2*), *Val* (*Loc pa*) # *drop* (*topDepth rho*) *S*),
((*h*, *k*), *k0*, (*q*, *j* + *1*), *Val* (*Loc pa*) # *S*),
((*h*, *k*), *k0*, (*q*, *j*), *S*)] , *Suc 0*#*tds* ∧ *fst* (*the* (*map-of cs q*)) ! *Suc* (*Suc* (*Suc j*)) = *POPCONT*
apply *simp*
apply (*subgoal-tac is′* = *fst* (*the* (*map-of* (*concat codes*) *p′*)))
apply (*subgoal-tac j* < *length* (*fst* (*the* (*map-of* (*concat codes*) *p′*))))
prefer *2* **apply** (*rule j-length*)
apply (*frule-tac i*=*j* **in** *nth-drop′*,*simp*)
apply (*subgoal-tac* (*Suc j*) < *length* (*fst* (*the* (*map-of* (*concat codes*) *p′*))))
prefer *2* **apply** (*rule-tac j*=*Suc j* **in** *j-length*)
apply (*frule-tac i*=*Suc j* **in** *nth-drop′*,*simp*)
apply (*subgoal-tac* (*Suc* (*Suc j*)) < *length* (*fst* (*the* (*map-of* (*concat codes*) *p′*))))

prefer *2* **apply** (*rule-tac j*=(*Suc* (*Suc j*)) **in** *j-length*)
apply (*frule-tac i*=(*Suc* (*Suc j*)) **in** *nth-drop′*,*simp*)
apply (*subgoal-tac* (*Suc* (*Suc* (*Suc j*))) < *length* (*fst* (*the* (*map-of* (*concat codes*) *p′*))))
prefer *2* **apply** (*rule-tac j*=(*Suc* (*Suc* (*Suc j*))) **in** *j-length*)

apply (*frule-tac i=(Suc (Suc (Suc j))) in nth-drop',simp*)
apply (*elim conjE*)
apply (*simp add*: *Let-def*, *split split-if-asm*, *simp*)
apply (*rule execSVMBalanced.step*, *simp-all*)
apply (*rule execSVMBalanced.step*, *simp-all*)
apply (*rule execSVMBalanced.step*, *simp-all*)
apply (*rule execSVMBalanced.init*)
apply (*unfold execSVM-def*)
apply (*simp add*: *Let-def*)
apply (*rule identityEnvironment-Val*, *assumption+*, *simp*)
apply (*rule conjI*, *simp*, *simp*)
apply (*simp add*: *Let-def*, *rule conjI*, *simp*, *simp*)
apply (*simp add*: *Let-def*, *rule conjI*, *simp*, *simp*)
apply (*simp add*: *extendsL-def add*: *extends-def*)
by *clarsimp*

lemma *execSVMBalanced-CopyE*:
⟦*E1 x = Some (Loc pa)*;
E2 r = Some j;
j ≤ k; *0 < m*;
SafeHeap.copy (h, k) j pa = ((h', k), p');
(q, ls, is, cs1) = trE p'a funm fname rho (x @ r a);
(cs1 @ [(q, is', fname)]) ⊑ cs; *cs = concat codes*;
drop ja is' = is;
P = ((cs, contm), p, ct, st);
(E1, E2) ⋈ (rho , S) ;
td = topDepth rho; *k0' ≤ k*;
S' = drop td S;
length rho > 0;
topDepth rho < length S;
s0 = ((h, k), k0', (q, ja), S)⟧
⟹ *P⊢s0 , td#tds −svm→ [((h', k) ↓ k0', k0', (q, Suc (Suc (Suc (Suc ja)))), Val (Loc p') # S'),*
((h', k), k0', (q, Suc (Suc (Suc ja))), Val (Loc p') # drop (topDepth rho) S),
((h', k), k0', (q, Suc (Suc ja)), Val (Loc p') # S),
((h, k), k0', (q, Suc ja), Val (Loc pa) # Reg j # S),
((h, k), k0', (q, ja), S)] , Suc 0#tds
∧ *fst (the (map-of cs q)) ! Suc (Suc (Suc (Suc ja))) = POPCONT*
apply *simp*
apply (*subgoal-tac is' = fst (the (map-of (concat codes) p'a))*)
apply (*subgoal-tac ja < length (fst (the (map-of (concat codes) p'a))))*
prefer *2* **apply** (*rule j-length*)
apply (*frule-tac i=ja in nth-drop',simp*)
apply (*subgoal-tac (Suc ja) < length (fst (the (map-of (concat codes) p'a))))*
prefer *2* **apply** (*rule-tac j=Suc ja in j-length*)
apply (*frule-tac i=Suc ja in nth-drop',simp*)
apply (*subgoal-tac (Suc (Suc ja)) < length (fst (the (map-of (concat codes) p'a))))*

prefer *2* **apply** (*rule-tac j*=(*Suc* (*Suc ja*)) **in** *j-length*)
apply (*frule-tac i*=(*Suc* (*Suc ja*)) **in** *nth-drop'*,*simp*)
apply (*subgoal-tac* (*Suc* (*Suc* (*Suc ja*))) < *length* (*fst* (*the* (*map-of* (*concat codes*) *p'a*))))
prefer *2* **apply** (*rule-tac j*=(*Suc* (*Suc* (*Suc ja*))) **in** *j-length*)
apply (*frule-tac i*=(*Suc* (*Suc* (*Suc ja*))) **in** *nth-drop'*,*simp*)
apply (*subgoal-tac* (*Suc* (*Suc* (*Suc* (*Suc ja*)))) < *length* (*fst* (*the* (*map-of* (*concat codes*) *p'a*))))
prefer *2* **apply** (*rule-tac j*=(*Suc* (*Suc* (*Suc* (*Suc ja*)))) **in** *j-length*)
apply (*frule-tac i*=(*Suc* (*Suc* (*Suc* (*Suc ja*)))) **in** *nth-drop'*,*simp*)
apply (*elim conjE*)
apply (*simp add*: *Let-def*, *split split-if-asm*, *simp*)
apply (*rule execSVMBalanced.step*, *simp-all*)
apply (*rule execSVMBalanced.step*, *simp-all*)
apply (*rule execSVMBalanced.step*, *simp-all*)
apply (*rule execSVMBalanced.step*, *simp-all*)
apply (*rule execSVMBalanced.init*)
apply (*unfold execSVM-def*)
apply (*simp add*: *Let-def*)
apply (*rule conjI*) **apply** (*frule identityEnvironment-Val*, *assumption+*, *simp*, *assumption*)
apply (*frule identityEnvironment-Reg*, *assumption+*, *simp*, *assumption*)
apply (*rule conjI*, *simp*, *simp*)
apply (*simp*, *rule conjI*, *simp*, *simp*)
apply (*simp*, *rule conjI*, *simp*, *simp*)
apply (*simp*, *simp add*: *Let-def*)
apply (*rule conjI*, *simp*, *simp*)
apply (*simp add*: *extendsL-def add*: *extends-def*)
by *clarsimp*

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

lemma *execSVMBalanced-ReuseE*:
⟦*E1 x* = *Some* (*Loc pa*);
h pa = *Some c*;
SafeHeap.fresh q h;
(*qa*, *ls*, *is*, *cs1*) = *trE p' funm fname rho* (*ReuseE x a*);
(*cs1* @ [(*qa*, *is'*, *fname*)]) ⊑ *cs*;
cs = *concat codes*;
drop j is' = *is*;
P = ((*cs*, *contm*), *p*, *ct*, *st*); *finite* (*dom h*);
td = *topDepth rho*;
k0 ≤ *k*;
(*E1*, *E2*) ⋈ (*rho* , *S*) ;
length rho > *0*;
topDepth rho < *length S*;
S' = *drop td S*;
s0 = ((*h*, *k*), *k0*, (*qa*, *j*), *S*)⟧

$\Longrightarrow P{\vdash}s0\ ,\ td\#tds\ -svm{\rightarrow}\ [((h(pa := None)(q \mapsto c),\ k) \downarrow k0,\ k0,\ (qa,\ Suc\ (Suc\ (Suc\ (Suc\ j)))),\ Val\ (Loc\ q)\ \#\ S'),$
$((h(pa := None)(q \mapsto c),\ k),\ k0,\ (qa,\ Suc\ (Suc\ (Suc\ j))),\ Val\ (Loc\ q)\ \#\ drop\ (topDepth\ rho)\ S),$
$((h(pa := None)(q \mapsto c),\ k),\ k0,\ (qa,\ Suc\ (Suc\ j)),\ Val\ (Loc\ q)\ \#\ S),$
$((h,\ k),\ k0,\ (qa,\ Suc\ j),\ Val\ (Loc\ pa)\ \#\ S),\ ((h,\ k),\ k0,\ (qa,\ j),\ S)]\ ,\ Suc\ 0\#tds$
$\wedge\ fst\ (the\ (map\text{-}of\ cs\ qa))\ !\ Suc\ (Suc\ (Suc\ (Suc\ j)))\ =\ POPCONT$
apply *simp*
apply (*subgoal-tac is′ = fst* (*the* (*map-of* (*concat codes*) *qa*)))
apply (*subgoal-tac j* < *length* (*fst* (*the* (*map-of* (*concat codes*) *qa*))))
prefer *2* **apply** (*rule j-length*)
apply (*frule-tac i=j* **in** *nth-drop′,simp*)
apply (*subgoal-tac* (*Suc j*) < *length* (*fst* (*the* (*map-of* (*concat codes*) *qa*))))
prefer *2* **apply** (*rule-tac j=Suc j* **in** *j-length*)
apply (*frule-tac i=Suc j* **in** *nth-drop′,simp*)
apply (*subgoal-tac* (*Suc* (*Suc j*)) < *length* (*fst* (*the* (*map-of* (*concat codes*) *qa*))))

prefer *2* **apply** (*rule-tac j=*(*Suc* (*Suc j*)) **in** *j-length*)
apply (*frule-tac i=*(*Suc* (*Suc j*)) **in** *nth-drop′,simp*)
apply (*subgoal-tac* (*Suc* (*Suc* (*Suc j*))) < *length* (*fst* (*the* (*map-of* (*concat codes*) *qa*))))
prefer *2* **apply** (*rule-tac j=*(*Suc* (*Suc* (*Suc j*))) **in** *j-length*)
apply (*frule-tac i=*(*Suc* (*Suc* (*Suc j*))) **in** *nth-drop′,simp*)
apply (*subgoal-tac* (*Suc* (*Suc* (*Suc* (*Suc j*)))) < *length* (*fst* (*the* (*map-of* (*concat codes*) *qa*))))
prefer *2* **apply** (*rule-tac j=*(*Suc* (*Suc* (*Suc* (*Suc j*)))) **in** *j-length*)
apply (*frule-tac i=*(*Suc* (*Suc* (*Suc* (*Suc j*)))) **in** *nth-drop′,simp*)
apply (*elim conjE*)
apply (*simp add*: *Let-def*, *split split-if-asm*, *simp*)
apply (*rule execSVMBalanced.step*, *simp-all*)
apply (*rule execSVMBalanced.step*, *simp-all*)
apply (*rule execSVMBalanced.step*, *simp-all*)
apply (*rule execSVMBalanced.step*, *simp-all*)
apply (*rule execSVMBalanced.init*)
apply (*unfold execSVM-def*)
apply (*simp add*: *Let-def*)
apply (*rule identityEnvironment-Val*, *assumption+*)
apply *simp* **apply** (*rule conjI*, *simp*, *simp*)
apply (*unfold Let-def*)
apply (*simp add*: *Let-def*)
apply (*rule conjI*)
apply (*frule getFresh-fresh*) **apply** *simp*
apply (*erule getFresh-fresh*)
apply (*rule conjI*, *simp*, *simp*)
apply *simp* **apply** (*rule conjI*, *simp*, *simp*)
apply (*simp add*: *Let-def*)
apply (*rule conjI*, *simp*, *simp*)

apply (*simp add*: *extendsL-def add*: *extends-def*)
by *clarsimp*

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

lemma *execSVMBalanced-Let1*:
⟦*x1* ∉ *dom E1*; *e1* ≠ *ConstrE C as r1 a'*;
closureCalled (*Let x1* = *e1 In e2 a*) *defs* ⊆ *definedFuns defs*;
((*p*, *funm*, *contm*), *codes*) = *mapAccumL trF* (*1*, *empty*, []) *defs*;
cs = *concat codes*;
P = ((*cs*, *contm*), *p*, *ct*, *st*);
(*q*, *ls*, *is*, *cs1*) = *trE p' funm fname rho* (*Let x1* = *e1 In e2 a*);
(*cs1* @ [(*q*, *is'*, *fname*)]) ⊑ *cs*;
drop j is' = *is*;
(*E1*, *E2*) ⋈ (*rho*,*S*);
td = *topDepth rho*;
k0 ≤ *k*;
S' = *drop td S*;
s0 = ((*h*, *k*), *k0*, (*q*, *j*), *S*);
(*q1*, *ls1*, *is1*, *cs1'*) = *trE* (*q2* + *1*) *funm fname* (*rho* ++) *e1*;
(*q2*, *ls2*, *is2*, *cs2*) = *trE p' funm fname* (*rho* + [(*x1*, *1*)]) *e2*;
cs1 = *cs2* @ (*q2*, *is2*, *fname*) # *cs1'*; *q* = *q1*; *ls* = *q2* # *ls1* @ *ls2*; *is* = *PUSHCONT q2* # *is1*;
P⊢((*h*, *k*), *k*, (*q1*, *j* + *1*), *Cont* (*k0*, *q2*) # *S*) , *0*#*td*#*tds* −*svm*→ *s* # *ss* , *Suc 0*#*td*#*tds*;
P⊢ ((*h'* |` {*p* ∈ *dom h'*. *fst* (*the* (*h' p*)) ≤ *k*}, *k*), *k0*, (*q2*, *0*), *Val v1* # *S*), (*td*+*1*)#*tds* −*svm*→ *sa* # *ssa* , *Suc 0*#*tds*;
s = ((*h'* |` {*p* ∈ *dom h'*. *fst* (*the* (*h' p*)) ≤ *k*}, *k*)↓ *k*, *k*, (*q'*, *i*), *Val v1* # *drop 0* (*Cont* (*k0*, *q2*) # *S*));
sa = ((*h''*, *k*) ↓ *k0*, *k0*, (*q'a*, *ia*), *Val v2* # *S'*);
fst (*the* (*map-of cs q'*)) ! *i* = *POPCONT*;
fst (*the* (*map-of cs q'a*)) ! *ia* = *POPCONT*⟧
⟹ *P*⊢*s0* , *td*#*tds* −*svm*→ *sa* # *ssa* @ *s* # *ss* @ [((*h*, *k*), *k0*, (*q1*, *j*), *S*)] , *Suc 0*#*tds*
apply (*subgoal-tac fst* (*the* (*map-of cs q1*)) ! *j* = *PUSHCONT q2*)
prefer *2*
apply (*subgoal-tac is'* = *fst* (*the* (*map-of* (*concat codes*) *q1*)))
apply (*subgoal-tac j* < *length* (*fst* (*the* (*map-of* (*concat codes*) *q1*))))
prefer *2* **apply** (*rule j-length*)
apply (*frule-tac i*=*j* **in** *nth-drop'*,*simp*)
apply (*frule-tac ?cs2.0*=*cs2* **and** *cs1'*=*cs1'* **in** *codestore-let-disjoint*, *assumption*+)
apply (*simp add*: *disjointList-def*)
apply (*simp add*: *extendsL-def add*: *extends-def*)
apply *clarsimp*
apply (*rule execSVMBalanced-n-step*) **prefer** *3* **apply** *simp*
prefer *2* **apply** *simp*

apply (*rule execSVMBalanced.step*)

apply (*rule-tac s''=((h, k), k, (q1, j + 1), Cont (k0, q2) # S)* **in** *execSVMBalanced-n-step*)
apply *simp*
apply (*rule execSVMBalanced.step*)
apply (*rule execSVMBalanced.init*)
apply (*unfold execSVM-def*)
apply (*unfold Let-def*) **apply** (*simp*) **apply** *simp* **apply** *simp* **apply** *simp* **apply**
(*rule conjI*, *simp*, *simp*)
apply *simp* **apply** *simp*
apply *simp*
apply (*rule restrictToRegion-monotone*)
apply *simp* **apply** *simp*
apply *simp*
apply (*rule conjI*) **apply** *simp* **apply** *simp*
done

lemma *head-execSVMBalanced*:
  $P \vdash s0\ ,\ n\ -svm\rightarrow\ s\ \#\ ss,\ m \Longrightarrow s0 = last\ (s\#ss)$
by (*erule execSVMBalanced.induct*, *simp-all*)

lemma *head-cons-execSVMBalanced*:
  $[\![\ ss \neq []; s0 = last\ ss]\!] \Longrightarrow \exists\ ss'.\ ss = ss' @ [s0]$
by (*rule-tac x=butlast ss* **in** *exI*, *simp*)

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

lemma *exec-let2-3* [*rule-format*]:
  (*E1*,*E2*) $\bowtie$ (*rho*,*S*)
   $\longrightarrow$ *fvs as* $\subseteq$ *dom E1*
   $\longrightarrow$ ($\forall$ *a*$\in$ *set as*. *atom a*)
   $\longrightarrow$ *map* (*item2Val S*) (*map* (*atom2item rho*) *as*) = *map* (*atom2val E1*) *as*
apply (*induct as*, *simp*, *clarsimp*)
apply (*case-tac a*,*simp-all*)
apply (*rename-tac a as x n*)
apply (*case-tac x*,*simp-all*)
apply (*rename-tac a as x n*)
apply (*simp add*: *identityEnvironment.simps*)
apply (*elim conjE*)
apply (*erule-tac x=x* **in** *ballE*)
by (*case-tac S !+ (rho |> x)*,*simp-all*)

lemma *execSVMBalanced-Let2*:
  $[\![$*E2 r = Some j*; *j* $\leq$ *k*; *fresh pa h*; *x1* $\notin$ *dom E1*; *r* $\neq$ *self*; *length rho > 0*;
    cs = concat codes; *P* = ((*cs*, *contm*), *p*, *ct*, *st*); *finite* (*dom h*);
    fv (*Let x1 = ConstrE C as r a' In e2 a*) $\subseteq$ *dom* (*fst* (*E1*,*E2*));

$(\forall\ a \in set\ as.\ atom\ a);\ x1 \notin fvs\ as;$
$finite\ (dom\ (h(pa \mapsto (j,\ C,\ map\ (atom2val\ E1)\ as))));$
$(cs1\ @\ [(q,\ is',\ fname)]) \sqsubseteq cs;\ drop\ ja\ is' = is;$
$(q2,ls2,is2,cs2) = trE\ p'\ funm\ fname\ (rho+[(x1,1)])\ e2;$
$((p,\ funm,\ contm),\ codes) = mapAccumL\ trF\ (1,\ empty,\ [])\ defs;$
$(E1,\ E2) \bowtie (\ rho\ ,\ S)\ ;\ td = topDepth\ rho;\ k0 \leq k;\ S' = drop\ td\ S;\ s0 = ((h, k),\ k0,\ (q,\ ja),\ S);$
$items = map\ (atom2item\ rho)\ as;\ item = region2item\ rho\ r;\ q = q2;$
$ls = ls2;\ is = BUILDCLS\ C\ items\ item\ \#\ is2;\ cs1 = cs2;$
$P \vdash ((h(pa \mapsto (j,\ C,\ map\ (atom2val\ E1)\ as)),\ k),\ k0,\ (q,\ Suc\ ja),\ Val\ (Loc\ pa)\ \#\ S)\ ,\ (td\ +\ 1)\#tds\ -svm\rightarrow\ sa\ \#\ ss\ ,\ Suc\ 0\#tds;$
$sa = ((h',\ k) \downarrow k0,\ k0,\ (q',\ i),\ Val\ v2\ \#\ S')]\!]$
$\Longrightarrow P \vdash s0\ ,\ td\#tds\ -svm\rightarrow\ sa\ \#\ ss\ @\ [((h,\ k),\ k0,\ (q,\ ja),\ S)]\ ,\ Suc\ 0\#tds$
apply $(subgoal\text{-}tac\ fst\ (the\ (map\text{-}of\ (concat\ codes)\ q2))\ !\ ja =\ BUILDCLS\ C\ (map\ (atom2item\ rho)\ as)\ (ItemVar\ (rho\ |>\ r)))$
prefer *2*
apply $(subgoal\text{-}tac\ is' = fst\ (the\ (map\text{-}of\ (concat\ codes)\ q2)))$
apply $(subgoal\text{-}tac\ ja < length\ (fst\ (the\ (map\text{-}of\ (concat\ codes)\ q2))))$
prefer *2* **apply** $(rule\ j\text{-}length)$
apply $(frule\text{-}tac\ i{=}ja$ **in** $nth\text{-}drop',simp)$
apply $(frule\ codestore\text{-}let2\text{-}disjoint)$
apply $(simp\ add{:}\ extendsL\text{-}def\ add{:}\ extends\text{-}def)$
apply *clarsimp*
apply $(rule\text{-}tac\ s''{=}((h(pa \mapsto (j,\ C,\ map\ (atom2val\ E1)\ as)),\ k),\ k0,\ (q,\ Suc\ ja),\ Val\ (Loc\ pa)\ \#\ S)$ **in** $execSVMBalanced\text{-}n\text{-}step)$
prefer *3* **apply** *simp*
prefer *2* **apply** *simp*
apply $(rule\ execSVMBalanced.step)$
apply $(simp,\ rule\ execSVMBalanced.init)$
prefer *2* **apply** *simp*
prefer *2* **apply** *simp*
prefer *2* **apply** *simp*
apply $(unfold\ execSVM\text{-}def)$
apply $(unfold\ Let\text{-}def)$
apply *simp*
apply $(frule\ identityEnvironment\text{-}Reg,\ assumption{+},\ simp)$
apply *simp*
apply $(simp\ add{:}\ Let\text{-}def)$
apply $(frule\ getFresh\text{-}fresh)$
apply *simp*
apply $(subgoal\text{-}tac\ map\ (item2Val\ S)\ (map\ (atom2item\ rho)\ as) = map\ (atom2val\ E1)\ as,\ simp)$
apply $(subgoal\text{-}tac\ \exists h'.\ h' = h(pa \mapsto (j,\ C,\ map\ (atom2val\ E1)\ as)))$ **prefer** *2*
apply *simp*
apply $(erule\ exE)$
apply $(subgoal\text{-}tac$
$h'a = h(pa \mapsto (j,\ C,\ map\ (atom2val\ E1)\ as)) \longrightarrow$
$h'a\ pa = Some\ (j,\ C,\ map\ (atom2val\ E1)\ as))$
prefer *2* **apply** *simp*

apply (*frule exec-let2-3*, *blast*)
by (*erule-tac x=x* **and** *A=set as* **in** *ballE*,*simp*,*simp*,*simp*)

lemma *execSVMBalanced-Case*:
⟦ *cs* = *concat codes*; *P* = ((*cs*, *contm*), *p*, *ct*, *st*);
E1 x = *Some* (*Loc pa*); *h pa* = *Some* (*j*, *C*, *vs*);
nr = *int* (*length vs*);
(*q*, *ls*, *is*, *cs1*) = *trE p′ funm fname rho* (*Case VarE x a Of alts a′*);
i < *length alts*;
alts!*i* = (*pati*, *ei*);
pati = *ConstrP C ps ms*;
xs = *map pat2var ps*;
(*qs* ! *i*, *lss* !*i*, *isi*, *csi*) = *trE p′ funm fname* (*rho* + *zip xs* (*decreasing* (*length xs*))) *ei*;
(*cs1* @ [(*q*, *is′*, *fname*)]) ⊑ *cs*; *drop ja is′* = *is*; (*E1*, *E2*) ⋈ (*rho* , *S*) ; *td* = *topDepth rho*;
k0 ≤ *k*; *S′* = *drop td S*; *s0* = ((*h*, *k*), *k0*, (*q*, *ja*), *S*);
(*q*, *rss*) = *trAlts p′ funm fname rho alts*; (*lss*, *qs*, *css*) = *unzip3 rss*; *ls* = *concat lss*; *is* = [*MATCH* (*rho* |> *x*) *qs*];
cs1 = *concat css*;
map-of ct C = *Some* (*q1*,*r*,*q2*);
P⊢((*h*, *k*), *k0*, (*qs*!*r*, *0*), (*map Val vs*) @ *S*) , *nat* (*int td* + *nr*)#*tds* −*svm*→ *sa* # *ss* , *Suc 0*#*tds*;
length rho > *0*;
sa = ((*h′*, *k*) ↓ *k0*, *k0*, (*q′*, *ia*), *Val v* # *drop* (*nat* (*int td* + *nr*)) (*map Val vs* @ *S*));
fst (*the* (*map-of cs q′*)) ! *ia* = *POPCONT*⟧
⟹ *P*⊢*s0* , *td*#*tds* −*svm*→ *sa* # *ss* @ [((*h*, *k*), *k0*, (*q*, *ja*), *S*)] , *Suc 0*#*tds*
apply (*frule codestore-case-alts-disjoint*,*rule disjI1*,*rule refl*, *assumption*+)
apply (*rule execSVMBalanced-n-step*) **apply** *simp-all*
apply (*rule execSVMBalanced.step*) **apply** *simp-all*
apply (*rule execSVMBalanced.init*)
apply (*simp add*: *execSVM-def*)
apply (*simp add*: *Let-def*)
apply (*subgoal-tac fst* (*the* (*map-of* (*concat codes*) *q*)) ! *ja* = *MATCH* (*rho* |> *x*) *qs*,*simp*)
apply (*subgoal-tac S* !+ (*rho* |> *x*) = *Val* (*Loc pa*),*simp*)
apply (*rule identityEnvironment-Val*, *assumption*+) **apply** *simp*
apply (*subgoal-tac is′* = *fst* (*the* (*map-of* (*concat codes*) *q*)))
apply (*subgoal-tac ja* < *length* (*fst* (*the* (*map-of* (*concat codes*) *q*))))
prefer *2* **apply** (*rule j-length*)
apply (*frule-tac i=ja* **in** *nth-drop′*,*simp*)
apply (*simp add*: *extendsL-def add*: *extends-def*)
apply *clarsimp*
apply (*subgoal-tac fst* (*the* (*map-of* (*concat codes*) *q*)) ! *ja* = *MATCH* (*rho* |> *x*) *qs*,*simp*)
apply (*subgoal-tac is′* = *fst* (*the* (*map-of* (*concat codes*) *q*)))
apply (*subgoal-tac ja* < *length* (*fst* (*the* (*map-of* (*concat codes*) *q*))))

prefer *2* **apply** (*rule j-length*)
apply (*frule-tac i=ja* **in** *nth-drop′,simp*)
apply (*simp add*: *extendsL-def add*: *extends-def*)
by *clarsimp*

lemma *execSVMBalanced-Case-1-1*:
⟦ *cs* = *concat codes*; *P* = ((*cs*, *contm*), *p*, *ct*, *st*); (*E1*,*E2*) ⋈ (*rho*,*S*); *0* < *length rho*;
E1 x = *Some* (*IntT n*);
i < *length alts*;
alts!*i* = (*pati*, *ei*);
(*qs* ! *i*, *lss* !*i*, *isi*, *csi*) = *trE p′ funm fname rho ei*;
(*q*, *ls*, *is*, *cs1*) = *trE p′ funm fname rho* (*Case VarE x a Of alts a′*);
(*cs1* @ [(*q*, *is′*, *fname*)]) ⊑ *cs*; *drop j is′* = *is*; (*E1*, *E2*) ⋈ (*rho* , *S*) ; *td* = *topDepth rho*;
k0 ≤ *k*; *S′* = *drop td S*; *s0* = ((*h*, *k*), *k0*, (*q*, *j*), *S*);
(*q*, *rss*) = *trAlts p′ funm fname rho alts*; (*lss*, *qs*, *css*) = *unzip3 rss*; *ls* = *concat lss*; *is* = [*MATCHN* (*rho* |> *x*) *v′ m′ qs*];
cs1 = *concat css*;
P⊢((*h*, *k*), *k0*, (*qs* ! *i*, *0*), *S*) , *td* # *tds* −*svm*→ *sa* # *ss* , *Suc 0* # *tds*;
length rho > *0*;
sa = ((*h′*, *k*) ↓ *k0*, *k0*, (*q′*, *ia*), *Val v* # *drop td S*);
fst (*the* (*map-of cs q′*)) ! *ia* = *POPCONT*⟧
⟹ *P*⊢*s0* , *td* # *tds* −*svm*→ *sa* # *ss* @ [((*h*, *k*), *k0*, (*q*, *j*), *S*)] , *Suc 0* # *tds*
apply (*frule codestore-caseL-alts-disjoint*, *assumption+*)
apply *simp* **apply** (*unfold Let-def*)
apply (*rule execSVMBalanced-n-step*) **apply** *simp-all*
apply (*rule execSVMBalanced.step*) **apply** *simp-all*
apply (*rule execSVMBalanced.init*)
apply (*simp add*: *execSVM-def*)
apply (*simp add*: *Let-def*)
apply (*subgoal-tac fst* (*the* (*map-of* (*concat codes*) *q*)) ! *j* = *MATCHN* (*rho* |> *x*) *v′ m′ qs*,*simp*)
apply (*subgoal-tac S* !+ (*rho* |> *x*) = *Val* (*IntT n*),*simp*)
apply (*simp add*: *Let-def*)
apply (*frule SVM-Semantic-branch-Nat-consistent*,*assumption+*,*simp*)
apply (*rule identityEnvironment-Val*,*assumption+*,*simp*)
apply (*subgoal-tac is′* = *fst* (*the* (*map-of* (*concat codes*) *q*)))
apply (*subgoal-tac j* < *length* (*fst* (*the* (*map-of* (*concat codes*) *q*))))
prefer *2* **apply** (*rule j-length*)
apply (*frule-tac i=j* **in** *nth-drop′,simp*)
apply (*simp add*: *extendsL-def add*: *extends-def*)
apply *clarsimp*
apply (*subgoal-tac fst* (*the* (*map-of* (*concat codes*) *q*)) ! *j* = *MATCHN* (*rho* |> *x*) *v′ m′ qs*,*simp*)
apply (*subgoal-tac is′* = *fst* (*the* (*map-of* (*concat codes*) *q*)))
apply (*subgoal-tac j* < *length* (*fst* (*the* (*map-of* (*concat codes*) *q*))))
prefer *2* **apply** (*rule j-length*)

apply (*frule-tac i=j* **in** *nth-drop′,simp*)
apply (*simp add*: *extendsL-def add*: *extends-def*)
by *clarsimp*

lemma *execSVMBalanced-Case-1-2*:
⟦ *cs* = *concat codes*; *P* = ((*cs*, *contm*), *p*, *ct*, *st*); (*E1*,*E2*) ⋈ (*rho*,*S*); *0* < *length rho*;
i < *length alts*;
alts!*i* = (*pati*, *ei*);
E1 x = *Some* (*BoolT b*);
(*q*, *ls*, *is*, *cs1*) = *trE p′ funm fname rho* (*Case VarE x a Of alts a′*);
(*qs* ! *i*, *lss* !*i*, *isi*, *csi*) = *trE p′ funm fname rho ei*;
(*cs1* @ [(*q*, *is′*, *fname*)]) ⊑ *cs*; *drop j is′* = *is*; (*E1*, *E2*) ⋈ (*rho* , *S*) ; *td* = *topDepth rho*;
k0 ≤ *k*; *S′* = *drop td S*; *s0* = ((*h*, *k*), *k0*, (*q*, *j*), *S*);
(*q*, *rss*) = *trAlts p′ funm fname rho alts*; (*lss*, *qs*, *css*) = *unzip3 rss*; *ls* = *concat lss*; *is* = [*MATCHN* (*rho* |> *x*) *0 2 qs*];
cs1 = *concat css*;
P⊢((*h*, *k*), *k0*, (*qs* ! *i*, *0*), *S*) , *td* # *tds* −*svm*→ *sa* # *ss* , *Suc 0* # *tds*;
length rho > *0*;
sa = ((*h′*, *k*) ↓ *k0*, *k0*, (*q′*, *ia*), *Val v* # *drop td S*);
fst (*the* (*map-of cs q′*)) ! *ia* = *POPCONT*⟧
⟹ *P*⊢*s0* , *td* # *tds* −*svm*→ *sa* # *ss* @ [((*h*, *k*), *k0*, (*q*, *j*), *S*)] , *Suc 0* # *tds*
apply (*frule codestore-caseL-alts-disjoint*, *assumption*+)
apply *simp* **apply** (*unfold Let-def*)
apply (*rule execSVMBalanced-n-step*) **apply** *simp-all*
apply (*rule execSVMBalanced.step*) **apply** *simp-all*
apply (*rule execSVMBalanced.init*)
apply (*simp add*: *execSVM-def*)
apply (*simp add*: *Let-def*)
apply (*subgoal-tac fst* (*the* (*map-of* (*concat codes*) *q*)) ! *j* = *MATCHN* (*rho* |> *x*) *0 2 qs*,*simp*)
apply (*subgoal-tac S* !+ (*rho* |> *x*) = *Val* (*BoolT b*),*simp*)
apply (*simp add*: *Let-def*)
apply (*frule SVM-Semantic-branch-Bool-consistent*,*assumption*+)
apply (*rule identityEnvironment-Val*,*assumption*+,*simp*)
apply (*subgoal-tac is′* = *fst* (*the* (*map-of* (*concat codes*) *q*)))
apply (*subgoal-tac j* < *length* (*fst* (*the* (*map-of* (*concat codes*) *q*))))
prefer *2* **apply** (*rule j-length*)
apply (*frule-tac i=j* **in** *nth-drop′,simp*)
apply (*simp add*: *extendsL-def add*: *extends-def*)
apply *clarsimp*
apply (*subgoal-tac fst* (*the* (*map-of* (*concat codes*) *q*)) ! *j* = *MATCHN* (*rho* |> *x*) *0 2 qs*,*simp*)
apply (*subgoal-tac is′* = *fst* (*the* (*map-of* (*concat codes*) *q*)))
apply (*subgoal-tac j* < *length* (*fst* (*the* (*map-of* (*concat codes*) *q*))))
prefer *2* **apply** (*rule j-length*)

apply (*frule-tac i=j* **in** *nth-drop′,simp*)
apply (*simp add*: *extendsL-def add*: *extends-def*)
by *clarsimp*

lemma *execSVMBalanced-CaseD*:
⟦ *cs* = *concat codes*; *P* = ((*cs*, *contm*), *p*, *ct*, *st*);
E1 x = *Some* (*Loc pa*); *h pa* = *Some* (*j*, *C*, *vs*);
i < *length alts*;
alts!*i* = (*pati*, *ei*);
pati = *ConstrP C ps ms*;
xs = *map pat2var ps*;
(*qs* ! *i*, *lss* !*i*, *isi*, *csi*) = *trE p′ funm fname* (*rho* + *zip xs* (*decreasing* (*length xs*))) *ei*;
nr = *int* (*length vs*);
(*q*, *ls*, *is*, *cs1*) = *trE p′ funm fname rho* (*CaseD VarE x a Of alts a′*);
(*cs1* @ [(*q*, *is′*, *fname*)]) ⊑ *cs*; *drop ja is′* = *is*; (*E1*, *E2*) ⋈ (*rho* , *S*) ; *td* = *topDepth rho*;
k0 ≤ *k*; *S′* = *drop td S*; *s0* = ((*h*, *k*), *k0*, (*q*, *ja*), *S*);
(*q*, *rss*) = *trAlts p′ funm fname rho alts*; (*lss*, *qs*, *css*) = *unzip3 rss*; *ls* = *concat lss*; *is* = [*MATCHD* (*rho* |> *x*) *qs*];
cs1 = *concat css*;
map-of ct C = *Some* (*q1*,*r*,*q2*);
P⊢((*h*(*pa* := *None*), *k*), *k0*, (*qs*!*r*, *0*), (*map Val vs*) @ *S*) , *nat* (*int td* + *nr*)#*tds* −*svm*→ *sa* # *ss* , *Suc 0*#*tds*;
length rho > *0*;
sa = ((*h′*, *k*) ↓ *k0*, *k0*, (*q′*, *ia*), *Val v* # *drop* (*nat* (*int td* + *nr*)) (*map Val vs* @ *S*));
fst (*the* (*map-of cs q′*)) ! *ia* = *POPCONT*⟧
⟹ *P*⊢*s0* , *td*#*tds* −*svm*→ *sa* # *ss* @ [((*h*, *k*), *k0*, (*q*, *ja*), *S*)] , *Suc 0*#*tds*
apply (*frule codestore-case-alts-disjoint*, *rule disjI2*, *rule refl*, *assumption*+)
apply (*rule execSVMBalanced-n-step*) **apply** *simp-all*
apply (*rule execSVMBalanced.step*) **apply** *simp-all*
apply (*rule execSVMBalanced.init*)
apply (*simp add*: *execSVM-def*)
apply (*simp add*: *Let-def*)
apply (*subgoal-tac fst* (*the* (*map-of* (*concat codes*) *q*)) ! *ja* = *MATCHD* (*rho* |> *x*) *qs*,*simp*)
apply (*subgoal-tac S* !+ (*rho* |> *x*) = *Val* (*Loc pa*),*simp*)
apply (*simp add*: *Let-def*)
apply (*rule identityEnvironment-Val*, *assumption*+) **apply** *simp*
apply (*subgoal-tac is′* = *fst* (*the* (*map-of* (*concat codes*) *q*)))
apply (*subgoal-tac ja* < *length* (*fst* (*the* (*map-of* (*concat codes*) *q*))))
prefer *2* **apply** (*rule j-length*)
apply (*frule-tac i=ja* **in** *nth-drop′,simp*)
apply (*simp add*: *extendsL-def add*: *extends-def*)
apply *clarsimp*
apply (*subgoal-tac fst* (*the* (*map-of* (*concat codes*) *q*)) ! *ja* = *MATCHD* (*rho* |> *x*) *qs*,*simp*)

apply (*subgoal-tac is′ = fst* (*the* (*map-of* (*concat codes*) *q*)))
apply (*subgoal-tac ja* < *length* (*fst* (*the* (*map-of* (*concat codes*) *q*))))
prefer *2* **apply** (*rule j-length*)
apply (*frule-tac i=ja* **in** *nth-drop′,simp*)
apply (*simp add*: *extendsL-def add*: *extends-def*)
by *clarsimp*

lemma *execSVMBalanced-App*:
⟦*n* = *length xs*; *l* = *length rs*; *primops f* = *None*; *length as* = *length xs*; *length rr* = *length rs*;
qf = *the* (*funm f*);
cs = *concat codes*; *P* = ((*cs*, *contm*), *p*, *ct*, *st*); *finite* (*dom h*); (*q*, *ls*, *is*, *cs1*) = *trE p′ funm fname rho* (*AppE f as rr a*);
(*cs1* @ [(*q*, *is′*, *fname*)]) ⊑ *cs*; *drop j is′* = *is*; (*E1*, *E2*) ⋉ (*rho* , *S*) ; *td* = *topDepth rho*; *k0* ≤ *k*; *S′* = *drop td S*;
s0 = ((*h*, *k*), *k0*, (*q*, *j*), *S*); *topDepth rho* < *length S*;
P⊢((*h*, *Suc k*), *k0*, (*qf*, *0*),
append (*map* (*item2Stack k S*) (*map* (*atom2item rho*) *as*))
(*append* (*map* (*item2Stack k S*) (*map* (*region2item rho*) *rr*)) (*drop td S*))) , (*n* + *l*)#*tds* −*svm*→ *sa* # *ss* , *Suc 0*#*tds*;
sa = ((*h′*, *Suc k*) ↓ *k0*, *k0*, (*q′*, *i*), *Val v* # (*drop td S*)); *fst* (*the* (*map-of cs q′*)) ! *i* = *POPCONT*⟧
⟹ *P*⊢*s0* , *td*#*tds* −*svm*→ *sa* # *ss* @
[((*h*, *k*), *k0*, (*q*, *Suc* (*Suc j*)), *append* (*append* (*map* (*item2Stack k S*) (*map* (*atom2item rho*) *as*))
(*map* (*item2Stack k S*) (*map* (*region2item rho*) *rr*))) (*drop td S*)),
((*h*, *k*), *k0*, (*q*, *Suc j*), *append* (*append* (*map* (*item2Stack k S*) (*map* (*atom2item rho*) *as*))
(*map* (*item2Stack k S*) (*map* (*region2item rho*) *rr*))) *S*),
((*h*, *k*), *k0*, (*q*, *j*), *S*)] , *Suc 0*#*tds*
apply *simp*
apply (*subgoal-tac is′ = fst* (*the* (*map-of* (*concat codes*) *p′*)))
apply (*subgoal-tac j* < *length* (*fst* (*the* (*map-of* (*concat codes*) *p′*))))
prefer *2* **apply** (*rule j-length*)
apply (*frule-tac i=j* **in** *nth-drop′,simp*)
apply (*subgoal-tac* (*Suc j*) < *length* (*fst* (*the* (*map-of* (*concat codes*) *p′*))))
prefer *2* **apply** (*rule-tac j=Suc j* **in** *j-length*)
apply (*frule-tac i=Suc j* **in** *nth-drop′,simp*)
apply (*subgoal-tac* (*Suc* (*Suc j*)) < *length* (*fst* (*the* (*map-of* (*concat codes*) *p′*))))

prefer *2* **apply** (*rule-tac j=*(*Suc* (*Suc j*)) **in** *j-length*)
apply (*frule-tac i=*(*Suc* (*Suc j*)) **in** *nth-drop′,simp*)
apply (*simp add*: *Let-def*) **apply** (*elim conjE*)
apply (*rule execSVMBalanced-n-step*) **apply** *simp-all*
apply (*rule execSVMBalanced.step*) **apply** *simp-all*

apply (*rule execSVMBalanced.step*) **apply** *simp-all*
apply (*rule execSVMBalanced.step*) **apply** *simp-all*
apply (*rule execSVMBalanced.init*)
apply (*simp add*: *execSVM-def*)
apply (*rule conjI*) **apply** *simp* **apply** *simp*
apply (*simp add*: *execSVM-def*)
apply (*rule conjI*) **apply** *simp* **apply** *simp*
apply (*simp add*: *execSVM-def*) **apply** (*simp add*: *Let-def*)
apply (*simp add*: *Let-def*, *elim conjE*)
apply (*simp add*: *extendsL-def add*: *extends-def*)
by *clarsimp*

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

lemma *identityEnvironment-item-val* [*rule-format*]:
(*E1*,*E2*) ⋈ (*rho*,*S*)
⟶ *fv a* ⊆ *dom E1*
⟶ *atom a*
⟶ *item2Stack k S* (*atom2item rho a*) = *Val* (*atom2val E1 a*)
apply (*case-tac a*,*simp-all*)
apply (*rename-tac n a'*)
apply (*case-tac n*, *simp-all*)
apply (*rename-tac x a'*)
apply (*rule impI*)
by (*simp add*: *identityEnvironment.simps*)

lemma *execSVMBalanced-App-primops*:
⟦*primops f* = *Some oper*; *v1* = *atom2val E1 a1*; *v2* = *atom2val E1 a2*; *v* = *execOp oper v1 v2*;
fv (*AppE f* [*a1*, *a2*] [] *a*) ⊆ *dom* (*fst* (*E1*, *E2*));
∀ *a* ∈ *set* [*a1*, *a2*]. *atom a*;
cs = *concat codes*; *P* = ((*cs*, *contm*), *p*, *ct*, *st*); *finite* (*dom h*); (*q*, *ls*, *is*, *cs1*) = *trE p' funm fname rho* (*AppE f* [*a1*, *a2*] [] *a*);
(*cs1* @ [(*q*, *is'*, *fname*)]) ⊑ *cs*; *drop j is'* = *is*; (*E1*, *E2*) ⋈ (*rho* , *S*) ; *td* = *topDepth rho*; *k0* ≤ *k*; *S'* = *drop td S*;
s0 = ((*h*, *k*), *k0*, (*q*, *j*), *S*)⟧
⟹ *P*⊢*s0* , *td* # *tds* −*svm*→
[((*h*, *k*) ↓ *k0*, *k0*, (*q*, *Suc* (*Suc* (*Suc* (*Suc j*)))), *Val* (*execOp oper* (*atom2val E1 a1*) (*atom2val E1 a2*)) # *drop* (*topDepth rho*) *S*),
((*h*, *k*), *k0*, (*q*, *Suc* (*Suc* (*Suc j*))), *Val* (*execOp oper* (*atom2val E1 a1*) (*atom2val E1 a2*)) # *drop* (*topDepth rho*) *S*),
((*h*, *k*), *k0*, (*q*, *Suc* (*Suc j*)), *Val v* # *S*),
((*h*, *k*), *k0*, (*q*, *Suc j*), (*map* (*item2Stack k S*) [*atom2item rho a1*, *atom2item rho a2*]) @ *S*),
((*h*, *k*), *k0*, (*q*, *j*), *S*)] , *Suc 0* # *tds* ∧ *fst* (*the* (*map-of cs q*)) ! *Suc* (*Suc* (*Suc* (*Suc j*))) = *POPCONT*

apply *simp* **apply** (*simp add*: *Let-def*) **apply** (*elim conjE*)
apply (*subgoal-tac topDepth rho* < *length S*)
prefer *2* **apply** (*rule lengthS-topDepth*,*assumption*)
apply (*subgoal-tac is'* = *fst* (*the* (*map-of* (*concat codes*) *p'*)))
apply (*subgoal-tac j* < *length* (*fst* (*the* (*map-of* (*concat codes*) *p'*))))
prefer *2* **apply** (*rule j-length*)
apply (*frule-tac i*=*j* **in** *nth-drop'*,*simp*)
apply (*subgoal-tac* (*Suc j*) < *length* (*fst* (*the* (*map-of* (*concat codes*) *p'*))))
prefer *2* **apply** (*rule-tac j*=*Suc j* **in** *j-length*)
apply (*frule-tac i*=*Suc j* **in** *nth-drop'*,*simp*)
apply (*subgoal-tac* (*Suc* (*Suc j*)) < *length* (*fst* (*the* (*map-of* (*concat codes*) *p'*))))

prefer *2* **apply** (*rule-tac j*=(*Suc* (*Suc j*)) **in** *j-length*)
apply (*frule-tac i*=(*Suc* (*Suc j*)) **in** *nth-drop'*,*simp*)
apply (*subgoal-tac* (*Suc* (*Suc* (*Suc j*))) < *length* (*fst* (*the* (*map-of* (*concat codes*) *p'*))))
prefer *2* **apply** (*rule-tac j*=(*Suc* (*Suc* (*Suc j*))) **in** *j-length*)
apply (*frule-tac i*=(*Suc* (*Suc* (*Suc j*))) **in** *nth-drop'*,*simp*)
apply (*subgoal-tac* (*Suc* (*Suc* (*Suc* (*Suc j*)))) < *length* (*fst* (*the* (*map-of* (*concat codes*) *p'*))))
prefer *2* **apply** (*rule-tac j*=(*Suc* (*Suc* (*Suc* (*Suc j*)))) **in** *j-length*)
apply (*frule-tac i*=(*Suc* (*Suc* (*Suc* (*Suc j*)))) **in** *nth-drop'*,*simp*)
apply (*elim conjE*)
apply (*split split-if-asm*, *simp*)
apply (*rule execSVMBalanced.step*)
apply (*rule execSVMBalanced.step*)
apply (*rule execSVMBalanced.step*)
apply (*rule execSVMBalanced.step*)
apply (*rule execSVMBalanced.init*)
apply (*unfold execSVM-def*) **apply** (*simp add*: *Let-def*)
apply *simp* **apply** *simp* **apply** *simp*
apply (*rule conjI*) **apply** *simp* **apply** *simp*

apply (*simp add*: *Let-def*)
apply (*frule-tac a*=*a1* **and** *k*=*k* **in** *identityEnvironment-item-val*,*assumption*+)
apply (*case-tac item2Stack k S* (*atom2item rho a1*) = *Val v1*) **apply** *simp*
apply (*frule-tac a*=*a2* **and** *k*=*k* **in** *identityEnvironment-item-val*,*assumption*+)
apply (*case-tac item2Stack k S* (*atom2item rho a2*) = *Val v2*) **apply** *simp*
apply (*simp add*: *Let-def*) **apply** *simp* **apply** *simp*
apply *simp* **apply** *simp* **apply** *simp*
apply (*rule conjI*) **apply** *simp* **apply** *simp*

apply (*simp add*: *Let-def*)
apply *simp* **apply** *simp* **apply** *simp*
apply (*rule conjI*) **apply** *simp* **apply** *simp*

apply (*simp add*: *Let-def*)
apply *simp* **apply** *simp* **apply** *simp*
apply (*rule conjI*) **apply** (*rule impI*)

apply *simp* **apply** *simp* **apply** *simp*
apply (*simp add*: *extendsL-def add*: *extends-def*)
by *clarsimp*

end

12 Certification of the translation CoreSafe to SVM. diff property

theory *diff-lemmas*
imports *Main ../CoreSafe/SafeRASemantics SVMSemantics CoreSafeToSVM CertifSafeToSVM-definitions*
begin

lemma *empty-diff*: $[]_k = diff\ k\ h\ h$
by (*simp add*: *diff-def add*: *emptyDelta-def*, *rule ext*, *simp*)

lemma *card-union-monotone*:
$finite\ (dom\ h) \Longrightarrow$
$card\ (\{w.\ w = q \wedge\ region\ (h(p := None)(q \mapsto c),\ k)\ q = i\} \cup$
$\{w \in dom\ h.\ w \neq q \wedge w \neq p \wedge region\ (h(p := None)(q \mapsto c),\ k)\ w = i\})$
$=$
$card\ (\{w.\ w = q \wedge\ region\ (h(p := None)(q \mapsto c),\ k)\ q = i\})\ +$
$card\ (\{w \in dom\ h.\ w \neq q \wedge w \neq p \wedge region\ (h(p := None)(q \mapsto c),\ k)\ w = i\})$
apply (*subgoal-tac* $\{w.\ w = q \wedge\ region\ (h(p := None)(q \mapsto c),\ k)\ q = i\} \cap$
$\{w \in dom\ h.\ w \neq q \wedge w \neq p \wedge region\ (h(p := None)(q \mapsto c),\ k)\ w = i\} = \{\}$)
apply (*subgoal-tac finite* $\{w.\ w = q \wedge\ region\ (h(p := None)(q \mapsto c),\ k)\ q = i\}$)
apply (*subgoal-tac finite* $\{w \in dom\ h.\ w \neq q \wedge w \neq p \wedge region\ (h(p := None)(q \mapsto c),\ k)\ w = i\}$)
apply (*simp add*: *card-Un-Int*)
apply *simp*
apply (*case-tac region* $(h(p := None)(q \mapsto c),\ k)\ q = i$,*simp*,*simp*)
by *blast*

lemma *card-union-monotone-2*:
$finite\ (dom\ h)$
$\Longrightarrow card\ (\{w \in dom\ h.\ w = p \wedge region\ (h,\ k)\ p = i\} \cup$
$\{w \in dom\ h.\ w \neq q \wedge w \neq p \wedge region\ (h,\ k)\ w = i\}) =$
$card\ (\{w \in dom\ h.\ w = p \wedge region\ (h,\ k)\ p = i\}) +$
$card\ (\{w \in dom\ h.\ w \neq q \wedge w \neq p \wedge region\ (h,\ k)\ w = i\})$
apply (*subgoal-tac* $\{w \in dom\ h.\ w = p \wedge region\ (h,\ k)\ p = i\} \cap$
$\{w \in dom\ h.\ w \neq q \wedge w \neq p \wedge region\ (h,\ k)\ w = i\} = \{\}$)
apply (*subgoal-tac finite* $\{w \in dom\ h.\ w = p \wedge region\ (h,\ k)\ p = i\}$)

apply (*subgoal-tac finite* $\{w \in dom\ h.\ w \neq q \wedge w \neq p \wedge region\ (h, k)\ w = i\}$)
apply (*simp add*: *card-Un-Int*)
apply *simp*
apply *simp*
by *blast*

lemma *diff-monotone-aux-2*:
$\{w \in dom\ h.\ w \neq q \wedge w \neq p \wedge region\ (h(p := None)(q \mapsto c), k)\ w = i\} =$
$\{w \in dom\ h.\ w \neq q \wedge w \neq p \wedge region\ (h, k)\ w = i\}$
apply (*subgoal-tac*
$w \in dom\ h \longrightarrow w \neq q \wedge w \neq p \wedge region\ (h(p := None)(q \mapsto c), k)\ w = i$
$\Longrightarrow w \in dom\ h \longrightarrow w \neq p \wedge region\ (h(p := None), k)\ w = i$)
apply (*subgoal-tac* $w \in dom\ h \longrightarrow w \neq p \wedge region\ (h(p := None), k)\ w = i$
$\Longrightarrow w \in dom\ h \longrightarrow region\ (h, k)\ w = i$)
by (*auto*, *simp-all add*: *region-def*)

lemma *diff-monotone-aux*:
⟦*finite* (*dom h*);
h p = *Some c*;
SafeHeap.fresh q h ⟧
$\Longrightarrow$ *diff k* (*h*, *k*) ((*h*(*p* := *None*))(*q* ↦ *c*), *k*) = *diff k* (*h*,*k*) (*h*,*k*)
apply (*simp add*: *diff-def*)
apply (*unfold numCellsRegion-def*)
apply (*rule ext*)
apply (*case-tac i* <= *k*)
apply *simp-all*
apply (*subgoal-tac*
$\{w.\ (((w = q) \vee ((w \in (dom\ h)) \wedge (w \neq p))) \wedge ((region\ (((h(p := None))(q \mapsto c)), k)\ w) = i)\} =$
$\{w.\ w{=}q \wedge region\ (h(p := None)(q \mapsto c), k)\ q = i\} \cup$
$\{w{\in}\ dom\ h.\ w \in dom\ h \wedge w{\neq}\ q \wedge w \neq p \wedge region\ (h(p := None)(q \mapsto c), k)\ w = i\}$)
prefer *2* **apply** *auto*
apply (*subgoal-tac*
⟦ *h p* = *Some c*;
SafeHeap.fresh q h ⟧
$\Longrightarrow \{w \in dom\ h.\ region\ (h, k)\ w = i\} =$
$\{w \in dom\ h.\ w = p \wedge region\ (h, k)\ p = i\} \cup$
$\{w \in dom\ h.\ w \in dom\ h \wedge w{\neq}\ q \wedge w{\neq}\ p \wedge region\ (h, k)\ w = i\}$,*simp*)
prefer *2* **apply** (*simp add*: *SafeHeap.fresh-def*, *blast*)
apply (*subgoal-tac*
finite (*dom h*)
$\Longrightarrow card\ (\{w.\ w = q \wedge\ region\ (h(p := None)(q \mapsto c), k)\ q = i\} \cup$
$\{w \in dom\ h.\ w \neq q \wedge w \neq p \wedge region\ (h(p := None)(q \mapsto c), k)\ w = i\}) =$
card ($\{w.\ w = q \wedge\ region\ (h(p := None)(q \mapsto c), k)\ q = i\}$) +
card ($\{w \in dom\ h.\ w \neq q \wedge w \neq p \wedge region\ (h(p := None)(q \mapsto c), k)\ w = i\}$),*simp*)

prefer *2* **apply** (*rule card-union-monotone*,*assumption*)
apply (*subgoal-tac*
  finite (*dom h*)
  ⟹ *card* ({*w* ∈ *dom h*. *w* = *p* ∧ *region* (*h*, *k*) *p* = *i*} ∪
        {*w* ∈ *dom h*. *w* ≠ *q* ∧ *w* ≠ *p* ∧ *region* (*h*, *k*) *w* = *i*}) =
    card ({*w* ∈ *dom h*. *w* = *p* ∧ *region* (*h*, *k*) *p* = *i*}) +
    card ({*w* ∈ *dom h*. *w* ≠ *q* ∧ *w* ≠ *p* ∧ *region* (*h*, *k*) *w* = *i*}),*simp*)
prefer *2* **apply** (*rule card-union-monotone-2*,*assumption*+)
apply (*subgoal-tac*
  ⟦ *h p* = *Some c* ⟧
  ⟹ *card* {*w*. *w*=*q* ∧ *region* (*h*(*p* := *None*)(*q* ↦ *c*), *k*) *q* = *i*} =
    card {*w*. *w* = *p* ∧ *region* (*h*, *k*) *p* = *i*},*simp*)
prefer *2*
apply (*subgoal-tac region* (*h*(*p* := *None*)(*q* ↦ *c*), *k*) *q* = *region* (*h*, *k*) *p*,*simp*)
apply (*case-tac region* (*h*, *k*) *p* = *i*,*simp*,*simp*)
apply (*simp add*: *region-def*)
apply (*subgoal-tac*
  {*w* ∈ *dom h*. *w* ≠ *q* ∧ *w* ≠ *p* ∧ *region* (*h*(*p* := *None*)(*q* ↦ *c*), *k*) *w* = *i*} =
  {*w* ∈ *dom h*. *w* ≠ *q* ∧ *w* ≠ *p* ∧ *region* (*h*, *k*) *w* = *i*},*simp*)
prefer *2* **apply** (*rule diff-monotone-aux-2*)
apply (*subgoal-tac p* ∈ *dom h*) **prefer** *2* **apply** (*erule domI*)
apply (*subgoal-tac*
  p ∈ *dom h*
  ⟹ {*w*. *w* = *p* ∧ *region* (*h*, *k*) *p* = *i*} =
    {*w* ∈ *dom h*. *w* = *p* ∧ *region* (*h*, *k*) *p* = *i*})
apply *simp*
by *blast*

lemma *diff-monotone*: ⟦ *finite* (*dom h*); *h p* = *Some c*; *SafeHeap.fresh q h* ⟧
    ⟹ []$_k$ = *diff k* (*h*, *k*) (*h*(*p* := *None*)(*q* ↦ *c*), *k*)
apply (*frule-tac h*=*h* **and** *k*=*k* **in** *diff-monotone-aux*,*assumption*+,*simp*)
by (*rule empty-diff*)

lemma *diff-IntT*:
  []$_k$ = *diff k* (*h*, *k*) (*h*, *k*)
by (*rule empty-diff*)

lemma *diff-BoolT*:
  []$_k$ = *diff k* (*h*, *k*) (*h*, *k*)
by (*rule empty-diff*)

lemma *diff-VarE*:
  []$_k$ = *diff k* (*h*, *k*) (*h*, *k*)
by (*rule empty-diff*)

lemma *diff-CopyE*:
  ⟦*m* = *SafeRASemantics.size h pa*;

$j <= k$;
$SafeHeap.copy\ (h,\ k)\ j\ pa = ((h',\ k),\ p')\ ⟧$
$\Longrightarrow [j \mapsto m] = diff\ k\ (h,\ k)\ (h',\ k)$
by (*frule A7*, *elim conjE*, *simp*)

lemma *diff-ReuseE*:
$⟦\ finite\ (dom\ h);\ h\ pa = Some\ c;\ SafeHeap.fresh\ q\ h\ ⟧$
$\Longrightarrow []_k = diff\ k\ (h,\ k)\ (h(pa := None)(q \mapsto c),\ k)$
by (*rule diff-monotone*,*assumption*+)

lemma *diff-Let1*:
$⟦(\delta 1,\ m1,\ s1) = (\delta,\ m,\ w);\ (\delta 2,\ m2,\ s2) = (\delta',\ ma,\ wa);$
$\delta = diff\ k\ (h,\ k)\ (h'\ |`\ \{p \in dom\ h'.\ fst\ (the\ (h'\ p)) \leq k\},\ k);$
$\delta' = diff\ k\ (h'\ |`\ \{p \in dom\ h'.\ fst\ (the\ (h'\ p)) \leq k\},\ k)\ (h'',\ k)⟧$
$\Longrightarrow \delta 1 \oplus \delta 2 = diff\ k\ (h,\ k)\ (h'',\ k)$
apply (*unfold diff-def*)
apply (*unfold addDelta-def*)
apply (*rule ext*) **apply** *simp-all*
apply *auto*
done

lemma *diff-let2-aux2-1*:
$\{w.\ w = p\ \wedge region\ (ha(p \mapsto (j,\ C,\ as)),\ ka)\ w = j\} =$
$\{w.\ w = p\ \wedge region\ ([p \mapsto (j,\ C,\ as)],\ ka)\ w = j\}$
apply *auto*
by (*simp-all add*: *region-def*)

lemma *diff-let2-aux2-2*:
$fresh\ p\ ha \Longrightarrow$
$\{w.\ w \neq p \wedge w \in dom\ ha \wedge region\ (ha(p \mapsto (j,\ C,\ as)),\ ka)\ w = j\} =$
$\{w.\ w \in dom\ ha \wedge region\ (ha,\ ka)\ w = j\}$
apply *auto*
apply (*simp-all add*: *region-def*)
apply (*simp add*: *fresh-def*)
apply (*simp add*: *dom-def*)
done

lemma *diff-let2-aux2-3*:
$⟦\ finite\ (dom\ ha);\ \ fresh\ p\ ha\ ⟧ \Longrightarrow$
$card\ (\{w \in dom\ ha.\ region\ (ha,\ ka)\ w = j\} \cup$
$\{w.\ w = p \wedge region\ ([p \mapsto (j,\ C,\ as)],\ ka)\ w = j\}) =$
$card\ (\{w \in dom\ ha.\ region\ (ha,\ ka)\ w = j\}) +$
$card\ (\{w.\ w = p \wedge region\ ([p \mapsto (j,\ C,\ as)],\ ka)\ w = j\})$
apply (*unfold region-def*)
apply (*unfold fresh-def*)
apply *auto*

done

lemma *diff-let2-aux2*:
⟦ *finite* (*dom ha*); *fresh p ha*⟧ ⟹
numCellsRegion (*ha*(*p* ↦ (*j*, *C*, *as*)), *ka*) *j* =
numCellsRegion (*ha*, *ka*) *j* + *numCellsRegion* (*empty*(*p* ↦ (*j*, *C*, *as*)), *ka*) *j*
apply (*unfold numCellsRegion-def*)
apply *auto*
apply (*subgoal-tac*
{*w*. (*w* = *p* ∨ *w* ∈ *dom ha*) ∧ *region* (*ha*(*p* ↦ (*j*, *C*, *as*)), *ka*) *w* = *j*} =
{*w*. *w* ≠ *p* ∧ *w* ∈ *dom ha* ∧ *region* (*ha*(*p* ↦ (*j*, *C*, *as*)), *ka*) *w* = *j*} ∪
{*w*. *w* = *p* ∧ *region* (*ha*(*p* ↦ (*j*, *C*, *as*)), *ka*) *w* = *j*})
prefer *2* **apply** *blast*
apply (*subgoal-tac*
{*w*. *w* = *p* ∧ *region* (*ha*(*p* ↦ (*j*, *C*, *as*)), *ka*) *w* = *j*} =
{*w*. *w* = *p* ∧ *region* ([*p* ↦ (*j*, *C*, *as*)], *ka*) *w* = *j*}) **prefer** *2* **apply** (*rule diff-let2-aux2-1*)
apply (*subgoal-tac*
{*w*. *w* ≠ *p* ∧ *w* ∈ *dom ha* ∧ *region* (*ha*(*p* ↦ (*j*, *C*, *as*)), *ka*) *w* = *j*} =
{*w*. *w* ∈ *dom ha* ∧ *region* (*ha*, *ka*) *w* = *j*}) **prefer** *2* **apply** (*rule diff-let2-aux2-2*,*assumption*+)
apply *simp*
apply (*subgoal-tac*
finite (*dom ha*) ⟹
card ({*w* ∈ *dom ha*. *region* (*ha*, *ka*) *w* = *j*} ∪
{*w*. *w* = *p* ∧ *region* ([*p* ↦ (*j*, *C*, *as*)], *ka*) *w* = *j*}) =
card ({*w* ∈ *dom ha*. *region* (*ha*, *ka*) *w* = *j*}) +
card ({*w*. *w* = *p* ∧ *region* ([*p* ↦ (*j*, *C*, *as*)], *ka*) *w* = *j*})) **prefer** *2* **apply** (*rule diff-let2-aux2-3*,*assumption*+)
apply *simp*
done

lemma *diff-let2-aux1*:
⟦ *finite* (*dom ha*); *fresh p ha*⟧ ⟹
numCellsRegion (*ha*(*p* ↦ (*j*,(*C*,*as*))),*ka*) *j* =
numCellsRegion (*ha*, *ka*) *j* + *1*
apply (*subgoal-tac*
numCellsRegion (*ha*(*p* ↦ (*j*, *C*, *as*)), *ka*) *j* =
numCellsRegion (*ha*, *ka*) *j* + *numCellsRegion* (*empty*(*p* ↦ (*j*, *C*, *as*)), *ka*) *j*,*simp*)
apply (*simp add*: *numCellsRegion-def*)
apply (*simp add*: *region-def*)
by (*rule diff-let2-aux2*)

lemma *diff-let2-aux4-1*:

⟦ *SafeHeap.fresh p ha*; $i \neq j$ ⟧ ⟹
$\{w.\ w \neq p \wedge w \in dom\ ha \wedge region\ (ha(p \mapsto (j, C, as)), ka)\ w = i\} =$
$\{w.\ w \in dom\ ha \wedge region\ (ha, ka)\ w = i\}$
apply *auto*
apply (*unfold region-def*) **apply** (*unfold SafeHeap.fresh-def*)
apply *auto*
done

lemma *diff-let2-aux4-2*:
⟦ *SafeHeap.fresh p ha*; $i \neq j$ ⟧ ⟹
$\{w.\ w = p \wedge region\ (ha(p \mapsto (j, C, as)), ka)\ w = i\} \cup$
$\{w \in dom\ ha.\ region\ (ha, ka)\ w = i\} =$
$\{w \in dom\ ha.\ region\ (ha, ka)\ w = i\}$
apply *auto*
apply (*unfold region-def*) **apply** (*unfold SafeHeap.fresh-def*)
apply *auto*
done

lemma *diff-let2-aux4*:
⟦ *finite* (*dom ha*); *SafeHeap.fresh p ha*; $j \leq ka$; $i \neq j$⟧ ⟹
numCellsRegion $(ha(p \mapsto (j, C, as)), ka)\ i$ = *numCellsRegion* $(ha, ka)\ i$
apply (*unfold numCellsRegion-def*)
apply *auto*
apply (*subgoal-tac*
$\{w.\ (w = p \vee w \in dom\ ha) \wedge region\ (ha(p \mapsto (j, C, as)), ka)\ w = i\} =$
$\{w.\ w = p \wedge region\ (ha(p \mapsto (j, C, as)), ka)\ w = i\} \cup$
$\{w.\ w \neq p \wedge w \in dom\ ha \wedge region\ (ha(p \mapsto (j, C, as)), ka)\ w = i\}$)
prefer *2* **apply** *blast*
apply *simp*
apply (*subgoal-tac*
$\{w.\ w \neq p \wedge w \in dom\ ha \wedge region\ (ha(p \mapsto (j, C, as)), ka)\ w = i\} =$
$\{w.\ w \in dom\ ha \wedge region\ (ha, ka)\ w = i\}$)
apply *simp*
apply (*subgoal-tac*
$\{w.\ w = p \wedge region\ (ha(p \mapsto (j, C, as)), ka)\ w = i\} \cup$
$\{w \in dom\ ha.\ region\ (ha, ka)\ w = i\} =$
$\{w \in dom\ ha.\ region\ (ha, ka)\ w = i\}$)
apply *simp*
apply (*rule diff-let2-aux4-2*,*assumption*+)
apply (*rule diff-let2-aux4-1*,*assumption*+)
done

lemma *diff-let2-aux2-1-otro*:
$\{w.\ w = p \wedge region\ (ha(p \mapsto (j, C, as)), ka)\ w = i\} =$
$\{w.\ w = p \wedge region\ ([p \mapsto (j, C, as)], ka)\ w = i\}$

apply *auto*
by (*simp-all add*: *region-def*)

lemma *diff-let2-aux2-2-otro*:
fresh p ha $\Longrightarrow$
$\{w.\ w \neq p \wedge w \in dom\ ha \wedge region\ (ha(p \mapsto (j,\ C,\ as)),\ ka)\ w = i\} =$
$\{w.\ w \in dom\ ha \wedge region\ (ha,\ ka)\ w = i\}$
apply *auto*
apply (*simp-all add*: *region-def*)
apply (*simp add*: *fresh-def*)
apply (*simp add*: *dom-def*)
apply (*simp add*: *fresh-def*)
apply *auto*
done

lemma *diff-let2-aux2-3-otro*:
$[\![$ *finite* $(dom\ ha);$ *fresh p ha*; $i \neq j\ ]\!] \Longrightarrow$
card $(\{w \in dom\ ha.\ region\ (ha,\ ka)\ w = i\} \cup$
$\{w.\ w = p \wedge region\ ([p \mapsto (j,\ C,\ as)],\ ka)\ w = i\}) =$
card $(\{w \in dom\ ha.\ region\ (ha,\ ka)\ w = i\}) +$
card $(\{w.\ w = p \wedge region\ ([p \mapsto (j,\ C,\ as)],\ ka)\ w = i\})$
apply (*unfold region-def*)
apply (*unfold fresh-def*)
apply *auto*
done

lemma *diff-let2-aux2-otro*:
$[\![$ *finite* $(dom\ ha)$; *fresh p ha*; $i \neq j]\!] \Longrightarrow$
numCellsRegion $(ha(p \mapsto (j,\ C,\ as)),\ ka)\ i =$
numCellsRegion $(ha,\ ka)\ i$ + *numCellsRegion* $(empty(p \mapsto (j,\ C,\ as)),\ ka)\ i$
apply (*unfold numCellsRegion-def*)
apply *auto*
apply (*subgoal-tac*
$\{w.\ (w = p \vee w \in dom\ ha) \wedge region\ (ha(p \mapsto (j,\ C,\ as)),\ ka)\ w = i\} =$
$\{w.\ w \neq p \wedge w \in dom\ ha \wedge region\ (ha(p \mapsto (j,\ C,\ as)),\ ka)\ w = i\} \cup$
$\{w.\ w = p\ \wedge region\ (ha(p \mapsto (j,\ C,\ as)),\ ka)\ w = i\}$)
prefer *2* **apply** *blast*
apply (*subgoal-tac*
$\{w.\ w = p\ \wedge region\ (ha(p \mapsto (j,\ C,\ as)),\ ka)\ w = i\} =$
$\{w.\ w = p\ \wedge region\ ([p \mapsto (j,\ C,\ as)],\ ka)\ w = i\})$ **prefer** *2* **apply** (*rule diff-let2-aux2-1-otro*)
apply (*subgoal-tac*
$\{w.\ w \neq p \wedge w \in dom\ ha \wedge region\ (ha(p \mapsto (j,\ C,\ as)),\ ka)\ w = i\} =$
$\{w.\ w \in dom\ ha \wedge region\ (ha,\ ka)\ w = i\}$) **prefer** *2* **apply** (*rule diff-let2-aux2-2-otro*,*assumption*+)
apply *simp*
apply (*subgoal-tac*
finite $(dom\ ha) \Longrightarrow$

card ({$w \in$ *dom ha*. *region* (*ha*, *ka*) $w = i$} $\cup$
{w. $w = p \wedge$ *region* ([$p \mapsto$ (j, C, *as*)], *ka*) $w = i$}) =
card ({$w \in$ *dom ha*. *region* (*ha*, *ka*) $w = i$}) +
card ({w. $w = p \wedge$ *region* ([$p \mapsto$ (j, C, *as*)], *ka*) $w = i$})) **prefer** *2* **apply** (*rule diff-let2-aux2-3-otro*,*assumption*+)
apply *simp*
done

lemma *diff-Let2*:
⟦ *finite* (*dom h*);
fresh pa h;
$j \leq k$;
(δ, m, s) = (δ', *ma*, w);
δ' = *diff k* (*h*(*pa* $\mapsto$ (j, C, *map* (*atom2val E1*) *as*)), *k*) (*h'*, *k*)⟧
$\Longrightarrow \delta \oplus [j \mapsto 1]$ = *diff k* (*h*, *k*) (*h'*, *k*)
apply (*unfold diff-def*)
apply (*rule ext*) **apply** *simp-all*
apply (*case-tac i <= k*)
apply (*unfold addDelta-def*)
apply *clarsimp*
apply (*case-tac i=j*)
apply (*subgoal-tac*
numCellsRegion (*h*(*pa* $\mapsto$ (j, C, *map* (*atom2val E1*) *as*)), *k*) *j* =
numCellsRegion (*h*, *k*) *j* + *1*)
prefer *2* **apply** (*rule diff-let2-aux1*,*assumption*+)
apply *auto*
apply (*subgoal-tac*
numCellsRegion (*h*(*pa* $\mapsto$ (j, C, *map* (*atom2val E1*) *as*)), *k*) *i* =
numCellsRegion (*h*, *k*) *i* + *numCellsRegion* (*empty*(*pa* $\mapsto$ (j, C, *map* (*atom2val E1*) *as*)), *k*) *i*)
prefer *2* **apply** (*rule diff-let2-aux2-otro*,*assumption*+) **apply** *simp*
apply (*simp add*: *numCellsRegion-def*)
apply (*simp add*: *region-def*)
done

lemma *cased-aux1*:
⟦*finite* (*dom h*); *h pa* = *Some* (j, C, *vs*); $j \leq k$; $i \leq k$; $i \neq j$⟧
$\Longrightarrow$ *numCellsRegion* (*h*(*pa* := *None*), *k*) *i* = *numCellsRegion* (*h*, *k*) *i*
apply (*unfold numCellsRegion-def*, *auto*)
apply (*subgoal-tac* {$w \in$ *dom h*. $w \neq pa \wedge$ *region* (*h*(*pa* := *None*), *k*) $w = i$} =
{$w \in$ *dom h*. *region* (*h*, *k*) $w = i$},*simp*)
by (*simp add*: *region-def*, *auto*)

lemma *cased-aux2-1*: *h pa* = *Some* (j, C, *vs*) $\Longrightarrow$ {$w \in$ *dom h*. w=*pa* $\wedge$ *region* (*h*,*k*) $w = j$} = {*pa*}
by (*simp add*: *region-def*,*auto*)

lemma *cased-aux2-2*:
⟦ *finite* (*dom h*); *h pa* = *Some* (*j*, *C*, *vs*) ⟧
⟹ *int* (*card* ({*w* ∈ *dom h*. *region* (*h*, *k*) *w* = *j*} − {*pa*})) = *int* (*card* {*w* ∈ *dom h*. *region* (*h*, *k*) *w* = *j*}) − *1*
apply (*subgoal-tac pa* ∈ {*w* ∈ *dom h*. *region* (*h*, *k*) *w* = *j*})
apply (*subst card-Diff-singleton*,*simp*,*simp*)
apply (*subgoal-tac card* {*w* ∈ *dom h*. *region* (*h*, *k*) *w* = *j*} >= *1*,*simp*)
apply (*subgoal-tac* ∃ *S'*. {*w* ∈ *dom h*. *region* (*h*, *k*) *w* = *j*} = *S'* ∪ {*pa*} ∧ *pa* ∉ *S'*)
apply (*erule exE*, *simp*) **apply** (*elim conjE*) **apply** (*subst card-insert-if*)
defer apply *simp*
apply (*rule-tac x*={*w* ∈ *dom h*. *w* ≠ *pa* ∧ *region* (*h*, *k*) *w* = *j*} **in** *exI*, *simp*)
apply (*simp add*: *region-def*, *auto*)
apply (*simp add*: *region-def*)
apply (*subgoal-tac S'* = {*w* ∈ *dom h*. *w* ≠ *pa* ∧ *region* (*h*, *k*) *w* = *j*},*simp*)
by (*simp add*: *region-def*,*auto*)

lemma *cased-aux2*:
⟦*finite* (*dom h*); *h pa* = *Some* (*j*, *C*, *vs*); *j* ≤ *k*⟧
⟹ *numCellsRegion* (*h*(*pa* := *None*), *k*) *j* = *numCellsRegion* (*h*, *k*) *j* − *1*
apply (*simp add*: *numCellsRegion-def*)
apply (*subgoal-tac* {*w* ∈ *dom h*. *w* ≠ *pa* ∧ *region* (*h*(*pa* := *None*), *k*) *w* = *j*} =
{*w* ∈ *dom h*. *region* (*h*, *k*) *w* = *j*} −
{*w* ∈ *dom h*. *w*=*pa* ∧ *region* (*h*,*k*) *w* = *j*},*simp*)
apply (*subst cased-aux2-1*,*assumption*+)
apply (*rule cased-aux2-2*, *assumption*+)
by (*simp add*: *region-def*,*auto*)

lemma *diff-CaseD*:
⟦ *finite* (*dom h*);
h pa = *Some* (*j*, *C*, *vs*);
(*δ*, *m*, *s*) = (*δ'*, *ma*, *w*); *j* <= *k*;
δ' = *diff k* (*h*(*pa* := *None*), *k*) (*h'*, *k*)⟧
⟹ *δ* ⊕ [*j* ↦ −*1*] = *diff k* (*h*, *k*) (*h'*, *k*)
apply (*unfold diff-def*)
apply (*rule ext*, *simp-all*)
apply (*case-tac i* <= *k*)
apply (*unfold addDelta-def*, *clarsimp*)
apply (*case-tac i*=*j*)
apply (*subgoal-tac numCellsRegion* (*h*(*pa*:=*None*), *k*) *j* = *numCellsRegion* (*h*, *k*) *j* − *1*, *auto*)
apply (*rule cased-aux2*, *assumption*+)
by (*rule cased-aux1*, *assumption*+)

lemma *diff-App1*:
diff k (*h*, *k*) (*h'* |` {*p* ∈ *dom h'*. *fst* (*the* (*h' p*)) ≤ *k*}, *k*) = *diff k* (*h*, *k*) (*h'*, *k*)

apply (*unfold diff-def*)
apply (*rule ext*, *simp*, *rule impI*)
apply (*unfold numCellsRegion-def*, *simp add*: *restrict-map-def*, *unfold region-def*)

apply (*case-tac* ($\lambda x.$ *if* $x \in dom\ h' \wedge fst\ (the\ (h'\ x)) \leq k$ *then* $h'\ x$ *else None*, k),*auto*)
apply (*subgoal-tac* $i \leq k \Longrightarrow$
$\{w.\ (w \in dom\ h' \wedge fst\ (the\ (h'\ w)) \leq k \longrightarrow$
$w \in dom\ (\lambda x.\ if\ x \in dom\ h' \wedge fst\ (the\ (h'\ x)) \leq k\ then\ h'\ x\ else\ None) \wedge$
$option\text{-}case\ undefined\ (prod\text{-}case\ (\lambda r.\ prod\text{-}case\ (\lambda c\ xs.\ r)))\ (h'\ w) = i) \wedge$
$((w \in dom\ h' \longrightarrow \neg\ fst\ (the\ (h'\ w)) \leq k) \longrightarrow$
$w \in dom\ (\lambda x.\ if\ x \in dom\ h' \wedge fst\ (the\ (h'\ x)) \leq k\ then\ h'\ x\ else\ None) \wedge undefined = i)\} =$
$\{w \in dom\ h'.\ option\text{-}case\ undefined\ (prod\text{-}case\ (\lambda r.\ prod\text{-}case\ (\lambda c\ xs.\ r)))\ (h'\ w) = i\}$, *simp*, *auto*)
apply (*split split-if-asm*)
apply (*erule-tac* x=a **in** *allE*, *erule-tac* x=aa **in** *allE*, *erule-tac* x=b **in** *allE*, *simp*)
apply *simp*
apply (*split split-if-asm*)
apply (*erule-tac* x=a **in** *allE*, *erule-tac* x=aa **in** *allE*, *erule-tac* x=b **in** *allE*, *simp*)
by (*simp*, *split split-if-asm*,*simp*,*simp*)

lemma *diff-App2*:
diff k (h,k) $(h',k) = (diff\ (k+1)\ (h,k+1)\ (h',k+1))(k + 1 := None)$
apply (*unfold diff-def*)
apply (*rule ext*)
apply *simp*
apply (*rule impI*) **apply** (*rule impI*)
apply (*subgoal-tac numCellsRegion* $(h, Suc\ k)\ i = numCellsRegion\ (h, k)\ i$,*simp*)
apply (*subgoal-tac numCellsRegion* $(h', k)\ i = numCellsRegion\ (h', Suc\ k)\ i$,*simp*)
apply (*simp add*: *numCellsRegion-def add*: *region-def*)
by (*simp add*: *numCellsRegion-def add*: *region-def*)

lemma *diff-App*:
$[\![(\delta, m, s) = (\delta', ma, w);$
$\delta' = diff\ (k + 1)\ (h, k + 1)\ (h', k + 1)]\!]$
$\Longrightarrow \delta(k + 1 := None) = diff\ k\ (h, k)\ (h' |` \{p \in dom\ h'.\ fst\ (the\ (h'\ p)) \leq k\}, k)$
apply *simp*
apply (*subgoal-tac diff* k (h,k) $(h',k) = (diff\ (k+1)\ (h,k+1)\ (h',k+1))(k + 1 := None)$,*simp*)
apply (*subgoal-tac diff* $k\ (h, k)\ (h' |` \{p \in dom\ h'.\ fst\ (the\ (h'\ p)) \leq k\}, k) =$

diff k (h, k) (h', k),simp)
apply (*rule diff-App1*)
by (*rule diff-App2*)

end

13 Certification of the translation CoreSafe to SVM. maxFreshCells property

theory *maxFreshCells-lemmas*
imports *Main ../CoreSafe/SafeRASemantics SVMSemantics CoreSafeToSVM CertifSafeToSVM-definitions basic-properties*
begin

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

lemma *sizeH-monotone*:
finite (dom h) $\Longrightarrow$
sizeH (h |` {p ∈ dom h. fst (the (h p)) ≤ k0}, k0) − sizeH (h, k) <= 0
apply (*simp add*: *sizeH.simps*)
apply (*subgoal-tac dom h ∩ {p ∈ dom h. fst (the (h p)) ≤ k0} ⊆ dom h*)
apply (*frule card-mono*, *simp*,*clarsimp*)
by *blast*

lemma *sizeH-add-loc*:
⟦ *finite (dom h)*; *fresh pa h*; *(h(pa ↦ (j, C, map (atom2val E1) as)), k) = (h', k) ↓ k0*⟧
$\Longrightarrow$ *sizeH ((h', k) ↓ k0) − sizeH (h, k) = 1*
apply (*drule-tac t=(h', k) ↓ k0* **in** *sym*)
by (*simp add*: *fresh-def add*: *sizeH.simps*)

lemma *sizeH-restrictToRegion-monotone*:
finite (dom h) $\Longrightarrow$
sizeH ((h, k) ↓ k0) <= sizeH (h, k0)
apply (*unfold restrictToRegion.simps*, *unfold Let-def*)
apply (*simp add*: *sizeH.simps*)
apply (*subgoal-tac dom h ∩ {p ∈ dom h. fst (the (h p)) ≤ k0} ⊆ dom h*)
apply (*frule card-mono*, *simp*,*clarsimp*)
by *blast*

lemma *sizeH-upd-same*:
⟦*h pa = Some c*;

finite (*dom h*);
SafeHeap.fresh q h ⟧
⟹ *sizeH* (*h*(*pa* := *None*)(*q* ↦ *c*), *k*) = *sizeH* (*h*, *k'*)
apply (*simp add*: *sizeH.simps*)
apply (*simp add*: *SafeHeap.fresh-def*)
apply (*subgoal-tac h pa* = *Some c* ⟹ *pa* ∈ *dom h*)
apply (*rule card-Suc-Diff1*, *assumption*+)
by *auto*

lemma *restrictToRegion-monotone*:
(*h'* |` {*p* ∈ *dom h'*. *fst* (*the* (*h' p*)) ≤ *k*}, *k*) ↓ *k* = (*h'* |` {*p* ∈ *dom h'*. *fst* (*the* (*h'* *p*)) ≤ *k*}, *k*)
apply (*unfold restrictToRegion.simps*, *unfold Let-def*, *auto*)
apply (*subgoal-tac*
{*p* ∈ *dom h'*. *fst* (*the* (*h' p*)) ≤ *k* ∧ *fst* (*the* ((*h'* |` {*p* ∈ *dom h'*. *fst* (*the* (*h' p*)) ≤ *k*}) *p*)) ≤ *k*} =
{*p* ∈ *dom h'*. *fst* (*the* (*h' p*)) ≤ *k*})
apply (*simp*, *auto*)
apply (*subgoal-tac x* ∈ *dom h'*, *clarsimp*)
by (*simp add*: *dom-def*)

lemma *restrictToRegion-monotone-Suc-k*:
k0 <= *k* ⟹ (*h'*, *Suc k*) ↓ *k0* = (*h'* |` {*p* ∈ *dom h'*. *fst* (*the* (*h' p*)) ≤ *k*}, *k*) ↓ *k0*
apply (*unfold restrictToRegion.simps*, *unfold Let-def*,*clarsimp*)
apply (*subgoal-tac*
{*p* ∈ *dom h'*. *fst* (*the* (*h' p*)) ≤ *k*} ∩
{*p* ∈ *dom h'*. *fst* (*the* (*h' p*)) ≤ *k* ∧ *fst* (*the* ((*h'* |` {*p* ∈ *dom h'*. *fst* (*the* (*h' p*)) ≤ *k*}) *p*)) ≤ *k0*} =
{*p* ∈ *dom h'*. *fst* (*the* (*h' p*)) ≤ *k0*},*simp*)
apply *auto*
by (*subgoal-tac x* ∈ *dom h'*, *clarsimp*, *simp add*: *dom-def*)+

lemma *max-assoc*:
max (*max* (*x*::*int*) (*y*::*int*)) (*z*::*int*) = *max* (*x*::*int*) (*max* (*y*::*int*) (*z*::*int*))
by *simp*

lemma *max-assoc-2*:
max (*x*::*int*) (*max* (*y*::*int*) (*z*::*int*)) = *max* (*max* (*x*::*int*) (*y*::*int*)) (*z*::*int*)
by *simp*

lemma *max-sym*:
max (*x*::*int*) (*y*::*int*) = *max* (*y*::*int*) (*x*::*int*)
by *simp*

lemma *max-sub*:
max ((*x*::*int*)−(*z*::*int*)) ((*y*::*int*)−(*z*::*int*)) = *max* (*x*::*int*) (*y*::*int*) − (*z*::*int*)

by *auto*

lemma *max-sub-2*:
max (*max* (*x*::*int*) (*y*::*int*)) (*z*::*int*) = *max* (*max* (*z*::*int*) (*x*::*int*)) (*y*::*int*)
by *auto*

lemma *foldl-max-assoc*:
foldl max (*max* (*x*::*int*) (*y*::*int*)) *xs* = *max* (*x*::*int*) (*foldl max* (*y*::*int*) *xs*)
by (*induct xs arbitrary*: *y*) (*simp-all add*: *max-assoc*)

lemma *foldl-max-assoc-2*:
foldl max (*max* (*x*::*int*) (*y*::*int*)) *xs* = *max* (*y*::*int*) (*foldl max* (*x*::*int*) *xs*)
by (*induct xs arbitrary*: *y*) (*simp-all add*: *max-assoc*)

lemma *foldl-max-absorb0*:
shows *max* (*x*::*int*) (*foldl max* (*y*::*int*) *zs*) = *foldl max* (*y*::*int*) ((*x*::*int*)#*zs*)
by (*induct zs*) (*simp-all add*:*foldl-max-assoc*)

lemma *maximum-map-sub-sizeH*:
maximum (*map* (λ(*h*, *r*, *pc*, *s*). *sizeH h* − *sizeH x*) (*y*#*ys*)) =
(*maximum* (*map* (λ(*h*, *r*, *pc*, *s*). *sizeH h*) (*y*#*ys*))) − *sizeH x*
apply (*induct ys*)
apply (*unfold maximum.simps*, *case-tac y*, *case-tac b*, *case-tac ba*, *simp*)
apply (*case-tac y*, *case-tac b*, *case-tac ba*, *rename-tac hy bay ry bayy pcy sy*, *simp*)
apply (*case-tac a*, *case-tac b*, *case-tac ba*, *rename-tac ha ba ra baa pca sa*, *simp*)
apply (*subst foldl-max-assoc-2*)
apply (*subst foldl-max-assoc-2*)
by *simp*

lemma *maximum-map-sub-sizeST*:
maximum (*map* (λ(*h*, *r*, *pc*, *w*). *sizeST w* − *sizeST x*) (*y*#*ys*)) =
(*maximum* (*map* (λ(*h*, *r*, *pc*, *w*). *sizeST w*) (*y*#*ys*))) − *sizeST x*
apply (*induct ys*)
apply (*unfold maximum.simps*, *case-tac y*, *case-tac b*, *case-tac ba*, *simp*)
apply (*case-tac y*, *case-tac b*, *case-tac ba*, *rename-tac hy bay ry bayy pcy sy*, *simp*)
apply (*case-tac a*, *case-tac b*, *case-tac ba*, *rename-tac ha ba ra baa pca sa*, *simp*)
apply (*subst foldl-max-assoc-2*)
apply (*subst foldl-max-assoc-2*)
by *simp*

lemma *max-maximum-sub*:
max ((*maximum* (*map f* (*x*#*xs*))) − *z*) ((*maximum* (*map f* (*y*#*ys*))) − *z*) =
maximum (*map f* ((*x*#*xs*) @ (*y*#*ys*))) − *z*
apply (*induct xs ys rule*:*list-induct2′*)

apply (*simp add*: *maximum.simps*)

apply (*simp add*: *maximum.simps*)

apply (*simp add*: *maximum.simps*)
apply (*subst max-sub*, *subst foldl-max-absorb0*)
apply (*simp*, *subst max-sub-2*)
apply (*simp add*: *maximum.simps*)

apply (*simp add*: *maximum.simps*)
apply (*subst foldl-max-assoc-2* [**where** *y*=(*f xa*)])
apply (*subst max-sub*, *subst max-assoc* , *subst foldl-max-assoc-2*)
apply (*subst max-assoc-2* [**where** *y*=*f ya*])
apply (*subst max-sym* [**where** *y*=*f ya*])
apply (*subst max-assoc*, *subst max-assoc-2*)

apply (*subst foldl-max-assoc-2* [**where** *y*=*f ya*])
apply (*subst foldl-max-assoc-2* [**where** *y*= *f xa*])
apply (*subst max-assoc* [**where** *x*=*f xa* **and** *z*=*f y*])
apply (*subst foldl-max-assoc* [**where** *x*= *f xa*])
apply (*subst max-assoc-2* [**where** *y*=*f xa*])
apply (*subst max-sym*)
by *auto*

lemma *maximum-map-append*:
maximum (*map f* (*xs* @ *x* # *x* # *ys*)) = *maximum* (*map f* (*xs* @ *x* # *ys*))
apply (*induct xs ys rule*:*list-induct2*′)
by (*simp add*: *maximum.simps*)+

declare *map.simps* [*simp del*]
declare *maxFreshCells.simps* [*simp del*]
declare *maximum.simps* [*simp del*]

lemma *maxFreshCells2-P*:
maxFreshCells (*ss1* @ [*s*] @ *ss2*) =
max (*maxFreshCells* (*ss1* @ [*s*]))
((*maxFreshCells* ([*s*] @ *ss2*)) + *sizeSVMStateH s* − *sizeSVMStateH* (*hd* (*ss1* @ [*s*])))
apply (*case-tac s*, *case-tac b*, *case-tac ba*) **apply** (*rename-tac hi bi ri bai pci si*)
apply (*induct ss1 ss2 rule*:*list-induct2*′)

apply (*simp add*: *maxFreshCells.simps*)

apply (*simp add*: *maxFreshCells.simps*)
apply (*simp add*: *maximum.simps*)
apply (*case-tac x*, *case-tac b*, *case-tac ba*) **apply** (*rename-tac hx bx rx bax pcx sx*) **apply** *simp*
apply (*simp add*: *maxFreshCells.simps add*: *maximum.simps add*: *map.simps*)

apply (*simp*, *simp add*: *maxFreshCells.simps add*: *maximum.simps add*: *map.simps*)
apply (*subst foldl-max-absorb0*)
apply (*subgoal-tac 0 = sizeH hi − sizeH hi*,*simp*)
apply (*subst max-assoc*, *subst max-sym* [*of - 0*])
apply (*subst max-assoc-2*, *simp*,*simp*)

apply *simp*
apply (*case-tac x*, *case-tac b*, *case-tac ba*) **apply** (*rename-tac hx bx rx bax pcx sx*) **apply** *simp*
apply (*simp add*: *maxFreshCells.simps*)
apply (*case-tac y*, *case-tac b*, *case-tac ba*) **apply** (*rename-tac hy bby ry bay pcy sy*) **apply** *simp*
apply (*subgoal-tac 0 = sizeH hx − sizeH hx*) **prefer** *2* **apply** *simp*
apply (*subst maximum-map-sub-sizeH* [**where** *ys*=*xs* @ (*hi*, *ri*, *pci*, *si*) # (*hy*, *ry*, *pcy*, *sy*) # *ys*])
apply (*subst maximum-map-sub-sizeH* [**where** *ys*=*xs* @ [(*hi*, *ri*, *pci*, *si*)]])
apply (*subst maximum-map-sub-sizeH* [**where** *ys*=(*hy*, *ry*, *pcy*, *sy*) # *ys*])
apply (*simp*,*subst max-maximum-sub*,*simp*,*rule sym*)
apply (*subgoal-tac*
maximum ((*map* (λ(*h*, *r*, *pc*, *s*). *sizeH h*) (((*hx*, *rx*, *pcx*, *sx*) # *xs*) @ (*hi*, *ri*, *pci*, *si*) #
((*hi*, *ri*, *pci*, *si*) # (*hy*, *ry*, *pcy*, *sy*) # *ys*)))) =
maximum ((*map* (λ(*h*, *r*, *pc*, *s*). *sizeH h*) (((*hx*, *rx*, *pcx*, *sx*) # *xs*) @
((*hi*, *ri*, *pci*, *si*) # (*hy*, *ry*, *pcy*, *sy*) # *ys*)))),*simp*)
by (*rule maximum-map-append*)

lemma *maxFreshCells-P-rev*:
maxFreshCells (*rev* (*ss2* @ [*s*] @ *ss1*)) =
max (*maxFreshCells* (*rev* ([*s*] @ *ss1*)))
((*maxFreshCells* (*rev* (*ss2* @ [*s*]))) + *sizeSVMStateH s* − *sizeSVMStateH* (*hd* (*rev* ([*s*] @ *ss1*))))
apply (*case-tac s*, *case-tac b*, *case-tac ba*) **apply** (*rename-tac hi bi ri bai pci si*, *simp*)
apply (*induct* (*rev ss1*) (*rev ss2*) *rule*:*list-induct2'*)

apply (*simp add*: *maxFreshCells.simps*)

apply (*simp add*: *maxFreshCells.simps*)
apply (*simp add*: *maximum.simps*)
apply (*case-tac x*, *case-tac b*, *case-tac ba*) **apply** (*rename-tac hx bx rx bax pcx sx*) **apply** *simp*
apply (*simp add*: *maxFreshCells.simps add*: *maximum.simps add*: *map.simps*)

apply (*simp*, *simp add*: *maxFreshCells.simps add*: *maximum.simps add*: *map.simps*)
apply (*subst foldl-max-absorb0*)
apply (*subgoal-tac 0 = sizeH hi − sizeH hi*,*simp*)
apply (*subst max-assoc*, *subst max-sym* [*of - 0*])
apply (*subst max-assoc-2*, *simp*,*simp*)

apply *simp*
apply (*case-tac x*, *case-tac b*, *case-tac ba*) **apply** (*rename-tac hx bx rx bax pcx sx*) **apply** *simp*
apply (*simp add*: *maxFreshCells.simps*)
apply (*case-tac y*, *case-tac b*, *case-tac ba*) **apply** (*rename-tac hy bby ry bay pcy sy*) **apply** *simp*
apply (*subgoal-tac 0 = sizeH hx − sizeH hx*) **prefer** *2* **apply** *simp*
apply (*subst maximum-map-sub-sizeH* [**where** *ys=xs* @ (*hi*, *ri*, *pci*, *si*) # (*hy*, *ry*, *pcy*, *sy*) # *ys*])
apply (*subst maximum-map-sub-sizeH* [**where** *ys=xs* @ [(*hi*, *ri*, *pci*, *si*)]])
apply (*subst maximum-map-sub-sizeH* [**where** *ys=*(*hy*, *ry*, *pcy*, *sy*) # *ys*])
apply (*simp*,*subst max-maximum-sub*,*simp*,*rule sym*)
apply (*subgoal-tac*
maximum ((*map* (λ(*h*, *r*, *pc*, *s*). *sizeH h*) (((*hx*, *rx*, *pcx*, *sx*) # *xs*) @ (*hi*, *ri*, *pci*, *si*) #
((*hi*, *ri*, *pci*, *si*) # (*hy*, *ry*, *pcy*, *sy*) # *ys*)))) =
maximum ((*map* (λ(*h*, *r*, *pc*, *s*). *sizeH h*) (((*hx*, *rx*, *pcx*, *sx*) # *xs*) @
((*hi*, *ri*, *pci*, *si*) # (*hy*, *ry*, *pcy*, *sy*) # *ys*)))),*simp*)
by (*rule maximum-map-append*)

lemma *maxFreshCells-IntT*:
⟦*finite* (*dom h*);
s0 = ((*h*, *k*), *k0*, (*q*, *j*), *S*)⟧
⟹ *0* = *maxFreshCells*
(*rev* (((*h*, *k*) ↓ *k0*, *k0*, (*q*, *Suc* (*Suc* (*Suc j*))), *Val* (*IntT i*) # *S′*) #
[((*h*, *k*), *k0*, (*q*, *j* + *2*), *Val* (*IntT i*) # *drop* (*topDepth rho*) *S*), ((*h*, *k*), *k0*, (*q*, *j* + *1*), *Val* (*IntT i*) # *S*),
((*h*, *k*), *k0*, (*q*, *j*), *S*)]))
apply (*simp add*: *maxFreshCells.simps*)
apply (*simp add*: *Let-def add*: *map.simps add*: *maximum.simps*)
apply (*subgoal-tac sizeH* (*h* |‘ {*p* ∈ *dom h*. *fst* (*the* (*h p*)) ≤ *k0*}, *k0*) − *sizeH* (*h*, *k*) <= *0*)
apply *simp*
by (*erule sizeH-monotone*)

lemma *maxFreshCells-BoolT*:
⟦*finite* (*dom h*);
s0 = ((*h*, *k*), *k0*, (*q*, *j*), *S*)⟧
⟹ *0* = *maxFreshCells*
(*rev* (((*h*, *k*) ↓ *k0*, *k0*, (*q*, *Suc* (*Suc* (*Suc j*))), *Val* (*BoolT b*) # *S′*) #
[((*h*, *k*), *k0*, (*q*, *j* + *2*), *Val* (*BoolT b*) # *drop* (*topDepth rho*) *S*), ((*h*, *k*), *k0*, (*q*, *j* + *1*), *Val* (*BoolT b*) # *S*),
((*h*, *k*), *k0*, (*q*, *j*), *S*)]))
apply (*simp add*: *maxFreshCells.simps*)
apply (*simp add*: *Let-def add*: *map.simps add*: *maximum.simps*)
apply (*subgoal-tac sizeH* (*h* |‘ {*p* ∈ *dom h*. *fst* (*the* (*h p*)) ≤ *k0*}, *k0*) − *sizeH* (*h*, *k*) <= *0*)

apply *simp*
by (*erule sizeH-monotone*)

lemma *maxFreshCells-VarE*:
⟦*finite* (*dom h*);
s0 = ((*h*, *k*), *k0*, (*q*, *j*), *S*)⟧
⟹ *0* = *maxFreshCells*
(*rev* (((*h*, *k*) ↓ *k0*, *k0*, (*q*, *Suc* (*Suc* (*Suc j*))), *Val* (*Loc pa*) # *S'*) #
[((*h*, *k*), *k0*, (*q*, *j* + *2*), *Val* (*Loc pa*) # *drop* (*topDepth rho*) *S*), ((*h*, *k*), *k0*, (*q*, *j* + *1*), *Val* (*Loc pa*) # *S*),
((*h*, *k*), *k0*, (*q*, *j*), *S*)]))
apply (*simp add*: *maxFreshCells.simps*)
apply (*simp add*: *Let-def add*: *map.simps add*: *maximum.simps*)
apply (*subgoal-tac sizeH* (*h* |` {*p* ∈ *dom h*. *fst* (*the* (*h p*)) ≤ *k0*}, *k0*) − *sizeH* (*h*, *k*) <= *0*)
apply *simp*
by (*erule sizeH-monotone*)

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

lemma *maxFreshCells-CopyE*:
⟦*E1 x* = *Some* (*Loc pa*); *E2 r* = *Some j*; *j* ≤ *k*;
0 < *m*; *SafeHeap.copy* (*h*, *k*) *j pa* = ((*h'*, *k*), *p'*); *m* = *SafeRASemantics.size h pa*;
finite (*dom h*); *finite* (*dom h'*);
extends h h'; *SafeRASemantics.size h' p'* = *SafeRASemantics.size h pa*;
sizeH (*h'*, *k*) − *sizeH* (*h*, *k*) = *SafeRASemantics.size h' p'*⟧
⟹ *m* = *maxFreshCells*
(*rev* [((*h'*, *k*) ↓ *k0*, *k0*, (*q*, *Suc* (*Suc* (*Suc* (*Suc ja*)))), *Val* (*Loc p'*) # *S'*),
((*h'*, *k*), *k0*, (*q*, *Suc* (*Suc* (*Suc ja*))), *Val* (*Loc p'*) # *drop* (*topDepth rho*) *S*),
((*h'*, *k*), *k0*, (*q*, *Suc* (*Suc ja*)), *Val* (*Loc p'*) # *S*), ((*h*, *k*), *k0*, (*q*, *Suc ja*), *Val* (*Loc pa*) # *Reg j* # *S*),
((*h*, *k*), *k0*, (*q*, *ja*), *S*)])
apply (*simp add*: *maxFreshCells.simps*)
apply (*simp add*: *map.simps add*: *maximum.simps*)
apply (*subgoal-tac sizeH* ((*h'*, *k*) ↓ *k0*) <= *sizeH* (*h'*, *k0*))
apply (*subgoal-tac sizeH* (*h'*, *k0*) = *sizeH* (*h'*, *k*),*simp*)
apply (*simp add*: *sizeH.simps*)
by (*rule sizeH-restrictToRegion-monotone*,*assumption*+)

lemma *maxFreshCells-ReuseE*:
⟦*h pa* = *Some c*;
finite (*dom h*);
SafeHeap.fresh q h ⟧
⟹ *0* = *maxFreshCells*

(*rev* [((*h*(*pa* := *None*)(*q* ↦ *c*), *k*) ↓ *k0*, *k0*, (*qa*, *Suc* (*Suc* (*Suc* (*Suc* *j*)))), *Val* (*Loc* *q*) # *S'*),
((*h*(*pa* := *None*)(*q* ↦ *c*), *k*), *k0*, (*qa*, *Suc* (*Suc* (*Suc* *j*))), *Val* (*Loc* *q*) # *drop* (*topDepth* *rho*) *S*),
((*h*(*pa* := *None*)(*q* ↦ *c*), *k*), *k0*, (*qa*, *Suc* (*Suc* *j*)), *Val* (*Loc* *q*) # *S*), ((*h*, *k*), *k0*, (*qa*, *Suc* *j*), *Val* (*Loc* *pa*) # *S*),
((*h*, *k*), *k0*, (*qa*, *j*), *S*)])
apply (*subgoal-tac* *sizeH* (*h*(*pa* := *None*)(*q* ↦ *c*), *k*) = *sizeH* (*h*, *k*))
apply (*simp* *add*: *maxFreshCells.simps* *add*: *map.simps* *add*: *maximum.simps*)
apply (*subgoal-tac* *sizeH* ((*h*(*pa* := *None*)(*q* ↦ *c*), *k*) ↓ *k0*) <= *sizeH* (*h*(*pa* := *None*)(*q* ↦ *c*), *k0*))
apply (*subgoal-tac* *sizeH* (*h*(*pa* := *None*)(*q* ↦ *c*), *k0*) = *sizeH* (*h*, *k*),*auto*)
apply (*rule* *sizeH-upd-same*, *assumption*+)
apply (*subgoal-tac* *finite* (*dom* (*h*(*pa* := *None*)(*q* ↦ *c*))))
apply (*rule* *sizeH-restrictToRegion-monotone*,*assumption*+)
apply (*simp* *add*: *SafeHeap.fresh-def*)
by (*rule* *sizeH-upd-same*,*assumption*+)

lemma *cons-append-assoc*:
x#(*xs* @ *ys*) = (*x*#*xs*) @ *ys*
by *simp*

lemma *cons-append-assoc-2*:
(*x*#*xs*) @ *ys* = *x*#(*xs* @ *ys*)
by *simp*

lemma *cons-append*:
x#*xs* = [*x*] @ *xs*
by *simp*

lemma *cons-append-2*:
[*x*] @ *xs* = *x* # *xs*
by *simp*

lemma *append-assoc-2*:
(*xs* @ *ys*) @ *zs* = *xs* @ (*ys* @ *zs*)
by (*induct* *xs*) *auto*

lemma *cons-cons-append*:
[*x*,*y*] = [*x*] @ [*y*]
by *simp*

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

lemma *head-execSVMBalanced*:
P⊢*s0* , *n* −*svm*→ *s* # *ss*, *m* ⟹ *s0* = *last* (*s*#*ss*)
by (*erule* *execSVMBalanced.induct*, *simp-all*)

lemma *head-cons-execSVMBalanced*:
⟦ *ss*≠[]; *s0* = *last ss*⟧ ⟹ ∃ *ss'*. *ss* = *ss'* @ [*s0*]
by (*rule-tac x*=*butlast ss* **in** *exI*, *simp*)

lemma *let1-1*:
⟦ *P*⊢((*h*, *k*), *k*, (*q1*, *j* + *1*), *Cont* (*k0*, *q2*) # *S*) , *0*#*td*#*tds* −*svm*→ *s* # *ss* , *Suc 0*#*td*#*tds*;
s = ((*h'* |` {*p* ∈ *dom h'*. *fst* (*the* (*h'* *p*)) ≤ *k*}, *k*) ↓ *k*, *k*, (*q'*, *i*), *Val v1* # *drop 0* (*Cont* (*k0*, *q2*) # *S*));
(*δ1*, *m1*, *s1*) = (*δ*, *m*, *w*);
m = *maxFreshCells* (*rev* (*s* # *ss*));
m >= *0*⟧
⟹ *maxFreshCells*
(*rev* ([((((*h'* |` {*p* ∈ (*dom h'*). ((*fst* (*the* (*h'* *p*))) ≤ *k*)}), *k*) ↓ *k*), *k*, (*q'*, *i*),
((*Val v1*) # (*drop 0* ((*Cont* (*k0*, *q2*)) # *S*))))] @
(*ss* @ [((*h*, *k*), *k0*, (*q*, *j*), *S*)]))) = *m1*
apply (*frule-tac s*=*s* **in** *head-execSVMBalanced*)
apply (*case-tac ss*=[],*simp*,*simp*)
apply (*frule head-cons-execSVMBalanced*,*assumption*+)
apply (*erule exE*)
apply (*drule-tac t*=*last ss* **in** *sym*)
apply (*rule-tac t*=*ss* **and** *s*=*ss'* @ [((*h*, *k*), *k*, (*q1*, *Suc j*), *Cont* (*k0*, *q2*) # *S*)] **in** *ssubst*,*assumption*)
apply (*subst append-assoc-2*)
apply (*subst cons-append-assoc* [*of* - *ss'*])
apply (*subst maxFreshCells-P-rev*, *simp*)
apply (*unfold rev.simps*, *simp*)
by (*unfold maxFreshCells.simps maximum.simps map.simps*,*simp*)

lemma *let1-2*:
⟦ *P*⊢((*h'* |` {*p* ∈ *dom h'*. *fst* (*the* (*h'* *p*)) ≤ *k*}, *k*), *k0*, (*q2*, *0*), *Val v1* # *S*) , (*td* + *1*)#*tds* −*svm*→ *sa* # *ssa* , *Suc 0*#*tds*;
sa = ((*h''*, *k*) ↓ *k0*, *k0*, (*q'a*, *ia*), *Val v2* # *S'*);
(*δ1*, *m1*, *s1*) = (*δ*, *m*, *w*);
(*δ2*, *m2*, *s2*) = (*δ'*, *ma*, *wa*); *ma* >= *0*;
δ = *diff k* (*h*, *k*) (*h'* |` {*p* ∈ *dom h'*. *fst* (*the* (*h'* *p*)) ≤ *k*}, *k*);
ma = *maxFreshCells* (*rev* (*sa* # *ssa*))⟧
⟹ *maxFreshCells* (*rev* (*sa* # *ssa* @
[((*h'* |` {*p* ∈ *dom h'*. *fst* (*the* (*h'* *p*)) ≤ *k*}, *k*) ↓ *k*, *k*, (*q'*, *i*), *Val v1* # *drop 0* (*Cont* (*k0*, *q2*) # *S*))])) +
sizeSVMStateH (((*h'* |` {*p* ∈ *dom h'*. *fst* (*the* (*h'* *p*)) ≤ *k*}, *k*) ↓ *k*, *k*, (*q'*,

i), *Val v1 # drop 0 (Cont (k0, q2) # S))) −*
sizeSVMStateH (((h, k), k0, (q, j), S)) =
m2 + ‖ δ1 ‖
apply (*frule-tac s=sa* **in** *head-execSVMBalanced*)
apply (*case-tac ssa=[],simp*) **apply** (*simp add*: *rev.simps add*: *maxFreshCells.simps add*: *maximum.simps add*: *map.simps*)
apply (*elim conjE*)
apply (*drule-tac t= (h'', k) ↓ k0* **in** *sym*, *simp*)
apply (*subgoal-tac*
sizeH (h' |` {p ∈ dom h'. fst (the (h' p)) ≤ k}, k) − sizeH ((h' |` {p ∈ dom h'. fst (the (h' p)) ≤ k}, k) ↓ k) = 0,simp)
apply (*subst restrictToRegion-monotone [of - k]*)
apply (*rule let1-3*, *simp*)
apply (*rule sym*)
apply (*subgoal-tac (h' |` {p ∈ dom h'. fst (the (h' p)) ≤ k}, k) ↓ k = (h' |` {p ∈ dom h'. fst (the (h' p)) ≤ k}, k),simp*)
apply (*rule restrictToRegion-monotone*)
apply *simp*
apply (*frule head-cons-execSVMBalanced,assumption+*)
apply (*erule exE*)
apply (*drule-tac t=last ssa* **in** *sym*)
apply (*rule-tac t=ssa* **and** *s= ss' @ [((h' |` {p ∈ dom h'. fst (the (h' p)) ≤ k}, k), k0, (q2, 0), Val v1 # S)]* **in** *ssubst,assumption*)
apply (*rename-tac ssa'*) **apply** *simp*
apply (*subst cons-append-assoc [of - ssa']*)
apply (*subst cons-cons-append [of (((h' |` {p ∈ (dom h'). ((fst (the (h' p))) ≤ k)}), k), k0, (q2, 0), ((Val v1) # S))]*)
apply (*subst maxFreshCells-P-rev*, *simp*)
apply (*unfold rev.simps*, *simp*)
apply (*unfold maxFreshCells.simps maximum.simps map.simps*, *simp*)
apply (*subgoal-tac*
(h' |` {p ∈ dom h'. fst (the (h' p)) ≤ k}, k) ↓ k = (h' |` {p ∈ dom h'. fst (the (h' p)) ≤ k}, k))
prefer *2* **apply** (*rule restrictToRegion-monotone*)
apply *simp*
apply (*subgoal-tac*
sizeH ((h' |` {p ∈ (dom h'). ((fst (the (h' p))) ≤ k)}), k) − (sizeH (h, k)) =
‖ diff k (h,k) (h' |` {p ∈ dom h'. fst (the (h' p)) ≤ k}, k) ‖)
apply *simp*
by (*rule let1-3*)

lemma *maxFreshCells-Let1*:
⟦P⊢((h, k), k, (q1, j + 1), Cont (k0, q2) # S) , 0#td#tds −svm→ s # ss , Suc 0#td#tds;
P⊢((h' |` {p ∈ dom h'. fst (the (h' p)) ≤ k}, k), k0, (q2, 0), Val v1 # S) , (td + 1)#tds −svm→ sa # ssa , Suc 0#tds;
s = ((h' |` {p ∈ dom h'. fst (the (h' p)) ≤ k}, k) ↓ k, k, (q', i), Val v1 # drop 0 (Cont (k0, q2) # S));

$sa = ((h'', k) \downarrow k0, k0, (q'a, ia), Val\ v2\ \#\ S')$;
$(\delta 1, m1, s1) = (\delta, m, w)$;
$(\delta 2, m2, s2) = (\delta', ma, wa)$;
$\delta = diff\ k\ (h, k)\ (h' |` \{p \in dom\ h'.\ fst\ (the\ (h'\ p)) \leq k\}, k)$;
$m = maxFreshCells\ (rev\ (s\ \#\ ss))$; $m >= 0$;
$ma = maxFreshCells\ (rev\ (sa\ \#\ ssa))$; $ma >= 0$⟧
⟹ $max\ m1\ (m2 + \| \delta 1 \|) =$
$maxFreshCells$
$(rev\ (sa\ \#\ ssa\ @\ ((h' |` \{p \in dom\ h'.\ fst\ (the\ (h'\ p)) \leq k\}, k) \downarrow k, k, (q', i), Val\ v1\ \#\ drop\ 0\ (Cont\ (k0, q2)\ \#\ S))\ \#$
$ss\ @\ [((h, k), k0, (q, j), S)]))$

apply (*subst cons-append-assoc*)
apply (*subst cons-append* [*of* - *ss* @ [((*h*, *k*), *k0*, (*q*, *j*), *S*)]])
apply (*subst maxFreshCells-P-rev*)
apply (*subst let1-1*,*assumption*+)
apply (*subst cons-append-2*)
apply (*subst rev.simps*(*2*) [**where** *xs*=*ss* @ [((*h*, *k*), *k0*, (*q*, *j*), *S*)]])
apply (*subst rev-append* [*of ss*])
apply (*subst rev.simps*)
apply (*subst rev.simps*)
apply (*unfold append-Nil*)
apply (*subst cons-append-2*)
apply (*subst cons-append-assoc-2* [*of* ((*h*, *k*), *k0*, (*q*, *j*), *S*)])
apply (*unfold hd.simps*)
apply (*subst cons-append-assoc-2*)
apply (*subst let1-2* [**where** *ma*=*ma*],*assumption*+)
by (*rule refl*)

lemma *maxFreshCells-Let2*:
⟦*finite* (*dom h*);
fresh pa h;
$P \vdash ((h(pa \mapsto (j, C, map\ (atom2val\ E1)\ as)), k), k0, (q, Suc\ ja), Val\ (Loc\ pa)\ \#\ S)\ ,\ (td + 1)\#tds\ -svm\rightarrow\ sa\ \#\ ss\ ,\ Suc\ 0\#tds$;
$sa = ((h', k) \downarrow k0, k0, (q', i), Val\ v2\ \#\ S')$;
$(\delta, m, s) = (\delta', ma, w)$; $ma >= 0$;
$ma = maxFreshCells\ (rev\ (sa\ \#\ ss))$⟧
⟹ $m + 1 = maxFreshCells\ (rev\ (sa\ \#\ ss\ @\ [((h, k'), k0, (q, ja), S)]))$

apply (*frule-tac s*=*sa* **in** *head-execSVMBalanced*)
apply (*case-tac ss*=[],*simp*) **apply** (*simp add*: *rev.simps add*: *maxFreshCells.simps add*: *maximum.simps add*: *map.simps*)
apply (*elim conjE*)
apply (*drule-tac t*=(*h'*, *k*) ↓ *k0* **in** *sym*)
apply (*subgoal-tac*
sizeH (*h*(*pa* ↦ (*j*, *C*, *map* (*atom2val E1*) *as*)), *k*) − *sizeH* (*h*, *k'*) = *1*)
prefer *2* **apply** (*simp add*: *sizeH.simps add*: *fresh-def*) **apply** *clarsimp*
apply *simp*
apply (*frule head-cons-execSVMBalanced*,*assumption*+)

apply (*erule exE*)
apply (*drule-tac t=last ss* **in** *sym*)
apply (*rule-tac t=ss* **and** *s= ss'* @ [((*h*(*pa* ↦ (*j*, *C*, *map* (*atom2val E1*) *as*)), *k*), *k0*, (*q*, *Suc ja*), *Val* (*Loc pa*) # *S*)]
in *ssubst*,*assumption*)
apply (*subst cons-append-assoc*)
apply *simp*
apply (*subst cons-append-assoc* [**where** *xs=ss'*])
apply (*subst cons-append-assoc* [**where** *xs=ss'*])
apply (*subst cons-cons-append*)
apply (*subst maxFreshCells-P-rev*)
apply (*simp add*: *rev.simps add*: *hd.simps*)
apply (*simp add*: *maxFreshCells.simps add*: *maximum.simps add*: *map.simps*)
apply (*subgoal-tac*
sizeH (*h*(*pa* ↦ (*j*, *C*, *map* (*atom2val E1*) *as*)), *k*) − *sizeH* (*h*, *k'*) = *1*)
prefer *2* **apply** (*simp add*: *sizeH.simps add*: *fresh-def*)
by *auto*

lemma *maxFreshCells-Case*:
⟦ *finite* (*dom h*); *i* < *length alts*; *E1 x* = *Some* (*Loc pa*); *h pa* = *Some* (*j*, *C*, *vs*);
nr = *int* (*length vs*);
s0 = ((*h*, *k*), *k0*, (*q*, *ja*), *S*);
P⊢((*h*, *k*), *k0*, (*qs* ! *i*, *0*), *map Val vs* @ *S*) , *nat* (*int td* + *nr*)#*tds* −*svm*→ *sa* # *ss* , *Suc 0*#*tds*;
sa = ((*h'*, *k*) ↓ *k0*, *k0*, (*q'*, *ia*), *Val v* # *drop* (*nat* (*int td* + *nr*)) (*map Val vs* @ *S*));
(*δ*, *m*, *s*) = (*δ'*, *ma*, *w*);
ma = *maxFreshCells* (*rev* (*sa* # *ss*)); *ma* >= *0*⟧
⟹ *m* = *maxFreshCells* (*rev* (*sa* # *ss* @ [((*h*, *k*), *k0*, (*q*, *ja*), *S*)]))
apply (*frule-tac s=sa* **in** *head-execSVMBalanced*)
apply (*case-tac ss=[]*,*simp*)
apply (*simp add*: *rev.simps add*: *maxFreshCells.simps add*: *maximum.simps add*: *map.simps*, *simp*)
apply (*frule head-cons-execSVMBalanced*,*assumption+*)
apply (*erule exE*)
apply (*drule-tac t=last ss* **in** *sym*, *simp*)
apply (*subst cons-append-assoc* [**where** *ys=* [((*h*, *k*), *k0*, ((*qs* ! *i*), *0*), ((*map Val vs*) @ *S*)), ((*h*, *k*), *k0*, (*q*, *ja*), *S*)]])
apply (*subst cons-cons-append*)
apply (*subst maxFreshCells-P-rev*, *simp*)
by (*simp add*: *rev.simps add*: *maxFreshCells.simps add*: *maximum.simps add*: *map.simps*)

lemma *maxFreshCells-Case-1-x*:
⟦ *s0* = ((*h*, *k*), *k0*, (*q*, *ja*), *S*);
P⊢((*h*, *k*), *k0*, (*qs* ! *i*, *0*), *S*) , *td* # *tds* −*svm*→ *sa* # *ss* , *Suc 0* # *tds*;
sa = ((*h'*, *k*) ↓ *k0*, *k0*, (*q'*, *ia*), *Val v* # *drop td S*);

$(\delta,\ m,\ s) = (\delta',\ ma,\ w)$;
$ma = maxFreshCells\ (rev\ (sa\ \#\ ss))$; $ma >= 0$⟧
$\Longrightarrow m = maxFreshCells\ (rev\ (sa\ \#\ ss\ @\ [((h,\ k),\ k0,\ (q,\ ja),\ S)]))$
apply (*frule-tac s=sa* **in** *head-execSVMBalanced*)
apply (*case-tac ss=[],simp*)
apply (*simp add*: *rev.simps add*: *maxFreshCells.simps add*: *maximum.simps add*: *map.simps*, *simp*)
apply (*frule head-cons-execSVMBalanced,assumption+*)
apply (*erule exE*)
apply (*drule-tac t=last ss* **in** *sym*, *simp*)
apply (*subst cons-append-assoc* [**where** *ys*= $[((h,\ k),\ k0,\ ((qs\ !\ i),\ 0),\ S),\ ((h,\ k),\ k0,\ (q,\ ja),\ S)]$])
apply (*subst cons-cons-append*)
apply (*subst maxFreshCells-P-rev*, *simp*)
by (*simp add*: *rev.simps add*: *maxFreshCells.simps add*: *maximum.simps add*: *map.simps*)

lemma *maxFreshCells-CaseD*:
⟦ *finite* $(dom\ h)$; $i < length\ alts$; $E1\ x = Some\ (Loc\ pa)$; $h\ pa = Some\ (j,\ C,\ vs)$;
$nr = int\ (length\ vs)$;
$s0 = ((h,\ k),\ k0,\ (q,\ ja),\ S)$;
$P \vdash ((h(pa := None),\ k),\ k0,\ (qs\ !\ i,\ 0),\ map\ Val\ vs\ @\ S)\ ,\ nat\ (int\ td + nr) \# tds\ -svm\rightarrow\ sa\ \#\ ss\ ,\ Suc\ 0 \# tds$;
$sa = ((h',\ k) \downarrow k0,\ k0,\ (q',\ ia),\ Val\ v\ \#\ drop\ (nat\ (int\ td + nr))\ (map\ Val\ vs\ @\ S))$;
$(\delta,\ m,\ s) = (\delta',\ ma,\ w)$;
$ma = maxFreshCells\ (rev\ (sa\ \#\ ss))$; $ma >= 0$⟧
$\Longrightarrow max\ 0\ (m - 1) = maxFreshCells\ (rev\ (sa\ \#\ ss\ @\ [((h,\ k),\ k0,\ (q,\ ja),\ S)]))$
apply (*frule-tac s=sa* **in** *head-execSVMBalanced*)
apply (*case-tac ss=[],simp*)
apply (*simp add*: *rev.simps add*: *maxFreshCells.simps add*: *maximum.simps add*: *map.simps*)
apply (*elim conjE*)
apply (*drule-tac t=*$(h',\ k) \downarrow k0$ **in** *sym*)
apply (*subgoal-tac sizeH* $(h(pa := None),\ k) - sizeH\ (h,\ k) < 0$,*simp*)
apply (*subgoal-tac* $pa \in dom\ h$)
apply (*subgoal-tac card* $(dom\ h) >= 1$)
apply (*simp add*: *sizeH.simps*)
apply (*subgoal-tac card* $(dom\ h) >= 1$, *arith*)
apply (*subgoal-tac* $\exists h'.\ h=h'(pa:=Some\ (j,\ C,\ vs)) \wedge pa \notin dom\ h'$)
apply (*erule exE*, *simp*)
apply (*rule-tac x=h*(*pa*:=*None*) **in** *exI*, *simp*)
apply (*rule sym*, *rule map-upd-triv*, *assumption*)
apply (*simp add*: *dom-def*)
apply *simp*
apply (*frule head-cons-execSVMBalanced,assumption+*)
apply (*erule exE*)
apply (*drule-tac t=last ss* **in** *sym*, *simp*)

apply (*subst cons-append-assoc* [**where** *ys*= [((*h*(*pa* := *None*), *k*), *k0*, ((*qs* ! *i*), *0*), ((*map Val vs*) @ *S*)), ((*h*, *k*), *k0*, (*q*, *ja*), *S*)]])
apply (*subst cons-cons-append*)
apply (*subst maxFreshCells-P-rev*, *simp*)
apply (*simp add*: *rev.simps add*: *maxFreshCells.simps add*: *maximum.simps add*: *map.simps*)
apply (*subgoal-tac sizeH* (*h*(*pa* := *None*), *k*) − *sizeH* (*h*, *k*) = −*1*,*simp*)
apply (*simp add*: *sizeH.simps*)
apply (*subgoal-tac pa* ∈ *dom h*)
apply (*subst card-Diff-singleton*,*assumption*+)
apply (*subgoal-tac card* (*dom h*) >= *1*, *arith*)
apply (*subgoal-tac* ∃ *h'*. *h*=*h'*(*pa*:=*Some* (*j*, *C*, *vs*)) ∧ *pa* ∉ *dom h'*)
apply (*erule exE*, *simp*)
apply (*rule-tac x*=*h*(*pa*:=*None*) **in** *exI*, *simp*)
apply (*rule sym*, *rule map-upd-triv*, *assumption*)
by (*simp add*: *dom-def*)

lemma *maxFreshCells-App*:
⟦Σ *f* = *Some* (*xs*, *rs*, *e*); *E1'* = *map-of* (*zip xs* (*map* (*atom2val E1*) *as*)); *n* = *length xs*; *l* = *length rs*;
E2' = *map-of* (*zip rs* (*map* (*the* ∘ *E2*) *rr*))(*self* ↦ *Suc k*);
finite (*dom h*);
s0 = ((*h*, *k*), *k0*, (*q*, *j*), *S*);
P⊢((*h*, *Suc k*), *k0*, (*qf*, *0*),
append (*map* (*item2Stack k S*) (*map* (*atom2item rho*) *as*))
(*append* (*map* (*item2Stack k S*) (*map* (*region2item rho*) *rr*)) (*drop td S*))) , (*n* + *l*)#*tds* −*svm*→ *sa* # *ss* , *Suc 0*#*tds*;
sa = ((*h'*, *Suc k*) ↓ *k0*, *k0*, (*q'*, *i*), *Val v* # *drop td S*);
(δ, *m*, *s*) = (δ', *ma*, *w*);
ma = *maxFreshCells* (*rev* (*sa* # *ss*)); *ma* >= *0*⟧
⟹ *m* = *maxFreshCells*
(*rev* (*sa* # *ss* @ [((*h*, *k*), *k0*, (*q*, *Suc* (*Suc j*)),
(*map* (*item2Stack k S*) (*map* (*atom2item rho*) *as*) @ *map* (*item2Stack k S*) (*map* (*region2item rho*) *rr*)) @
drop td S),
((*h*, *k*), *k0*, (*q*, *Suc j*),
(*map* (*item2Stack k S*) (*map* (*atom2item rho*) *as*) @ *map* (*item2Stack k S*) (*map* (*region2item rho*) *rr*)) @
S),
((*h*, *k*), *k0*, (*q*, *j*), *S*)]))
apply (*frule-tac s*=*sa* **in** *head-execSVMBalanced*)
apply (*case-tac ss*=[],*simp*)
apply (*simp add*: *rev.simps add*: *maxFreshCells.simps add*: *maximum.simps add*: *map.simps*)
apply (*elim conjE*)
apply (*drule-tac t*=(*h'*, *Suc k*) ↓ *k0* **in** *sym*)
apply (*simp add*: *sizeH.simps*)
apply *simp*

apply (*frule head-cons-execSVMBalanced*,*assumption*+)
apply (*erule exE*)
apply (*drule-tac t*=*last ss* **in** *sym*, *simp*)
apply (*simp add*: *rev.simps add*: *maxFreshCells.simps add*: *maximum.simps add*: *map.simps*)
apply (*subgoal-tac sizeH* (*h*, *Suc k*) − *sizeH* (*h*, *k*) = *0*,*simp*)
by (*simp add*: *sizeH.simps*)

lemma *maxFreshCells-App-primops*:
⟦*finite* (*dom h*)⟧
⟹ *0* = *maxFreshCells*
(*rev* [((*h*, *k*) ↓ *k0*, *k0*, (*q*, *Suc* (*Suc* (*Suc* (*Suc j*)))), *Val* (*execOp oper* (*atom2val E1 a1*) (*atom2val E1 a2*)) # *S'*),
((*h*, *k*), *k0*, (*q*, *Suc* (*Suc* (*Suc j*))), *Val* (*execOp oper* (*atom2val E1 a1*) (*atom2val E1 a2*)) # *drop* (*topDepth rho*) *S*),
((*h*, *k*), *k0*, (*q*, *Suc* (*Suc j*)), *Val v* # *S*),
((*h*, *k*), *k0*, (*q*, *Suc j*), *map* (*item2Stack k S*) [*atom2item rho a1*, *atom2item rho a2*] @ *S*), ((*h*, *k*), *k0*, (*q*, *j*), *S*)])
apply (*simp add*: *maxFreshCells.simps*)
apply (*subgoal-tac sizeH* ((*h*, *k*) ↓ *k0*) <= *sizeH* (*h*, *k*))
apply (*simp add*: *rev.simps add*: *maxFreshCells.simps add*: *maximum.simps add*: *map.simps*)
apply (*unfold restrictToRegion.simps*, *unfold Let-def*)
apply (*simp add*: *sizeH.simps*)
apply (*subgoal-tac dom h* ∩ {*p* ∈ *dom h*. *fst* (*the* (*h p*)) ≤ *k0*} ⊆ *dom h*)
apply (*frule card-mono*, *simp*,*clarsimp*)
by *blast*

end

14 Certification of the translation CoreSafe to SVM. maxFreshWords property

theory *maxFreshWords-lemmas*
imports *Main ../CoreSafe/SafeRASemantics SVMSemantics CoreSafeToSVM CertifSafeToSVM-definitions basic-properties*
begin

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

lemma *nth-drop'*:
i < *length xs* ⟹ *drop i xs* = *xs* ! *i* # *drop* (*Suc i*) *xs*

apply (*induct i arbitrary*: *xs*)
apply (*simp add*: *neq-Nil-conv*)
apply (*erule exE*)+
apply *simp*
apply (*case-tac xs*)
apply *simp*
apply *simp-all*
done

lemma *maxFreshWords-1* [*rule-format*]:
length S > *td* ⟶ *sizeST* (*drop td S*) − *sizeST S* <= *0*
apply (*induct td*,*simp*,*auto*)
apply (*subgoal-tac drop td S* = *S* ! *td* # *drop* (*Suc td*) *S*,*simp*)
apply (*case-tac S*!*td*)
apply (*simp add*: *sizeST.simps*)+
by (*rule nth-drop'*,*simp*)

lemma *max-assoc*:
max (*max* (*x*::*int*) (*y*::*int*)) (*z*::*int*) = *max* (*x*::*int*) (*max* (*y*::*int*) (*z*::*int*))
by *simp*

lemma *max-assoc-2*:
max (*x*::*int*) (*max* (*y*::*int*) (*z*::*int*)) = *max* (*max* (*x*::*int*) (*y*::*int*)) (*z*::*int*)
by *simp*

lemma *max-sym*:
max (*x*::*int*) (*y*::*int*) = *max* (*y*::*int*) (*x*::*int*)
by *simp*

lemma *max-sub*:
max ((*x*::*int*)−(*z*::*int*)) ((*y*::*int*)−(*z*::*int*)) = *max* (*x*::*int*) (*y*::*int*) − (*z*::*int*)
by *auto*

lemma *max-sub-2*:
max (*max* (*x*::*int*) (*y*::*int*)) (*z*::*int*) = *max* (*max* (*z*::*int*) (*x*::*int*)) (*y*::*int*)
by *auto*

lemma *foldl-max-assoc*:
foldl max (*max* (*x*::*int*) (*y*::*int*)) *xs* = *max* (*x*::*int*) (*foldl max* (*y*::*int*) *xs*)
by (*induct xs arbitrary*: *y*) (*simp-all add*: *max-assoc*)

lemma *foldl-max-assoc-2*:
foldl max (*max* (*x*::*int*) (*y*::*int*)) *xs* = *max* (*y*::*int*) (*foldl max* (*x*::*int*) *xs*)
by (*induct xs arbitrary*: *y*) (*simp-all add*: *max-assoc*)

lemma *foldl-max-absorb0*:
shows *max* (*x*::*int*) (*foldl max* (*y*::*int*) *zs*) = *foldl max* (*y*::*int*) ((*x*::*int*)#*zs*)
by (*induct zs*) (*simp-all add*:*foldl-max-assoc*)

lemma *maximum-map-sub-sizeH*:
maximum (*map* (λ(*h*, *r*, *pc*, *s*). *sizeH h* − *sizeH x*) (*y*#*ys*)) =
(*maximum* (*map* (λ(*h*, *r*, *pc*, *s*). *sizeH h*) (*y*#*ys*))) − *sizeH x*
apply (*induct ys*)
apply (*unfold maximum.simps*, *case-tac y*, *case-tac b*, *case-tac ba*, *simp*)
apply (*case-tac y*, *case-tac b*, *case-tac ba*, *rename-tac hy bay ry bayy pcy sy*, *simp*)
apply (*case-tac a*, *case-tac b*, *case-tac ba*, *rename-tac ha ba ra baa pca sa*, *simp*)
apply (*subst foldl-max-assoc-2*)
apply (*subst foldl-max-assoc-2*)
by *simp*

lemma *maximum-map-sub-sizeST*:
maximum (*map* (λ(*h*, *r*, *pc*, *w*). *sizeST w* − *sizeST x*) (*y*#*ys*)) =
(*maximum* (*map* (λ(*h*, *r*, *pc*, *w*). *sizeST w*) (*y*#*ys*))) − *sizeST x*
apply (*induct ys*)
apply (*unfold maximum.simps*, *case-tac y*, *case-tac b*, *case-tac ba*, *simp*)
apply (*case-tac y*, *case-tac b*, *case-tac ba*, *rename-tac hy bay ry bayy pcy sy*, *simp*)
apply (*case-tac a*, *case-tac b*, *case-tac ba*, *rename-tac ha ba ra baa pca sa*, *simp*)
apply (*subst foldl-max-assoc-2*)
apply (*subst foldl-max-assoc-2*)
by *simp*

lemma *max-maximum-sub*:
max ((*maximum* (*map f* (*x*#*xs*))) − *z*) ((*maximum* (*map f* (*y*#*ys*))) − *z*) =
maximum (*map f* ((*x*#*xs*) @ (*y*#*ys*))) − *z*
apply (*induct xs ys rule*:*list-induct2′*)
apply (*simp add*: *maximum.simps*)

apply (*simp add*: *maximum.simps*)

apply (*simp add*: *maximum.simps*)
apply (*subst max-sub*, *subst foldl-max-absorb0*)
apply (*simp*, *subst max-sub-2*)
apply (*simp add*: *maximum.simps*)

apply (*simp add*: *maximum.simps*)
apply (*subst foldl-max-assoc-2* [**where** *y*=(*f xa*)])
apply (*subst max-sub*, *subst max-assoc* , *subst foldl-max-assoc-2*)
apply (*subst max-assoc-2* [**where** *y*=*f ya*])
apply (*subst max-sym* [**where** *y*=*f ya*])
apply (*subst max-assoc*, *subst max-assoc-2*)

apply (*subst foldl-max-assoc-2* [**where** *y*=*f ya*])

apply (*subst foldl-max-assoc-2* [**where** *y*= *f xa*])
apply (*subst max-assoc* [**where** *x*=*f xa* **and** *z*=*f y*])
apply (*subst foldl-max-assoc* [**where** *x*= *f xa*])
apply (*subst max-assoc-2* [**where** *y*=*f xa*])
apply (*subst max-sym*)
by *auto*

lemma *maximum-map-append*:
maximum (*map f* (*xs* @ *x* # *x* # *ys*)) = *maximum* (*map f* (*xs* @ *x* # *ys*))
apply (*induct xs ys rule*:*list-induct2*′)
by (*simp add*: *maximum.simps*)+

declare *map.simps* [*simp del*]
declare *maxFreshWords.simps* [*simp del*]
declare *maximum.simps* [*simp del*]

lemma *maxFreshWords-P*:
maxFreshWords (*ss1* @ [*s*] @ *ss2*) =
max (*maxFreshWords* (*ss1* @ [*s*]))
((*maxFreshWords* ([*s*] @ *ss2*)) + *sizeSVMStateST s* − *sizeSVMStateST* (*hd* (*ss1* @ [*s*])))
apply (*case-tac s*, *case-tac b*, *case-tac ba*) **apply** (*rename-tac hi bi ri bai pci si*)
apply (*induct ss1 ss2 rule*:*list-induct2*′)

apply (*simp add*: *maxFreshWords.simps*)

apply (*simp add*: *maxFreshWords.simps add*: *maximum.simps*)
apply (*case-tac x*, *case-tac b*, *case-tac ba*) **apply** (*rename-tac hx bx rx bax pcx sx*) **apply** *simp*
apply (*simp add*: *maxFreshWords.simps add*: *maximum.simps add*: *map.simps*)

apply (*simp*, *simp add*: *maxFreshWords.simps add*: *maximum.simps add*: *map.simps*)
apply (*subst foldl-max-absorb0*)
apply (*subgoal-tac 0* = *sizeST si* − *sizeST si*,*simp*)
apply (*subst max-assoc*, *subst max-sym* [*of - 0*])
apply (*subst max-assoc-2*, *simp*, *simp*)

apply *simp*
apply (*case-tac x*, *case-tac b*, *case-tac ba*) **apply** (*rename-tac hx bx rx bax pcx sx*) **apply** *simp*
apply (*simp add*: *maxFreshWords.simps*)
apply (*case-tac y*, *case-tac b*, *case-tac ba*) **apply** (*rename-tac hy bby ry bay pcy sy*) **apply** *simp*
apply (*subgoal-tac 0* = *sizeST sx* − *sizeST sx*) **prefer** *2* **apply** *simp*
apply (*subst maximum-map-sub-sizeST* [**where** *ys*=*xs* @ (*hi*, *ri*, *pci*, *si*) # (*hy*, *ry*, *pcy*, *sy*) # *ys*])
apply (*subst maximum-map-sub-sizeST* [**where** *ys*=*xs* @ [(*hi*, *ri*, *pci*, *si*)]])
apply (*subst maximum-map-sub-sizeST* [**where** *ys*=(*hy*, *ry*, *pcy*, *sy*) # *ys*])

apply (*simp*, *subst max-maximum-sub*, *simp*, *rule sym*)
apply (*subgoal-tac*
maximum (*map* (λ(*h*, *r*, *pc*, *y*). *sizeST y*) (((*hx*, *rx*, *pcx*, *sx*) # *xs*) @ (*hi*, *ri*, *pci*, *si*) # (*hi*, *ri*, *pci*, *si*) # (*hy*, *ry*, *pcy*, *sy*) # *ys*)) =
maximum (*map* (λ(*h*, *r*, *pc*, *y*). *sizeST y*) (((*hx*, *rx*, *pcx*, *sx*) # *xs*) @ (*hi*, *ri*, *pci*, *si*) # (*hy*, *ry*, *pcy*, *sy*) # *ys*)), *simp*)
by (*rule maximum-map-append*)

lemma *maxFreshWords-P-rev*:
maxFreshWords (*rev* (*ss2* @ [*s*] @ *ss1*)) =
max (*maxFreshWords* (*rev* ([*s*] @ *ss1*)))
((*maxFreshWords* (*rev* (*ss2* @ [*s*]))) + *sizeSVMStateST s* − *sizeSVM-StateST* (*hd* (*rev* ([*s*] @ *ss1*))))
apply (*case-tac s*, *case-tac b*, *case-tac ba*) **apply** (*rename-tac hi bi ri bai pci si*, *simp*)
apply (*induct rev ss1 rev ss2 rule*:*list-induct2'*)

apply (*simp add*: *maxFreshWords.simps*)

apply (*simp add*: *maxFreshWords.simps add*: *maximum.simps*)
apply (*case-tac x*, *case-tac b*, *case-tac ba*) **apply** (*rename-tac hx bx rx bax pcx sx*) **apply** *simp*
apply (*simp add*: *maxFreshWords.simps add*: *maximum.simps add*: *map.simps*)

apply (*simp*, *simp add*: *maxFreshWords.simps add*: *maximum.simps add*: *map.simps*)
apply (*subst foldl-max-absorb0*)
apply (*subgoal-tac 0* = *sizeST si* − *sizeST si*,*simp*)
apply (*subst max-assoc*, *subst max-sym* [*of - 0*])
apply (*subst max-assoc-2*, *simp*, *simp*)

apply *simp*
apply (*case-tac x*, *case-tac b*, *case-tac ba*) **apply** (*rename-tac hx bx rx bax pcx sx*) **apply** *simp*
apply (*simp add*: *maxFreshWords.simps*)
apply (*case-tac y*, *case-tac b*, *case-tac ba*) **apply** (*rename-tac hy bby ry bay pcy sy*) **apply** *simp*
apply (*subgoal-tac 0* = *sizeST sx* − *sizeST sx*) **prefer** *2* **apply** *simp*
apply (*subst maximum-map-sub-sizeST* [**where** *ys*=*xs* @ (*hi*, *ri*, *pci*, *si*) # (*hy*, *ry*, *pcy*, *sy*) # *ys*])
apply (*subst maximum-map-sub-sizeST* [**where** *ys*=*xs* @ [(*hi*, *ri*, *pci*, *si*)]])
apply (*subst maximum-map-sub-sizeST* [**where** *ys*=(*hy*, *ry*, *pcy*, *sy*) # *ys*])
apply (*simp*, *subst max-maximum-sub*, *simp*, *rule sym*)
apply (*subgoal-tac*
maximum (*map* (λ(*h*, *r*, *pc*, *y*). *sizeST y*) (((*hx*, *rx*, *pcx*, *sx*) # *xs*) @ (*hi*, *ri*, *pci*, *si*) # (*hi*, *ri*, *pci*, *si*) # (*hy*, *ry*, *pcy*, *sy*) # *ys*)) =
maximum (*map* (λ(*h*, *r*, *pc*, *y*). *sizeST y*) (((*hx*, *rx*, *pcx*, *sx*) # *xs*) @ (*hi*, *ri*, *pci*, *si*) # (*hy*, *ry*, *pcy*, *sy*) # *ys*)), *simp*)
by (*rule maximum-map-append*)

lemma *maxFreshWords-IntT*:
⟦*s0* = ((*h*, *k*), *k0*, (*q*, *j*), *S*);
length S > *topDepth rho*;
S' = *drop* (*topDepth rho*) *S*⟧
⟹ *1* = *maxFreshWords*
(*rev* (((*h*, *k*) ↓ *k0*, *k0*, (*q*, *Suc* (*Suc* (*Suc j*))), *Val* (*IntT i*) # *S'*) #
[((*h*, *k*), *k0*, (*q*, *j* + *2*), *Val* (*IntT i*) # *drop* (*topDepth rho*) *S*), ((*h*, *k*), *k0*, (*q*, *j* + *1*), *Val* (*IntT i*) # *S*),
((*h*, *k*), *k0*, (*q*, *j*), *S*)]))
apply (*simp add*: *maxFreshWords.simps*)
apply (*simp add*: *map.simps add*: *maximum.simps*)
apply (*simp add*: *sizeST.simps*)
apply (*subgoal-tac sizeST* (*drop* (*topDepth rho*) *S*) − *sizeST S* <= *0*)
apply *simp*
by (*rule maxFreshWords-1*)

lemma *maxFreshWords-BoolT*:
⟦*s0* = ((*h*, *k*), *k0*, (*q*, *j*), *S*);
length S > *topDepth rho*;
S' = *drop* (*topDepth rho*) *S*⟧
⟹ *1* = *maxFreshWords*
(*rev* (((*h*, *k*) ↓ *k0*, *k0*, (*q*, *Suc* (*Suc* (*Suc j*))), *Val* (*BoolT b*) # *S'*) #
[((*h*, *k*), *k0*, (*q*, *j* + *2*), *Val* (*BoolT b*) # *drop* (*topDepth rho*) *S*), ((*h*, *k*), *k0*, (*q*, *j* + *1*), *Val* (*BoolT b*) # *S*),
((*h*, *k*), *k0*, (*q*, *j*), *S*)]))
apply (*simp add*: *maxFreshWords.simps*)
apply (*simp add*: *map.simps add*: *maximum.simps*)
apply (*simp add*: *sizeST.simps*)
apply (*subgoal-tac sizeST* (*drop* (*topDepth rho*) *S*) − *sizeST S* <= *0*)
apply *simp*
by (*rule maxFreshWords-1*)

lemma *maxFreshWords-VarE*:
⟦*s0* = ((*h*, *k*), *k0*, (*q*, *j*), *S*);
length S > *topDepth rho*;
S' = *drop* (*topDepth rho*) *S*⟧
⟹ *1* = *maxFreshWords*
(*rev* (((*h*, *k*) ↓ *k0*, *k0*, (*q*, *Suc* (*Suc* (*Suc j*))), *Val* (*Loc pa*) # *S'*) #
[((*h*, *k*), *k0*, (*q*, *j* + *2*), *Val* (*Loc pa*) # *drop* (*topDepth rho*) *S*), ((*h*, *k*), *k0*, (*q*, *j* + *1*), *Val* (*Loc pa*) # *S*),
((*h*, *k*), *k0*, (*q*, *j*), *S*)]))
apply (*simp add*: *maxFreshWords.simps*)
apply (*simp add*: *map.simps add*: *maximum.simps*)
apply (*simp add*: *sizeST.simps*)

apply (*subgoal-tac sizeST* (*drop* (*topDepth rho*) *S*) − *sizeST S* <= *0*)
apply *simp*
by (*rule maxFreshWords-1*)

lemma *maxFreshWords-CopyE*:
⟦*s0* = ((*h*, *k*), *k0*, (*q*, *ja*), *S*);
length S > *topDepth rho*;
S′ = *drop* (*topDepth rho*) *S*⟧
⟹ *2* = *maxFreshWords*
(*rev* [((*h′*, *k*) ↓ *k0*, *k0*, (*q*, *Suc* (*Suc* (*Suc* (*Suc ja*)))), *Val* (*Loc p′*) # *S′*),
((*h′*, *k*), *k0*, (*q*, *Suc* (*Suc* (*Suc ja*))), *Val* (*Loc p′*) # *drop* (*topDepth rho*) *S*),
((*h′*, *k*), *k0*, (*q*, *Suc* (*Suc ja*)), *Val* (*Loc p′*) # *S*), ((*h*, *k*), *k0*, (*q*, *Suc ja*), *Val* (*Loc pa*) # *Reg j* # *S*),
((*h*, *k*), *k0*, (*q*, *ja*), *S*)])
apply (*simp add*: *maxFreshWords.simps*)
apply (*simp add*: *map.simps add*: *maximum.simps*)
apply (*simp add*: *sizeST.simps*)
apply (*subgoal-tac sizeST* (*drop* (*topDepth rho*) *S*) − *sizeST S* <= *0*)
apply *simp*
by (*rule maxFreshWords-1*)

lemma *maxFreshWords-ReuseE*:
⟦ *S′* = *drop* (*topDepth rho*) *S*;
length S > *topDepth rho*⟧
⟹ *1* = *maxFreshWords*
(*rev* [((*h*(*pa* := *None*)(*q* ↦ *c*), *k*) ↓ *k0*, *k0*, (*qa*, *Suc* (*Suc* (*Suc* (*Suc j*)))), *Val* (*Loc q*) # *S′*),
((*h*(*pa* := *None*)(*q* ↦ *c*), *k*), *k0*, (*qa*, *Suc* (*Suc* (*Suc j*))), *Val* (*Loc q*) # *drop* (*topDepth rho*) *S*),
((*h*(*pa* := *None*)(*q* ↦ *c*), *k*), *k0*, (*qa*, *Suc* (*Suc j*)), *Val* (*Loc q*) # *S*), ((*h*, *k*), *k0*, (*qa*, *Suc j*), *Val* (*Loc pa*) # *S*),
((*h*, *k*), *k0*, (*qa*, *j*), *S*)])
apply (*simp add*: *maxFreshWords.simps*)
apply (*simp add*: *map.simps add*: *maximum.simps*)
apply (*simp add*: *sizeST.simps*)
apply (*subgoal-tac sizeST* (*drop* (*topDepth rho*) *S*) − *sizeST S* <= *0*)
apply *simp*
by (*rule maxFreshWords-1*)

lemma *head-execSVMBalanced*:
P⊢*s0* , *n* −*svm*→ *s* # *ss*, *m* ⟹ *s0* = *last* (*s*#*ss*)
by (*erule execSVMBalanced.induct*, *simp-all*)

lemma *head-cons-execSVMBalanced*:
⟦ *ss*≠[]; *s0* = *last ss*⟧ ⟹ ∃ *ss′*. *ss* = *ss′* @ [*s0*]
by (*rule-tac x*=*butlast ss* **in** *exI*, *simp*)

lemma *cons-append-assoc*:
$x\#(xs\ @\ ys) = (x\#xs)\ @\ ys$
by *simp*

lemma *cons-append-assoc-2*:
$(x\#xs)\ @\ ys = x\#(xs\ @\ ys)$
by *simp*

lemma *cons-append*:
$x\#xs = [x]\ @\ xs$
by *simp*

lemma *cons-append-2*:
$[x]\ @\ xs = x\ \#\ xs$
by *simp*

lemma *append-assoc-2*:
$xs\ @\ (ys\ @\ zs) = (xs\ @\ ys)\ @\ zs$
by $(induct\ xs)\ auto$

lemma *cons-cons-append*:
$[x,y] = [x]\ @\ [y]$
by *simp*

lemma *append-Nil3*:
$xs = xs\ @\ []$
by *simp*

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

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

lemma *let1-1*:
$[\![s = ((h' \mid `` \{p \in dom\ h'.\ fst\ (the\ (h'\ p)) \leq k\},\ k) \downarrow k,\ k,\ (q',\ i),\ Val\ v1\ \#\ drop\ 0\ (Cont\ (k0,\ q2)\ \#\ S));$
$(\delta 1,\ m1,\ s1) = (\delta,\ m,\ w);$
$P \vdash ((h,\ k),\ k,\ (q1,\ j\ +\ 1),\ Cont\ (k0,\ q2)\ \#\ S)\ ,\ 0\#td\#tds\ -svm\rightarrow\ s\ \#\ ss\ ,\ Suc\ 0\#td\#tds;$
$w = maxFreshWords\ (rev\ (s\ \#\ ss));$
$0 < w\ ]\!]$
$\Longrightarrow (maxFreshWords$
$(rev\ ([((h' \mid `` \{p \in dom\ h'.\ fst\ (the\ (h'\ p)) \leq k\},\ k),\ k0,\ (q2,\ 0),\ Val\ v1\ \#\ S)]\ @$
$((h' \mid `` \{p \in dom\ h'.\ fst\ (the\ (h'\ p)) \leq k\},\ k) \downarrow k,\ k,\ (q',\ i),\ Val\ v1\ \#\ Cont\ (k0,\ q2)\ \#\ S)\ \#$
$ss\ @\ [((h,\ k),\ k0,\ (q,\ j),\ S)])))\ =\ s1+2$
apply (*frule-tac s=s* **in** *head-execSVMBalanced*)

apply (*case-tac ss=[],simp,simp*)
apply (*frule head-cons-execSVMBalanced,assumption+*)
apply (*erule exE*)
apply (*drule-tac t=last ss* **in** *sym*)
apply (*rule-tac t=ss* **and** *s=ss'* @ [((*h*, *k*), *k*, (*q1*, *Suc j*), *Cont* (*k0*, *q2*) # *S*)] **in** *ssubst,assumption*)
apply *simp*

apply (*subst cons-append* [*of* ((*h'* |` {*p* ∈ *dom h'*. *fst* (*the* (*h' p*)) ≤ *k*}, *k*), *k0*, (*q2*, *0*), *Val v1* # *S*)])
apply (*subst cons-append* [*of* ((*h'* |` {*p* ∈ *dom h'*. *fst* (*the* (*h' p*)) ≤ *k*}, *k*) ↓ *k*, *k*, (*q'*, *i*), *Val v1* # *Cont* (*k0*, *q2*) # *S*)])
apply (*subst maxFreshWords-P-rev*)

apply (*subst append-assoc-2*)
apply (*subst cons-cons-append*)
apply (*subst maxFreshWords-P-rev*)
apply (*simp add*: *rev.simps add*: *map.simps add*: *maxFreshWords.simps add*: *sizeSVM-StateST.simps*)
by (*simp add*: *sizeST.simps add*: *maximum.simps*)

lemma *let1-2*:
⟦*sa* = ((*h''*, *k*) ↓ *k0*, *k0*, (*q'a*, *ia*), *Val v2* # *S'*);
(*δ2*, *m2*, *s2*) = (*δ'*, *ma*, *wa*);
wa = *maxFreshWords* (*rev* (*sa* # *ssa*));
ssa = *ssa'* @ [((*h'* |` {*p* ∈ *dom h'*. *fst* (*the* (*h' p*)) ≤ *k*}, *k*), *k0*, (*q2*, *0*), *Val v1* # *S*)]⟧
⟹ (*maxFreshWords*
(*rev* (((*h''*, *k*) ↓ *k0*, *k0*, (*q'a*, *ia*), *Val v2* # *S'*) #
ssa' @ [((*h'* |` {*p* ∈ *dom h'*. *fst* (*the* (*h' p*)) ≤ *k*}, *k*), *k0*, (*q2*, *0*), *Val v1* # *S*)])) +
sizeST (*Val v1* # *S*) −
sizeSVMStateST
(*hd* (*rev* (((*h'* |` {*p* ∈ *dom h'*. *fst* (*the* (*h' p*)) ≤ *k*}, *k*), *k0*, (*q2*, *0*), *Val v1* # *S*) #
((*h'* |` {*p* ∈ *dom h'*. *fst* (*the* (*h' p*)) ≤ *k*}, *k*) ↓ *k*, *k*, (*q'*, *i*), *Val v1* # *Cont* (*k0*, *q2*) # *S*) #
ss @ [((*h*, *k*), *k0*, (*q*, *j*), *S*)]))))) = *s2* + *1*
apply (*simp add*: *rev.simps add*: *map.simps add*: *maxFreshWords.simps add*: *sizeSVM-StateST.simps*)
apply (*simp add*: *sizeST.simps*)
done

lemma *maxFreshWords-Let1*:
⟦*s* = ((*h'* |` {*p* ∈ *dom h'*. *fst* (*the* (*h' p*)) ≤ *k*}, *k*) ↓ *k*, *k*, (*q'*, *i*), *Val v1* # *drop* *0* (*Cont* (*k0*, *q2*) # *S*));

$sa = ((h'', k) \downarrow k0, k0, (q'a, ia), Val\ v2\ \#\ S');\ (\delta 1, m1, s1) = (\delta, m, w); (\delta 2, m2, s2) = (\delta', ma, wa);$
$P \vdash ((h, k), k, (q1, j + 1), Cont\ (k0, q2)\ \#\ S)\ , 0\#td\#tds\ -svm\rightarrow\ s\ \#\ ss\ , Suc\ 0\#td\#tds;$
$P \vdash ((h' |` \{p \in dom\ h'.\ fst\ (the\ (h'\ p)) \leq k\}, k), k0, (q2, 0), Val\ v1\ \#\ S)\ , (td + 1)\#tds\ -svm\rightarrow\ sa\ \#\ ssa\ , Suc\ 0\#tds;$
$w = maxFreshWords\ (rev\ (s\ \#\ ss)); ma = maxFreshCells\ (rev\ (sa\ \#\ ssa)); wa = maxFreshWords\ (rev\ (sa\ \#\ ssa));\ 0 < w\rrbracket$
$\Longrightarrow max\ (s1 + 2)\ (s2 + 1) =$
$maxFreshWords$
$(rev\ (sa\ \#\ ssa\ @\ ((h' |` \{p \in dom\ h'.\ fst\ (the\ (h'\ p)) \leq k\}, k) \downarrow k, k, (q', i), Val\ v1\ \#\ drop\ 0\ (Cont\ (k0, q2)\ \#\ S))\ \#$
$ss\ @\ [((h, k), k0, (q, j), S)]))$
apply (*frule-tac s=sa* **in** *head-execSVMBalanced*)
apply (*case-tac ssa=[]*) **defer**
apply *simp*
apply (*frule head-cons-execSVMBalanced,assumption+*)
apply (*erule exE*)
apply (*drule-tac t=last ssa* **in** *sym*)
apply (*rule-tac t=ssa* **and** *s=ss'* @ *[((h' |` {p ∈ dom h'. fst (the (h' p)) ≤ k}, k), k0, (q2, 0), Val v1 # S)]* **in** *ssubst,assumption*)
apply (*rename-tac ssa'*)
apply *simp*
apply (*subst cons-append-assoc*)
apply (*subst cons-append-assoc*)
apply (*subst cons-append* [**where** *xs =((h' |` {p ∈ dom h'. fst (the (h' p)) ≤ k}, k) ↓ k, k, (q', i), Val v1 # Cont (k0, q2) # S) #*
ss @ [((h, k), k0, (q, j), S)]])
apply (*subst maxFreshWords-P-rev*)
apply (*subst let1-1,simp-all*)
apply (*subst let1-2,simp-all*)
apply (*rule conjI,simp*)+
apply (*rule refl*)
apply (*frule-tac s=(((h'', k) ↓ k0) ↓ k, k, (q', i), Val v2 # Cont (k0, q'a) # S'*) **in** *head-execSVMBalanced*)
apply (*case-tac ss=[],simp*) **apply** *simp*
apply (*frule head-cons-execSVMBalanced,assumption+*)
apply (*erule exE*)
apply (*drule-tac t=last ss* **in** *sym*)
apply (*rule-tac t=ss* **and** *s=ss' @ [((h, k), k, (q1, Suc j), Cont (k0, q'a) # S')]* **in** *ssubst,assumption*)
apply (*subst append-assoc*)
apply (*subst cons-append-assoc [of - - ([((h, k), k, (q1, (Suc j)), ((Cont (k0, q'a)) # S'))] @ [((h, k), k0, (q, j), S')])]*)
apply (*subst cons-append-assoc [of ((h'', k) ↓ k0, k0, (q'a, 0), Val v2 # S')]*)
apply (*subst maxFreshWords-P-rev*)
apply (*simp add: rev.simps add: map.simps add: maxFreshWords.simps add: sizeSVM-StateST.simps*)
apply (*simp add: sizeST.simps add: maximum.simps*)

done

lemma *maxFreshWords-Let2*:
⟦*sa* = ((*h'*, *k*) ↓ *k0*, *k0*, (*q'*, *i*), *Val v2* # *S'*);
P⊢((*h*(*pa* ↦ (*j*, *C*, *b*)), *k*), *k0*, (*q*, *Suc ja*), *Val* (*Loc pa*) # *S*) , (*td* + *1*)#*tds* −*svm*→ *sa* # *ss* , *Suc 0*#*tds*;
(δ, *m*, *s*) = (δ', *ma*, *w*);
w = *maxFreshWords* (*rev* (*sa* # *ss*)); *0* < *w*⟧
⟹ *s* + *1* = *maxFreshWords* (*rev* (*sa* # *ss* @ [((*h*, *k'*), *k0*, (*q*, *ja*), *S*)]))
apply (*frule-tac s=sa* **in** *head-execSVMBalanced*)
apply (*case-tac ss=*[],*simp*) **apply** (*simp add*: *rev.simps add*: *maxFreshWords.simps add*: *maximum.simps add*: *map.simps*)
apply *simp*
apply (*frule head-cons-execSVMBalanced*,*assumption*+)
apply (*erule exE*)
apply (*drule-tac t=last ss* **in** *sym*)
apply (*rule-tac t=ss* **and** *s= ss'* @ [((*h*(*pa* ↦ (*j*, *C*, *b*)), *k*), *k0*, (*q*, *Suc ja*), *Val* (*Loc pa*) # *S*)] **in** *ssubst*,*assumption*)
apply *simp*
apply (*subst cons-append-assoc*)
apply (*subst cons-append-assoc*)
apply (*subst cons-cons-append*)
apply (*subst maxFreshWords-P-rev*)
apply (*simp add*: *rev.simps add*: *map.simps add*: *maxFreshWords.simps add*: *sizeSVM-StateST.simps*)
apply (*simp add*: *sizeST.simps*) **apply** (*simp add*: *maximum.simps*)
done

lemma *max-1*:
⟦ *max 0 a* = *a*; *max c b* > *0* ⟧ ⟹ *max* (*c*) (*b*) + *a* = *max* (*max* (*0*::*int*) (*a*::*int*)) (*max* (*c*) (*b*) + *a*)
by *simp*

lemma *max-2*:
(*a*::*int*) > (*b*::*int*) ⟹ (*max f c* + *a* − *b*) = (*max f c*) + (*a* − *b*)
by *simp*

lemma *sizeST-Val-vs-length-vs*:
int (*length vs*) = *sizeST* (*map Val vs* @ *S*) − *sizeST S*
apply (*induct vs arbitrary*: *S*)
apply (*simp add*: *map.simps*)
by (*simp add*: *map.simps add*: *sizeST.simps*)

lemma *maxFreshWords-Case*:
⟦ *finite* (*dom h*); *i* < *length alts*; *E1 x* = *Some* (*Loc pa*); *h pa* = *Some* (*j*, *C*, *vs*);
nr = *int* (*length vs*);
s0 = ((*h*, *k*), *k0*, (*q*, *ja*), *S*);

$P\vdash((h,\ k),\ k0,\ (qs\ !\ i,\ 0),\ map\ Val\ vs\ @\ S)\ ,\ nat\ (int\ td\ +\ nr)\#tds\ -svm\rightarrow sa\ \#\ ss\ ,\ Suc\ 0\#tds;$
$sa = ((h',\ k) \downarrow k0,\ k0,\ (q',\ ia),\ Val\ v\ \#\ drop\ (nat\ (int\ td\ +\ nr))\ (map\ Val\ vs\ @\ S));$
$(\delta,\ m,\ s) = (\delta',\ ma,\ w);\ w > 0;$
$w = maxFreshWords\ (rev\ (sa\ \#\ ss))\rrbracket$
$\Longrightarrow$
$s + nr = maxFreshWords\ (rev\ (sa\ \#\ ss\ @\ [((h,\ k),\ k0,\ (q,\ ja),\ S)]))$
apply (*frule-tac s=sa* **in** *head-execSVMBalanced*)
apply (*case-tac ss=[],simp*)
apply (*simp add*: *rev.simps add*: *maxFreshWords.simps add*: *maximum.simps add*: *map.simps*)
apply *clarsimp*
apply (*frule head-cons-execSVMBalanced,assumption+*)
apply (*erule exE*)
apply (*drule-tac t=last ss* **in** *sym*, *simp*)
apply (*subst cons-append-assoc* [**where** *ys*= [((*h*, *k*), *k0*, ((*qs* ! *i*), *0*), ((*map Val vs*) @ *S*)), ((*h*, *k*), *k0*, (*q*, *ja*), *S*)]])
apply (*subst cons-cons-append*)
apply (*subst maxFreshWords-P-rev*, *simp*)
apply (*simp add*: *rev.simps add*: *maxFreshWords.simps add*: *maximum.simps add*: *map.simps*)
apply (*case-tac vs,simp*)
apply (*subgoal-tac sizeST* (*map Val* [] @ *S*) − *sizeST S* = *0,simp*)
apply (*simp add*: *map.simps*)
apply *simp*
apply (*subgoal-tac int* (*length vs*) > *0*) **prefer** *2* **apply** *simp*
apply (*subgoal-tac int* (*length vs*) = *sizeST* (*map Val vs* @ *S*) − *sizeST S,simp*)
apply (*subgoal-tac sizeST* (*map Val vs* @ *S*) > *sizeST S*)
apply (*subgoal-tac max 0* (*sizeST* (*map Val vs* @ *S*) − *sizeST S*) = *sizeST* (*map Val vs* @ *S*) − *sizeST S,simp*)
apply (*subst max-1,simp+*)
apply (*subst max-2,simp*)
apply (*simp*, *simp*, *simp*)
by (*rule sizeST-Val-vs-length-vs*)

lemma *maxFreshWords-Case-1-x*:
$\llbracket\ s0 = ((h,\ k),\ k0,\ (q,\ ja),\ S);$
$P\vdash((h,\ k),\ k0,\ (qs\ !\ i,\ 0),\ S)\ ,\ td\ \#\ tds\ -svm\rightarrow sa\ \#\ ss\ ,\ Suc\ 0\ \#\ tds;$
$sa = ((h',\ k) \downarrow k0,\ k0,\ (q',\ ia),\ Val\ v\ \#\ drop\ td\ S);$
$(\delta,\ m,\ s) = (\delta',\ ma,\ w);\ w > 0;$
$w = maxFreshWords\ (rev\ (sa\ \#\ ss))\rrbracket$
$\Longrightarrow$
$s\ = maxFreshWords\ (rev\ (sa\ \#\ ss\ @\ [((h,\ k),\ k0,\ (q,\ ja),\ S)]))$
apply (*frule-tac s=sa* **in** *head-execSVMBalanced*)
apply (*case-tac ss=[],simp*)
apply (*simp add*: *rev.simps add*: *maxFreshWords.simps add*: *maximum.simps add*: *map.simps*)
apply *clarsimp*

apply (*frule head-cons-execSVMBalanced*,*assumption*+)
apply (*erule exE*)
apply (*drule-tac t*=*last ss* **in** *sym*, *simp*)
apply (*subst cons-append-assoc* [**where** *ys*= [((*h*, *k*), *k0*, ((*qs* ! *i*), *0*), *S*), ((*h*, *k*), *k0*, (*q*, *ja*), *S*)]])
apply (*subst cons-cons-append*)
apply (*subst maxFreshWords-P-rev*, *simp*)
by (*simp add*: *rev.simps add*: *maxFreshWords.simps add*: *maximum.simps add*: *map.simps*)

lemma *maxFreshWords-CaseD*:
⟦ *finite* (*dom h*); *i* < *length alts*; *E1 x* = *Some* (*Loc pa*); *h pa* = *Some* (*j*, *C*, *vs*);
nr = *int* (*length vs*);
s0 = ((*h*, *k*), *k0*, (*q*, *ja*), *S*);
P⊢((*h*(*pa* := *None*), *k*), *k0*, (*qs* ! *i*, *0*), *map Val vs* @ *S*) , *nat* (*int td* + *nr*)#*tds* −*svm*→ *sa* # *ss* , *Suc 0*#*tds*;
sa = ((*h*′, *k*) ↓ *k0*, *k0*, (*q*′, *ia*), *Val v* # *drop* (*nat* (*int td* + *nr*)) (*map Val vs* @ *S*));
(*δ*, *m*, *s*) = (*δ*′, *ma*, *w*); *w* > *0*;
w = *maxFreshWords* (*rev* (*sa* # *ss*))⟧
⟹
s + *nr* = *maxFreshWords* (*rev* (*sa* # *ss* @ [((*h*, *k*), *k0*, (*q*, *ja*), *S*)]))
apply (*frule-tac s*=*sa* **in** *head-execSVMBalanced*)
apply (*case-tac ss*=[],*simp*)
apply (*simp add*: *rev.simps add*: *maxFreshWords.simps add*: *maximum.simps add*: *map.simps*)
apply *clarsimp*
apply (*frule head-cons-execSVMBalanced*,*assumption*+)
apply (*erule exE*)
apply (*drule-tac t*=*last ss* **in** *sym*, *simp*)
apply (*subst cons-append-assoc* [**where** *ys*= [((*h*(*pa* := *None*), *k*), *k0*, ((*qs* ! *i*), *0*), ((*map Val vs*) @ *S*)), ((*h*, *k*), *k0*, (*q*, *ja*), *S*)]])
apply (*subst cons-cons-append*)
apply (*subst maxFreshWords-P-rev*, *simp*)
apply (*simp add*: *rev.simps add*: *maxFreshWords.simps add*: *maximum.simps add*: *map.simps*)
apply (*case-tac vs*,*simp*)
apply (*subgoal-tac sizeST* (*map Val* [] @ *S*) − *sizeST S* = *0*,*simp*)
apply (*simp add*: *map.simps*)
apply *simp*
apply (*subgoal-tac int* (*length vs*) > *0*) **prefer** *2* **apply** *simp*
apply (*subgoal-tac int* (*length vs*) = *sizeST* (*map Val vs* @ *S*) − *sizeST S*,*simp*)
apply (*subgoal-tac sizeST* (*map Val vs* @ *S*) > *sizeST S*)
apply (*subgoal-tac max 0* (*sizeST* (*map Val vs* @ *S*) − *sizeST S*) = *sizeST* (*map Val vs* @ *S*) − *sizeST S*,*simp*)
apply (*subst max-1*,*simp*+)
apply (*subst max-2*,*simp*)
apply (*simp*, *simp*, *simp*)

by (*rule sizeST-Val-vs-length-vs*)

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

lemma *identityEnvironment-item-val* [*rule-format*]:
(*E1*,*E2*) ⋈ (*rho*,*S*)
⟶ *fv a* ⊆ *dom E1*
⟶ *atom a*
⟶ *item2Stack k S* (*atom2item rho a*) = *Val* (*atom2val E1 a*)
apply (*case-tac a*,*simp-all*)
apply (*rename-tac n a'*)
apply (*case-tac n*, *simp-all*)
apply (*rename-tac x a'*)
apply (*rule impI*)
by (*simp add*: *identityEnvironment.simps*)

lemma *maxFreshWords-App-primops*:
⟦*v1* = *atom2val E1 a1*; *v2* = *atom2val E1 a2*; *v* = *execOp oper v1 v2*; *topDepth rho* < *length S*;
fv (*AppE f* [*a1*, *a2*] [] *a*) ⊆ *dom* (*fst* (*E1*, *E2*));
∀ *a* ∈ *set* [*a1*, *a2*]. *atom a*;
(*E1*, *E2*) ⋈ (*rho* , *S*) ; *td* = *topDepth rho*; *k0* ≤ *k*; *S'* = *drop td S*⟧
⟹ 2 = *maxFreshWords*
(*rev* [((*h*, *k*) ↓ *k0*, *k0*, (*q*, *Suc* (*Suc* (*Suc* (*Suc j*)))), *Val* (*execOp oper* (*atom2val E1 a1*) (*atom2val E1 a2*)) # *S'*),
((*h*, *k*), *k0*, (*q*, *Suc* (*Suc* (*Suc j*))), *Val* (*execOp oper* (*atom2val E1 a1*) (*atom2val E1 a2*)) # *drop* (*topDepth rho*) *S*),
((*h*, *k*), *k0*, (*q*, *Suc* (*Suc j*)), *Val v* # *S*),
((*h*, *k*), *k0*, (*q*, *Suc j*), *map* (*item2Stack k S*) [*atom2item rho a1*, *atom2item rho a2*] @ *S*), ((*h*, *k*), *k0*, (*q*, *j*), *S*)])
apply (*simp*,*elim conjE*)
apply (*frule-tac a*=*a1* **and** *k*=*k* **in** *identityEnvironment-item-val*,*assumption*+)
apply (*frule-tac a*=*a2* **and** *k*=*k* **in** *identityEnvironment-item-val*,*assumption*+)
apply (*simp add*: *rev.simps add*: *maxFreshWords.simps add*: *maximum.simps add*: *map.simps*)
apply *clarsimp*
apply (*simp add*: *sizeST.simps*)
apply (*frule maxFreshWords-1*)
by *simp*

lemma *sizeST-append*:
sizeST (*xs* @ *ys*) = *sizeST xs* + *sizeST ys*
apply (*induct xs arbitrary*: *ys*) **apply** (*simp add*: *sizeST.simps*)
apply (*simp add*: *sizeST.simps*) **apply** (*case-tac a*)
apply (*simp add*: *sizeST.simps*)+
done

lemma *sizeST-list-Val*:
$\forall\ a \in set\ (map\ (item2Stack\ k\ S)\ (map\ (atom2item\ rho)\ as)).\ a = Val\ v$
$\Longrightarrow sizeST\ ((map\ (item2Stack\ k\ S)\ (map\ (atom2item\ rho)\ as))) = int\ (length\ as)$
apply (*induct as*)
apply (*simp add*: *map.simps*)
apply (*simp add*: *sizeST.simps*)
apply (*simp add*: *map.simps*)
apply (*simp add*: *sizeST.simps*)
done

lemma *sizeST-list-Reg*:
$\forall\ r \in set\ (map\ (item2Stack\ k\ S)\ (map\ (region2item\ rho)\ rr)).\ r = Reg\ r'$
$\Longrightarrow sizeST\ ((map\ (item2Stack\ k\ S)\ (map\ (region2item\ rho)\ rr))) = int\ (length\ rr)$
apply (*induct rr*)
apply (*simp add*: *map.simps*)
apply (*simp add*: *sizeST.simps*)
apply (*simp add*: *map.simps*)
apply (*simp add*: *sizeST.simps*)
done

lemma *app1*:
$[\![length\ as = length\ xs; \forall\ a \in set\ (map\ (item2Stack\ k\ S)\ (map\ (atom2item\ rho)\ as)).\ a = Val\ v;\ td < length\ S;$
$length\ rr = length\ rs; \forall\ r \in set\ (map\ (item2Stack\ k\ S)\ (map\ (region2item\ rho)\ rr)).\ r = Reg\ r']\!]$
$\Longrightarrow maxFreshWords$
$(rev\ ([((h, Suc\ k), k0, (qf, 0),$
$append\ (map\ (item2Stack\ k\ S)\ (map\ (atom2item\ rho)\ as))$
$(append\ (map\ (item2Stack\ k\ S)\ (map\ (region2item\ rho)\ rr))\ (drop\ td\ S)))] @$
$((h, k), k0, (q, Suc\ (Suc\ j)),$
$append\ (map\ (item2Stack\ k\ S)\ (map\ (atom2item\ rho)\ as))$
$(append\ (map\ (item2Stack\ k\ S)\ (map\ (region2item\ rho)\ rr))\ (drop\ td\ S))) \#$
$[((h, k), k0, (q, Suc\ j),$
$map\ (item2Stack\ k\ S)\ (map\ (atom2item\ rho)\ as)\ @\ map\ (item2Stack\ k\ S)\ (map\ (region2item\ rho)\ rr)\ @\ S)] @$
$[((h, k), k0, (q, j), S)])) = int\ (length\ xs) + int\ (length\ rs)$
apply (*simp add*: *rev.simps add*: *maxFreshWords.simps add*: *maximum.simps add*: *map.simps*)
apply (*subst sizeST-append*)+ **apply** *simp*
apply (*subst sizeST-list-Val* [**where** *v*=*v*],*simp*)+
apply (*subst sizeST-list-Reg* [**where** *r'*=*r'*],*simp*)+ **apply** *simp*

by (*frule maxFreshWords-1*,*simp*)

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

lemma *app2*:
⟦(*E1*, *E2*) ⋈ (*rho*,*S*); *td* = *topDepth rho*;
length as = *length xs*; ∀ *a* ∈ *set* (*map* (*item2Stack k S*) (*map* (*atom2item rho*) *as*)). *a* = *Val v*;
length rr = *length rs*; ∀ *r* ∈ *set* (*map* (*item2Stack k S*) (*map* (*region2item rho*) *rr*)). *r* = *Reg r'*; *topDepth rho* < *length S*⟧
⟹
sizeSVMStateST
((*h*, *Suc k*), *k0*, (*qf*, *0*),
append (*map* (*item2Stack k S*) (*map* (*atom2item rho*) *as*))
(*append* (*map* (*item2Stack k S*) (*map* (*region2item rho*) *rr*))
(*drop* (*topDepth rho*) *S*))) −
sizeSVMStateST
(*hd* (*rev* ([((*h*, *Suc k*), *k0*, (*qf*, *0*),
append (*map* (*item2Stack k S*) (*map* (*atom2item rho*) *as*))
(*append* (*map* (*item2Stack k S*) (*map* (*region2item rho*) *rr*)) (*drop* (*topDepth rho*) *S*)))] @
((*h*, *k*), *k0*, (*q*, *Suc* (*Suc j*)),
append (*map* (*item2Stack k S*) (*map* (*atom2item rho*) *as*))
(*append* (*map* (*item2Stack k S*) (*map* (*region2item rho*) *rr*)) (*drop* (*topDepth rho*) *S*))) #
[((*h*, *k*), *k0*, (*q*, *Suc j*),
append (*map* (*item2Stack k S*) (*map* (*atom2item rho*) *as*))
(*append* (*map* (*item2Stack k S*) (*map* (*region2item rho*) *rr*)) (*S*)))] @
[((*h*, *k*), *k0*, (*q*, *j*), *S*)])))
= *int* (*length xs*) + *int* (*length rs*) − *int* (*topDepth rho*)
apply (*simp add*: *rev.simps add*: *maxFreshWords.simps add*: *maximum.simps add*: *map.simps*)
apply (*subst sizeST-append*)+
apply (*subst sizeST-list-Val* [**where** *v*=*v*],*simp*)
apply (*subst sizeST-list-Reg* [**where** *r'*=*r'*],*simp*,*simp*)
by (*frule continuations-bueno*,*assumption*+,*simp*)

lemma *maxFreshWords-App*:
⟦Σ *f* = *Some* (*xs*, *rs*, *e*); *E1'* = *map-of* (*zip xs* (*map* (*atom2val E1*) *as*)); *n* = *length xs*; *l* = *length rs*;
E2' = *map-of* (*zip rs* (*map* (*the* ∘ *E2*) *rr*))(*self* ↦ *Suc k*);
length as = *length xs*; *length rr* = *length rs*;
finite (*dom h*); (*E1*, *E2*) ⋈ (*rho*,*S*); *td* = *topDepth rho*;
s0 = ((*h*, *k*), *k0*, (*q*, *j*), *S*); *td* = *topDepth rho*; *topDepth rho* < *length S*;
P⊢((*h*, *Suc k*), *k0*, (*qf*, *0*),
append (*map* (*item2Stack k S*) (*map* (*atom2item rho*) *as*))

(*append* (*map* (*item2Stack k S*) (*map* (*region2item rho*) *rr*)) (*drop td S*))) , (*n* + *l*)#*tds* −*svm*→ *sa* # *ss* , *Suc 0*#*tds*;
sa = ((*h'*, *Suc k*) ↓ *k0*, *k0*, (*q'*, *i*), *Val v* # *drop td S*);
(δ, *m*, *s*) = (δ', *ma*, *w*);
w = *maxFreshWords* (*rev* (*sa* # *ss*)); *w* > *0*⟧
⟹ *max* (*int n* + *int l*) (*s* + *int n* + *int l* − *int td*) =
maxFreshWords
(*rev* (*sa* # *ss* @
[((*h*, *k*), *k0*, (*q*, *Suc* (*Suc j*)), *append* (*append* (*map* (*item2Stack k S*) (*map* (*atom2item rho*) *as*))
(*map* (*item2Stack k S*) (*map* (*region2item rho*) *rr*))) (*drop td S*)),
((*h*, *k*), *k0*, (*q*, *Suc j*), *append* (*append* (*map* (*item2Stack k S*) (*map* (*atom2item rho*) *as*))
(*map* (*item2Stack k S*) (*map* (*region2item rho*) *rr*))) *S*),
((*h*, *k*), *k0*, (*q*, *j*), *S*)]))
apply (*subgoal-tac* ∀ *a* ∈ *set* (*map* (*item2Stack k S*) (*map* (*atom2item rho*) *as*)). *a* = *Val v*)
prefer *2* **apply** (*rule app1-1*)
apply (*subgoal-tac* ∀ *r* ∈ *set* (*map* (*item2Stack k S*) (*map* (*region2item rho*) *rr*)). *r* = *Reg r'*)
prefer *2* **apply** (*rule app1-2*)
apply (*frule-tac s*=*sa* **in** *head-execSVMBalanced*)
apply (*case-tac ss*=[],*simp*)
apply (*simp add*: *rev.simps add*: *maxFreshWords.simps add*: *maximum.simps add*: *map.simps*)
apply *clarsimp*
apply (*frule head-cons-execSVMBalanced*,*assumption*+)
apply (*erule exE*)
apply (*drule-tac t*=*last ss* **in** *sym*, *simp*)
apply (*subst cons-append-assoc* [**where** *x*=((*h'*, *Suc k*) ↓ *k0*, *k0*, (*q'*, *i*), *Val v* # *drop* (*topDepth rho*) *S*)])
apply (*subst cons-cons-append*)
apply (*subst cons-append*
[**where** *xs*=((*h*, *k*), *k0*, (*q*, *Suc* (*Suc j*)),
map (*item2Stack k S*) (*map* (*atom2item rho*) *as*) @ *map* (*item2Stack k S*) (*map* (*region2item rho*) *rr*) @ *drop* (*topDepth rho*) *S*) #
[((*h*, *k*), *k0*, (*q*, *Suc j*),
map (*item2Stack k S*) (*map* (*atom2item rho*) *as*) @ *map* (*item2Stack k S*) (*map* (*region2item rho*) *rr*) @ *S*)] @
[((*h*, *k*), *k0*, (*q*, *j*), *S*)]])
apply (*subst cons-append-assoc* [**where** *x*=((*h'*, *Suc k*) ↓ *k0*, *k0*, (*q'*, *i*), *Val v* # *drop* (*topDepth rho*) *S*)])
apply (*subst maxFreshWords-P-rev*)
apply (*subgoal-tac* ∀ *a* ∈ *set* (*map* (*item2Stack k S*) (*map* (*atom2item rho*) *as*)). *a* = *Val v*)
prefer *2* **apply** (*rule app1-1*)
apply (*subgoal-tac* ∀ *r* ∈ *set* (*map* (*item2Stack k S*) (*map* (*region2item rho*) *rr*)).

```
r = Reg r')
 prefer 2 apply (rule app1-2)
apply (subst app1 [where xs=xs and rs=rs], assumption+)
apply (subgoal-tac
  sizeSVMStateST
      ((h, Suc k), k0, (qf, 0),
          append (map (item2Stack k S) (map (atom2item rho) as))
                 (append (map (item2Stack k S) (map (region2item rho) rr))
                 (drop  (topDepth rho) S))) -
   sizeSVMStateST
              (hd (rev ([((h, Suc k), k0, (qf, 0),
                        append (map (item2Stack k S) (map (atom2item rho) as))
                             (append (map (item2Stack k S) (map (region2item rho)
rr)) ( drop (topDepth rho) S)))] @
                        ((h, k), k0, (q, Suc (Suc j)),
                         append (map (item2Stack k S) (map (atom2item rho) as))
                             (append (map (item2Stack k S) (map (region2item rho)
rr)) ( drop (topDepth rho) S))) #
                        [((h, k), k0, (q, Suc j),
                         append (map (item2Stack k S) (map (atom2item rho) as))
                             (append (map (item2Stack k S) (map (region2item rho)
rr)) ( S)))] @
                        [((h, k), k0, (q, j), S)])))
  = int (length xs) + int (length rs) - int (topDepth rho),simp)
by (rule app2, assumption+, simp, assumption+)

end
```

15 Certification of the translation CoreSafe to SVM. identityEnvironment property

```
theory identityEnvironment-lemmas
imports Main ../CoreSafe/SafeRASemantics SVMSemantics CoreSafeToSVM CertifSafeToSVM-definitions
                                        basic-properties

begin



fun addEnv-good :: Env ⇒ (string ×  nat) list ⇒ bool where
 addEnv-good rho xs = (∀ x ∈ set (map fst xs). x ∉ domRho rho)

declare identityEnvironment.simps [simp del]
declare envAdd.simps [simp del]
declare envSearch.simps [simp del]
declare decreasing.simps [simp del]
declare addEnv-good.simps [simp del]
```

declare *rho-good.simps* [*simp del*]
declare *rho-good'.simps* [*simp del*]

lemma *length-decreasing-leq-length*:
  length xs $\leq$ *length* (*decreasing* (*length xs*))
apply (*induct xs*, *simp*, *clarsimp*)
by (*subst decreasing.simps*,*simp*)

lemma *length-decreasing-eq-length*:
  (*length* (*decreasing* (*length xs*))) = *length xs*
apply (*induct xs*, *simp*)
by (*subst decreasing.simps*, *simp*)+

lemma *rho-good-addEnv* [*rule-format*]:
  length rho > *0* $\longrightarrow$ *rho-good rho*
   $\longrightarrow$ *rho-good* (*rho* + *zip xs* (*decreasing* (*length xs*)))
apply (*induct xs*, *simp*)
 apply (*case-tac rho*, *simp*, *clarsimp*)
 apply (*rename-tac delta m l ret*)
 apply (*simp add*: *envAdd.simps add*: *rho-good.simps*)
apply *clarsimp*
apply (*case-tac rho*, *simp*)
apply (*rename-tac x xs delta rest*)
apply (*subst decreasing.simps*, *simp*)
apply (*simp add*: *envAdd.simps*, *clarsimp*)
apply (*rename-tac delta m n rest*)
apply (*simp add*: *Let-def add*: *rho-good.simps*)
apply (*simp add*: *rho-good'.simps*)
apply (*rule conjI*, *rule length-decreasing-leq-length*)
apply (*rule ballI*, *rule conjI*, *rule impI*, *rule length-decreasing-leq-length*)
apply (*rule impI*, *erule conjE*)
apply (*erule-tac x=xa* **in** *ballE*) **apply** *simp*
apply *simp*
done

lemma *envSearch'-monotone* [*rule-format*]:
  rho-good' rho $\longrightarrow$ *x*$\in$ *domRho rho*
  $\longrightarrow$ *envSearch' rho x* (*Suc l*) = *Suc* (*envSearch' rho x l*)
apply (*induct rho arbitrary*: *l*, *simp*, *auto*)
apply (*simp add*: *rho-good'.simps*, *erule conjE*)
apply (*case-tac rho*) **apply** *simp*
apply (*erule-tac x=x* **in** *ballE*) **apply** (*elim conjE*) **apply** *simp*
apply (*simp add*: *dom-def*)

apply *clarsimp*
apply (*rename-tac delta m n l l' delta' m' n' rest*)
apply (*unfold rho-good'.simps*, *erule conjE*)

apply (*erule-tac x=x* **in** *ballE*) **apply** (*elim conjE*) **apply** *simp*
apply (*simp add*: *dom-def*)
apply *clarsimp*
apply (*rename-tac delta m n rho l*)
apply (*case-tac delta x*, *simp*) **apply** *simp*
apply (*erule-tac x=x* **in** *ballE*, *simp*) **apply** (*elim conjE*)
apply *simp*
by (*simp add*: *dom-def*)

lemma *envSearch'-monotone-Suc* [*rule-format*]:
rho-good' rho ⟶ *x*∈ *domRho rho*
⟶ *envSearch' rho x* (*Suc* (*Suc l*)) = *Suc* (*Suc* (*envSearch' rho x l*))
apply (*induct rho arbitrary*: *l*, *simp*, *auto*)
apply (*simp add*: *rho-good'.simps*, *erule conjE*)
apply (*case-tac rho*) **apply** *simp*
apply (*erule-tac x=x* **in** *ballE*) **apply** (*elim conjE*) **apply** *simp*
apply (*simp add*: *dom-def*)

apply *clarsimp*
apply (*rename-tac delta m n l l' delta' m' n' rest*)
apply (*unfold rho-good'.simps*, *erule conjE*)
apply (*erule-tac x=x* **in** *ballE*) **apply** (*elim conjE*) **apply** *simp*
apply (*simp add*: *dom-def*)
apply *clarsimp*
apply (*rename-tac delta m n rho l*)
apply (*case-tac delta x*, *simp*) **apply** *simp*
apply (*erule-tac x=x* **in** *ballE*, *simp*) **apply** (*elim conjE*)
apply *simp*
by (*simp add*: *dom-def*)

lemma *envAdd-monotone-dom*:
⟦*x* ≠ *x1*; *x* ∈ *domRho* (*rho* + [(*x1*, *1*)]); *length rho* > *0*⟧
⟹ *x* ∈ *domRho rho*
apply (*simp*)
apply (*case-tac rho*) **apply** *simp*
apply (*simp add*: *envAdd.simps add*: *envSearch.simps*, *clarsimp*)
done

lemma *domRho-addEnv-aux* [*rule-format*]:
x ∉ *set xs* ⟶ *distinct xs* ⟶ *i* < *length xs* ⟶ *xs* ! *i* ∈ *domRho* (*rho* + *zip* (*x* # *xs*) (*decreasing* (*Suc* (*length xs*))))
⟶ *xs* ! *i* ∈ *domRho* (*rho* + *zip xs* (*decreasing* (*length xs*)))
apply (*clarsimp*, *case-tac rho*, *simp*)
apply (*simp add*: *envAdd.simps*)
apply (*clarsimp*, *rename-tac delta m l rest*)
apply (*simp add*: *envAdd.simps add*: *Let-def*)
apply (*erule disjE*, *rule disjI1*, *simp*)

apply (*subgoal-tac*
  (*zip* (*x* # *xs*) (*decreasing* (*Suc* (*length xs*)))) =
  ((*x*,*Suc* (*length xs*)) # *zip xs* (*decreasing* (*length xs*))),*simp*)
 apply (*erule disjE*, *clarsimp*, *simp*)
apply (*subst decreasing.simps*, *simp*)
by *simp*

lemma *addEnv-good-aux* [*rule-format*]:
  x ∉ *set xs* ⟶ *distinct xs* ⟶ *addEnv-good rho* (*zip* (*x* # *xs*) (*decreasing* (*Suc* (*length xs*))))
  ⟶ *addEnv-good rho* (*zip xs* (*decreasing* (*length xs*)))
apply (*case-tac rho*, *simp*)
apply (*simp add*: *addEnv-good.simps*)
apply (*clarsimp*, *rename-tac delta m l rest*)
apply (*simp add*: *addEnv-good.simps*)
apply (*subgoal-tac*
  (*zip* (*x* # *xs*) (*decreasing* (*Suc* (*length xs*)))) =
  ((*x*,*Suc* (*length xs*)) # *zip xs* (*decreasing* (*length xs*))))
apply (*rule ballI*)
apply (*erule-tac x*=*xa* **in** *ballE*, *clarsimp*, *clarsimp*)
by (*subst decreasing.simps*, *simp*)

lemma *z-in-domRho-addEnv* [*rule-format*]:
length rho > *0* ⟶ *distinct xs* ⟶ *i* < *length xs* ⟶ *xs*!*i* ∈ *domRho* (*rho* + *zip xs* (*decreasing* (*length xs*))) ⟶ *rho-good rho*
  ⟶ *addEnv-good rho* (*zip xs* (*decreasing* (*length xs*)))
  ⟶ (*rho* + *zip xs* (*decreasing* (*length xs*))) |> (*xs* ! *i*) = *i*
apply (*induct xs i rule*: *list-induct3*, *simp-all*)

apply (*clarsimp*, *subst decreasing.simps*, *simp*, *simp add*: *envAdd.simps*)
apply (*case-tac rho*, *simp*, *clarsimp*)
apply (*simp add*: *Let-def*)
apply (*rename-tac x xs delta m rest*)
apply (*simp add*: *envSearch.simps*)

apply *clarsimp*
apply (*subgoal-tac xs* ! *i* ∈ *domRho* (*rho* + *zip xs* (*decreasing* (*length xs*))),*simp*)
 prefer *2* **apply** (*rule domRho-addEnv-aux*, *assumption*+)
apply (*subgoal-tac addEnv-good rho* (*zip xs* (*decreasing* (*length xs*))),*simp*)
 prefer *2* **apply** (*rule addEnv-good-aux*, *assumption*+)
apply (*case-tac xs*=[],*simp*)
apply (*subgoal-tac length rho* > *0*) **prefer** *2* **apply** *simp*
apply (*frule-tac xs*=*xs* **in** *rho-good-addEnv*,*assumption*)
apply (*case-tac rho*, *simp*)
apply (*rename-tac tup rest*, *clarsimp*)
apply (*rename-tac delta m n rest*)
apply (*subst decreasing.simps*, *simp*)

apply (*simp add*: *envAdd.simps add*: *Let-def add*: *envSearch.simps*)
apply (*rule conjI*, *clarsimp*, *rule impI*)
apply (*case-tac* ((*delta* ++ *map-of* (*map* (λ(*x*, *j*). (*x*, *m* + *j*)) (*zip xs* (*decreasing* (*length xs*))))) (*xs* ! *i*)))
apply (*rule-tac s=None* **in** *ssubst*,*assumption*, *clarsimp*)
apply (*erule disjE*)+ **apply** (*simp add*: *dom-def*)
apply (*erule disjE*)+ **apply** (*simp add*: *dom-def*)
apply (*unfold rho-good.simps*, *simp*) **apply** (*elim conjE*)
apply (*unfold rho-good'.simps*, *simp*) **apply** (*elim conjE*)
apply (*subst envSearch'-monotone*, *simp*, *assumption*+, *simp*)
apply (*erule disjE*)+ **apply** (*simp add*: *dom-def*)
apply (*subst envSearch'-monotone*, *simp*, *assumption*+, *simp*)
apply (*erule disjE*)+ **apply** (*simp add*: *dom-def*)
apply (*subst envSearch'-monotone*, *simp*, *assumption*+, *simp*)
apply (*erule disjE*)+ **apply** (*simp add*: *dom-def*)
apply (*subst envSearch'-monotone*, *simp*, *assumption*+, *simp*)
apply (*rename-tac l*)
apply (*rule-tac s=Some l* **in** *ssubst*,*assumption*, *simp*)
apply (*subgoal-tac* (*length* (*decreasing* (*length xs*))) = *length xs*,*simp*)
prefer *2* **apply** (*rule length-decreasing-eq-length*)
apply (*elim conjE*, *erule-tac x=xs*!*i* **in** *ballE*)
apply (*elim conjE*)
apply (*erule-tac x=xs*!*i* **in** *ballE*)
apply (*elim conjE*,*simp*)
apply (*simp*, *elim conjE*, *simp add*: *addEnv-good.simps*)
apply (*subgoal-tac xs* ! *i* ∈ *dom* (*map-of* (*map* (λ(*x*, *j*). (*x*, *m* + *j*)) (*zip xs* (*decreasing* (*length xs*))))),*simp*)
apply (*erule-tac x=xs*!*i* **in** *ballE*)
apply (*elim conjE*,*simp*)
apply *simp*
by *clarsimp*

lemma *z-in-domRho* [*rule-format*]:
length rho > *0* ⟶ *z* ∉ *set xs* ⟶ *distinct xs* ⟶ *z* ∈ *domRho rho* ⟶ *rho-good rho*
⟶ *addEnv-good rho* (*zip xs* (*decreasing* (*length xs*)))
⟶ (*rho* + *zip xs* (*decreasing* (*length xs*))) |> *z* = (*rho* |> *z*) + *length xs*
apply (*induct xs*, *simp-all*)

apply (*clarsimp*, *simp add*: *envAdd.simps*)
apply (*case-tac rho*, *simp*, *clarsimp*)
apply (*simp add*: *envSearch.simps*)

apply *clarsimp*
apply (*subgoal-tac addEnv-good rho* (*zip xs* (*decreasing* (*length xs*))),*simp*)
prefer *2* **apply** (*rule-tac x=a* **in** *addEnv-good-aux*, *assumption*+)
apply (*case-tac xs*=[],*simp*)
apply (*subgoal-tac length rho* > *0*) **prefer** *2* **apply** *simp*

apply (*frule-tac xs=xs* **in** *rho-good-addEnv*,*assumption*)
apply (*case-tac rho*, *simp*)
apply (*rename-tac tup rest*, *clarsimp*, *rename-tac delta m n rest*)
apply (*subst decreasing.simps*, *simp*, *simp add*: *envAdd.simps*)
apply (*simp add*: *Let-def add*: *envSearch.simps*)
apply (*erule disjE*)
apply (*case-tac delta z*,*simp*)
apply (*simp add*: *dom-def*)
apply (*simp*, *unfold rho-good.simps*, *simp*, *elim conjE*)
apply (*simp*, *unfold rho-good′.simps*)
apply (*elim conjE*, *erule-tac x=z* **in** *ballE*)
apply (*elim conjE*) **apply** *simp*
apply (*simp add*: *dom-def*)
apply *clarsimp*
apply (*case-tac delta z*,*simp*)
apply (*subst envSearch′-monotone*, *simp*, *assumption*+, *simp*)
apply *clarsimp*
apply (*erule-tac x=z* **in** *ballE*)
apply (*erule-tac x=z* **in** *ballE*, *clarsimp*)
apply *arith*
apply (*simp add*: *dom-def*)
apply (*simp add*: *dom-def*)
apply (*rename-tac x xs*)
apply (*subgoal-tac length rho* > *0*) **prefer** *2* **apply** *simp*
apply (*frule-tac xs=xs* **in** *rho-good-addEnv*,*assumption*)
apply (*case-tac rho*, *simp*)
apply (*rename-tac tup rest*, *clarsimp*, *rename-tac delta m n rest*)
apply (*subst decreasing.simps*, *simp*)
apply (*simp add*: *envAdd.simps add*: *Let-def add*: *envSearch.simps*)
apply (*case-tac* ((*delta* ++ *map-of* (*map* (λ(*x*, *j*). (*x*, *m* + *j*)) (*zip xs* (*decreasing* (*length xs*))))) *z*))
apply (*rule-tac s=None* **in** *ssubst*,*assumption*, *clarsimp*)
apply (*erule disjE*)+ **apply** (*simp add*: *dom-def*)
apply (*unfold rho-good.simps*,*simp*, *elim conjE*)
apply (*unfold rho-good′.simps*)
apply (*subst envSearch′-monotone*,*simp*, *assumption*+, *simp*)

apply (*rename-tac l*,*clarsimp*)
apply (*drule-tac s=m* + *min* (*length xs*) (*length* (*decreasing* (*length xs*))) − *l* **in** *sym*, *simp*)
apply (*erule-tac x=z* **and**
A=dom (*map-of* (*map* (λ(*x*, *j*). (*x*, *m* + *j*)) (*zip xs* (*decreasing* (*length xs*))))) ∪ *dom delta* **in** *ballE*)
apply (*elim conjE*)
apply (*erule-tac P=map-of* (*map* (λ(*x*, *j*). (*x*, *m* + *j*)) (*zip xs* (*decreasing* (*length xs*)))) *z* = *Some l* **in** *disjE*,*simp*)
apply *clarsimp*
apply (*subgoal-tac* (*delta* ++ *map-of* (*map* (λ(*x*, *j*). (*x*, *m* + *j*)) (*zip xs* (*decreasing* (*length xs*))))) *z* = *Some l*,*simp*)

apply (*simp add*: *map-add-def*)
by *clarsimp*

lemma *stackobject-nth* [*rule-format*]:
length vs > *0* ⟶ *i* < *length vs* ⟶ ((*map Val vs*) @ *S*) ! *i* = *Val* (*vs*!*i*)
apply (*induct vs i rule*: *list-induct3*, *simp-all*)
by *clarsimp*

lemma *stackobject-nth-plus* [*rule-format*]:
length vs > *0* ⟶ *i* < *length vs* ⟶ ((*map Val vs*) @ *S*) !+ *i* = *Val* (*vs*!*i*)
apply (*induct vs i rule*: *list-induct3*, *simp-all*)
by *clarsimp*

lemma *stackobject-envSearch* [*rule-format*]:
length vs > *0* ⟶ (*map Val vs* @ *S*) ! ((*rho* |> *z*) + *length vs*) = *S* ! *rho* |> *z*
apply (*induct vs*, *simp-all*)
by (*case-tac vs*=[], *simp*, *simp*)

lemma *stackobject-envSearch-plus* [*rule-format*]:
length vs > *0* ⟶ (*map Val vs* @ *S*) !+ (((*rho* |> *z*) + *length vs*)) = *S* !+ (*rho* |> *z*)
apply (*induct vs*, *simp-all*)
by (*case-tac vs*=[], *simp*, *simp*)

lemma *extend-nth* [*rule-format*]:
length xs > *0* ⟶ *length xs* = *length vs* ⟶ *distinct xs* ⟶ *i* < *length xs* ⟶ *extend E1 xs vs* (*xs* ! *i*) = *Some* (*vs* ! *i*)
apply (*induct xs vs arbitrary*: *i rule*: *list-induct2′*, *simp-all*)
apply (*case-tac xs*=[], *simp*)
apply (*simp add*: *extend-def*, *clarsimp*)
apply (*case-tac i*, *simp*)
apply (*simp add*: *extend-def*)
apply (*simp*)
apply (*simp add*: *extend-def*)
by (*rule impI*,*clarsimp*)

lemma *extend-nth-the* [*rule-format*]:
length xs > *0* ⟶ *length xs* = *length vs* ⟶ *distinct xs* ⟶ *i* < *length xs* ⟶ *the* (*extend E1 xs vs* (*xs* ! *i*)) = (*vs* ! *i*)
apply (*induct xs vs arbitrary*: *i rule*: *list-induct2′*, *simp-all*)
apply (*case-tac xs*=[], *simp*)
apply (*simp add*: *extend-def*, *clarsimp*)
apply (*case-tac i*, *simp*)
apply (*simp add*: *extend-def*)
apply (*simp*)
apply (*simp add*: *extend-def*)

by (*rule impI*,*clarsimp*)

lemma *not-in-extend*: $z \notin set\ xs \Longrightarrow E1\ z = extend\ E1\ xs\ vs\ z$
apply (*induct xs vs rule*: *list-induct2'*)
by (*simp*, *simp add*: *extend-def*)+

lemma *dom-decreasing*:
$[\![length\ xs = length\ vs]\!]$
$\Longrightarrow dom\ (map\text{-}of\ (map\ (\lambda(x, j).\ (x, m + j))\ (zip\ xs\ (decreasing\ (length\ vs)))))$
$= set\ xs$
apply (*induct xs vs rule*:*list-induct2'*, *simp-all*)
by (*simp add*: *decreasing.simps*)

lemma *domRho-case*:
$[\![length\ rho > 0;\ domRho\ rho = dom\ E1 \cup dom\ E2 - \{self\};\ \forall\ x \in set\ xs.\ x \neq self;\ length\ xs = length\ vs;\ distinct\ xs]\!]$
$\Longrightarrow domRho\ (rho + zip\ xs\ (decreasing\ (length\ vs))) = dom\ (extend\ E1\ xs\ vs) \cup dom\ E2 - \{self\}$
apply (*unfold envAdd.simps*)
apply (*case-tac rho*) **apply** (*simp*)
apply (*clarsimp*)
apply (*simp add*: *Let-def*)
apply (*simp add*: *extend-def*)
apply (*subgoal-tac dom* (*map-of* (*map* ($\lambda(x, j).\ (x, aa + j)$) (*zip xs* (*decreasing* (*length vs*))))) = *set xs*)
apply (*subgoal-tac dom* (*map-of* (*zip xs vs*)) = *set xs*)
apply *simp* **apply** *blast* **apply** *simp*
by (*rule dom-decreasing*,*assumption*)

lemma *envAdd-list-monotone-dom*:
$[\![x \notin set\ xs;\ x \in domRho\ (rho + zip\ xs\ (decreasing\ (length\ xs)));\ length\ rho > 0]\!]$
$\Longrightarrow x \in domRho\ rho$
apply (*case-tac rho*, *simp*)
apply (*simp add*: *envAdd.simps*, *clarsimp*, *simp add*: *Let-def*)
apply (*erule disjE*)
apply (*subgoal-tac length* (*decreasing* (*length xs*)) = *length xs*)
apply (*drule-tac t*=*length xs* **in** *sym*)
apply (*frule-tac m*=*aa* **in** *dom-decreasing*, *simp*)
apply (*rule length-decreasing-eq-length*)
by (*simp add*: *dom-def*)

lemma *set-conv-nth-not*: $\forall i{<}length\ xs.\ z \neq xs\ !\ i \Longrightarrow z \notin set\ xs$
by (*auto simp*:*set-conv-nth*)

lemma *envPlusPlus-rho-suc-rho* [*rule-format*]:
$length\ rho > 0 \longrightarrow rho\text{-}good\ rho \longrightarrow x \in domRho\ rho \longrightarrow rho\ ++\ |>\ x = rho\ |>\ x + 2$

apply (*induct rho*,*simp*,*clarsimp*)
apply (*rename-tac delta m n rho*)
apply (*simp add*: *envPlusPlus.simps add*: *envSearch.simps add*: *rho-good.simps*, *elim conjE*, *simp add*: *rho-good'.simps*, *auto*)
apply (*erule-tac x*=*x* **in** *ballE*, *erule conjE*, *simp*, *simp add*: *dom-def*)+
apply (*case-tac delta x*,*simp*)
apply (*case-tac rho*,*simp*)
apply (*rule envSearch'-monotone-Suc*, *simp*, *assumption*+)
apply (*erule-tac x*=*x* **in** *ballE*,*simp*, *arith*)
apply (*simp add*: *dom-def*)
apply (*case-tac delta x*,*simp*)
apply (*case-tac rho*,*simp*)
apply (*rule envSearch'-monotone-Suc*, *simp*, *assumption*+)
apply (*erule-tac x*=*x* **in** *ballE*,*simp*, *arith*)
by (*simp add*: *dom-def*)

lemma *envPlusPlus-monotone-dom*:
length rho > *0* ⟹ *domRho rho* = *domRho* (*rho* ++)
by (*simp*, *case-tac rho*,*simp*,*clarsimp*)

lemma *envAdd-rho-suc-rho* [*rule-format*]:
length rho > *0* ⟶ *rho-good rho* ⟶ *x*≠*x1* ⟶ *x*∈ *domRho rho* ⟶ (*rho* + [(*x1*, *1*)]) |> *x* = (*rho* |> *x*) + *1*
apply (*induct rho*,*simp*,*clarsimp*)
apply (*rename-tac delta m n rho*)
apply (*simp add*: *envAdd.simps add*: *envSearch.simps add*: *rho-good.simps*, *elim conjE*, *simp add*: *rho-good'.simps*, *auto*)
apply (*erule-tac x*=*x* **in** *ballE*, *erule conjE*, *simp*, *simp add*: *dom-def*)+
apply (*case-tac delta x*,*simp*)
apply (*case-tac rho*,*simp*)
apply (*rule envSearch'-monotone*, *simp*, *assumption*+)
apply (*erule-tac x*=*x* **in** *ballE*,*simp*, *arith*)
apply (*simp add*: *dom-def*)
apply (*case-tac delta x*,*simp*)
apply (*case-tac rho*,*simp*)
apply (*rule envSearch'-monotone*, *simp*, *assumption*+)
apply (*erule-tac x*=*x* **in** *ballE*,*simp*, *arith*)
by (*simp add*: *dom-def*)

lemma *identityEnvironment-e1*:
⟦ *length rho* > *0*; (*E1*,*E2*) ⋈ (*rho*,*S*); *rho-good rho*; *self* ∈ *dom* (*snd* (*E1*, *E2*)); *dom E1* ∩ *dom E2* = {} ⟧
⟹ (*E1*,*E2*) ⋈ (*rho*++,*Cont* (*k0*, *q2*) # *S*)
apply (*subgoal-tac self* ∉ *dom E1*)
prefer *2* **apply** *simp* **apply** *blast*
apply (*unfold identityEnvironment.simps*)
apply (*elim conjE*)

apply (*subgoal-tac domRho rho = domRho* (*rho* ++))
apply (*rule conjI*, *simp*)
apply (*rule conjI*)
apply (*rule ballI*)
apply (*erule-tac x=x* **in** *ballE*)
apply (*subgoal-tac x∈ domRho rho*)
apply (*frule envPlusPlus-rho-suc-rho*) **apply** *simp* **apply** *simp* **apply** *simp* **apply** *blast*
apply (*simp add*: *dom-def*)
apply (*rule ballI*)
apply (*erule-tac x=r* **and** *A=dom E2* − {*self*} **in** *ballE*)
apply (*subgoal-tac r∈ domRho rho*)
apply (*frule envPlusPlus-rho-suc-rho*) **apply** *simp* **apply** *simp* **apply** *simp* **apply** *blast*
apply (*simp add*: *dom-def*)
by (*erule envPlusPlus-monotone-dom*)

lemma *identityEnvironment-e2*:
⟦(*E1*,*E2*) ⋈ (*rho*,*S*); *length rho* > *0*; *x1* ∉ *dom E1*; *x1* ≠ *self*; *rho-good rho*; *self* ∈ *dom E2*; *dom* (*E1*(*x1* ↦ *v1*)) ∩ *dom E2* = {}; *boundVar* (*Let x1* = *ConstrE C as r a′ In e2 a*) ∩ *dom E2* = {}⟧
⟹ (*E1*(*x1* ↦ *v1*), *E2*) ⋈ (*rho* + [(*x1*, *1*)],*Val v1* # *S*)
apply (*subgoal-tac dom E1* ∩ *dom E2* = {})
prefer *2* **apply** *simp*
apply (*unfold identityEnvironment.simps*)
apply (*elim conjE*)
apply (*rule conjI*)
prefer *2*
apply (*rule conjI*)
apply (*rule ballI*)
apply (*erule-tac x=x* **in** *ballE*)
apply (*case-tac x=x1*)
apply (*subgoal-tac* (*Val v1* # *S*) !+ ((*rho* + [(*x1*, *1*)]) |> *x1*) = (*Val v1* # *S*) ! *0*)
apply (*subgoal-tac* (*Val v1* # *S*) ! *0* = *Val v1*)
apply (*simp*, *simp*, *simp add*: *envAdd.simps add*: *envSearch.simps*, *induct rho*, *simp*, *clarsimp*)

apply (*subgoal-tac* (*Val v1* # *S*) !+ ((*rho* + [(*x1*, *1*)]) |> *x*) = (*Val v1* # *S*) !+ ((*rho* |> *x*) + *1*))
prefer *2* **apply** (*frule-tac x=x* **in** *envAdd-rho-suc-rho*,*assumption*+)
apply (*subgoal-tac* ∀ *x* ∈ *dom E1*. *x* ≠ *self*)
prefer *2* **apply** *blast* **apply** *simp* **apply** *blast*
apply *simp*
apply (*subgoal-tac* (*Val v1* # *S*) !+ (((*rho* |> *x*) + *1*)) = *S* !+ (*rho* |> *x*))
prefer *2* **apply** (*induct rho*,*simp*,*simp*)
apply (*case-tac S* !+ (*rho* |> *x*))
apply (*simp*,*simp*,*simp*)

apply (*case-tac x=x1*)
apply (*subgoal-tac* (*Val v1 # S*) !+ ((*rho* + [(*x1*, *1*)]) |> *x1*) = (*Val v1 # S*) ! *0*)
apply (*subgoal-tac* (*Val v1 # S*) ! *0* = *Val v1*)
apply (*simp*, *simp*, *simp add*: *envAdd.simps add*: *envSearch.simps*, *induct rho*, *simp*, *clarsimp*)
apply (*subgoal-tac x* ∉ *domRho* (*rho* + [(*x1*, *1*)]))
prefer *2* **apply** (*simp add*: *envAdd.simps add*: *envSearch.simps*, *induct rho*, *simp*, *clarsimp*)
apply *simp*

apply (*rule ballI*)
apply (*erule-tac x=r* **and** *A=dom E2* − {*self*} **in** *ballE*)
apply (*subgoal-tac r≠x1*)
apply (*subgoal-tac r* ∈ *domRho rho*)
apply (*subgoal-tac length rho* > *0*)
apply (*subgoal-tac* (*Val v1 # S*) !+ ((*rho* + [(*x1*, *1*)]) |> *r*) = (*Val v1 # S*) !+ ((*rho* |> *r*) + *1*))
prefer *2* **thm** *envAdd-rho-suc-rho* **apply** (*frule-tac x=r* **in** *envAdd-rho-suc-rho*,*assumption*+)

apply *simp*
apply (*subgoal-tac* (*Val v1 # S*) !+ (((*rho* |> *r*) + *1*)) = *S* !+ (*rho* |> *r*))
prefer *2* **apply** (*induct rho*,*simp*,*simp*)
apply (*case-tac* *S* !+ (*rho* |> *r*))
apply (*simp*,*simp*,*simp*) **apply** *simp* **apply** *blast* **apply** *blast* **apply** *simp*
apply (*unfold envAdd.simps*)
by (*case-tac rho*, *simp*, *clarsimp*, *blast*)

lemma *identityEnvironment-e2-let2*:
⟦(*E1*,*E2*) ⋈ (*rho*,*S*); *length rho* > *0*; *x1* ∉ *dom E1*; *x1* ≠ *self*; *rho-good rho*; *self* ∈ *dom E2*; *dom* (*E1*(*x1* ↦ *Loc pa*)) ∩ *dom E2* = {}; *boundVar* (*Let x1* = *ConstrE C as r a′ In e2 a*) ∩ *dom E2* = {}⟧
⟹ (*E1*(*x1* ↦ *Loc pa*), *E2*) ⋈ (*rho* + [(*x1*, *1*)] , *Val* (*Loc pa*) # *S*)
by (*rule identityEnvironment-e2*,*assumption*+)

lemma *addEnv-empty* [*rule-format*]:
length rho > *0* ⟶ *rho-good rho* ⟶ *rho* + [] = *rho*
apply (*induct rho*,*simp*,*clarsimp*)
apply (*case-tac rho*,*clarsimp*)
apply (*simp add*: *envAdd.simps add*: *rho-good.simps*)
apply *clarsimp*
by (*simp add*: *envAdd.simps add*: *rho-good.simps*)

lemma *addEnv-dom-empty* [*rule-format*]:
length rho > *0* ⟶ *domRho* (*rho* + []) = *domRho rho*
apply (*induct rho*,*simp*,*simp*)

apply (*case-tac rho*, *simp*)
apply (*simp add*: *envAdd.simps*,*clarsimp*)
by (*simp add*: *envAdd.simps*,*clarsimp*)

lemma *extend-empty*:
  extend E1 [] [] = *E1*
by (*simp add*: *extend-def*)

lemma *qq3* [*rule-format*]:
  i < *length xs*
   ⟶ *length rho* > *0*
   ⟶ *xs*!*i* ∈ *domRho* (*rho* + *zip xs* (*decreasing* (*length xs*)))
apply (*induct xs i rule*: *list-induct3*, *simp-all*)
 apply (*case-tac rho*,*simp*,*clarsimp*)
 apply (*rename-tac delta m l ret*)
 apply (*simp add*: *envAdd.simps add*: *rho-good.simps*)
 apply (*simp add*: *Let-def*)
 apply (*subst decreasing.simps*, *simp*,*clarsimp*)
apply (*case-tac rho*,*simp*,*clarsimp*)
apply (*rename-tac delta m l ret*)
apply (*simp add*: *envAdd.simps add*: *rho-good.simps*)
apply (*simp add*: *Let-def*)
by (*subst decreasing.simps*, *simp*)

lemma *pattern-extend-case-good* [*rule-format*]:
  i < *length alts*
   ⟶ *alts*!*i* = (*pati*, *ei*)
   ⟶ *pati* = *ConstrP C ps ms*
   ⟶ *xs* = *map pat2var ps*
   ⟶ *boundVar* (*Case* (*VarE x a*) *Of alts a'*) ∩ *dom E2* = {}
   ⟶ *set xs* ∩ *dom E2* = {}
apply (*induct alts arbitrary*: *i*,*simp*,*simp*)
by (*case-tac i*,*clarsimp*,*auto*)

lemma *pattern-extend-cased-good* [*rule-format*]:
  i < *length alts*
   ⟶ *alts*!*i* = (*pati*, *ei*)
   ⟶ *pati* = *ConstrP C ps ms*
   ⟶ *xs* = *map pat2var ps*
   ⟶ *boundVar* (*CaseD* (*VarE x a*) *Of alts a'*) ∩ *dom E2* = {}
   ⟶ *set xs* ∩ *dom E2* = {}
apply (*induct alts arbitrary*: *i*,*simp*,*simp*)
by (*case-tac i*,*clarsimp*,*auto*)

lemma *identityEnvironment-case-aux*:
  ⟦*length rho* > *0*;
   nr = *int* (*length vs*); *rho-good rho*;
   addEnv-good rho (*zip xs* (*decreasing* (*length xs*)));

```
   (E1, E2) ⋈ (rho,S); def-extend E1 xs vs;
   self ∈ dom E2;  dom E1 ∩ dom E2 = {};
   set xs ∩ dom E2 = {}⟧
   ⟹ (extend E1 xs vs, E2) ⋈ ( rho + zip xs (decreasing (length xs)) , map Val
vs @ S)
apply (unfold def-extend-def, elim conjE)
apply (case-tac xs=[])
 apply simp
  apply (simp add: extend-def, subst addEnv-empty,simp,assumption,simp)
apply (unfold identityEnvironment.simps)
apply (elim conjE)
apply (rule conjI)
 apply (clarsimp, rule sym)
 apply (rule domRho-case, simp, rule sym, simp, assumption+)

apply (rule conjI)
apply (rule ballI) apply (rename-tac z)
apply (erule-tac x=z in ballE)
 apply (case-tac ∃ i<length xs. z = xs!i)
  apply (erule exE, erule conjE)
  apply (subgoal-tac length rho > 0) prefer 2 apply simp
  apply (frule-tac i=i in z-in-domRho-addEnv,assumption+,simp)
  apply (drule sym [where t=length vs]) apply simp
  apply (frule qq3) apply simp apply simp apply simp apply simp
  apply (subst stackobject-nth-plus, simp, clarsimp,clarsimp,simp)
  apply (rule extend-nth-the) apply (simp,clarsimp,assumption,simp,simp)

 apply simp
 apply (drule-tac t=length vs in sym,simp)
 apply (frule set-conv-nth-not)
 apply (subgoal-tac z ∈ domRho rho)
  apply (subgoal-tac length rho > 0) prefer 2 apply simp
  apply (frule z-in-domRho, assumption+, simp)
  apply (subgoal-tac (map Val vs @  S) !+ (((rho |> z) + length vs)) =
                  S !+ (rho |> z)) prefer 2 apply (rule stackobject-envSearch-plus,
simp)
  apply simp
  apply (erule-tac x=z in ballE)
   apply (case-tac  S !+ (rho |> z), simp)
    apply (subgoal-tac E1 z = extend E1 xs vs z,simp)
    apply (rule not-in-extend, assumption+)
   apply simp
   apply simp
  apply (simp add: extend-def)
  apply (simp add: extend-def) apply blast

apply simp
apply (subgoal-tac z ∈ domRho rho)
 apply (subgoal-tac length rho > 0) prefer 2 apply simp
```

apply (*frule z-in-domRho*, *assumption+*, *simp*)
apply (*subgoal-tac* (*map Val vs* @ *S*) !+ (((*rho* |> *z*) + *length vs*)) =
S !+ (*rho* |> *z*)) **prefer** *2* **apply** (*rule stackobject-envSearch-plus*,
simp,*clarsimp*)
apply *simp*
apply (*erule-tac x*=*z* **in** *ballE*)
apply (*case-tac* *S* !+ (*rho* |> *z*), *simp*)
apply (*subgoal-tac E1 z* = *extend E1 xs vs z*,*simp*)
apply (*rule not-in-extend*, *assumption+*)
apply *simp*
apply *simp*
apply (*simp add*: *extend-def*)
apply (*simp add*: *extend-def*)
apply (*subgoal-tac z* ≠ *self*,*blast*)
apply *blast*

apply (*rule ballI*) **apply** (*rename-tac z*)
apply (*erule-tac x*=*z* **and** *A*=*dom E2* − {*self*} **in** *ballE*)
prefer *2* **apply** *simp*

apply *simp*
apply (*subgoal-tac z* ∉ *set xs*)
apply (*subgoal-tac z* ∈ *domRho rho*)
apply (*subgoal-tac length rho* > *0*) **prefer** *2* **apply** *simp*
apply (*frule z-in-domRho*, *assumption+*, *simp*)
apply (*subgoal-tac* (*map Val vs* @ *S*) !+ (((*rho* |> *z*) + *length vs*)) =
S !+ (*rho* |> *z*)) **prefer** *2* **apply** (*rule stackobject-envSearch-plus*,
simp,*clarsimp*)
apply *simp*
apply *blast*
apply (*elim conjE*)
by *blast*

lemma *identityEnvironment-case*:
⟦*length rho* > *0*;
i < *length alts*;
alts!*i* = (*pati*, *ei*);
pati = *ConstrP C ps ms*;
xs = *map pat2var ps*;
nr = *int* (*length vs*); *rho-good rho*;
addEnv-good rho (*zip xs* (*decreasing* (*length xs*)));
(*E1*, *E2*) ⋈ (*rho*,*S*); *def-extend E1 xs vs*;
self ∈ *dom E2*; *dom* (*extend E1 xs vs*) ∩ *dom E2* = {};
boundVar (*Case* (*VarE x a*) *Of alts a'*) ∩ *dom E2* = {}⟧
⟹ (*extend E1 xs vs*, *E2*) ⋈ (*rho* + *zip xs* (*decreasing* (*length xs*)) , *map Val vs* @ *S*)
apply (*frule-tac ?E2.0*=*E2* **in** *pattern-extend-case-good*,*assumption+*)
apply (*erule-tac V*=*xs* = *map pat2var ps* **in** *thin-rl*)
apply (*subgoal-tac dom E1* ∩ *dom E2* = {})

prefer *2* **apply** (*simp add*: *extend-def*) **apply** *blast*
by (*rule identityEnvironment-case-aux*,*assumption*+)

lemma *identityEnvironment-cased*:
⟦*length rho* > *0*;
i < *length alts*;
alts!*i* = (*pati*, *ei*);
pati = *ConstrP C ps ms*;
xs = *map pat2var ps*;
nr = *int* (*length vs*); *rho-good rho*;
addEnv-good rho (*zip xs* (*decreasing* (*length xs*)));
(*E1*, *E2*) ⋈ (*rho*,*S*); *def-extend E1 xs vs*;
self ∈ *dom E2*; *dom* (*extend E1 xs vs*) ∩ *dom E2* = {};
boundVar (*CaseD* (*VarE x a*) *Of alts a'*) ∩ *dom E2* = {}⟧
⟹ (*extend E1 xs vs*, *E2*) ⋈ (*rho* + *zip xs* (*decreasing* (*length xs*)) , *map Val vs* @ *S*)
apply (*frule-tac ?E2.0=E2* **in** *pattern-extend-cased-good*,*assumption*+)
apply (*erule-tac V=xs* = *map pat2var ps* **in** *thin-rl*)
apply (*subgoal-tac dom E1* ∩ *dom E2* = {})
prefer *2* **apply** (*simp add*: *extend-def*) **apply** *blast*
by (*rule identityEnvironment-case-aux*,*assumption*+)

declare *item2Stack.simps* [*simp del*]
declare *atom2item.simps* [*simp del*]

lemma *length-xs-eq-map-item-atom-as*:
length as = *length xs* ⟹ *length xs* = *length* (*map* (*item2Stack k S*) (*map* (*atom2item rho*) *as*))
by (*induct as*,*clarsimp*,*clarsimp*)

lemma *length-decreasing-add-eq-length-add*:
length (*decreasing* (*length xs* + *length ys*)) = (*length xs* + *length ys*)
apply (*induct xs ys rule*:*list-induct2'*)
apply (*simp*, *subst decreasing.simps*,*simp*)
apply (*simp*, *subst decreasing.simps*, *simp*, *rule length-decreasing-eq-length*)
apply (*simp*, *subst decreasing.simps*, *simp*, *rule length-decreasing-eq-length*)
by (*simp*, *subst decreasing.simps*,*simp*, *subst decreasing.simps*,*simp*)

lemma *dom-map-zip* [*rule-format*]:
distinct xs ⟶ *length xs* = *length ys*
⟶ *dom* (*map-of* (*map* (λ(*x*, *y*). (*x*, *y*)) (*zip xs ys*))) = *set xs*
apply (*induct xs arbitrary*: *ys*, *simp*, *clarsimp*)
by (*case-tac ys*,*simp*,*clarsimp*)

lemma *dom-map-zip2* [*rule-format*]:
distinct xs $\longrightarrow$ *length xs* = *length ys*
$\longrightarrow$ *dom* (*map-of* (*zip xs ys*)) = *set xs*
by (*induct xs arbitrary*: *ys*, *simp*, *clarsimp*)

lemma *domRho-app*:
⟦ *distinct* (*xs* @ *rs*); *self* ∉ *set xs*; *self* ∉ *set rs*;
length as = *length xs*; *length rr* = *length rs*;
E1′ = *map-of* (*zip xs* (*map* (*atom2val E1*) *as*));
E2′ = *map-of* (*zip rs* (*map* (*the* ∘ *E2*) *rr*))(*self* ↦ *Suc k*) ⟧
⟹ *dom E1′* ∪ *dom E2′* − {*self*} = *domRho* ([(*empty*, *0*, *0*)] + *zip* (*xs* @ *rs*) (*decreasing* (*length xs* + *length rs*)))
apply *clarsimp*
apply (*simp add*: *envAdd.simps*, *simp add*: *Let-def*)
apply (*subst dom-map-zip*, *simp*, *subst length-decreasing-add-eq-length-add*, *simp*)
apply (*subgoal-tac* (*set xs* ∪ *set rs*) ∩ {*self*} = {},*simp*)
by *blast*

lemma *xs-i-in-domRho*:
⟦*distinct xs*; *distinct rs*; *set xs* ∩ *set rs* = {}; *i* < *length xs*⟧
⟹ *xs* ! *i* ∈ *domRho* ([(*empty*, *0*, *0*)] + *zip* (*xs* @ *rs*) (*decreasing* (*length xs* + *length rs*)))
apply (*simp add*: *envAdd.simps*, *simp add*: *Let-def*)
apply (*subst dom-map-zip*, *simp*)
apply (*subst length-decreasing-add-eq-length-add*, *simp*)
by *clarsimp*

lemma *rs-j-in-domRho*:
⟦*distinct xs*; *distinct rs*; *set xs* ∩ *set rs* = {}; *j* < *length rs*⟧
⟹ *rs* ! *j* ∈ *domRho* ([(*empty*, *0*, *0*)] + *zip* (*xs* @ *rs*) (*decreasing* (*length xs* + *length rs*)))
apply (*simp add*: *envAdd.simps*, *simp add*: *Let-def*)
apply (*subst dom-map-zip*, *simp*)
apply (*subst length-decreasing-add-eq-length-add*,*simp*)
by *clarsimp*

lemma *rho-good-xs*:
rho-good ([(*empty*, *0*, *0*)] + *zip xs* (*decreasing* (*length xs*)))
apply (*induct xs*)
apply (*simp add*: *envAdd.simps*, *simp add*: *Let-def*, *simp add*: *rho-good.simps*, *simp add*: *rho-good′.simps*)
apply (*clarsimp*, *subst decreasing.simps*, *simp*)
apply (*simp add*: *envAdd.simps*, *simp add*: *Let-def*, *simp add*: *rho-good.simps*, *simp add*: *rho-good′.simps*)
apply (*rule conjI*)

apply (*rule length-decreasing-leq-length*)
apply (*rule ballI*)
apply (*rule conjI*, *rule impI*, *rule length-decreasing-leq-length*)
apply (*rule impI*, *erule-tac x=x* **in** *ballE*)
apply (*clarsimp*, *arith*)
by (*simp add*: *dom-def*)

lemma *rho-good-xs-ys* [*rule-format*]:
distinct (*xs* @ *ys*) ⟶ *rho-good* ([(*empty*, *0*, *0*)] + *zip* (*xs* @ *ys*) (*decreasing* (*length xs* + *length ys*)))
apply (*induct xs*)
apply (*simp*, *rule impI*, *rule rho-good-xs*)
apply (*clarsimp*, *subst decreasing.simps*, *simp*)
apply (*simp add*: *envAdd.simps*, *simp add*: *Let-def*, *simp add*: *rho-good.simps*, *simp add*: *rho-good′.simps*)
apply (*rule conjI*)
apply (*subst length-decreasing-add-eq-length-add*, *simp*)
apply (*rule ballI*, *rule conjI*)
apply (*rule impI*, *subst length-decreasing-add-eq-length-add*, *simp*)
apply (*rule impI*, *clarsimp*)
apply (*erule-tac x=x* **in** *ballE*)
apply (*elim conjE*,*simp*)
by (*simp add*: *dom-def*)

lemma *elements-domRho*:
⟦ *distinct* (*append xs rs*); *x* ∈ *domRho* ([(*empty*, *0*, *0*)] + *zip* (*xs* @ *rs*) (*decreasing* (*length xs* + *length rs*))) ⟧
⟹ (∃ *i*<*length xs*. *x* = *xs*!*i*) ∨ (∃ *j*<*length rs*. *x* = *rs*!*j*)
apply (*simp add*: *envAdd.simps*, *simp add*: *Let-def*)
apply (*subgoal-tac dom* (*map-of* (*map* (λ(*x*, *y*). (*x*, *y*)) (*zip* (*xs* @ *rs*) (*decreasing* (*length xs* + *length rs*))))) =
set (*append xs rs*),*simp*)
prefer *2* **apply** (*rule dom-map-zip*, *simp*, *simp*, *subst length-decreasing-add-eq-length-add*, *simp*)
by (*auto simp*:*set-conv-nth*)

lemma *addEnv-empty-xs-i* [*rule-format*]:
distinct (*xs* @ *rs*) ⟶ *i* < *length xs* ⟶ *xs*!*i* ∈ *domRho* ([(*empty*, *0*, *0*)] + *zip* (*xs* @ *rs*) (*decreasing* (*length xs* + *length rs*)))
⟶ *rho-good* ([(*empty*, *0*, *0*)] + *zip* (*xs* @ *rs*) (*decreasing* (*length xs* + *length rs*)))
⟶ ([(*empty*, *0*, *0*)] + *zip* (*xs* @ *rs*) (*decreasing* (*length xs* + *length rs*))) |> (*xs*!*i*) = *i*
apply (*induct xs i rule*: *list-induct3*)

apply *simp*

apply (*clarsimp*, *subst decreasing.simps*, *simp*)
apply (*simp add*: *envAdd.simps*, *simp add*: *Let-def*, *simp add*: *envSearch.simps*)

apply *clarsimp*

apply *clarsimp*
apply (*subgoal-tac xs* ! *i* ∈ *domRho* ([(*empty*, *0*, *0*)] + *zip* (*xs* @ *rs*) (*decreasing* (*length xs* + *length rs*))),*simp*)
apply (*subgoal-tac rho-good* ([(*empty*, *0*, *0*)] + *zip* (*xs* @ *rs*) (*decreasing* (*length xs* + *length rs*))),*simp*)
apply (*subst decreasing.simps*, *simp*, *simp add*: *envAdd.simps*, *simp add*: *Let-def*, *simp add*: *envSearch.simps*)
apply (*rule conjI*)
apply (*clarsimp*, *rule impI*)
apply (*case-tac* (*map-of* (*map* (λ(*x*, *y*). (*x*, *y*)) (*zip* (*xs* @ *rs*) (*decreasing* (*length xs* + *length rs*)))) (*xs* ! *i*)))
apply (*rule-tac s=None* **in** *ssubst*,*assumption*, *clarsimp*)
apply *clarsimp*
apply (*subst length-decreasing-add-eq-length-add*,*simp*)
apply (*subst length-decreasing-add-eq-length-add*,*simp*)
apply (*subgoal-tac min* (*length xs* + *length rs*) (*length* (*decreasing* (*length xs* + *length rs*))) = (*length xs* + *length rs*), *simp*)
apply (*simp add*: *rho-good.simps*, *simp add*: *rho-good′.simps*)
apply (*erule-tac x=*(*xs* ! (*length xs* + *length rs* − *a*)) **in** *ballE*)
apply (*elim conjE*, *erule-tac x=*(*xs* ! (*length xs* + *length rs* − *a*)) **in** *ballE*)
apply (*elim conjE*,*clarsimp*,*arith*)
apply (*simp add*: *dom-def*)
apply *clarsimp*
apply (*subst length-decreasing-add-eq-length-add*,*simp*)
apply (*rule rho-good-xs-ys*,*simp*)
by (*rule xs-i-in-domRho*,*assumption*+)

lemma *map-arguments-i* [*rule-format*]:
i < *length as* ⟶ (∀ *a* ∈ *set* (*map* (*item2Stack k S*) (*map* (*atom2item rho*) *as*)). ∃ *v*. *a* = *Val v*)
⟶ *append* (*append* (*map* (*item2Stack k S*) (*map* (*atom2item rho*) *as*)) (*map* (*item2Stack k S*) (*map* (*region2item rho*) *rr*)))
(*drop td S*) !+ *i* =
((*map* (*item2Stack k S*) (*map* (*atom2item rho*) *as*))!+ *i*)
by (*induct as i rule*: *list-induct3*, *clarsimp*+)

lemma *map-of-length* [*rule-format*]:
map-of (*map* (λ(*x*, *y*). (*x*, *y*)) (*zip ys* (*decreasing* (*length ys*)))) *x* = *Some y* ⟶
y ≤ *length ys* ∧ *y* ≤ *length* (*decreasing* (*length ys*)) ∧ *Suc 0* ≤ *y*
apply (*induct ys*,*simp*,*clarsimp*)
apply (*subgoal-tac* (*decreasing* (*Suc* (*length ys*))) = (*Suc* (*length ys*)) # (*decreasing* (*length ys*)),*simp*)
apply (*split split-if-asm*,*simp*)
apply (*subst length-decreasing-eq-length*,*simp*)
apply *simp*
by (*subst decreasing.simps*, *simp*)

lemma *rho-good-map-of*:
rho-good [(*map-of* (*map* (λ(*x*, *y*). (*x*, *y*)) (*zip* (*xs* @ *r* # *rs*) (*decreasing* (*Suc* (*length xs* + *length rs*))))),
min (*Suc* (*length xs* + *length rs*)) (*length* (*decreasing* (*Suc* (*length xs* + *length rs*)))), *0*)]
apply (*induct xs*)
apply (*simp add*: *rho-good.simps*, *simp add*: *rho-good′.simps*)
apply (*rule ballI*, *clarsimp*)
apply (*subgoal-tac decreasing* (*Suc* (*length rs*)) = *Suc* (*length rs*) # *decreasing* (*length rs*),*simp*)
apply (*split split-if-asm*)
apply (*simp*, *subst length-decreasing-eq-length*, *simp*)
apply (*frule map-of-length*,*simp*)
apply (*subst decreasing.simps*,*simp*)
apply *clarsimp*
apply (*subgoal-tac decreasing* (*Suc* (*length xs* + *length rs*)) =
Suc (*length xs* + *length rs*) # *decreasing* (*length xs* + *length rs*),*simp*)
apply (*subst decreasing.simps*,*simp*)
apply (*simp add*: *rho-good.simps*, *simp add*: *rho-good′.simps*)
apply (*rule conjI*)
apply (*subst decreasing.simps*, *simp*)
apply (*subst length-decreasing-add-eq-length-add*, *simp*)
apply *clarsimp*
apply (*rule conjI*)
apply (*rule impI*, *subst decreasing.simps*, *simp*)
apply (*subst length-decreasing-add-eq-length-add*, *simp*)
apply (*rule impI*)
apply (*erule-tac x*=*x* **in** *ballE*,*clarsimp*)
apply (*subgoal-tac min* (*length xs* + *length rs*) (*length* (*decreasing* (*length xs* + *length rs*))) =
(*length xs* + *length rs*),*simp*)
apply (*subst decreasing.simps*,*simp*)
apply (*subst length-decreasing-add-eq-length-add*,*simp*)
apply (*simp add*: *dom-def*)
by (*subst decreasing.simps*,*simp*)

lemma *addEnv-empty-xs-r* [*rule-format*]:
r ∉ *set xs* ⟶ *distinct xs* ⟶ *r* ∉ *set rs* ⟶ *distinct rs* ⟶ *set xs* ∩ *set rs* = {}
⟶ ([(*empty*, *0*, *0*)] + *zip* (*xs* @ *r* # *rs*) (*decreasing* (*Suc* (*length xs* + *length rs*)))) |> *r* = *length xs*
apply (*induct xs*)
apply (*clarsimp*, *subst decreasing.simps*, *simp*)
apply (*simp add*: *envAdd.simps*, *simp add*: *Let-def*, *simp add*: *envSearch.simps*)

apply (*clarsimp*, *subst decreasing.simps*, *simp*)
apply (*simp add*: *envAdd.simps*, *simp add*: *Let-def*, *simp add*: *envSearch.simps*)
apply (*case-tac* (*map-of* (*map* (λ(*x*, *y*). (*x*, *y*)) (*zip* (*xs* @ *r* # *rs*) (*decreasing* (*Suc* (*length xs* + *length rs*))))) *r*),*simp*)
apply (*subgoal-tac r* ∈ *dom* (*map-of* (*map* (λ(*x*, *y*). (*x*, *y*)) (*zip* (*xs* @ *r* # *rs*) (*decreasing* (*Suc* (*length xs* + *length rs*)))))))
prefer *2* **apply** (*subst dom-map-zip*, *simp*, *simp*, *subst decreasing.simps*, *simp*, *subst length-decreasing-add-eq-length-add*, *simp*, *simp*)
apply (*simp add*: *dom-def*)
apply *clarsimp*
apply (*subgoal-tac*
rho-good [(*map-of* (*map* (λ(*x*, *y*). (*x*, *y*)) (*zip* (*xs* @ *r* # *rs*) (*decreasing* (*Suc* (*length xs* + *length rs*))))),
min (*Suc* (*length xs* + *length rs*)) (*length* (*decreasing* (*Suc* (*length xs* + *length rs*)))), *0*)])
apply (*simp add*: *rho-good.simps*, *simp add*: *rho-good′.simps*)
apply (*erule-tac x*=*r* **in** *ballE*)
apply (*elim conjE*, *clarsimp*, *arith*)
apply (*simp add*: *dom-def*)
by (*rule rho-good-map-of*)

lemma *map-of-rs-j-aux* [*rule-format*]:
x ∉ *set rs* ⟶ *j* < *length rs* ⟶ *r* ∉ *set rs* ⟶ *r* ∉ *set xs* ⟶ *set xs* ∩ *set rs* = {}
⟶ *map-of* (*map* (λ(*x*, *y*). (*x*, *y*)) (*zip* (*x* # *xs* @ *r* # *rs*) (*decreasing* (*Suc* (*Suc* (*length xs* + *length rs*)))))) (*rs* ! *j*) = *Some y*
⟶ *map-of* (*map* (λ(*x*, *y*). (*x*, *y*)) (*zip* (*x* # *xs* @ *rs*) (*decreasing* (*Suc* (*length xs* + *length rs*))))) (*rs* ! *j*) = *Some y*
apply (*induct xs*)
apply (*clarsimp*, *subst decreasing.simps*, *simp*)
apply (*subgoal-tac* (*decreasing* (*Suc* (*Suc* (*length rs*)))) = (*Suc* (*Suc* (*length rs*))) # *Suc* (*length rs*) # (*decreasing* (*length rs*)))
apply (*clarsimp*, *rule conjI*)
apply (*rule impI*,*clarsimp*)
apply (*rule impI*, *clarsimp*)
apply (*subgoal-tac r* ≠ *rs* ! *j*,*simp*,*clarsimp*)
apply (*subst decreasing.simps*, *simp*, *subst decreasing.simps*, *simp*)
apply (*clarsimp*, *subst decreasing.simps*, *simp*)

apply (*subgoal-tac* (*decreasing* (*Suc* (*Suc* (*Suc* (*length xs* + *length rs*))))) =
(*Suc* (*Suc* (*Suc* (*length xs* + *length rs*)))) # (*decreasing* (*Suc* (*Suc*
(*length xs* + *length rs*)))),*simp*)
apply (*subgoal-tac* (*decreasing* (*Suc* (*Suc* (*length xs* + *length rs*)))) =
(*Suc* (*Suc* (*length xs* + *length rs*))) # (*decreasing* (*Suc* (*length xs*
+ *length rs*))),*simp*)
apply (*subgoal-tac* (*decreasing* (*Suc* (*length xs* + *length rs*))) =
(*Suc* (*length xs* + *length rs*)) # (*decreasing* (*length xs* + *length*
rs)),*simp*)
apply (*rule conjI*)
apply (*rule impI*, *clarsimp*)
apply (*rule impI*, *rule conjI*)
apply (*rule impI*, *clarsimp*)
apply *clarsimp*
by (*subst decreasing.simps*, *simp*)+

lemma *addEnv-empty-rs-r-j* [*rule-format*]:
distinct xs ⟶ *distinct rs* ⟶ *set xs* ∩ *set rs* = {} ⟶ *j* < *length rs*
⟶ *rs*!*j* ∈ *domRho* ([(*empty*, *0*, *0*)] + *zip* (*xs* @ *rs*) (*decreasing* (*length xs* +
length rs)))
⟶ ([(*empty*, *0*, *0*)] + *zip* (*xs* @ *rs*) (*decreasing* (*length xs* + *length rs*))) |>
(*rs*!*j*) = (*length xs* + *j*)
apply (*induct rs j rule*: *list-induct3*)

apply *simp*

apply *clarsimp*
apply (*rename-tac r rs*, *rule addEnv-empty-xs-r*, *assumption*+)

apply *clarsimp*

apply *clarsimp*
apply (*rename-tac r rs j*)
apply (*subgoal-tac rs* ! *j* ∈ *domRho* ([(*empty*, *0*, *0*)] + *zip* (*xs* @ *rs*) (*decreasing*
(*length xs* + *length rs*))),*simp*)
prefer *2* **apply** (*frule-tac xs*=*xs* **and** *rs*=*rs* **in** *rs-j-in-domRho*, *assumption*+)

apply (*case-tac xs*,*clarsimp*)
apply (*subst decreasing.simps*, *simp*)
apply (*subgoal-tac rho-good* ([(*empty*, *0*, *0*)] + *zip rs* (*decreasing* (*length rs*))))
prefer *2* **apply** (*rule rho-good-xs*)
apply (*simp add*: *envAdd.simps add*: *Let-def add*: *envSearch.simps*)
apply (*case-tac* (*map-of* (*map* (λ(*x*, *y*). (*x*, *y*)) (*zip rs* (*decreasing* (*length rs*))))
(*rs* ! *j*)),*clarsimp*)
apply (*clarsimp*, *rule conjI*)
apply (*clarsimp*)
apply (*rule impI*)

apply (*subgoal-tac* (*min* (*length rs*) (*length* (*decreasing* (*length rs*)))) = *length rs*,*simp*)
apply (*simp add*: *rho-good.simps*, *simp add*: *rho-good'.simps*)
apply (*erule-tac x*=*rs* ! (*length rs* − *a*) **in** *ballE*)
apply (*elim conjE*, *clarsimp*, *arith*)
apply (*clarsimp*)
apply (*subst length-decreasing-eq-length*, *simp*)
apply *clarsimp*
apply (*rename-tac r rs j x xs*)
apply (*simp add*: *envAdd.simps*, *simp add*: *Let-def*, *simp add*: *envSearch.simps*, *clarsimp*)
apply (*subgoal-tac* (*min* (*Suc* (*length xs* + *length rs*)) (*length* (*decreasing* (*Suc* (*length xs* + *length rs*))))) =
(*Suc* (*length xs* + *length rs*)),*simp*)
prefer *2* **apply** (*subst decreasing.simps*, *simp*)
apply (*subst length-decreasing-add-eq-length-add*, *simp*)
apply (*subgoal-tac min* (*Suc* (*Suc* (*length xs* + *length rs*))) (*length* (*decreasing* (*Suc* (*Suc* (*length xs* + *length rs*)))))) =
(*Suc* (*Suc* (*length xs* + *length rs*))),*simp*)
prefer *2* **apply** (*subst decreasing.simps*, *simp*)
apply (*frule-tac x*=*x* **and** *r*=*r* **and** *j*=*j* **and** *xs*=*xs* **and** *rs*=*rs* **in** *map-of-rs-j-aux*, *assumption*+)
by (*clarsimp*, *arith*)

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

lemma *in-set-conv-nth-1*:
x ∈ *set xs* ⟹ ∃ *i*<*length xs*. *x* = *xs*!*i*
by(*auto simp*:*set-conv-nth*)

lemma *conv-nth-in-set*:
i < *length xs* ⟹ *xs*!*i* ∈ *set xs*
by(*auto simp*:*set-conv-nth*)

lemma *xi1*:
⟦*i* < *length xs*; *length as* = *length xs*; *xi* = *xs* ! *i*; *distinct xs*; *as* ! *i* = *VarE x a*⟧
⟹ *Val* (*the* (*map-of* (*zip xs* (*map* (*atom2val E1*) *as*)) (*xs* ! *i*))) = *Val* (*the* (*E1* (*x*)))
apply (*induct xs as arbitrary*:*i rule*: *list-induct2'*,*simp*,*simp*,*simp*,*clarsimp*)
apply (*rule conjI*)
apply (*rule impI*,*clarsimp*)
apply (*case-tac i*,*simp*,*clarsimp*)
apply (*rule impI*)
by (*case-tac i*,*simp*,*clarsimp*)

lemma *ri1*:

⟦i < *length rs*; *length rr* = *length rs*; *ri* = *rs* ! i; *distinct rs* ⟧
⟹ *Reg* (*the* (*map-of* (*zip rs* (*map* (*the* ∘ *E2*) *rr*)) (*rs* ! i))) = *Reg* (*the* (*E2* (*rr* ! i)))
apply (*induct rr rs arbitrary*:i *rule*: *list-induct2'*,*simp*,*simp*,*simp*,*clarsimp*)
apply (*rule conjI*)
apply (*rule impI*,*clarsimp*)
apply (*case-tac i*,*simp*,*clarsimp*)
apply (*rule impI*)
by (*case-tac i*,*simp*,*clarsimp*)

lemma *app-var-arguments-good* [*rule-format*]:
i < *length as*
⟶ *as* ! i = *VarE x a*
⟶ *fvs'* *as* ⊆ *dom E1*
⟶ x ∈ *dom E1*
apply (*induct as arbitrary*:i,*simp*,*clarsimp*)
apply (*case-tac i*)
by (*simp add*:*dom-def*)+

lemma *identityEnvironment-App*:
⟦*length rho*> 0; *rho-good rho*; *distinct* (*append xs rs*); *length as* = *length xs*; *length rr* = *length rs*;
∀ a ∈ *set as*. *atom a*; *self* ∉ *set xs*; *self* ∉ *set rs*;
fvReg (*AppE f as rr a*) ⊆ *dom* (*snd* (*E1*, *E2*));
fv (*AppE f as rr a*) ⊆ *dom* (*fst* (*E1*, *E2*));
E1' = *map-of* (*zip xs* (*map* (*atom2val E1*) *as*));
E2' = *map-of* (*zip rs* (*map* (*the* ∘ *E2*) *rr*))(*self* ↦ *Suc k*);
n = *length xs*;
l = *length rs*;
(*E1*, *E2*) ⋈ (*rho* ,*S*)⟧
⟹ (*E1'*, *E2'*) ⋈ ([(*empty*, 0, 0)] +
zip (*xs* @ *rs*)
(*decreasing*
(*length xs* + *length rs*)) , *map* (*item2Stack k S*) (*map* (*atom2item rho*) *as*) @
map (*item2Stack k S*) (*map* (*region2item rho*) *rr*) @ *drop td S*)
apply (*subst identityEnvironment.simps*, *rule conjI*)
apply (*rule domRho-app*,*assumption*+)
apply (*rule conjI*)

apply *clarsimp*
apply (*simp add*: *identityEnvironment.simps*) **apply** (*elim conjE*)
apply (*frule in-set-conv-nth-1*)

apply (*erule exE*, *elim conjE*,*simp*)

apply (*erule-tac x=as!i* **in** *ballE*)
apply (*case-tac as!i*, *simp-all*)

apply (*rule xiConstE*)

apply (*rename-tac xi i x a*)
apply (*erule-tac x=x* **and** *A=dom E1* **in** *ballE*)
apply (*subgoal-tac Val* (*the* (*map-of* (*zip xs* (*map* (*atom2val E1*) *as*)) (*xs* ! *i*)))
=
Val (*the* (*E1* (*x*))), *simp*)
prefer *2* **apply** (*rule xi1*,*assumption+*)
apply (*subgoal-tac S* !+ (*rho* |> (*x*)) =
append (*append* (*map* (*item2Stack k S*) (*map* (*atom2item rho*)
as)) (*map* (*item2Stack k S*) (*map* (*region2item rho*) *rr*)))
(*drop td S*) !+ *i*,*simp*)
prefer *2* **apply** (*rule xi2*)
apply (*subgoal-tac* ([(*empty*, *0*, *0*)] + *zip* (*xs* @ *rs*) (*decreasing* (*length xs* + *length*
rs))) |> (*xs*!*i*) = *i*,*simp*)
apply (*rule xi3*)
apply (*subgoal-tac i* < *length as*)
apply (*frule app-var-arguments-good*,*assumption+*,*simp*,*simp*)

apply (*simp add*: *identityEnvironment.simps*) **apply** (*elim conjE*)
apply (*rule ballI*)
apply (*rule conjI*)
apply (*rule impI*) **apply** *simp*
apply (*rule impI*)
apply (*frule in-set-conv-nth-1*)

apply (*erule exE*, *elim conjE*) **apply** *simp*
apply (*erule-tac x=rr!i* **and** *A=dom E2* − {*self*} **in** *ballE*)
apply (*subgoal-tac Reg* (*the* (*map-of* (*zip rs* (*map* (*the* ∘ *E2*) *rr*)) (*rs* ! *i*))) =
Reg (*the* (*E2* (*rr* ! *i*))), *simp*)
prefer *2* **apply** (*rule ri1*,*assumption+*)
apply (*subgoal-tac S* !+ (*rho* |> (*rr* ! *i*)) =
append (*append* (*map* (*item2Stack k S*) (*map* (*atom2item rho*)
as)) (*map* (*item2Stack k S*) (*map* (*region2item rho*) *rr*)))
(*drop td S*) !+ (*length xs* + *i*),*simp*)
prefer *2* **apply** (*rule ri2*)
apply (*subgoal-tac* (*length xs* + *i*) = ([(*empty*, *0*, *0*)] + *zip* (*xs* @ *rs*) (*decreasing*
(*length xs* + *length rs*))) |> (*rs*!*i*),*simp*)
apply (*rule ri3*)
apply (*subgoal-tac* ∀ *r* ∈ *set rr*. *r* ∈ *dom E2* − {*self*},*clarsimp*)
apply (*rule good-app-region-arguments*)
done

end

16 Certification of the translation CoreSafe to SVM

theory *CertifSafeToSVM*
imports *Main ../CoreSafe/SafeRASemantics SVMSemantics CoreSafeToSVM CertifSafeToSVM-definitions execSVMBalanced-lemmas diff-lemmas maxFreshCells-lemmas maxFreshWords-lemmas identityEnvironment-lemmas basic-properties*
begin

16.1 Big-step semantics enriched with resource consumption

declare *envAdd.simps* [*simp del*]
declare *envSearch.simps* [*simp del*]
declare *decreasing.simps* [*simp del*]
declare *addEnv-good.simps* [*simp del*]
declare *rho-good.simps* [*simp del*]

lemma *closureCalled-e1*:
closureCalled e1 defs $\subseteq$ *closureCalled* (*Let x1* = *e1 In e2 a*) *defs*
apply (*rule subsetI*)
apply (*erule closureCalled.induct*)
apply (*rule closureCalled.init*,*simp*)
by (*simp*, *rule closureCalled.step*, *assumption*+,*simp*)

lemma *closureCalled-e2*:
closureCalled e2 defs $\subseteq$ *closureCalled* (*Let x1* = *e1 In e2 a*) *defs*
apply (*rule subsetI*)
apply (*erule closureCalled.induct*)
apply (*rule closureCalled.init*,*simp*)
by (*simp*, *rule closureCalled.step*, *assumption*+,*simp*)

lemma *closureCalled-case-aux* [*rule-format*]:
i < *length alts* $\longrightarrow$ *alts* ! *i* = (*p*, *ei*) $\longrightarrow$ *f* $\in$ *called ei* $\longrightarrow$ *f* $\in$ *calledList alts*
apply (*induct alts arbitrary*: *i*,*simp*)
apply (*rule impI*)+
apply *simp*
apply (*case-tac i*)
apply (*rule disjI1*,*simp*)
by (*rule disjI2*,*simp*)

lemma *closureCalled-case*:
⟦*i* < *length alts*;
alts ! *i* = (*ConstrP C ps ms*, *ei*) ⟧
$\Longrightarrow$ *closureCalled ei defs* $\subseteq$ *closureCalled* (*Case VarE x a Of alts a'*) *defs*

apply (*rule subsetI*)
apply (*erule closureCalled.induct*)
apply (*rule closureCalled.init*,*simp*)
apply (*rule closureCalled-case-aux*,*assumption*+)
by (*simp*, *rule closureCalled.step*, *assumption*+,*simp*)

lemma *closureCalled-case-1-x*:
⟦*i* < *length alts*;
alts ! *i* = (*ConstP p*, *ei*) ⟧
⟹ *closureCalled ei defs* ⊆ *closureCalled* (*Case VarE x a Of alts a'*) *defs*
apply (*rule subsetI*)
apply (*erule closureCalled.induct*)
apply (*rule closureCalled.init*,*simp*)
apply (*rule closureCalled-case-aux*,*assumption*+)
by (*simp*, *rule closureCalled.step*, *assumption*+,*simp*)

lemma *closureCalled-cased-aux* [*rule-format*]:
i < *length alts* ⟶ *alts* ! *i* = (*p*, *ei*) ⟶ *f* ∈ *called ei* ⟶ *f* ∈ *calledList' alts*
apply (*induct alts arbitrary*: *i*,*simp*)
apply (*rule impI*)+
apply *simp*
apply (*case-tac i*)
apply (*rule disjI1*,*simp*)
by (*rule disjI2*,*simp*)

lemma *closureCalled-cased*:
⟦*i* < *length alts*;
alts ! *i* = (*ConstrP C ps ms*, *ei*) ⟧
⟹ *closureCalled ei defs* ⊆ *closureCalled* (*CaseD VarE x a Of alts a'*) *defs*
apply (*rule subsetI*)
apply (*erule closureCalled.induct*)
apply (*rule closureCalled.init*,*simp*)
apply (*rule closureCalled-cased-aux*,*assumption*+)
by (*simp*, *rule closureCalled.step*, *assumption*+,*simp*)

lemma *disjointDomain*:
p ∉ *dom m* ⟹ ((*n* ++ *m*) *p*) = (*n p*)
apply (*unfold map-add-def*)
by (*case-tac* (*m p*), *simp*, *simp add*: *dom-def*)

lemma *extendsL-append*:
disjointList c c' ⟹ *extendsL* (*c* @ *c'*) *cs* = (*extendsL c cs* ∧ *extendsL c' cs*)
apply (*simp add*: *extendsL-def add*: *extends-def add*: *disjointList-def*, *auto*)
apply (*erule-tac x*=*x* **in** *ballE*, *simp*, *simp add*: *dom-def*)

apply (*erule-tac x=x* **in** *ballE*)
apply (*subgoal-tac x* $\notin$ *dom* (*map-of c*))
apply (*frule-tac n=*(*map-of c'*) **in** *disjointDomain*, *simp*)
apply (*blast*,*simp*)
apply (*erule conjE*, *simp add*: *dom-def*)
apply (*erule-tac x=x* **in** *ballE*, *simp*) **defer**
apply (*erule-tac x=x* **in** *ballE*)
apply (*erule-tac x=x* **in** *ballE*, *simp*, *simp add*: *dom-def*)
apply (*erule-tac x=x* **in** *ballE*, *simp*)
by (*frule-tac n=map-of c'* **in** *disjointDomain*, *simp*, *clarsimp*, *clarsimp*)

lemma *extendsL-e1*:
⟦*disjointList* (*cs2* @ [(*q2*, *is2*, *fname*)]) (*cs1'* @ [(*q1*, *is'*, *fname*)]);
(*cs2* @ (*q2*, *is2*, *fname*) # *cs1'* @ [(*q1*, *is'*, *fname*)]) ⊑ *cs*⟧
⟹ (*cs1'* @ [(*q1*, *is'*, *fname*)]) ⊑ *cs*
by (*frule-tac cs=cs* **in** *extendsL-append*, *simp*)

lemma *extendsL-e2*:
⟦ *disjointList* (*cs2* @ [(*q2*, *is2*, *fname*)]) (*cs1'* @ [(*q1*, *is'*, *fname*)]);
(*cs2* @ (*q2*, *is2*, *fname*) # *cs1'* @ [(*q1*, *is'*, *fname*)]) ⊑ *cs*⟧
⟹ (*cs2* @ [(*q2*, *is2*, *fname*)]) ⊑ *cs*
by (*frule-tac cs=cs* **in** *extendsL-append*, *simp*)

lemma *extendsL-case*:
⟦ *subList* (*csi* @ [(*qs* ! *i*, *isi*, *fname*)]) *cs1*; *q* $\notin$ *dom* (*map-of cs1*);
i < *length alts*; *qs*!*i* $\notin$ *dom* (*map-of cs1*);
(*cs1* @ [(*q*, *is'*, *fname*)]) ⊑ *cs*⟧
⟹ (*csi* @ [(*qs* ! *i*, *isi*, *fname*)]) ⊑ *cs*
apply (*simp add*: *subList.simps*, *elim exE*)
apply (*subgoal-tac disjointList cs1* [(*q*, *is'*, *fname*)],*simp*)
by (*simp add*: *disjointList-def*)

lemma *rho1*:
length rho > *0* ⟹ *topDepth* (*rho*++) = *0*
by (*induct rho*, *simp*, *simp add*: *envPlusPlus.simps*, *auto*)

lemma *rho2*:
0 < *length rho* ⟹ *0* < *length* (*rho* ++)
by (*induct rho*, *simp*, *simp add*: *envPlusPlus.simps*, *auto*)

lemma *rho3* [*rule-format*]:
0 < *length rho* ⟶ *td* = *topDepth rho* ⟶ *rho-good rho*
⟶ *td* + *1* = *topDepth* (*rho* + [(*x1*, *1*)])

apply (*induct rho*, *simp*, *clarsimp*)
by (*simp add*: *rho-good.simps add*: *envAdd.simps add*: *envSearch.simps*)

lemma *rho-case* [*rule-format*]:
0 < length rho ⟶ *td = topDepth rho* ⟶ *rho-good rho* ⟶ *nr = int* (*length xs*)
⟶ *nat* (*int td + nr*) = *topDepth* (*rho + zip xs* (*decreasing* (*length xs*)))
apply (*induct rho xs rule*: *list-induct2′*,*simp-all*)
apply *clarsimp*
apply (*simp add*: *rho-good.simps add*: *envAdd.simps add*: *envSearch.simps*)
apply *clarsimp*
apply (*simp add*: *rho-good.simps add*: *envAdd.simps add*: *envSearch.simps*)
apply (*simp add*: *Let-def*)
apply (*subgoal-tac length* (*decreasing* (*Suc* (*length ys*))) = *Suc* (*length ys*),*simp*)
apply (*induct-tac ys*, *simp*)
by (*subst decreasing.simps*,*simp*)+

lemma *length-decreasing-add-assoc*:
length (*decreasing* (*length xs + length rs*)) = *length xs + length rs*
apply (*induct xs rs rule*:*list-induct2′*, *simp-all*)
apply (*subst decreasing.simps*,*simp*)
apply (*subst decreasing.simps*, *simp*, *subst length-decreasing-eq-length*, *simp*)
apply (*subst decreasing.simps*, *simp*, *subst length-decreasing-eq-length*, *simp*)
by (*subst decreasing.simps*,*simp*)+

lemma *drop-general*:
drop n xs = y#ys ⟹ *drop* (*Suc n*) *xs = ys*
apply (*induct xs arbitrary*: *n*)
apply *simp*
apply (*simp add*:*drop-Cons*)
apply (*simp split*:*nat.splits*)
done

lemma *nth-drop′*:
i < length xs ⟹ *drop i xs = xs* ! *i* # *drop* (*Suc i*) *xs*
apply (*induct i arbitrary*: *xs*)
apply (*simp add*: *neq-Nil-conv*)
apply (*erule exE*)+
apply *simp*
apply (*case-tac xs*)
apply *simp-all*
done

lemma *equals-dom-map-upd-none*:

dom (λ*a*. *if a* = *x then None else m a*) = *dom* (*m*(*x* := *None*))
by (*auto*, *split split-if-asm*, *simp*, *simp*)

lemma *dom-map-insert*:
dom (λ*a*. *if a* = *x then Some z else m a*) = *insert x* (*dom m*)
by(*auto simp add*:*dom-def*)

declare *envPlusPlus.simps* [*simp del*]
declare *envAdd.simps* [*simp del*]
declare *restrictToRegion.simps* [*simp del*]
declare *sizeST.simps* [*simp del*]
declare *maxFreshCells.simps* [*simp del*]
declare *maxFreshWords.simps* [*simp del*]

lemma *correctness* [*rule-format*]:
(*E1*,*E2*) ⊢ *h* , *k* , *td* , *e* ⇓ *h'* , *k* , *v* , *r* ⟶
finite (*dom h'*) ⟶
(∀ *x*∈ *varProg e*. *x*≠ *self*) ⟶
fvReg e ⊆ *dom* (*snd* (*E1*,*E2*)) ⟶
fv e ⊆ *dom* (*fst* (*E1*,*E2*)) ⟶
self ∈ *dom* (*snd* (*E1*,*E2*)) ⟶
boundVar e ∩ *dom* (*snd* (*E1*,*E2*)) = {} ⟶
(*closureCalled e defs* ⊆ *definedFuns defs*
∧ ((*p*, *funm*, *contm*), *codes*) = *mapAccumL trF* (*1*, *empty*, []) *defs*
∧ *cs* = *concat codes*
∧ *P* = ((*cs*, *contm*),*p*,*ct*,*st*)
∧ *finite* (*dom h*)
⟶
(∀ *rho S S' k0 s0 p' q ls is is' cs1 j tds*.
(*q*, *ls*, *is*, *cs1*) = *trE p' funm fname rho e*
∧ (*append cs1* [(*q*,*is'*,*fname*)]) ⊑ *cs*
∧ *drop j is'* = *is*
∧ (*E1*,*E2*) ⋈ (*rho*,*S*)
∧ *td* = *topDepth rho*
∧ *k0* ≤ *k*
∧ *S'* = *drop td S*
∧ *s0* = ((*h*, *k*), *k0*, (*q*, *j*), *S*)
⟶
(∃ *q' i s ss δ m w*.
P⊢*s0* , *td*#*tds* −*svm*→ *s* # *ss*, *Suc 0*#*tds*
∧ *s* = ((*h'*, *k*) ↓ *k0*, *k0*, (*q'*, *i*), *Val v* # *S'*)
∧ *fst* (*the* (*map-of cs q'*))!*i* = *POPCONT*
∧ *r* = (*δ*,*m*,*w*)
∧ *δ* = *diff k* (*h*,*k*) (*h'*,*k*)
∧ *m* = *maxFreshCells* (*rev* (*s*#*ss*))
∧ *w* = *maxFreshWords* (*rev* (*s*#*ss*)))))

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

apply (*rule impI*)+
apply (*rule allI*)+
apply (*rule impI*)
apply (*elim conjE*)
apply (*rule-tac x=q* **in** *exI*)
apply (*rule-tac x=Suc (Suc (Suc j))* **in** *exI*)
apply (*rule-tac x=((h, k) ↓ k0, k0, (q, Suc (Suc (Suc j))), Val (IntT i) # S′)* **in** *exI*)
apply (*rule-tac x=[((h, k), k0, (q, j+2), Val (IntT i) # drop (topDepth rho) S),*
((h, k), k0, (q, j+1), Val (IntT i) # S),
((h, k), k0, (q, j), S)] **in** *exI*)
apply (*rule-tac x=[]k* **in** *exI*)
apply (*rule-tac x=0* **in** *exI*)
apply (*rule-tac x=1* **in** *exI*)
apply (*frule execSVMBalanced-IntT*) **apply** (*assumption+*) **apply** (*erule conjE*)
apply (*rule conjI, simp*)
apply (*rule conjI, simp*)
apply (*rule conjI, assumption*)
apply (*rule conjI, rule refl*)
apply (*rule conjI, rule diff-IntT*)
apply (*rule conjI, rule maxFreshCells-IntT, assumption+*)
apply (*frule lengthS-topDepth*)
apply (*rule maxFreshWords-IntT*) **apply** (*assumption+, simp*)

apply (*rule impI*)+
apply (*rule allI*)+
apply (*rule impI*)
apply (*elim conjE*)
apply (*rule-tac x=q* **in** *exI*)
apply (*rule-tac x=Suc (Suc (Suc j))* **in** *exI*)
apply (*rule-tac x=((h, k) ↓ k0, k0, (q, Suc (Suc (Suc j))), Val (BoolT b) # S′)* **in** *exI*)
apply (*rule-tac x=[((h, k), k0, (q, j+2), Val (BoolT b) # drop (topDepth rho) S),*
((h, k), k0, (q, j+1), Val (BoolT b) # S),
((h, k), k0, (q, j), S)] **in** *exI*)
apply (*rule-tac x=[]k* **in** *exI*)
apply (*rule-tac x=0* **in** *exI*)
apply (*rule-tac x=1* **in** *exI*)
apply (*frule execSVMBalanced-BoolT, assumption+, erule conjE*)
apply (*rule conjI, assumption*)
apply (*rule conjI, simp*)

apply (*rule conjI*, *assumption*)
apply (*rule conjI*, *rule refl*)
apply (*rule conjI*, *rule diff-BoolT*)
apply (*rule conjI*, *rule maxFreshCells-BoolT*, *assumption+*)
apply (*frule lengthS-topDepth*)
apply (*rule maxFreshWords-BoolT*,*assumption+*, *simp*)

apply (*rule impI*)+
apply (*rule allI*)+
apply (*rule impI*)
apply (*elim conjE*)
apply (*rule-tac x=q* **in** *exI*)
apply (*rule-tac x=Suc* (*Suc* (*Suc j*)) **in** *exI*)
apply (*rule-tac x=*((*h*, *k*) ↓ *k0*, *k0*, (*q*, *Suc* (*Suc* (*Suc j*))), *Val* (*Val.Loc pa*) # *S′*) **in** *exI*)
apply (*rule-tac x=*[((*h*, *k*), *k0*, (*q*, *j+2*), *Val* (*Val.Loc pa*) # *drop* (*topDepth rho*) *S*),
((*h*, *k*), *k0*, (*q*, *j+1*), *Val* (*Val.Loc pa*) # *S*),
((*h*, *k*), *k0*, (*q*, *j*), *S*)] **in** *exI*)
apply (*rule-tac x=*[]$_k$ **in** *exI*)
apply (*rule-tac x=0* **in** *exI*)
apply (*rule-tac x=1* **in** *exI*)
apply (*frule rho-good*, *elim conjE*)
apply (*frule lengthS-topDepth*)
apply (*frule execSVMBalanced-VarE*, *assumption+*, *erule conjE*)
apply (*rule conjI*, *assumption*)
apply (*rule conjI*, *simp*)
apply (*rule conjI*, *assumption*)
apply (*rule conjI*, *rule refl*)
apply (*rule conjI*, *rule diff-VarE*)
apply (*rule conjI*, *rule maxFreshCells-VarE*, *assumption+*)
apply (*frule lengthS-topDepth*)
apply (*rule maxFreshWords-VarE*,*assumption+*, *simp*)

apply (*rule impI*)+
apply (*rule allI*)+
apply (*rule impI*)
apply (*elim conjE*)
apply (*frule A7*)
apply (*elim conjE*)
apply (*rule-tac x=q* **in** *exI*)
apply (*rule-tac x=*(*Suc* (*Suc* (*Suc* (*Suc ja*)))) **in** *exI*)
apply (*rule-tac x=*((*h′*, *k*) ↓ *k0*, *k0*, (*q*, (*Suc* (*Suc* (*Suc* (*Suc ja*))))), *Val* (*Val.Loc p′*) # *S′*) **in** *exI*)
apply (*rule-tac x=*[((*h′*, *k*), *k0*, (*q*, *Suc* (*Suc* (*Suc ja*))), *Val* (*Loc p′*) # *drop*

(*topDepth rho*) *S*),
((*h'*, *k*), *k0*, (*q*, *Suc* (*Suc ja*)), *Val* (*Loc p'*) # *S*),
((*h*, *k*), *k0*, (*q*, *Suc ja*), *Val* (*Loc pa*) # *Reg j* # *S*),
((*h*, *k*), *k0*, (*q*, *ja*), *S*)] **in** *exI*)
apply (*rule-tac x*=[*j* ↦ *m*] **in** *exI*)
apply (*rule-tac x*=*m* **in** *exI*)
apply (*rule-tac x*=*2* **in** *exI*)
apply (*frule rho-good*, *elim conjE*)
apply (*frule lengthS-topDepth*)
apply (*frule-tac k0'*=*k0* **in** *execSVMBalanced-CopyE*, *assumption*+, *erule conjE*)
apply (*rule conjI*, *assumption*)
apply (*rule conjI*, *simp*)
apply (*rule conjI*, *assumption*)
apply (*rule conjI*, *rule refl*)
apply (*rule conjI*, *rule diff-CopyE*, *assumption*+)
apply (*rule conjI*, *rule maxFreshCells-CopyE*, *assumption*+)
apply (*frule lengthS-topDepth*)
apply (*rule maxFreshWords-CopyE*,*assumption*+, *simp*)

apply (*rule impI*)+
apply (*rule allI*)+
apply (*rule impI*)
apply (*elim conjE*)
apply (*rule-tac x*=*qa* **in** *exI*)
apply (*rule-tac x*=(*Suc* (*Suc* (*Suc* (*Suc j*)))) **in** *exI*)
apply (*rule-tac x*=(((*h*(*pa*:=*None*))(*q*↦*c*), *k*) ↓ *k0*, *k0*, (*qa*, *Suc* (*Suc* (*Suc* (*Suc* *j*)))), *Val* (*Loc q*) # *S'*) **in** *exI*)
apply (*rule-tac x*=[(((*h*(*pa*:=*None*))(*q*↦*c*), *k*), *k0*, (*qa*, *Suc* (*Suc* (*Suc j*))), *Val* (*Loc q*) # *drop* (*topDepth rho*) *S*),
(((*h*(*pa*:=*None*))(*q*↦*c*), *k*), *k0*, (*qa*, *Suc* (*Suc j*)), *Val* (*Loc q*) # *S*),
((*h*, *k*), *k0*, (*qa*, *Suc j*), *Val* (*Loc pa*) # *S*),
((*h*, *k*), *k0*, (*qa*, *j*), *S*)] **in** *exI*)
apply (*rule-tac x*=[]$_k$ **in** *exI*)
apply (*rule-tac x*=*0* **in** *exI*)
apply (*rule-tac x*=*1* **in** *exI*)
apply (*frule rho-good*, *elim conjE*)
apply (*frule lengthS-topDepth*)
apply (*frule execSVMBalanced-ReuseE*, *assumption*+, *erule conjE*)
apply (*rule conjI*, *assumption*)
apply (*rule conjI*, *rule refl*)
apply (*rule conjI*, *assumption*)
apply (*rule conjI*, *rule refl*)
apply (*rule conjI*, *rule diff-ReuseE*, *assumption*+)
apply (*rule conjI*, *rule maxFreshCells-ReuseE*, *assumption*+)
apply (*frule lengthS-topDepth*)
apply (*rule maxFreshWords-ReuseE*,*simp*,*assumption*)

apply (*rule impI*)+ **apply**(*elim conjE*)
apply (*rule allI*)+ **apply** (*rule impI*, *elim conjE*)
apply (*frule trE-let*, *simp*, *elim exE*)

apply (*erule-tac x=C* **in** *allE*)
apply (*erule-tac x= h′ |‘ {p. p ∈ dom h′ & fst (the (h′ p)) ≤ k}* **in** *allE*)
apply (*erule-tac x=as* **in** *allE*)
apply (*erule-tac x=v1* **in** *allE*)
apply (*erule-tac x=r1* **in** *allE*)
apply (*erule-tac x=a′* **in** *allE*)
apply (*elim conjE*)

apply (*frule-tac e=e1* **in** *finite-dom-h*,*elim conjE*)
apply (*frule-tac e=e2* **in** *finite-dom-h*,*elim conjE*)
apply (*frule finite-dom-h′*) **apply** *assumption*
apply (*drule mp*,*assumption*)+
apply (*drule mp*, *rule x-not-self*, *rule all-exp-core*)+
apply (*drule mp*,*simp*)+
apply (*subst dom-map-insert*, *assumption*)
apply (*drule mp*,*simp*)+
apply (*subgoal-tac closureCalled e1 defs ⊆ closureCalled* (*Let x1 = e1 In e2 a*) *defs*)
prefer *2* **apply** (*rule closureCalled-e1*)
apply (*erule subset-trans*, *assumption*)
apply (*drule mp*, *simp*)
apply (*subgoal-tac closureCalled e2 defs ⊆ closureCalled* (*Let x1 = e1 In e2 a*) *defs*)
prefer *2* **apply** (*rule closureCalled-e2*)
apply (*erule subset-trans*, *assumption*)

apply (*erule-tac x=*(*rho++*) **in** *allE*)
apply (*erule-tac x=*(*rho+*[(*x1*,*1*)]) **in** *allE*)

apply (*erule-tac x= Cont* (*k0*,*q2*) # *S* **in** *allE*)
apply (*erule-tac x=*(*Val v1*#*S*) **in** *allE*)

apply (*erule-tac x=drop 0* (*Cont* (*k0*,*q2*) # *S*) **in** *allE*)
apply (*erule-tac x=S′* **in** *allE*)

apply (*erule-tac x=k* **in** *allE*)
apply (*erule-tac x=k0* **in** *allE*)

apply (*erule-tac x=*((*h*, *k*), *k*, (*q1*, *j+1*), *Cont* (*k0*,*q2*) # *S*) **in** *allE*)
apply (*erule-tac x=*((*h′ |‘ {p. p ∈ dom h′ & fst* (*the* (*h′ p*)) ≤ *k*}, *k*), *k0*, (*q2*,

0), *Val v1 # S*) **in** *allE*)

apply (*erule-tac x*=(*q2*+*1*) **in** *allE*)
apply (*erule-tac x*=*p'* **in** *allE*)

apply (*erule-tac x*=*q1* **in** *allE*)
apply (*erule-tac x*=*q2* **in** *allE*)

apply (*erule-tac x*=*ls1* **in** *allE*)
apply (*erule-tac x*=*ls2* **in** *allE*)

apply (*erule-tac x*=*is1* **in** *allE*)
apply (*erule-tac x*=*is2* **in** *allE*)

apply (*erule-tac x*=*is'* **in** *allE*)
apply (*erule-tac x*=*is2* **in** *allE*)

apply (*erule-tac x*=*cs1'* **in** *allE*)
apply (*erule-tac x*=*cs2* **in** *allE*)

apply (*erule-tac x*=*j*+*1* **in** *allE*)
apply (*erule-tac x*=*0* **in** *allE*)

apply (*erule-tac x*=*td*#*tds* **in** *allE*)
apply (*erule-tac x*=*tds* **in** *allE*)

apply (*drule mp*)
apply (*frule rho-good*, *elim conjE*)
apply (*rule conjI*) **apply** *assumption*
apply (*rule conjI*) **apply** (*frule-tac ?cs2.0*=*cs2* **and** *cs1'*=*cs1'* **in** *codestore-let-disjoint*, *assumption*+)
apply (*simp*, *elim conjE*, *erule extendsL-e1*, *assumption*+)
apply (*rule conjI*) **apply** (*simp*, *erule drop-general*)
apply (*rule conjI*) **apply** (*erule identityEnvironment-e1*, *assumption*+)
apply (*rule conjI*) **apply** (*rule sym*, *erule rho1*)
apply (*rule conjI*) **apply** *simp*
apply (*rule conjI*) **apply** *simp*
apply *simp*

apply (*drule mp*)
apply (*frule rho-good*, *elim conjE*)
apply (*rule conjI*) **apply** *assumption*
apply (*rule conjI*) **apply** (*frule-tac ?cs2.0*=*cs2* **and** *cs1'*=*cs1'* **in** *codestore-let-disjoint*, *assumption*+)
apply (*simp*, *elim conjE*, *erule extendsL-e2*, *assumption*+)
apply (*rule conjI*) **apply** *simp*
apply (*rule conjI*) **apply** (*erule identityEnvironment-e2*,*assumption*+,*clarsimp*,*assumption*+,*simp*)
apply (*rule conjI*) **apply** (*erule rho3*,*assumption*+)

apply (*rule conjI*) **apply** *assumption*
apply (*rule conjI*) **apply** *simp*
apply (*rule refl*)

apply (*elim exE*)
apply (*elim conjE*)

apply (*rule-tac x=q′a* **in** *exI*)
apply (*rule-tac x=ia* **in** *exI*)

apply (*rule-tac x=sa* **in** *exI*)

apply (*rule-tac x= ssa* @
((*h′* |‘ {*p* ∈ *dom h′*. *fst* (*the* (*h′ p*)) ≤ *k*}, *k*) ↓ *k*, *k*, (*q′*, *i*), *Val v1* # *drop 0* (*Cont* (*k0*,*q2*) # *S*)) # *ss* @
[((*h*,*k*),*k0*,(*q*,*j*), *S*)] **in** *exI*)
apply (*rule-tac x=δ1* ⊕ *δ2* **in** *exI*)
apply (*rule-tac x= max m1* (*m2* + ‖ *δ1* ‖) **in** *exI*)
apply (*rule-tac x= max* (*s1* + *2*) (*s2* + *1*) **in** *exI*)
apply (*rule conjI*, *frule execSVMBalanced-Let1*, *assumption+*, *simp*)
apply (*rule conjI*, *assumption*)
apply (*rule conjI*, *assumption*)
apply (*rule conjI*, *rule refl*)
apply (*rule conjI*, *rule diff-Let1*, *assumption+*)
apply (*frule-tac v=v1* **in** *resources-gre-0*) **apply** (*elim exE*, *elim conjE*)
apply (*frule-tac v=v2* **in** *resources-gre-0*, *elim exE*, *elim conjE*)
apply (*rule conjI*, *rule maxFreshCells-Let1*, *assumption+*, *simp*, *simp*, *simp*)
apply (*rule maxFreshWords-Let1*,*assumption+*, *simp*)

apply (*rule impI*)+ **apply** (*elim conjE*)
apply (*rule allI*)+ **apply** (*rule impI*, *elim conjE*)
apply (*frule trE-let2*, *elim exE*)

apply (*frule-tac e=e2* **in** *finite-dom-h*,*elim conjE*)
apply (*drule mp*, *rule finite-dom-h′*, *assumption*, *simp*)
apply (*drule mp*, *rule x-not-self*, *rule all-exp-core*)
apply (*drule mp*, *simp*)+
apply (*subst dom-map-insert*, *assumption*)
apply (*drule mp*, *simp*)
apply (*drule mp*, *simp*)
apply (*drule mp*)

apply (*rule conjI*)
apply (*subgoal-tac closureCalled e2 defs* $\subseteq$ *closureCalled* (*Let x1 = ConstrE C as r a′ In e2 a*) *defs*)
prefer *2* **apply** (*rule closureCalled-e2*)
apply (*erule subset-trans*, *assumption*)
apply (*frule finite-dom-h*) **apply** (*erule conjI*)+
apply (*elim conjE*,*assumption*)

apply (*erule-tac x*=(*rho*+[(*x1*,*1*)]) **in** *allE*)
apply (*erule-tac x*=(*Val* (*Loc pa*) #*S*) **in** *allE*)
apply (*erule-tac x*=*S′* **in** *allE*)
apply (*erule-tac x*=*k0* **in** *allE*)
apply (*erule-tac x*=((*h*(*pa* $\mapsto$ (*j*, *C*, *map* (*atom2val E1*) *as*)), *k*), *k0*, (*q*, *Suc ja*), *Val* (*Loc pa*) # *S*) **in** *allE*)

apply (*erule-tac x*=*p′* **in** *allE*)

apply (*erule-tac x*=*q* **in** *allE*)
apply (*erule-tac x*=*ls2* **in** *allE*)
apply (*erule-tac x*=*is2* **in** *allE*)
apply (*erule-tac x*=*is′* **in** *allE*)
apply (*erule-tac x*=*cs2* **in** *allE*)
apply (*erule-tac x*=*Suc ja* **in** *allE*)
apply (*erule-tac x*=*tds* **in** *allE*)

apply (*drule mp*)
apply (*frule rho-good*, *elim conjE*)
apply (*rule conjI*) **apply** *simp*
apply (*rule conjI*) **apply** *simp*
apply (*rule conjI*) **apply** *simp*
apply (*rule drop-general*,*assumption*)
apply (*rule conjI*) **apply** (*erule identityEnvironment-e2-let2*,*assumption*+,*simp*,*assumption*+,*simp*)
apply (*rule conjI*) **apply** (*erule rho3*,*assumption*+)
apply (*rule conjI*) **apply** *assumption*
apply (*rule conjI*) **apply** *simp*
apply (*rule refl*)
apply (*elim exE*)
apply (*elim conjE*)

apply (*rule-tac x*=*q′* **in** *exI*)
apply (*rule-tac x*=*i* **in** *exI*)

apply (*rule-tac x*=*sa* **in** *exI*)
apply (*rule-tac x*= *ss* @ [((*h*, *k*), *k0*, (*q*, *ja*), *S*)] **in** *exI*)
apply (*rule-tac x*=$\delta \oplus [j \mapsto 1]$ **in** *exI*)
apply (*rule-tac x*= *m* + *1* **in** *exI*)
apply (*rule-tac x*=*s*+*1* **in** *exI*)

apply (*frule rho-good*, *elim conjE*)
apply (*frule finite-dom-h*) **apply** (*elim conjE*)
apply (*subgoal-tac core* (*Let x1* = *ConstrE C as r a′ In e2 a*))
prefer *2* **apply** (*rule all-exp-core*)
apply (*frule let-atoms*,*elim conjE*)
apply (*rule conjI*, *rule execSVMBalanced-Let2*,*assumption*+)
apply (*rule conjI*, *assumption*)
apply (*rule conjI*, *assumption*)
apply (*rule conjI*, *rule refl*)
apply (*rule conjI*, *rule diff-Let2*,*assumption*+)
apply (*frule-tac v=v2* **in** *resources-gre-0*, *elim exE*, *elim conjE*)
apply (*rule conjI*, *rule maxFreshCells-Let2*, *assumption*+, *simp*, *simp*)
apply (*rule maxFreshWords-Let2*,*assumption*+,*simp*)

apply (*rule impI*)+
apply (*elim conjE*)

apply (*rule allI*)+
apply (*rule impI*, *elim conjE*)

apply (*erule-tac x=pa* **in** *allE*)
apply (*erule-tac x=extend E1 xs vs* **in** *allE*)
apply (*erule-tac x=xs* **in** *allE*)
apply (*erule-tac x=vs* **in** *allE*)
apply (*erule-tac x=j* **in** *allE*)
apply (*erule-tac x=C* **in** *allE*)
apply (*elim conjE*)

apply (*frule trE-case*, *assumption*+)
apply (*elim exE*, *elim conjE*)

apply (*frule finite-dom-h*,*elim conjE*)
apply (*drule mp*, *simp*)
apply (*drule mp*, *simp*) **apply** (*rule x-not-self*, *rule all-exp-core*)
apply (*drule mp*, *simp*)+
apply (*subgoal-tac closureCalled ei defs* ⊆ *closureCalled* (*Case VarE x a Of alts a′*) *defs*)
prefer *2* **apply** (*rule closureCalled-case*,*assumption*+)
apply (*erule subset-trans*, *assumption*)

apply (*frule case-constructor-good*,*assumption*+)

apply (*erule-tac x=rho* + *zip xs* (*decreasing* (*length xs*)) **in** *allE*)
apply (*erule-tac x=map Val vs* @ *S* **in** *allE*)
apply (*erule-tac x=drop* (*nat* (*int td* + *nr*)) (*map Val vs* @ *S*) **in** *allE*)

apply (*erule-tac x=k0* **in** *allE*)
apply (*erule-tac x=((h, k), k0, (qs!i, 0), (map Val vs) @ S)* **in** *allE*)

apply (*erule-tac x=p'* **in** *allE*)
apply (*erule-tac x=qs!i* **in** *allE*)
apply (*erule-tac x=lss!i* **in** *allE*)
apply (*erule-tac x=isi* **in** *allE*)
apply (*erule-tac x=isi* **in** *allE*)
apply (*erule-tac x=csi* **in** *allE*)
apply (*erule-tac x=0* **in** *allE*)
apply (*erule-tac x=tds* **in** *allE*)

apply (*drule mp*)
apply (*frule rho-good*, *elim conjE*)
apply (*frule codestore-case-alts-disjoint*, *rule disjI1*, *rule refl*, *assumption+*, *elim conjE*)
apply (*rule conjI*) **apply** *simp*
apply (*rule conjI*) **apply** (*rule extendsL-case*, *assumption+*,*simp*,*assumption+*,*simp*,*assumption*)
apply (*rule conjI*) **apply** *simp*
apply (*rule conjI*) **apply** (*rule identityEnvironment-case*,*simp*,*assumption+*)
apply (*rule addEnv-rho-good*, *assumption+*,*simp*)
apply (*rule conjI*) **apply** (*rule rho-case*, *assumption+*, *simp*, *unfold def-extend-def*, *elim conjE*, *simp*)
apply (*rule conjI*) **apply** *assumption*
apply (*rule conjI*) **apply** (*rule refl*)
apply (*rule refl*)

apply (*elim exE*)
apply (*elim conjE*)

apply (*rule-tac x=q'* **in** *exI*)
apply (*rule-tac x=ia* **in** *exI*)

apply (*rule-tac x=sa* **in** *exI*)
apply (*rule-tac x= ss @ [((h, k), k0, (q, j), S)]* **in** *exI*)
apply (*rule-tac x=δ* **in** *exI*)
apply (*rule-tac x=m* **in** *exI*)
apply (*rule-tac x=s+nr* **in** *exI*)

apply (*rule conjI*) **apply** (*frule rho-good*, *elim conjE*)
apply (*rule execSVMBalanced-Case* ,*assumption+*)
apply (*erule-tac V=(qs ! i, lss ! i, isi, csi) = trE p' funm fname (rho + zip xs (decreasing (length xs))) ei* **in** *thin-rl*)
apply (*rule conjI*) **apply** (*subgoal-tac (nat (int td + int (length vs)) − length vs) =*
(*nat (int td + int (length vs) − int (length vs)*)),*simp*,*simp*)

apply (*rule conjI*, *assumption*)
apply (*rule conjI*, *rule refl*)
apply (*rule conjI*, *simp*)
apply (*frule-tac v=v* **in** *resources-gre-0*, *elim exE*, *elim conjE*)
apply (*rule conjI*) **apply** (*rule maxFreshCells-Case*, *assumption+*, *simp*)
apply (*rule maxFreshWords-Case*,*assumption+*,*simp*,*assumption*)

apply (*rule impI*)+
apply (*elim conjE*)

apply (*erule-tac x=n* **in** *allE*)
apply (*elim conjE*)

apply (*rule allI*)+
apply (*rule impI*, *elim conjE*)

apply (*frule trE-case-1-1*, *assumption+*)
apply (*elim exE*, *elim conjE*)
apply (*frule finite-dom-h*,*elim conjE*)
apply (*drule mp*, *simp*)
apply (*drule mp*, *simp*, *rule x-not-self*, *rule all-exp-core*)
apply (*drule mp*, *simp*)+
apply (*subgoal-tac closureCalled ei defs* $\subseteq$ *closureCalled* (*Case VarE x a Of alts a'*) *defs*)
prefer *2* **apply** (*rule closureCalled-case-1-x*,*assumption+*)
apply (*erule subset-trans*, *assumption*)

apply (*erule-tac x=rho* **in** *allE*)
apply (*erule-tac x=S* **in** *allE*)
apply (*erule-tac x=drop td S* **in** *allE*)
apply (*erule-tac x=k0* **in** *allE*)
apply (*erule-tac x=*((*h*, *k*), *k0*, (*qs*!*i*, *0*), *S*) **in** *allE*)

apply (*erule-tac x=p'* **in** *allE*)
apply (*erule-tac x=qs*!*i* **in** *allE*)
apply (*erule-tac x=lss*!*i* **in** *allE*)
apply (*erule-tac x=isi* **in** *allE*)
apply (*erule-tac x=isi* **in** *allE*)
apply (*erule-tac x=csi* **in** *allE*)
apply (*erule-tac x=0* **in** *allE*)
apply (*erule-tac x=tds* **in** *allE*)

apply (*drule mp*)
apply (*frule codestore-caseL-alts-disjoint*, *assumption+*, *elim conjE*)
apply (*rule conjI*) **apply** *assumption*

apply (*rule conjI*) **apply** (*rule extendsL-case*, *assumption*+,*simp*,*assumption*+,*simp*,*assumption*)

apply (*rule conjI*) **apply** *simp*
apply (*rule conjI*) **apply** *assumption*
apply (*rule conjI*) **apply** *assumption*
apply (*rule conjI*) **apply** *assumption*
apply (*rule conjI*) **apply** (*rule refl*)
apply (*rule refl*)
apply (*elim exE*)
apply (*elim conjE*)

apply (*rule-tac* $x{=}q'$ **in** *exI*)
apply (*rule-tac* $x{=}ia$ **in** *exI*)

apply (*rule-tac* $x{=}sa$ **in** *exI*)
apply (*rule-tac* $x{=}$ *ss* @ [((*h*, *k*), *k0*, (*q*, *j*), *S*)] **in** *exI*)
apply (*rule-tac* $x{=}\delta$ **in** *exI*)
apply (*rule-tac* $x{=}m$ **in** *exI*)
apply (*rule-tac* $x{=}s$ **in** *exI*)

apply (*rule conjI*) **apply** (*frule rho-good*, *elim conjE*)
apply (*rule execSVMBalanced-Case-1-1*, *assumption*+)
apply (*rule conjI*) **apply** *simp*
apply (*rule conjI*, *assumption*)
apply (*rule conjI*, *rule refl*)
apply (*rule conjI*, *simp*)
apply (*frule-tac* $v{=}v$ **in** *resources-gre-0*, *elim exE*, *elim conjE*)
apply (*rule conjI*) **apply** (*rule maxFreshCells-Case-1-x*, *assumption*+,*simp*)
apply (*rule maxFreshWords-Case-1-x*, *assumption*+,*simp*,*assumption*)

apply (*rule impI*)+
apply (*elim conjE*)

apply (*erule-tac* $x{=}b$ **in** *allE*)
apply (*elim conjE*)

apply (*rule allI*)+
apply (*rule impI*, *elim conjE*)

apply (*frule trE-case-1-2*, *assumption*+)
apply (*elim exE*, *elim conjE*)

apply (*frule finite-dom-h*,*elim conjE*)
apply (*drule mp*, *simp*)
apply (*drule mp*, *simp*, *rule x-not-self*, *rule all-exp-core*)
apply (*drule mp*, *simp*)+

apply (*subgoal-tac closureCalled ei defs* $\subseteq$ *closureCalled* (*Case VarE x a Of alts a'*) *defs*)
prefer *2* **apply** (*rule closureCalled-case-1-x*,*assumption+*)
apply (*erule subset-trans*, *assumption*)

apply (*erule-tac x=rho* **in** *allE*)
apply (*erule-tac x=S* **in** *allE*)
apply (*erule-tac x=drop td S* **in** *allE*)
apply (*erule-tac x=k0* **in** *allE*)
apply (*erule-tac x=*((*h*, *k*), *k0*, (*qs*!*i*, *0*), *S*) **in** *allE*)

apply (*erule-tac x=p'* **in** *allE*)
apply (*erule-tac x=qs*!*i* **in** *allE*)
apply (*erule-tac x=lss*!*i* **in** *allE*)
apply (*erule-tac x=isi* **in** *allE*)
apply (*erule-tac x=isi* **in** *allE*)
apply (*erule-tac x=csi* **in** *allE*)
apply (*erule-tac x=0* **in** *allE*)
apply (*erule-tac x=tds* **in** *allE*)

apply (*drule mp*)
apply (*frule codestore-caseL-alts-disjoint*, *assumption+*, *elim conjE*)
apply (*rule conjI*) **apply** *assumption*
apply (*rule conjI*) **apply** (*rule extendsL-case*, *assumption+*,*simp*,*assumption+*,*simp*,*assumption*)

apply (*rule conjI*) **apply** *simp*
apply (*rule conjI*) **apply** *assumption*
apply (*rule conjI*) **apply** *assumption*
apply (*rule conjI*) **apply** *assumption*
apply (*rule conjI*) **apply** (*rule refl*)
apply (*rule refl*)
apply (*elim exE*)
apply (*elim conjE*)

apply (*rule-tac x=q'* **in** *exI*)
apply (*rule-tac x=ia* **in** *exI*)

apply (*rule-tac x=sa* **in** *exI*)
apply (*rule-tac x= ss* @ [((*h*, *k*), *k0*, (*q*, *j*), *S*)] **in** *exI*)
apply (*rule-tac x=*δ **in** *exI*)
apply (*rule-tac x=m* **in** *exI*)
apply (*rule-tac x=s* **in** *exI*)

apply (*rule conjI*) **apply** (*frule rho-good*, *elim conjE*)
apply (*rule execSVMBalanced-Case-1-2*, *assumption+*)
apply (*rule conjI*) **apply** *simp*
apply (*rule conjI*, *assumption*)
apply (*rule conjI*, *rule refl*)
apply (*rule conjI*, *simp*)

apply (*frule-tac v=v* **in** *resources-gre-0*, *elim exE*, *elim conjE*)
apply (*rule conjI*) **apply** (*rule maxFreshCells-Case-1-x*, *assumption+*,*simp*)
apply (*rule maxFreshWords-Case-1-x*, *assumption+*,*simp*,*assumption*)

apply (*rule impI*)+
apply (*elim conjE*)

apply (*rule allI*)+
apply (*rule impI*, *elim conjE*)

apply (*erule-tac x=pa* **in** *allE*)
apply (*erule-tac x=extend E1 xs vs* **in** *allE*)
apply (*erule-tac x=xs* **in** *allE*)
apply (*erule-tac x=vs* **in** *allE*)
apply (*erule-tac x=j* **in** *allE*)
apply (*erule-tac x=C* **in** *allE*)
apply (*elim conjE*)

apply (*frule trE-cased*, *assumption+*)
apply (*elim exE*, *elim conjE*)

apply (*frule finite-dom-h*,*elim conjE*)
apply (*drule mp*, *simp*)
apply (*drule mp*, *simp*, *rule x-not-self*, *rule all-exp-core*)
apply (*drule mp*, *simp*)+
apply (*subgoal-tac finite* (*dom* (*h*(*pa* := *None*)))) **prefer** *2* **apply** *simp*
apply (*subgoal-tac closureCalled ei defs* ⊆ *closureCalled* (*CaseD VarE x a Of alts a'*) *defs*)
prefer *2* **apply** (*rule closureCalled-cased*,*assumption+*)
apply (*rule conjI*) **apply** (*erule subset-trans*, *assumption*)
apply (*subst equals-dom-map-upd-none*, *assumption*)

apply (*frule cased-constructor-good*,*assumption+*)

apply (*erule-tac x=rho + zip xs* (*decreasing* (*length xs*)) **in** *allE*)
apply (*erule-tac x=map Val vs* @ *S* **in** *allE*)
apply (*erule-tac x=drop* (*nat* (*int td + nr*)) (*map Val vs* @ *S*) **in** *allE*)
apply (*erule-tac x=k0* **in** *allE*)
apply (*erule-tac x=*((*h*(*pa* := *None*), *k*), *k0*, (*qs*!*i*, *0*), (*map Val vs*) @ *S*) **in** *allE*)

apply (*erule-tac x=p'* **in** *allE*)
apply (*erule-tac x=qs*!*i* **in** *allE*)
apply (*erule-tac x=lss*!*i* **in** *allE*)
apply (*erule-tac x=isi* **in** *allE*)
apply (*erule-tac x=isi* **in** *allE*)

apply (*erule-tac x=csi* **in** *allE*)
apply (*erule-tac x=0* **in** *allE*)
apply (*erule-tac x=tds* **in** *allE*)

apply (*drule mp*)
apply (*frule rho-good*, *elim conjE*)
apply (*frule codestore-case-alts-disjoint*, *rule disjI2*, *rule refl*, *assumption+*, *elim conjE*)
apply (*rule conjI*) **apply** *assumption*
apply (*rule conjI*) **apply** (*rule extendsL-case*, *assumption+*,*simp*,*assumption+*,*simp*,*assumption*)
apply (*rule conjI*) **apply** *simp*
apply (*rule conjI*) **apply** (*rule identityEnvironment-cased*,*simp*,*assumption+*)
apply (*rule addEnv-rho-good*, *assumption+*,*simp*,*simp add*: *def-extend-def*, *simp*, *assumption+*, *simp*)
apply (*rule conjI*) **apply** (*rule rho-case*,*assumption+*,*simp*)
apply (*rule conjI*) **apply** *assumption*
apply (*rule conjI*) **apply** (*rule refl*)
apply (*rule refl*)
apply (*elim exE*)
apply (*elim conjE*)

apply (*rule-tac x=q′* **in** *exI*)
apply (*rule-tac x=ia* **in** *exI*)

apply (*rule-tac x=sa* **in** *exI*)
apply (*rule-tac x= ss* @ [((*h*, *k*), *k0*, (*q*, *ja*), *S*)] **in** *exI*)
apply (*rule-tac x=δ* ⊕ [*j* ↦ −*1*] **in** *exI*)
apply (*rule-tac x=max 0* (*m* − *1*) **in** *exI*)
apply (*rule-tac x=s+nr* **in** *exI*)

apply (*rule conjI*) **apply** (*frule rho-good*, *elim conjE*)
apply (*rule execSVMBalanced-CaseD* ,*assumption+*)
apply (*erule-tac V=*(*qs* ! *i*, *lss* ! *i*, *isi*, *csi*) = *trE p′ funm fname* (*rho* + *zip xs* (*decreasing* (*length xs*))) *ei* **in** *thin-rl*)
apply (*rule conjI*) **apply** (*subgoal-tac* (*nat* (*int td* + *int* (*length vs*)) − *length vs*) =
(*nat* (*int td* + *int* (*length vs*) − *int* (*length vs*))),*simp*,*simp*)
apply (*rule conjI*, *assumption*)
apply (*rule conjI*, *rule refl*)
apply (*rule conjI*) **apply** (*rule diff-CaseD*, *assumption+*)
apply (*frule-tac v=v* **in** *resources-gre-0*, *elim exE*, *elim conjE*)
apply (*rule conjI*) **apply** (*rule maxFreshCells-CaseD*, *assumption+*,*simp*)
apply (*rule maxFreshWords-CaseD*,*assumption+*,*simp*,*assumption*)

apply (*rule impI*)+
apply (*elim conjE*)

apply (*rule allI*)+
apply (*rule impI*, *elim conjE*)

apply (*rule-tac x*=*q* **in** *exI*)
apply (*rule-tac x*= *Suc* (*Suc* (*Suc* (*Suc j*))) **in** *exI*)

apply (*rule-tac x*=((*h*, *k*) ↓ *k0*, *k0*, (*q*, *Suc* (*Suc* (*Suc* (*Suc j*)))), *Val* (*execOp oper* (*atom2val E1 a1*) (*atom2val E1 a2*)) # *S'*) **in** *exI*)
apply (*rule-tac x*=[((*h*, *k*), *k0*, (*q*, *Suc* (*Suc* (*Suc j*))), *Val* (*execOp oper* (*atom2val E1 a1*) (*atom2val E1 a2*)) # *drop* (*topDepth rho*) *S*),
((*h*, *k*), *k0*, (*q*, Suc (*Suc j*)), *Val v* # *S*),
((*h*, *k*), *k0*, (*q*, *Suc j*), (*map* (*item2Stack k S*) [*atom2item rho a1*, *atom2item rho a2*]) @ *S*),
((*h*, *k*), *k0*, (*q*, *j*), *S*)] **in** *exI*)
apply (*rule-tac x*=[]$_k$ **in** *exI*)
apply (*rule-tac x*=*0* **in** *exI*)
apply (*rule-tac x*=*2* **in** *exI*)
apply (*subgoal-tac core* (*AppE f* [*a1*, *a2*] [] *a*))
prefer *2* **apply** (*rule all-exp-core*)
apply (*frule app-atoms*)
apply (*frule execSVMBalanced-App-primops*, *assumption*+, *erule conjE*)
apply (*rule conjI*) **apply** *simp*
apply (*rule conjI*) **apply** *clarsimp*
apply (*rule conjI*) **apply** *assumption*
apply (*rule conjI*) **apply** (*rule refl*)
apply (*rule conjI*) **apply** (*rule empty-diff*)
apply (*rule conjI*) **apply** (*rule maxFreshCells-App-primops*, *assumption*+)
apply (*frule lengthS-topDepth*)
apply (*rule maxFreshWords-App-primops*, *assumption*+)

apply (*rule impI*)+
apply (*elim conjE*)

apply (*rule allI*)+
apply (*rule impI*, *elim conjE*)

apply (*frule trE-app*, *assumption*+)
apply (*elim exE*, *elim conjE*)

apply (*frule finite-dom-h*,*elim conjE*)
apply (*drule mp*) **apply** (*erule finite-dom-h'*, *assumption*)
apply (*drule mp*, *simp*, *rule x-not-self*, *rule all-exp-core*)
apply (*drule mp*, *simp*)+
apply (*subst dom-map-insert*, *assumption*)
apply (*drule mp*, *simp*)+ **apply** *clarsimp*
apply (*drule mp*, *simp*)+ **apply** (*subst dom-map-insert*) **apply** (*elim conjE*) **apply** *clarsimp*

apply (*drule mp*)
 apply (*subgoal-tac closureCalled e defs* $\subseteq$ *closureCalled* (*AppE f as rr a*) *defs*)
  prefer *2* **apply** (*rule closureCalled-app*)
apply (*simp*,*erule subset-trans*,*assumption*)

apply (*erule-tac x*=[(*empty*,*0*,*0*)] + (*zip* (*xs* @ *rs*) (*decreasing* (*length xs* + *length rs*))) **in** *allE*)
apply (*erule-tac x*=*map* (*item2Stack k S*) (*map* (*atom2item rho*) *as*) @
 map (*item2Stack k S*) (*map* (*region2item rho*) *rr*) @ *drop td S* **in** *allE*)
apply (*erule-tac x*= *drop td S* **in** *allE*)
apply (*erule-tac x*=*k0* **in** *allE*)
apply (*erule-tac x*=((*h*, *Suc k*), *k0*, (*qf*, *0*), *map* (*item2Stack k S*) (*map* (*atom2item rho*) *as*) @
 map (*item2Stack k S*) (*map* (*region2item rho*) *rr*) @ *drop td S*) **in** *allE*)

apply (*erule-tac x*=*p'* **in** *allE*)
apply (*erule-tac x*=*qf* **in** *allE*)
apply (*erule-tac x*=*lsf* **in** *allE*)
apply (*erule-tac x*=*is* **in** *allE*)
apply (*erule-tac x*=*is* **in** *allE*)
apply (*erule-tac x*=*csf* **in** *allE*)
apply (*erule-tac x*=*0* **in** *allE*)
apply (*erule-tac x*=*tds* **in** *allE*)

apply (*drule mp*)
 apply (*rule conjI*) **apply** *assumption*
 apply (*rule conjI*) **apply** (*rule codestore-extend*,*assumption+*)
 apply (*rule conjI*) **apply** *simp*
 apply (*rule conjI*) **apply** (*frule rho-good*, *elim conjE*)
  apply (*frule app-distinct-boundVar*,*elim conjE*)
  apply (*subgoal-tac core* (*AppE f as rr a*))
  prefer *2* **apply** (*rule all-exp-core*)
  apply (*frule app-arguments-good*,*assumption+*,*elim conjE*)
  apply (*frule app-atoms*)
 apply (*rule identityEnvironment-App*,*simp*,*assumption+*,*rule sym*, *simp*, *rule sym*, *simp*,*assumption+*)
 apply (*rule conjI*) **apply** (*simp add*: *envAdd.simps*) **apply** (*simp add*: *Let-def*)
  apply (*subst length-decreasing-add-assoc*,*simp*)
 apply (*rule conjI*) **apply** *simp*
 apply (*rule conjI*) **apply** (*subgoal-tac core* (*AppE f as rr a*))
  prefer *2* **apply** (*rule all-exp-core*)
  apply (*frule app-arguments-good*, *assumption+*,*elim conjE*)
  apply *simp*
 apply (*rule refl*)

apply (*elim exE*)
apply (*elim conjE*)

apply (*rule-tac x=q′* **in** *exI*)
apply (*rule-tac x=i* **in** *exI*)

apply (*rule-tac x=sa* **in** *exI*)
apply (*rule-tac x= ss* @
[((*h*, *k*), *k0*, (*q*, *Suc* (*Suc j*)), *append* (*append* (*map* (*item2Stack k S*) (*map* (*atom2item rho*) *as*))
(*map* (*item2Stack k S*) (*map* (*region2item rho*) *rr*))) (*drop td S*)),
((*h*, *k*), *k0*, (*q*, *Suc j*), *append* (*append* (*map* (*item2Stack k S*) (*map* (*atom2item rho*) *as*))
(*map* (*item2Stack k S*) (*map* (*region2item rho*) *rr*))) *S*),
((*h*, *k*), *k0*, (*q*, *j*), *S*)] **in** *exI*)
apply (*rule-tac x=δ*(*k* + *1* := *None*) **in** *exI*)
apply (*rule-tac x=m* **in** *exI*)
apply (*rule-tac x=max* (*int n* + *int l*) (*s* + *int n* + *int l* − *int td*) **in** *exI*)

apply (*subgoal-tac core* (*AppE f as rr a*))
prefer *2* **apply** (*rule all-exp-core*)

apply (*rule conjI*) **apply** (*frule lengthS-topDepth*)
apply (*frule app-arguments-good*,*assumption+*,*elim conjE*)
apply (*rule-tac l=l* **in** *execSVMBalanced-App*,*assumption+*,*rule sym*,*simp*,*rule sym*,*simp*,*assumption+*)
apply (*rule conjI*) **apply** (*simp*, *rule restrictToRegion-monotone-Suc-k*, *assumption*)
apply (*rule conjI*, *assumption*)
apply (*rule conjI*) **apply** (*rule refl*)
apply (*rule conjI*) **apply** (*rule diff-App*, *assumption+*,*simp*)
apply (*frule-tac v=v* **in** *resources-gre-0*, *elim exE*, *elim conjE*)
apply (*rule conjI*) **apply** (*rule maxFreshCells-App*, *assumption+*, *simp*)
apply (*frule app-arguments-good*,*assumption+*)
apply (*frule lengthS-topDepth*)
apply (*frule-tac v=v* **in** *resources-gre-0*, *elim exE*, *elim conjE*)
by (*rule maxFreshWords-App*,*assumption+*, *simp*, *rule sym*, *simp*, *assumption+*, *simp*)

end