Certification of the translation from SVM to JVM

By Javier de Dios and Ricardo Peña

March, 2009

Contents

1 Java Source Language Some Auxiliary Definitions

theory *JBasis* **imports** *Main* **begin**

lemmas [*simp*] = *Let-def*

2 unique

constdefs
unique :: ('*a* × '*b*) *list* => *bool*
unique == *distinct* ∘ *map fst*

lemma *fst-in-set-lemma* [*rule-format* (*no-asm*)]:
(*x*, *y*) : *set xys* −−> *x* : *fst* ' *set xys*
apply (*induct-tac xys*)

apply *auto*
done

lemma *unique-Nil* [*simp*]: *unique* []
apply (*unfold unique-def*)
apply (*simp* (*no-asm*))
done

lemma *unique-Cons* [*simp*]: *unique* ((*x*,*y*)#*l*) = (*unique l* & (!*y*. (*x*,*y*) ~: *set l*))
apply (*unfold unique-def*)
apply (*auto dest*: *fst-in-set-lemma*)
done

lemma *unique-append* [*rule-format* (*no-asm*)]: *unique l'* ==> *unique l* -->
(!(*x*,*y*):*set l*. !(*x'*,*y'*):*set l'*. *x'* ~= *x*) --> *unique* (*l* @ *l'*)
apply (*induct-tac l*)
apply (*auto dest*: *fst-in-set-lemma*)
done

lemma *unique-map-inj* [*rule-format* (*no-asm*)]:
unique l --> *inj f* --> *unique* (*map* (%(*k*,*x*). (*f k*, *g k x*)) *l*)
apply (*induct-tac l*)
apply (*auto dest*: *fst-in-set-lemma simp add*: *inj-eq*)
done

3 More about Maps

lemma *map-of-SomeI* [*rule-format* (*no-asm*)]:
unique l --> (*k*, *x*) : *set l* --> *map-of l k* = *Some x*
apply (*induct-tac l*)
apply *auto*
done

lemma *Ball-set-table'*:
(∀(*x*,*y*)∈*set l*. *P x y*) --> (∀ *x*. ∀ *y*. *map-of l x* = *Some y* --> *P x y*)
apply(*induct-tac l*)
apply(*simp-all* (*no-asm*))
apply *safe*
apply *auto*
done
lemmas *Ball-set-table* = *Ball-set-table'* [*THEN mp*]

lemma *table-of-remap-SomeD* [*rule-format* (*no-asm*)]:
map-of (*map* (λ((*k*,*k'*),*x*). (*k*,(*k'*,*x*))) *t*) *k* = *Some* (*k'*,*x*) -->
map-of t (*k*, *k'*) = *Some x*
apply (*induct-tac t*)
apply *auto*
done

end

4 Java types

theory *Type* **imports** *JBasis* **begin**

— typedecl cnam changed to:
types *cnam* = *string*

— exceptions
datatype
xcpt
= *NullPointer*
| *ClassCast*
| *OutOfMemory*
| *ArrayIndexOutOfBounds*
| *NegativeArraySize*
| *ArrayStore*

— class names
datatype *cname*
= *Object*
| *Xcpt xcpt*
| *Cname cnam*

— variable or field name
— typedecl vnam changed to
types *vnam* = *string*

— method name
— typedecl mname changed to
types *mname* = *string*

— names for *This* pointer and local/field variables
datatype *vname*
= *This*
| *VName vnam*

— primitive type, cf. 4.2
datatype *prim-ty*
= *Void* — 'result type' of void methods
| *Boolean*
| *Integer*
| *RetA nat* — bytecode return addresses

— reference type, cf. 4.3
datatype *ref-ty*
= *NullT* — null type, cf. 4.1

| *ClassT cname* — class type
| *ArrayT ty* — array type
— any type, cf. 4.1
and *ty*
= *PrimT prim-ty* — primitive type
| *RefT ref-ty* — reference type

syntax
NT :: *ty*
Class :: *cname* => *ty*
RA :: *nat* => *ty*
Array :: *ty* ⇒ *ty* (-.[] [*90*] *90*)

translations
NT == *RefT NullT*
Class C == *RefT* (*ClassT C*)
T.[] == *RefT* (*ArrayT T*)
RA pc == *PrimT* (*RetA pc*)

consts
the-Class :: *ty* ⇒ *cname*
isArray :: *ty* ⇒ *bool*

recdef *the-Class* {}
the-Class (*Class C*) = *C*
the-Class (*T*.[]) = *Object*

recdef *isArray* {}
isArray (*T*.[]) = *True*
isArray T = *False*

end

5 Class Declarations and Programs

theory *Decl* **imports** *Type* **begin**

types
fdecl = *vname* × *ty* — field declaration, cf. 8.3 (, 9.3)

sig = *mname* × *ty list* — signature of a method, cf. 8.4.2

'c mdecl = *sig* × *ty* × *'c* — method declaration in a class

'c class = *cname* × *fdecl list* × *'c mdecl list*
— class = superclass, fields, methods

$'c\ cdecl = cname \times 'c\ class$ — class declaration, cf. 8.1

$'c\ prog = 'c\ cdecl\ list$ — program

translations
$fdecl <= (type)\ vname \times ty$
$sig <= (type)\ mname \times ty\ list$
$mdecl\ c <= (type)\ sig \times ty \times c$
$class\ c <= (type)\ cname \times fdecl\ list \times (c\ mdecl)\ list$
$cdecl\ c <= (type)\ cname \times (c\ class)$
$prog\ c <= (type)\ (c\ cdecl)\ list$

constdefs
$class :: 'c\ prog => (cname \rightharpoonup 'c\ class)$
$class \equiv map\text{-}of$

$is\text{-}class :: 'c\ prog => cname => bool$
$is\text{-}class\ G\ C \equiv class\ G\ C \neq None$

lemma *finite-is-class*: $finite\ \{C.\ is\text{-}class\ G\ C\}$
apply (*unfold is-class-def class-def*)
apply (*fold dom-def*)
apply (*rule finite-dom-map-of*)
done

consts
$is\text{-}type :: 'c\ prog \Rightarrow ty \Rightarrow bool$
$isrtype :: 'c\ prog \Rightarrow ref\text{-}ty \Rightarrow bool$
primrec
$is\text{-}type\ G\ (PrimT\ T) = True$
$is\text{-}type\ G\ (RefT\ T) = isrtype\ G\ T$
$isrtype\ G\ (NullT) = True$
$isrtype\ G\ (ClassT\ C) = is\text{-}class\ G\ C$
$isrtype\ G\ (ArrayT\ T) = is\text{-}type\ G\ T$

consts
$is\text{-}RA :: ty => bool$
recdef *is-RA* {}
$is\text{-}RA\ (RA\ pc) = True$
$is\text{-}RA\ t = False$

end

6 Relations between Java Types

theory *TypeRel* **imports** *Decl* **begin**

— direct subclass, cf. 8.1.3
inductive
  subcls1 :: *'c prog* => [*cname*, *cname*] => *bool* (- ⊢ - ≺*C1* - [*71*,*71*,*71*] *70*)
  for *G* :: *'c prog*
where
  subcls1I: ⟦*class G C* = *Some* (*D*,*rest*); *C* ≠ *Object*⟧ ⟹ *G*⊢*C*≺*C1D*

abbreviation
  subcls :: *'c prog* => [*cname*, *cname*] => *bool* (- ⊢ - ⪯*C* - [*71*,*71*,*71*] *70*)
  where *G*⊢*C* ⪯*C* *D* ≡ (*subcls1 G*)^** *C D*

lemma *subcls1D*:
  G⊢*C*≺*C1D* ⟹ *C* ≠ *Object* ∧ (∃*fs ms*. *class G C* = *Some* (*D*,*fs*,*ms*))
apply (*erule subcls1.cases*)
apply *auto*
done

lemma *subcls1-def2*:
  subcls1 G = (λ*C D*. (*C*, *D*) ∈
    (*SIGMA C*: {*C*. *is-class G C*} . {*D*. *C*≠*Object* ∧ *fst* (*the* (*class G C*))=*D*}))
  by (*auto simp add*: *is-class-def expand-fun-eq dest*: *subcls1D intro*: *subcls1I*)

lemma *finite-subcls1*: *finite* {(*C*, *D*). *subcls1 G C D*}
apply(*simp add*: *subcls1-def2 del*: *mem-Sigma-iff*)
thm *finite-SigmaI*
apply(*rule finite-SigmaI* [*OF finite-is-class*])
thm *finite-subset*
apply(*rule-tac B* = {*fst* (*the* (*class G C*))} **in** *finite-subset*)
apply *auto*
done

lemma *subcls-is-class*: (*subcls1 G*)^++ *C D* ==> *is-class G C*
apply (*unfold is-class-def*)
apply(*erule tranclp-trans-induct*)
apply (*auto dest*!: *subcls1D*)
done

lemma *subcls-is-class2* [*rule-format* (*no-asm*)]:
  G⊢*C*⪯*C D* ⟹ *is-class G D* ⟶ *is-class G C*
apply (*unfold is-class-def*)
apply (*erule rtranclp-induct*)
apply (*drule-tac* [*2*] *subcls1D*)
apply *auto*
done

constdefs
class-rec :: *'c prog* ⇒ *cname* ⇒ *'a* ⇒
(*cname* ⇒ *fdecl list* ⇒ *'c mdecl list* ⇒ *'a*
⇒ *'a*) ⇒ *'a*
class-rec G == *wfrec* {(*C*, *D*). (*subcls1 G*)^−−1 *C D*}
(λ*r C t f*. *case class G C of*
None ⇒ *arbitrary*
| *Some* (*D*,*fs*,*ms*) ⇒
f C fs ms (*if C* = *Object then t else r D t f*))

lemma *class-rec-lemma*: *wfP* ((*subcls1 G*)^−−1) ⟹ *class G C* = *Some* (*D*,*fs*,*ms*)
⟹
class-rec G C t f = *f C fs ms* (*if C*=*Object then t else class-rec G D t f*)
by (*simp add*: *class-rec-def wfrec* [*to-pred*, **where** *r*=(*subcls1 G*)^−−1, *simplified*]
cut-apply [**where** *r*={(*C*, *D*). *subcls1 G D C*}, *simplified*, *OF subcls1I*])

definition
wf-class G = *wfP* ((*subcls1 G*)^−−1)

lemma *class-rec-func* [*code func*]:
class-rec G C t f = (*if wf-class G then*
(*case class G C*
of None ⇒ *arbitrary*
| *Some* (*D*, *fs*, *ms*) ⇒ *f C fs ms* (*if C* = *Object then t else class-rec G D t f*))
else class-rec G C t f)
proof (*cases wf-class G*)
case *False* **then show** *?thesis* **by** *auto*
next
case *True*
from ‹*wf-class G*› **have** *wf*: *wfP* ((*subcls1 G*)^−−1)
unfolding *wf-class-def* **.**
show *?thesis*
proof (*cases class G C*)
case *None*
with *wf* **show** *?thesis*
by (*simp add*: *class-rec-def wfrec* [*to-pred*, **where** *r*=(*subcls1 G*)^−−1, *simplified*]
cut-apply [**where** *r*={(*C*, *D*).*subcls1 G D C*}, *simplified*, *OF subcls1I*])
next
case (*Some x*) **show** *?thesis*
proof (*cases x*)
case (*fields D fs ms*)
then have *is-some*: *class G C* = *Some* (*D*, *fs*, *ms*) **using** *Some* **by** *simp*
note *class-rec* = *class-rec-lemma* [*OF wf is-some*]

```
    show ?thesis unfolding class-rec by (simp add: is-some)
   qed
  qed
qed

consts

  method :: 'c prog × cname => ( sig  ⇀ cname × ty × 'c)
  method' :: 'c prog × cname => ( sig  ⇀ cname × ty × 'c)
  field  :: 'c prog × cname => ( vname ⇀ cname × ty    )
  fields :: 'c prog × cname => ((vname × cname) × ty) list

— methods of a class, with inheritance, overriding and hiding, cf. 8.4.6

defs method-def: method ≡ λ(G,C). class-rec G C empty (λC fs ms ts.
                        ts ++ map-of (map (λ(s,m). (s,(C,m))) ms))

— methods of a class, with out inheritance, overriding and hiding
defs method'-def: method' ≡ λ(G,C). map-of (map (λ(s,m). (s,(C,m)))
                                            (snd (snd (the (class G C)))))


lemma method-rec-lemma: [|class G C = Some (D,fs,ms); wfP ((subcls1 G)^--1)|]
==>
  method (G,C) = (if C = Object then empty else method (G,D)) ++
  map-of (map (λ(s,m). (s,(C,m))) ms)
apply (unfold method-def)
apply (simp split del: split-if)
apply (erule (1) class-rec-lemma [THEN trans])
apply auto
done


— list of fields of a class, including inherited and hidden ones
defs fields-def: fields ≡ λ(G,C). class-rec G C []    (λC fs ms ts.
                        map (λ(fn,ft). ((fn,C),ft)) fs @ ts)

lemma fields-rec-lemma: [|class G C = Some (D,fs,ms); wfP ((subcls1 G)^--1)|]
==>
 fields (G,C) =
  map (λ(fn,ft). ((fn,C),ft)) fs @ (if C = Object then [] else fields (G,D))
apply (unfold fields-def)
apply (simp split del: split-if)
apply (erule (1) class-rec-lemma [THEN trans])
apply auto
done


defs field-def: field == map-of o (map (λ((fn,fd),ft). (fn,(fd,ft)))) o fields
```

lemma *table-of-remap-SomeD* [*rule-format* (*no-asm*)]:
map-of (*map* (λ((*k*,*k'*),*x*). (*k*,(*k'*,*x*))) *t*) *k* = *Some* (*k'*,*x*) −−>
map-of t (*k*, *k'*) = *Some x*
apply (*induct-tac t*)
apply *auto*
done

lemma *field-fields*:
field (*G*,*C*) *fn* = *Some* (*fd*, *fT*) ⟹ *map-of* (*fields* (*G*,*C*)) (*fn*, *fd*) = *Some fT*
apply (*unfold field-def*)
apply (*rule table-of-remap-SomeD*)
apply *simp*
done

— widening, viz. method invocation conversion,cf. 5.3 i.e. sort of syntactic subtyping
inductive
widen :: 'c *prog* => [*ty* , *ty*] => *bool* (- ⊢ - ≼ - [71,71,71] 70)
for *G* :: 'c *prog*
where
refl [*intro*!, *simp*]: *G*⊢ *T* ≼ *T* — identity conv., cf. 5.1.1
| *subcls* : *G*⊢*C*≼*C D* ==> *G*⊢*Class C* ≼ *Class D*
| *null* [*intro*!]: *G*⊢ *NT* ≼ *RefT R*

lemmas *refl* = *HOL.refl*

— casting conversion, cf. 5.5 / 5.1.5
— left out casts on primitve types
inductive
cast :: 'c *prog* => [*ty* , *ty*] => *bool* (- ⊢ - ≼? - [71,71,71] 70)
for *G* :: 'c *prog*
where
widen: *G*⊢ *C*≼ *D* ==> *G*⊢*C* ≼? *D*
| *subcls*: *G*⊢ *D*≼*C C* ==> *G*⊢*Class C* ≼? *Class D*

lemma *widen-PrimT-RefT* [*iff*]: (*G*⊢*PrimT pT*≼*RefT rT*) = *False*
apply (*rule iffI*)
apply (*erule widen.cases*)
apply *auto*
done

lemma *widen-RefT*: *G*⊢*RefT R*≼*T* ==> ∃ *t*. *T*=*RefT t*
apply (*ind-cases G*⊢*RefT R*≼*T*)
apply *auto*
done

lemma *widen-RefT2*: $G\vdash S\preceq RefT\ R ==> \exists\, t.\ S=RefT\ t$
apply (*ind-cases* $G\vdash S\preceq RefT\ R$)
apply *auto*
done

lemma *widen-Class*: $G\vdash Class\ C\preceq T ==> \exists\, D.\ T=Class\ D$
apply (*ind-cases* $G\vdash Class\ C\preceq T$)
apply *auto*
done

lemma *widen-Class-NullT* [*iff*]: $(G\vdash Class\ C\preceq NT) = False$
apply (*rule iffI*)
apply (*ind-cases* $G\vdash Class\ C\preceq NT$)
apply *auto*
done

lemma *widen-Class-Class* [*iff*]: $(G\vdash Class\ C\preceq Class\ D) = (G\vdash C\preceq C\ D)$
apply (*rule iffI*)
apply (*ind-cases* $G\vdash Class\ C \preceq Class\ D$)
apply (*auto elim*: *widen.subcls*)
done

lemma *widen-NT-Class* [*simp*]: $G \vdash T \preceq NT \Longrightarrow G \vdash T \preceq Class\ D$
by (*ind-cases* $G \vdash T \preceq NT$, *auto*)

lemma *cast-PrimT-RefT* [*iff*]: $(G\vdash PrimT\ pT\preceq?\ RefT\ rT) = False$
apply (*rule iffI*)
apply (*erule cast.cases*)
apply *auto*
done

lemma *cast-RefT*: $G \vdash C \preceq?\ Class\ D \Longrightarrow \exists\ rT.\ C = RefT\ rT$
apply (*erule cast.cases*)
apply *simp* **apply** (*erule widen.cases*)
apply *auto*
done

theorem *widen-trans*[*trans*]: $[\![G\vdash S\preceq U;\ G\vdash U\preceq T]\!] \Longrightarrow G\vdash S\preceq T$
proof −
  assume $G\vdash S\preceq U$ **thus** $\bigwedge T.\ G\vdash U\preceq T \Longrightarrow G\vdash S\preceq T$
  proof *induct*
    case (*refl T T'*) **thus** $G\vdash T\preceq T'$ **.**
  next
    case (*subcls C D T*)
    then obtain *E* **where** $T = Class\ E$ **by** (*blast dest*: *widen-Class*)
    with *subcls* **show** $G\vdash Class\ C\preceq T$ **by** *auto*
  next
    case (*null R RT*)

then obtain *rt* **where** *RT* = *RefT rt* **by** (*blast dest*: *widen-RefT*)
thus *G*⊢*NT*⪯*RT* **by** *auto*
qed
qed

end

7 Java Values

theory *Value* **imports** *Type* **begin**

— typedecl loc' locations, i.e. abstract references on objects

types *loc'* = *nat* — locations, i.e. abstract references on objects

datatype *loc*
= *XcptRef xcpt* — special locations for pre-allocated system exceptions
| *Loc loc'* — usual locations (references on objects)

datatype *val*
= *Unit* — dummy result value of void methods
| *Null* — null reference
| *Bool bool* — Boolean value
| *Intg int* — integer value, name Intg instead of Int because of clash with HOL/Set.thy
| *Addr loc* — addresses, i.e. locations of objects
| *RetAddr nat* — return address of JSR instruction, for bytecode only

consts
the-Bool :: *val* => *bool*
the-Intg :: *val* => *int*
the-Addr :: *val* => *loc*
the-RetAddr :: *val* => *nat*

primrec
the-Bool (*Bool b*) = *b*

primrec
the-Intg (*Intg i*) = *i*

primrec
the-Addr (*Addr a*) = *a*

primrec
the-RetAddr (*RetAddr r*) = *r*

consts

defpval :: *prim-ty* => *val* — default value for primitive types
default-val :: *ty* => *val* — default value for all types

primrec
defpval Void = *Unit*
defpval Boolean = *Bool False*
defpval Integer = *Intg 0*
defpval (*RetA pc*) = *RetAddr pc*

primrec
default-val (*PrimT pt*) = *defpval pt*
default-val (*RefT r*) = *Null*

end

8 Program State

theory *State* **imports** *TypeRel Value* **begin**

types
fields- = (*vname* × *cname* ⇀ *val*) — field name, defining class to value
entries- = *nat* ⇀ *val* — array index to value

datatype
heap-entry = *Obj cname fields-* — class instance with class name and fields
| *Arr ty nat entries-* — array with component type, length, and entries

constdefs
obj-ty :: *heap-entry* ⇒ *ty*
obj-ty entry ≡ *case entry of Obj C fs* ⇒ *Class C* | *Arr T len entries* ⇒ *T*.[]

init-vars :: (′*a* × *ty*) *list* => (′*a* ⇀ *val*)
init-vars ≡ *map-of* ∘ *map* (λ(*n*,*T*). (*n*,*default-val T*))

consts
the-obj :: *heap-entry* ⇒ *cname* × *fields-*
the-arr :: *heap-entry* ⇒ *ty* × *nat* × *entries-*

recdef *the-obj* {}
the-obj (*Obj C fs*) = (*C*,*fs*)

recdef *the-arr* {}
the-arr (*Arr T len entries*) = (*T*,*len*,*entries*)

types *aheap* = *loc* ⇀ *heap-entry* — "*heap*" used in a translation below
locals = *vname* ⇀ *val* — simple state, i.e. variable contents
state = *aheap* × *locals* — heap, local parameter including This
xstate = *xcpt option* × *state* — state including exception information

syntax
heap :: *state* => *aheap*
locals :: *state* => *locals*
Norm :: *state* => *xstate*

translations
heap => *fst*
locals => *snd*
Norm s == (*None*,*s*)

constdefs
new-Addr :: *aheap* => *loc* × *xcpt option*
new-Addr h == *SOME* (*a*,*x*). (*h a* = *None* ∧ *x* = *None*) | *x* = *Some OutOfMemory*

raise-if :: *bool* => *xcpt* => *xcpt option* => *xcpt option*
raise-if c x xo == *if c* ∧ (*xo* = *None*) *then Some x else xo*

np :: *val* => *xcpt option* => *xcpt option*
np v == *raise-if* (*v* = *Null*) *NullPointer*

c-hupd :: *aheap* => *xstate* => *xstate*
c-hupd h'== λ(*xo*,(*h*,*l*)). *if xo* = *None then* (*None*,(*h'*,*l*)) *else* (*xo*,(*h*,*l*))

cast-ok :: *'c prog* => *cname* => *aheap* => *val* => *bool*
cast-ok G C h v == *v* = *Null* ∨ *G*⊢*obj-ty* (*the* (*h* (*the-Addr v*)))≼ *Class C*

lemma *obj-ty-def2* [*simp*]: *obj-ty* (*Obj C fs*) = *Class C*
apply (*unfold obj-ty-def*)
apply (*simp* (*no-asm*))
done

lemma *obj-ty-def3* [*simp*]: *obj-ty* (*Arr T len entries*) = *T*.[]
by (*unfold obj-ty-def*) *simp*

lemma *raise-if-True* [*simp*]: *raise-if True x y* ≠ *None*
apply (*unfold raise-if-def*)
apply *auto*
done

lemma *raise-if-False* [*simp*]: *raise-if False x y = y*
apply (*unfold raise-if-def*)
apply *auto*
done

lemma *raise-if-Some* [*simp*]: *raise-if c x (Some y) ≠ None*
apply (*unfold raise-if-def*)
apply *auto*
done

lemma *raise-if-Some2* [*simp*]:
raise-if c z (if x = None then Some y else x) ≠ None
apply (*unfold raise-if-def*)
apply(*induct-tac x*)
apply *auto*
done

lemma *raise-if-SomeD* [*rule-format* (*no-asm*)]:
raise-if c x y = Some z ⟶ c ∧ Some z = Some x | y = Some z
apply (*unfold raise-if-def*)
apply *auto*
done

lemma *raise-if-NoneD* [*rule-format* (*no-asm*)]:
raise-if c x y = None −−> ¬ c ∧ y = None
apply (*unfold raise-if-def*)
apply *auto*
done

lemma *np-NoneD* [*rule-format* (*no-asm*)]:
np a′ x′ = None −−> x′ = None ∧ a′ ≠ Null
apply (*unfold np-def raise-if-def*)
apply *auto*
done

lemma *np-None* [*rule-format* (*no-asm*), *simp*]: *a′ ≠ Null −−> np a′ x′ = x′*
apply (*unfold np-def raise-if-def*)
apply *auto*
done

lemma *np-Some* [*simp*]: *np a′ (Some xc) = Some xc*
apply (*unfold np-def raise-if-def*)
apply *auto*
done

lemma *np-Null* [*simp*]: *np Null None = Some NullPointer*
apply (*unfold np-def raise-if-def*)
apply *auto*
done

lemma *np-Addr* [*simp*]: *np* (*Addr a*) *None* = *None*
apply (*unfold np-def raise-if-def*)
apply *auto*
done

lemma *np-raise-if* [*simp*]: (*np Null* (*raise-if c xc None*)) =
Some (*if c then xc else NullPointer*)
apply (*unfold raise-if-def*)
apply (*simp* (*no-asm*))
done

syntax (*xsymbols*)
conf :: *'c prog => aheap => val => ty => bool*
(-,- ⊢ - ::≼ - [*51*,*51*,*51*,*51*] *50*)

end

9 Java Virtual Machine State of the JVM

theory *JVMState*
imports *State*
begin

For object initialization, we tag each location with the current init status. The tags use an extended type system for object initialization (that gets reused in the BV).

We have either

- usual initialized types, or
- a class that is not yet initialized and has been created by a *New* instruction at a certain line number, or
- a partly initialized class (on which the next super class constructor has to be called). We store the name of the class the super class constructor has to be called of.

datatype *init-ty* = *Init ty* | *UnInit cname nat* | *PartInit cname*

10 Frame Stack

types

opstack = *val list*
locvars = *val list*
p-count = *nat*
ref-upd = (*val* × *val*)
sheap = *cname* × *vname* ⇀ *val*

frame = *opstack* ×
locvars ×
cname ×
sig ×
p-count ×
ref-upd

— operand stack
— local variables (including this pointer and method parameters)
— name of class where current method is defined
— method name + parameter types
— program counter within frame
— ref update for obj init proof

11 Exceptions

constdefs
raise-system-xcpt :: *bool* ⇒ *xcpt* ⇒ *val option*
raise-system-xcpt b x ≡ *if b then Some* (*Addr* (*XcptRef x*)) *else None*

— redefines State.new_Addr:
new-Addr :: *aheap* ⇒ *loc* × *val option*
new-Addr h ≡ *let* (*a*, *x*) = *State.new-Addr h*
in (*a*, *raise-system-xcpt* (*x* ~= *None*) *OutOfMemory*)

types
init-heap = *loc* ⇒ *init-ty*
— type tags to track init status of objects

jvm-state = *val option* × *sheap* × *aheap* × *init-heap* × *frame list*
— exception flag,static heap, heap, tag heap, frames

a new, blank object with default values in all fields:

constdefs
blank :: 'c *prog* ⇒ *cname* ⇒ *heap-entry*
blank G C ≡ *Obj C* (*init-vars* (*fields*(*G*,*C*)))

blank-arr :: *ty* ⇒ *nat* ⇒ *heap-entry*
blank-arr T len ≡ *Arr T len* (λ*x*. *if x* < *len then Some* (*default-val T*) *else None*)

start-heap :: 'c *prog* ⇒ *aheap*
start-heap G ≡ *empty* (*XcptRef NullPointer* ↦ *blank G* (*Xcpt NullPointer*))

$(XcptRef\ ClassCast \mapsto blank\ G\ (Xcpt\ ClassCast))$
$(XcptRef\ OutOfMemory \mapsto blank\ G\ (Xcpt\ OutOfMemory))$
$(XcptRef\ ArrayIndexOutOfBounds \mapsto blank\ G\ (Xcpt\ ArrayIndex-OutOfBounds))$
$(XcptRef\ NegativeArraySize \mapsto blank\ G\ (Xcpt\ NegativeArraySize))$
$(XcptRef\ ArrayStore \mapsto blank\ G\ (Xcpt\ ArrayStore))$

$start\text{-}sheap :: 'c\ prog \Rightarrow sheap$
$start\text{-}sheap\ G \equiv empty$

$start\text{-}iheap :: init\text{-}heap$
$start\text{-}iheap \equiv ((((((\lambda x.\ arbitrary)$
$(XcptRef\ NullPointer := Init\ (Class\ (Xcpt\ NullPointer))))$
$(XcptRef\ ClassCast := Init\ (Class\ (Xcpt\ ClassCast))))$
$(XcptRef\ OutOfMemory := Init\ (Class\ ((Xcpt\ OutOfMemory)))))$
$(XcptRef\ ArrayIndexOutOfBounds := Init\ (Class\ (Xcpt\ ArrayIndex-OutOfBounds))))$
$(XcptRef\ NegativeArraySize := Init\ (Class\ (Xcpt\ NegativeArraySize))))$
$(XcptRef\ ArrayStore := Init\ (Class\ (Xcpt\ ArrayStore)))$

end

12 Primitive operators for SVM anf JVM

theory *BinOP*
imports *Main*
begin

Primitive operators

datatype $PrimOp = Add \mid Substract \mid Times \mid Divide \mid LessThan \mid LessEqual \mid Equal \mid GreaterThan \mid GreaterEqual \mid NotEqual$

end

13 Instructions of the JVM

theory *JVMInstructions*
imports *JVMState BinOP* **begin**

Apply Binary Operation

consts
$applyBinOp :: [PrimOp, val, val] \Rightarrow val$

primrec
applyBinOp Equal b1 b2 = (if (the-Intg(b1) = the-Intg(b2)) then Intg 1 else Intg 0)
applyBinOp NotEqual b1 b2 = (if (the-Intg(b1) ≠ the-Intg(b2)) then Intg 1 else Intg 0)
applyBinOp GreaterEqual b1 b2 = (if (the-Intg(b1) ≥ the-Intg(b2)) then Intg 1 else Intg 0)
applyBinOp GreaterThan b1 b2 = (if (the-Intg(b1) > the-Intg(b2)) then Intg 1 else Intg 0)
applyBinOp LessThan b1 b2 = (if (the-Intg(b1) < the-Intg(b2)) then Intg 1 else Intg 0)
applyBinOp LessEqual b1 b2 = (if (the-Intg(b1) ≤ the-Intg(b2)) then Intg 1 else Intg 0)
applyBinOp Add b1 b2 = Intg (the-Intg(b1) + the-Intg(b2))
applyBinOp Substract b1 b2 = Intg (the-Intg(b1) − the-Intg(b2))
applyBinOp Times b1 b2 = Intg (the-Intg(b1) ∗ the-Intg(b2))
applyBinOp Divide b1 b2 = Intg (the-Intg(b1) div the-Intg(b2))

Apply ifcmp Generic for integers

consts
applyIf :: [*PrimOp*,*val*,*val*] ⇒ *bool*

primrec
applyIf Equal b1 b2 =(the-Intg(b1) = the-Intg(b2))
applyIf NotEqual b1 b2 = (the-Intg(b1) ≠ the-Intg(b2))
applyIf GreaterEqual b1 b2 = (the-Intg(b1) ≥ the-Intg(b2))
applyIf GreaterThan b1 b2 = (the-Intg(b1) > the-Intg(b2))
applyIf LessThan b1 b2 = (the-Intg(b1) < the-Intg(b2))
applyIf LessEqual b1 b2 = (the-Intg(b1) ≤ the-Intg(b2))

datatype
instr = *Load nat* — load from local variable
| *Store nat* — store into local variable
| *LitPush val* — push a literal (constant)
| *New cname* — create object
| *Getfield vname cname* — Fetch field from object
| *Putfield vname cname* — Set field in object
| *Checkcast cname* — Check whether object is of given type
| *Invoke cname mname* (*ty list*) — inv. instance meth of an object
| *Invoke-special cname mname ty list*
— no dynamic type lookup, for constructors
| *Return* — return from method
| *Return-Void* — return from void method
| *Pop* — pop top element from opstack
| *Dup* — duplicate top element of opstack

| *Dup2* — duplicate two element of opstack
| *Dup-x1* — duplicate next to top element
| *Dup-x2* — duplicate 3rd element
| *Swap* — swap top and next to top element
| *IAdd* — integer addition
| *Goto int* — goto relative address
| *Ifcmpeq int* — branch if int/ref comparison succeeds
| *Throw* — throw top of stack as exception
| *Jsr int* — jump to subroutine
| *Ret nat* — return from subroutine
| *ArrLoad* — load indexed entry from array
| *ArrStore* — store value into indexed array entry
| *ArrLength* — get array length
| *ArrNew ty* — create new 1-dimensional array
| *Invoke-static cname mname* (*ty list*) — invoke a class (static) method
| *Getstatic vname cname* — get value of static field
| *Putstatic vname cname* — set value of static field
| *Tableswitch int int* (*int list*) — jump according to a table
| *BinOp PrimOp* — Binary Operation
| *Ifcmp PrimOp int* — Generic Branch

types
bytecode = *instr list*
exception-entry = *p-count* × *p-count* × *p-count* × *cname*
— start-pc, end-pc, handler-pc, exception type
exception-table = *exception-entry list*
jvm-method = *nat* × *nat* × *bytecode* × *exception-table*
jvm-prog = *jvm-method prog*

end

14 JVM Instruction Semantics

theory *JVMExecInstr*
imports *JVMInstructions*
begin

the method name of constructors:

consts
init :: *mname*

replace a by b in l:

constdefs
replace :: *'a* ⇒ *'a* ⇒ *'a list* ⇒ *'a list*
replace a b l == *map* (λ*x*. *if x* = *a then b else x*) *l*

some lemmas about replace

lemma *replace-removes-elem*:
a ≠ *b* ⟹ *a* ∉ *set* (*replace a b l*)
by (*unfold replace-def*) *auto*

lemma *replace-length* [*simp*]:
length (*replace a b l*) = *length l* **by** (*simp add*: *replace-def*)

lemma *replace-Nil* [*iff*]:
replace x y [] = [] **by** (*simp add*: *replace-def*)

lemma *replace-Cons*:
replace x y (*l*#*ls*) = (*if l* = *x then y else l*)#(*replace x y ls*)
by (*simp add*: *replace-def*)

lemma *replace-map*:
inj f ==> *replace* (*f x*) (*f y*) (*map f l*) = *map f* (*replace x y l*)
apply (*induct l*)
apply (*simp add*: *replace-def*)
apply (*simp add*: *replace-def*)
apply *clarify*
apply (*drule injD*, *assumption*)
apply *simp*
done

lemma *replace-id*:
x ∉ *set l* ∨ *x* = *y* ⟹ *replace x y l* = *l*
apply (*induct l*)
apply (*auto simp add*: *replace-def*)
done

single execution step for each instruction:

consts
exec-instr :: [*instr*, *jvm-prog*, *sheap*, *aheap*, *init-heap*, *opstack*, *locvars*,
cname, *sig*, *p-count*, *ref-upd*, *frame list*] => *jvm-state*
primrec
exec-instr (*Load idx*) *G shp hp ihp stk vars Cl sig pc z frs* =
(*None*,*shp*, *hp*, *ihp*, ((*vars* ! *idx*) # *stk*, *vars*, *Cl*, *sig*, *pc*+*1*, *z*)#*frs*)

exec-instr (*Store idx*) *G shp hp ihp stk vars Cl sig pc z frs* =
(*None*,*shp*, *hp*, *ihp*, (*tl stk*, *vars*[*idx*:=*hd stk*], *Cl*, *sig*, *pc*+*1*, *z*)#*frs*)

exec-instr (*LitPush v*) *G shp hp ihp stk vars Cl sig pc z frs* =
(*None*,*shp*, *hp*, *ihp*, (*v* # *stk*, *vars*, *Cl*, *sig*, *pc*+*1*, *z*)#*frs*)

exec-instr (*New C*) *G shp hp ihp stk vars Cl sig pc z frs* =
(*let* (*oref*,*xp'*) = *new-Addr hp*;
hp' = *if xp'*=*None then hp*(*oref* ↦ *blank G C*) *else hp*;
ihp' = *if xp'*=*None then ihp*(*oref* := *UnInit C pc*) *else ihp*;

```
      stk'      = if xp'=None then (Addr oref)#stk else stk;
      pc'       = if xp'=None then pc+1 else pc
  in (xp',shp, hp', ihp', (stk', vars, Cl, sig, pc', z)#frs))

exec-instr (Getfield F C) G shp hp ihp stk vars Cl sig pc z frs =
 (let oref    = hd stk;
      xp'     = raise-system-xcpt (oref=Null) NullPointer;
      (oc,fs) = the-obj (the(hp(the-Addr oref)));
      stk'    = if xp'=None then the(fs(F,C))#(tl stk) else tl stk;
      pc'     = if xp'=None then pc+1 else pc
  in  (xp',shp, hp, ihp, (stk', vars, Cl, sig, pc', z)#frs))

exec-instr (Putfield F C) G shp hp ihp stk vars Cl sig pc z frs =
 (let (fval,oref) = (hd stk, hd(tl stk));
      xp'         = raise-system-xcpt (oref=Null) NullPointer;
      a           = the-Addr oref;
     (oc,fs)      = the-obj (the(hp a));
      hp'         = if xp'=None then hp(a ↦ Obj oc (fs((F,C) ↦ fval)))
                                else hp;
      pc'         = if xp'=None then pc+1 else pc
  in  (xp',shp, hp', ihp, (tl (tl stk), vars, Cl, sig, pc', z)#frs))

exec-instr (Checkcast C) G shp hp ihp stk vars Cl sig pc z frs =
 (let oref  = hd stk;
      xp'   = raise-system-xcpt (¬cast-ok G C hp oref) ClassCast;
      stk'  = if xp'=None then stk else tl stk;
      pc'   = if xp'=None then pc+1 else pc
  in  (xp',shp, hp, ihp, (stk', vars, Cl, sig, pc', z)#frs))

exec-instr (Invoke C mn ps) G shp hp ihp stk vars Cl sig pc z frs =
 (let n    = length ps;
      args = take n stk;
      oref = stk!n;
      xp'  = raise-system-xcpt (oref=Null) NullPointer;
      dynT = the-Class (obj-ty (the(hp (the-Addr oref))));
      (dc,mh,mxs,mxl,c) = the (method (G,dynT) (mn,ps));
      frs' = (if xp'=None then
              [([],oref#(rev args)@(replicate mxl arbitrary),dc,(mn,ps),0,arbitrary)]
              else [])
  in (xp',shp, hp, ihp, frs'@(stk, vars, Cl, sig, pc, z)#frs))
```

— Because exception handling needs the pc of the Invoke instruction,
— Invoke doesn't change *stk* and *pc* yet (*Return* does that).

```
exec-instr (Invoke-special C mn ps) G shp hp ihp stk vars Cl sig pc z frs =
 (let n    = length ps;
      args = take n stk;
      oref = stk!n;
```

```
      addr  = the-Addr oref;
      x'    = raise-system-xcpt (oref=Null) NullPointer;
      dynT  = the-Class (obj-ty (the(hp addr)));
      (dc,mh,mxs,mxl,c)= the (method (G,C) (mn,ps));
      (addr',x'') = new-Addr hp;
      xp'   = if x' = None then x'' else x';
      hp'   = if xp' = None then hp(addr' ↦ blank G dynT) else hp;
      ihp'  = if C = Object then ihp(addr':= Init (Class dynT))
              else ihp(addr' := PartInit C);
      ihp'' = if xp' = None then ihp' else ihp;
      z'    = if C = Object then (Addr addr', Addr addr') else (Addr addr', Null);
      frs'  = (if xp'=None then
             [([],(Addr addr')#(rev args)@(replicate mxl arbitrary),dc,(mn,ps),0,z')]
              else [])
  in (xp',shp, hp', ihp'', frs'@(stk, vars, Cl, sig, pc, z)#frs))


exec-instr Return G shp hp ihp stk0 vars Cl sig0 pc z0 frs =
 (if frs=[] then (None, shp, hp, ihp, [])
  else let
    val     = hd stk0;
    (mn,pt) = sig0;
    (stk,loc,C,sig,pc,z) = hd frs;
    (b,c)   = z0;
    (a,c')  = z;
    n       = length pt;
    addr    = stk!n;
    stk'    = if mn=init then val#replace addr c stk else val#stk;
    loc'    = if mn=init then replace addr c loc else loc;
    z'      = if mn=init ∧ z = (addr,Null) then (a,c) else z
  in (None, shp , hp, ihp, (stk',loc',C,sig,pc+1,z')#tl frs))

 — Return drops arguments from the caller's stack and increases
 — the program counter in the caller

 — z is only updated if we are in a constructor and have initialized the
 — same reference as the constructor in the frame above (otherwise we are
 — in the last constructor of the init chain)


exec-instr Return-Void G shp hp ihp stk0 vars Cl sig0 pc z0 frs =
 (if frs=[] then (None, shp, hp, ihp, [])
  else let
    (mn,pt) = sig0;
    (stk,loc,C,sig,pc,z) = hd frs;
    (b,c)   = z0;
    (a,c')  = z;
    n       = length pt;
    addr    = stk!n;
```

```
    stk'   = if mn=init then replace addr c stk else stk;
    loc'   = if mn=init then replace addr c loc else loc;
    z'     = if mn=init ∧ z = (addr,Null) then (a,c) else z
  in (None, shp , hp, ihp, (stk',loc',C,sig,pc+1,z')#tl frs))
```

— All items on the current method's operand stack are discarded

```
exec-instr Pop G shp hp ihp stk vars Cl sig pc z frs =
    (None, shp, hp, ihp, (tl stk, vars, Cl, sig, pc+1, z)#frs)

exec-instr Dup G shp hp ihp stk vars Cl sig pc z frs =
    (None, shp, hp, ihp, (hd stk # stk, vars, Cl, sig, pc+1, z)#frs)

exec-instr Dup2 G shp hp ihp stk vars Cl sig pc z frs =
    (None, shp, hp, ihp, (hd stk # hd (tl stk) # stk,vars,Cl,sig,pc+1,z)#frs)

exec-instr Dup-x1 G shp hp ihp stk vars Cl sig pc z frs =
    (None, shp, hp, ihp, (hd stk # hd (tl stk) # hd stk # (tl (tl stk)),
              vars, Cl, sig, pc+1, z)#frs)

exec-instr Dup-x2 G shp hp ihp stk vars Cl sig pc z frs =
    (None,shp, hp, ihp,
     (hd stk # hd (tl stk) # (hd (tl (tl stk))) # hd stk # (tl (tl (tl stk)))),
     vars, Cl, sig, pc+1, z)#frs)

exec-instr Swap G shp hp ihp stk vars Cl sig pc z frs =
    (let (val1,val2) = (hd stk,hd (tl stk))
    in (None,shp, hp, ihp, (val2#val1#(tl (tl stk)), vars, Cl, sig, pc+1, z)#frs))

exec-instr IAdd G shp hp ihp stk vars Cl sig pc z frs =
    (let (val1,val2) = (hd stk,hd (tl stk))
    in (None,shp, hp, ihp, (Intg ((the-Intg val1)+(the-Intg val2))#(tl (tl stk)),
       vars, Cl, sig, pc+1, z)#frs))

exec-instr (Ifcmpeq i) G shp hp ihp stk vars Cl sig pc z frs =
      (let (val1,val2) = (hd stk, hd (tl stk));
           pc' = if val1 = val2 then nat(int pc+i) else pc+1
       in (None,shp, hp, ihp, (tl (tl stk), vars, Cl, sig, pc', z)#frs))

exec-instr (Goto i) G shp hp ihp stk vars Cl sig pc z frs =
    (None,shp, hp, ihp, (stk, vars, Cl, sig, nat(int pc+i), z)#frs)

exec-instr Throw G shp hp ihp stk vars Cl sig pc z frs =
    (let xcpt  = raise-system-xcpt (hd stk = Null) NullPointer;
         xcpt' = if xcpt = None then Some (hd stk) else xcpt
     in (xcpt',shp, hp, ihp, (stk, vars, Cl, sig, pc, z)#frs))

exec-instr (Jsr i) G shp hp ihp stk vars Cl sig pc z frs =
```

$(None,shp,\ hp,\ ihp,\ (RetAddr\ (pc{+}1)\#stk,\ vars,\ Cl,\ sig,\ nat(int\ pc{+}i),\ z)\#frs)$

exec-instr $(Ret\ idx)\ G\ shp\ hp\ ihp\ stk\ vars\ Cl\ sig\ pc\ z\ frs =$
$(None,shp,hp,\ ihp,\ (stk,\ vars,\ Cl,\ sig,\ the\text{-}RetAddr\ (vars!idx),\ z)\#frs)$

exec-instr $(ArrLoad)\ G\ shp\ hp\ ihp\ stk\ vars\ Cl\ sig\ pc\ z\ frs =$
$(let\ idx = the\text{-}Intg\ (hd\ stk);$
$aref = hd\ (tl\ stk);$
$xp'' = raise\text{-}system\text{-}xcpt\ (aref{=}Null)\ NullPointer;$
$(T,l,en) = the\text{-}arr\ (the(hp(the\text{-}Addr\ aref)));$
$xp' = if\ xp''{=}None\ then\ raise\text{-}system\text{-}xcpt\ (int\ l \le idx \vee idx < 0)\ ArrayIndexOutOfBounds$
$else\ xp'';$
$stk' = the\ (en\ (nat\ idx))\ \#\ (tl\ (tl\ stk));$
$pc' = if\ xp'{=}None\ then\ pc{+}1\ else\ pc$
$in\ \ (xp',shp,\ hp,\ ihp,\ (stk',\ vars,\ Cl,\ sig,\ pc',\ z)\#frs))$

exec-instr $(ArrStore)\ G\ shp\ hp\ ihp\ stk\ vars\ Cl\ sig\ pc\ z\ frs =$
$(let\ val = hd\ stk;$
$idx = the\text{-}Intg\ (hd\ (tl\ stk));$
$aref = hd\ (tl\ (tl\ stk));$
$xp'' = raise\text{-}system\text{-}xcpt\ (aref{=}Null)\ NullPointer;$
$a = the\text{-}Addr\ aref;$
$(T,l,en) = the\text{-}arr\ (the(hp\ a));$
$xp''' = if\ xp''{=}None\ then\ raise\text{-}system\text{-}xcpt\ (int\ l \le idx \vee idx < 0)\ ArrayIndexOutOfBounds$
$else\ xp'';$
$xp' = if\ xp'''{=}None\ then\ raise\text{-}system\text{-}xcpt\ (\ False)\ ArrayStore\ else\ xp''';$
$hp' = if\ xp'{=}None\ then\ hp(a \mapsto Arr\ T\ l\ (en(nat\ idx \mapsto val)))\ else\ hp;$
$pc' = if\ xp'{=}None\ then\ pc{+}1\ else\ pc$
$in\ \ (xp',shp,\ hp',\ ihp,\ (tl\ (tl\ (tl\ stk)),\ vars,\ Cl,\ sig,\ pc',\ z)\#frs))$

exec-instr $(ArrLength)\ G\ shp\ hp\ ihp\ stk\ vars\ Cl\ sig\ pc\ z\ frs =$
$(let\ aref = hd\ stk;$
$xp' = raise\text{-}system\text{-}xcpt\ (aref{=}Null)\ NullPointer;$
$a = the\text{-}Addr\ aref;$
$(T,l,en) = the\text{-}arr\ (the(hp\ a));$
$pc' = if\ xp'{=}None\ then\ pc{+}1\ else\ pc$
$in\ \ (xp',\ shp,\ hp,\ ihp,\ (Intg\ (int\ l)\#(tl\ stk),\ vars,\ Cl,\ sig,\ pc',\ z)\#frs))$

exec-instr $(ArrNew\ T)\ G\ shp\ hp\ ihp\ stk\ vars\ Cl\ sig\ pc\ z\ frs =$
$(let\ len = the\text{-}Intg\ (hd\ stk);$
$xp''' = raise\text{-}system\text{-}xcpt\ (len < 0)\ NegativeArraySize;$
$(aref,xp'') = new\text{-}Addr\ hp;$
$xp' = if\ xp'''{=}None\ then\ xp''\ else\ xp''';$
$hp' = if\ xp'{=}None\ then\ hp(aref \mapsto blank\text{-}arr\ T\ (nat\ len))\ else\ hp;$
$ihp' = if\ xp'{=}None\ then\ ihp(aref := Init\ (T.[]))\ else\ ihp;$

```
      stk'       = (Addr aref)#tl stk;
      pc'        = if xp'=None then pc+1 else pc
  in  (xp',shp, hp', ihp', (stk', vars, Cl, sig, pc', z)#frs))


exec-instr (BinOp bop) G shp hp ihp stk vars Cl sig pc z frs =
 (let (val2,val1) = (hd stk, hd (tl stk))
  in  (None,shp, hp, ihp, ((applyBinOp bop val1 val2 )#(tl (tl stk)), vars, Cl, sig,
pc + 1, z)#frs))


exec-instr(Invoke-static C mn ps) G shp hp ihp stk vars Cl sig pc z frs =
 (let n    = length ps;
      args = take n stk;
      (dc,mh,mxs,mxl,c) = the (method (G,C) (mn,ps));
       frs' = [([], (rev args)@(replicate mxl arbitrary),dc,(mn,ps),0,arbitrary)]
  in  (None,shp,hp,ihp,frs'@(stk,vars,Cl,sig,pc,z)#frs))


exec-instr(Tableswitch n m t) G shp hp ihp stk vars Cl sig pc z frs =
 (let val = hd stk;
      v   = the-Intg val;
      pc' = if (v < n) ∨ (v > m) then pc + nat(t!(nat(m − n + 1)))  else pc +
nat(t!(nat(v − n)))
  in  (None,shp,hp,ihp,((tl stk),vars,Cl,sig,pc',z)#frs))


exec-instr (Getstatic F C) G shp hp ihp stk vars Cl sig pc z frs =
 (let v   =  the (shp (C,F));
      pc' = pc + 1
  in  (None,shp, hp, ihp, ((v#stk), vars, Cl, sig, pc', z)#frs))


exec-instr (Putstatic F C) G shp hp ihp stk vars Cl sig pc z frs =
 (let shp' =  shp((C,F) ↦ hd stk );
      pc'  = pc + 1
  in  (None, shp', hp, ihp, ((tl stk), vars, Cl, sig, pc', z)#frs))


exec-instr (Ifcmp bop i) G shp hp ihp stk vars Cl sig pc z frs =
 (let (val1,val2) = (hd stk, hd (tl stk));
     pc'       = if ((applyIf bop val1 val2) = True) then nat(int pc+i) else pc+1
  in  (None,shp, hp, ihp, (tl (tl stk), vars, Cl, sig, pc', z)#frs))


end
```

15 Exception handling in the JVM

theory *JVMExceptions*
imports *JVMInstructions*
begin
constdefs
match-exception-entry :: *jvm-prog* ⇒ *cname* ⇒ *p-count* ⇒ *exception-entry* ⇒ *bool*
match-exception-entry G cn pc ee ==
let (*start-pc*, *end-pc*, *handler-pc*, *catch-type*) = *ee in*
start-pc <= *pc* ∧ *pc* < *end-pc* ∧ *G*⊢ *cn* ⪯*C catch-type*

consts
match-exception-table :: *jvm-prog* ⇒ *cname* ⇒ *p-count* ⇒ *exception-table* ⇒ *p-count option*
primrec
match-exception-table G cn pc [] = *None*
match-exception-table G cn pc (*e*#*es*) = (*if match-exception-entry G cn pc e then Some* (*fst* (*snd* (*snd e*))) *else match-exception-table G cn pc es*)

consts
cname-of :: *aheap* ⇒ *val* ⇒ *cname*
ex-table-of :: *jvm-method* ⇒ *exception-table*

translations
cname-of hp v == *the-Class* (*obj-ty* (*the* (*hp* (*the-Addr v*))))
ex-table-of m == *snd* (*snd* (*snd m*))

consts
find-handler :: *jvm-prog* ⇒ *val option* ⇒ *sheap* ⇒ *aheap* ⇒ *init-heap* ⇒ *frame list* ⇒ *jvm-state*
primrec
find-handler G xcpt shp hp ihp [] = (*xcpt*,*shp*, *hp*, *ihp*, [])
find-handler G xcpt shp hp ihp (*fr*#*frs*) =
(*case xcpt of*
None ⇒ (*None*, *shp*,*hp*, *ihp*, *fr*#*frs*)
| *Some xc* ⇒
let (*stk*,*loc*,*C*,*sig*,*pc*,*r*) = *fr in*
(*case match-exception-table G* (*cname-of hp xc*) *pc*
(*ex-table-of* (*snd*(*snd*(*the*(*method* (*G*,*C*) *sig*))))) *of*
None ⇒ *find-handler G* (*Some xc*) *shp hp ihp frs*
| *Some handler-pc* ⇒ (*None*,*shp*, *hp*, *ihp*, ([*xc*], *loc*, *C*, *sig*, *handler-pc*, *r*)#*frs*)))

Expresses that a value is tagged with an initialized type (only applies to addresses and then only if the heap contains a value for the address)

constdefs
is-init :: *aheap* ⇒ *init-heap* ⇒ *val* ⇒ *bool*
is-init hp ih v ≡
∀ *loc*. *v* = *Addr loc* ⟶ *hp loc* ≠ *None* ⟶ (∃ *t*. *ih loc* = *Init t*)

System exceptions are allocated in all heaps.

constdefs
preallocated :: *aheap* ⇒ *init-heap* ⇒ *bool*
preallocated hp ihp ≡ ∀ *x*. ∃ *fs*. *hp* (*XcptRef x*) = *Some* (*Obj* (*Xcpt x*) *fs*) ∧ *is-init hp ihp* (*Addr* (*XcptRef x*))

lemma *preallocatedD* [*simp*,*dest*]:
preallocated hp ihp ⟹ ∃ *fs*. *hp* (*XcptRef x*) = *Some* (*Obj* (*Xcpt x*) *fs*) ∧ *is-init hp ihp* (*Addr* (*XcptRef x*))
by (*unfold preallocated-def*) *fast*

lemma *preallocatedE* [*elim?*]:
preallocated hp ihp ⟹
(⋀*fs*. *hp* (*XcptRef x*) = *Some* (*Obj* (*Xcpt x*) *fs*) ⟹ *is-init hp ihp* (*Addr* (*XcptRef x*)) ⟹ *P hp ihp*)
⟹ *P hp ihp*
by *fast*

lemma *cname-of-xcp*:
raise-system-xcpt b x = *Some xcp* ⟹ *preallocated hp ihp*
⟹ *cname-of hp xcp* = *Xcpt x*
proof −
assume *raise-system-xcpt b x* = *Some xcp*
hence *xcp* = *Addr* (*XcptRef x*)
by (*simp add*: *raise-system-xcpt-def split*: *split-if-asm*)
moreover
assume *preallocated hp ihp*
then obtain *fs* **where** *hp* (*XcptRef x*) = *Some* (*Obj* (*Xcpt x*) *fs*) **..**
ultimately show *?thesis* **by** *simp*
qed

lemma *preallocated-start*:
preallocated (*start-heap G*) *start-iheap*
apply (*unfold preallocated-def*)
apply (*unfold start-heap-def start-iheap-def*)
apply (*rule allI*)
apply (*case-tac x*)
apply (*auto simp add*: *blank-def is-init-def*)
done

Only program counters that are mentioned in the exception table can be returned by *match-exception-table*:

lemma *match-exception-table-in-et*:

match-exception-table G C pc et = Some pc' ⟹ ∃ e ∈ set et. pc' = fst (snd (snd e))
by (*induct et*) (*auto split*: *split-if-asm*)

end

16 Program Execution in the JVM

theory *JVMExec*
imports *JVMExecInstr JVMExceptions*
begin

consts
exec :: *jvm-prog* × *jvm-state* => *jvm-state option*

— recdef only used for pattern matching
recdef *exec* {}
exec (*G*, *xp*, *shp*, *hp*, *ihp*, []) = *None*

exec (*G*, *None*, *shp*, *hp*, *ihp*, (*stk*,*loc*,*C*,*sig*,*pc*,*z*)#*frs*) =
(*let*
i = *fst*(*snd*(*snd*(*snd*(*snd*(*the*(*method'* (*G*,*C*) *sig*)))))) ! *pc*;
(*xcpt'*, *shp'*, *hp'*, *ihp'*, *frs'*) = *exec-instr i G shp hp ihp stk loc C sig pc z frs*
in Some (*find-handler G xcpt' shp' hp' ihp' frs'*))

exec (*G*, *Some xp*, *hp*, *ihp*, *frs*) = *None*

constdefs
exec-all :: [*jvm-prog*,*jvm-state*,*jvm-state*] => *bool*
(- |- - −*jvm*→ - [*61*,*61*,*61*]*60*)
G |- *s* −*jvm*→ *t* == (*s*,*t*) ∈ {(*s*,*t*). *exec*(*G*,*s*) = *Some t*}^*

syntax (*xsymbols*)
exec-all :: [*jvm-prog*,*jvm-state*,*jvm-state*] => *bool*
(- ⊢ - −*jvm*→ - [*61*,*61*,*61*]*60*)

The start configuration of the JVM: in the start heap, we call a method m of class C in program G. The *this* pointer of the frame is set to *Null* to simulate a static method invokation.

constdefs
start-state :: *jvm-prog* ⇒ *cname* ⇒ *mname* ⇒ *jvm-state*
start-state G C m ≡
let (*C'*,*rT*,*mxs*,*mxl*,*ins*,*et*) = *the* (*method* (*G*,*C*) (*m*,[])) *in*

(*None*,*start-sheap G*, *start-heap G*, *start-iheap*,
[([], *Null # replicate mxl arbitrary*, *C*, (*m*,[]), *0*, (*Null*,*Null*))])

end

17 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
$(h,k) \downarrow k0 = (let\ A = \{\ p\ .\ \ p \in dom\ h\ \&\ fst\ (the\ (h\ p)) <= k0\}$
$in\ (h\ |`\ A,k0))$

constdefs
$fresh :: Location \Rightarrow HeapMap\ \Rightarrow bool$
$fresh\ p\ h \equiv p \notin dom\ h$

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

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

consts
$copy :: [Heap,Region,Location] => Heap \times Location$

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

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

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

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

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 ≡ *fst* (*the* ((*fst h*) *l*))

constdefs *domHeap* :: *Heap* ⇒ *Location set*
domHeap h ≡ *dom* (*fst h*)

constdefs *isNonBasicValue* :: *ArgType* ⇒ *bool*
isNonBasicValue a == (*a* = *NonRecursive*) ∨ (*a* = *Recursive*)

constdefs *isRecursive* :: *ArgType* ⇒ *bool*
isRecursive a == (*a* = *Recursive*)

consts
theLocation :: *Val* ⇒ *Location*
primrec *theLocation* (*Loc l*) = *l*

constdefs
getArgType C ≡ *snd* (*snd* (*the* (*ConstructorTable C*)))

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

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

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

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

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

18 State of the SVM

theory *SVMState*

imports *SafeHeap ../JVMSAFE/BinOP*
begin

18.1 Sizes Table

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

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

types *SizesTable* = *ncell* × *sizeRegions* × *sizeStackS*

18.2 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*

18.3 Code Store and SafeImp program

— Items are the components of environments and closures
datatype *Item* = *ItemConst Val*
| *ItemVar StackOffset*
| *ItemRegSelf*

— The SVM instruction repertory
datatype *SafeInstr* = *DECREGION*
| *POPCONT*
| *PUSHCONT CodeLabel*
| *COPY*
| *REUSE*
| *CALL CodeLabel*
| *PRIMOP PrimOp*
| *MATCH StackOffset* (*CodeLabel list*)

```
  | MATCHD StackOffset (CodeLabel list)
  | MATCHN StackOffset nat nat (CodeLabel list)
  | BUILDENV (Item list)
  | BUILDCLS Constructor (Item list)  Item
  | SLIDE nat nat
```

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

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

A Safe program, when translated into SafeImp, produces four components (1) a map from labels to pairs consisting of a code sequence and a function name. It is given as a list in order to be able to 'traverse' the map; (2) a map from function names to pairs consisting of a label and a list of labels. The 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 × SizesTable
```

18.4 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
```

$primops :: string \rightharpoonup PrimOp$

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

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

primrec
$execOp\ Equal\ b1\ b2 = BoolT\ (the\text{-}IntT(b1) = the\text{-}IntT(b2))$
$execOp\ NotEqual\ b1\ b2 = BoolT\ (the\text{-}IntT(b1) \neq the\text{-}IntT(b2))$
$execOp\ GreaterEqual\ b1\ b2 = BoolT\ (the\text{-}IntT(b1) \geq the\text{-}IntT(b2))$
$execOp\ GreaterThan\ b1\ b2 = BoolT\ (the\text{-}IntT(b1) > the\text{-}IntT(b2))$
$execOp\ LessThan\ b1\ b2 = BoolT\ (the\text{-}IntT(b1) < the\text{-}IntT(b2))$
$execOp\ LessEqual\ b1\ b2 = BoolT\ (the\text{-}IntT(b1) \leq the\text{-}IntT(b2))$

$execOp\ Add\ b1\ b2 = IntT\ (the\text{-}IntT(b1) + the\text{-}IntT(b2))$
$execOp\ Substract\ b1\ b2 = IntT\ (the\text{-}IntT(b1) - the\text{-}IntT(b2))$
$execOp\ Times\ b1\ b2 = IntT\ (the\text{-}IntT(b1) * the\text{-}IntT(b2))$
$execOp\ Divide\ \ b1\ b2 = IntT\ (the\text{-}IntT(b1)\ div\ the\text{-}IntT(b2))$

end

19 Specification Core RTS

theory *RTSCore*
imports *../JVMSAFE/JVMInstructions*
begin

— Run-time system class name

constdefs
$System :: cname$
$System \equiv Cname\ ''java/lang/System''$
$objectC :: cname$
$objectC \equiv Cname\ ''Object''$
$heapC :: cname$
$heapC \equiv Cname\ ''rtsCore/Heap''$

$stackC :: cname$
$stackC \equiv Cname\ ''rtsCore/Stack\text{-}S''$
$dirCellC :: cname$
$dirCellC \equiv Cname\ ''rtsCore/DirectoryCell''$
$consTableC :: cname$
$consTableC \equiv Cname\ ''rtsCore/ConsTable''$
$consDataC :: cname$
$consDataC \equiv Cname\ ''rtsCore/ConsData''$
$cellC :: cname$
$cellC \equiv Cname\ ''rtsCore/Cell''$

— Run-time system method name
$Init :: mname$
$Init \equiv ''<init>''$
$exitM :: mname$
$exitM \equiv ''exit''$
$decregion :: mname$
$decregion \equiv ''decregion''$
$slide :: mname$
$slide \equiv ''slide''$
$pushRegion :: mname$
$pushRegion \equiv ''pushRegion''$
$popRegion :: mname$
$popRegion \equiv ''popRegion''$
$copyCell :: mname$
$copyCell \equiv ''copyCell''$
$copyCellAux :: mname$
$copyCellAux \equiv ''copyCellAux''$
$clone :: mname$
$clone \equiv ''clone''$
$copyRTS :: mname$
$copyRTS \equiv ''copy''$
$reserveCell :: mname$
$reserveCell \equiv ''reserveCell''$
$releaseCell :: mname$
$releaseCell \equiv ''releaseCell''$
$insertCell :: mname$
$insertCell \equiv ''insertCell''$
$makeHeap :: mname$
$makeHeap \equiv ''makeHeap''$
$makeDirectory :: mname$
$makeDirectory \equiv ''makeDirectory''$
$setmaxSize :: mname$
$setmaxSize \equiv ''setmaxSize''$
$makeConsTable :: mname$
$makeConsTable \equiv ''makeConsTable''$

— Run-time system field name

constdefs

izqf :: *vname*
izqf ≡ *VName ''izq''*
derf :: *vname*
derf ≡ *VName ''der''*
idf :: *vname*
idf ≡ *VName ''id''*
freef :: *vname*
freef ≡ *VName ''free''*
tipoArgsf :: *vname*
tipoArgsf ≡ *VName ''tipoArgs''*
Sf :: *vname*
Sf ≡ *VName ''S''*
topf :: *vname*
topf ≡ *VName ''top''*
k0f :: *vname*
k0f ≡ *VName ''k0''*
kf :: *vname*
kf ≡ *VName ''k''*
regionsf :: *vname*
regionsf ≡ *VName ''regions''*
safeDirf :: *vname*
safeDirf ≡ *VName ''safeDir''*
tagLf :: *vname*
tagLf ≡ *VName ''tagL''*
tagGf :: *vname*
tagGf ≡ *VName ''tagG''*
nargsf :: *vname*
nargsf ≡ *VName ''nargs''*
tablef :: *vname*
tablef ≡ *VName ''table''*
arg1 :: *vname*
arg1 ≡ *VName ''arg1''*
arg2 :: *vname*
arg2 ≡ *VName ''arg2''*
arg3 :: *vname*
arg3 ≡ *VName ''arg3''*
arg4 :: *vname*
arg4 ≡ *VName ''arg4''*
arg5 :: *vname*
arg5 ≡ *VName ''arg5''*
arg6 :: *vname*
arg6 ≡ *VName ''arg6''*
arg7 :: *vname*
arg7 ≡ *VName ''arg7''*
arg8 :: *vname*
arg8 ≡ *VName ''arg8''*

Cell Class

```
— Cell Field Declarations
constdefs cellFdecl :: fdecl list
cellFdecl ≡ [(izqf,RefT (ClassT cellC)),
             (derf,RefT (ClassT cellC)),
             (idf,PrimT Integer),
             (tagGf,PrimT Integer),
             (arg1,PrimT Integer),
             (arg2,PrimT Integer),
             (arg3,PrimT Integer),
             (arg4,PrimT Integer),
             (arg5,PrimT Integer),
             (arg6,PrimT Integer),
             (arg7,PrimT Integer),
             (arg8,PrimT Integer)]
```

ConsTable Class

```
— ConsTable Field Declarations
constdefs consTableFdecl :: fdecl list
 consTableFdecl ≡ [(tablef, RefT (ArrayT (RefT (ClassT consDataC))))]
```

```
constdefs
 EndFor1 :: int
 EndFor1 ≡ 12
 bucle1 :: int
 bucle1 ≡ − 14
constdefs instMakeConsTable :: instr list
instMakeConsTable ≡
             [Load 0,
              ArrNew (RefT (ClassT consDataC)),
              Putstatic tablef consTableC, (∗ table = new ConsData[maxCons] ∗)
              LitPush (Intg 0),
              Store 1,                    (∗ i ← 0 ∗)
              Load 1,
              Getstatic tablef consTableC,
              ArrLength,
              Ifcmp GreaterEqual EndFor1, (∗ i < table.length ? ∗)
              Getstatic tablef consTableC,
              Load 1,
              New consDataC,
              Dup,
              Invoke-special consDataC Init [],
              ArrStore,
              Load 1,
              LitPush (Intg 1),
              BinOp Add,
```

```
        Store 1,                    (* i ← i + 1 *)
        Goto bucle1,
        Return-Void
            ]
```

ConsData Class

— ConsData Field Declarations

```
constdefs consDataFdecl :: fdecl list
 consDataFdecl ≡ [(tagLf, PrimT Integer),
                  (nargsf, PrimT Integer),
                  (tipoArgsf, RefT (ArrayT (PrimT Integer)))]


constdefs instInitConsData :: instr list
 instInitConsData ≡ [Load 0,                    (* Load this *)
                     LitPush (Intg 8),
                     ArrNew (PrimT Integer), (* tipoArgs ← new int[8] *)
                     Putfield tipoArgsf consDataC,
                     Return-Void]
```

Stack$_S$*Class*

— Stack$_S$*FieldDeclarations*

```
constdefs stackSFdecl :: fdecl list
 stackSFdecl ≡ [(Sf, RefT (ArrayT (PrimT Integer))),(topf, PrimT Integer)]


constdefs instSetMaxSize :: instr list
 instSetMaxSize ≡ [Load 0,
                   ArrNew (PrimT Integer),
                   Putstatic Sf stackC,      (* S ← new int[maxSize]  *)
                   LitPush (Intg −1),
                   Putstatic topf stackC,    (* top ← −1 *)
                   Return-Void
                   ]

 constdefs
 EndFor2 :: int
 EndFor2 ≡ 16
 Bucle2 :: int
 Bucle2 ≡ −17
constdefs instSlide :: instr list
 instSlide ≡ [Getstatic topf stackC,
              Load 0,                    (* Load local0 = m *)
              BinOp Substract,
              Load 1,                    (* Load local1 = n *)
              BinOp Substract,
              LitPush (Intg 1),
```

```
BinOp Add,
Store 2,                    (* local2← top − m − n + 1 *)
Getstatic topf stackC,
Load 0,
BinOp Substract,
LitPush (Intg 1),
BinOp Add,
Store 3,                    (* local3 ←top − m + 1 *)
Load 3,                     (* Label Bucle2 *)
Getstatic topf stackC,
Ifcmp GreaterThan EndFor2,      (* local3 <= top ?      *)
Getstatic Sf stackC,
Load 2,
Getstatic Sf stackC,
Load 3,
ArrLoad,
ArrStore,                   (* S[local2] ← S[local3] *)
Load 2,
LitPush (Intg 1),
BinOp Add,
Store 2,                    (* local2 ← local2 + 1   *)
Load 3,
LitPush (Intg 1),
BinOp Add,
Store 3,                    (* local3 ← local3 + 1   *)
Goto Bucle2,
Getstatic topf stackC,          (* Label EndFor2          *)
Load 1,
BinOp Substract,
Putstatic topf stackC,          (* top ← top − n          *)
Return-Void
]
```

DirectoryCell Class

— DirectoryCell Field Declarations

```
constdefs dirCFdecl :: fdecl list
 dirCFdecl ≡ [(safeDirf, RefT (ArrayT (RefT (ClassT cellC)))),
              (freef, RefT (ClassT cellC))]
```

```
constdefs
 EndFor3 :: int
EndFor3 ≡ 77
 If1 :: int
 If1 ≡ 25
 If2 :: int
 If2 ≡ 19
```

```
  If3 :: int
  If3 ≡ 32
  EndIf2 :: int
  EndIf2 ≡ 38
  Bucle3 :: int
 Bucle3 ≡ −82

— Instructions List
constdefs instMakeDirectory :: instr list
 instMakeDirectory ≡ [
            Load 0,
            ArrNew (Class cellC),
            Putstatic safeDirf dirCellC, (∗ safeDir ← new Cell[maxCell] ∗)
            LitPush (Intg 0),
            Store 1,                    (∗ i ← 0 ∗)
            Load 1,
            Getstatic safeDirf dirCellC,
            ArrLength,
            Ifcmp GreaterEqual EndFor3, (∗ i < safeDir.length ? ∗)
            Getstatic safeDirf dirCellC,
            Load 1,
            New cellC,
            Dup,
            Invoke-special cellC Init [],
            ArrStore,                   (∗ safeDir[i] ← new Cell() ∗)
            Getstatic safeDirf dirCellC,
            Load 1,
            ArrLoad,
            Load 1,
            Putfield idf cellC,         (∗ safeDir[i].id ← i ∗)
            Load 1,
            LitPush (Intg 0),
            Ifcmp LessEqual If1,        (∗ i > 0 ? ∗)
            Load 1,
            Getstatic safeDirf dirCellC,
            ArrLength,
            LitPush (Intg 1),
            BinOp Substract,
            Ifcmp GreaterThan If2,      (∗ i < safeDir.length − 1 ∗)
            Getstatic safeDirf dirCellC,
            Load 1,
            ArrLoad,
            Getstatic safeDirf dirCellC,
            Load 1,
            LitPush (Intg 1),
            BinOp Substract,
            ArrLoad,
            Putfield izqf cellC,        (∗ safeDir[i].izq ← safeDir[i − 1] ∗)
            Getstatic safeDirf dirCellC,
```

```
Load 1,
ArrLoad,
Getfield izqf cellC,
Getstatic safeDirf dirCellC,
Load 1,
ArrLoad,
Putfield derf cellC,        (* safeDir[i].izq.der ← safeDir[i] *)
Goto EndIf2,
Load 1,
Getstatic safeDirf dirCellC,
ArrLength,
LitPush (Intg 1),
BinOp Substract,
Ifcmp NotEqual If3,         (* i = safeDir.length − 1 ? *)
Getstatic safeDirf dirCellC,
Load 1,
ArrLoad,
Getstatic safeDirf dirCellC,
Load 1,
LitPush (Intg 1),
BinOp Substract,
ArrLoad,
Putfield izqf cellC,        (* safeDir[i].izq ← safeDir[i − 1] *)
Getstatic safeDirf dirCellC,
Load 1,
ArrLoad,
Getstatic izqf cellC,
Getstatic safeDirf dirCellC,
Load 1,
ArrLoad,
Putfield derf cellC,        (* safeDir[i].izq.der ← safeDir[i]  *)
Getstatic safeDirf dirCellC,
Load 1,
ArrLoad,
Getstatic safeDirf dirCellC,
LitPush (Intg 0),
ArrLoad,
Putfield derf cellC,        (* safeDir[i].der ← safeDir[0] *)
Getstatic safeDirf dirCellC,
LitPush (Intg 0),
ArrLoad,
Getstatic safeDirf dirCellC,
Load 1,
ArrLoad,
Putfield izqf cellC,        (* safeDir[0].izq ← safeDir[i] *)
Load 1,
LitPush (Intg 1),
BinOp Add,
Store 1,                    (* i ← i + 1 *)
```

```
        Goto Bucle3,
        Getstatic safeDirf dirCellC,
        LitPush (Intg 0),
        ArrLoad,
        Putstatic freef cellC,      (* free ← safeDir[0] *)
        Return-Void
         ]

constdefs
 If4 :: int
 If4 ≡ 18
 EndIf4 :: int
 EndIf4 ≡ 3

constdefs instReserveCell :: instr list
 instReserveCell ≡ [
     Getstatic freef cellC,
     Getfield derf cellC,
     Store 0,                (* p ← free.der *)
     Load 0,
     Getstatic freef cellC,
     Ifcmpeq If4,             (* p ≠ free   ? *)
     Load 0,
     Getfield derf cellC,
     Store 1,                (* p2 ← p.der  *)
     Getstatic freef cellC,
     Load 1,
     Putfield derf cellC,     (* free.der ← p2 *)
     Load 1,
     Load 0,
     Getfield izqf cellC,
     Putfield izqf cellC,     (* p2.izq ← p.izq *)
     Load 0,
     Load 0,
     LitPush (Null),
     Dup-x1,
     Putfield derf cellC,    (* p.der ← null    *)
     Putfield izqf cellC,    (* p.izq ← null    *)
     Goto EndIf4,
     LitPush(Intg 1),
     Invoke-static System exitM [PrimT Integer],  (* Abort program *)
     Load 0,
     Getfield idf cellC,
     Return                  (* return p.id      *)
     ]

constdefs instReleaseCell :: instr list
 instReleaseCell ≡ [
       Getstatic safeDirf dirCellC,
```

```
        Load 0,
        ArrLoad,
        Store 1,            (* cell ← safeDir[index] *)
        Load 1,
        Getfield izqf cellC,
        Store 2,            (* temp1 ← cell.izq      *)
        Load 1,
        Getfield derf cellC,
        Store 3,            (* temp2 ← cell.der      *)
        Load 2,
        Load 1,
        Getfield derf cellC,
        Putfield derf cellC, (* temp1.der ← cell.der  *)
        Load 3,
        Load 1,
        Getfield izqf cellC,
        Putfield izqf cellC, (* temp2.izq ← cell.izq  *)
        Load 1,
        Getstatic freef dirCellC,
        Getfield derf cellC,
        Putfield derf cellC, (* cell.der ← free.der   *)
        Getstatic freef dirCellC,
        Getfield derf cellC,
        Load 1,
        Putfield izqf cellC, (* free.der.izq ← cell   *)
        Getstatic freef cellC,
        Load 1,
        Putfield derf cellC, (* free.der ← cell       *)
        Load 1,
        Getstatic freef dirCellC,
        Putfield izqf cellC, (* cell.izq ← free       *)
        Return-Void
        ]
```

— Heap Field Declarations

```
constdefs heapFdecl :: fdecl list
  heapFdecl ≡ [(regionsf, RefT (ArrayT (RefT (ClassT cellC)))),
               (kf, PrimT Integer),
              (k0f,PrimT Integer)]


constdefs instMakeHeap :: instr list
 instMakeHeap ≡ [
       Load 0,
       ArrNew (Class cellC),
       Putstatic regionsf heapC,    (* regions ← new Cell[maxRegions] *)
       LitPush (Intg −1),
       Putstatic kf heapC,          (* k ← −1 *)
```

```
      LitPush (Intg −1),
      Putstatic k0f heapC,          (∗ k0 ← −1 ∗)
      Return-Void
        ]


constdefs
  bucle5 :: int
  bucle5 ≡ 3
 endbucle5 :: int
  endbucle5 ≡ −4

constdefs instDecregion :: instr list
 instDecregion ≡ [
       Getstatic kf heapC,
       Getstatic k0f heapC,
       Ifcmp LessEqual bucle5,                (∗ k > k0 ?  ∗)
       Invoke-static heapC popRegion [],   (∗ call PopRegion() method ∗)
       Goto endbucle5,
       Return-Void
          ]



constdefs

labelEnd :: int
 labelEnd ≡ 4

constdefs instPushRegion :: instr list
 instPushRegion ≡ [
      Getstatic kf heapC,
      LitPush (Intg 1),
      BinOp Add,
      Putstatic kf heapC,  (∗ k ← k + 1 ∗)
      Invoke-static dirCellC reserveCell [],
      Store 0,              (∗ cell ← reserveCell() ∗)
      Getstatic regionsf heapC,
      Getstatic kf heapC,
      Getstatic safeDirf dirCellC,
      Load 0,
      ArrLoad,
      ArrStore,             (∗ regions[k] ← safeDir[cell] ∗)
      Getstatic regionsf heapC,
      Getstatic kf heapC,
      ArrLoad,
      Getstatic regionsf heapC,
      Getstatic kf heapC,
      ArrLoad,
```

```
    Getstatic regionsf heapC,
    Getstatic kf heapC,
    ArrLoad,
    Dup-x1,
    Putfield izqf cellC, (* regions[k].izq ← regions[k] *)
    Putfield derf cellC, (* regions[k].der <− regions[k].izq *)
    Goto labelEnd,
    Store 0,            (* Exception e *)
    LitPush (Intg 1),
    Invoke-static System exitM [PrimT Integer],  (* Abort program with error *)
    Return-Void
    ]

constdefs instPopRegion :: instr list
 instPopRegion ≡ [
    Getstatic freef dirCellC,
    Getfield izqf cellC,
    Getstatic regionsf heapC,
    Getstatic kf heapC,
    ArrLoad,
    Putfield derf cellC,     (* free.izq.der ← regions[k] *)
    Getstatic freef dirCellC,
    Getfield izqf cellC,
    Store 0,                 (* c1 ← free.izq   *)
    Getstatic regionsf heapC,
    Getstatic kf heapC,
    ArrLoad,
    Getfield izqf cellC,
    Store 1,                 (* c2 ← regions[k].izq *)
    Getstatic freef cellC,
    Load 1,
    Putfield izqf cellC,     (* free.izq ← c2       *)
    Load 1,
    Getstatic freef cellC,
    Putfield derf cellC,     (* c2.der ← free       *)
    Getstatic regionsf heapC,
    Getstatic kf heapC,
    ArrLoad,
    Load 0,
    Putfield izqf cellC,     (* regions[k].izq ← c1 *)
    Getstatic regionsf heapC,
    Getstatic kf heapC,
    LitPush (Null),
    ArrStore,                (* regions[k] ← NULL   *)
    Getstatic kf heapC,
    LitPush(Intg 1),
    BinOp Substract,
    Putstatic kf heapC,      (* k ← k − 1           *)
    Return-Void
```

```
    ]

constdefs instInsertCell :: instr list
 instInsertCell ≡ [
      Getstatic safeDirf dirCellC,
      Load 1,
      ArrLoad,
      Store 2,                (∗ cell ← safeDir[p] ∗)
      Load 2,
      Getstatic regionsf heapC,
      Load 0,
      Getfield izqf cellC,
      Putfield izqf cellC,    (∗ cell.izq ← regions[r].izq ∗)
      Getstatic regionsf heapC,
      Load 0,
      ArrLoad,
      Getfield izqf cellC,
      Load 2,
      Putfield derf cellC,    (∗ regions[r].izq.der ← cell ∗)
      Getstatic regionsf heapC,
      Load 0,
      ArrLoad,
      Load 2,
      Putfield izqf cellC,    (∗ regions[r].izq ← cell     ∗)
      Load 2,
      Getstatic regionsf heapC,
      Load 0,
      ArrLoad,
      Putfield derf cellC,    (∗ cell.der ← regions[r]     ∗)
      Return-Void
 ]

constdefs instCopyCell :: instr list
 instCopyCell ≡ [
      Invoke-static dirCellC reserveCell [],
      Store 1,                (∗ freshCell ← reserveCell()   ∗)
      Getstatic safeDirf dirCellC,
      Load 1,
      ArrLoad,
      Store 2,                (∗ target ← safeDir[freshCell] ∗)
      Getstatic safeDirf dirCellC,
      Load 0,
      ArrLoad,
      Store 3,                (∗ source ← safeDir[p]         ∗)
      Load 3,
      Load 2,
      Invoke-static dirCellC copyCellAux [Class cellC,
                                          Class cellC],   (∗ copyCellAux(source,target) ∗)
      Load 0,
```

```
    Invoke-static dirCellC releaseCell [PrimT Integer], (* releaseCell(p) *)
    Load 2,
    Getfield derf cellC,
    Load 2,
    Putfield izqf cellC,       (* target.der.izq ← target *)
    Load 2,
    Getfield izqf cellC,
    Load 2,
    Putfield derf cellC,       (* target.izq.der ← target *)
    Load 1,
    Return                  (* devuelve freshCell       *)
  ]

constdefs instClone :: instr list
 instClone ≡ [
    Invoke-static dirCellC reserveCell [],
    Store 2,                (*  freshCell ← reserveCell()  *)
    Getstatic safeDirf dirCellC,
    Load 2,
    ArrLoad,
    Store 3,                (* target ← safeDir[freshCell] *)
    Getstatic safeDirf dirCellC,
    Load 1,
    ArrLoad,
    Store 4,                (* source ← safeDir[p]         *)
    Load 4,
    Load 3,
    Invoke-static heapC copyCellAux [Class cellC,
                                     Class cellC],  (* copyCellAux(source,target) *)
    Load 0,
    Load 2,
    Invoke-static heapC insertCell [PrimT Integer,
                                    PrimT Integer], (* insertCell(reg,freshCell)  *)
    Load 2,
    Return                  (* devuelve entero freshCell *)
]

constdefs instCopyCellAux :: instr list
 instCopyCellAux ≡ [
    Load 1,
    Load 0,
    Getfield derf cellC,
    Putfield derf cellC,       (* c2.der ← c1.der *)
    Load 1,
    Load 0,
    Getfield izqf cellC,
    Putfield izqf cellC,       (* c2.izq ← c1.izq *)
    Load 1,
    Load 0,
```

```
 Getfield tagGf cellC,
 Putfield tagGf cellC,      (* c2.tagG ← c1.tagG *)
 Load 1,
 Load 0,
 Getfield arg1 cellC,
 Putfield arg1 cellC,       (* c2.arg1 ← c1.arg1 *)
 Load 1,
 Load 0,
 Getfield arg2 cellC,
 Putfield arg2 cellC,       (* c2.arg2 ← c1.arg2 *)
 Load 1,
 Load 0,
 Getfield arg3 cellC,
 Putfield arg3 cellC,       (* c2.arg3 ← c1.arg3 *)
 Load 1,
 Load 0,
 Getfield arg4 cellC,
 Putfield arg4 cellC,       (* c2.arg4 ← c1.arg4 *)
 Load 1,
 Load 0,
 Getfield arg5 cellC,
 Putfield arg5 cellC,       (* c2.arg5 ← c1.arg5 *)
 Load 1,
 Load 0,
 Getfield arg6 cellC,
 Putfield arg6 cellC,       (* c2.arg6 ← c1.arg6 *)
 Load 1,
 Load 0,
 Getfield arg7 cellC,
 Putfield arg7 cellC,       (* c2.arg7 ← c1.arg7 *)
 Load 1,
 Load 0,
 Getfield arg8 cellC,
 Putfield arg8 cellC        (* c2.arg8 ← c1.arg8 *)
]
```

constdefs

```
 endBucle6 :: int
endBucle6 ≡ 72
 If5 :: int
 If5 ≡ 62
lab1 :: int
lab1 ≡ 1
endlab1 :: int
endlab1 ≡ 49
lab2 :: int
lab2 ≡ lab1 + 7
```

```
  endlab2 :: int
  endlab2 ≡ endlab1 − 7
  lab3 :: int
  lab3 ≡ lab2 + 7
  endlab3 :: int
  endlab3 ≡ endlab2 − 7
  lab4 :: int
  lab4 ≡ lab3 + 7
  endlab4 :: int
  endlab4 ≡ endlab3 − 7
  lab5 :: int
  lab5 ≡ lab4 + 7
  endlab5 :: int
  endlab5 ≡ endlab4 − 7
  lab6 :: int
  lab6 ≡ lab5 + 7
  endlab6 :: int
  endlab6 ≡ endlab5 − 7
  lab7 :: int
  lab7 ≡ lab6 + 7
  endlab7 :: int
  endlab7 ≡ endlab6 − 7
  lab8 :: int
  lab8 ≡ lab7 + 7
  ldef :: int
  ldef ≡ lab8 + 6
  bucle6 :: int
  bucle6 ≡ −77
 constdefs instCopy :: instr list
instCopy ≡ [
  Load 1,
  Load 0,
  Invoke-static heapC clone [PrimT Integer,
                             PrimT Integer],
  Store 2,              (∗ target ← clone (j,b) ∗)
  Getstatic safeDirf dirCellC,
  Load 0,
  ArrLoad,
  Store 3,              (∗ source ← safeDir[b]  ∗)
  Getstatic tablef consDataC,
  Load 3,
  Getfield tagGf cellC,
  ArrLoad,
  Store 4,              (∗ info ← table[source.tagG] ∗)
  LitPush (Intg 0),
  Store 5,              (∗ i ← 0 ∗)
  Load 5,
  Load 4,
  Getfield nargsf consDataC,
```

```
Ifcmp GreaterEqual  endBucle6,        (* i < info.nargs ?          *)
Load 4,
Getfield tipoArgsf consDataC,
Load 5,
ArrLoad,                (* Integer Array  *)
Store 6,               (* tipo ← info.tipoArgs[i]   *)
Load 6,
LitPush (Intg 4),
Ifcmp NotEqual If5,
Getstatic safeDirf dirCellC,
Load 2,
ArrLoad,
Store 7,               (* targ ← safeDir[target]     *)
Load 5,                (* Load local5 = i    *)
Tableswitch 0 7 [lab1,
                 lab2,
                 lab3,
                 lab4,
                 lab5,
                 lab6,
                 lab7,
                 lab8,
                 ldef],

 Load 7,              (* label 1  *)
 Load 3,
 Getfield arg1 cellC,
 Load 1,
 Invoke-static heapC copyRTS [PrimT Integer,
                              PrimT Integer],
 Putfield arg1 cellC, (* targ.arg1 ← copy(source.arg1,j) *)
 Goto endlab1,        (* end label 1 *)

 Load 7,              (* label 2  *)
 Load 3,
 Getfield arg2 cellC,
 Load 1,
 Invoke-static heapC copyRTS [PrimT Integer,
                              PrimT Integer],
 Putfield arg2 cellC, (* targ.arg2 ← copy(source.arg2,j) *)
 Goto endlab2,        (* end label 2 *)



 Load 7,              (* label 3  *)
 Load 3,
 Getfield arg3 cellC,
 Load 1,
 Invoke-static heapC copyRTS [PrimT Integer,
```

```
                    PrimT Integer],
Putfield arg3 cellC, (* targ.arg3 ← copy(source.arg3,j) *)
Goto endlab3,        (* end label 3 *)


Load 7,              (* label 4  *)
Load 3,
Getfield arg4 cellC,
Load 1,
Invoke-static heapC copyRTS [PrimT Integer,
                    PrimT Integer],
Putfield arg4 cellC, (* targ.arg4 ← copy(source.arg4,j) *)
Goto endlab4,        (* end label 4 *)

Load 7,              (* label 5  *)
Load 3,
Getfield arg5 cellC,
Load 1,
Invoke-static heapC copyRTS [PrimT Integer,
                    PrimT Integer],
Putfield arg5 cellC, (* targ.arg5 ← copy(source.arg5,j) *)
Goto endlab5,        (* end label 5 *)


Load 7,              (* label 6  *)
Load 3,
Getfield arg6 cellC,
Load 1,
Invoke-static heapC copyRTS [PrimT Integer,
                    PrimT Integer],
Putfield arg6 cellC, (* targ.arg6 ← copy(source.arg6,j) *)
Goto endlab6,        (* end label 6 *)

Load 7,              (* label 7  *)
Load 3,
Getfield arg7 cellC,
Load 1,
Invoke-static heapC copyRTS [PrimT Integer,
                    PrimT Integer],
Putfield arg7 cellC, (* targ.arg1 ← copy(source.arg1,j) *)
Goto endlab7,        (* end label 7 *)

Load 7,              (* label 8  *)
Load 3,
Getfield arg8 cellC,
Load 1,
Invoke-static heapC copyRTS [PrimT Integer,
                    PrimT Integer],
```

```
    Putfield arg8 cellC, (* targ.arg8 ← copy(source.arg8,j) *)
    Load 5,
    LitPush (Intg 1),
    BinOp Add,
    Store 5,            (* i ← i + 1 *)
    Goto bucle6,
    Load 2,
    Return              (* return target *)
]


— Program RTS with all Class declarations
constdefs  Prog-RTS :: jvm-prog

Prog-RTS ≡
(let
  classCell = (cellC,objectC,cellFdecl,[]);
 consTableMethod = ((makeConsTable,[PrimT Integer]), PrimT Void,(4,2,instMakeConsTable,[]));
  classConsTable = (consTableC,objectC,consTableFdecl,[consTableMethod]);
  initConsDataMethod = ( (Init, []), PrimT Void, (2,1,instInitConsData,[]));
  classConsData =(consDataC,objectC,consDataFdecl,[initConsDataMethod]);
 setMaxSizeM = ((setmaxSize,[PrimT Integer]), PrimT Void, (1,1,instSetMaxSize,[]));
  slideM = ((slide,[PrimT Integer,PrimT Integer]), PrimT Void, (4,4,instSlide,[]));
  classStackS = (stackC, objectC,stackSFdecl,[setMaxSizeM,slideM]);
 makeDirM = ((makeDirectory, [PrimT Integer]), PrimT Void, (4,2,instMakeDirectory,[]));
  reserM = ((reserveCell,[]), PrimT Integer, (4,2,instReserveCell ,[]));
  releaM = ((releaseCell,[PrimT Integer]), PrimT Void, (2,4,instReleaseCell ,[]));
  classDir = (dirCellC, objectC,dirCFdecl,[makeDirM,reserM,releaM]);
 makeHeapM = ((makeHeap,[PrimT Integer]),PrimT Void, (1,1,instMakeHeap,[]));
  decregionM = ((decregion,[]),PrimT Void,(2,0,instDecregion,[]));
  pushRegionM = ((pushRegion,[]),PrimT Void,(4,1,instPushRegion,[(0,23,26,
                Xcpt ArrayIndexOutOfBounds)]));
  popRegionM = ((popRegion,[]),PrimT Void,(3,2,instPopRegion,[]));
 insertCellM = ((insertCell,[PrimT Integer,PrimT Integer]), PrimT Void,(3,3,instInsertCell,[]));
  copyCellM = ((copyCell,[PrimT Integer]),PrimT Integer,(2,4,instCopyCell,[]));
 cloneM = ((clone,[PrimT Integer,PrimT Integer]),PrimT Integer,(2,5,instClone,[]));
  copyCellAuxM = ((copyCellAux,[RefT (ClassT cellC),RefT (ClassT cellC)]),
                PrimT Void,(2,2,instCopyCellAux,[]));
 copyM = ((copyRTS,[PrimT Integer,PrimT Integer]),PrimT Integer,(3,8,instCopy,[]));

 classHeap = (heapC, objectC, heapFdecl,[makeHeapM,decregionM,pushRegionM,popRegionM,
             insertCellM,copyCellM,cloneM,copyCellAuxM,copyM])
in [classCell,classConsTable,classConsData,classStackS,classDir,classHeap])

end
```

20 Useful functions and theorems from the Haskell Library or Prelude

theory *HaskellLib*
imports *Main*
begin

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

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

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

Some lemmas about mapAccumL

lemma *mapAccumL-non-empty*:
```
  ⟦ (s'',ys) = mapAccumL f s xs;
   xs = x#xx
   ⟧ ⟹ (∃ s' y ys'.
            (s',y) = f s x
            ∧ ys = y # ys')
```
apply *clarify*
apply (*unfold mapAccumL.simps*)
apply (*rule-tac x=fst* (*f s x*) **in** *exI*)
apply (*rule-tac x=snd* (*f s x*) **in** *exI*)
apply (*rule-tac x=snd* (*mapAccumL f* (*fst* (*f s x*)) *xx*) **in** *exI*)
apply (*rule conjI*)
apply *simp*
apply (*case-tac f s x*,*simp*)
by (*case-tac mapAccumL f a xx*,*simp*)

lemma *mapAccumL-non-empty2*:
```
  ⟦ (s'',ys) = mapAccumL f s xs;
   xs = x#xx
   ⟧ ⟹ (∃ s' y ys'.
            (s',y) = f s x
            ∧ (s'',ys') = mapAccumL f s' xx
            ∧ ys = y # ys')
```
apply *clarify*
apply (*unfold mapAccumL.simps*)
apply (*rule-tac x=fst* (*f s x*) **in** *exI*)
apply (*rule-tac x=snd* (*f s x*) **in** *exI*)

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

axioms *mapAccumL-non-empty3*:
⟦ (*s''*,*ys*) = *mapAccumL f s xs*;
0<*length xs*
⟧ ⟹ (∃ *s' y ys'*.
(*s'*,*y*) = *f s* (*xs*!*0*)
∧ (*s''*,*ys'*) = *mapAccumL f s'* (*tl xs*))

axioms *mapAccumL-two-elements*:
⟦ (*s3*,*ys*) = *mapAccumL f s xs*;
xs = *x1*#*x2*#*xx*
⟧ ⟹ (∃ *s1 s2 y1 y2 ys3*.
(*s1*,*y1*) = *f s x1*
∧ (*s2*,*y2*) = *f s1 x2*
∧ (*s3*,*ys3*) = *mapAccumL f s2 xx*
∧ *ys* = *y1*#*y2*#*ys3*)

axioms *mapAccumL-split*:
⟦ (*s2*,*ys*) = *mapAccumL f s xs*;
xs1 @ *xs2*=*xs*
⟧ ⟹ (∃ *s1 ys1 ys2* .
(*s1*,*ys1*) = *mapAccumL f s xs1*
∧ (*s2*,*ys2*) = *mapAccumL f s1 xs2*
∧ *ys* = *ys1* @ *ys2*)

axioms *mapAccumL-one-more*:
⟦ (*s1*,*ys*) = *mapAccumL f s xs*;
(*s2*,*y*) = *f s1 x*
⟧ ⟹ (*s2*,*ys*@[*y*]) = *mapAccumL f s* (*xs*@[*x*])

Some integer arithmetic lemmas

lemma *sum-nat*:
⟦ (*x1*::*nat*)=*x2*;(*y1*::*nat*)=*y2*⟧ ⟹ *x1*+*y1*=*x2*+*y2*
apply *arith*
done

axioms *sum-substract*:
(*x*::*nat*)−*y*+(*z*−*x*)=*z*−*y*

axioms *additions1*:
$[\![i< m; Suc\ m + n \leq l]\!] \Longrightarrow$
$m - i < nat\ (int\ l - 1) - n + 1 - (nat\ (int\ l - 1) - Suc\ m - n + 1)$
axioms *additions2*:
$[\![i< m; Suc\ m + n \leq l]\!] \Longrightarrow$
$nat\ (int\ l - 1) - m + (m - Suc\ i) = nat\ (int\ l - 1) - Suc\ m + (m - i)$
axioms *additions3*:
$[\![i< m; Suc\ m + n \leq l]\!] \Longrightarrow$
$nat\ (int\ l - 1) - Suc\ (m + n) + (m - i) = nat\ (int\ l - 1) - (m + n) + (m - Suc\ i)$
axioms *additions4*:
$[\![Suc\ m + n \leq l]\!] \Longrightarrow$
$nat\ (int\ l - 1) - m = Suc\ (nat\ (int\ l - 1) - Suc\ m)$
axioms *additions5*:
$[\![Suc\ m + n \leq l]\!] \Longrightarrow$
$Suc\ (nat\ (int\ l - 1) - Suc\ (m + n)) = nat\ (int\ l - 1 - int\ n - int\ m)$
axioms *additions6*:
$[\![Suc\ m + n \leq l]\!] \Longrightarrow$
$n + (nat\ (int\ l - 1) - Suc\ (m + n)) < nat\ (int\ l - 1)$

Some lemmas about lists

lemma *list-non-empty*:
$0<length\ xs \Longrightarrow (\exists\ y\ ys\ .\ xs = y\ \#\ ys)$
apply *auto*
apply (*insert neq-Nil-conv* [*of xs*])
by *simp*

axioms *drop-nth*:
$n < length\ xs \Longrightarrow (\exists\ y\ ys\ .\ drop\ n\ xs = y\#ys \wedge xs!n = y)$

axioms *drop-nth3*:
$n < length\ xs \Longrightarrow drop\ n\ xs = (xs!n)\ \#\ drop\ (Suc\ n)\ xs$

axioms *drop-take-Suc*:
$xs=(take\ n\ xs)@(z\#zs) \Longrightarrow drop\ (Suc\ n)\ xs=zs$

axioms *drop-nth2*:
$[\![n<length\ xs; drop\ n\ xs = ys]\!]$
$\Longrightarrow ys = xs!n\ \#\ tl\ ys$

axioms *drop-append2*:
$[\![drop\ n\ xs = zs1\ @\ ys1\ @\ ys2\ @\ zs2\ @\ rest;$
$drop\ (m-n)\ (zs1\ @\ ys1\ @\ ys2\ @\ zs2) = ys1\ @\ rest']\!] \Longrightarrow$
$drop\ (m+length\ ys1-n)\ (zs1\ @\ ys1\ @\ ys2\ @\ zs2) = ys2\ @\ zs2$

axioms *drop-append3*:

⟦ *drop n xs = xs1 @ rest*;
drop (m−n) xs1 = ys1 @ ys2
⟧ ⟹
drop m xs = ys1 @ ys2 @ rest

lemma *nth-via-drop-append*: *drop n xs = (y#ys)@zs ==> xs!n = y*
apply (*induct xs arbitrary*: *n*, *simp*)
by(*simp add*:*drop-Cons nth-Cons split*:*nat.splits*)

lemma *drop-Suc-append*:
drop n xs = (y#ys)@zs ⟹ drop (Suc n) xs = ys@zs
apply (*induct xs arbitrary*: *n*,*simp*)
apply (*simp add*:*drop-Cons*)
by (*simp split*:*nat.splits*)

lemma *nth-via-drop-append-2*: *drop n xs = ((y # ys) @ ws @ zs) @ ms ==> xs!n = y*
apply (*induct xs arbitrary*: *n*, *simp*)
by(*simp add*:*drop-Cons nth-Cons split*:*nat.splits*)

lemma *drop-Suc-append-2*:
drop n xs = ((y # ys) @ ws @ zs) @ ms ⟹ drop (Suc n) xs = ys @ ws @ zs @ ms
apply (*induct xs arbitrary*: *n*,*simp*)
apply (*simp add*:*drop-Cons*)
by (*simp split*:*nat.splits*)

axioms *drop-append-length*:
drop n xs = [] @ ys @ zs @ ms ⟹ drop (n + length ys) xs = zs @ ms

axioms *take-length*:
n ≤ length xs ⟹ n = length (take n xs)

axioms *take-append2*:
n<length xs ⟹ x # take n xs = take n (x # xs) @ [(x # xs)!n]

axioms *take-append3*:
Suc n ≤ length xs ⟹ take (Suc n) xs = take n xs @ [xs!n]

axioms *concat1*:
xs @ y # ys = (xs @ [y]) @ ys

axioms *concat2*:
xs1 = xs2 ⟹ xs1 @ ys = xs2 @ ys

axioms *upt-length*:
n ≤ m ⟹ length [n..<m]=m−n

Some lemmas about finite maps

axioms *map-of-distinct*:
⟦ *distinct* (*map fst xys*);
l < *length xys*;
(*x*,*y*) = *xys* ! *l*
⟧ ⟹ *map-of xys x* = *Some y*

axioms *map-of-distinct2*:
map-of xys x = *Some y*
⟹ (∃ *l* . *l* < *length xys* ∧ (*x*,*y*) = *xys* ! *l*)

axioms *map-upds-nth*:
i < *m*−*n* ⟹ (*A*([*n*..<*m*] [↦] *xs*)) (*n*+*i*) = *Some* (*xs* ! *i*)

— The unzip3 function of Haskell library

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

axioms *unzip3-length*:
unzip3 xs = (*ys1*,*ys2*,*ys3*) ⟹ *length ys1* = *length ys2*

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* = []

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

— The Haskell sum type Either

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

— insertion sort for list of strings

constdefs
leString :: *string* => *string* => *bool*

leString s1 s2 == True

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

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

fun *subList :: ′a list ⇒ ′a list ⇒ bool*
where
subList xs ys = (∃ *hs ts. ys = hs @ xs @ ts*)

end

21 Translation from SVM to JVM

theory *SVM2JVM*
imports *../JVMSAFE/JVMInstructions SVMState RTSCore HaskellLib*
begin

types *pc = nat*
codeMap = PC ⇀ pc
contMap = CodeLabel ⇀ nat
consMap = Constructor ⇀ nat

21.1 Initialisation code

Initialise cell creation, region stack, and machine stack. The corresponding initialisation method of each class is called.

constdefs *initSizeTable :: SizesTable ⇒ bytecode*
initSizeTable st ≡ (
case st of (*c*,*r*,*s*) ⇒
[*LitPush* (*Intg* (*int c*)),
Invoke-static dirCellC makeDirectory [*PrimT Integer*],
LitPush (*Intg* (*int r*)),
Invoke-static heapC makeHeap [*PrimT Integer*],
LitPush (*Intg* (*int s*)),
Invoke-static stackC setmaxSize [*PrimT Integer*]])

Initialise the runtime constructor table.

```
constdefs auxOneArg :: (ArgType × nat) ⇒ bytecode
auxOneArg arT ≡
      (let j      = snd arT;
           argTyp = (case (fst arT) of
                       IntArg ⇒ 1
                     | BoolArg ⇒ 2
                     | NonRecursive ⇒ 3
                     | Recursive ⇒ 4)
       in [Dup,
           LitPush (Intg (int j)),
           LitPush (Intg argTyp),
           ArrStore])


constdefs fillsOneConstructor :: ((nat × nat × ArgType list) × nat)
                                 ⇒ instr list
fillsOneConstructor tuple ≡
      (let ((tagl,nargs,argTypes),i) = tuple;
           typePairs                 = zip argTypes [0..<nargs];
           codePairs                 = map auxOneArg typePairs
       in [Getstatic tablef consTableC,
           LitPush (Intg (int i)),
           ArrLoad,
           Dup, Dup,
           LitPush (Intg (int tagl)),
           Putfield tagLf consDataC,
           LitPush (Intg (int nargs)),
           Putfield nargsf consDataC,
           Getfield tipoArgsf consDataC]
           @ concat codePairs @ [Pop])

constdefs initConsTable :: ConstructorTableType ⇒ instr list
initConsTable ct ≡
      (let len      = length ct;
           tuples   = map (% (C,(n,tag,atypes)) . (tag,n,atypes)) ct;
           fillCode = map fillsOneConstructor (zip tuples [0..<len])
       in [LitPush (Intg (int len)),
           Invoke-static consTableC makeConsTable [PrimT Integer]]
           @ concat fillCode)
```

21.2 Generic translation functions

The translation functions are:

- *trSVM2JVM* translates a complete SafeImp program into a JVM program consisting in a single class "PSafe", with a single method called "PSafeMain", containing the in-line JVM code of the translation.

- *trCodeStore* takes the code store part of the SafeImp program and produces the bytecode part of the JVM program, together with a code map mapping SVM program counters into JVM ones, and a continuation map mapping each SafeImp continuation code label into a distinct natural number global to the program. The first part of the bytecode is a function table solving forward references in the code, and a continuation switch translating the continuation natural numbers into jumps to the correponding continuation code labels.

- *trSeq* translates a single SafeImp sequence of SVM instructions. It receives the next available JVM pc and returns the next not used pc. It also updates the current code and continuation maps.

- *trInstrAux* translates a single SVM instruction into a bytecode sequence. It updates the next available JVM pc and the codemap. The actual bytecode generation is done by *trInstr*.

consts
trInstr :: [*pc*,*codeMap*,*contMap*,*consMap*,*pc*,*SafeInstr*] => *instr list*

Translate Constructor Table: Each constructor is given a distinct number

constdefs *trConsTable* :: *ConstructorTableType* => *consMap*
trConsTable ct ≡ (
  let len = *length ct*
  in map-of (*zip* (*map fst ct*) [*0*..<*len*]))

constdefs
trInstrAux :: [*CodeLabel*,*contMap*,*consMap*,*pc*,*pc*×*nat*×*codeMap*,*SafeInstr*]
  => (*pc*×*nat*×*codeMap*) × *instr list*
trInstrAux p ctmap com pcc state safeinstr == (
  case state of (*pc*,*i*,*cdmap*) =>
  let cdmap′ = *cdmap* ((*p*,*i*) ↦ *pc*);
    instrs = *trInstr pc cdmap′ ctmap com pcc safeinstr*;
    n = *length instrs*
  in ((*pc*+*n*,*i*+*1*,*cdmap′*),*instrs*))

constdefs
trSeq :: [*contMap*,*consMap*,*pc*,*pc*×*codeMap*,*CodeLabel*×*CodeSequence*×*FunName*]
  => (*pc*×*codeMap*)× *instr list*
trSeq ctmap com pcc state seq == (
  let (*pc*,*cdmap*) = *state*;
    (*p*,*svms*,*f*) = *seq*;
    ((*pc′*,*n*,*cdmap′*),*instss*) =
      mapAccumL (*trInstrAux p ctmap com pcc*) (*pc*,*0*,*cdmap*) *svms*
  in ((*pc′*,*cdmap′*),*concat instss*))

axioms *svms-good*:
distinct (*map fst* (*svms*::*SVMCode*))

constdefs
trCodeStore :: [*CodeLabel*,*pc*,*ContinuationMap*,*consMap*,*SVMCode*]
⇒ *instr list* × *codeMap* × *contMap*
trCodeStore inip inipc ctmap com svmss == (
let (*fs*,*ps*,*contss*) = *unzip3 ctmap*;
conts = *concat contss*;
nc = *length conts*;
nf = *length fs*;
cdini = *map-of* (*zip* (*zip ps* (*replicate nf 0*))[*inipc*+*1*..<*inipc*+*nf*+*1*]);
ctm = *map-of* (*zip conts* [*1*..<*nc*+*1*]);
pcc = *inipc* + *nf* + *1*;
((*pc*,*cdm*),*iss*) = *mapAccumL* (*trSeq ctm com pcc*) (*pcc* + *1*,*cdini*) *svmss*;
funJumps = *zipWith* (% *p n*. *int*(*the*(*cdm* (*p*,*0*)))−*int n*)
ps [*inipc*+*1*..<*inipc*+*nf*+*1*];
funTable = *map Goto funJumps*;
contSwitch = *Tableswitch 1* (*int nc*) (*map*
((% *n*. *n*−*int pcc*) ∘ *int* ∘ *the* ∘ *cdm* ∘ (% *p*.(*p*,*0*)))
conts)
in (*Goto* (*int* (*the* (*cdm* (*inip*,*0*)))−*int inipc*) # *funTable*
@ (*contSwitch* # *concat iss*),*cdm*,*ctm*))

— Names and signature of the translated program single class and method

constdefs
safeP :: *cname*
safeP ≡ *Cname* ''*SafeP*''
safeMain :: *mname*
safeMain ≡ ''*PSafeMain*''
sigSafeMain :: *sig*
sigSafeMain ≡ (*safeMain*,[])

It generates code to initialize the runtime system by creating the constructor table, the cells, the heap and the stack. Then, it translates the SVM instructions.

constdefs
trSVM2JVM :: *SafeImpProg* ⇒ *jvm-prog* × *codeMap* × *contMap*
× *consMap*
trSVM2JVM prog == (
let ((*svms*,*ctmap*),*ini*,*constable*,*sizeTable*) = *prog*;
instrConsTable = *initConsTable constable*;
lenCT = *length instrConsTable*;
comap = *trConsTable constable*;
instrSizeTable = *initSizeTable sizeTable*;
lenST = *length instrSizeTable*;
(*instrs*,*cdm*,*ctm*) = *trCodeStore ini* (*lenCT*+*lenST*) *ctmap comap svms*;

```
    method          = (sigSafeMain,PrimT Void,
                       (10,10,instrConsTable @ instrSizeTable @ instrs,[]));
    classes         = [(safeP,objectC,[],[method])]
in  (classes @ Prog-RTS,cdm,ctm,comap))
```

constdefs
extractBytecode :: *jvm-prog* ⇒ *bytecode*
extractBytecode P ≡ (*fst* (*snd* (*snd* (*snd* (*snd* (*the* (*method′* (*P*, *safeP*) *sigSafeMain*)))))))

21.3 Specific translation from each SafeImp instruction to bytecode

Auxiliary functions of trInstr

trAddr translates two absolute addresses into a relative one. Used in JVM instructions Goto and Tableswitch.

constdefs
trAddr :: [*nat*,*nat*] ⇒ *int*
trAddr addr1 addr2 == *int addr1* − *int addr2*

nat2Str translates a nat into a string

constdefs
nat2Str :: *nat* ⇒ *string*
nat2Str v == (*if v* = *1 then "1"*
else if v = *2 then "2"*
else if v = *3 then "3"*
else if v = *4 then "4"*
else if v = *5 then "5"*
else if v = *6 then "6"*
else if v = *7 then "7"*
else "8")

PC Increases

constdefs
incCall :: *nat*
incCall ≡ *1*
incPop :: *nat*
incPop ≡ *24*
incMatchN :: *nat*
incMatchN ≡ *7*

These are fragments of MATCH and MATCHD translations.

consts
endlabel1 :: *int*
endlabel2 :: *int*
endlabel3 :: *int*

endlabel4 :: *int*
endlabel5 :: *int*
endlabel6 :: *int*
endlabel7 :: *int*
endlabel8 :: *int*
label1 :: *int*
label2 :: *int*
label3 :: *int*
label4 :: *int*
label5 :: *int*
label6 :: *int*
label7 :: *int*
label8 :: *int*
labelEndLoop :: *int*
labelLoop :: *int*

constdefs
InstLabel1 :: *instr list*
InstLabel1 ≡ [*Getstatic Sf stackC*, (∗ *S*[*top*−*i*] ← *o.arg1* ∗)
Getstatic topf stackC,
Load 5,
BinOp Substract,
Load 6,
Getfield arg1 cellC,
ArrStore,
Goto endlabel1]
nlab1 :: *nat*
nlab1 ≡ *length InstLabel1*

InstLabel2 :: *instr list*
InstLabel2 ≡ [*Getstatic Sf stackC*, (∗ *S*[*top*−*i*] ← *o.arg2* ∗)
Getstatic topf stackC,
Load 5,
BinOp Substract,
Load 6,
Getfield arg2 cellC,
ArrStore,
Goto endlabel2]
nlab2 :: *nat*
nlab2 ≡ *length InstLabel2*

InstLabel3 :: *instr list*
InstLabel3 ≡ [*Getstatic Sf stackC*, (∗ *S*[*top*−*i*] ← *o.arg3* ∗)
Getstatic topf stackC,
Load 5,
BinOp Substract,
Load 6,
Getfield arg3 cellC,
ArrStore,

$$
\begin{array}{l}
\qquad\qquad \mathit{Goto\ endlabel3}] \\
\mathit{nlab3} :: \mathit{nat} \\
\mathit{nlab3} \equiv \mathit{length\ InstLabel3} \\
\\
\mathit{InstLabel4} :: \mathit{instr\ list} \\
\mathit{InstLabel4} \equiv [\mathit{Getstatic\ Sf\ stackC},\ (*\ S[\mathit{top}-i] \leftarrow \mathit{o.arg4}\ *) \\
\qquad\qquad \mathit{Getstatic\ topf\ stackC}, \\
\qquad\qquad \mathit{Load\ 5}, \\
\qquad\qquad \mathit{BinOp\ Substract}, \\
\qquad\qquad \mathit{Load\ 6}, \\
\qquad\qquad \mathit{Getfield\ arg4\ cellC}, \\
\qquad\qquad \mathit{ArrStore}, \\
\qquad\qquad \mathit{Goto\ \ endlabel4}] \\
\mathit{nlab4} :: \mathit{nat} \\
\mathit{nlab4} \equiv \mathit{length\ InstLabel4} \\
\\
\mathit{InstLabel5} :: \mathit{instr\ list} \\
\mathit{InstLabel5} \equiv [\mathit{Getstatic\ Sf\ stackC},\ (*\ S[\mathit{top}-i] \leftarrow \mathit{o.arg5}\ *) \\
\qquad\qquad \mathit{Getstatic\ topf\ stackC}, \\
\qquad\qquad \mathit{Load\ 5}, \\
\qquad\qquad \mathit{BinOp\ Substract}, \\
\qquad\qquad \mathit{Load\ 6}, \\
\qquad\qquad \mathit{Getfield\ arg5\ cellC}, \\
\qquad\qquad \mathit{ArrStore}, \\
\qquad\qquad \mathit{Goto\ endlabel5}] \\
\mathit{nlab5} :: \mathit{nat} \\
\mathit{nlab5} \equiv \mathit{length\ InstLabel5} \\
\\
\mathit{InstLabel6} :: \mathit{instr\ list} \\
\mathit{InstLabel6} \equiv [\mathit{Getstatic\ Sf\ stackC},\ (*\ S[\mathit{top}-i] \leftarrow \mathit{o.arg6}\ *) \\
\qquad\qquad \mathit{Getstatic\ topf\ stackC}, \\
\qquad\qquad \mathit{Load\ 5}, \\
\qquad\qquad \mathit{BinOp\ Substract}, \\
\qquad\qquad \mathit{Load\ 6}, \\
\qquad\qquad \mathit{Getfield\ arg6\ cellC}, \\
\qquad\qquad \mathit{ArrStore}, \\
\qquad\qquad \mathit{Goto\ endlabel6}] \\
\mathit{nlab6} :: \mathit{nat} \\
\mathit{nlab6} \equiv \mathit{length\ InstLabel6} \\
\\
\mathit{InstLabel7} :: \mathit{instr\ list} \\
\mathit{InstLabel7} \equiv [\mathit{Getstatic\ Sf\ stackC},\ (*\ S[\mathit{top}-i] \leftarrow \mathit{o.arg7}\ *) \\
\qquad\qquad \mathit{Getstatic\ topf\ stackC}, \\
\qquad\qquad \mathit{Load\ 5}, \\
\qquad\qquad \mathit{BinOp\ Substract}, \\
\qquad\qquad \mathit{Load\ 6}, \\
\qquad\qquad \mathit{Getfield\ arg7\ cellC}, \\
\qquad\qquad \mathit{ArrStore}, \\
\qquad\qquad \mathit{Goto\ endlabel7}]
\end{array}
$$

```
nlab7 :: nat
nlab7 ≡ length InstLabel7

InstLabel8 :: instr list
InstLabel8 ≡ [Getstatic Sf stackC, (* S[top−i] ← o.arg8 *)
              Getstatic topf stackC,
              Load 5,
              BinOp Substract,
              Load 6,
              Getfield arg8 cellC,
              ArrStore,
              Goto endlabel8]
nlab8 :: nat
nlab8 ≡ length InstLabel8

Match11 :: nat ⇒ instr list
Match11 l ≡ [Getstatic Sf stackC, (* load S!l *)
             Getstatic topf stackC,
             LitPush (Intg (int l)),
             BinOp Substract,
             ArrLoad,
             Store 1,              (* local1 ← b *)
             Getstatic safeDirf dirCellC,
             Load 1,
             ArrLoad,
             Store 6,              (* local6 ← o *)
             Load 6,
             Getfield tagGf cellC,
             Store 2,              (* local2 ← TagG *)
             Getstatic tablef consTableC,
             Load 2,
             ArrLoad,
             Dup,
             Getfield tagLf consDataC,
             Store 3,              (* local3 ← tagL *)
             Getfield nargsf consDataC,
             Store 4,              (* local4 ← nargs *)
             LitPush (Intg 0),       (* i ← 0 *)
             Store 5,              (* local5 ← i*)
             Getstatic topf stackC,
             Load 4,
             BinOp Add,
             Putstatic topf stackC (* top ← top + nargs *)]
nMatch11 :: nat
nMatch11 ≡ length (Match11 0)

Match12 :: instr list
Match12 ≡ [Load 4,           (* nargs *)
             Load 5,             (* i *)
```

Ifcmp GreaterEqual labelEndLoop, (∗ $i \geq$ *nargs ?* ∗)
Load 5, (∗ *no, load argument i* ∗)
Tableswitch 0 7 [*label1*,*label2*,*label3*,*label4*,
label5,*label6*,*label7*,*label8*]]
nMatch12 :: *nat*
nMatch12 ≡ *length Match12*

Match2 :: *instr list*
Match2 ≡ [*Load 5*,
LitPush (*Intg 1*),
BinOp Add,
Store 5, (∗ $i = i + 1$ ∗)
Goto labelLoop]
nMatch2 :: *nat*
nMatch2 ≡ *5*

PC increases for MATCH and MATCHD

defs
endlabel1-def: *endlabel1* ≡ *int* (*nlab2* + *nlab3* + *nlab4* + *nlab5* + *nlab6* + *nlab7* + *nlab8* + *1*)
endlabel2-def: *endlabel2* ≡ *int* (*nlab3* + *nlab4* + *nlab5* + *nlab6* + *nlab7* + *nlab8* + *1*)
endlabel3-def: *endlabel3* ≡ *int* (*nlab4* + *nlab5* + *nlab6* + *nlab7* + *nlab8* + *1*)
endlabel4-def: *endlabel4* ≡ *int* (*nlab5* + *nlab6* + *nlab7* + *nlab8* + *1*)
endlabel5-def: *endlabel5* ≡ *int* (*nlab6* + *nlab7* + *nlab8* + *1*)
endlabel6-def: *endlabel6* ≡ *int* (*nlab7* + *nlab8* + *1*)
endlabel7-def: *endlabel7* ≡ *int* (*nlab8* + *1*)
endlabel8-def: *endlabel8* ≡ *1*

defs
label1-def: *label1* ≡ *1*
label2-def: *label2* ≡ *int* (*nlab1* + *1*)
label3-def: *label3* ≡ *int* (*nlab1* + *nlab2* + *1*)
label4-def: *label4* ≡ *int* (*nlab1* + *nlab2* + *nlab3* + *1*)
label5-def: *label5* ≡ *int* (*nlab1* + *nlab2* + *nlab3* + *nlab4* + *1*)
label6-def: *label6* ≡ *int* (*nlab1* + *nlab2* + *nlab3* + *nlab4* + *nlab5* + *1*)
label7-def: *label7* ≡ *int* (*nlab1* + *nlab2* + *nlab3* + *nlab4* + *nlab5* + *nlab6* + *1*)
label8-def: *label8* ≡ *int* (*nlab1* + *nlab2* + *nlab3* + *nlab4* + *nlab5* + *nlab6* + *nlab7* + *1*)

defs
labelEndLoop-def: *labelEndLoop* ≡ *int* (*nlab1* + *nlab2* + *nlab3* + *nlab4* + *nlab5* + *nlab6* + *nlab7*
+ *nlab8* + *3* + *nMatch2*)

defs
labelLoop-def: *labelLoop* ≡ −*int* (*nlab1* + *nlab2* + *nlab3* + *nlab4* + *nlab5* + *nlab6* + *nlab7*
+ *nlab8* + *nMatch2* + *nMatch12*) + *1*

constdefs

incMatch :: *nat*

incMatch ≡ *nMatch11* + *nMatch12* + *nlab1* + *nlab2* + *nlab3* + *nlab4* + *nlab5* + *nlab6* + *nlab7*
+ *nlab8* + *nMatch2* + *1*

incMatchD :: *nat*

incMatchD ≡ *nMatch11* + *nMatch12* + *nlab1* + *nlab2* + *nlab3* + *nlab4* + *nlab5* + *nlab6* + *nlab7*
+ *nlab8* + *nMatch2* + *3*

These are fragments of BUILDENV and BUILDCLS translations

consts

pushAux′ :: *Item* ⇒ *nat* ⇒ *instr list*

primrec

pushAux′ (*ItemConst v*) *i* =
[*Getstatic Sf stackC*,
Load 1, (∗ *load top+n* ∗)
LitPush (*Intg* (*int i*)),
BinOp Substract,
(*if* (*isBool v* = *True*)
then LitPush (*Bool* (*the-BoolT v*))
else LitPush (*Intg* (*the-IntT v*))),
ArrStore (∗ *S*[*top+n−i*] ← *v* ∗)
]

pushAux′ (*ItemVar l*) *i* =
[*Getstatic Sf stackC*,
Getstatic topf stackC,
LitPush (*Intg* (*int l*)),
BinOp Substract,
ArrLoad,
Store 2, (∗ *v* ← *S*!*l* ∗)
Getstatic Sf stackC,
Load 1, (∗ *load top+n* ∗)
LitPush (*Intg* (*int i*)),
BinOp Substract,
Load 2, (∗ *load v* ∗)
ArrStore (∗ *S*[*top+n−i*] ← *v* ∗)
]

pushAux′ (*ItemRegSelf*) *i* =
[*Getstatic Sf stackC*,
Load 1, (∗ *load top+n* ∗)
LitPush (*Intg* (*int i*)),
BinOp Substract,
Getstatic kf heapC,
ArrStore (∗ *S*[*top+n−i*] ← *k* ∗)

]

constdefs
pushAux :: *Item* × *nat* ⇒ *instr list*
pushAux pair == *case pair of* (*it*,*i*) => *pushAux′ it i*

regAux selects the region where to insert the fresh cell

consts
regAux :: *Item* ⇒ *instr list*
primrec
regAux (*ItemRegSelf*) =
[*Getstatic kf heapC*,
Store 3]
regAux (*ItemVar l*) =
[*Getstatic Sf stackC*,
Getstatic topf stackC,
LitPush (*Intg* (*int l*)),
BinOp Substract,
ArrLoad,
Store 3]

fillAux' fills the cell

consts
fillAux′ :: *Item* ⇒ *nat* ⇒ *instr list*
primrec
fillAux′ (*ItemVar l*) *i* =
[*Load 2*, (∗ *Load object Cell* ∗)
Getstatic Sf stackC, (∗ *load S!l* ∗)
Getstatic topf stackC,
LitPush (*Intg* (*int l*)),
BinOp Substract,
ArrLoad,
Putfield (*VName* (*''arg''*@ *nat2Str i*)) *cellC*]

fillAux′ (*ItemConst v*) *i* =
[*Load 2*, (∗ *load object Cell* ∗)
(*if* (*isBool v* = *True*) *then LitPush* (*Bool* (*the-BoolT v*))
else LitPush (*Intg* (*the-IntT v*))),
Putfield (*VName* (*''arg''*@ *nat2Str i*)) *cellC*]

constdefs
fillAux :: *Item* × *nat* ⇒ *instr list*
fillAux pair == *case pair of* (*it*,*i*) => *fillAux′ it i*

Translation to bytecode of each SafeImp Instruction

primrec

trInstr pc cdm ctm com pcc DECREGION =
[*Invoke-static heapC decregion* []]

```
trInstr pc cdm ctm com pcc (SLIDE m n) =
                    [LitPush (Intg (int m)),
                     LitPush (Intg (int n)),
                     Invoke-static stackC slide [PrimT Integer,
                                                 PrimT Integer]]

trInstr pc cdm ctm com pcc (CALL p) = (
                     let pc1    = the(cdm(p,0));
                         offset = trAddr pc1 (pc + incCall)
                     in  [Invoke-static heapC pushRegion [],
                          Goto offset] )

trInstr pc cdm ctm com pcc (PRIMOP oper) =
                   [Getstatic Sf stackC,   (* load S[top - 1] *)
                    Getstatic topf stackC,
                    LitPush (Intg 1),
                    BinOp Substract,
                    Dup2,                  (* Dup 2 top opstack *)
                    ArrLoad,
                    Store 1,               (* save v2 *)
                    Getstatic Sf stackC,
                    Getstatic topf stackC,
                    ArrLoad,
                    Load 1,                (* push v2 on top of v1 *)
                    BinOp oper,            (* compute v1 op v2 *)
                    ArrStore,              (* store it at S[top - 1] *)
                    Getstatic topf stackC, (* top <- top - 1 *)
                    LitPush (Intg 1),
                    BinOp Substract,
                    Putstatic topf stackC]

trInstr pc cdm ctm com pcc REUSE =
                   [Getstatic Sf stackC,
                    Getstatic topf stackC,
                    ArrLoad,
                    Invoke-static heapC copyCell [PrimT Integer],
                    Store 1,               (* local1 <- p *)
                    Getstatic Sf stackC,
                    Getstatic topf stackC,
                    Load 1,                (* S[top] <- p *)
                    ArrStore]

trInstr pc cdm ctm com pcc COPY =
                   [Getstatic Sf stackC,
                    Getstatic topf stackC,
                    ArrLoad,
                    Store 1,               (* local1 <- b *)
                    Getstatic Sf stackC,
```

```
                    Getstatic topf stackC,
                    LitPush (Intg 1),
                    BinOp Substract,
                    ArrLoad,
                    Store 2,
                    Load 1,                    (* local2 <- j *)
                    Load 2,
                    Invoke-static heapC copyRTS [PrimT Integer,
                                                 PrimT Integer],
                    Store 3,                   (* local3 <- b' *)
                    Getstatic topf stackC,
                    LitPush (Intg 1),
                    BinOp Substract,
                    Putstatic topf stackC,     (* top <- top - 1 *)
                    Getstatic Sf stackC,
                    Getstatic topf stackC,
                    Load 3,
                    ArrStore]

trInstr pc cdm ctm com pcc POPCONT =
                   [Getstatic Sf stackC,
                    Getstatic topf stackC,
                    Dup2,
                    Dup2,
                    Dup2,
                    ArrLoad,
                    Store 1,                   (* local1 <- b *)
                    LitPush (Intg 1),
                    BinOp Substract,
                    ArrLoad,
                    Store 2,                   (* local2 <- k' *)
                    LitPush (Intg 2),
                    BinOp Substract,
                    ArrLoad,
                    Store 3,                   (* local3 <- p *)
                    LitPush (Intg 2),
                    BinOp Substract,
                    Dup,
                    Putstatic topf stackC,     (* top <- top - 2 *)
                    Load 1,
                    ArrStore,                  (* S[top] <- b *)
                    Load 2,
                    Putstatic k0f heapC,       (* k0 <- k' *)
                    Load 3,                    (* jump to continuation *)
                    Goto (trAddr pcc (pc + incPop))]

trInstr pc cdm ctm com pcc  (PUSHCONT p) =
                     ( let n = the(ctm(p))
                       in
```

```
                [Getstatic topf stackC,
                 LitPush (Intg 1),
                 BinOp Add,
                 Putstatic topf stackC,      (* top <- top + 1 *)
                 Getstatic Sf stackC,
                 Getstatic topf stackC,
                 LitPush (Intg (int n)),
                 ArrStore,                   (* S[top] <- p' *)
                 Getstatic topf stackC,
                 LitPush (Intg 1),
                 BinOp Add,
                 Putstatic topf stackC,      (* top <- top + 1 *)
                 Getstatic Sf stackC,
                 Getstatic topf stackC,
                 Getstatic k0f heapC,
                 ArrStore,                   (* S[top] <- k0 *)
                 Getstatic kf heapC,         (* k0 <- k *)
                 Putstatic k0f heapC])

trInstr pc cdm ctm com pcc (MATCHN l v m ps) =
            (let pcs  = map (%p.(the(cdm(p,0)))) ps;
                 pcs' = map (%n.(trAddr n (pc + incMatchN))) pcs
             in  [Getstatic Sf stackC,       (* load S!l *)
                 Getstatic topf stackC,
                 LitPush (Intg (int l)),
                 BinOp Substract,
                 ArrLoad,
                 LitPush (Intg (int v)),     (* substract v *)
                 BinOp Substract,
                 Tableswitch 0 (int (m + 1)) pcs'])

trInstr pc cdm ctm com pcc (MATCH l ps) =
            (let len  = length ps;
                 pcs  = map (%p.(the(cdm(p,0)))) ps;
                 pcs' = map (%n.(trAddr n(pc + incMatch))) pcs
             in  Match11 l  @ Match12 @
                 InstLabel1 @ InstLabel2 @ InstLabel3 @ InstLabel4 @
                 InstLabel5 @ InstLabel6 @ InstLabel7 @ InstLabel8 @
                 Match2 @
                 [Load 3,                    (* Load tagL *)
                 Tableswitch 0 (int (len - 1)) pcs'])

trInstr pc cdm ctm com pcc (MATCHD l ps) =
            (let len = length ps;
                 pcs = map (%p.(the(cdm(p,0)))) ps;
                 pcs' = map (%n.(trAddr n(pc + incMatchD))) pcs
             in  Match11 l  @ Match12 @
                 InstLabel1 @ InstLabel2 @ InstLabel3 @ InstLabel4 @
                 InstLabel5 @ InstLabel6 @ InstLabel7 @ InstLabel8 @
```

```
                Match2 @
                [Load 1,                    (* release Cell b *)
                Invoke-static dirCellC releaseCell [PrimT Integer],
                Load 3,                     (* Load tagL *)
                Tableswitch 0 (int (len − 1)) pcs']

trInstr pc cdm ctm com pcc (BUILDENV its) =
          (let n       = length its;
               genCode = map pushAux (zip its [0..<n])
           in  [Getstatic topf stackC,
               LitPush (Intg (int n)),
               BinOp Add,
               Store 1]            (* local1 ← top + n *)
               @ concat genCode
               @ [Load 1, Putstatic topf stackC])

trInstr pc cdm ctm com pcc (BUILDCLS c its item) =
          (let tagg   = the(com c);
               n      = length its;
               codRes = [Invoke-static dirCellC reserveCell [],
                        Store 1,   (* local1 ← p *)
                        Getstatic safeDirf dirCellC,
                        Load 1,
                        ArrLoad,
                        Store 2, (* local2 ← o cell *)
                        Load 2,
                        LitPush (Intg (int tagg)),
                        Putfield tagGf cellC];  (* o.tagG = tagg *)
               codFil = map fillAux (zip its [1..<n+1]);
               codIns = [Load 3,
                        Load 1,
                        Invoke-static heapC insertCell [PrimT Integer,
                                                        PrimT Integer],
                        Getstatic topf stackC,
                        LitPush (Intg 1),
                        BinOp Add,
                        Putstatic topf stackC, (* top ← top + 1 *)
                        Getstatic Sf stackC,
                        Getstatic topf stackC,
                        Load 1,
                        ArrStore]              (* S[top] ← b *)
           in codRes @ regAux item @ concat codFil @ codIns)
```

end

22 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
                  Val v#[]              => Left (h,k0,pc,S)
                | Val v#Cont (k0,l)#S' => Right (h,k0,(l,0),Val 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 Left (h,k0,pc,S))

  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 (IntT i)# Val (IntT i')#S' =>
                   let v = execOp oper (IntT i) (IntT i') 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) =>
                            if j ≤ snd h then
                              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)
                            else Left (h,k0,pc,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*)
$P \vdash s\ -svm\rightarrow\ s' \equiv (s,s') \in \{(s,t)\ .\ execSVM\ P\ s = Right\ t\}\hat{}*$

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: $P \vdash s,\ n\#ns\ -svm\rightarrow\ [s],\ n\#ns$
| *step*: ⟦ $P \vdash s,\ n\#ns\ -svm\rightarrow\ s'\#ss,\ m\#ms$;
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*)⟧
⟹
$P \vdash s,\ n\#ns\ -svm\rightarrow\ s''\#s'\#ss,\ ms'$

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*]:
$P \vdash s'',\ n''\#ns''\ -svm\rightarrow\ sss\ ,\ n'''\#ns'''$ ⟶ (∀ *s''' ss'' ss*. *sss* = *s'''*#*ss''* ⟶

$P \vdash s',\ n'\#ns'\ -svm\rightarrow\ s''\ \#\ ss',\ n''\#ns''$ ⟶
ss = *ss''* @ *ss'* ⟶

$P \vdash s'$, $n'\#ns' -svm\rightarrow s'''\# ss$, $n'''\#ns''')$
apply (*rule 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*:
$[\![P \vdash s', n'\#ns' -svm\rightarrow s''\ \#\ ss', n''\#ns''$;
$P \vdash s'', n''\#ns'' -svm\rightarrow s'''\# ss'', n'''\#ns'''$;
$ss = ss'' @ ss']\!]$
$\Longrightarrow P \vdash s', n'\#ns' -svm\rightarrow s'''\# ss, n'''\#ns'''$
apply (*rule execSVMBalanced-n-step-aux*)
apply *simp-all*

done

end

23 Certification of the translation from SVM to JVM

theory *CertifSVM2JVM*
imports *../JVMSAFE/JVMExec SVM2JVM SVMSemantics*
begin

23.1 Equivalence relations between SVM and JVM states

types *injection* = *Location* ⇀ *loc*

— Equivalence of values under a cell directory and an injection

fun *equivV* :: *Val* => *val* => *entries-* => *injection* => *bool*
where
  equivV (*IntT i*) *v d g* = (∃ *j*. *v* = *Intg j* ∧ (*i* = *j*))
| *equivV* (*BoolT b*) *v d g* = (∃ *i*. *v* = *Intg i* ∧
                ((*i* ≠ *0* ∧ *b*) ∨ (*i* = *0* ∧ ¬ *b*)))
| *equivV* (*Val.Loc l*) *v d g* = (∃ *i l′*. *v* = *Intg i*
                ∧ *d* (*nat i*) = *Some* (*Addr l′*) ∧ *g l* = *Some l′*)

— Cell to the right of a given one in JVM heap
constdefs
 nextCell :: *aheap*⇒ *loc*⇒ *loc*
 nextCell h l == (*case h l of*
          Some (*Obj cls flds*) =>
            case flds (*derf*,*cellC*) *of*
              Some (*Addr next*) => *next*)

— Cells belonging to a region in the JVM heap, starting from a given one
inductive-set
  cellsReg :: *aheap* ⇒ *loc* ⇒ *loc set*
  for *h* :: *aheap* **and** *l*:: *loc*
where
  cellReg-basic: *l* ∈ *cellsReg h l*
| *cellReg-step*: ⟦ *l* ∈ *cellsReg h l*; *l′* = *nextCell h l*⟧
          ⟹ *l′* ∈ *cellsReg h l*

— this gives us the set of references belonging to a given region
constdefs
 region :: *entries-* => *aheap* => *nat* => *loc set*
 region rs h j == (
      let header = *the-Addr* (*the* (*rs j*));

```
        allcells = cellsReg h header
    in  allcells − {header})
```

— this gives us the set of references belonging to all regions

constdefs

```
 activeCells :: entries- => aheap => nat => loc set
 activeCells rs h k ≡ {p . ∃ j . j ≤ k ∧ p ∈ region rs h j}
```

constdefs

```
 passiveCells :: entries- => nat => loc set
 passiveCells rs k ≡ {p. ∃  j. j  ≤ k ∧  p = the-Addr (the (rs j))}
```

— This gives us the i-th argument of a cell

constdefs

```
 argCell :: fields- => nat => val
 argCell flds i == (
      let arg-i = if i=1 then arg1 else
                  if i=2 then arg2 else
                   if i=3 then arg3 else
                    if i=4 then arg4 else
                     if i=5 then arg5 else
                      if i=6 then arg6 else
                       if i=7 then arg7 else arg8
      in the (flds (arg-i,cellC)))
```

A SVM cell is equivalent to a JVM cell if the JVM object is an instance of the class Cell, if they live in the same region j, the constructor name and the global tag are made equivalent by the constructor map cm, and all the arguments are equivalent.

— Equivalence of cells under a constructor map, a region stack, a cell directory
— and an injection

fun

```
 equivC :: [Region × Cell,aheap,loc,heap-entry,nat,consMap,entries-,entries-,
           injection] ⇒ bool
```

where

```
  equivC cell h ref cell′ k cm regStack d g = (
     ∃ j C vs Obj cname flds tag n reg-j tag′.
       cell = (j,C,vs) ∧
       j ≤ k ∧
       cell′ = Obj cname flds ∧
       cm C = Some tag ∧
       n = length vs ∧
       reg-j = region regStack h j ∧
       tag′ = nat (the-Intg (the (flds (tagGf,cname)))) ∧
       cname = cellC ∧
       ref ∈ reg-j ∧
       tag = tag′ ∧
       (∀ i∈ {k . 0≤k & k < n}. equivV (vs!i) (argCell flds i) d g))
```

— Equivalence between SVM and JVM heaps under a constructor map,
— a region stack, a cell directory and an injection

fun
equivH :: [*Heap*,*aheap*,*nat*,*consMap*,*entries-*,*entries-*,*injection*] => *bool*
where
equivH (*H*,*k1*) *h k2 cm regStack d g* = (
k1=*k2* ∧
ran g = *activeCells regStack h k2* ∧
finite (*dom H*) ∧
(∀ *l*∈ *dom H* . ∃ *l'*. *l'* = *the* (*g l*)
∧ *equivC* (*the* (*H l*)) *h l'* (*the* (*h l'*)) *k2 cm regStack d g*))

— Equivalence between SVM and JVM stacks under a continuation map, a cell
— directory and an injection
fun
equivS :: [*Stack*,*entries-*,*int*,*contMap*,*entries-*,*injection*] => *bool*
where
equivS [] *S'* *n ctm d g* = (*n* = (−*1*))
| *equivS* (*Cont* (*k*,*p*)#*S*) *S'* *n ctm d g* = (*n* >= *1*
∧ *k* = *nat* (*the-Intg* (*the* (*S'* (*nat n*))))
∧ *the* (*ctm p*) = *nat* (*the-Intg* (*the* (*S'* ((*nat n* − *1*)))))
∧ *equivS S S'* (*n* − *2*) *ctm d g*)
| *equivS* (*Reg j*#*S*) *S'* *n ctm d g* = (*n* >= *0*
∧ *j* = *nat* (*the-Intg* (*the* (*S'* (*nat n*))))
∧ *equivS S S'* (*n* − *1*) *ctm d g*)
| *equivS* (*Val v*#*S*) *S'* *n ctm d g* = (*n* >= *0*
∧ *equivV v* (*the* (*S'* (*nat n*))) *d g*
∧ *equivS S S'* (*n* − *1*) *ctm d g*)

— Equivalence between SVM and JVM states
constdefs
equivState :: [*codeMap*,*contMap*,*consMap*,*SVMState*,*jvm-state*] => *bool*
(- , -, - ⊢ - ≙ - [*71*,*71*,*71*,*71*,*71*] *70*)
equivState cdm ctm com st1 st2 == (
∃ *H k k0 PC S sh h inih vs pc ref k' k0' l ty m S' n l' regS l''*
m' m'' d g .
st1 = ((*H*,*k*),*k0*,*PC*,*S*) ∧
st2 = (*None*,*sh*,*h*,*inih*,([],*vs*,*safeP*,*sigSafeMain*,*pc*,*ref*)#[]) ∧
k' = *nat* (*the-Intg* (*the* (*sh* (*heapC*,*kf*)))) ∧
k0' = *nat* (*the-Intg* (*the* (*sh* (*heapC*,*k0f*)))) ∧
sh (*stackC*,*Sf*) = *Some* (*Addr l*) ∧
distinct [*l*,*l'*,*l''*] ∧
activeCells regS h k' ∩ {*l*,*l'*,*l''*} = {} ∧
h l = *Some* (*Arr ty m S'*) ∧
sh (*stackC*,*topf*) = *Some* (*Intg n*) ∧

$sh\ (heapC,regionsf) = Some\ (Addr\ l') \wedge$
$h\ l' = Some\ (Arr\ ty\ m'\ regS) \wedge$
$sh\ (dirCellC,safeDirf) = Some\ (Addr\ l'') \wedge$
$h\ l'' = Some\ (Arr\ ty\ m''\ d) \wedge$
$dom\ g = dom\ H \wedge$
$inj\text{-}on\ g\ (dom\ H) \wedge$
$equivH\ (H,k)\ h\ k'\ com\ regS\ d\ g \wedge$
$n < int\ m \wedge$
$equivS\ S\ S'\ n\ ctm\ d\ g \wedge$
$k0 = k0' \wedge$
$pc = the\ (cdm\ PC))$

axioms *PC-good*:
$[\![\ (P',cdm,ctm,com) = trSVM2JVM\ ((svms,\ ctmap),\ ini,\ ct,\ st);$
$equivState\ cdm\ ctm\ com\ ((H,k),k0,(l,i),S)\ st2$
$]\!] \Longrightarrow\ l : dom\ (map\text{-}of\ svms)$
$\wedge\ i < length\ (fst\ (the\ (map\text{-}of\ svms\ l)))$

axioms *RightNotUndefined* : $\forall\ x\ .\ \neg\ (undefined = Right\ x)$

axioms *execSVMInstr-COPY* :
$[\![\ (P',\ cdm,\ ctm,\ com) = trSVM2JVM\ ((svms,\ ctmap),\ ini,\ ct,\ ah,\ ai,\ bc);$
$cdm\ ,\ ctm,\ com \vdash ((hm,\ k),\ k0,\ (l,\ i),\ S) \stackrel{\triangle}{=} S1';$
$(fst\ (the\ (map\text{-}of\ svms\ l))\ !\ i) = COPY;$
$execSVMInst\ COPY\ (map\text{-}of\ ct)\ (hm,\ k)\ k0\ (l,\ i)\ S = Either.Right\ S2;$
$drop\ (the\ (cdm\ (l,\ i)))\ (extractBytecode\ P') =$
$trInstr\ (the\ (cdm\ (l,\ i)))\ cdm'\ ctm'\ com\ pcc\ COPY\ @\ bytecode'$
$]\!] \Longrightarrow\ \exists\ v'\ sh'\ dh'\ ih'\ fms'\ .$
$P' \vdash\ S1' -jvm\rightarrow (v',sh',dh',ih',\ fms') \wedge$
$cdm\ ,\ ctm,\ com\ \vdash\ S2\ \stackrel{\triangle}{=}\ (v',sh',dh',ih',\ fms')$

axioms *execSVMInstr-MATCHD* :
$[\![\ (P',\ cdm,\ ctm,\ com) = trSVM2JVM\ ((svms,\ ctmap),\ ini,\ ct,\ ah,\ ai,\ bc);$
$cdm\ ,\ ctm,\ com \vdash ((hm,\ k),\ k0,\ (l,\ i),\ S) \stackrel{\triangle}{=} S1';$
$(fst\ (the\ (map\text{-}of\ svms\ l))\ !\ i) = MATCHD\ off\ ps;$
$execSVMInst\ (MATCHD\ off\ ps)\ (map\text{-}of\ ct)\ (hm,\ k)\ k0\ (l,\ i)\ S =$
$Either.Right\ S2;$
$drop\ (the\ (cdm\ (l,\ i)))\ (extractBytecode\ P') =$
$trInstr\ (the\ (cdm\ (l,\ i)))\ cdm'\ ctm'\ com\ pcc\ (MATCHD\ off\ ps)\ @\ bytecode'$
$]\!] \Longrightarrow\ \exists\ v'\ sh'\ dh'\ ih'\ fms'\ .$
$P' \vdash\ S1' -jvm\rightarrow (v',sh',dh',ih',\ fms') \wedge$
$cdm\ ,\ ctm,\ com\ \vdash\ S2\ \stackrel{\triangle}{=}\ (v',sh',dh',ih',\ fms')$

axioms *equivS-ge-n*:

$equivS\ S\ S'\ n\ ctm\ d\ g \Longrightarrow$
$(\forall\ \ p > 0.\ equivS\ S\ (S'(nat\ (p + n)\ \mapsto A))\ n\ ctm\ d\ g)$

axioms *equivS-length*:
$equivS\ S\ S'\ n\ ctm\ d\ g \Longrightarrow n = int\ (length\ S) - 1$

axioms *length-vs*:
$length\ vs = 10$

lemma *l-not-in-cellReg*:
$[\![\ l \notin activeCells\ regS\ h\ k;\ l \notin passiveCells\ regS\ k\ ]\!]$
$\Longrightarrow \forall\ j \leq k.\ l \notin cellsReg\ h\ (the\text{-}Addr\ (the\ (regS\ j)))$
apply (*simp add*: *activeCells-def*)
apply (*simp add*: *passiveCells-def*)
apply (*simp add*: *region-def*)
done

axioms *safeDir-bounds*:
$the\text{-}Intg\ (the\ (S'\ i)) < int\ m \wedge 0 \leq the\text{-}Intg\ (the\ (S'\ i))$

axioms *l-not-in-cellReg*:
$l \notin activeCells\ regS\ h\ k$
$\Longrightarrow \forall\ j\ .\ l \notin cellsReg\ h\ (the\text{-}Addr\ (the\ (regS\ j)))$

axioms *activeCells*:
$l \notin activeCells\ regS\ h\ k$
$\Longrightarrow l \notin activeCells\ regS\ (h(l \mapsto A))\ k$

axioms *activeCells-2*:
$[\![\ l' \notin activeCells\ regS\ h\ k;\ l \neq l'\ ]\!]$
$\Longrightarrow l' \notin activeCells\ regS\ (h(l \mapsto A))\ k$

axioms *method-safeMain*:
$(P',\ cdm,\ ctm,\ com) = trSVM2JVM\ ((svms,\ ctmap),\ ini,\ ct,\ ah,\ ai,\ bc)$
$\Longrightarrow (safeP,PrimT\ Void,10,10,(instrConsTable\ @\ instrSizeTable\ @\ instrs,[])) =$

$(the\ (method'\ (P',\ safeP)\ (safeMain,\ [])))$

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

lemma *cellsReg-monotone-1* [*rule-format*]:
$x \in cellsReg\ h\ l$
$\longrightarrow x \neq l$
$\longrightarrow l \neq l'$
$\longrightarrow x = nextCell\ (h(l' \mapsto A))\ l$

apply (*rule impI*)
apply (*erule cellsReg.induct*,*simp*)
by (*simp add*: *nextCell-def*)

lemma *cellsReg-monotone-2* [*rule-format*]:
$x \in cellsReg\ (h(l' \mapsto A))\ l$
$\longrightarrow x \neq l$
$\longrightarrow l \neq l'$
$\longrightarrow x = nextCell\ h\ l$
apply (*rule impI*)
apply (*erule cellsReg.induct*,*simp*)
by (*simp add*: *nextCell-def*)

lemma *l-not-in-regs*:
$l \notin cellsReg\ h\ (the\text{-}Addr\ (the\ (regS\ j)))$
$\Longrightarrow the\text{-}Addr\ (the\ (regS\ j)) \neq l$
apply (*subgoal-tac the-Addr* (*the* (*regS j*)) ∈ *cellsReg h* (*the-Addr* (*the* (*regS j*))))

apply *blast*
by (*rule cellReg-basic*)

end

24 Lemmas on the static properties of the translation SVM to JVM

theory *dem-translation*
imports *../JVMSAFE/JVMExec SVM2JVM SVMSemantics CertifSVM2JVM*
begin

declare *trInstr.simps* [*simp del*]
declare *equivS.simps* [*simp del*]
declare *drop-append* [*simp del*]

lemma *fun-trInstr1-aux* [*rule-format*]:
(∀ *pc i cdm pc' i' cdm' insts* .
(∀ *j* ≥ *i* . (*p*,*j*) ∉ *dom cdm*) ⟶
((*pc'*,*i'*,*cdm'*),*insts*) = *mapAccumL* (*trInstrAux p ctm com pcc*) (*pc*,*i*,*cdm*) *svms*
⟶ *cdm* ⊆$_m$ *cdm'*∧ (∀ *j* ≥ *i'*. (*p*,*j*) ∉ *dom cdm'*))

apply (*induct svms*)

apply *simp*

apply (*rule allI*)+
apply (*rename-tac inst svms pc1 i1 cdm1 pc2 i2 cdm2 insts2*)
apply (*unfold mapAccumL.simps*)
apply (*case-tac trInstrAux p ctm com pcc* (*pc1*, *i1*, *cdm1*) *inst*)
apply (*rename-tac state bc*)
apply (*case-tac state*, *rename-tac pc3 rest*)
apply (*case-tac rest*, *rename-tac i3 cdm3*, *simp*)
apply (*case-tac mapAccumL* (*trInstrAux p ctm com pcc*) (*pc3*, *i3*, *cdm3*) *svms*)
apply (*rename-tac state4 insts4*)
apply (*case-tac state4*,*rename-tac pc4 rest4*)
apply (*case-tac rest4*, *rename-tac i4 cdm4*,*clarsimp*)
apply (*subgoal-tac* ((*pc4*, *i4*, *cdm4*), *insts4*)
= *mapAccumL* (*trInstrAux p ctm com pcc*) (*pc3*, *i3*, *cdm3*) *svms*)
prefer *2* **apply** *simp*

apply (*erule-tac x=pc3* **in** *allE*)
apply (*rotate-tac −1*)
apply (*erule-tac x=i3* **in** *allE*)
apply (*rotate-tac −1*)
apply (*erule-tac x=cdm3* **in** *allE*)

apply (*erule impE*)
prefer *2*
apply (*rotate-tac −1*)
apply (*erule-tac x=pc4* **in** *allE*)
apply (*rotate-tac −1*)
apply (*erule-tac x=i4* **in** *allE*)
apply (*rotate-tac −1*)
apply (*erule-tac x=cdm4* **in** *allE*)
apply (*erule impE*)
apply (*rule-tac x=insts4* **in** *exI*,*assumption*)

apply (*elim conjE*)
apply (*rule conjI*)
apply (*unfold trInstrAux-def*) **apply** (*simp*)
apply (*elim conjE*)
apply (*subgoal-tac cdm1* $\subseteq_m$ *cdm1*((*p*, *i1*) $\mapsto$ *pc1*))
apply (*erule map-le-trans*,*simp*)

apply (*erule-tac x=i1* **in** *allE*)
apply (*erule impE*,*simp*)
apply (*unfold map-le-def*) **apply**(*force*)
apply *assumption*
apply *simp*
apply (*elim conjE*)

apply *clarify*
apply (*erule-tac x=j* **in** *allE*)
apply (*drule mp*)
by *auto*

lemma *fun-trInstr1* [*rule-format*]:
(∀ *pc i cdm pc′ i′ cdm′ insts* .
(∀ *j* ≥ *i* . (*p*,*j*) ∉ *dom cdm*) ⟶
((*pc′*,*i′*,*cdm′*),*insts*) = *mapAccumL* (*trInstrAux p ctm com pcc*) (*pc*,*i*,*cdm*) *svms*
⟶ *cdm* ⊆$_m$ *cdm′*)
apply (*rule allI*)+
apply (*rule impI*)+
thm *fun-trInstr1-aux*
apply (*subgoal-tac cdm* ⊆$_m$ *cdm′*∧ (∀ *j* ≥ *i′*. (*p*,*j*) ∉ *dom cdm′*))
prefer *2* **apply** (*rule fun-trInstr1-aux*)
apply (*erule-tac x=j* **in** *allE*) **apply** (*drule mp*)
by *auto*

lemma *fun-trInstrAux*:
((*pc′*,*cdm′*),*bc′*) = *trSeq ctm com pcc* (*pc*,*cdm*) (*p*,*svms*,*fn*) ⟶
svms = *is* # *iss* ⟶
(∀ *j*≥*0* . (*p*,*j*) ∉ *dom cdm*) ⟶
drop pc bc = *bc′* @ *rest* ⟶
(∀ *i* . *i* < *length svms* ⟶
(∃ *pc″ cdm″ bc″ rest′ insts1*.
((*pc″*+*length bc″*,*i*+*1*,*cdm″*((*p*,*i*)↦ *pc″*)),*bc″*) =
trInstrAux p ctm com pcc (*pc″*,*i*,*cdm″*) (*svms*!*i*)
∧ *drop* (*pc″*−*pc*) *bc′* = *bc″* @ *rest′*
∧ ((*pc″*,*i*,*cdm″*),*insts1*)=
mapAccumL (*trInstrAux p ctm com pcc*) (*pc*,*0*,*cdm*) (*take i svms*)
∧ *cdm″*((*p*,*i*)↦ *pc″*) ⊆$_m$ *cdm′*))

apply (*rule impI*)+
apply (*rule allI*)

apply (*induct-tac i*)

apply (*rule impI*)
apply (*unfold trSeq-def*)
apply *clarify*
apply (*case-tac* (*pc*,*cdm*))
apply (*case-tac* (*p*, *is* # *iss*, *fn*))
apply (*rename-tac pc2 cdm2 p2 rest2*)
apply (*case-tac rest2*)
apply (*rename-tac instr ff*)
apply (*simp del*: *mapAccumL.simps*)
apply (*case-tac mapAccumL* (*trInstrAux p2 ctm com pcc*) (*pc2*, *0*, *cdm2*) *instr*)

apply (*rename-tac result instss2*)
apply (*case-tac result*)
apply (*rename-tac pc3 rest2*)
apply (*case-tac rest2*)
apply (*simp del*: *mapAccumL.simps*)
apply *clarify*

apply (*rename-tac nn bca*)
apply (*subgoal-tac* (∃ *s' y ys'*.
(*s'*,*y*) = *trInstrAux p ctm com pcc* (*pc*, *0*, *cdm*) *is*
∧ ((*pc'*,*nn*,*cdm'*),*ys'*)=*mapAccumL* (*trInstrAux p ctm com pcc*) *s' iss*
∧ *instss2* = *y* # *ys'*))
prefer *2*
apply (*subgoal-tac* ((*pc'*, *nn*, *cdm'*), *instss2*)=
mapAccumL (*trInstrAux p ctm com pcc*) (*pc*, *0*, *cdm*) (*is* # *iss*))
prefer *2* **apply** *simp*
apply (*erule mapAccumL-non-empty2*,*simp*)
apply (*elim exE*,*elim conjE*)
apply *clarify*
apply (*rename-tac pc4 nn4 cdm4 bc0 ys'*)

apply (*rule-tac x*=*pc* **in** *exI*)
apply (*rule-tac x*=*cdm* **in** *exI*)
apply (*rule-tac x*=*bc0* **in** *exI*)

apply (*subgoal-tac* (*pc4*, *nn4*, *cdm4*)=
(*pc* + *length bc0*, *Suc 0*, λ *a*. *if a* = (*p*, *0*) *then Some pc else cdm a*))
apply (*rule conjI*)
apply *simp*
apply (*rule conjI*)
apply (*rule-tac x*=*concat ys'* **in** *exI*)
apply *force*
apply (*rule conjI*)
apply (*rule-tac x*=[] **in** *exI*,*simp*)
apply (*simp del*: *mapAccumL.simps add*: *trInstrAux-def*)
apply (*fold fun-upd-apply*)
apply (*elim conjE*)
apply *clarify*
thm *fun-trInstr1*
apply (*rule-tac i*=*Suc 0* **and** *p*=*p* **in** *fun-trInstr1*)
apply (*erule-tac x*=*j* **in** *allE*)
apply *force*
apply *simp*
apply (*simp del*: *mapAccumL.simps add*: *trInstrAux-def*)
apply (*fold fun-upd-apply*)
apply *force*

apply *clarify*
apply (*case-tac* (*pc*,*cdm*))
apply (*case-tac* (*p*, *is* # *iss*, *fn*))
apply (*rename-tac pc2 cdm2 p2 rest2*)
apply (*case-tac rest2*)
apply (*rename-tac instr ff*)
apply (*simp del*: *mapAccumL.simps*)
apply (*case-tac mapAccumL* (*trInstrAux p2 ctm com pcc*) (*pc2*, *0*, *cdm2*) *instr*)
apply (*rename-tac result instss*)
apply (*case-tac result*)
apply (*rename-tac pc3 rest3*)
apply (*case-tac rest3*)
apply (*rename-tac n3 cdm3*)
apply (*simp del*: *mapAccumL.simps*)
apply *clarify*

apply (*subgoal-tac* (*take n* (*is*#*iss*)) @ (*drop n* (*is*#*iss*))=*is*#*iss*)
prefer *2*
apply (*rule append-take-drop-id*)
apply (*subgoal-tac* ∃ *s1 insts1 insts2* . (*s1*,*insts1*) =
mapAccumL (*trInstrAux p ctm com pcc*) (*pc*, *0*, *cdm*) (*take n* (*is* # *iss*))
∧ ((*pc′*, *n3*, *cdm′*), *insts2*)=
mapAccumL (*trInstrAux p ctm com pcc*) *s1* (*drop n* (*is* # *iss*))
∧ *instss*= *insts1* @ *insts2*)
prefer *2*
apply (*rule mapAccumL-split* [**where** *xs*=*is* # *iss*]) **apply**(*simp*,*simp*)
apply (*elim exE*, *elim conjE*)
apply (*case-tac s1*)
apply (*rename-tac pc1 rest1*)
apply (*case-tac rest1*)
apply (*rename-tac nn1 cdm1*)
apply *clarify*
apply (*subgoal-tac* ((*pc″*, *n*, *cdm″*), *insts1*)=((*pc1*, *nn1*, *cdm1*), *insts1a*))
prefer *2*
apply *simp*
apply *clarify*
apply (*thin-tac ?x*=*mapAccumL ?f* (*pc*,*0*,*cdm*) *?xs*)

apply (*subgoal-tac nn1* < *length* (*is*#*iss*))
apply (*subgoal-tac* (∃ *is2 iss2* . *drop nn1* (*is* # *iss*) = *is2*#*iss2*
∧ (*is* # *iss*) ! *nn1* = *is2*))
prefer *2*
apply (*erule drop-nth*)
prefer *2*
apply *simp*

apply (*elim exE*)
apply *clarify*

apply (*subgoal-tac* (∃ *s′ y ys′*.
(*s′*,*y*) = *trInstrAux p ctm com pcc* (*pc1*, *nn1*, *cdm1*) ((*is* # *iss*) ! *nn1*)
∧ ((*pc′*, *n3*, *cdm′*),*ys′*) = *mapAccumL* (*trInstrAux p ctm com pcc*) *s′ iss2*
∧ *insts2* = *y* # *ys′*))
prefer *2*
apply (*erule mapAccumL-non-empty2*,*simp*)
apply (*elim exE*)
apply (*elim conjE*)
apply (*simp del*: *mapAccumL.simps drop-append*)
apply (*rename-tac s′ bc2 ys′*)
apply (*case-tac s′*)
apply (*rename-tac pc2 rest2*)
apply (*case-tac rest2*, *rename-tac nn2 cdm2*,*clarify*)
apply (*subgoal-tac* ((*pc1* + *length bc′′*, *Suc nn1*,
% *a*. *if a* = (*p*, *nn1*) *then Some pc1 else cdm1 a*), *bc′′*)
=((*pc2*, *nn2*, *cdm2*), *bc2*))
prefer *2*
apply *simp*
apply (*clarify*, *thin-tac ?x*=*trInstrAux p ctm com pcc ?y ?z*)

apply (*subgoal-tac* ∃ *iss3*. *iss2*=(*iss* ! *nn1*)#*iss3*)
apply (*elim exE*,*clarify*)
apply (*subgoal-tac* (∃ *s1 s2 y1 y2 ys3* .
(*s1*,*y1*)=*trInstrAux p ctm com pcc* (*pc1*, *nn1*, *cdm1*) ((*is* # *iss*) ! *nn1*)
∧ (*s2*,*y2*)=*trInstrAux p ctm com pcc s1* (*iss*!*nn1*)
∧ ((*pc′*, *n3*, *cdm′*), *ys3*)=*mapAccumL* (*trInstrAux p ctm com pcc*) *s2 iss3*
∧ *bc2*#*ys′*=*y1*#*y2*#*ys3*))
prefer *2*
apply (*erule mapAccumL-two-elements*)
apply *simp*
apply (*elim exE*)
apply (*case-tac s1*, *rename-tac pc3 rest3*,*clarify*)
apply (*rename-tac nn3 cdm3*)
apply (*subgoal-tac* ((*pc3*, *nn3*, *cdm3*), *y1*)=
((*pc1* + *length y1*, *Suc nn1*,
% *a*. *if a* = (*p*, *nn1*) *then Some pc1 else cdm1 a*), *y1*))
prefer *2*
apply *simp*
apply (*clarify*,*thin-tac ?x*=*trInstrAux p ctm com pcc ?y ?z*)
apply (*rename-tac pc4 nn4 cdm4 y1 y2 ys3 pc3 nn3 cdm3*)

prefer *2*
apply (*rule-tac x*=*tl iss2* **in** *exI*)
apply (*subgoal-tac drop* (*Suc nn1*) (*is*#*iss*) = *iss2*)

prefer *2*
apply (*subgoal-tac drop* (*Suc nn1*) (*is* # *iss*)=*drop nn1* (*tl* (*is*#*iss*)))
prefer *2*
apply (*rule drop-Suc*)
apply (*subgoal-tac is* # *iss*=*take nn1* (*is* # *iss*) @ (*is* # *iss*) ! *nn1* # *iss2*)
prefer *2*
apply *simp*
apply (*erule drop-take-Suc*)
apply (*rule drop-nth2*,*assumption*,*simp*)

apply (*rule-tac x*=*pc1* + *length y1* **in** *exI*)
apply (*rule-tac x*= % *a*. *if a* = (*p*, *nn1*) *then Some pc1 else cdm1 a* **in** *exI*)
apply (*rule-tac x*=*y2* **in** *exI*)
apply (*rule conjI*)

apply (*thin-tac mapAccumL ?x ?y ?z* = *?w*)
apply (*thin-tac ?w*=*mapAccumL ?x ?y ?z*)
apply (*thin-tac ?w*=*mapAccumL ?x ?y ?z*)
apply (*thin-tac ?x*=*trInstrAux ?x1 ?x2 ?x3 ?x4 ?x5 ?x6*)
apply (*simp add*: *trInstrAux-def*)
apply *clarify*
apply (*fold fun-upd-apply*)
apply *force*

apply (*rule conjI*)
apply (*rule-tac x*=*concat ys3* **in** *exI*)
apply (*simp only*: *concat.simps*)
apply (*rule drop-append2*,*simp*,*simp*)

apply (*rule conjI*)
apply (*rule-tac x*=*insts1a* @ [*y1*] **in** *exI*)
apply (*subgoal-tac is* # *take nn1 iss*=(*take nn1* (*is* # *iss*)) @ [(*is* # *iss*) ! *nn1*])
apply (*erule-tac t*=*is* # *take nn1 iss* **in** *ssubst*)
prefer *2*
apply (*erule take-append2*)
apply (*rule mapAccumL-one-more*,*simp*,*assumption*)

apply (*thin-tac drop ?n ?x* = *?y*)
apply (*thin-tac drop ?n ?x* = *?y*)
apply (*thin-tac drop ?n ?x* = *?y*)
apply (*thin-tac* (*take ?n ?x*)@*?z* = *?y*)
apply (*thin-tac ?x* < *?y*)
apply (*subgoal-tac cdm4* = *cdm1*((*p*, *nn1*) ↦ *pc1*,
(*p*, *Suc nn1*) ↦ *pc1* + *length y1*))

prefer *2* **apply** (*unfold trInstrAux-def*)
apply *force*
apply (*simp del*:*mapAccumL.simps trInstr.simps*)
apply (*fold fun-upd-apply*)
apply (*elim conjE*)
apply (*thin-tac mapAccumL ?f ?s ?x = ?y*)

apply (*subgoal-tac cdm1* $((p,\ nn1) \mapsto pc1,\ (p,\ Suc\ nn1) \mapsto pc1 +$ *length*
$(trInstr\ pc1\ (cdm1((p,\ nn1) \mapsto pc1))\ ctm\ com\ pcc\ ((is\ \#\ iss)\ !\ nn1)))$
$\subseteq_m cdm' \wedge (\forall\ j \geq n3.(p,j) \notin dom\ cdm')$)
prefer *2*
apply (*rule fun-trInstr1-aux*)
prefer *2* **apply** *simp*
prefer *2* **apply** (*elim conjE*,*assumption*)

apply (*subgoal-tac cdm* $\subseteq_m cdm1 \wedge (\forall\ j \geq nn1.(p,j) \notin dom\ cdm1)$)
prefer *2*
apply (*rule fun-trInstr1-aux*)
prefer *2* **apply** *simp*
apply (*erule-tac x=ja* **in** *allE*,*assumption*)
apply (*thin-tac ?y = mapAccumL ?f ?s ?x*)
apply (*thin-tac ?y = mapAccumL ?f ?s ?x*)
apply (*thin-tac ?y = mapAccumL ?f ?s ?x*)
apply (*thin-tac ?mapAccumL ?f ?s ?x = ?y*)
apply (*elim conjE*)
apply *clarify*
apply (*unfold fun-upd-apply*)
apply (*rotate-tac 1*)
apply (*erule-tac x=j* **in** *allE*)
apply (*drule mp*,*simp*)
apply *force*
done

lemma *fun-trInstr* :
$[\![\ ((pc',cdm'),bc') = trSeq\ ctm\ com\ pcc\ (pc,cdm)\ (p,svms,fn);$
$svms = is1\ \#\ iss;$
$(\forall\ j \geq 0\ .\ (p,j) \notin dom\ cdm);$
$drop\ pc\ bc = bc' @ rest;$
$svm = svms\ !\ i;$
$i < length\ svms$
$]\!] \Longrightarrow (\exists\ pc''\ cdm''\ bc''\ rest\ .$
$bc'' = trInstr\ pc''\ cdm''\ ctm\ com\ pcc\ svm$
$\wedge\ drop\ pc''\ bc = bc'' @ rest$
$\wedge\ cdm''\ (p,i) = Some\ pc''$
$\wedge\ cdm'' \subseteq_m cdm')$

apply (*insert fun-trInstrAux* [*of pc' cdm' bc' ctm com pcc pc cdm*
p svms fn is1 iss bc rest])

apply (*drule mp*,*assumption*)+
apply (*rotate-tac* −*1*)
apply (*erule-tac x*=*i* **in** *allE*)
apply (*drule mp*,*assumption*)
apply (*elim exE*, *elim conjE*)

apply (*rule-tac x*=*pc''* **in** *exI*)
apply (*rule-tac x*=*cdm''*((*p*, *i*) ↦ *pc''*) **in** *exI*)
apply (*rule-tac x*=*bc''* **in** *exI*)
apply (*rule-tac x*=*rest'* @ *rest* **in** *exI*)

apply (*rule conjI*)
apply (*unfold trInstrAux-def*)
apply *simp*

apply (*rule conjI*)
apply *clarsimp*
apply (*erule drop-append3*,*assumption*)
apply (*rule conjI*,*simp*)

by *assumption*

axioms *fun-trSeq1*:
(∃ *pc2 cdm2 bc2* . ((*pc2*,*cdm2*),*bc2*) = *trSeq ctm com pcc* (*pc1*,*cdm1*) *svm*
∧ *pc2*−*pc1*=*length bc2*)

axioms *fun-trSeq2*:
((*pc2*,*cdm2*),*bc2*) = *trSeq ctm com pcc* (*pc1*,*cdm1*) *seq*
⟹ *cdm1* ⊆$_m$ *cdm2*

lemma *fun-trSeq3* [*rule-format*]:
(∀ *pc1 cdm1 pc2 cdm2 bcs* .
((*pc2*,*cdm2*),*bcs*) = *mapAccumL* (*trSeq ctm com pcc*) (*pc1*,*cdm1*) *svms* ⟶
cdm1 ⊆$_m$ *cdm2*)

apply (*induct svms*)

apply *simp*

apply (*rule allI*)+
apply (*rename-tac seq svms pc1 cdm1 pc2 cdm2 bcs*)
apply (*unfold mapAccumL.simps*)
apply (*case-tac trSeq ctm com pcc* (*pc1*, *cdm1*) *seq*)
apply (*rename-tac state bc*)
apply (*case-tac state*, *rename-tac pc3 cdm3*,*simp*)

apply (*case-tac mapAccumL* (*trSeq ctm com pcc*) (*pc3*, *cdm3*) *svms*,*rename-tac state4 bcs4*)
apply (*case-tac state4*,*rename-tac pc4 cdm4*,*clarsimp*)
apply (*subgoal-tac* ((*pc4*, *cdm4*), *bcs4*)
    = *mapAccumL* (*trSeq ctm com pcc*) (*pc3*, *cdm3*) *svms*)
 prefer *2* **apply** *simp*
apply (*erule-tac x=pc3* **in** *allE*)
apply (*erule-tac x=cdm3* **in** *allE*)
apply (*erule-tac x=pc4* **in** *allE*)
apply (*erule-tac x=cdm4* **in** *allE*)
apply (*erule impE*)
apply (*rule-tac x=bcs4* **in** *exI*,*assumption*)
apply (*subgoal-tac cdm1* $\subseteq_m$ *cdm3*)
 prefer *2* **apply** (*rule fun-trSeq2*,*rule sym*,*simp*)
by (*erule map-le-trans*,*assumption*)

lemma *fun-trSeq-aux* :
 ((*pc*,*cdm*),*bcs*) = *mapAccumL* (*trSeq ctm com pcc*) (*pcini*,*cdini*) *svms* $\longrightarrow$
  drop pcini bc = *concat bcs* $\longrightarrow$
  ($\forall$ *i* . *i* < *length svms* $\longrightarrow$
  ($\exists$ *pc1 cdm1 pc2 cdm2 bcs1 bcs2 bc2* .
  ((*pc1*,*cdm1*),*bcs1*) = *mapAccumL* (*trSeq ctm com pcc*) (*pcini*,*cdini*) (*take i svms*)
  $\wedge$ ((*pc2*,*cdm2*),*bc2*) = *trSeq ctm com pcc* (*pc1*,*cdm1*) (*svms* ! *i*)
  $\wedge$ ((*pc*,*cdm*),*bcs2*) = *mapAccumL* (*trSeq ctm com pcc*) (*pc1*,*cdm1*) (*drop i svms*)
  $\wedge$ *bcs* = *bcs1* @ *bcs2*
  $\wedge$ (*pc1*−*pcini*) = *length* (*concat bcs1*)
  $\wedge$ *cdm2* $\subseteq_m$ *cdm*))

apply (*rule impI*)+
apply (*rule allI*)

apply (*induct-tac i*)

apply (*rule impI*)
apply (*rule-tac x=pcini* **in** *exI*)
apply (*rule-tac x=cdini* **in** *exI*)
apply (*subgoal-tac* ($\exists$ *pc2 cdm2 bc2* .
    ((*pc2*,*cdm2*),*bc2*) = *trSeq ctm com pcc* (*pcini*,*cdini*) (*svms* ! *0*)
    $\wedge$ *pc2*−*pcini*=*length bc2*))
 prefer *2*
 apply (*rule fun-trSeq1*)
apply (*elim exE*, *elim conjE*)
apply (*rule-tac x=pc2* **in** *exI*)
apply (*rule-tac x=cdm2* **in** *exI*)
apply (*rule-tac x=*[] **in** *exI*)
apply (*rule-tac x=bcs* **in** *exI*)
apply (*rule-tac x=bc2* **in** *exI*)

apply (*rule conjI*,*simp*)
apply (*rule conjI*,*assumption*)
apply (*rule conjI*,*simp*)
apply (*rule conjI*,*simp*)
apply (*rule conjI*,*simp*)
apply (*subgoal-tac* ∃ *s' y ys'*.
(*s'*,*y*) = *trSeq ctm com pcc* (*pcini*, *cdini*) (*svms*!*0*)
∧ ((*pc*, *cdm*),*ys'*) = *mapAccumL* (*trSeq ctm com pcc*) *s'* (*tl svms*))
prefer *2* **apply** (*erule mapAccumL-non-empty3*,*assumption*)
apply (*elim exE*,*elim conjE*)
apply *clarify* **apply** (*rename-tac pc2' cdm2' bc2' bcs'*)
apply (*subgoal-tac* ((*pc2'*, *cdm2'*), *bc2'*)=((*pc2*, *cdm2*), *bc2*))
prefer *2* **apply** *simp*
apply *clarify*
apply (*erule fun-trSeq3*)

apply (*rule impI*, *drule mp*, *simp*)
apply (*elim exE*,*elim conjE*)

apply (*subgoal-tac drop n svms* = (*svms* ! *n*) # *drop* (*Suc n*) *svms*)
apply *clarify*
prefer *2* **apply** (*rule drop-nth3*, *simp*)
apply (*subgoal-tac* ∃ *s' y ys'*.
(*s'*,*y*) = *trSeq ctm com pcc* (*pc1*, *cdm1*) (*svms* ! *n*)
∧ ((*pc*, *cdm*),*ys'*) = *mapAccumL* (*trSeq ctm com pcc*) *s'* (*drop* (*Suc n*) *svms*)
∧ *bcs2* = *y* # *ys'*)
prefer *2*
apply (*rule mapAccumL-non-empty2* [**where** *f*=*trSeq ctm com pcc*])
apply (*blast*,*simp*)
apply (*elim exE*,*elim conjE*)
apply (*case-tac s'*)
apply *clarify*
apply (*rename-tac bc2' bcs3 pc2' cdm2'*)
apply (*subgoal-tac* ((*pc2'*, *cdm2'*), *bc2'*)=((*pc2*, *cdm2*), *bc2*))
prefer *2* **apply** *simp*
apply *clarify*
apply (*thin-tac ?x*=*trSeq ?x1 ?x2 ?x3 ?x4 ?x5*)

apply (*rule-tac x*=*pc2* **in** *exI*)
apply (*rule-tac x*=*cdm2* **in** *exI*)
apply (*subgoal-tac* ∃ *pc2a cdm2a bc2a* .
((*pc2a*, *cdm2a*), *bc2a*) = *trSeq ctm com pcc* (*pc2*, *cdm2*) (*svms* ! *Suc n*)
∧ *pc2a* − *pc2* = *length bc2a*)
prefer *2* **apply** (*rule fun-trSeq1*)
apply (*elim exE*,*elim conjE*)

apply (*rule-tac x=pc2a* **in** *exI*)
apply (*rule-tac x=cdm2a* **in** *exI*)
apply (*rule-tac x=bcs1 @ [bc2]* **in** *exI*)
apply (*rule-tac x=bcs3* **in** *exI*)
apply (*rule-tac x=bc2a* **in** *exI*)

apply (*rule conjI*)
apply (*subgoal-tac take (Suc n) svms=(take n svms)@[svms ! n]*)
prefer *2*
apply (*rule take-append3,simp*)
apply (*erule-tac t=take (Suc n) svms* **in** *ssubst*)
apply (*erule mapAccumL-one-more,assumption*)

apply (*rule conjI,assumption*)

apply (*rule conjI,assumption*)

apply (*rule conjI, simp*)

apply (*subgoal-tac* ∃ *pc2b cdm2b bc2b* .
((*pc2b*, *cdm2b*), *bc2b*) = *trSeq ctm com pcc* (*pc1*, *cdm1*) (*svms* ! *n*)
∧ *pc2b* − *pc1* = *length bc2b*)
prefer *2*
apply (*rule fun-trSeq1*)
apply (*elim exE,elim conjE*)
apply (*subgoal-tac* ((*pc2*, *cdm2*), *bc2*)=((*pc2b*, *cdm2b*), *bc2b*))
prefer *2* **apply** *simp*
apply *clarify*
apply *simp*
apply (*subgoal-tac* (*pc1* − *pcini*)+(*pc2b* − *pc1*)=
(*length* (*concat bcs1*))+(*length bc2b*))
prefer *2*
apply (*erule sum-nat,assumption*)
apply (*rule conjI*)
apply (*rule sym*)
apply (*subgoal-tac length* (*concat bcs1*) + *length bc2b*
= *pc1* − *pcini* + (*pc2b* − *pc1*))
prefer *2* **apply** *simp*
apply (*erule-tac t=length* (*concat bcs1*) + *length bc2b* **in** *ssubst*)
apply (*rule sum-substract*)
apply (*rename-tac cdm3 bc3 pc2 cdm2 bc2*)

apply (*subgoal-tac* ∃ *s′ y ys′*.
    (*s′*,*y*) = *trSeq ctm com pcc* (*pc2*, *cdm2*) ((*drop* (*Suc n*) *svms*)!*0*)
    ∧ ((*pc*, *cdm*),*ys′*) =
      mapAccumL (*trSeq ctm com pcc*) *s′* (*tl* (*drop* (*Suc n*) *svms*)))
prefer *2* **apply** (*erule mapAccumL-non-empty3*)
apply *force*
apply (*elim exE*,*elim conjE*)
apply *clarify* **apply** (*rename-tac pc2′ cdm3′ bc3′ bcs′*)
apply (*subgoal-tac* ((*pc2′*, *cdm3′*), *bc3′*)=((*pc2a*, *cdm3*), *bc3*))
prefer *2* **apply** *simp*
apply *clarify*
apply (*erule fun-trSeq3*)
done

lemma *fun-trSeq* :
  ⟦ ((*pc*,*cdm*),*bcs*) = *mapAccumL* (*trSeq ctm com pcc*) (*pcini*,*cdini*) *svms*;
    drop pcini bc = *concat bcs*;
    l < *length svms*;
    seq = *svms* ! *l*
  ⟧ ⟹ (∃ *pc′ cdm′ pc″ cdm″ bc′ rest* .
        ((*pc″*,*cdm″*),*bc′*) = *trSeq ctm com pcc* (*pc′*,*cdm′*) *seq*
        ∧ *drop pc′ bc* = *bc′* @ *rest*
        ∧ *cdm″* ⊆$_m$ *cdm*)

apply (*insert fun-trSeq-aux* [*of pc cdm bcs ctm com pcc pcini cdini svms bc*])
apply (*drule mp*,*assumption*)+
apply (*erule-tac x*=*l* **in** *allE*)
apply (*drule mp*,*assumption*)
apply (*elim exE*, *elim conjE*)

apply (*rule-tac x*=*pc1* **in** *exI*)
apply (*rule-tac x*=*cdm1* **in** *exI*)
apply (*rule-tac x*=*pc2* **in** *exI*)
apply (*rule-tac x*=*cdm2* **in** *exI*)
apply (*rule-tac x*=*bc2* **in** *exI*)
apply (*rule-tac x*=*drop* (*pc2*−*pc1*) (*concat bcs2*) **in** *exI*)

apply (*rule conjI*, *simp*)

apply (*rule conjI*)
apply (*thin-tac ?x*=*mapAccumL ?y ?z ?u*)
apply (*thin-tac ?x*=*mapAccumL ?y ?z ?u*)
apply (*subgoal-tac drop l svms*=(*svms*!*l*) # *drop* (*Suc l*) *svms*)
prefer *2*
apply (*erule drop-nth3*, *clarify*)

apply (*case-tac trSeq ctm com pcc* (*pc1*, *cdm1*) (*svms* ! *l*), *clarsimp*)
apply (*rename-tac pc2 cdm2 bc2*)
apply (*case-tac mapAccumL* (*trSeq ctm com pcc*) (*pc2*,*cdm2*) (*drop* (*Suc l*) *svms*),*clarsimp*)
apply (*rename-tac bcs3*)
apply (*subgoal-tac* ∃ *pc2' cdm2' bc2'* .
((*pc2'*,*cdm2'*),*bc2'*) = *trSeq ctm com pcc* (*pc1*, *cdm1*) (*svms* ! *l*)
∧ *pc2'*−*pc1* = *length bc2'*)
prefer *2* **apply** (*rule fun-trSeq1*, *clarify*)
apply (*subgoal-tac* ((*pc2'*, *cdm2'*), *bc2'*)=((*pc2*, *cdm2*), *bc2*))
prefer *2* **apply** *simp*
apply *clarify*
apply *auto*
apply (*subgoal-tac drop pc1 bc* = [] @ [] @ (*bc2* @ *concat bcs3*))
prefer *2* **apply** (*rule drop-append3*)
apply (*simp*,*simp*,*simp add*: *drop-append*)
done

lemma *fun-trCodeStore*:
(*P'*,*cdm*,*ctm*,*com*) = *trSVM2JVM* ((*svms*, *ctmap*), *ini*, *ct*, *st*) ⟶
bc = *extractBytecode P'* ⟶
(∃ *cdm' ctm' pcc pcini cdini pc bcs* .
((*pc*,*cdm*),*bcs*) = *mapAccumL* (*trSeq ctm' com pcc*) (*pcini*,*cdini*) *svms*
∧ *drop pcini bc* = *concat bcs*)

apply (*rule impI*)+
apply (*unfold trSVM2JVM-def*)
apply *clarsimp*
apply (*case-tac trCodeStore ini* (*length* (*initConsTable ct*) +
length (*initSizeTable st*)) *ctmap* (*trConsTable ct*) *svms*)
apply (*rename-tac instrs1 rest*)
apply (*case-tac rest*)
apply (*rename-tac cdm1 ctm1*)
apply *clarsimp*
apply (*subgoal-tac extractBytecode*
((*safeP*, *objectC*, [], [(*sigSafeMain*, *PrimT Void*, *10*, *10*,
initConsTable ct @ *initSizeTable st* @ *instrs1*, [])]) # *Prog-RTS*) =
initConsTable ct @ *initSizeTable st* @ *instrs1*)
apply *simp*
apply (*thin-tac extractBytecode ?x=?y*)
prefer *2*
apply (*unfold extractBytecode-def*)
apply (*unfold method'-def*)
apply *simp*
apply (*subgoal-tac class* ((*safeP*,*objectC*,[],[(*sigSafeMain*,*PrimT Void*,
10,*10*,*initConsTable ct*@*initSizeTable st*@*instrs1*,[])]) # *Prog-RTS*)
safeP = *Some* (*objectC*, [], [(*sigSafeMain*, *PrimT Void*, *10*, *10*,
initConsTable ct @ *initSizeTable st* @ *instrs1*, [])]))
apply *simp*

apply (*unfold Prog-RTS-def*)
apply (*simp add*:*class-def*)

apply (*unfold trCodeStore-def*)
apply (*case-tac unzip3 ctmap*,*rename-tac fs1 rest*,
case-tac rest,*rename-tac ps1 contss1*)
apply *clarify* **apply** (*simp del*: *mapAccumL.simps trConsTable-def zip.simps upt-Suc*)
apply (*case-tac mapAccumL* (*trSeq* (*map-of* (*zip* (*concat contss1*)
[*Suc 0*..<*Suc* (*length* (*concat contss1*))])) (*trConsTable ct*)
(*Suc* (*length* (*initConsTable ct*) + *length* (*initSizeTable st*) + *length fs1*)))
(*Suc* (*Suc* (*length* (*initConsTable ct*) + *length* (*initSizeTable st*)
+ *length fs1*)), *map-of* (*zip* (*zip ps1* (*replicate* (*length fs1*) *0*))
[*Suc* (*length* (*initConsTable ct*) + *length* (*initSizeTable st*))..<
Suc (*length* (*initConsTable ct*) + *length* (*initSizeTable st*) +
length fs1)])) *svms*)
apply (*rename-tac rest iss1*,*case-tac rest*, *rename-tac pc1 cdm1*)
apply (*clarify*) **apply** (*simp del*: *mapAccumL.simps trConsTable-def zip.simps upt-Suc*)
apply (*elim conjE*,*clarify*)

apply (*rule-tac x*=*map-of* (*zip* (*concat contss1*)
[*Suc 0*..<*Suc* (*length* (*concat contss1*))]) **in** *exI*)
apply (*rule-tac x*=*Suc* (*length* (*initConsTable ct*) + *length* (*initSizeTable st*)
+ *length fs1*) **in** *exI*)
apply (*rule-tac x*=*Suc* (*Suc* (*length* (*initConsTable ct*)
+ *length* (*initSizeTable st*) + *length fs1*)) **in** *exI*)
apply (*rule-tac x*= *map-of* (*zip* (*zip ps1* (*replicate* (*length fs1*) *0*))
[*Suc* (*length* (*initConsTable ct*) + *length* (*initSizeTable st*))..<
Suc (*length* (*initConsTable ct*) + *length* (*initSizeTable st*)
+ *length fs1*)]) **in** *exI*)
apply (*rule-tac x*=*pc1* **in** *exI*)
apply (*rule-tac x*=*iss1* **in** *exI*)

apply (*rule conjI*)
apply *simp*

apply (*subgoal-tac length fs1* =*length ps1*)
prefer *2* **apply** (*rule unzip3-length*,*simp*)
apply (*subgoal-tac length* (*map Goto*
(*zipWith* (% *p n*. *int* (*the* (*cdm* (*p*, *0*))) − *int n*) *ps1*
[*Suc* (*length* (*initConsTable ct*) + *length* (*initSizeTable st*))..<
Suc (*length* (*initConsTable ct*) + *length* (*initSizeTable st*) + *length fs1*)]))
= *length fs1*)
prefer *2*
apply (*subgoal-tac length* [*Suc* (*length* (*initConsTable ct*) +

```
      length (initSizeTable st))..<
      Suc (length (initConsTable ct) + length (initSizeTable st))
      + length fs1] =
      (Suc (length (initConsTable ct) + length (initSizeTable st))
      + length fs1) -
      (Suc (length (initConsTable ct) + length (initSizeTable st))))
  prefer 2 apply(rule upt-length,simp)
 apply (subgoal-tac length (zipWith ( % p n. int (the (cdm (p, 0)))
      - int n) ps1
      [Suc (length (initConsTable ct) + length (initSizeTable st))..<
       Suc (length (initConsTable ct) + length (initSizeTable st)
       + length fs1)]) =
       min (length ps1) (length [Suc (length (initConsTable ct) +
       length (initSizeTable st))..<
       Suc (length (initConsTable ct) + length (initSizeTable st)
       + length fs1)]))
  prefer 2 apply (rule zipWith-length)
  apply simp
apply (simp add: drop-append)
done


axioms fun-SVM2JVM-aux:
 ⟦ l < length svms; svms ! l = (p, seq, fn);
   ((pc, cdm), bcs) = mapAccumL (trSeq ctm' com pcc) (pcini, cdini) svms;
   ((pc2, cdm2), bcSeq) = trSeq ctm' com pcc (pcSeq, cdmSeq) (svms ! l)
 ⟧ ⟹ (∀ j≥0. (p, j) ∉ dom cdmSeq)



lemma fun-SVM2JVM [rule-format]:
    (P', cdm, ctm, com) = trSVM2JVM ((svms, ctmap), ini, ct, st) ⟶
    l < length svms ⟶
    svms ! l = (p,seq,fn) ⟶
    i < length seq ⟶
    svm = fst (the (map-of svms p)) ! i ⟶
    pc = the (cdm (p,i)) ⟶
    bytecode = extractBytecode P' ⟶
    (∃ cdm' ctm' pcc inss bytecode' .
              inss = trInstr pc cdm' ctm' com pcc svm ∧
              drop pc bytecode = inss @ bytecode' ∧
              cdm' ⊆m cdm)

apply (rule impI)+


apply (insert fun-trCodeStore [of P' cdm ctm com svms ctmap ini ct st bytecode])
apply (drule mp, assumption)+
apply (erule exE)+
```

apply(*elim conjE*)

apply (*subgoal-tac* (∃ *pc′ cdm′ pc′′ cdm′′ bc′ rest* .
((*pc′′*,*cdm′′*),*bc′*) = *trSeq ctm′ com pcc* (*pc′*,*cdm′*) (*svms* ! *l*)
∧ *drop pc′ bytecode* = *bc′* @ *rest*
∧ *cdm′′* ⊆$_m$ *cdm*))
prefer *2* **apply** (*rule fun-trSeq*) **apply**(*simp*,*assumption*,*simp*,*simp*)
apply (*elim exE*, *elim conjE*)
apply *clarify*
apply (*rename-tac pcSeq cdmSeq pc2 cdm2 bcSeq restSeq*)

apply (*subgoal-tac* ∃ *is iss* . *seq* = *is*#*iss*)
prefer *2* **apply** (*rule list-non-empty*,*force*)
apply (*elim exE*)
apply (*subgoal-tac seq*=*fst* (*the* (*map-of svms p*)))
prefer *2* **apply** (*subgoal-tac map-of svms p*= *Some* (*seq*,*fn*))
prefer *2* **apply** (*rule map-of-distinct*)
apply (*rule svms-good*,*simp*,*simp*)
apply *simp*
apply (*insert svms-good* [*of svms*])
apply (*subgoal-tac* (∃ *pc′′ cdm′′ bc′′ rest* .
bc′′ = *trInstr pc′′ cdm′′ ctm′ com pcc* (*seq* ! *i*)
∧ *drop pc′′* (*extractBytecode P′*) = *bc′′* @ *rest*
∧ *cdm′′* (*p*,*i*) = *Some pc′′*
∧ *cdm′′* ⊆$_m$ *cdm2*))
prefer *2* **apply** (*rule fun-trInstr*)
apply (*simp*)
apply (*rule sym*)
apply *force*
apply (*erule fun-SVM2JVM-aux* [*of l svms p seq fn*],*assumption*,*simp*,*simp*)
apply *simp*
apply *simp*
apply *simp*
apply (*elim exE*, *elim conjE*)

apply (*rule-tac x*=*cdm′′* **in** *exI*)
apply (*rule-tac x*=*ctm′* **in** *exI*)
apply (*rule-tac x*=*pcc* **in** *exI*)
apply (*rule-tac x*=*bc′′* **in** *exI*)
apply (*rule-tac x*=*rest* **in** *exI*)
apply (*subgoal-tac cdm′′* ⊆$_m$ *cdm*)
prefer *2* **apply** (*erule map-le-trans*,*assumption*)
apply (*subgoal-tac the* (*cdm* (*p*,*i*))=*pc′′*)
prefer *2* **apply** (*unfold map-le-def*)
apply *force*
apply *clarify*

apply *simp*
done

consts
nonJumping :: *SafeInstr* ⇒ *bool*
primrec
nonJumping DECREGION = *True*
nonJumping (*SLIDE m n*) = *True*
nonJumping (*PRIMOP oper*) = *True*
nonJumping REUSE = *True*
nonJumping COPY = *True*
nonJumping (*PUSHCONT p*) = *True*
nonJumping (*BUILDENV its*) = *True*
nonJumping (*BUILDCLS c is i*) = *True*
nonJumping (*CALL p*) = *False*
nonJumping POPCONT = *False*
nonJumping (*MATCHN l v m ps*) = *False*
nonJumping (*MATCH l ps*) = *False*
nonJumping (*MATCHD l ps*) = *False*

axioms *nonJumping-Suc-pc* :
⟦ (*P'*, *cdm*, *ctm*, *com*) = *trSVM2JVM* ((*svms*, *ctmap*), *ini*, *ct*, *st*);
svm = *fst* (*the* (*map-of svms l*)) ! *i*;
nonJumping svm;
iss = *trInstr* (*the* (*cdm* (*l*, *i*))) *cdm' ctm' com pcc svm*
⟧ ⟹ *the* (*cdm* (*l*,*i*+*1*)) = *the* (*cdm* (*l*,*i*)) + *length iss*

end

25 Lemmas on the dynamic properties of the translation SVM to JVM

theory *dem-PUSHCONT*
imports *../JVMSAFE/JVMExec SVM2JVM SVMSemantics CertifSVM2JVM dem-translation*
begin

no-translations *Norm s* == (*None*,*s*)
no-translations *ex-table-of m* == *snd* (*snd* (*snd m*))

declare *trInstr.simps* [*simp del*]
declare *equivS.simps* [*simp del*]

axioms *maxPush-PUSHCONT*:
2 + *n* < *int m*

lemma *equivS-PUSHCONT-aux*:
equivS (*v1*#*S*) *S'* *n* *ctm* *d* *g*
⟹ *n* + *1* >= *0*
apply (*induct S*)
apply (*case-tac v1*,*simp-all*)
apply (*unfold equivS.simps*)
apply (*case-tac* (*nat n*), *simp*, *simp*)
apply (*case-tac nata*, *simp*, *simp*)
apply (*case-tac x*,*simp*)
apply (*simp add*: *equivS.simps*)
apply (*case-tac v1*,*simp-all*)
apply (*simp add*: *equivS.simps*)
apply (*simp add*: *equivS.simps*)
apply (*case-tac x*)
by (*simp add*: *equivS.simps*)

lemma *equivS-PUSHCONT*:
equivS (*drop* (*Suc 0*) *S*) *S'* *n* *ctm* *d* *g* ⟹ *n* + *1* >= *0*
apply (*induct S*)
apply (*simp add*: *equivS.simps*,*simp*)
apply (*case-tac S*,*simp*)
by (*simp*,*erule equivS-PUSHCONT-aux*)

lemma *equivS-good*:
equivS S S' *n* *ctm* *d* *g* ⟹ *1* ≤ *2* + *n*
apply (*induct S*)
apply (*unfold equivS.simps*, *simp*, *clarsimp*)
apply (*case-tac a*, *simp-all*)
apply (*simp add*: *equivS.simps*)
apply (*simp add*: *equivS.simps*)
apply (*case-tac x*)
by (*simp add*: *equivS.simps*)

axioms *equiv-ctm*:
the (*ctm p*) = *the* (*ctm' p*)

lemma *equivS-PUSHCONT-S2*:
⟦ *equivS S S' n ctm d g* ⟧
⟹ *equivS* (*Cont* (*nat* (*the-Intg* (*the* (*sha* (*heapC*, *k0f*)))), *p*) # *S*)
(*S'*(*nat* (*n* + *1*) ↦ *Intg* (*int* (*the* (*ctm' p*))), *nat* (*2* + *n*) ↦ *the* (*sha* (*heapC*, *k0f*)))) (*2* + *n*) *ctm d g*
apply (*frule equivS-good*)
apply (*unfold equivS.simps*)
apply (*rule conjI*, *simp*)
apply (*rule conjI*, *clarsimp*)
apply (*rule conjI*, *auto*)
apply (*rule equiv-ctm*)
apply (*drule equivS-ge-n*, *erule-tac x=1* **in** *allE*,*simp*)
apply (*subgoal-tac 1* + *n* = *n* + *1*,*simp*) **prefer** *2* **apply** *simp*

by (*drule equivS-ge-n*, *erule-tac x=2* **in** *allE*,*simp*)

lemma *activeCells-PUSHCONT*:
⟦ *la* ∉ *activeCells regS h* (*nat* (*the-Intg* (*the* (*sha* (*heapC*, *kf*)))))⟧
⟹ *activeCells regS h* (*nat* (*the-Intg* (*the* (*sha* (*heapC*, *kf*))))) =
activeCells regS (*h*(*la* ↦ *Arr ty m* (*S'*(*nat* (*n* + *1*) ↦ *Intg* (*int* (*the* (*ctm p*))), *nat* (*2* + *n*) ↦ *the* (*sha* (*heapC*, *k0f*))))))
(*nat* (*the-Intg* (*the* (*sha* (*heapC*, *kf*)))))
apply (*frule l-not-in-cellReg*)
apply (*unfold activeCells-def*,*auto*)
apply (*unfold region-def*, *simp add*: *Let-def*, *elim conjE*)
apply (*rule-tac x=j* **in** *exI*,*simp*)
apply (*rule cellReg-step*)
apply (*rule cellReg-basic*)
apply (*erule-tac x=j* **in** *allE*)
apply (*rule cellsReg-monotone-1*,*assumption*+)
apply (*erule l-not-in-regs*)
apply (*simp add*: *Let-def*, *elim conjE*)
apply (*rule-tac x=j* **in** *exI*, *simp*)
apply (*rule cellReg-step*)
apply (*rule cellReg-basic*)
apply (*erule-tac x=j* **in** *allE*)
apply (*erule cellsReg-monotone-2*,*assumption*+)
by (*erule l-not-in-regs*)

lemma *domH-PUSHCONT*:
⟦ *la* ∉ *activeCells regS h* (*nat* (*the-Intg* (*the* (*sha* (*heapC*, *kf*))))));
∀ *l*∈*dom H*. ∃ *l'*. *l'* = *the* (*g l*) ∧ *equivC* (*the* (*H l*)) *h l'* (*the* (*h l'*)) (*nat* (*the-Intg* (*the* (*sha* (*heapC*, *kf*))))) *com regS d g* ⟧
⟹ ∀ *l*∈*dom H*.
∃ *l'*. *l'* = *the* (*g l*) ∧
equivC (*the* (*H l*)) (*h*(*la* ↦ *Arr ty m* (*S'*(*nat* (*n* + *1*) ↦ *Intg* (*int* (*the* (*ctm p*))),
nat (*2* + *n*) ↦ *the* (*sha* (*heapC*, *k0f*))))))
l' (*the* ((*h*(*la* ↦ *Arr ty m* (*S'*(*nat* (*n* + *1*) ↦ *Intg* (*int* (*the* (*ctm p*))),
nat (*2* + *n*) ↦ *the* (*sha* (*heapC*, *k0f*))))))) *l'*))
(*nat* (*the-Intg* (*the* (*sha* (*heapC*, *kf*)))))
com regS d g
apply (*frule l-not-in-cellReg*)
apply (*rule ballI*)

apply (*erule-tac x=l* **in** *ballE*)
prefer *2* **apply** *simp*
apply (*erule exE*)
apply (*rule-tac x=l'* **in** *exI*) **apply** (*elim conjE*)
apply (*rule conjI*, *assumption*)
apply *clarsimp*
apply (*rule conjI*) **apply** (*rule impI*)
apply *clarsimp* **apply** (*simp add*: *region-def*)
apply *clarsimp*
apply (*rule-tac x=Obj* **in** *exI*)
apply (*rule-tac x=flds* **in** *exI*) **apply** *simp*
apply (*simp add*: *region-def*) **apply** (*elim conjE*)
apply (*rule cellReg-step*)
apply (*rule cellReg-basic*)
apply (*erule-tac x=j* **in** *allE*)
apply (*rule cellsReg-monotone-1*,*assumption*+)
by (*erule l-not-in-regs*)

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

lemma *equivH-PUSHCONT*:
⟦ ((*hm*, *k*), *k0*, (*l*, *i*), *S*) = ((*H*, *ka*), *k0a*, *PC*, *Sa*);
k' = *nat* (*the-Intg* (*the* (*sha* (*heapC*, *kf*))));
equivH (*H*, *ka*) *h k'* *com regS d g*;
activeCells regS h k' ∩ {*la*, *l'*, *l''*} = {}⟧
⟹ *equivH* (*hm*, *k*) (*h*(*la* ↦ *Arr ty m* (*S'*(*nat* (*n* + *1*) ↦ *Intg* (*int* (*the* (*ctm* *p*))),
nat (*2* + *n*) ↦ *the* (*sha* (*heapC*, *k0f*))))))
(*nat* (*the-Intg* (*the* ((*sha*((*stackC*, *topf*) ↦ *Intg* (*n* + *1* + *1*),
(*heapC*, *k0f*) ↦ *the* (*sha* (*heapC*, *kf*)))) (*heapC*, *kf*))))) *com regS d g*
apply *simp* **apply** (*elim conjE*)
apply (*frule l-not-in-cellReg*)
apply (*simp add*: *heapC-def add*: *stackC-def*)
apply (*simp add*: *kf-def add*: *k0f-def*)
apply (*fold heapC-def*, *fold kf-def*, *fold k0f-def*)
apply (*unfold equivH.simps*)
apply (*elim conjE*)
apply (*rule conjI*, *simp*)
apply (*rule conjI*) **apply** (*frule activeCells-PUSHCONT*,*simp*)
apply (*rule conjI*, *assumption*)
by (*rule domH-PUSHCONT*,*assumption*+)

declare *exec-instr.simps* [*simp*]

lemma *execSVMInstr-PUSHCONT* :
[| (*P'*, *cdm*, *ctm*, *com*) = *trSVM2JVM* ((*svms*, *ctmap*), *ini*, *ct*, *ah*, *ai*, *bc*);
cdm , *ctm*, *com* ⊢ ((*hm*, *k*), *k0*, (*l*, *i*), *S*) ≙ *S1'*;
(*fst* (*the* (*map-of svms l*)) ! *i*) = *PUSHCONT p*;
execSVMInst (*PUSHCONT p*) (*map-of ct*) (*hm*, *k*) *k0* (*l*, *i*) *S* = *Either.Right S2*;
drop (*the* (*cdm* (*l*, *i*))) (*extractBytecode P'*) =
trInstr (*the* (*cdm* (*l*, *i*))) *cdm' ctm' com pcc* (*PUSHCONT p*) @ *bytecode'*
|] ⟹ ∃ *v' sh' dh' ih' fms'* .
P' ⊢ *S1'* −*jvm*→ (*v'*,*sh'*,*dh'*,*ih'*, *fms'*) ∧
cdm , *ctm*, *com* ⊢ *S2* ≙ (*v'*,*sh'*,*dh'*,*ih'*, *fms'*)

apply (*case-tac S1'*)
apply (*rename-tac v tup*)
apply (*case-tac tup*)
apply (*rename-tac sh tup*)
apply (*case-tac tup*)
apply (*rename-tac dh tup*)
apply (*case-tac tup*)
apply (*rename-tac ih fms*)
apply (*simp*)

apply (*subgoal-tac* ∃ *S'*. *S* = *drop* (*Suc 0*) *S'*)
prefer *2* **apply** (*rule-tac x=snd* (*snd* (*snd S2*)) **in** *exI*, *clarsimp*)
apply (*unfold equivState-def*)
apply (*elim exE*,*elim conjE*)

apply (*rule-tac x=None* **in** *exI*)
apply (*rule-tac x=sha*((*stackC*, *topf*) ↦ *Intg* (*n* + *1* + *1*), (*heapC*, *k0f*) ↦ *the* (*sha* (*heapC*, *kf*))) **in** *exI*)
apply (*rule-tac x=h*(*la* ↦ *Arr ty m* (*S'a*(*nat* (*n* + *1*) ↦ *Intg* (*int* (*the* (*ctm' p*))), *nat* (*2* + *n*) ↦ *the* (*sha* (*heapC*, *k0f*))))) **in** *exI*)
apply (*rule-tac x=inih* **in** *exI*)
apply (*rule-tac x=*[([], *vs*, *safeP*, *sigSafeMain*, *the* (*cdm* (*l*, *i*)) +
1 + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1*, *ref*)] **in** *exI*)

apply (*subgoal-tac* ∃ *vs pc* . *fms*=([],*vs*,*safeP*,*sigSafeMain*,*pc*,*ref*)#[])
prefer *2* **apply** *simp*
apply (*erule exE*)+
apply (*unfold exec-all-def*)
apply (*rule conjI*)

apply (*subgoal-tac PC = (l,i)*) **prefer** *2* **apply** *simp*
apply *simp* **apply** (*elim conjE*) **apply** *clarsimp*
apply (*subgoal-tac P'⊢ (None,sha,h,inih,[([], vs, safeP, sigSafeMain, the (cdm (l,i)), ag,bd)]) −jvm→*
(None,sha,h,inih,[([the (sha (stackC,topf))], vs, safeP, sigSafeMain, the (cdm (l,i)) + 1, ag,bd)]))
apply (*unfold exec-all-def*)
apply (*erule rtrancl-trans*)

prefer *2*
apply (*rule r-into-rtrancl*)
apply (*clarify*)
apply (*unfold JVMExec.exec.simps*)
apply (*unfold Let-def*)
apply (*unfold extractBytecode-def*)
apply (*subgoal-tac (fst (snd (snd (snd (snd (the (method' (P', safeP) sigSafeMain))))))) !*
the (cdm (l, i))) = Getstatic topf stackC,simp)
apply (*unfold trInstr.simps, unfold Let-def*)
apply (*erule nth-via-drop-append*)

apply (*drule drop-Suc-append*)
apply (*subgoal-tac P'⊢ (None, sha, h, inih,*
[([the (sha (stackC, topf))],
vs, safeP, sigSafeMain, the (cdm (l, i)) + 1, ag, bd)])
−jvm→
(None,sha,h,inih,
[([Intg 1,
the (sha (stackC,topf))],
vs, safeP, sigSafeMain, the (cdm (l, i)) + 1 + 1, ag, bd)]))
apply (*unfold exec-all-def*)
apply (*erule rtrancl-trans*)
prefer *2*
apply (*rule r-into-rtrancl*)
apply (*clarify*)
apply (*unfold JVMExec.exec.simps*)
apply (*unfold Let-def*)
apply (*subgoal-tac (fst (snd (snd (snd (snd (the (method' (P', safeP) sigSafeMain))))))) ! Suc (the (cdm (l, i)))) =*
LitPush (Intg 1),simp)
apply (*erule nth-via-drop-append*)

apply (*drule drop-Suc-append*)
apply (*subgoal-tac P'⊢ (None,sha,h,inih,*
[([Intg 1,
the (sha (stackC,topf))],

vs, *safeP*, *sigSafeMain*, *the* (*cdm* (*l*, *i*)) + *1* + *1*, *ag*, *bd*)])
−jvm→
(*None*,*sha*,*h*,*inih*,
[([*Intg* (*the-Intg* (*the* (*sha* (*stackC*, *topf*))) + *1*)],
vs, *safeP*, *sigSafeMain*, *the* (*cdm* (*l*, *i*)) + *1* + *1* + *1*, *ag*, *bd*)]))
apply (*unfold exec-all-def*)
apply (*erule rtrancl-trans*)
prefer *2*
apply (*rule r-into-rtrancl*)
apply (*clarify*)
apply (*unfold JVMExec.exec.simps*)
apply (*unfold Let-def*)
apply (*subgoal-tac* (*fst* (*snd* (*snd* (*snd* (*snd* (*the* (*method′* (*P′*, *safeP*) *sigSafeMain*)))))) ! (*Suc* (*Suc* (*the* (*cdm* (*l*, *i*)))))) =
BinOp Add,*simp*)
apply (*erule nth-via-drop-append*)

apply (*drule drop-Suc-append*)
apply (*subgoal-tac P′⊢* (*None*,*sha*,*h*,*inih*,
[([*Intg* (*the-Intg* (*the* (*sha* (*stackC*, *topf*))) + *1*)],
vs, *safeP*, *sigSafeMain*, *the* (*cdm* (*l*, *i*)) + *1* + *1* + *1*, *ag*, *bd*)])
−jvm→
(*None*,*sha*((*stackC*, *topf*) *↦ Intg* (*n* + *1*)),*h*,*inih*,
[([],
vs, *safeP*, *sigSafeMain*, *the* (*cdm* (*l*, *i*)) + *1* + *1* + *1* + *1*, *ag*, *bd*)]))
apply (*unfold exec-all-def*)
apply (*erule rtrancl-trans*)
prefer *2*
apply (*rule r-into-rtrancl*)
apply (*clarify*)
apply (*unfold JVMExec.exec.simps*)
apply (*unfold Let-def*)
apply (*subgoal-tac* (*fst* (*snd* (*snd* (*snd* (*snd* (*the* (*method′* (*P′*, *safeP*) *sigSafeMain*)))))) ! (*Suc* (*Suc* (*Suc* (*the* (*cdm* (*l*, *i*))))))) =
Putstatic topf stackC,*simp*)
apply (*erule nth-via-drop-append*)

apply (*drule drop-Suc-append*)
apply (*subgoal-tac P′⊢* (*None*,*sha*((*stackC*, *topf*) *↦ Intg* (*n* + *1*)),*h*,*inih*,
[([],
vs, *safeP*, *sigSafeMain*, *the* (*cdm* (*l*, *i*)) + *1* + *1* + *1* + *1*, *ag*, *bd*)])
−jvm→

```
                    (None,sha((stackC, topf) ↦ Intg (n + 1)),h,inih,
                    [([the (sha (stackC, Sf))],
                     vs, safeP, sigSafeMain, the (cdm (l, i)) + 1 + 1 + 1 + 1 +
1, ag, bd)]))
apply (unfold exec-all-def)
apply (erule rtrancl-trans)
prefer 2
 apply (rule r-into-rtrancl)
 apply (clarify)
 apply (unfold JVMExec.exec.simps)
 apply (unfold Let-def)
 apply (subgoal-tac (fst (snd (snd (snd (snd (the (method' (P', safeP) sigSafe-
Main)))))) !
                  (Suc (Suc (Suc (Suc (the (cdm (l, i))))))) =
                   Getstatic Sf stackC,simp)
 apply (simp add: Sf-def add: topf-def)
apply (erule nth-via-drop-append)


apply (drule drop-Suc-append)
apply (subgoal-tac P'⊢ (None,sha((stackC, topf) ↦ Intg (n + 1)),h,inih,
                    [([the (sha (stackC, Sf))],
                     vs, safeP, sigSafeMain, the (cdm (l, i)) + 1 + 1 + 1 + 1 +
1, ag, bd)])
                 −jvm→
                    (None,sha((stackC, topf) ↦ Intg (n + 1)),h,inih,
                       [([Intg (the-Intg (the (sha (stackC, topf))) + 1), the (sha
(stackC, Sf))],
                     vs, safeP, sigSafeMain, the (cdm (l, i)) + 1 + 1 + 1 + 1 +
1 + 1, ag, bd)]))
apply (unfold exec-all-def)
apply (erule rtrancl-trans)
prefer 2
 apply (rule r-into-rtrancl)
 apply (clarify)
 apply (unfold JVMExec.exec.simps)
 apply (unfold Let-def)
 apply (subgoal-tac (fst (snd (snd (snd (snd (the (method' (P', safeP) sigSafe-
Main)))))) !
                  (Suc (Suc (Suc (Suc (Suc (the (cdm (l, i)))))))) =
                   Getstatic topf stackC,simp)
apply (erule nth-via-drop-append)


apply (drule drop-Suc-append)
apply (subgoal-tac P'⊢ (None,sha((stackC, topf) ↦ Intg (n + 1)),h,inih,
                       [([Intg (the-Intg (the (sha (stackC, topf))) + 1), the (sha
(stackC, Sf))],
                     vs, safeP, sigSafeMain, the (cdm (l, i)) + 1 + 1 + 1 + 1 +
```

```
1 + 1, ag, bd)])
                  −jvm→
                      (None,sha((stackC, topf) ↦ Intg (n + 1)),h,inih,
                      [([Intg (int (the (ctm′ p))),
                        Intg (the-Intg (the (sha (stackC, topf))) + 1),
                        the (sha (stackC, Sf))],
                      vs, safeP, sigSafeMain, the (cdm (l, i)) + 1 + 1 + 1 + 1 +
1 + 1 + 1, ag, bd)]))
apply (unfold exec-all-def)
apply (erule rtrancl-trans)
prefer 2
 apply (rule r-into-rtrancl)
 apply (clarify)
 apply (unfold JVMExec.exec.simps)
 apply (unfold Let-def)
 apply (subgoal-tac (fst (snd (snd (snd (snd (the (method′ (P′, safeP) sigSafe-
Main)))))) !
                   (Suc (Suc (Suc (Suc (Suc (Suc (the (cdm (l, i)))))))))) =
                    LitPush (Intg (int (the (ctm′ p)))),simp)
apply (erule nth-via-drop-append)



apply (drule drop-Suc-append)
apply (subgoal-tac P′⊢ (None,sha((stackC, topf) ↦ Intg (n + 1)),h,inih,
                      [([Intg (int (the (ctm′ p))),
                        Intg (the-Intg (the (sha (stackC, topf))) + 1),
                        the (sha (stackC, Sf))],
                      vs, safeP, sigSafeMain, the (cdm (l, i)) + 1 + 1 + 1 + 1 +
1 + 1 + 1, ag, bd)])
                  −jvm→
                      (None,sha((stackC, topf) ↦ Intg (n + 1)),
                        h(la ↦ Arr ty m (S′a(nat (n + 1) ↦ Intg (int (the (ctm′
p)))))),inih,
                      [([],
                      vs, safeP, sigSafeMain, the (cdm (l, i)) + 1 + 1 + 1 + 1 +
1 + 1 + 1 + 1, ag, bd)]))
apply (unfold exec-all-def)
apply (erule rtrancl-trans)
prefer 2
 apply (rule r-into-rtrancl)
 apply (clarify)
 apply (unfold JVMExec.exec.simps)
 apply (unfold Let-def)
 apply (subgoal-tac (fst (snd (snd (snd (snd (the (method′ (P′, safeP) sigSafe-
Main)))))) !
                   (Suc (Suc (Suc (Suc (Suc (Suc (the (cdm (l, i)))))))))) =
                    ArrStore,simp)
apply (simp add: raise-system-xcpt-def)
```

apply (*frule equivS-PUSHCONT*,*simp*)
apply (*subgoal-tac 2 + n < int m*,*simp*)
apply (*rule maxPush-PUSHCONT*)
apply (*erule nth-via-drop-append*)

apply (*drule drop-Suc-append*)
apply (*subgoal-tac P'⊢* (*None*,*sha*((*stackC*, *topf*) ↦ *Intg* (*n* + *1*)),
h(*la* ↦ *Arr ty m* (*S'a*(*nat* (*n* + *1*) ↦ *Intg* (*int* (*the* (*ctm'* *p*)))))),*inih*,
[([],
vs, *safeP*, *sigSafeMain*, *the* (*cdm* (*l*, *i*)) + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1*, *ag*, *bd*)])
−jvm→
(*None*,*sha*((*stackC*, *topf*) ↦ *Intg* (*n* + *1*)),
h(*la* ↦ *Arr ty m* (*S'a*(*nat* (*n* + *1*) ↦ *Intg* (*int* (*the* (*ctm'* *p*)))))),*inih*,
[([*Intg* (*n* + *1*)],
vs, *safeP*, *sigSafeMain*, *the* (*cdm* (*l*, *i*)) + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1*, *ag*, *bd*)]))
apply (*unfold exec-all-def*)
apply (*erule rtrancl-trans*)
prefer *2*
apply (*rule r-into-rtrancl*)
apply (*clarify*)
apply (*unfold JVMExec.exec.simps*)
apply (*unfold Let-def*)
apply (*subgoal-tac* (*fst* (*snd* (*snd* (*snd* (*snd* (*the* (*method'* (*P'*, *safeP*) *sigSafeMain*)))))))) !
(*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*the* (*cdm* (*l*, *i*))))))))))) =
Getstatic topf stackC,*simp*)
apply (*erule nth-via-drop-append*)

apply (*drule drop-Suc-append*)
apply (*subgoal-tac P'⊢* (*None*,*sha*((*stackC*, *topf*) ↦ *Intg* (*n* + *1*)),
h(*la* ↦ *Arr ty m* (*S'a*(*nat* (*n* + *1*) ↦ *Intg* (*int* (*the* (*ctm'* *p*)))))),*inih*,
[([*Intg* (*n* + *1*)],
vs, *safeP*, *sigSafeMain*, *the* (*cdm* (*l*, *i*)) + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1*, *ag*, *bd*)])
−jvm→
(*None*,*sha*((*stackC*, *topf*) ↦ *Intg* (*n* + *1*)),
h(*la* ↦ *Arr ty m* (*S'a*(*nat* (*n* + *1*) ↦ *Intg* (*int* (*the* (*ctm'* *p*)))))),*inih*,
[([*Intg 1*,

```
                    Intg (n + 1)],
                  vs, safeP, sigSafeMain, the (cdm (l, i)) + 1 + 1 + 1 + 1 +
1 + 1 + 1 + 1 + 1 + 1, ag, bd)]))
apply (unfold exec-all-def)
apply (erule rtrancl-trans)
prefer 2
 apply (rule r-into-rtrancl)
 apply (clarify)
 apply (unfold JVMExec.exec.simps)
 apply (unfold Let-def)
 apply (subgoal-tac (fst (snd (snd (snd (snd (the (method' (P', safeP) sigSafe-
Main)))))) !
                   (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (the (cdm (l,
i))))))))))))) =
                  LitPush (Intg 1),simp)
apply (erule nth-via-drop-append)


apply (drule drop-Suc-append)
apply (subgoal-tac P'⊢ (None,sha((stackC, topf) ↦ Intg (n + 1)),
                    h(la ↦ Arr ty m (S'a(nat (n + 1) ↦ Intg (int (the (ctm'
p)))))),inih,
                  [([Intg 1,
                     Intg (n + 1)],
                  vs, safeP, sigSafeMain, the (cdm (l, i)) + 1 + 1 + 1 + 1 +
1 + 1 + 1 + 1 + 1 + 1, ag, bd)])
               −jvm→
                  (None,sha((stackC, topf) ↦ Intg (n + 1)),
                    h(la ↦ Arr ty m (S'a(nat (n + 1) ↦ Intg (int (the (ctm'
p)))))),inih,
                  [([Intg (n + 1 + 1)],
                  vs, safeP, sigSafeMain, the (cdm (l, i)) + 1 + 1 + 1 + 1 +
1 + 1 + 1 + 1 + 1 + 1 + 1, ag, bd)]))
apply (unfold exec-all-def)
apply (erule rtrancl-trans)
prefer 2
 apply (rule r-into-rtrancl)
 apply (clarify)
 apply (unfold JVMExec.exec.simps)
 apply (unfold Let-def)
 apply (subgoal-tac (fst (snd (snd (snd (snd (the (method' (P', safeP) sigSafe-
Main)))))) !
                   (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (the (cdm
(l, i)))))))))))))) =
                  BinOp Add,simp)
apply (erule nth-via-drop-append)
```

apply (*drule drop-Suc-append*)
apply (*subgoal-tac P'⊢*(*None*,*sha*((*stackC*, *topf*) ↦ *Intg* (*n* + *1*)),
h(*la* ↦ *Arr ty m* (*S'a*(*nat* (*n* + *1*) ↦ *Intg* (*int* (*the* (*ctm'* *p*)))))),*inih*,
[([*Intg* (*n* + *1* + *1*)],
vs, *safeP*, *sigSafeMain*, *the* (*cdm* (*l*, *i*)) + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1*, *ag*, *bd*)])
−*jvm*→
(*None*,*sha*((*stackC*, *topf*) ↦ *Intg* (*n* + *1* + *1*)),
h(*la* ↦ *Arr ty m* (*S'a*(*nat* (*n* + *1*) ↦ *Intg* (*int* (*the* (*ctm'* *p*)))))),*inih*,
[([],
vs, *safeP*, *sigSafeMain*, *the* (*cdm* (*l*, *i*)) + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1*, *ag*, *bd*)]))
apply (*unfold exec-all-def*)
apply (*erule rtrancl-trans*)
prefer *2*
apply (*rule r-into-rtrancl*)
apply (*clarify*)
apply (*unfold JVMExec.exec.simps*)
apply (*unfold Let-def*)
apply (*subgoal-tac* (*fst* (*snd* (*snd* (*snd* (*snd* (*the* (*method'* (*P'*, *safeP*) *sigSafeMain*)))))) !
(*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*the* (*cdm* (*l*, *i*))))))))))))) =
Putstatic topf stackC,*simp*)
apply (*erule nth-via-drop-append*)

apply (*drule drop-Suc-append*)
apply (*subgoal-tac P'⊢* (*None*,*sha*((*stackC*, *topf*) ↦ *Intg* (*n* + *1* + *1*)),
h(*la* ↦ *Arr ty m* (*S'a*(*nat* (*n* + *1*) ↦ *Intg* (*int* (*the* (*ctm'* *p*)))))),*inih*,
[([],
vs, *safeP*, *sigSafeMain*, *the* (*cdm* (*l*, *i*)) + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1*, *ag*, *bd*)])
−*jvm*→
(*None*,*sha*((*stackC*, *topf*) ↦ *Intg* (*n* + *1* + *1*)),
h(*la* ↦ *Arr ty m* (*S'a*(*nat* (*n* + *1*) ↦ *Intg* (*int* (*the* (*ctm'* *p*)))))),*inih*,
[([*Addr la*],
vs, *safeP*, *sigSafeMain*, *the* (*cdm* (*l*, *i*)) + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1*, *ag*, *bd*)]))
apply (*unfold exec-all-def*)
apply (*erule rtrancl-trans*)
prefer *2*
apply (*rule r-into-rtrancl*)
apply (*clarify*)

apply (*unfold JVMExec.exec.simps*)
apply (*unfold Let-def*)
apply (*subgoal-tac* (*fst* (*snd* (*snd* (*snd* (*snd* (*the* (*method'* (*P'*, *safeP*) *sigSafeMain*))))))) !
(*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*the* (*cdm* (*l*, *i*))))))))))))))) =
Getstatic Sf stackC,*simp*)
apply (*simp add*: *Sf-def add*: *topf-def*)
apply (*erule nth-via-drop-append*)

apply (*drule drop-Suc-append*)
apply (*subgoal-tac P'*⊢ (*None*,*sha*((*stackC*, *topf*) ↦ *Intg* (*n* + *1* + *1*)),
h(*la* ↦ *Arr ty m* (*S'a*(*nat* (*n* + *1*) ↦ *Intg* (*int* (*the* (*ctm'* *p*)))))),*inih*,
[([*Addr la*],
vs, *safeP*, *sigSafeMain*, *the* (*cdm* (*l*, *i*)) + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1*, *ag*, *bd*)])
−*jvm*→
(*None*,*sha*((*stackC*, *topf*) ↦ *Intg* (*n* + *1* + *1*)),
h(*la* ↦ *Arr ty m* (*S'a*(*nat* (*n* + *1*) ↦ *Intg* (*int* (*the* (*ctm'* *p*)))))),*inih*,
[([*Intg* (*n* + *1* + *1*),
Addr la],
vs, *safeP*, *sigSafeMain*, *the* (*cdm* (*l*, *i*)) + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1*, *ag*, *bd*)]))
apply (*unfold exec-all-def*)
apply (*erule rtrancl-trans*)
prefer *2*
apply (*rule r-into-rtrancl*)
apply (*clarify*)
apply (*unfold JVMExec.exec.simps*)
apply (*unfold Let-def*)
apply (*subgoal-tac* (*fst* (*snd* (*snd* (*snd* (*snd* (*the* (*method'* (*P'*, *safeP*) *sigSafeMain*))))))) !
(*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*the* (*cdm* (*l*, *i*))))))))))))))) =
Getstatic topf stackC,*simp*)
apply (*erule nth-via-drop-append*)

apply (*drule drop-Suc-append*)
apply (*subgoal-tac P'*⊢ (*None*,*sha*((*stackC*, *topf*) ↦ *Intg* (*n* + *1* + *1*)),
h(*la* ↦ *Arr ty m* (*S'a*(*nat* (*n* + *1*) ↦ *Intg* (*int* (*the* (*ctm'* *p*)))))),*inih*,
[([*Intg* (*n* + *1* + *1*),
Addr la],
vs, *safeP*, *sigSafeMain*, *the* (*cdm* (*l*, *i*)) + *1* + *1* + *1* + *1* +

1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1, ag, bd)])
−jvm→
(*None*,*sha*((*stackC*, *topf*) ↦ *Intg* (*n* + *1* + *1*)),
h(*la* ↦ *Arr ty m* (*S′a*(*nat* (*n* + *1*) ↦ *Intg* (*int* (*the* (*ctm′*
p))))))),*inih*,
[([*the* (*sha* (*heapC*,*k0f*)),
Intg (*n* + *1* + *1*),
Addr la],
vs, *safeP*, *sigSafeMain*, *the* (*cdm* (*l*, *i*)) + *1* + *1* + *1* + *1* +
1 + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1*, *ag*, *bd*)]))
apply (*unfold exec-all-def*)
apply (*erule rtrancl-trans*)
prefer *2*
apply (*rule r-into-rtrancl*)
apply (*clarify*)
apply (*unfold JVMExec.exec.simps*)
apply (*unfold Let-def*)
apply (*subgoal-tac* (*fst* (*snd* (*snd* (*snd* (*snd* (*the* (*method′* (*P′*, *safeP*) *sigSafe-*
Main))))))) !
(*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc*
(*Suc* (*Suc* (*the* (*cdm* (*l*, *i*)))))))))))))))) =
Getstatic k0f heapC,*simp*)
apply (*simp add*: *heapC-def add*: *stackC-def add*: *k0f-def add*: *topf-def*)
apply (*erule nth-via-drop-append*)

apply (*drule drop-Suc-append*)
apply (*subgoal-tac P′*⊢ (*None*,*sha*((*stackC*, *topf*) ↦ *Intg* (*n* + *1* + *1*)),
h(*la* ↦ *Arr ty m* (*S′a*(*nat* (*n* + *1*) ↦ *Intg* (*int* (*the* (*ctm′*
p))))))),*inih*,
[([*the* (*sha* (*heapC*,*k0f*)),
Intg (*n* + *1* + *1*),
Addr la],
vs, *safeP*, *sigSafeMain*, *the* (*cdm* (*l*, *i*)) + *1* + *1* + *1* + *1* +
1 + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1*, *ag*, *bd*)])
−jvm→
(*None*,*sha*((*stackC*, *topf*) ↦ *Intg* (*n* + *1* + *1*)),
h(*la* ↦ *Arr ty m* (*S′a*(*nat* (*n* + *1*) ↦ *Intg* (*int* (*the* (*ctm′*
p))), *nat* (*2* + *n*) ↦ *the* (*sha* (*heapC*, *k0f*))))),
inih,
[([],
vs, *safeP*, *sigSafeMain*,
the (*cdm* (*l*, *i*)) + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* +
1 + *1* + *1* + *1* + *1* + *1* + *1*, *ag*, *bd*)]))
apply (*unfold exec-all-def*)
apply (*erule rtrancl-trans*)
prefer *2*
apply (*rule r-into-rtrancl*)
apply (*clarify*)

apply (*unfold JVMExec.exec.simps*)
apply (*unfold Let-def*)
apply (*subgoal-tac* (*fst* (*snd* (*snd* (*snd* (*snd* (*the* (*method'* (*P'*, *safeP*) *sigSafeMain*))))))) !
(*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*the* (*cdm* (*l*, *i*))))))))))))))))))) =
ArrStore,*simp*)
apply (*simp add*: *raise-system-xcpt-def*)
apply (*subgoal-tac int m* > *2* + *n*,*simp*)
prefer *2* **apply** (*rule maxPush-PUSHCONT*)
apply (*rule conjI*, *rule impI*)
apply (*frule equivS-PUSHCONT*,*simp*)
apply (*rule impI*)
apply (*frule equivS-PUSHCONT*,*simp*)
apply (*erule nth-via-drop-append*)

apply (*drule drop-Suc-append*)
apply (*subgoal-tac P'*⊢ (*None*,*sha*((*stackC*, *topf*) ↦ *Intg* (*n* + *1* + *1*)),
h(*la* ↦ *Arr ty m* (*S'a*(*nat* (*n* + *1*) ↦ *Intg* (*int* (*the* (*ctm'* *p*))), *nat* (*2* + *n*) ↦ *the* (*sha* (*heapC*, *k0f*))))),
inih,
[([],
vs, *safeP*, *sigSafeMain*,
the (*cdm* (*l*, *i*)) + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1*, *ag*, *bd*)])
−*jvm*→
(*None*,*sha*((*stackC*, *topf*) ↦ *Intg* (*n* + *1* + *1*)),
h(*la* ↦ *Arr ty m* (*S'a*(*nat* (*n* + *1*) ↦ *Intg* (*int* (*the* (*ctm'* *p*))), *nat* (*2* + *n*) ↦ *the* (*sha* (*heapC*, *k0f*))))),
inih,
[([*the* (*sha* (*heapC*, *kf*))],
vs, *safeP*, *sigSafeMain*,
the (*cdm* (*l*, *i*)) + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1*, *ag*, *bd*)]))
apply (*unfold exec-all-def*)
apply (*erule rtrancl-trans*)
prefer *2*
apply (*rule r-into-rtrancl*)
apply (*clarify*)
apply (*unfold JVMExec.exec.simps*)
apply (*unfold Let-def*)
apply (*subgoal-tac* (*fst* (*snd* (*snd* (*snd* (*snd* (*the* (*method'* (*P'*, *safeP*) *sigSafeMain*))))))) !
(*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*the* (*cdm* (*l*, *i*))))))))))))))))))) =
Getstatic kf heapC,*simp*)
apply (*simp add*: *heapC-def add*: *stackC-def add*: *kf-def add*: *topf-def*)

apply (*erule nth-via-drop-append*)

apply (*drule drop-Suc-append*)
apply (*subgoal-tac P'⊢* (*None*,*sha*((*stackC*, *topf*) ↦ *Intg* (*n* + *1* + *1*)),
h(*la* ↦ *Arr ty m* (*S'a*(*nat* (*n* + *1*) ↦ *Intg* (*int* (*the* (*ctm'* *p*))), *nat* (*2* + *n*) ↦ *the* (*sha* (*heapC*, *k0f*))))),
inih,
[([*the* (*sha* (*heapC*, *kf*))],
vs, *safeP*, *sigSafeMain*,
the (*cdm* (*l*, *i*)) + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1*, *ag*, *bd*)])
−jvm→
(*None*,*sha*((*stackC*, *topf*) ↦ *Intg* (*n* + *1* + *1*), (*heapC*, *k0f*) ↦ *the* (*sha* (*heapC*, *kf*))),
h(*la* ↦ *Arr ty m* (*S'a*(*nat* (*n* + *1*) ↦ *Intg* (*int* (*the* (*ctm'* *p*))), *nat* (*2* + *n*) ↦ *the* (*sha* (*heapC*, *k0f*))))),
inih,
[([],
vs, *safeP*, *sigSafeMain*,
the (*cdm* (*l*, *i*)) + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1*, *ag*, *bd*)]))
apply (*unfold exec-all-def*)
apply (*erule rtrancl-trans*)
prefer *2*
apply (*rule r-into-rtrancl*)
apply (*clarify*)
apply (*unfold JVMExec.exec.simps*)
apply (*unfold Let-def*)
apply (*subgoal-tac* (*fst* (*snd* (*snd* (*snd* (*snd* (*the* (*method'* (*P'*, *safeP*) *sigSafeMain*))))))) !
(*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc*
(*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*the* (*cdm* (*l*, *i*))))))))))))))))))))) =
Putstatic k0f heapC,*simp*)
apply (*erule nth-via-drop-append*)
apply *simp*

apply (*frule nonJumping-Suc-pc*)
apply (*erule-tac sym* [**where** *t*=*PUSHCONT p*])
apply (*simp add*: *nonJumping.simps*)
apply (*simp add*: *trInstr.simps*)

apply (*rule-tac x*=*hm* **in** *exI*)
apply (*rule-tac x*=*k* **in** *exI*)

apply (*rule-tac x=k* **in** *exI*)
apply (*rule-tac x=(l,Suc i)* **in** *exI*)
apply (*rule-tac x=Cont (k0, p) # S* **in** *exI*)

apply (*rule-tac x= sha((stackC, topf) ↦ Intg (n + 1 + 1), (heapC, k0f) ↦ the (sha (heapC, kf)))* **in** *exI*)
apply (*rule-tac x= h(la ↦ Arr ty m (S'a(nat (n + 1) ↦ Intg (int (the (ctm' p))),*
nat (2 + n) ↦ the (sha (heapC, k0f))))) **in** *exI*)
apply (*rule-tac x=inih* **in** *exI*)
apply (*rule-tac x=vs* **in** *exI*)
apply (*rule-tac x=the (cdm (l, i)) + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1* **in** *exI*)
apply (*rule-tac x=ref* **in** *exI*)

apply (*rule-tac x=nat (the-Intg (the ((sha((stackC, topf) ↦ Intg (n + 1 + 1),*
(heapC, k0f) ↦ the (sha (heapC, kf))))
(heapC, kf)))) **in** *exI*)

apply (*rule-tac x=nat (the-Intg (the ((sha((stackC, topf) ↦ Intg (n + 1 + 1),*
(heapC, k0f) ↦ the (sha (heapC, kf))))
(heapC, k0f)))) **in** *exI*)

apply (*rule-tac x=la* **in** *exI*)
apply (*rule-tac x=ty* **in** *exI*)
apply (*rule-tac x=m* **in** *exI*)
apply (*rule-tac x=S'a(nat (n + 1) ↦ Intg (int (the (ctm' p))),*
nat (2 + n) ↦ the (sha (heapC, k0f))) **in** *exI*)
apply (*rule-tac x=(2 + n)* **in** *exI*)
apply (*rule-tac x=l'* **in** *exI*)
apply (*rule-tac x=regS* **in** *exI*)
apply (*rule-tac x=l''* **in** *exI*)
apply (*rule-tac x=m'* **in** *exI*)
apply (*rule-tac x=m''* **in** *exI*)
apply (*rule-tac x=d* **in** *exI*)
apply (*rule-tac x=g* **in** *exI*)

apply (*rule conjI*, *drule-tac t=S2* **in** *sym*, *assumption*)
apply (*rule conjI*, *rule refl*)
apply (*rule conjI*, *rule refl*)
apply (*rule conjI*, *rule refl*)
apply (*rule conjI*) **apply** *clarsimp*
apply (*simp add*: *Sf-def add*: *topf-def*)
apply (*simp add*: *stackC-def add*: *heapC-def*)
apply (*rule conjI*, *assumption*)
apply (*rule conjI*) **apply** *clarsimp*
apply (*simp add*: *stackC-def add*: *heapC-def*)
apply (*simp add*: *kf-def add*: *k0f-def*)
apply (*rule conjI*) **apply** (*erule activeCells*)

apply (*rule conjI*) **apply** (*erule activeCells-2*,*assumption*)
apply (*erule activeCells-2*,*assumption*)
apply (*rule conjI*, *simp*)
apply (*rule conjI*) **apply** *clarsimp*
apply (*simp add*: *stackC-def add*: *heapC-def*)
apply (*rule conjI*) **apply** *clarsimp*
apply (*simp add*: *stackC-def add*: *heapC-def*)
apply (*simp add*: *regionsf-def add*: *k0f-def*)
apply (*rule conjI*) **apply** *clarsimp*
apply (*rule conjI*) **apply** *clarsimp*
apply (*simp add*: *dirCellC-def add*: *stackC-def*)
apply (*simp add*: *heapC-def add*: *safeDirf-def*)
apply (*rule conjI*) **apply** *clarsimp*
apply (*rule conjI*) **apply** *clarsimp*
apply (*rule conjI*) **apply** *clarsimp*
apply (*rule conjI*) **apply** (*rule equivH-PUSHCONT*,*assumption*+)
apply (*rule conjI*) **apply** (*rule maxPush-PUSHCONT*)
apply (*rule conjI*) **apply** (*simp*, *rule equivS-PUSHCONT-S2*,*simp*)
apply (*rule conjI*) **apply** (*unfold equivH.simps*,*simp*)
by *simp*

end

theory *dem-POPCONT*
imports *../JVMSAFE/JVMExec SVM2JVM SVMSemantics CertifSVM2JVM*
begin

no-translations *Norm s* == (*None*,*s*)

no-translations *ex-table-of m* == *snd* (*snd* (*snd m*))

declare *trInstr.simps* [*simp del*]
declare *equivS.simps* [*simp del*]

axioms *maxPush-POPCONT*:
2 + *n* < *int m*

lemma *equivS-POPCONT*:
equivS (*v1* # *Cont* (*k1*, *l1*) # *S*) *S'* *n* *ctm* *d* *g* ⟹ *n* >= *2*
apply (*induct S*)
apply (*case-tac v1*,*simp-all*)
apply (*unfold equivS.simps*)
apply (*case-tac* (*nat n*), *simp*, *simp*)
apply (*case-tac nata*, *simp*, *simp*)
apply (*case-tac x*,*simp*)
apply (*simp add*: *equivS.simps*)
apply (*case-tac v1*,*simp-all*)
apply (*simp add*: *equivS.simps*)
apply (*simp add*: *equivS.simps*)
apply (*case-tac x*)
by (*simp add*: *equivS.simps*)

lemma *equivS2-POPCONT*:
⟦ *2* < *length vs*; *equivS* (*Val v1* # *Cont* (*k1*, *l1*) # *S'*) *S'a* *n* *ctm* *d* *g*⟧
⟹ *equivS* (*Val v1* # *S'*) (*S'a*(*nat* (*n* − *2*) ↦ *vs*[*Suc 0* := *the* (*S'a* (*nat n*))] !
Suc 0)) (*n* − *2*) *ctm* *d* *g*
apply (*simp add*: *equivS.simps*, *clarsimp*)
apply (*drule equivS-ge-n*)
apply (*erule-tac x=1* **in** *allE*,*simp*)
by (*subgoal-tac* −*2* + *n*= *n* − *2*,*simp*,*simp*)

lemma *equivS-k1*:
⟦ *equivS* (*Val v1* # *Cont* (*k1*, *l1*) # *S'*) *S'a* *n* *ctm* *d* *g*⟧
⟹ *k1* = *nat* (*the-Intg* (*the* (*S'a* (*nat* (*n* − *1*)))))
by (*simp add*: *equivS.simps*)

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

lemma *activeCells-POPCONT*:
⟦ *la* ∉ *activeCells regS h* (*nat* (*the-Intg* (*the* (*sha* (*heapC*, *kf*)))))⟧
⟹ *activeCells regS h* (*nat* (*the-Intg* (*the* (*sha* (*heapC*, *kf*))))) =
activeCells regS (*h*(*la* ↦ *Arr ty m* (*S'a*(*nat* (*n* − *2*) ↦ *the* (*S'a* (*nat n*))))))
(*nat* (*the-Intg* (*the* (*sha* (*heapC*, *kf*)))))
apply (*frule l-not-in-cellReg*)
apply (*unfold activeCells-def*,*auto*)
apply (*unfold region-def*, *simp add*: *Let-def*, *elim conjE*)
apply (*rule-tac x=j* **in** *exI*,*simp*)
apply (*rule cellReg-step*)
apply (*rule cellReg-basic*)
apply (*erule-tac x=j* **in** *allE*)
apply (*rule cellsReg-monotone-1*,*assumption*+)
apply (*erule l-not-in-regs*)
apply (*simp add*: *Let-def*, *elim conjE*)

apply (*rule-tac x=j* **in** *exI*, *simp*)
apply (*rule cellReg-step*)
apply (*rule cellReg-basic*)
apply (*erule-tac x=j* **in** *allE*)
apply (*erule cellsReg-monotone-2*,*assumption*+)
by (*erule l-not-in-regs*)

lemma *domH-POPCONT*:
⟦ *la* ∉ *activeCells regS h* (*nat* (*the-Intg* (*the* (*sha* (*heapC*, *kf*))))));
∀ *l*∈*dom H*. ∃ *l'*. *l'* = *the* (*g l*) ∧ *equivC* (*the* (*H l*)) *h l'* (*the* (*h l'*)) (*nat* (*the-Intg* (*the* (*sha* (*heapC*, *kf*))))) *com regS d g* ⟧
⟹ ∀ *l*∈*dom H*.
∃ *l'*. *l'* = *the* (*g l*) ∧
equivC (*the* (*H l*)) (*h*(*la* ↦ *Arr ty m* (*S'a*(*nat* (*n* − *2*) ↦ *the* (*S'a* (*nat n*)))))) *l'*
(*the* ((*h*(*la* ↦ *Arr ty m* (*S'a*(*nat* (*n* − *2*) ↦ *the* (*S'a* (*nat n*)))))) *l'*))
(*nat* (*the-Intg* (*the* (*sha* (*heapC*, *kf*)))))
com regS d g
apply (*frule l-not-in-cellReg*)
apply (*rule ballI*)
apply (*erule-tac x=l* **in** *ballE*)
prefer *2* **apply** *simp*
apply (*erule exE*)
apply (*rule-tac x=l'* **in** *exI*) **apply** (*elim conjE*)
apply (*rule conjI*, *assumption*)
apply *clarsimp*
apply (*rule conjI*, *rule impI*)
apply (*simp add*: *region-def*)
apply (*rule impI*)
apply (*rule-tac x=Obj* **in** *exI*)
apply (*rule-tac x=flds* **in** *exI*) **apply** *simp*
apply (*simp add*: *region-def*)
apply *clarsimp*
apply (*rule cellReg-step*)
apply (*rule cellReg-basic*)
apply (*erule-tac x=j* **in** *allE*)
apply (*rule cellsReg-monotone-1*,*assumption*+)
by (*erule l-not-in-regs*)

lemma *equivH-POPCONT*:
⟦((*hm*, *k*), *k0*, (*l*, *i*), *v1* # *Cont* (*k1*, *l1*) # *S'*) = ((*H*, *ka*), *k0a*, *PC*, *S*);
k' = *nat* (*the-Intg* (*the* (*sha* (*heapC*, *kf*)))); *k0'* = *nat* (*the-Intg* (*the* (*sha* (*heapC*, *k0f*)))); *sha* (*stackC*, *Sf*) = *Some* (*Addr la*);

distinct [*la*,*l'*,*l''*]; *h la* = *Some* (*Arr ty m S'a*); *sha* (*stackC*, *topf*) = *Some* (*Intg n*);
sha (*heapC*, *regionsf*) = *Some* (*Addr l'*); *h l'* = *Some* (*Arr ty m' regS*); *sha* (*dirCellC*, *safeDirf*) = *Some* (*Addr l''*);
h l'' = *Some* (*Arr ty m'' d*); *inj-on g* (*dom H*); *equivH* (*H*, *ka*) *h k' com regS d g*; *k0a* = *k0'*;
activeCells regS h k' ∩ {*la*, *l'*, *l''*} = {}; *2* < *length vs*⟧
⟹ *equivH* (*hm*, *k*) (*h*(*la* ↦ *Arr ty m* (*S'a*(*nat* (*n* − *2*) ↦ *vs*[*Suc 0* := *the* (*S'a* (*nat n*))] ! *Suc 0*))))
(*nat* (*the-Intg*
(*the* ((*sha*((*stackC*, *topf*) ↦ *Intg* (*n* − *2*), (*heapC*, *k0f*) ↦
vs[*Suc 0* := *the* (*S'a* (*nat n*)), *2* := *the* (*S'a* (*nat* (*n* − *1*)))] ! *2*))
(*heapC*, *kf*)))))
com regS d g
apply (*simp add*: *heapC-def add*: *stackC-def*)
apply (*simp add*: *kf-def add*: *k0f-def*)
apply (*fold heapC-def*, *fold kf-def*)
apply (*unfold equivH.simps*, *elim conjE*)
apply (*rule conjI*, *assumption*)
apply (*rule conjI*, *frule activeCells-POPCONT*,*simp*)
apply (*rule conjI*, *assumption*)
by (*rule domH-POPCONT*,*assumption*+)

declare *extractBytecode-def* [*simp del*]
declare *initConsTable-def* [*simp del*]
declare *equivS.simps* [*simp del*]

axioms *Tableswitch-POPCONT*:
drop pcc (*extractBytecode P'*) =
[*Tableswitch 1* (*int* (*length ps*)) *ps*] @ *bytecode'* ∧ *ctm* (*nat* (*ps*!*i*)) = *Some i*

axioms *equivpc-POPCONT*:
if' (*the-Intg* (*vs*[*Suc 0* := *the* (*S'a* (*nat n*)),
2 := *the* (*S'a* (*nat* (*n* − *1*))),
3 := *the* (*S'a* (*nat* (*n* − *2*)))] ! *3*) < *1* ∨
int (*length ps*) < *the-Intg* (*vs*[*Suc 0* := *the* (*S'a* (*nat n*)),
2 := *the* (*S'a* (*nat* (*n* − *1*))),
3 := *the* (*S'a* (*nat* (*n* − *2*)))] ! *3*))
(*nat* (*24* + *int* (*the* (*cdm* (*l*, *i*))) + *trAddr pcc* (*the* (*cdm* (*l*, *i*)) + *incPop*)) + *nat* (*ps* ! *nat* (*int* (*length ps*))))
(*nat* (*24* + *int* (*the* (*cdm* (*l*, *i*))) + *trAddr pcc* (*the* (*cdm* (*l*, *i*)) + *incPop*)) +
nat (*ps* ! *nat* (*the-Intg* (*vs*[*Suc 0* := *the* (*S'a* (*nat n*)),
2 := *the* (*S'a* (*nat* (*n* − *1*))),
3 := *the* (*S'a* (*nat* (*n* − *2*)))] ! *3*) − *1*))) =
the (*cdm* (*l1*, *0*))

constdefs *if'* :: [*bool*, *'a*, *'a*] => *'a*
if' c a b ≡ *if c then a else b*

declare *if'-def* [*simp del*]

lemma *execSVMInstr-POPCONT* :
⟦ (*P'*, *cdm*, *ctm*, *com*) = *trSVM2JVM* ((*svms*, *ctmap*), *ini*, *ct*, *ah*, *ai*, *bc*);
cdm , *ctm*, *com* ⊢ ((*hm*, *k*), *k0*, (*l*, *i*), *S*) ≙ *S1'*;
(*fst* (*the* (*map-of svms l*)) ! *i*) = *POPCONT*;
execSVMInst POPCONT (*map-of ct*) (*hm*, *k*) *k0* (*l*, *i*) *S* = *Either.Right S2*;
drop (*the* (*cdm* (*l*, *i*))) (*extractBytecode P'*) =
trInstr (*the* (*cdm* (*l*, *i*))) *cdm' ctm' com pcc POPCONT* @ *bytecode'*
⟧ ⟹ ∃ *v' sh' dh' ih' fms'* .
P' ⊢ *S1'* −*jvm*→ (*v'*,*sh'*,*dh'*,*ih'*, *fms'*) ∧
cdm , *ctm*, *com* ⊢ *S2* ≙ (*v'*,*sh'*,*dh'*,*ih'*, *fms'*)

apply (*case-tac S1'*)
apply (*rename-tac v tup*)
apply (*case-tac tup*)
apply (*rename-tac sh tup*)
apply (*case-tac tup*)
apply (*rename-tac dh tup*)
apply (*case-tac tup*)
apply (*rename-tac ih fms*)
apply (*simp*)
apply (*subgoal-tac* ∃ *v1 k1 l1 S'*. *S*=*Val v1*#*Cont* (*k1*,*l1*)#*S'*,*clarsimp*)

prefer *2*
apply (*case-tac S*)
apply *simp*
apply (*insert RightNotUndefined*)
apply (*erule-tac x*= *S2* **in** *allE*, *force*)
apply *simp*
apply (*case-tac a*)
apply *simp*
apply (*case-tac list*)
apply *simp*
apply *simp*
apply (*case-tac aa*)
apply *simp*
apply (*erule-tac x*= *S2* **in** *allE*,*force*)
apply *simp*
apply *simp*
apply *simp*
apply *simp*

apply (*unfold equivState-def*)
apply (*elim exE*,*elim conjE*)

apply (*rule-tac x*=*None* **in** *exI*)
apply (*rule-tac x*=*sha*((*stackC*, *topf*) ↦ *Intg* (*n* − *2*),
(*heapC*, *k0f*) ↦ *vs*[*Suc 0* := *the* (*S*′*a* (*nat n*)), *2* := *the* (*S*′*a* (*nat* (*n* − *1*)))] ! *2*) **in** *exI*)
apply (*rule-tac x*=*h*(*la* ↦ *Arr ty m* (*S*′*a*(*nat* (*n* − *2*) ↦ *vs*[*Suc 0* := *the* (*S*′*a* (*nat n*))] ! *Suc 0*))) **in** *exI*)
apply (*rule-tac x*=*inih* **in** *exI*)
apply (*rule-tac x*=[([],
vs[*Suc 0* := *the* (*S*′*a* (*nat n*)), *2* := *the* (*S*′*a* (*nat* (*n* − *1*))), *3* := *the* (*S*′*a* (*nat* (*n* − *2*)))],
safeP, *sigSafeMain*,
if′ (((*the-Intg* (*vs*[*Suc 0* := *the* (*S*′*a* (*nat n*)),
2 := *the* (*S*′*a* (*nat* (*n* − *1*))),
3 := *the* (*S*′*a* (*nat* (*n* − *2*)))] ! *3*) < *1*) ∨
(*int* (*length ps*) < *the-Intg* (*vs*[*Suc 0* := *the* (*S*′*a* (*nat n*)),
2 := *the* (*S*′*a* (*nat* (*n* − *1*))),
3 := *the* (*S*′*a* (*nat* (*n* − *2*)))] ! *3*))))
(*nat* (*24* + *int* (*the* (*cdm* (*l*, *i*))) + *trAddr pcc* (*the* (*cdm* (*l*, *i*)) + *incPop*)) +
nat (*ps* ! *nat* (*int* (*length ps*))))
(*nat* (*24* + *int* (*the* (*cdm* (*l*, *i*))) + *trAddr pcc* (*the* (*cdm* (*l*, *i*)) + *incPop*)) +
nat (*ps* ! *nat* (*the-Intg* (*vs*[*Suc 0* := *the* (*S*′*a* (*nat n*)),
2 := *the* (*S*′*a* (*nat* (*n* − *1*))),
3 := *the* (*S*′*a* (*nat* (*n* − *2*)))] ! *3*) − *1*))),
ref)] **in** *exI*)

apply (*subgoal-tac* ∃ *vs pc* . *fms*=([],*vs*,*safeP*,*sigSafeMain*,*pc*,*ref*)#[])
prefer *2* **apply** *simp*
apply (*erule exE*)+
apply (*unfold exec-all-def*)
apply (*rule conjI*)

apply (*subgoal-tac PC* = (*l*,*i*)) **prefer** *2* **apply** *simp*
apply *simp* **apply** (*elim conjE*) **apply** *clarsimp*
apply (*subgoal-tac P*′⊢ (*None*,*sha*,*h*,*inih*,[([], *vs*, *safeP*, *sigSafeMain*, *the* (*cdm* (*l*,*i*)), *aa*,*ba*)]) −*jvm*→
(*None*,*sha*,*h*,*inih*,[([*the* (*sha* (*stackC*,*Sf*))], *vs*, *safeP*, *sigSafeMain*, *the* (*cdm* (*l*,*i*)) + *1*, *aa*,*ba*)]))
apply (*unfold exec-all-def*)

apply (*erule rtrancl-trans*)

prefer *2*
apply (*rule r-into-rtrancl*)
apply (*clarify*)
apply (*unfold JVMExec.exec.simps*)
apply (*unfold Let-def*)
apply (*unfold extractBytecode-def*)
apply (*subgoal-tac* (*fst* (*snd* (*snd* (*snd* (*snd* (*the* (*method'* (*P'*, *safeP*) *sigSafe-Main*))))))) !
the (*cdm* (*l*, *i*))) = *Getstatic Sf stackC*,*simp*)
apply (*unfold trInstr.simps*, *erule nth-via-drop-append*)

apply (*drule drop-Suc-append*)
apply (*subgoal-tac P'*⊢ (*None*, *sha*, *h*, *inih*,
[([*the* (*sha* (*stackC*, *Sf*))],
vs, *safeP*, *sigSafeMain*, *the* (*cdm* (*l*, *i*)) + *1*, *aa*, *ba*)])
−*jvm*→
(*None*,*sha*,*h*,*inih*,
[([*the* (*sha* (*stackC*,*topf*)),*the* (*sha* (*stackC*,*Sf*))],
vs, *safeP*, *sigSafeMain*, *the* (*cdm* (*l*, *i*)) + *1* + *1*, *aa*, *ba*)]))
apply (*unfold exec-all-def*)
apply (*erule rtrancl-trans*)
prefer *2*
apply (*rule r-into-rtrancl*)
apply (*clarify*)
apply (*unfold JVMExec.exec.simps*)
apply (*unfold Let-def*)
apply (*subgoal-tac* (*fst* (*snd* (*snd* (*snd* (*snd* (*the* (*method'* (*P'*, *safeP*) *sigSafe-Main*))))))) ! *Suc* (*the* (*cdm* (*l*, *i*)))) =
Getstatic topf stackC,*simp*)
apply (*erule nth-via-drop-append*)

apply (*drule drop-Suc-append*)
apply (*subgoal-tac P'*⊢ (*None*,*sha*,*h*,*inih*,
[([*the* (*sha* (*stackC*,*topf*)),*the* (*sha* (*stackC*, *Sf*))],
vs, *safeP*, *sigSafeMain*, *the* (*cdm* (*l*, *i*)) + *1* + *1*, *aa*, *ba*)])
−*jvm*→
(*None*,*sha*,*h*,*inih*,
[([*the* (*sha* (*stackC*,*topf*)),*the* (*sha* (*stackC*, *Sf*)),*the* (*sha* (*stackC*,*topf*)),*the* (*sha* (*stackC*,*Sf*))],
vs, *safeP*, *sigSafeMain*, *the* (*cdm* (*l*, *i*)) + *1* + *1* + *1*, *aa*, *ba*)]))
apply (*unfold exec-all-def*)
apply (*erule rtrancl-trans*)
prefer *2*

apply (*rule r-into-rtrancl*)
apply (*clarify*)
apply (*unfold JVMExec.exec.simps*)
apply (*unfold Let-def*)
apply (*subgoal-tac* (*fst* (*snd* (*snd* (*snd* (*snd* (*the* (*method'* (*P'*, *safeP*) *sigSafeMain*))))))) ! (*Suc* (*Suc* (*the* (*cdm* (*l*, *i*)))))) =
Dup2,*simp*)
apply (*erule nth-via-drop-append*)

apply (*drule drop-Suc-append*)
apply (*subgoal-tac P'*⊢ (*None*,*sha*,*h*,*inih*,
[([*the* (*sha* (*stackC*,*topf*)),*the* (*sha* (*stackC*, *Sf*)),
the (*sha* (*stackC*,*topf*)),*the* (*sha* (*stackC*,*Sf*))],
vs, *safeP*, *sigSafeMain*, *the* (*cdm* (*l*, *i*)) + *1* + *1* + *1*, *aa*, *ba*)])
−*jvm*→
(*None*,*sha*,*h*,*inih*,
[([*the* (*sha* (*stackC*,*topf*)),*the* (*sha* (*stackC*, *Sf*)),
the (*sha* (*stackC*,*topf*)),*the* (*sha* (*stackC*, *Sf*)),
the (*sha* (*stackC*,*topf*)),*the* (*sha* (*stackC*,*Sf*))],
vs, *safeP*, *sigSafeMain*, *the* (*cdm* (*l*, *i*)) + *1* + *1* + *1* + *1*, *aa*, *ba*)]))
apply (*unfold exec-all-def*)
apply (*erule rtrancl-trans*)
prefer *2*
apply (*rule r-into-rtrancl*)
apply (*clarify*)
apply (*unfold JVMExec.exec.simps*)
apply (*unfold Let-def*)
apply (*subgoal-tac* (*fst* (*snd* (*snd* (*snd* (*snd* (*the* (*method'* (*P'*, *safeP*) *sigSafeMain*))))))) ! (*Suc* (*Suc* (*Suc* (*the* (*cdm* (*l*, *i*))))))) =
Dup2,*simp*)
apply (*erule nth-via-drop-append*)

apply (*drule drop-Suc-append*)
apply (*subgoal-tac P'*⊢ (*None*,*sha*,*h*,*inih*,
[([*the* (*sha* (*stackC*,*topf*)),*the* (*sha* (*stackC*, *Sf*)),
the (*sha* (*stackC*,*topf*)),*the* (*sha* (*stackC*, *Sf*)),
the (*sha* (*stackC*,*topf*)),*the* (*sha* (*stackC*,*Sf*))],
vs, *safeP*, *sigSafeMain*, *the* (*cdm* (*l*, *i*)) + *1* + *1* + *1* + *1*, *aa*, *ba*)])
−*jvm*→
(*None*,*sha*,*h*,*inih*,
[([*the* (*sha* (*stackC*,*topf*)),*the* (*sha* (*stackC*, *Sf*)),
the (*sha* (*stackC*,*topf*)),*the* (*sha* (*stackC*, *Sf*)),

```
                    the (sha (stackC,topf)),the (sha (stackC, Sf)),
                    the (sha (stackC,topf)),the (sha (stackC,Sf))],
                  vs, safeP, sigSafeMain, the (cdm (l, i)) + 1 + 1 + 1 + 1 +
1, aa, ba)]))
apply (unfold exec-all-def)
apply (erule rtrancl-trans)
prefer 2
 apply (rule r-into-rtrancl)
 apply (clarify)
 apply (unfold JVMExec.exec.simps)
 apply (unfold Let-def)
 apply (subgoal-tac (fst (snd (snd (snd (snd (the (method' (P', safeP) sigSafe-
Main)))))) !
                  (Suc (Suc (Suc (Suc (the (cdm (l, i)))))))) =
                   Dup2,simp)
apply (erule nth-via-drop-append)


apply (drule drop-Suc-append)
apply (subgoal-tac P'⊢ (None,sha,h,inih,
                  [([the (sha (stackC,topf)),the (sha (stackC, Sf)),
                    the (sha (stackC,topf)),the (sha (stackC, Sf)),
                    the (sha (stackC,topf)),the (sha (stackC, Sf)),
                    the (sha (stackC,topf)),the (sha (stackC,Sf))],
                  vs, safeP, sigSafeMain, the (cdm (l, i)) + 1 + 1 + 1 + 1 +
1, aa, ba)])
                -jvm→
                  (None,sha,h,inih,
                  [([the (S'a (nat n)),
                    the (sha (stackC,topf)),the (sha (stackC, Sf)),
                    the (sha (stackC,topf)),the (sha (stackC, Sf)),
                    the (sha (stackC,topf)),the (sha (stackC, Sf))],
                  vs, safeP, sigSafeMain, the (cdm (l, i)) + 1 + 1 + 1 + 1 +
1 + 1, aa, ba)]))
apply (unfold exec-all-def)
apply (erule rtrancl-trans)
prefer 2
 apply (rule r-into-rtrancl)
 apply (clarify)
 apply (unfold JVMExec.exec.simps)
 apply (unfold Let-def)
 apply (subgoal-tac (fst (snd (snd (snd (snd (the (method' (P', safeP) sigSafe-
Main)))))) !
                  (Suc (Suc (Suc (Suc (Suc (the (cdm (l, i))))))))) =
                   ArrLoad,simp)

  apply (frule equivS-POPCONT, simp)
  apply (simp add: raise-system-xcpt-def)
apply (erule nth-via-drop-append)
```

apply (*drule drop-Suc-append*)
apply (*subgoal-tac P'⊢* (*None*,*sha*,*h*,*inih*,
[([*the* (*S'a* (*nat n*)),
the (*sha* (*stackC*,*topf*)),*the* (*sha* (*stackC*, *Sf*)),
the (*sha* (*stackC*,*topf*)),*the* (*sha* (*stackC*, *Sf*)),
the (*sha* (*stackC*,*topf*)),*the* (*sha* (*stackC*,*Sf*))],
vs, *safeP*, *sigSafeMain*, *the* (*cdm* (*l*, *i*)) + *1* + *1* + *1* + *1* + *1* + *1*, *aa*, *ba*)])
−jvm→
(*None*,*sha*,*h*,*inih*,
[([*the* (*sha* (*stackC*,*topf*)),*the* (*sha* (*stackC*, *Sf*)),
the (*sha* (*stackC*,*topf*)),*the* (*sha* (*stackC*, *Sf*)),
the (*sha* (*stackC*,*topf*)),*the* (*sha* (*stackC*,*Sf*))],
vs[*Suc 0* := *the* (*S'a* (*nat n*))],
safeP, *sigSafeMain*, *the* (*cdm* (*l*, *i*)) + *1* + *1* + *1* + *1* + *1* + *1* + *1*, *aa*, *ba*)]))
apply (*unfold exec-all-def*)
apply (*erule rtrancl-trans*)
prefer *2*
apply (*rule r-into-rtrancl*)
apply (*clarify*)
apply (*unfold JVMExec.exec.simps*)
apply (*unfold Let-def*)
apply (*subgoal-tac* (*fst* (*snd* (*snd* (*snd* (*snd* (*the* (*method'* (*P'*, *safeP*) *sigSafeMain*))))))) !
(*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*the* (*cdm* (*l*, *i*))))))))) =
Store 1,*simp*)
apply (*erule nth-via-drop-append*)

apply (*drule drop-Suc-append*)
apply (*subgoal-tac P'⊢* (*None*,*sha*,*h*,*inih*,
[([*the* (*sha* (*stackC*,*topf*)),*the* (*sha* (*stackC*, *Sf*)),
the (*sha* (*stackC*,*topf*)),*the* (*sha* (*stackC*, *Sf*)),
the (*sha* (*stackC*,*topf*)),*the* (*sha* (*stackC*,*Sf*))],
vs[*Suc 0* := *the* (*S'a* (*nat n*))],
safeP, *sigSafeMain*, *the* (*cdm* (*l*, *i*)) + *1* + *1* + *1* + *1* + *1* + *1* + *1*, *aa*, *ba*)])
−jvm→
(*None*,*sha*,*h*,*inih*,
[([*Intg 1* ,
the (*sha* (*stackC*,*topf*)),*the* (*sha* (*stackC*, *Sf*)),
the (*sha* (*stackC*,*topf*)),*the* (*sha* (*stackC*, *Sf*)),
the (*sha* (*stackC*,*topf*)),*the* (*sha* (*stackC*,*Sf*))],
vs[*Suc 0* := *the* (*S'a* (*nat n*))],
safeP, *sigSafeMain*, *the* (*cdm* (*l*, *i*)) + *1* + *1* + *1* + *1* + *1* + *1*

```
+ 1 + 1 + 1, aa, ba)]))
apply (unfold exec-all-def)
apply (erule rtrancl-trans)
prefer 2
 apply (rule r-into-rtrancl)
 apply (clarify)
 apply (unfold JVMExec.exec.simps)
 apply (unfold Let-def)
 apply (subgoal-tac (fst (snd (snd (snd (snd (the (method' (P', safeP) sigSafe-
Main)))))) !
                    (Suc (Suc (Suc (Suc (Suc (Suc (Suc (the (cdm (l, i))))))))))) =
                     LitPush (Intg 1),simp)
apply (erule nth-via-drop-append)



apply (drule drop-Suc-append)
apply (subgoal-tac P'⊢ (None,sha,h,inih,
                        [([Intg 1,
                           the (sha (stackC,topf)),the (sha (stackC, Sf)),
                           the (sha (stackC,topf)),the (sha (stackC, Sf)),
                           the (sha (stackC,topf)),the (sha (stackC,Sf))],
                        vs[Suc 0 := the (S'a (nat n))],
                         safeP, sigSafeMain, the (cdm (l, i)) + 1 + 1 + 1 + 1 + 1
+ 1 + 1 + 1 + 1, aa, ba)])
                    −jvm→
                        (None,sha,h,inih,
                        [([Intg (the-Intg (the (sha (stackC, topf)))− 1),
                           the (sha (stackC, Sf)),the (sha (stackC,topf)),
                           the (sha (stackC, Sf)),the (sha (stackC,topf)),
                           the (sha (stackC,Sf))],
                        vs[Suc 0 := the (S'a (nat n))],
                         safeP, sigSafeMain, the (cdm (l, i)) + 1 + 1 + 1 + 1 + 1
+ 1 + 1 + 1 + 1 + 1, aa, ba)]))
apply (unfold exec-all-def)
apply (erule rtrancl-trans)
prefer 2
 apply (rule r-into-rtrancl)
 apply (clarify)
 apply (unfold JVMExec.exec.simps)
 apply (unfold Let-def)
 apply (subgoal-tac (fst (snd (snd (snd (snd (the (method' (P', safeP) sigSafe-
Main)))))) !
                    (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (the (cdm (l, i))))))))))))
=
                        BinOp Substract,simp)
apply (erule nth-via-drop-append)
```

apply (*drule drop-Suc-append*)
apply (*subgoal-tac P'⊢* (*None*,*sha*,*h*,*inih*,
[([*Intg* (*the-Intg* (*the* (*sha* (*stackC*, *topf*))) − *1*),
the (*sha* (*stackC*, *Sf*)),*the* (*sha* (*stackC*,*topf*)),
the (*sha* (*stackC*, *Sf*)),*the* (*sha* (*stackC*,*topf*)),
the (*sha* (*stackC*,*Sf*))],
vs[*Suc 0* := *the* (*S'a* (*nat n*))],
safeP, *sigSafeMain*, *the* (*cdm* (*l*, *i*)) + *1* + *1* + *1* + *1* + *1*
+ *1* + *1* + *1* + *1*, *aa*, *ba*)])
−jvm→
(*None*,*sha*,*h*,*inih*,
[([*the* (*S'a* (*nat* (*n* − *1*))),
the (*sha* (*stackC*,*topf*)),*the* (*sha* (*stackC*, *Sf*)),
the (*sha* (*stackC*,*topf*)),*the* (*sha* (*stackC*,*Sf*))],
vs[*Suc 0* := *the* (*S'a* (*nat n*))],
safeP, *sigSafeMain*, *the* (*cdm* (*l*, *i*)) + *1* + *1* + *1* + *1* + *1*
+ *1* + *1* + *1* + *1* + *1*, *aa*, *ba*)]))
apply (*unfold exec-all-def*)
apply (*erule rtrancl-trans*)
prefer *2*
apply (*rule r-into-rtrancl*)
apply (*clarify*)
apply (*unfold JVMExec.exec.simps*)
apply (*unfold Let-def*)
apply (*subgoal-tac* (*fst* (*snd* (*snd* (*snd* (*snd* (*the* (*method'* (*P'*, *safeP*) *sigSafe-Main*)))))) !
(*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*the* (*cdm* (*l*,
i))))))))))) =
ArrLoad,*simp*)

apply (*frule equivS-POPCONT*, *simp*)
apply (*simp add*: *raise-system-xcpt-def*)
apply (*erule nth-via-drop-append*)

apply (*drule drop-Suc-append*)
apply (*subgoal-tac P'⊢* (*None*,*sha*,*h*,*inih*,
[([*the* (*S'a* (*nat* (*n* − *1*))),
the (*sha* (*stackC*,*topf*)),*the* (*sha* (*stackC*, *Sf*)),
the (*sha* (*stackC*,*topf*)),*the* (*sha* (*stackC*,*Sf*))],
vs[*Suc 0* := *the* (*S'a* (*nat n*))],
safeP, *sigSafeMain*, *the* (*cdm* (*l*, *i*)) + *1* + *1* + *1* + *1* + *1*
+ *1* + *1* + *1* + *1* + *1*, *aa*, *ba*)])
−jvm→
(*None*,*sha*,*h*,*inih*,
[([*the* (*sha* (*stackC*,*topf*)),*the* (*sha* (*stackC*, *Sf*)),
the (*sha* (*stackC*,*topf*)),*the* (*sha* (*stackC*,*Sf*))],
vs[*Suc 0* := *the* (*S'a* (*nat n*)),

```
                    2 := the (S'a (nat (n − 1)))],
                safeP, sigSafeMain, the (cdm (l, i)) + 1 + 1 + 1 + 1 + 1
+ 1 + 1 + 1 + 1 + 1 + 1, aa, ba)]))
apply (unfold exec-all-def)
apply (erule rtrancl-trans)
prefer 2
 apply (rule r-into-rtrancl)
 apply (clarify)
 apply (unfold JVMExec.exec.simps)
 apply (unfold Let-def)
 apply (subgoal-tac (fst (snd (snd (snd (snd (the (method' (P', safeP) sigSafe-
Main)))))) !
                (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (the (cdm
(l, i))))))))))))) =
                Store 2,simp)
apply (erule nth-via-drop-append)



apply (drule drop-Suc-append)
apply (subgoal-tac P'⊢ (None,sha,h,inih,
                [([the (sha (stackC,topf)),the (sha (stackC, Sf)),
                   the (sha (stackC,topf)),the (sha (stackC,Sf))],
                vs[Suc 0 := the (S'a (nat n)),
                    2 := the (S'a (nat (n − 1)))],
                 safeP, sigSafeMain, the (cdm (l, i)) + 1 + 1 + 1 + 1 + 1
+ 1 + 1 + 1 + 1 + 1 + 1, aa, ba)])
             −jvm→
                (None,sha,h,inih,
                [([Intg 2,
                   the (sha (stackC,topf)),the (sha (stackC, Sf)),
                   the (sha (stackC,topf)),the (sha (stackC,Sf))],
                vs[Suc 0 := the (S'a (nat n)),
                    2 := the (S'a (nat (n − 1)))],
                 safeP, sigSafeMain, the (cdm (l, i)) + 1 + 1 + 1 + 1 + 1
+ 1 + 1 + 1 + 1 + 1 + 1, aa, ba)]))
apply (unfold exec-all-def)
apply (erule rtrancl-trans)
prefer 2
 apply (rule r-into-rtrancl)
 apply (clarify)
 apply (unfold JVMExec.exec.simps)
 apply (unfold Let-def)
 apply (subgoal-tac (fst (snd (snd (snd (snd (the (method' (P', safeP) sigSafe-
Main)))))) !
                (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (the
(cdm (l, i))))))))))))) =
                LitPush (Intg 2),simp)
apply (erule nth-via-drop-append)
```

apply (*drule drop-Suc-append*)
apply (*subgoal-tac P'⊢* (*None*,*sha*,*h*,*inih*,
[([*Intg 2*,
the (*sha* (*stackC*,*topf*)),*the* (*sha* (*stackC*, *Sf*)),
the (*sha* (*stackC*,*topf*)),*the* (*sha* (*stackC*,*Sf*))],
vs[*Suc 0* := *the* (*S'a* (*nat n*)),
2 := *the* (*S'a* (*nat* (*n* − *1*)))],
safeP, *sigSafeMain*, *the* (*cdm* (*l*, *i*)) + *1* + *1* + *1* + *1* + *1*
+ *1* + *1* + *1* + *1* + *1* + *1* + *1*, *aa*, *ba*)])
−*jvm*→
(*None*,*sha*,*h*,*inih*,
[([*Intg* (*the-Intg* (*the* (*sha* (*stackC*, *topf*))) − *2*),
the (*sha* (*stackC*, *Sf*)),*the* (*sha* (*stackC*,*topf*)),
the (*sha* (*stackC*,*Sf*))],
vs[*Suc 0* := *the* (*S'a* (*nat n*)),
2 := *the* (*S'a* (*nat* (*n* − *1*)))],
safeP, *sigSafeMain*, *the* (*cdm* (*l*, *i*)) + *1* + *1* + *1* + *1* + *1*
+ *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1*, *aa*, *ba*)]))
apply (*unfold exec-all-def*)
apply (*erule rtrancl-trans*)
prefer *2*
apply (*rule r-into-rtrancl*)
apply (*clarify*)
apply (*unfold JVMExec.exec.simps*)
apply (*unfold Let-def*)
apply (*subgoal-tac* (*fst* (*snd* (*snd* (*snd* (*snd* (*the* (*method'* (*P'*, *safeP*) *sigSafe-Main*))))))) !
(*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc*
(*the* (*cdm* (*l*, *i*))))))))))))))) =
BinOp Substract,*simp*)
apply (*erule nth-via-drop-append*)

apply (*drule drop-Suc-append*)
apply (*subgoal-tac P'⊢* (*None*,*sha*,*h*,*inih*,
[([*Intg* (*the-Intg* (*the* (*sha* (*stackC*, *topf*))) − *2*),
the (*sha* (*stackC*, *Sf*)),*the* (*sha* (*stackC*,*topf*)),
the (*sha* (*stackC*,*Sf*))],
vs[*Suc 0* := *the* (*S'a* (*nat n*)),
2 := *the* (*S'a* (*nat* (*n* − *1*)))],
safeP, *sigSafeMain*, *the* (*cdm* (*l*, *i*)) + *1* + *1* + *1* + *1* + *1*
+ *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1*, *aa*, *ba*)])
−*jvm*→
(*None*,*sha*,*h*,*inih*,
[([*the* (*S'a* (*nat* (*n* − *2*))),
the (*sha* (*stackC*,*topf*)),*the* (*sha* (*stackC*,*Sf*))],

```
                    vs[Suc 0 := the (S'a (nat n)),
                          2 := the (S'a (nat (n - 1)))],
                     safeP, sigSafeMain, the (cdm (l, i)) + 1 + 1 + 1 + 1 + 1
+ 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1, aa, ba)]))
apply (unfold exec-all-def)
apply (erule rtrancl-trans)
prefer 2
 apply (rule r-into-rtrancl)
 apply (clarify)
 apply (unfold JVMExec.exec.simps)
 apply (unfold Let-def)
 apply (subgoal-tac (fst (snd (snd (snd (snd (the (method' (P', safeP) sigSafe-
Main)))))) !
                   (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc
(Suc (the (cdm (l, i)))))))))))))))) =
                    ArrLoad,simp)

  apply (frule equivS-POPCONT, simp)
  apply (simp add: raise-system-xcpt-def)
apply (erule nth-via-drop-append)


apply (drule drop-Suc-append)
apply (subgoal-tac P'⊢ (None,sha,h,inih,
                    [([the (S'a (nat (n - 2))),
                       the (sha (stackC,topf)),the (sha (stackC,Sf))],
                    vs[Suc 0 := the (S'a (nat n)),
                          2 := the (S'a (nat (n - 1)))],
                     safeP, sigSafeMain, the (cdm (l, i)) + 1 + 1 + 1 + 1 + 1
+ 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1, aa, ba)])
                  −jvm→
                    (None,sha,h,inih,
                    [([the (sha (stackC,topf)),the (sha (stackC,Sf))],
                    vs[Suc 0 := the (S'a (nat n)),
                          2 := the (S'a (nat (n - 1))),
                          3 := the (S'a (nat (n - 2)))],
                     safeP, sigSafeMain, the (cdm (l, i)) + 1 + 1 + 1 + 1 + 1
+ 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1, aa, ba)]))
apply (unfold exec-all-def)
apply (erule rtrancl-trans)
prefer 2
 apply (rule r-into-rtrancl)
 apply (clarify)
 apply (unfold JVMExec.exec.simps)
 apply (unfold Let-def)
 apply (subgoal-tac (fst (snd (snd (snd (snd (the (method' (P', safeP) sigSafe-
Main)))))) !
                   (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc
(Suc (Suc (the (cdm (l, i)))))))))))))))) =
```

Store 3,*simp*)
apply (*erule nth-via-drop-append*)

apply (*drule drop-Suc-append*)
apply (*subgoal-tac P*′⊢ (*None*,*sha*,*h*,*inih*,
[([*the* (*sha* (*stackC*,*topf*)),*the* (*sha* (*stackC*,*Sf*))],
vs[*Suc 0* := *the* (*S*′*a* (*nat n*)),
2 := *the* (*S*′*a* (*nat* (*n* − *1*))),
3 := *the* (*S*′*a* (*nat* (*n* − *2*)))],
safeP, *sigSafeMain*, *the* (*cdm* (*l*, *i*)) + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1*, *aa*, *ba*)])
−*jvm*→
(*None*,*sha*,*h*,*inih*,
[([*Intg 2*,
the (*sha* (*stackC*,*topf*)),*the* (*sha* (*stackC*,*Sf*))],
vs[*Suc 0* := *the* (*S*′*a* (*nat n*)),
2 := *the* (*S*′*a* (*nat* (*n* − *1*))),
3 := *the* (*S*′*a* (*nat* (*n* − *2*)))],
safeP, *sigSafeMain*, *the* (*cdm* (*l*, *i*)) + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1*, *aa*, *ba*)]))
apply (*unfold exec-all-def*)
apply (*erule rtrancl-trans*)
prefer *2*
apply (*rule r-into-rtrancl*)
apply (*clarify*)
apply (*unfold JVMExec.exec.simps*)
apply (*unfold Let-def*)
apply (*subgoal-tac* (*fst* (*snd* (*snd* (*snd* (*snd* (*the* (*method*′ (*P*′, *safeP*) *sigSafeMain*)))))) !
(*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*the* (*cdm* (*l*, *i*)))))))))))))))))))) =
LitPush (*Intg 2*),*simp*)
apply (*erule nth-via-drop-append*)

apply (*drule drop-Suc-append*)
apply (*subgoal-tac P*′⊢ (*None*,*sha*,*h*,*inih*,
[([*Intg 2*,
the (*sha* (*stackC*,*topf*)),*the* (*sha* (*stackC*,*Sf*))],
vs[*Suc 0* := *the* (*S*′*a* (*nat n*)),
2 := *the* (*S*′*a* (*nat* (*n* − *1*))),
3 := *the* (*S*′*a* (*nat* (*n* − *2*)))],
safeP, *sigSafeMain*, *the* (*cdm* (*l*, *i*)) + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1*, *aa*, *ba*)])
−*jvm*→
(*None*,*sha*,*h*,*inih*,

```
                    [([Intg (n − 2),
                      the (sha (stackC,Sf))],
                    vs[Suc 0 := the (S'a (nat n)),
                          2 := the (S'a (nat (n − 1))),
                          3 := the (S'a (nat (n − 2)))],
                   safeP, sigSafeMain, the (cdm (l, i)) + 1 + 1 + 1 + 1 + 1 +
1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1, aa, ba)]))
apply (unfold exec-all-def)
apply (erule rtrancl-trans)
prefer 2
 apply (rule r-into-rtrancl)
 apply (clarify)
 apply (unfold JVMExec.exec.simps)
 apply (unfold Let-def)
 apply (subgoal-tac (fst (snd (snd (snd (snd (the (method' (P', safeP) sigSafe-
Main)))))) !
                   (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc
(Suc (Suc (Suc (Suc (the (cdm (l, i)))))))))))))))))) =
                   BinOp Substract,simp)
apply (erule nth-via-drop-append)


apply (drule drop-Suc-append)
apply (subgoal-tac P'⊢ (None,sha,h,inih,
                    [([Intg (n − 2),
                      the (sha (stackC,Sf))],
                    vs[Suc 0 := the (S'a (nat n)),
                          2 := the (S'a (nat (n − 1))),
                          3 := the (S'a (nat (n − 2)))],
                   safeP, sigSafeMain, the (cdm (l, i)) + 1 + 1 + 1 + 1 + 1 +
1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1, aa, ba)])
                  −jvm→
                    (None,sha,h,inih,
                    [([Intg (n − 2),
                       Intg (n − 2),
                      the (sha (stackC,Sf))],
                    vs[Suc 0 := the (S'a (nat n)),
                          2 := the (S'a (nat (n − 1))),
                          3 := the (S'a (nat (n − 2)))],
                   safeP, sigSafeMain,
                   the (cdm (l, i)) + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1
+ 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1, aa, ba)]))
apply (unfold exec-all-def)
apply (erule rtrancl-trans)
prefer 2
 apply (rule r-into-rtrancl)
 apply (clarify)
 apply (unfold JVMExec.exec.simps)
 apply (unfold Let-def)
```

apply (*subgoal-tac* (*fst* (*snd* (*snd* (*snd* (*snd* (*the* (*method'* (*P'*, *safeP*) *sigSafeMain*))))))) !
(*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc*
(*Suc* (*Suc* (*Suc* (*Suc* (*the* (*cdm* (*l*, *i*))))))))))))))))))) =
Dup,*simp*)
apply (*erule nth-via-drop-append*)

apply (*drule drop-Suc-append*)
apply (*subgoal-tac P'* $\vdash$ (*None*,*sha*,*h*,*inih*,
[([*Intg* (*n* − *2*),
Intg (*n* − *2*),
the (*sha* (*stackC*,*Sf*))],
vs[*Suc 0* := *the* (*S'a* (*nat n*)),
2 := *the* (*S'a* (*nat* (*n* − *1*))),
3 := *the* (*S'a* (*nat* (*n* − *2*)))],
safeP, *sigSafeMain*,
the (*cdm* (*l*, *i*)) + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1*, *aa*, *ba*)])
−*jvm*→
(*None*, *sha*((*stackC*, *topf*) ↦ *Intg* (*n* − *2*)),*h*,*inih*,
[([*Intg* (*n* − *2*),
the (*sha* (*stackC*,*Sf*))],
vs[*Suc 0* := *the* (*S'a* (*nat n*)),
2 := *the* (*S'a* (*nat* (*n* − *1*))),
3 := *the* (*S'a* (*nat* (*n* − *2*)))],
safeP, *sigSafeMain*,
the (*cdm* (*l*, *i*)) + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1*, *aa*, *ba*)]))
apply (*unfold exec-all-def*)
apply (*erule rtrancl-trans*)
prefer *2*
apply (*rule r-into-rtrancl*)
apply (*clarify*)
apply (*unfold JVMExec.exec.simps*)
apply (*unfold Let-def*)
apply (*subgoal-tac* (*fst* (*snd* (*snd* (*snd* (*snd* (*the* (*method'* (*P'*, *safeP*) *sigSafeMain*))))))) !
(*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc*
(*Suc* (*Suc* (*Suc* (*Suc* (*the* (*cdm* (*l*, *i*))))))))))))))))))) =
Putstatic topf stackC,*simp*)
apply (*erule nth-via-drop-append*)

apply (*drule drop-Suc-append*)
apply (*subgoal-tac P'* $\vdash$ (*None*, *sha*((*stackC*, *topf*) ↦ *Intg* (*n* − *2*)),*h*,*inih*,

```
                    [([Intg (n − 2),
                      the (sha (stackC,Sf))],
                    vs[Suc 0 := the (S'a (nat n)),
                          2 := the (S'a (nat (n − 1))),
                          3 := the (S'a (nat (n − 2)))],
                  safeP, sigSafeMain,
                  the (cdm (l, i)) + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1
+ 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1, aa, ba)])
                −jvm→
                    (None, sha((stackC, topf) ↦ Intg (n − 2)),h,inih,
                    [([vs[Suc 0 := the (S'a (nat n))]! Suc 0,
                      Intg (n − 2),
                      the (sha (stackC,Sf))],
                     vs[Suc 0 := the (S'a (nat n)),
                          2 := the (S'a (nat (n − 1))),
                          3 := the (S'a (nat (n − 2)))],
                  safeP, sigSafeMain,
                  the (cdm (l, i)) + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1
+ 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1, aa, ba)]))
apply (unfold exec-all-def)
apply (erule rtrancl-trans)
prefer 2
 apply (rule r-into-rtrancl)
 apply (clarify)
 apply (unfold JVMExec.exec.simps)
 apply (unfold Let-def)
 apply (subgoal-tac (fst (snd (snd (snd (snd (the (method' (P', safeP) sigSafe-
Main)))))) !
                     (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc
(Suc (Suc (Suc
                    (Suc (Suc (Suc (Suc (the (cdm (l, i))))))))))))))))))))) =
                     Load 1,simp)
apply (erule nth-via-drop-append)


apply (drule drop-Suc-append)
apply (subgoal-tac P'⊢ (None, sha((stackC, topf) ↦ Intg (n − 2)),h,inih,
                    [([vs[Suc 0 := the (S'a (nat n))]! Suc 0,
                      Intg (n − 2),
                      the (sha (stackC,Sf))],
                     vs[Suc 0 := the (S'a (nat n)),
                          2 := the (S'a (nat (n − 1))),
                          3 := the (S'a (nat (n − 2)))],
                  safeP, sigSafeMain,
                  the (cdm (l, i)) + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1
+ 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1, aa, ba)])
                −jvm→
                    (None, sha((stackC, topf) ↦ Intg (n − 2)),
                          h(la ↦ Arr ty m (S'a(nat (n − 2) ↦ vs[Suc 0 := the
```

```
(S'a (nat n))] ! Suc 0))),inih,
                        [([],
                        vs[Suc 0 := the (S'a (nat n)),
                              2 := the (S'a (nat (n - 1))),
                              3 := the (S'a (nat (n - 2)))],
                    safeP, sigSafeMain,
                    the (cdm (l, i)) + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1
+ 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 +1, aa, ba)]))
apply (unfold exec-all-def)
apply (erule rtrancl-trans)
prefer 2
 apply (rule r-into-rtrancl)
 apply (clarify)
 apply (unfold JVMExec.exec.simps)
 apply (unfold Let-def)
 apply (subgoal-tac (fst (snd (snd (snd (snd (the (method' (P', safeP) sigSafe-
Main)))))) !
                     (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc
(Suc (Suc (Suc (Suc
                    (Suc (Suc (Suc (Suc (the (cdm (l, i))))))))))))))))))))) =
                     ArrStore,simp)

  apply (frule equivS-POPCONT, simp)
  apply (simp add: raise-system-xcpt-def)
apply (erule nth-via-drop-append)


apply (drule drop-Suc-append)
apply (subgoal-tac P'⊢ (None, sha((stackC, topf) ↦ Intg (n - 2)),
                        h(la ↦ Arr ty m (S'a(nat (n - 2) ↦ vs[Suc 0 := the
(S'a (nat n))] ! Suc 0))),inih,
                        [([],
                        vs[Suc 0 := the (S'a (nat n)),
                              2 := the (S'a (nat (n - 1))),
                              3 := the (S'a (nat (n - 2)))],
                    safeP, sigSafeMain,
                    the (cdm (l, i)) + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1
+ 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 +1, aa, ba)])
                   −jvm→
                        (None, sha((stackC, topf) ↦ Intg (n - 2)),
                              h(la ↦ Arr ty m (S'a(nat (n - 2) ↦ vs[Suc 0 := the
(S'a (nat n))] ! Suc 0))),inih,
                        [([vs[Suc 0 := the (S'a (nat n)),
                                 2 := the (S'a (nat (n - 1)))]!2],
                        vs[Suc 0 := the (S'a (nat n)),
                              2 := the (S'a (nat (n - 1))),
                              3 := the (S'a (nat (n - 2)))],
                    safeP, sigSafeMain,
                    the (cdm (l, i)) + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1
```

```
+ 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 +1+1, aa, ba)]))
apply (unfold exec-all-def)
apply (erule rtrancl-trans)
prefer 2
 apply (rule r-into-rtrancl)
 apply (clarify)
 apply (unfold JVMExec.exec.simps)
 apply (unfold Let-def)
 apply (subgoal-tac (fst (snd (snd (snd (snd (the (method' (P', safeP) sigSafe-
Main)))))) !
                    (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc
(Suc (Suc (Suc (Suc (Suc
                    (Suc (Suc (Suc (Suc (the (cdm (l, i))))))))))))))))))))))) =
                     Load 2,simp)
apply (erule nth-via-drop-append)



apply (drule drop-Suc-append)
apply (subgoal-tac P'⊢ (None, sha((stackC, topf) ↦ Intg (n - 2)),
                              h(la ↦ Arr ty m (S'a(nat (n - 2) ↦ vs[Suc 0 := the
(S'a (nat n))] ! Suc 0))),inih,
                        [([vs[Suc 0 := the (S'a (nat n)),
                             2 := the (S'a (nat (n - 1)))]!2],
                        vs[Suc 0 := the (S'a (nat n)),
                              2 := the (S'a (nat (n - 1))),
                              3 := the (S'a (nat (n - 2)))],
                    safeP, sigSafeMain,
                    the (cdm (l, i)) + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1
+ 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 +1+1, aa, ba)])
                   −jvm→
                       (None, sha((stackC, topf) ↦ Intg (n - 2), (heapC, k0f) ↦
vs[Suc 0 := the (S'a (nat n)),
                                                                  2 := the
(S'a (nat (n - 1)))] ! 2),
                              h(la ↦ Arr ty m (S'a(nat (n - 2) ↦ vs[Suc 0 := the
(S'a (nat n))] ! Suc 0))),inih,
                        [([],
                        vs[Suc 0 := the (S'a (nat n)),
                              2 := the (S'a (nat (n - 1))),
                              3 := the (S'a (nat (n - 2)))],
                    safeP, sigSafeMain,
                  the (cdm (l, i)) + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1
+ 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 +1 + 1 + 1, aa, ba)]))
apply (unfold exec-all-def)
apply (erule rtrancl-trans)
prefer 2
 apply (rule r-into-rtrancl)
 apply (clarify)
```

apply (*unfold JVMExec.exec.simps*)
apply (*unfold Let-def*)
apply (*subgoal-tac* (*fst* (*snd* (*snd* (*snd* (*snd* (*the* (*method'* (*P'*, *safeP*) *sigSafe-Main*)))))) !
(*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc*
(*Suc* (*Suc* (*Suc* (*Suc* (*the* (*cdm* (*l*, *i*))))))))))))))))))))))))))) =
Putstatic k0f heapC,*simp*)
apply (*erule nth-via-drop-append*)

apply (*drule drop-Suc-append*)
apply (*subgoal-tac P'*⊢ (*None*, *sha*((*stackC*, *topf*) ↦ *Intg* (*n* − *2*), (*heapC*, *k0f*) ↦ *vs*[*Suc 0* := *the* (*S'a* (*nat n*)),
2 := *the* (*S'a* (*nat* (*n* − *1*)))] ! *2*),
h(*la* ↦ *Arr ty m* (*S'a*(*nat* (*n* − *2*) ↦ *vs*[*Suc 0* := *the* (*S'a* (*nat n*))] ! *Suc 0*))),*inih*,
[([],
vs[*Suc 0* := *the* (*S'a* (*nat n*)),
2 := *the* (*S'a* (*nat* (*n* − *1*))),
3 := *the* (*S'a* (*nat* (*n* − *2*)))],
safeP, *sigSafeMain*,
the (*cdm* (*l*, *i*)) + *1* +*1* + *1* + *1*, *aa*, *ba*)])
−*jvm*→
(*None*, *sha*((*stackC*, *topf*) ↦ *Intg* (*n* − *2*), (*heapC*, *k0f*) ↦ *vs*[*Suc 0* := *the* (*S'a* (*nat n*)),
2 := *the* (*S'a* (*nat* (*n* − *1*)))] ! *2*),
h(*la* ↦ *Arr ty m* (*S'a*(*nat* (*n* − *2*) ↦ *vs*[*Suc 0* := *the* (*S'a* (*nat n*))] ! *Suc 0*))),*inih*,
[([*vs*[*Suc 0* := *the* (*S'a* (*nat n*)),
2 := *the* (*S'a* (*nat* (*n* − *1*))),
3 := *the* (*S'a* (*nat* (*n* − *2*)))]!*3*],
vs[*Suc 0* := *the* (*S'a* (*nat n*)),
2 := *the* (*S'a* (*nat* (*n* − *1*))),
3 := *the* (*S'a* (*nat* (*n* − *2*)))],
safeP, *sigSafeMain*,
the (*cdm* (*l*, *i*)) + *1* +*1* + *1* + *1* + *1*, *aa*, *ba*)]))
apply (*unfold exec-all-def*)
apply (*erule rtrancl-trans*)
prefer *2*
apply (*rule r-into-rtrancl*)
apply (*clarify*)
apply (*unfold JVMExec.exec.simps*)
apply (*unfold Let-def*)

apply (*subgoal-tac* (*fst* (*snd* (*snd* (*snd* (*snd* (*the* (*method'* (*P'*, *safeP*) *sigSafe-Main*)))))) !
(*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc*
(*Suc* (*Suc* (*Suc* (*Suc* (*the* (*cdm* (*l*, *i*)))))))))))))))))))))))))) =
Load 3,*simp*)
apply (*erule nth-via-drop-append*)

apply (*drule drop-Suc-append*)
apply (*subgoal-tac P'*⊢ (*None*, *sha*((*stackC*, *topf*) ↦ *Intg* (*n* − *2*), (*heapC*, *k0f*) ↦ *vs*[*Suc 0* := *the* (*S'a* (*nat n*)),
2 := *the* (*S'a* (*nat* (*n* − *1*)))] ! *2*),
h(*la* ↦ *Arr ty m* (*S'a*(*nat* (*n* − *2*) ↦ *vs*[*Suc 0* := *the* (*S'a* (*nat n*))] ! *Suc 0*))),*inih*,
[([*vs*[*Suc 0* := *the* (*S'a* (*nat n*)),
2 := *the* (*S'a* (*nat* (*n* − *1*))),
3 := *the* (*S'a* (*nat* (*n* − *2*)))]!*3*],
vs[*Suc 0* := *the* (*S'a* (*nat n*)),
2 := *the* (*S'a* (*nat* (*n* − *1*))),
3 := *the* (*S'a* (*nat* (*n* − *2*)))],
safeP, *sigSafeMain*,
the (*cdm* (*l*, *i*)) + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* +*1* + *1* + *1* + *1*, *aa*, *ba*)])
−*jvm*→
(*None*, *sha*((*stackC*, *topf*) ↦ *Intg* (*n* − *2*), (*heapC*, *k0f*) ↦ *vs*[*Suc 0* := *the* (*S'a* (*nat n*)),
2 := *the* (*S'a* (*nat* (*n* − *1*)))] ! *2*),
h(*la* ↦ *Arr ty m* (*S'a*(*nat* (*n* − *2*) ↦ *vs*[*Suc 0* := *the* (*S'a* (*nat n*))] ! *Suc 0*))),*inih*,
[([*vs*[*Suc 0* := *the* (*S'a* (*nat n*)),
2 := *the* (*S'a* (*nat* (*n* − *1*))),
3 := *the* (*S'a* (*nat* (*n* − *2*)))]!*3*],
vs[*Suc 0* := *the* (*S'a* (*nat n*)),
2 := *the* (*S'a* (*nat* (*n* − *1*))),
3 := *the* (*S'a* (*nat* (*n* − *2*)))],
safeP, *sigSafeMain*,
nat (*24* + *int* (*the* (*cdm* (*l*, *i*))) +
trAddr pcc (*the* (*cdm* (*l*, *i*)) + *incPop*)), *aa*, *ba*)]))
apply (*unfold exec-all-def*)
apply (*erule rtrancl-trans*)
prefer *2*
apply (*rule r-into-rtrancl*)
apply (*clarify*)
apply (*unfold JVMExec.exec.simps*)
apply (*unfold Let-def*)
apply (*subgoal-tac* (*fst* (*snd* (*snd* (*snd* (*snd* (*the* (*method'* (*P'*, *safeP*) *sigSafe-*

Main)))))) !
(*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc*
(*Suc* (*Suc* (*Suc* (*Suc* (*the* (*cdm* (*l*, *i*)))))))))))))))))))))))))))) =
Goto (*trAddr pcc* (*the* (*cdm* (*l*, *i*)) + *incPop*)),*simp*)
apply (*erule nth-via-drop-append*)

apply (*subgoal-tac drop pcc* (*extractBytecode P′*) =
[*Tableswitch 1* (*int* (*length ps*)) *ps*] @ *bytecode′* ∧ *ctm* (*nat* (*ps*!*i*))
= *Some i*)
apply (*elim conjE*)
prefer *2* **apply** (*rule Tableswitch-POPCONT*)
apply (*subgoal-tac P′*⊢ (*None*, *sha*((*stackC*, *topf*) ↦ *Intg* (*n* − *2*), (*heapC*, *k0f*) ↦ *vs*[*Suc 0* := *the* (*S′a* (*nat n*)),
2 := *the*
(*S′a* (*nat* (*n* − *1*)))] ! *2*),
h(*la* ↦ *Arr ty m* (*S′a*(*nat* (*n* − *2*) ↦ *vs*[*Suc 0* := *the*
(*S′a* (*nat n*))] ! *Suc 0*))),*inih*,
[([*vs*[*Suc 0* := *the* (*S′a* (*nat n*)),
2 := *the* (*S′a* (*nat* (*n* − *1*))),
3 := *the* (*S′a* (*nat* (*n* − *2*)))]!*3*],
vs[*Suc 0* := *the* (*S′a* (*nat n*)),
2 := *the* (*S′a* (*nat* (*n* − *1*))),
3 := *the* (*S′a* (*nat* (*n* − *2*)))],
safeP, *sigSafeMain*,
nat (*24* + *int* (*the* (*cdm* (*l*, *i*))) +
trAddr pcc (*the* (*cdm* (*l*, *i*)) + *incPop*)), *aa*, *ba*)])
−*jvm*→
(*None*, *sha*((*stackC*, *topf*) ↦ *Intg* (*n* − *2*), (*heapC*, *k0f*) ↦
vs[*Suc 0* := *the* (*S′a* (*nat n*)),
2 := *the*
(*S′a* (*nat* (*n* − *1*)))] ! *2*),
h(*la* ↦ *Arr ty m* (*S′a*(*nat* (*n* − *2*) ↦ *vs*[*Suc 0* := *the*
(*S′a* (*nat n*))] ! *Suc 0*))),*inih*,
[([],
vs[*Suc 0* := *the* (*S′a* (*nat n*)),
2 := *the* (*S′a* (*nat* (*n* − *1*))),
3 := *the* (*S′a* (*nat* (*n* − *2*)))],
safeP, *sigSafeMain*,
if′ (((*the-Intg* (*vs*[*Suc 0* := *the* (*S′a* (*nat n*)),
2 := *the* (*S′a* (*nat* (*n* − *1*))),
3 := *the* (*S′a* (*nat* (*n* − *2*)))] ! *3*) < *1*) ∨
(*int* (*length ps*) < *the-Intg* (*vs*[*Suc 0* := *the* (*S′a* (*nat n*)),
2 := *the* (*S′a* (*nat* (*n* − *1*))),
3 := *the* (*S′a* (*nat* (*n* − *2*)))] ! *3*))))
(*nat* (*24* + *int* (*the* (*cdm* (*l*, *i*))) + *trAddr pcc* (*the* (*cdm* (*l*, *i*))
+ *incPop*)) +
nat (*ps* ! *nat* (*int* (*length ps*))))

```
                (nat (24 + int (the (cdm (l, i))) + trAddr pcc (the (cdm (l, i))
+ incPop)) +
                    nat (ps ! nat (the-Intg (vs[Suc 0 := the (S'a (nat n)),
                                                2 := the (S'a (nat (n - 1))),
                                               3 := the (S'a (nat (n - 2)))] ! 3)
- 1))),
                aa, ba)]))
apply (unfold exec-all-def)
apply (erule rtrancl-trans)
prefer 2
 apply (rule r-into-rtrancl)
 apply (clarify)
 apply (unfold extractBytecode-def)
 apply (unfold JVMExec.exec.simps)
 apply (unfold Let-def)
 apply (subgoal-tac (fst (snd (snd (snd (snd (the (method' (P', safeP) sigSafe-
Main)))))))) !
                (nat (24 + int (the (cdm (l, i))) + trAddr pcc (the (cdm (l, i))
+ incPop))) =
                Tableswitch 1 (int (length ps)) ps,simp)
apply (simp add: if'-def)
apply (subgoal-tac
 pcc = nat (24 + int (the (cdm (l, i))) + (int pcc - int (the (cdm (l, i)) +
incPop))))
 prefer 2 apply (simp add: incPop-def)
apply (simp add: trAddr-def)
apply (erule nth-via-drop)
apply simp



apply (rule-tac x=hm in exI)
apply (rule-tac x=k in exI)
apply (rule-tac x=k1 in exI)
apply (rule-tac x=(l1,0) in exI)
apply (rule-tac x=Val v1 # S' in exI)


apply (rule-tac x=sha((stackC, topf) ↦ Intg (n - 2),
                  (heapC, k0f) ↦ vs[Suc 0 := the (S'a (nat n)), 2 := the (S'a
(nat (n - 1)))] ! 2) in exI)
apply (rule-tac x=h(la ↦ Arr ty m (S'a(nat (n - 2) ↦ vs[Suc 0 := the (S'a
(nat n))] ! Suc 0))) in exI)
apply (rule-tac x=inih in exI)
apply (rule-tac x=vs[Suc 0 := the (S'a (nat n)),
                     2 := the (S'a (nat (n - 1))),
                     3 := the (S'a (nat (n - 2)))] in exI)
apply (rule-tac x=if' (((the-Intg (vs[Suc 0 := the (S'a (nat n)),
                                   2 := the (S'a (nat (n - 1))),
```

```
                                        3 := the (S'a (nat (n − 2)))] ! 3) < 1) ∨
                                (int (length ps) < the-Intg (vs[Suc 0 := the (S'a (nat n)),
                                                               2 := the (S'a (nat (n − 1))),
                                                             3 := the (S'a (nat (n − 2)))] ! 3))))
                            (nat (24 + int (the (cdm (l, i))) + trAddr pcc (the (cdm (l, i))
+ incPop)) +
                                  nat (ps ! nat (int (length ps))))
                            (nat (24 + int (the (cdm (l, i))) + trAddr pcc (the (cdm (l, i))
+ incPop)) +
                                  nat (ps ! nat (the-Intg (vs[Suc 0 := the (S'a (nat n)),
                                                               2 := the (S'a (nat (n − 1))),
                                                             3 := the (S'a (nat (n − 2)))] ! 3)
− 1))) in exI)
apply (rule-tac x=ref in exI)

apply (rule-tac x=nat (the-Intg (the ((sha((stackC, topf) ↦ Intg (n − 2),
                                          (heapC, k0f) ↦ vs[Suc 0 := the (S'a (nat n)),
                                                              2 := the (S'a (nat (n − 1)))]
! 2)) (heapC, kf)))) in exI)

apply (rule-tac x=nat (the-Intg (the ((sha((stackC, topf) ↦ Intg (n − 2),
                                          (heapC, k0f) ↦ vs[Suc 0 := the (S'a (nat n)),
                                                              2 := the (S'a (nat (n − 1)))]
! 2)) (heapC, k0f)))) in exI)

apply (rule-tac x=la in exI)
apply (rule-tac x=ty in exI)
apply (rule-tac x=m in exI)
apply (rule-tac x=S'a(nat (n − 2) ↦ vs[Suc 0 := the (S'a (nat n))] ! Suc 0) in
exI)

apply (rule-tac x= n − 2 in exI)
apply (rule-tac x=l' in exI)
apply (rule-tac x=regS in exI)
apply (rule-tac x=l'' in exI)
apply (rule-tac x=m' in exI)
apply (rule-tac x=m'' in exI)
apply (rule-tac x=d in exI)
apply (rule-tac x=g in exI)


apply (rule conjI, rule refl)
apply (rule conjI) apply (unfold trAddr-def)
                   apply (rule refl)
apply (rule conjI, rule refl)
apply (rule conjI, rule refl)
apply (rule conjI) apply clarsimp
                   apply (simp add: Sf-def add: topf-def)
                   apply (simp add: stackC-def add: heapC-def)
```

apply (*rule conjI*, *assumption*)
apply (*rule conjI*) **apply** *clarsimp*
apply (*simp add*: *stackC-def add*: *heapC-def*)
apply (*simp add*: *kf-def add*: *k0f-def*)
apply (*rule conjI*) **apply** (*erule activeCells*)
apply (*rule conjI*) **apply** (*erule activeCells-2*,*assumption*)
apply (*erule activeCells-2*,*assumption*)
apply (*rule conjI*, *simp*)
apply (*rule conjI*) **apply** (*subgoal-tac length vs = 10*)
apply (*simp add*: *stackC-def add*: *heapC-def*)
apply (*rule length-vs*)
apply (*rule conjI*) **apply** *clarsimp*
apply (*simp add*: *stackC-def add*: *heapC-def*)
apply (*simp add*: *regionsf-def add*: *k0f-def*)
apply (*rule conjI*) **apply** *clarsimp*
apply (*rule conjI*) **apply** *clarsimp*
apply (*simp add*: *dirCellC-def add*: *stackC-def*)
apply (*simp add*: *heapC-def add*: *safeDirf-def*)
apply (*rule conjI*) **apply** *clarsimp*
apply (*rule conjI*) **apply** *clarsimp*
apply (*rule conjI*) **apply** *clarsimp*
apply (*rule conjI*) **apply** (*subgoal-tac length vs = 10*)
apply (*rule equivH-POPCONT*,*assumption*+)
apply *simp*
apply (*rule length-vs*)
apply (*rule conjI*) **apply** *clarsimp*
apply (*rule conjI*) **apply** *clarsimp*
apply (*subgoal-tac length vs = 10*)
apply (*rule equivS2-POPCONT*,*simp*,*assumption*+)
apply (*rule length-vs*)
apply (*rule conjI*) **apply** (*subgoal-tac length vs = 10*)
apply *clarsimp*
apply (*rule equivS-k1*,*assumption*+)
apply (*rule length-vs*)
by (*rule equivpc-POPCONT*)

end

theory *dem-PRIMOP*
imports *../JVMSAFE/JVMExec SVM2JVM SVMSemantics CertifSVM2JVM*
dem-translation
begin

no-translations *Norm s* == (*None*,*s*)

no-translations *ex-table-of m* == *snd* (*snd* (*snd m*))

declare *trInstr.simps* [*simp del*]
declare *equivS.simps* [*simp del*]
declare *extractBytecode-def* [*simp del*]
declare *initConsTable-def* [*simp del*]
declare *exec-instr.simps* [*simp del*]

lemma *equivS-PRIMOP*:
equivS (*Val v1* # *Val v2* # *S*) *S'* *n ctm d g* ⟹ *n* >= *1*
apply (*induct S*)
apply (*unfold equivS.simps*, *clarsimp*)
by (*case-tac a*,*simp-all*)

lemma *equivS2-PRIMOP*:
⟦ *2* < *length vs*; *equivS* (*Val* (*IntT i*) # *Val* (*IntT i'*) # *S'*) *S'a n ctm d g*⟧
⟹ *equivS* (*Val* (*execOp oper* (*IntT i*) (*IntT i'*)) # *S'*)
(*S'a*(*nat* (*n* − *1*) ↦ *applyBinOp oper* (*the* (*S'a* (*nat n*)))
(*vs*[*Suc 0* := *the* (*S'a* (*nat* (*n* − *1*)))] ! *Suc 0*))) (*n* − *1*) *ctm d g*
apply (*simp add*: *equivS.simps*, *clarsimp*)
apply (*rule conjI*)
apply (*case-tac oper*,*simp-all*)
apply (*drule equivS-ge-n*)
apply (*erule-tac x=1* **in** *allE*,*simp*)
by (*subgoal-tac* −*1* + *n*= *n* − *1*,*simp*,*simp*)

lemma *activeCells-PRIMOP*:
⟦ *la* ∉ *activeCells regS h* (*nat* (*the-Intg* (*the* (*sha* (*heapC*, *kf*)))))⟧
⟹ *activeCells regS h* (*nat* (*the-Intg* (*the* (*sha* (*heapC*, *kf*))))) =
activeCells regS (*h*(*la* ↦ *Arr ty m* (*S'a*(*nat* (*n* − *1*) ↦
applyBinOp oper (*the* (*S'a* (*nat n*))) (*vs*[*Suc 0* := *the* (*S'a* (*nat* (*n*
− *1*)))] ! *Suc 0*)))))
(*nat* (*the-Intg* (*the* (*sha* (*heapC*, *kf*)))))
apply (*frule l-not-in-cellReg*)
apply (*unfold activeCells-def*,*auto*)
apply (*unfold region-def*, *simp add*: *Let-def*, *elim conjE*)
apply (*rule-tac x=j* **in** *exI*,*simp*)
apply (*rule cellReg-step*)
apply (*rule cellReg-basic*)
apply (*erule-tac x=j* **in** *allE*)
apply (*rule cellsReg-monotone-1*,*assumption*+)
apply (*erule l-not-in-regs*)
apply (*simp add*: *Let-def*, *elim conjE*)
apply (*rule-tac x=j* **in** *exI*, *simp*)
apply (*rule cellReg-step*)
apply (*rule cellReg-basic*)

apply (*erule-tac x=j* **in** *allE*)
apply (*erule cellsReg-monotone-2*,*assumption*+)
by (*erule l-not-in-regs*)

lemma *domH-PRIMOP*:
⟦ *la* ∉ *activeCells regS h* (*nat* (*the-Intg* (*the* (*sha* (*heapC*, *kf*))))));
∀ *l*∈*dom H*. ∃ *l'*. *l'* = *the* (*g l*) ∧ *equivC* (*the* (*H l*)) *h l'* (*the* (*h l'*)) (*nat* (*the-Intg* (*the* (*sha* (*heapC*, *kf*))))) *com regS d g* ⟧
⟹ ∀ *l*∈*dom H*.
∃ *l'*. *l'* = *the* (*g l*) ∧
equivC (*the* (*H l*))
(*h*(*la* ↦
Arr ty m (*S'a*(*nat* (*n* − *1*) ↦ *applyBinOp oper* (*the* (*S'a* (*nat n*))) (*vs*[*Suc 0* := *the* (*S'a* (*nat* (*n* − *1*)))] ! *Suc 0*)))))
l' (*the* ((*h*(*la* ↦
Arr ty m
(*S'a*(*nat* (*n* − *1*) ↦ *applyBinOp oper* (*the* (*S'a* (*nat n*))) (*vs*[*Suc 0* := *the* (*S'a* (*nat* (*n* − *1*)))] ! *Suc 0*)))))
l'))
(*nat* (*the-Intg* (*the* (*sha* (*heapC*, *kf*)))))
com regS d g
apply (*frule l-not-in-cellReg*)
apply (*rule ballI*)
apply (*erule-tac x=l* **in** *ballE*)
prefer *2* **apply** *simp*
apply (*erule exE*)
apply (*rule-tac x=l'* **in** *exI*) **apply** (*elim conjE*)
apply (*rule conjI*, *assumption*)
apply *clarsimp*
apply (*rule conjI*) **apply** (*rule impI*)
apply *clarsimp* **apply** (*simp add*: *region-def*)
apply *clarsimp*
apply (*rule-tac x=Obj* **in** *exI*)
apply (*rule-tac x=flds* **in** *exI*) **apply** *simp*
apply (*simp add*: *region-def*) **apply** (*elim conjE*)
apply (*rule cellReg-step*)
apply (*rule cellReg-basic*)
apply (*erule-tac x=j* **in** *allE*)
apply (*rule cellsReg-monotone-1*,*assumption*+)
by (*erule l-not-in-regs*)

lemma *equivH-PRIMOP*:
⟦*S* = *Val* (*IntT ia*) # *Val* (*IntT i'*) # *S'*; ((*hm*, *k*), *k0*, (*l*, *i*), *S*) = ((*H*, *ka*), *k0a*, *PC*, *Sa*);
k' = *nat* (*the-Intg* (*the* (*sha* (*heapC*, *kf*)))); *k0'* = *nat* (*the-Intg* (*the* (*sha* (*heapC*, *k0f*)))); *sha* (*stackC*, *Sf*) = *Some* (*Addr la*);

distinct [*la*, *l'*, *l''*]; *activeCells regS h k'* ∩ {*la*, *l'*, *l''*} = {}; *h la* = *Some* (*Arr ty m S'a*);
sha (*stackC*, *topf*) = *Some* (*Intg n*); *sha* (*heapC*, *regionsf*) = *Some* (*Addr l'*); *h l'* = *Some* (*Arr ty m' regS*);
sha (*dirCellC*, *safeDirf*) = *Some* (*Addr l''*); *h l''* = *Some* (*Arr ty m'' d*); *inj-on g* (*dom H*); *equivH* (*H*, *ka*) *h k' com regS d g*
⟧
⟹ *equivH* (*hm*, *k*)
(*h*(*la* ↦ *Arr ty m* (*S'a*(*nat* (*n* − *1*) ↦ *applyBinOp oper* (*the* (*S'a* (*nat n*))) (*vs*[*Suc 0* := *the* (*S'a* (*nat* (*n* − *1*)))] ! *Suc 0*)))))
(*nat* (*the-Intg* (*the* ((*sha*((*stackC*, *topf*) ↦ *Intg* (*n* − *1*))) (*heapC*, *kf*))))) *com regS d g*
apply (*simp add*: *heapC-def add*: *stackC-def*)
apply (*simp add*: *kf-def add*: *k0f-def*)
apply (*fold heapC-def*, *fold kf-def*)
apply (*unfold equivH.simps*, *elim conjE*)
apply (*rule conjI*, *assumption*)
apply (*rule conjI*, *frule activeCells-PRIMOP*,*simp*)
apply (*rule conjI*, *assumption*)
by (*rule domH-PRIMOP*,*assumption*+)

declare *exec-instr.simps* [*simp*]

lemma *execSVMInstr-PRIMOP* :
⟦ (*P'*, *cdm*, *ctm*, *com*) = *trSVM2JVM* ((*svms*, *ctmap*), *ini*, *ct*, *ah*, *ai*, *bc*);
cdm , *ctm*, *com* ⊢ ((*hm*, *k*), *k0*, (*l*, *i*), *S*) ≙ *S1'*;
(*fst* (*the* (*map-of svms l*)) ! *i*) = *PRIMOP oper*;
execSVMInst (*PRIMOP oper*) (*map-of ct*) (*hm*, *k*) *k0* (*l*, *i*) *S* = *Either.Right S2*;
drop (*the* (*cdm* (*l*, *i*))) (*extractBytecode P'*) =
trInstr (*the* (*cdm* (*l*, *i*))) *cdm' ctm' com pcc* (*PRIMOP oper*) @ *bytecode'*
⟧ ⟹ ∃ *v' sh' dh' ih' fms'* .
P' ⊢ *S1'* −*jvm*→ (*v'*,*sh'*,*dh'*,*ih'*, *fms'*) ∧
cdm , *ctm*, *com* ⊢ *S2* ≙ (*v'*,*sh'*,*dh'*,*ih'*, *fms'*)

apply (*case-tac S1'*)
apply (*rename-tac v tup*)
apply (*case-tac tup*)
apply (*rename-tac sh tup*)
apply (*case-tac tup*)
apply (*rename-tac dh tup*)
apply (*case-tac tup*)
apply (*rename-tac ih fms*)
apply (*simp*)

apply (*subgoal-tac* ∃ *i i′ S′*. *S* = *Val* (*IntT i*) # *Val* (*IntT i′*) #*S′*)
prefer *2*

apply (*case-tac S*,*simp-all*)
apply (*insert RightNotUndefined*, *erule-tac x*= *S2* **in** *allE*,*force*)
apply *clarsimp*
apply (*case-tac af*,*simp-all*)
apply (*case-tac Vala*,*simp-all*)
apply (*erule-tac x*= *S2* **in** *allE*,*force*)
apply (*case-tac list*,*simp*,*simp-all*)
apply *clarsimp*
apply (*case-tac a*,*simp-all*)
apply (*case-tac Vala*,*simp-all*)
apply (*insert RightNotUndefined*, *erule-tac x*= *S2* **in** *allE*,*force*)
apply (*erule-tac x*= *S2* **in** *allE*,*force*)
apply (*erule-tac x*= *S2* **in** *allE*,*force*)
apply (*erule-tac x*= *S2* **in** *allE*,*force*)

apply (*unfold equivState-def*)
apply (*elim exE*,*elim conjE*)

apply (*rule-tac x*=*None* **in** *exI*)
apply (*rule-tac x*=*sha*((*stackC*, *topf*) ↦ *Intg* (*n* − *1*)) **in** *exI*)
apply (*rule-tac x*=*h*(*la* ↦ *Arr ty m* (*S′a*(*nat* (*n* − *1*) ↦ *applyBinOp oper* (*the* (*S′a* (*nat n*)))
(*vs*[*Suc 0* := *the* (*S′a* (*nat* (*n* − *1*)))]
! *Suc 0*)))) **in** *exI*)
apply (*rule-tac x*=*inih* **in** *exI*)
apply (*rule-tac x*=[([], *vs*[*Suc 0* := *the* (*S′a* (*nat* (*n* − *1*)))], *safeP*, *sigSafeMain*, *the* (*cdm* (*l*, *i*)) +
1 + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1*, *ref*)] **in** *exI*)

apply (*subgoal-tac* ∃ *vs pc* . *fms*=([],*vs*,*safeP*,*sigSafeMain*,*pc*,*ref*)#[])
prefer *2* **apply** *simp*
apply (*erule exE*)+
apply (*unfold exec-all-def*)
apply (*rule conjI*)

apply (*subgoal-tac PC* = (*l*,*i*)) **prefer** *2* **apply** *simp*
apply *simp* **apply** (*elim conjE*) **apply** *clarsimp*
apply (*subgoal-tac P* ⊢ (*None*,*sha*,*h*,*inih*,[([], *vs*, *safeP*, *sigSafeMain*, *the* (*cdm* (*l*,*i*)), *ag*,*bd*)]) −*jvm*→

```
                (None,sha,h,inih,[([the (sha (stackC,Sf))], vs, safeP, sigSafeMain,
the (cdm (l,i)) + 1, ag,bd)]))
apply (unfold exec-all-def)
apply (erule rtrancl-trans)


prefer 2
 apply (rule r-into-rtrancl)
 apply (clarify)
 apply (unfold JVMExec.exec.simps)
 apply (unfold Let-def)
 apply (unfold extractBytecode-def)
 apply (subgoal-tac (fst (snd (snd (snd (snd (the (method' (P', safeP) sigSafe-
Main)))))) !
                     the (cdm (l, i))) = Getstatic Sf stackC,simp)
 apply (unfold trInstr.simps)
apply (erule nth-via-drop-append)


apply (drule drop-Suc-append)
apply (subgoal-tac P'⊢ (None, sha, h, inih,
                        [([the (sha (stackC, Sf))],
                        vs, safeP, sigSafeMain, the (cdm (l, i)) + 1, ag, bd)])
                    −jvm→
                        (None,sha,h,inih,
                        [([the (sha (stackC,topf)),
                           the (sha (stackC,Sf))],
                        vs, safeP, sigSafeMain, the (cdm (l, i)) + 1 + 1, ag, bd)]))
apply (unfold exec-all-def)
apply (erule rtrancl-trans)
prefer 2
 apply (rule r-into-rtrancl)
 apply (clarify)
 apply (unfold JVMExec.exec.simps)
 apply (unfold Let-def)
 apply (subgoal-tac (fst (snd (snd (snd (snd (the (method' (P', safeP) sigSafe-
Main)))))) ! Suc (the (cdm (l, i)))) =
                     Getstatic topf stackC,simp)
 apply (erule nth-via-drop-append)


apply (drule drop-Suc-append)
apply (subgoal-tac P'⊢ (None,sha,h,inih,
                        [([the (sha (stackC,topf)),
                           the (sha (stackC,Sf))],
                        vs, safeP, sigSafeMain, the (cdm (l, i)) + 1 + 1, ag, bd)])
                    −jvm→
                        (None,sha,h,inih,
                        [([Intg 1,
```

```
                    the (sha (stackC,topf)),
                    the (sha (stackC,Sf))],
                  vs, safeP, sigSafeMain, the (cdm (l, i)) + 1 + 1 + 1, ag, bd)]))
apply (unfold exec-all-def)
apply (erule rtrancl-trans)
prefer 2
 apply (rule r-into-rtrancl)
 apply (clarify)
 apply (unfold JVMExec.exec.simps)
 apply (unfold Let-def)
 apply (subgoal-tac (fst (snd (snd (snd (snd (the (method' (P', safeP) sigSafe-
Main)))))) ! (Suc (Suc (the (cdm (l, i)))))) =
                  LitPush (Intg 1),simp)
apply (erule nth-via-drop-append)



apply (drule drop-Suc-append)
apply (subgoal-tac P'⊢ (None,sha,h,inih,
                    [([Intg 1,
                       the (sha (stackC,topf)),
                       the (sha (stackC,Sf))],
                      vs, safeP, sigSafeMain, the (cdm (l, i)) + 1 + 1 + 1, ag, bd)])
                  −jvm→
                     (None,sha,h,inih,
                    [([Intg (the-Intg (the (sha (stackC, topf)))− 1),
                       the (sha (stackC,Sf))],
                     vs, safeP, sigSafeMain, the (cdm (l, i)) + 1 + 1 + 1 + 1, ag,
bd)]))
apply (unfold exec-all-def)
apply (erule rtrancl-trans)
prefer 2
 apply (rule r-into-rtrancl)
 apply (clarify)
 apply (unfold JVMExec.exec.simps)
 apply (unfold Let-def)
 apply (subgoal-tac (fst (snd (snd (snd (snd (the (method' (P', safeP) sigSafe-
Main)))))) ! (Suc (Suc (Suc (the (cdm (l, i))))))) =
                  BinOp Substract,simp)
apply (erule nth-via-drop-append)



apply (drule drop-Suc-append)
apply (subgoal-tac P'⊢ (None,sha,h,inih,
                    [([Intg (the-Intg (the (sha (stackC, topf)))− 1),
                       the (sha (stackC,Sf))],
                      vs, safeP, sigSafeMain, the (cdm (l, i)) + 1 + 1 + 1 + 1,
```

```
ag, bd)])
                  −jvm→
                     (None,sha,h,inih,
                     [([Intg (the-Intg (the (sha (stackC, topf)))− 1),
                        the (sha (stackC,Sf)),
                        Intg (the-Intg (the (sha (stackC, topf)))− 1),
                        the (sha (stackC,Sf))],
                      vs, safeP, sigSafeMain, the (cdm (l, i)) + 1 + 1 + 1 + 1 +
1, ag, bd)]))
apply (unfold exec-all-def)
apply (erule rtrancl-trans)
prefer 2
 apply (rule r-into-rtrancl)
 apply (clarify)
 apply (unfold JVMExec.exec.simps)
 apply (unfold Let-def)
 apply (subgoal-tac (fst (snd (snd (snd (snd (the (method′ (P′, safeP) sigSafe-
Main)))))) !
                   (Suc (Suc (Suc (Suc (the (cdm (l, i))))))) =
                    Dup2,simp)
apply (erule nth-via-drop-append)


apply (drule drop-Suc-append)
apply (subgoal-tac P′⊢ (None,sha,h,inih,
                     [([Intg (the-Intg (the (sha (stackC, topf)))− 1),
                        the (sha (stackC,Sf)),
                        Intg (the-Intg (the (sha (stackC, topf)))− 1),
                        the (sha (stackC,Sf))],
                      vs, safeP, sigSafeMain, the (cdm (l, i)) + 1 + 1 + 1 + 1 +
1, ag, bd)])
                  −jvm→
                     (None,sha,h,inih,
                     [([the (S′a (nat (n − 1))),
                        Intg (the-Intg (the (sha (stackC, topf)))− 1),
                        the (sha (stackC,Sf))],
                      vs, safeP, sigSafeMain, the (cdm (l, i)) + 1 + 1 + 1 + 1 +
1 + 1, ag, bd)]))
apply (unfold exec-all-def)
apply (erule rtrancl-trans)
prefer 2
 apply (rule r-into-rtrancl)
 apply (clarify)
 apply (unfold JVMExec.exec.simps)
 apply (unfold Let-def)
 apply (subgoal-tac (fst (snd (snd (snd (snd (the (method′ (P′, safeP) sigSafe-
Main)))))) !
                   (Suc (Suc (Suc (Suc (Suc (the (cdm (l, i)))))))) =
                    ArrLoad,simp)
```

```
  apply (frule equivS-PRIMOP, simp)
  apply (simp add: raise-system-xcpt-def)
apply (erule nth-via-drop-append)


apply (drule drop-Suc-append)
apply (subgoal-tac P'⊢ (None,sha,h,inih,
                    [([the (S'a (nat (n − 1))),
                       Intg (the-Intg (the (sha (stackC, topf)))− 1),
                       the (sha (stackC,Sf))],
                      vs, safeP, sigSafeMain, the (cdm (l, i)) + 1 + 1 + 1 + 1 + 1 +
1 + 1, ag, bd)])
                  −jvm→
                     (None,sha,h,inih,
                    [([Intg (the-Intg (the (sha (stackC, topf)))− 1),
                       the (sha (stackC,Sf))],
                      vs[Suc 0 := the (S'a (nat (n − 1)))],
                       safeP, sigSafeMain, the (cdm (l, i)) + 1 + 1 + 1 + 1 + 1
+ 1 + 1, ag, bd)]))
apply (unfold exec-all-def)
apply (erule rtrancl-trans)
prefer 2
 apply (rule r-into-rtrancl)
 apply (clarify)
 apply (unfold JVMExec.exec.simps)
 apply (unfold Let-def)
 apply (subgoal-tac (fst (snd (snd (snd (snd (the (method' (P', safeP) sigSafe-
Main)))))) !
                     (Suc (Suc (Suc (Suc (Suc (Suc (the (cdm (l, i)))))))))) =
                      Store 1,simp)
apply (erule nth-via-drop-append)


apply (drule drop-Suc-append)
apply (subgoal-tac P'⊢ (None,sha,h,inih,
                    [([Intg (the-Intg (the (sha (stackC, topf)))− 1),
                       the (sha (stackC,Sf))],
                      vs[Suc 0 := the (S'a (nat (n − 1)))],
                       safeP, sigSafeMain, the (cdm (l, i)) + 1 + 1 + 1 + 1 + 1 +
1 + 1, ag, bd)])
                  −jvm→
                     (None,sha,h,inih,
                    [([the (sha (stackC,Sf)),
                       Intg (the-Intg (the (sha (stackC, topf)))− 1),
                       the (sha (stackC,Sf))],
                      vs[Suc 0 := the (S'a (nat (n − 1)))],
                       safeP, sigSafeMain, the (cdm (l, i)) + 1 + 1 + 1 + 1 + 1
```

```
+ 1 + 1 + 1, ag, bd)]))
apply (unfold exec-all-def)
apply (erule rtrancl-trans)
prefer 2
 apply (rule r-into-rtrancl)
 apply (clarify)
 apply (unfold JVMExec.exec.simps)
 apply (unfold Let-def)
 apply (subgoal-tac (fst (snd (snd (snd (snd (the (method' (P', safeP) sigSafe-
Main)))))) !
                    (Suc (Suc (Suc (Suc (Suc (Suc (the (cdm (l, i)))))))))) =
                     Getstatic Sf stackC,simp)
apply (erule nth-via-drop-append)



apply (drule drop-Suc-append)
apply (subgoal-tac P'⊢  (None,sha,h,inih,
                        [([the (sha (stackC,Sf)),
                           Intg (the-Intg (the (sha (stackC, topf)))− 1),
                           the (sha (stackC,Sf))],
                          vs[Suc 0 := the (S'a (nat (n − 1)))],
                           safeP, sigSafeMain, the (cdm (l, i)) + 1 + 1 + 1 + 1 + 1
+ 1 + 1 + 1 + 1, ag, bd)])
                    −jvm→
                        (None,sha,h,inih,
                       [([the (sha (stackC,topf)),
                          the (sha (stackC,Sf)),
                          Intg (the-Intg (the (sha (stackC, topf)))− 1),
                          the (sha (stackC,Sf))],
                         vs[Suc 0 := the (S'a (nat (n − 1)))],
                          safeP, sigSafeMain, the (cdm (l, i)) + 1 + 1 + 1 + 1 + 1
+ 1 + 1 + 1 + 1 + 1, ag, bd)]))
apply (unfold exec-all-def)
apply (erule rtrancl-trans)
prefer 2
 apply (rule r-into-rtrancl)
 apply (clarify)
 apply (unfold JVMExec.exec.simps)
 apply (unfold Let-def)
 apply (subgoal-tac (fst (snd (snd (snd (snd (the (method' (P', safeP) sigSafe-
Main)))))) !
                   (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (the (cdm (l, i))))))))))))
=
                       Getstatic topf stackC,simp)
apply (erule nth-via-drop-append)
```

apply (*drule drop-Suc-append*)
apply (*subgoal-tac P'⊢* (*None*,*sha*,*h*,*inih*,
[([*the* (*sha* (*stackC*,*topf*)),
the (*sha* (*stackC*,*Sf*)),
Intg (*the-Intg* (*the* (*sha* (*stackC*, *topf*)))− *1*),
the (*sha* (*stackC*,*Sf*))],
vs[*Suc 0* := *the* (*S'a* (*nat* (*n* − *1*)))],
safeP, *sigSafeMain*, *the* (*cdm* (*l*, *i*)) + *1* + *1* + *1* + *1* + *1*
+ *1* + *1* + *1* + *1*, *ag*, *bd*)])
−*jvm*→
(*None*,*sha*,*h*,*inih*,
[([*the* (*S'a* (*nat n*)),
Intg (*the-Intg* (*the* (*sha* (*stackC*, *topf*)))− *1*),
the (*sha* (*stackC*,*Sf*))],
vs[*Suc 0* := *the* (*S'a* (*nat* (*n* − *1*)))],
safeP, *sigSafeMain*, *the* (*cdm* (*l*, *i*)) + *1* + *1* + *1* + *1* + *1*
+ *1* + *1* + *1* + *1* + *1*, *ag*, *bd*)]))
apply (*unfold exec-all-def*)
apply (*erule rtrancl-trans*)
prefer *2*
apply (*rule r-into-rtrancl*)
apply (*clarify*)
apply (*unfold JVMExec.exec.simps*)
apply (*unfold Let-def*)
apply (*subgoal-tac* (*fst* (*snd* (*snd* (*snd* (*snd* (*the* (*method'* (*P'*, *safeP*) *sigSafe-Main*))))))) !
(*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*the* (*cdm* (*l*,
i)))))))))))) =
ArrLoad,*simp*)

apply (*frule equivS-PRIMOP*, *simp*)
apply (*simp add*: *raise-system-xcpt-def*)
apply (*erule nth-via-drop-append*)

apply (*drule drop-Suc-append*)
apply (*subgoal-tac P'⊢* (*None*,*sha*,*h*,*inih*,
[([*the* (*S'a* (*nat n*)),
Intg (*the-Intg* (*the* (*sha* (*stackC*, *topf*)))− *1*),
the (*sha* (*stackC*,*Sf*))],
vs[*Suc 0* := *the* (*S'a* (*nat* (*n* − *1*)))],
safeP, *sigSafeMain*, *the* (*cdm* (*l*, *i*)) + *1* + *1* + *1* + *1* + *1*
+ *1* + *1* + *1* + *1* + *1*, *ag*, *bd*)])
−*jvm*→
(*None*,*sha*,*h*,*inih*,
[([*vs*[*Suc 0* := *the* (*S'a* (*nat* (*n* − *1*)))] ! *Suc 0*,
the (*S'a* (*nat n*)),
Intg (*the-Intg* (*the* (*sha* (*stackC*, *topf*)))− *1*),
the (*sha* (*stackC*,*Sf*))],

```
                    vs[Suc 0 := the (S'a (nat (n − 1)))],
                     safeP, sigSafeMain, the (cdm (l, i)) + 1 + 1 + 1 + 1 + 1
+ 1 + 1 + 1 + 1 + 1 + 1, ag, bd)]))
apply (unfold exec-all-def)
apply (erule rtrancl-trans)
prefer 2
 apply (rule r-into-rtrancl)
 apply (clarify)
 apply (unfold JVMExec.exec.simps)
 apply (unfold Let-def)
 apply (subgoal-tac (fst (snd (snd (snd (snd (the (method' (P', safeP) sigSafe-
Main)))))) !
                    (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (the (cdm
(l, i)))))))))))))) =
                    Load 1,simp)
apply (erule nth-via-drop-append)



apply (drule drop-Suc-append)
apply (subgoal-tac P'⊢ (None,sha,h,inih,
                     [([vs[Suc 0 := the (S'a (nat (n − 1)))] ! Suc 0,
                        the (S'a (nat n)),
                        Intg (the-Intg (the (sha (stackC, topf)))− 1),
                        the (sha (stackC,Sf))],
                      vs[Suc 0 := the (S'a (nat (n − 1)))],
                       safeP, sigSafeMain, the (cdm (l, i)) + 1 + 1 + 1 + 1 + 1
+ 1 + 1 + 1 + 1 + 1 + 1, ag, bd)])
                  −jvm→
                     (None,sha,h,inih,
                     [([applyBinOp oper (the (S'a (nat n))) (vs[Suc 0 := the (S'a
(nat (n − 1)))] ! Suc 0),
                        Intg (the-Intg (the (sha (stackC, topf)))− 1),
                        the (sha (stackC,Sf))],
                      vs[Suc 0 := the (S'a (nat (n − 1)))],
                       safeP, sigSafeMain, the (cdm (l, i)) + 1 + 1 + 1 + 1 + 1
+ 1 + 1 + 1 + 1 + 1 + 1 + 1, ag, bd)]))
apply (unfold exec-all-def)
apply (erule rtrancl-trans)
prefer 2
 apply (rule r-into-rtrancl)
 apply (clarify)
 apply (unfold JVMExec.exec.simps)
 apply (unfold Let-def)
 apply (subgoal-tac (fst (snd (snd (snd (snd (the (method' (P', safeP) sigSafe-
Main)))))) !
                    (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (the
(cdm (l, i)))))))))))))) =
                    BinOp oper,simp)
```

apply (*erule nth-via-drop-append*)

apply (*drule drop-Suc-append*)
apply (*subgoal-tac P'⊢* (*None*,*sha*,*h*,*inih*,
[([*applyBinOp oper* (*the* (*S'a* (*nat n*))) (*vs*[*Suc 0* := *the* (*S'a* (*nat* (*n* − *1*)))] ! *Suc 0*),
Intg (*the-Intg* (*the* (*sha* (*stackC*, *topf*)))− *1*),
the (*sha* (*stackC*,*Sf*))],
vs[*Suc 0* := *the* (*S'a* (*nat* (*n* − *1*)))],
safeP, *sigSafeMain*, *the* (*cdm* (*l*, *i*)) + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1*, *ag*, *bd*)])
−*jvm*→
(*None*,*sha*,
h(*la* ↦ *Arr ty m* (*S'a*(*nat* (*n* − *1*) ↦ *applyBinOp oper* (*the* (*S'a* (*nat n*))) (*vs*[*Suc 0* := *the* (*S'a* (*nat* (*n* − *1*)))] !
Suc 0)))),
inih,
[([],
vs[*Suc 0* := *the* (*S'a* (*nat* (*n* − *1*)))],
safeP, *sigSafeMain*, *the* (*cdm* (*l*, *i*)) + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1*, *ag*, *bd*)]))
apply (*unfold exec-all-def*)
apply (*erule rtrancl-trans*)
prefer *2*
apply (*rule r-into-rtrancl*)
apply (*clarify*)
apply (*unfold JVMExec.exec.simps*)
apply (*unfold Let-def*)
apply (*subgoal-tac* (*fst* (*snd* (*snd* (*snd* (*snd* (*the* (*method'* (*P'*, *safeP*) *sigSafeMain*)))))) !
(*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*the* (*cdm* (*l*, *i*))))))))))))))) =
ArrStore,*simp*)
apply (*simp add*: *raise-system-xcpt-def*)
apply (*frule equivS-PRIMOP*,*simp*)
apply (*erule nth-via-drop-append*)

apply (*drule drop-Suc-append*)
apply (*subgoal-tac P'⊢* (*None*,*sha*,
h(*la* ↦ *Arr ty m* (*S'a*(*nat* (*n* − *1*) ↦ *applyBinOp oper* (*the* (*S'a* (*nat n*))) (*vs*[*Suc 0* := *the* (*S'a* (*nat* (*n* − *1*)))] !
Suc 0)))),
inih,
[([],
vs[*Suc 0* := *the* (*S'a* (*nat* (*n* − *1*)))],

```
                    safeP, sigSafeMain, the (cdm (l, i)) + 1 + 1 + 1 + 1 + 1
+ 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1, ag, bd)])
              −jvm→
                  (None,sha,
                    h(la ↦ Arr ty m (S'a(nat (n − 1) ↦ applyBinOp oper (the
(S'a (nat n))) (vs[Suc 0 := the (S'a (nat (n − 1)))] !
                      Suc 0)))),
                    inih,
                    [([the (sha (stackC,topf))],
                    vs[Suc 0 := the (S'a (nat (n − 1)))],
                    safeP, sigSafeMain, the (cdm (l, i)) + 1 + 1 + 1 + 1 + 1
+ 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1, ag, bd)]))
apply (unfold exec-all-def)
apply (erule rtrancl-trans)
prefer 2
 apply (rule r-into-rtrancl)
 apply (clarify)
 apply (unfold JVMExec.exec.simps)
 apply (unfold Let-def)
 apply (subgoal-tac (fst (snd (snd (snd (snd (the (method' (P', safeP) sigSafe-
Main)))))) !
                  (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc
(Suc (the (cdm (l, i))))))))))))))) =
                  Getstatic topf stackC,simp)
apply (erule nth-via-drop-append)


apply (drule drop-Suc-append)
apply (subgoal-tac P'⊢ (None,sha,
                    h(la ↦ Arr ty m (S'a(nat (n − 1) ↦ applyBinOp oper (the
(S'a (nat n))) (vs[Suc 0 := the (S'a (nat (n − 1)))] !
                      Suc 0)))),
                    inih,
                    [([the (sha (stackC,topf))],
                    vs[Suc 0 := the (S'a (nat (n − 1)))],
                    safeP, sigSafeMain, the (cdm (l, i)) + 1 + 1 + 1 + 1 + 1
+ 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1, ag, bd)])
              −jvm→
                  (None,sha,
                    h(la ↦ Arr ty m (S'a(nat (n − 1) ↦ applyBinOp oper (the
(S'a (nat n))) (vs[Suc 0 := the (S'a (nat (n − 1)))] !
                      Suc 0)))),
                    inih,
                    [([Intg 1,
                       the (sha (stackC,topf))],
                    vs[Suc 0 := the (S'a (nat (n − 1)))],
                    safeP, sigSafeMain, the (cdm (l, i)) + 1 + 1 + 1 + 1 + 1
+ 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1, ag, bd)]))
apply (unfold exec-all-def)
```

apply (*erule rtrancl-trans*)
prefer *2*
apply (*rule r-into-rtrancl*)
apply (*clarify*)
apply (*unfold JVMExec.exec.simps*)
apply (*unfold Let-def*)
apply (*subgoal-tac* (*fst* (*snd* (*snd* (*snd* (*snd* (*the* (*method'* (*P'*, *safeP*) *sigSafe-Main*)))))) !
(*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*the* (*cdm* (*l*, *i*))))))))))))))))) =
LitPush (*Intg 1*),*simp*)
apply (*erule nth-via-drop-append*)

apply (*drule drop-Suc-append*)
apply (*subgoal-tac P'*⊢ (*None*,*sha*,
h(*la* ↦ *Arr ty m* (*S'a*(*nat* (*n* − *1*) ↦ *applyBinOp oper* (*the* (*S'a* (*nat n*))) (*vs*[*Suc 0* := *the* (*S'a* (*nat* (*n* − *1*)))] !
Suc 0)))),
inih,
[([*Intg 1*,
the (*sha* (*stackC*,*topf*))],
vs[*Suc 0* := *the* (*S'a* (*nat* (*n* − *1*)))],
safeP, *sigSafeMain*, *the* (*cdm* (*l*, *i*)) + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1*, *ag*, *bd*)])
−*jvm*→
(*None*,*sha*,
h(*la* ↦ *Arr ty m* (*S'a*(*nat* (*n* − *1*) ↦ *applyBinOp oper* (*the* (*S'a* (*nat n*))) (*vs*[*Suc 0* := *the* (*S'a* (*nat* (*n* − *1*)))] !
Suc 0)))),
inih,
[([*Intg* (*n* − *1*)],
vs[*Suc 0* := *the* (*S'a* (*nat* (*n* − *1*)))],
safeP, *sigSafeMain*,
the (*cdm* (*l*, *i*)) + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1*, *ag*, *bd*)]))
apply (*unfold exec-all-def*)
apply (*erule rtrancl-trans*)
prefer *2*
apply (*rule r-into-rtrancl*)
apply (*clarify*)
apply (*unfold JVMExec.exec.simps*)
apply (*unfold Let-def*)
apply (*subgoal-tac* (*fst* (*snd* (*snd* (*snd* (*snd* (*the* (*method'* (*P'*, *safeP*) *sigSafe-Main*)))))) !
(*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*the* (*cdm* (*l*, *i*))))))))))))))))) =
BinOp Substract,*simp*)
apply (*erule nth-via-drop-append*)

apply (*drule drop-Suc-append*)
apply (*subgoal-tac P′⊢ (None,sha,*
h(la ↦ Arr ty m (S′a(nat (n − 1) ↦ applyBinOp oper (the (S′a (nat n))) (vs[Suc 0 := the (S′a (nat (n − 1)))] !
Suc 0)))),
inih,
[([Intg (n − 1)],
vs[Suc 0 := the (S′a (nat (n − 1)))],
safeP, sigSafeMain,
the (cdm (l, i)) + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1, ag, bd)])
−jvm→
(None,sha((stackC, topf) ↦ Intg (n − 1)),
h(la ↦ Arr ty m (S′a(nat (n − 1) ↦ applyBinOp oper (the (S′a (nat n))) (vs[Suc 0 := the (S′a (nat (n − 1)))] !
Suc 0)))),
inih,
[([],
vs[Suc 0 := the (S′a (nat (n − 1)))],
safeP, sigSafeMain,
the (cdm (l, i)) + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1, ag, bd)]))
apply (*unfold exec-all-def*)
apply (*erule rtrancl-trans*)
prefer *2*
apply (*rule r-into-rtrancl*)
apply (*clarify*)
apply (*unfold JVMExec.exec.simps*)
apply (*unfold Let-def*)
apply (*subgoal-tac (fst (snd (snd (snd (snd (the (method′ (P′, safeP) sigSafeMain)))))) !*
(Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (the (cdm (l, i))))))))))))))))))) =
Putstatic topf stackC,simp)
apply (*erule nth-via-drop-append*)
apply *simp*

apply (*frule nonJumping-Suc-pc*)
apply (*erule-tac sym* [**where** *t=PRIMOP oper*])
apply (*simp add: nonJumping.simps*)
apply (*simp add: trInstr.simps*)

apply (*rule-tac x=hm* **in** *exI*)
apply (*rule-tac x=k* **in** *exI*)
apply (*rule-tac x=k0* **in** *exI*)

apply (*rule-tac x=(l,Suc i)* **in** *exI*)
apply (*rule-tac x=Val (execOp oper (IntT ia) (IntT i')) # S'* **in** *exI*)

apply (*rule-tac x=sha((stackC, topf) ↦ Intg (n − 1))* **in** *exI*)
apply (*rule-tac x=h(la ↦ Arr ty m (S'a(nat (n − 1) ↦ applyBinOp oper (the (S'a (nat n)))*
(vs[Suc 0 := the (S'a (nat (n − 1)))]
! Suc 0))))) **in** *exI*)
apply (*rule-tac x=inih* **in** *exI*)
apply (*rule-tac x=vs[Suc 0 := the (S'a (nat (n − 1)))]* **in** *exI*)
apply (*rule-tac x=the (cdm (l, i)) + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1* **in** *exI*)
apply (*rule-tac x=ref* **in** *exI*)

apply (*rule-tac x=nat (the-Intg (the ((sha((stackC, topf) ↦ Intg (n − 1))) (heapC, kf))))* **in** *exI*)
apply (*rule-tac x=nat (the-Intg (the ((sha((stackC, topf) ↦ Intg (n − 1))) (heapC, k0f))))* **in** *exI*)

apply (*rule-tac x=la* **in** *exI*)
apply (*rule-tac x=ty* **in** *exI*)
apply (*rule-tac x=m* **in** *exI*)
apply (*rule-tac x=(S'a(nat (n − 1) ↦ applyBinOp oper (the (S'a (nat n))) (vs[Suc 0 := the (S'a (nat (n − 1)))] ! Suc 0)))* **in** *exI*)

apply (*rule-tac x=n − 1* **in** *exI*)
apply (*rule-tac x=l'* **in** *exI*)
apply (*rule-tac x=regS* **in** *exI*)
apply (*rule-tac x=l''* **in** *exI*)
apply (*rule-tac x=m'* **in** *exI*)
apply (*rule-tac x=m''* **in** *exI*)
apply (*rule-tac x=d* **in** *exI*)
apply (*rule-tac x=g* **in** *exI*)

apply (*rule conjI*, *clarsimp*)
apply (*rule conjI*, *rule refl*)
apply (*rule conjI*, *rule refl*)
apply (*rule conjI*, *rule refl*)
apply (*rule conjI*) **apply** *clarsimp*
apply (*rule conjI*, *assumption*)
apply (*rule conjI*) **apply** *clarsimp*
apply (*simp add*: *heapC-def add*: *stackC-def*)
apply (*rule conjI*) **apply** (*rule activeCells*,*assumption*)
apply (*rule conjI*) **apply** (*rule activeCells-2*,*assumption+*)
apply (*rule activeCells-2*,*assumption+*)
apply (*rule conjI*) **apply** *clarsimp*
apply (*rule conjI*) **apply** *clarsimp*
apply (*rule conjI*) **apply** *clarsimp*
apply (*rule conjI*) **apply** *clarsimp*

apply (*rule conjI*) **apply** *clarsimp*
apply (*rule conjI*) **apply** *clarsimp*
apply (*rule conjI*) **apply** *clarsimp*
apply (*rule conjI*) **apply** *clarsimp*
apply (*rule conjI*) **apply** (*rule equivH-PRIMOP*, *assumption*+)
apply (*rule conjI*) **apply** *simp*
apply (*rule conjI*) **apply** *clarsimp*
apply (*subgoal-tac length vs = 10*)
apply (*rule equivS2-PRIMOP*,*simp*, *assumption*+)
apply (*rule length-vs*)
apply (*rule conjI*) **apply** *simp*
apply (*simp add*: *heapC-def add*: *stackC-def*)
by *simp*

end

theory *dem-SLIDE*
imports *../JVMSAFE/JVMExec ../JVMSAFE/State SVM2JVM SVMSemantics CertifSVM2JVM*
dem-translation
begin

no-translations *Norm s == (None,s)*
no-translations *ex-table-of m == snd (snd (snd m))*

declare *trInstr.simps* [*simp del*]
declare *equivS.simps* [*simp del*]
declare *extractBytecode-def* [*simp del*]
declare *initConsTable-def* [*simp del*]

constdefs
slideArray :: [*entries-* ,*int*,*nat*,*nat*] ⇒ *entries-*
slideArray S ntop m n ≡ (
let bs = map (% *i* . *the* (*S* (*nat*(*ntop*)−*m*+*i*+*1*))) [*0*..<*m*]
in S ([*nat*(*ntop*)−*m*−*n*+*1*..<*nat*(*ntop*)−*n*+*1*] [↦] *bs*))

axioms *slide-method*:
⟦ *s* = (*None*, *shp*, *hp*, *ihp*,
([*Intg* (*int n*),*Intg* (*int m*)],*loc*,*safeP*, *sigSafeMain*,*pc*,*z1*,*z2*)#[]);
shp (*stackC*,*Sf*) = *Some* (*Addr l*);
shp (*StackC*,*topf*) = *Some* (*Intg ntop*);
hp l = *Some* (*Arr ty ma S'*);
Invoke-static stackC slide [*PrimT Integer*,*PrimT Integer*] =
fst(*snd*(*snd*(*snd*(*snd*(*the*(*method'* (*P'*,*safeP*) *sigSafeMain*)))))) !
pc
⟧ ⟹

$P' \vdash s\ -jvm\rightarrow\ (None,\ shp\ ((stackC,topf)\mapsto Intg\ (ntop-int\ n)),$
$hp(l \mapsto Arr\ ty\ ma\ (slideArray\ S'\ ntop\ m\ n)),$
$ihp,\ ([],loc,safeP,sigSafeMain,pc+1,z1,z2)\#[])$

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

lemma *activeCells-SLIDE*:
⟦ *la* ∉ *activeCells regS h* (*nat* (*the-Intg* (*the* (*sh* (*heapC*, *kf*)))))⟧
⟹ *activeCells regS h* (*nat* (*the-Intg* (*the* (*sh* (*heapC*, *kf*))))) =
activeCells regS (*h*(*la* ↦ *Arr ty ma* (*slideArray S' na m n*)))
(*nat* (*the-Intg* (*the* (*sh* (*heapC*, *kf*)))))
apply (*frule l-not-in-cellReg*)
apply (*unfold activeCells-def*,*auto*)
apply (*unfold region-def*, *simp add*: *Let-def*, *elim conjE*)
apply (*rule-tac x=j* **in** *exI*,*simp*)
apply (*rule cellReg-step*)
apply (*rule cellReg-basic*)
apply (*erule-tac x=j* **in** *allE*)
apply (*rule cellsReg-monotone-1*,*assumption+*)
apply (*erule l-not-in-regs*)
apply (*simp add*: *Let-def*, *elim conjE*)
apply (*rule-tac x=j* **in** *exI*, *simp*)
apply (*rule cellReg-step*)
apply (*rule cellReg-basic*)
apply (*erule-tac x=j* **in** *allE*)
apply (*erule cellsReg-monotone-2*,*assumption+*)
by (*erule l-not-in-regs*)

lemma *domH-SLIDE*:
⟦ *la* ∉ *activeCells regS h* (*nat* (*the-Intg* (*the* (*sha* (*heapC*, *kf*))))); ∀ *l*∈*dom H*. ∃ *l'*. *l'* = *the* (*g l*) ∧ *equivC* (*the* (*H l*)) *h l'* (*the* (*h l'*)) (*nat* (*the-Intg* (*the* (*sha* (*heapC*, *kf*))))) *com regS d g* ⟧
⟹ ∀ *l*∈*dom H*.
∃ *l'*. *l'* = *the* (*g l*) ∧
equivC (*the* (*H l*)) (*h*(*la* ↦ *Arr ty ma* (*slideArray S' na m n*))) *l'*
(*the* ((*h*(*la* ↦ *Arr ty ma* (*slideArray S' na m n*))) *l'*))
(*nat* (*the-Intg* (*the* (*sha* (*heapC*, *kf*))))) *com regS d g*
apply (*frule l-not-in-cellReg*)
apply (*rule ballI*)
apply (*erule-tac x=l* **in** *ballE*)
prefer *2* **apply** *simp*
apply (*erule exE*)
apply (*rule-tac x=l'* **in** *exI*) **apply** (*elim conjE*)
apply (*rule conjI*, *assumption*)

apply *clarsimp*
apply (*rule conjI*, *rule impI*)
apply (*simp add*: *region-def*)
apply (*rule impI*)
apply (*rule-tac x=Obj* **in** *exI*)
apply (*rule-tac x=flds* **in** *exI*) **apply** *simp*
apply (*simp add*: *region-def*)
apply *clarsimp*
apply (*rule cellReg-step*)
apply (*rule cellReg-basic*)
apply (*erule-tac x=j* **in** *allE*)
apply (*rule cellsReg-monotone-1*,*assumption*+)
by (*erule l-not-in-regs*)

lemma *equivH-SLIDE*:
⟦ ((*hm*, *k*), *k0*, (*l*, *i*), *S*) = ((*H*, *ka*), *k0a*, *PC*, *Sa*);
k' = *nat* (*the-Intg* (*the* (*sha* (*heapC*, *kf*))));
k0' = *nat* (*the-Intg* (*the* (*sha* (*heapC*, *k0f*))));
sha (*stackC*, *Sf*) = *Some* (*Addr la*);
distinct [*la*,*l'*,*l''*]; *h la* = *Some* (*Arr ty ma S'*);
sha (*stackC*, *topf*) = *Some* (*Intg na*);
sha (*heapC*, *regionsf*) = *Some* (*Addr l'*);
h l' = *Some* (*Arr ty m' regS*);
sha (*dirCellC*, *safeDirf*) = *Some* (*Addr l''*);
h l'' = *Some* (*Arr ty m'' d*); *inj-on g* (*dom H*);
equivH (*H*, *ka*) *h k' com regS d g*; *k0a* = *k0'*;
activeCells regS h k' ∩ {*la*, *l'*, *l''*} = {}⟧
⟹ *equivH* (*hm*, *k*) (*h*(*la* ↦ *Arr ty ma* (*slideArray S' na m n*)))
(*nat* (*the-Intg* (*the* ((*sha*((*stackC*, *topf*) ↦ *Intg* (*na* − *int n*))) (*heapC*, *kf*))))) *com regS d g*
apply (*simp add*: *heapC-def add*: *stackC-def*)
apply (*fold heapC-def*)
apply (*unfold equivH.simps*, *elim conjE*)
apply (*rule conjI*, *clarsimp*)
apply (*rule conjI*, *frule activeCells-SLIDE*,*simp*)
apply (*rule conjI*, *assumption*)
by (*rule domH-SLIDE*,*assumption*+)

axioms *equivS-concat*:
⟦ *equivS* (*S1@S2*) *S'* *ntop ctm d g*;
n = *length S1*;
∀ *i* < *n* . ∀ *z* . *S*¦*i* ≠ *Cont z*;
∀ *i* < *n* . *S'* (*nat ntop* − *i*) = *S''* (*nat ntop* − *i*);
equivS S3 S'' (*ntop* − *int n*) *ctm d g*
⟧ ⟹ *equivS* (*S1@S3*) *S''* *ntop ctm d g*

axioms *equivS-concat2*:
⟦ *equivS* (*S1@S2*) *S'* *ntop ctm d g*;

$n = length\ S1$;
$\forall\ i < n\ .\ \forall\ z\ .\ S!i \neq Cont\ z$
$\rrbracket \Longrightarrow equivS\ S2\ S'\ (ntop - int\ n)\ ctm\ d\ g$

axioms *equivS-same-subarray*:
$equivS\ (drop\ (m + n)\ S)$
$(S'([nat\ ntop - m - n + 1..<nat\ ntop - n + 1]\ [\mapsto]$
$map\ (\lambda i.\ the\ (S'\ (nat\ ntop - m + i + 1)))\ [0..<m]))$
$(ntop - int\ n - int\ m)\ ctm\ d\ g$
$\Longrightarrow equivS\ (drop\ (Suc\ m + n)\ S)$
$(S'([nat\ ntop - Suc\ m - n + 1..<nat\ ntop - n + 1]\ [\mapsto]$
$map\ (\lambda i.\ the\ (S'\ (nat\ ntop - Suc\ m + i + 1)))\ [0..<Suc\ m]))$
$(ntop - int\ n - int\ m - 1)\ ctm\ d\ g$

axioms *equivS-nth*:
$\llbracket\ equivS\ S\ S'\ ntop\ ctm\ d\ g;$
$n \leq\ length\ S;$
$\forall\ i < n\ .\ \forall\ z\ .\ S!i \neq Cont\ z;$
$i < n;$
$S!i = Val\ v\rrbracket \Longrightarrow equivV\ v\ (the\ (S'\ (nat\ ntop - i)))\ d\ g$

axioms *equivS-nth-reg*:
$\llbracket\ equivS\ S\ S'\ ntop\ ctm\ d\ g;$
$n \leq\ length\ S;$
$\forall\ i < n\ .\ \forall\ z\ .\ S!i \neq Cont\ z;$
$i < n;$
$S!i = Reg\ reg\rrbracket \Longrightarrow reg =\ nat\ (the\text{-}Intg\ (the\ (S'\ (nat\ ntop - i))))$

lemma *equivS-SLIDE* [*rule-format*]:
$equivS\ S\ S'\ ntop\ ctm\ d\ g \longrightarrow$
$length\ S \geq m + n \longrightarrow$
$(\forall\ i < m+n\ .\ \forall\ z\ .\ S!i \neq Cont\ z) \longrightarrow$
$equivS\ (take\ m\ S\ @\ drop\ (m + n)\ S)$
$(slideArray\ S'\ ntop\ m\ n)\ (ntop - int\ n)\ ctm\ d\ g$
apply (*unfold slideArray-def*)
apply (*unfold Let-def*)
apply (*induct m*)

apply *simp*
apply (*induct-tac n*)
apply *simp*
apply (*rule impI*)+
apply (*drule mp*,*simp*)+
apply (*subgoal-tac drop n S = S!n # drop (Suc n) S*)
apply *clarsimp*
apply (*thin-tac equivS ?x ?y ?z ?u ?v ?w*)

apply (*case-tac* $S!n$)
 apply (*simp add*: *equivS.simps*)
 apply (*elim conjE*)
 apply (*subgoal-tac ntop − int n − 1=ntop − (1 + int n)*,*simp*,*simp*)
 apply (*simp add*: *equivS.simps*)
 apply (*elim conjE*)
 apply (*subgoal-tac ntop − int n − 1=ntop − (1 + int n)*,*simp*,*simp*)
 apply (*erule-tac x=n* **in** *allE*)
 apply (*drule mp*,*simp*,*force*)
apply (*rule drop-nth3*,*simp*)

apply (*rule impI*)+
apply (*drule mp*,*simp*)+
apply (*subgoal-tac take m S @ (S!m # drop (Suc m + n) S) =*
 take (Suc m) S @ drop (Suc m + n) S)
apply (*erule-tac s=take m S @ (S!m # drop (Suc m + n) S)* **in** *subst*)
 prefer *2*
 apply (*subgoal-tac (take m S @ [S ! m]) @ drop (Suc m + n) S =*
 take m S @ S ! m # drop (Suc m + n) S)
 prefer *2* **apply** (*rule sym*,*rule concat1*)
 apply (*erule subst*)
 apply (*rule concat2*)
 apply (*rule sym*,*rule take-append3*,*simp*)

apply (*subgoal-tac ntop = int (length S) − 1*)
 prefer *2* **apply** (*erule equivS-length*)
apply (*rule equivS-concat*)
 apply (*simp del*: *upt-Suc*)
 apply (*subgoal-tac m=length (take m S)*)
  prefer *2* **apply** (*rule take-length*,*simp*)
 apply *simp*
 apply (*rule allI*,*rule impI*,*erule-tac x=i* **in** *allE*,*drule mp*,*simp*,*simp*)
 defer

apply (*case-tac* $S!m$)

apply (*simp only*: *equivS.simps*)
 apply (*rule conjI*,*simp*)
 apply (*rule conjI*) **defer**
apply (*subgoal-tac equivS (drop (m + n) S)*
 (S′([nat ntop − m − n + 1..<nat ntop − n + 1] [↦]
 map (λi. the (S′ (nat ntop − m + i + 1))) [0..<m]))
 ((ntop − int n)− int m) ctm d g)
 prefer *2* **apply** (*rule equivS-concat2*) **apply** *simp*
 apply (*rule take-length*,*simp*)

apply (*rule allI*,*rule impI*,*erule-tac x*=*i* **in** *allE*,*drule mp*,*simp*,*simp*)
apply *clarify*
apply (*erule equivS-same-subarray*)

apply (*simp only*: *equivS.simps*)
apply (*rule conjI*,*simp*)
apply (*rule conjI*) **defer**
apply (*subgoal-tac equivS* (*drop* (*m* + *n*) *S*)
(*S'*([*nat ntop* − *m* − *n* + *1*..<*nat ntop* − *n* + *1*] [↦]
map (λ*i*. *the* (*S'* (*nat ntop* − *m* + *i* + *1*))) [*0*..<*m*]))
((*ntop* − *int n*)− *int m*) *ctm d g*)
prefer *2* **apply** *clarify* **apply** (*erule equivS-concat2*)
apply (*rule take-length*,*simp*)
apply (*rule allI*,*rule impI*,*erule-tac x*=*i* **in** *allE*,*drule mp*,*simp*,*simp*)
apply *clarify*
apply (*erule equivS-same-subarray*)

apply (*erule-tac x*=*m* **in** *allE*)
apply (*drule mp*,*simp*)
apply (*erule-tac x*=*x* **in** *allE*)
apply *force*

apply (*rule allI*,*rule impI*)
apply (*subgoal-tac* (*Suc* (*nat ntop* − (*m* + *n*))) + ((*m*−*i*)− *1*) = *nat* (*ntop* − *int n*) − *i*)
apply (*erule-tac s*=*Suc* (*nat ntop* − (*m* + *n*)) + ((*m*−*i*)− *1*) **in** *subst*)
apply (*subgoal-tac* (*S'*([*Suc* (*nat ntop* − (*m* + *n*))..<*Suc* (*nat ntop* − *n*)] [↦]
map (λ*i*. *the* (*S'* (*Suc* (*nat ntop* − *m* + *i*)))) [*0*..<*m*]))
(*Suc* (*nat ntop* − (*m* + *n*)) + ((*m* − *i*)− *1*))=
Some ((*map* (λ*i*. *the* (*S'* (*Suc* (*nat ntop* − *m* + *i*)))) [*0*..<*m*])!((*m* − *i*)− *1*)))
prefer *2* **apply** (*rule map-upds-nth*)
apply *simp*
apply *clarify*
apply (*rule-tac s*=*Some* (*map* (λ*i*. *the* (*S'* (*Suc* (*nat* (*int* (*length S*) − *1*) − *m* + *i*)))) [*0*..<*m*]
! (*m* − *i* − *1*)) **in** *subst*)
prefer *2* **apply** *simp*
apply (*thin-tac ?x*=*?y*)
apply (*subgoal-tac* (*nat* (*int* (*length S*) − *1*) − *Suc m* − *n* + *1*)+(*m*−*i*)=
Suc (*nat* (*int* (*length S*) − *1*) − (*m* + *n*)) + (*m* − *i* − *1*))
apply (*erule-tac s*=*nat* (*int* (*length S*) − *1*) − *Suc m* − *n* + *1* + (*m* − *i*) **in** *subst*)
apply (*subgoal-tac* (*S'*([*nat* (*int* (*length S*) − *1*) − *Suc m* − *n* + *1*..<*nat* (*int* (*length S*) − *1*) − *n* + *1*]

```
            [↦] map (λi. the (S' (nat (int (length S) − 1) − Suc m + i + 1)))
[0..<Suc m]))
            (nat (int (length S) − 1) − Suc m − n + 1 + (m − i))=
             Some ((map (λi. the (S' (nat (int (length S) − 1) − Suc m + i +
1))) [0..<Suc m])
            ! (m−i)))
 prefer 2 apply (rule map-upds-nth)
 apply (erule additions1,simp)
apply (rule-tac s=Some (map (λi. the (S' (nat (int (length S) − 1) − Suc m +
i + 1))) [0..<Suc m] !
              (m − i)) and
      t=(S'([nat (int (length S) − 1) − Suc m − n + 1..<nat (int (length S) −
1) − n + 1] [↦]
         map (λi. the (S' (nat (int (length S) − 1) − Suc m + i + 1))) [0..<Suc
m]))
        (nat (int (length S) − 1) − Suc m − n + 1 + (m − i)) in subst)
apply (rule sym,assumption)
apply (thin-tac ?x=?y)
apply (simp only: nth-map-upt)
apply (subgoal-tac Suc (nat (int (length S) − 1) − m + (0 + (m − i − 1)))=
                nat (int (length S) − 1) − Suc m + (0 + (m − i)) + 1)
apply (erule-tac s=Suc (nat (int (length S) − 1) − m + (0 + (m − i − 1))) in
subst,rule refl)


apply (simp (no-asm))
apply (erule additions2,simp)
apply (simp (no-asm))
apply (erule additions3,simp)
apply (simp (no-asm))


apply (subgoal-tac (nat (int (length S) − 1) − Suc m − n + 1) + 0
      = nat (int (length S) − 1 − int n − int m))
apply (erule-tac s=(nat (int (length S) − 1) − Suc m − n + 1) + 0 in subst)
apply (subgoal-tac
     (S'([nat (int (length S) − 1) − Suc m − n + 1..<nat (int (length S) − 1)
− n + 1] [↦]
       map (λi. the (S' (nat (int (length S) − 1) − Suc m + i + 1))) [0..<Suc
m]))
      (nat (int (length S) − 1) − Suc m − n + 1 + 0)=
       Some (map (λi. the (S' (nat (int (length S) − 1) − Suc m + i + 1)))
[0..<Suc m] ! 0))
 prefer 2 apply (rule map-upds-nth) apply simp
 defer
apply (rule-tac t=(S'([nat (int (length S) − 1) − Suc m − n + 1..<nat (int
(length S) − 1) − n + 1] [↦]
      map (λi. the (S' (nat (int (length S) − 1) − Suc m + i + 1))) [0..<Suc
m]))
```

```
      (nat (int (length S) - 1) - Suc m - n + 1 + 0) and
       s=Some (map (λi. the (S' (nat (int (length S) - 1) - Suc m + i + 1)))
[0..<Suc m] ! 0)
     in subst) apply (rule sym,assumption)
apply (thin-tac ?x=Some ?y)
apply (subgoal-tac
     map (λi. the (S' (nat (int (length S) - 1) - Suc m + i + 1))) [0..<Suc m]
! 0
      =(λi. the (S' (nat (int (length S) - 1) - Suc m + i + 1))) (0+0))
 prefer 2 apply (rule nth-map-upt, simp (no-asm))
apply (rule-tac s=the (S' (nat (int (length S) - 1) - Suc m + (0 + 0) + 1))
       and t=map (λi. the (S' (nat (int (length S) - 1) - Suc m + i + 1)))
[0..<Suc m] ! 0
      in subst,rule sym,assumption)
apply (rule-tac s=nat ntop - m and t=nat (int (length S) - 1) - Suc m + (0
+ 0) + 1 in subst)
apply (simp (no-asm)) apply clarify
apply (erule additions4)
apply (simp (no-asm),clarify)
apply (erule-tac n=Suc m + n in equivS-nth)
 apply (assumption,rule allI,rule impI,force,simp,assumption)
apply (simp (no-asm))
apply (erule additions5)
 prefer 2 apply (rule additions6,simp)


apply (subgoal-tac (nat (int (length S) - 1) - Suc m - n + 1) + 0
      = nat (int (length S) - 1 - int n - int m))
apply (erule-tac s=(nat (int (length S) - 1) - Suc m - n + 1) + 0 in subst)
apply (subgoal-tac
     (S'([nat (int (length S) - 1) - Suc m - n + 1..<nat (int (length S) - 1)
- n + 1] [↦]
       map (λi. the (S' (nat (int (length S) - 1) - Suc m + i + 1))) [0..<Suc
m]))
      (nat (int (length S) - 1) - Suc m - n + 1 + 0)=
       Some (map (λi. the (S' (nat (int (length S) - 1) - Suc m + i + 1)))
[0..<Suc m] ! 0))
 prefer 2 apply (rule map-upds-nth) apply simp
 defer
apply (rule-tac t=(S'([nat (int (length S) - 1) - Suc m - n + 1..<nat (int
(length S) - 1) - n + 1] [↦]
      map (λi. the (S' (nat (int (length S) - 1) - Suc m + i + 1))) [0..<Suc
m]))
     (nat (int (length S) - 1) - Suc m - n + 1 + 0) and
      s=Some (map (λi. the (S' (nat (int (length S) - 1) - Suc m + i + 1)))
[0..<Suc m] ! 0)
     in subst) apply (rule sym,assumption)
apply (thin-tac ?x=Some ?y)
apply (subgoal-tac
```

```
    map (λi. the (S' (nat (int (length S) − 1) − Suc m + i + 1))) [0..<Suc m]
! 0
    =(λi. the (S' (nat (int (length S) − 1) − Suc m + i + 1))) (0+0))
 prefer 2 apply (rule nth-map-upt, simp (no-asm))
apply (rule-tac s=the (S' (nat (int (length S) − 1) − Suc m + (0 + 0) + 1))
      and t=map (λi. the (S' (nat (int (length S) − 1) − Suc m + i + 1)))
[0..<Suc m] ! 0
      in subst,rule sym,assumption)
apply (rule-tac s=nat ntop − m and t=nat (int (length S) − 1) − Suc m + (0
+ 0) + 1 in subst)
apply (simp (no-asm)) apply clarify
apply (erule additions4)
apply (simp (no-asm),clarify)
apply (erule-tac n=Suc m + n in equivS-nth-reg)
 apply (assumption,rule allI,rule impI,force,simp,assumption)
apply (simp (no-asm))
apply (erule additions5)
apply (rule additions6,simp)
done

axioms good-stack-SLIDE:
  ⟦ (P', cdm, ctm, com) = trSVM2JVM ((svms, ctmap), ini, ct, ah, ai, bc);
     cdm , ctm, com ⊢ ((hm, k), k0, (l, i), S) ≙ S1';
     (fst (the (map-of svms l)) ! i) = SLIDE m n
   ⟧ ⟹ length S ≥ m + n ∧ (∀ i < m+n . ∀ z . S!i ≠ Cont z)

lemma execSVMInstr-SLIDE :
  ⟦ (P', cdm, ctm, com) = trSVM2JVM ((svms, ctmap), ini, ct, ah, ai, bc);
     cdm , ctm, com ⊢ ((hm, k), k0, (l, i), S) ≙ S1';
     (fst (the (map-of svms l)) ! i) = SLIDE m n;
     execSVMInst (SLIDE m n) (map-of ct) (hm, k) k0 (l, i) S = Either.Right
S2;
     drop (the (cdm (l, i))) (extractBytecode P') =
     trInstr (the (cdm (l, i))) cdm' ctm' com pcc (SLIDE m n) @ bytecode'
   ⟧ ⟹  ∃ v' sh' dh' ih' fms' .
   P' ⊢ S1' −jvm→ (v',sh',dh',ih', fms') ∧
    cdm , ctm, com ⊢ S2 ≙ (v',sh',dh',ih', fms') ∧
    cdm , ctm, com ⊢ S2 ≙ (v',sh',dh',ih', fms')


apply (subgoal-tac length S ≥ m + n ∧ (∀ i < m+n . ∀ z . S!i ≠ Cont z))
 prefer 2 apply (rule good-stack-SLIDE,assumption+)
apply (elim conjE)


apply (case-tac S1')
apply (rename-tac v tup)
apply (case-tac tup)
```

apply (*rename-tac sh tup*)
apply (*case-tac tup*)
apply (*rename-tac dh tup*)
apply (*case-tac tup*)
apply (*rename-tac ih fms*)
apply (*simp*)

apply (*unfold equivState-def*)
apply (*elim exE*,*elim conjE*)

apply (*rule-tac x=None* **in** *exI*)
apply (*rule-tac x=sha*((*stackC*, *topf*) ↦ *Intg* (*na* − *int*(*n*))) **in** *exI*)
apply (*rule-tac x=h*(*la* ↦ *Arr ty ma* (*slideArray S' na m n*)) **in** *exI*)
apply (*rule-tac x=ih* **in** *exI*)
apply (*rule-tac x=*[([], *vs*, *safeP*, *sigSafeMain*, *the* (*cdm* (*l*, *i*)) + *1* + *1* + *1*, *ref*)] **in** *exI*)
apply (*rule conjI*)
apply (*unfold extractBytecode-def*)
apply (*clarify*)

apply (*subgoal-tac P'*⊢ (*None*,*sha*,*h*,*inih*,[([], *vs*, *safeP*, *sigSafeMain*, *the* (*cdm* (*l*,*i*)), *ag*,*bd*)])
  −*jvm*→ (*None*,*sha*,*h*,*inih*,[([*Intg* (*int m*)], *vs*, *safeP*, *sigSafeMain*, *the* (*cdm* (*l*,*i*)) + *1*, *ag*,*bd*)]))
apply (*unfold exec-all-def*)
apply (*erule rtrancl-trans*)

prefer *2*
apply (*rule r-into-rtrancl*)
apply (*clarify*)
apply (*unfold JVMExec.exec.simps*)
apply (*unfold Let-def*)
apply (*subgoal-tac* (*fst* (*snd* (*snd* (*snd* (*snd* (*the* (*method'* (*P'*, *safeP*) *sigSafeMain*)))))) !
                the (*cdm* (*l*, *i*))) = *LitPush* (*Intg* (*int m*)),*simp*)

apply (*unfold trInstr.simps*)
apply (*erule nth-via-drop-append*)

apply (*drule drop-Suc-append*)
apply (*subgoal-tac P'*⊢ (*None*,*sha*,*h*,*inih*,[([*Intg* (*int m*)],
                vs, *safeP*, *sigSafeMain*, *the* (*cdm* (*l*,*i*))+*1*, *ag*,*bd*)])
  −*jvm*→ (*None*,*sha*,*h*,*inih*,[([*Intg* (*int n*), *Intg* (*int m*)],

vs, *safeP*, *sigSafeMain*, (*the* (*cdm* (*l*,*i*)) + *1*)+*1*, *ag*,*bd*)]))
apply (*unfold exec-all-def*)
apply (*erule rtrancl-trans*)
prefer *2*
apply (*rule r-into-rtrancl*)
apply (*clarify*)
apply (*unfold JVMExec.exec.simps*)
apply (*unfold Let-def*)
apply (*subgoal-tac* (*fst* (*snd* (*snd* (*snd* (*snd* (*the* (*method'* (*P'*, *safeP*) *sigSafeMain*))))))) !
Suc (*the* (*cdm* (*l*, *i*)))) = *LitPush* (*Intg* (*int n*))) **apply** (*simp*)
apply (*erule nth-via-drop-append*)

apply (*drule drop-Suc-append*)
apply (*fold exec-all-def*)
apply (*rule slide-method*)
apply (*simp*,*assumption*,*simp*,*simp*)

apply (*rule sym*)
apply (*rule-tac ys*=[] **in** *nth-via-drop-append*)
apply *simp*

apply (*frule nonJumping-Suc-pc*)
apply (*erule-tac sym* [**where** *t*=*SLIDE m n*])
apply (*simp add*: *nonJumping.simps*)
apply (*simp add*: *trInstr.simps*)

apply (*rule-tac x*=*hm* **in** *exI*)
apply (*rule-tac x*=*k* **in** *exI*)
apply (*rule-tac x*=*k0* **in** *exI*)
apply (*rule-tac x*=(*l*,*Suc i*) **in** *exI*)
apply (*rule-tac x*=*take m S* @ *drop* (*m* + *n*) *S* **in** *exI*)

apply (*rule-tac x*=*sha*((*stackC*, *topf*) ↦ *Intg* (*na* − *int n*)) **in** *exI*)
apply (*rule-tac x*=*h*(*la* ↦ *Arr ty ma* (*slideArray S' na m n*)) **in** *exI*)
apply (*rule-tac x*=*ih* **in** *exI*)
apply (*rule-tac x*=*vs* **in** *exI*)
apply (*rule-tac x*=*the* (*cdm* (*l*, *i*)) + *1* + *1* + *1* **in** *exI*)
apply (*rule-tac x*=*ref* **in** *exI*)

apply (*rule-tac x*=*nat* (*the-Intg* (*the* ((*sha*((*stackC*, *topf*) ↦ *Intg* (*na* − *int n*))) (*heapC*, *kf*)))) **in** *exI*)
apply (*rule-tac x*=*nat* (*the-Intg* (*the* ((*sha*((*stackC*, *topf*) ↦ *Intg* (*na* − *int n*))) (*heapC*, *k0f*)))) **in** *exI*)

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

apply (*rule-tac x=ty* **in** *exI*)
apply (*rule-tac x=ma* **in** *exI*)
apply (*rule-tac x=*(*slideArray S′ na m n*) **in** *exI*)

apply (*rule-tac x=*(*na − int n*) **in** *exI*)
apply (*rule-tac x=l′* **in** *exI*)
apply (*rule-tac x=regS* **in** *exI*)
apply (*rule-tac x=l″* **in** *exI*)
apply (*rule-tac x=m′* **in** *exI*)
apply (*rule-tac x=m″* **in** *exI*)
apply (*rule-tac x=d* **in** *exI*)
apply (*rule-tac x=g* **in** *exI*)

apply (*rule conjI*, *clarsimp*)
apply (*rule conjI*, *rule refl*)
apply (*rule conjI*, *rule refl*)
apply (*rule conjI*, *rule refl*)
apply (*rule conjI*) **apply** *clarsimp*
apply (*rule conjI*, *assumption*)
apply (*rule conjI*) **apply** *clarsimp*
apply (*simp add*: *heapC-def add*: *stackC-def*)
apply (*rule conjI*) **apply** (*rule activeCells*,*assumption*)
apply (*rule conjI*) **apply** (*rule activeCells-2*,*assumption*+)
apply (*rule activeCells-2*,*assumption*+)
apply (*rule conjI*) **apply** *clarsimp*
apply (*rule conjI*) **apply** *clarsimp*
apply (*rule conjI*) **apply** *clarsimp*
apply (*rule conjI*) **apply** *clarsimp*
apply (*rule conjI*) **apply** *clarsimp*
apply (*rule conjI*) **apply** *clarsimp*
apply (*rule conjI*) **apply** *clarsimp*
apply (*rule conjI*) **apply** *clarsimp*
apply (*rule conjI*) **apply** (*rule equivH-SLIDE*, *assumption*+)
apply (*rule conjI*) **apply** *simp*
apply (*rule conjI*) **apply** (*rule equivS-SLIDE*)
apply (*simp*,*assumption*,*erule-tac x=ia* **in** *allE*,*drule mp*,*assumption*,*case-tac z*,*force*)
apply (*rule conjI*) **apply** *simp*
apply (*simp add*: *heapC-def add*: *stackC-def*)
by *simp*

end

theory *dem-MATCHN*
imports *../JVMSAFE/JVMExec SVM2JVM SVMSemantics CertifSVM2JVM*
begin

no-translations *Norm s == (None,s)*

no-translations *ex-table-of m == snd (snd (snd m))*

declare *trInstr.simps* [*simp del*]
declare *equivS.simps* [*simp del*]

axioms *equivS-MATCHN*:
n >= int off

declare *extractBytecode-def* [*simp del*]
declare *initConsTable-def* [*simp del*]

constdefs *if'* :: [*bool*, *'a*, *'a*] => *'a*
if' c a b ≡ *if c then a else b*

axioms *pc-MATCHN*:
if' (the-Intg (Intg (the-Intg (the (S' (nat (n − int off)))) − int v)) < 0 ∨
int (m + 1) < the-Intg (Intg (the-Intg (the (S' (nat (n − int off))))
− int v)))
(Suc (Suc (Suc (Suc (Suc (Suc (Suc (the (cdm (l, i)) +
nat (map (λn. trAddr n (the (cdm (l, i)) + incMatchN)) (map (λp. the
(cdm' (p, 0))) ps) !
nat (int (2 + m)))))))))))
(Suc (Suc (Suc (Suc (Suc (Suc (Suc (the (cdm (l, i)) +
nat (map (λn. trAddr n (the (cdm (l, i)) + incMatchN)) (map (λp. the
(cdm' (p, 0))) ps) !
nat (the-Intg (Intg (the-Intg (the (S' (nat (n − int off)))) − int v)) −
0)))))))))) =
the (cdm (case S !+ off of Val (IntT i) ⇒ let r = nat i − v; p = if 0 ≤ r ∧ r ≤
m − 1 then ps ! r else ps ! m in (p, 0)
| Val (BoolT b) ⇒ let p = if ¬ b then ps ! 0 else ps ! 1 in (p, 0)))

declare *if'-def* [*simp del*]

lemma *execSVMInstr-MATCHN* :
⟦ *(P', cdm, ctm, com) = trSVM2JVM ((svms, ctmap), ini, ct, ah, ai, bc);*
cdm , ctm, com ⊢ ((hm, k), k0, (l, i), S) ≜ *S1';*
(fst (the (map-of svms l)) ! i) = MATCHN off v m ps;
execSVMInst (MATCHN off v m ps) (map-of ct) (hm, k) k0 (l, i) S =
Either.Right S2;
drop (the (cdm (l, i))) (extractBytecode P') =
trInstr (the (cdm (l, i))) cdm' ctm' com pcc (MATCHN off v m ps) @
bytecode';

$$\begin{array}{l} \quad cdm' \subseteq_m cdm \\ ]\!] \Longrightarrow \quad \exists\ v'\ sh'\ dh'\ ih'\ fms'\ . \\ P' \vdash \ S1' -jvm\rightarrow (v',sh',dh',ih',\ fms') \wedge \\ cdm\ ,\ ctm,\ com\ \vdash \ S2 \ \stackrel{\triangle}{=}\ (v',sh',dh',ih',\ fms') \end{array}$$

apply (*case-tac S1'*)
apply (*rename-tac v1 tup*)
apply (*case-tac tup*)
apply (*rename-tac sh tup*)
apply (*case-tac tup*)
apply (*rename-tac dh tup*)
apply (*case-tac tup*)
apply (*rename-tac ih fms*)
apply (*simp*)

apply (*unfold equivState-def*)
apply (*elim exE,elim conjE*)

apply (*rule-tac x=None* **in** *exI*)
apply (*rule-tac x=sha* **in** *exI*)
apply (*rule-tac x=h* **in** *exI*)
apply (*rule-tac x=inih* **in** *exI*)
apply (*rule-tac x=*[([], *vs*,
safeP, *sigSafeMain*,
if' ((*the-Intg* (*Intg* (*the-Intg* (*the* (*S'* (*nat* (*n* − *int off*)))) − *int v*)) < *0*) ∨
(*the-Intg* (*Intg* (*the-Intg* (*the* (*S'* (*nat* (*n* − *int off*)))) − *int v*)) > (*int* (*m* + *1*))))
(*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*the* (*cdm* (*l*, *i*)) +
nat (*map* (λ*n*. *trAddr n* (*the* (*cdm* (*l*, *i*)) + *incMatchN*)) (*map* (λ*p*. *the* (*cdm'* (*p*, *0*))) *ps*) !
nat (*int* (*2* + *m*))))))))))
(*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*the* (*cdm* (*l*, *i*)) +
nat (*map* (λ*n*. *trAddr n* (*the* (*cdm* (*l*, *i*)) + *incMatchN*)) (*map* (λ*p*. *the* (*cdm'* (*p*, *0*))) *ps*) !
(*nat* (*the-Intg* (*Intg* (*the-Intg* (*the* (*S'* (*nat* (*n* − *int off*)))) − *int v*)) − *0*))))))))))),
ref)] **in** *exI*)

apply (*subgoal-tac* ∃ *vs pc* . *fms*=([],*vs*,*safeP*,*sigSafeMain*,*pc*,*ref*)#[])
prefer *2* **apply** *simp*
apply (*erule exE*)+

apply (*rule conjI*)

apply (*unfold exec-all-def*)
apply (*subgoal-tac PC = (l,i)*) **prefer** *2* **apply** *simp*
apply *simp* **apply** (*elim conjE*) **apply** *clarsimp*

apply (*subgoal-tac P'⊢ (None,sha,h,inih,[([], vs, safeP, sigSafeMain, the (cdm (l,i)), ag,bd)]) −jvm→*
(None,sha,h,inih,[([the (sha (stackC,Sf))], vs, safeP, sigSafeMain, the (cdm (l,i)) + 1, ag,bd)]))
apply (*unfold exec-all-def*)
apply (*erule rtrancl-trans*)

prefer *2*
apply (*rule r-into-rtrancl*)
apply (*clarify*)
apply (*unfold JVMExec.exec.simps*)
apply (*unfold Let-def*)
apply (*unfold extractBytecode-def*)
apply (*subgoal-tac (fst (snd (snd (snd (snd (the (method' (P', safeP) sigSafeMain))))))) !*
the (cdm (l, i))) = Getstatic Sf stackC,simp)
apply (*unfold trInstr.simps, unfold Let-def*)
apply (*erule nth-via-drop-append*)

apply (*drule drop-Suc-append*)
apply (*subgoal-tac P'⊢ (None, sha, h, inih,*
[([the (sha (stackC, Sf))],
vs, safeP, sigSafeMain, the (cdm (l, i)) + 1, ag, bd)])
−jvm→
(None,sha,h,inih,
[([the (sha (stackC,topf)),the (sha (stackC,Sf))],
vs, safeP, sigSafeMain, the (cdm (l, i)) + 1 + 1, ag, bd)]))
apply (*unfold exec-all-def*)
apply (*erule rtrancl-trans*)
prefer *2*
apply (*rule r-into-rtrancl*)
apply (*clarify*)
apply (*unfold JVMExec.exec.simps*)
apply (*unfold Let-def*)
apply (*subgoal-tac (fst (snd (snd (snd (snd (the (method' (P', safeP) sigSafeMain))))))) ! Suc (the (cdm (l, i)))) =*
Getstatic topf stackC,simp)
apply (*erule nth-via-drop-append*)

apply (*drule drop-Suc-append*)
apply (*subgoal-tac P'⊢ (None,sha,h,inih,*

```
                    [([the (sha (stackC,topf)),the (sha (stackC, Sf))],
                    vs, safeP, sigSafeMain, the (cdm (l, i)) + 1 + 1, ag, bd)])
                −jvm→
                    (None,sha,h,inih,
                     [([Intg (int off),
                        the (sha (stackC,topf)),the (sha (stackC, Sf))],
                       vs, safeP, sigSafeMain, the (cdm (l, i)) + 1 + 1 + 1, ag,
bd)]))
apply (unfold exec-all-def)
apply (erule rtrancl-trans)
prefer 2
 apply (rule r-into-rtrancl)
 apply (clarify)
 apply (unfold JVMExec.exec.simps)
 apply (unfold Let-def)
 apply (subgoal-tac (fst (snd (snd (snd (snd (the (method′ (P′, safeP) sigSafe-
Main)))))) ! (Suc (Suc (the (cdm (l, i)))))) =
                    LitPush (Intg (int off)),simp)
apply (erule nth-via-drop-append)



apply (drule drop-Suc-append)
apply (subgoal-tac P′⊢ (None,sha,h,inih,
                     [([Intg (int off),
                        the (sha (stackC,topf)),the (sha (stackC, Sf))],
                    vs, safeP, sigSafeMain, the (cdm (l, i)) + 1 + 1 + 1, ag, bd)])
                −jvm→
                    (None,sha,h,inih,
                    [([Intg ( n − int off),
                       the (sha (stackC, Sf))],
                      vs, safeP, sigSafeMain, the (cdm (l, i)) + 1 + 1 + 1 + 1,
ag, bd)]))
apply (unfold exec-all-def)
apply (erule rtrancl-trans)
prefer 2
 apply (rule r-into-rtrancl)
 apply (clarify)
 apply (unfold JVMExec.exec.simps)
 apply (unfold Let-def)
 apply (subgoal-tac (fst (snd (snd (snd (snd (the (method′ (P′, safeP) sigSafe-
Main)))))) ! (Suc (Suc (Suc (the (cdm (l, i)))))))  =
                    BinOp Substract,simp)
apply (erule nth-via-drop-append)



apply (drule drop-Suc-append)
apply (subgoal-tac P′⊢ (None,sha,h,inih,
```

```
                    [([Intg (n − int off),
                       the (sha (stackC, Sf))],
                     vs, safeP, sigSafeMain, the (cdm (l, i)) + 1 + 1 + 1 + 1,
ag, bd)])
                −jvm→
                    (None,sha,h,inih,
                    [([the (S' (nat (n − int off)))],
                    vs, safeP, sigSafeMain, the (cdm (l, i)) + 1 + 1 + 1 + 1 +
1, ag, bd)]))
apply (unfold exec-all-def)
apply (erule rtrancl-trans)
prefer 2
 apply (rule r-into-rtrancl)
 apply (clarify)
 apply (unfold JVMExec.exec.simps)
 apply (unfold Let-def)
 apply (subgoal-tac (fst (snd (snd (snd (snd (the (method' (P', safeP) sigSafe-
Main)))))) !
                  (Suc (Suc (Suc (Suc (the (cdm (l, i)))))))) =
                   ArrLoad)

apply (case-tac S !+ off,simp)
 apply (case-tac Vala)
  apply (simp,insert RightNotUndefined, erule-tac x= S2 in allE,force)
  apply simp apply (rename-tac n1)
        apply (subgoal-tac n >= int off) prefer 2 apply (rule equivS-MATCHN,
simp add: raise-system-xcpt-def)
  apply simp apply (subgoal-tac n >= int off) prefer 2 apply (rule equivS-MATCHN,
simp add: raise-system-xcpt-def)
  apply (simp,insert RightNotUndefined, erule-tac x= S2 in allE,force)
  apply (simp,insert RightNotUndefined, erule-tac x= S2 in allE,force)
apply (erule nth-via-drop-append)


apply (drule drop-Suc-append)
apply (subgoal-tac P'⊢ (None,sha,h,inih,
                    [([the (S' (nat (n − int off)))],
                    vs, safeP, sigSafeMain, the (cdm (l, i)) + 1 + 1 + 1 + 1 +
1, ag, bd)])
                −jvm→
                    (None,sha,h,inih,
                    [([Intg (int v),
                       the (S' (nat (n − int off)))],
                    vs, safeP, sigSafeMain, the (cdm (l, i)) + 1 + 1 + 1 + 1 +
1 + 1, ag, bd)]))
apply (unfold exec-all-def)
apply (erule rtrancl-trans)
prefer 2
 apply (rule r-into-rtrancl)
```

apply (*clarify*)
apply (*unfold JVMExec.exec.simps*)
apply (*unfold Let-def*)
apply (*subgoal-tac* (*fst* (*snd* (*snd* (*snd* (*snd* (*the* (*method'* (*P'*, *safeP*) *sigSafeMain*))))))) !
(*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*the* (*cdm* (*l*, *i*))))))))) =
LitPush (*Intg* (*int v*)),*simp*)
apply (*erule nth-via-drop-append*)

apply (*drule drop-Suc-append*)
apply (*subgoal-tac P'* $\vdash$ (*None*,*sha*,*h*,*inih*,
[([*Intg* (*int v*),
the (*S'* (*nat* (*n* − *int off*)))],
vs, *safeP*, *sigSafeMain*, *the* (*cdm* (*l*, *i*)) + *1* + *1* + *1* + *1* + *1* + *1*, *ag*, *bd*)])
−*jvm*→
(*None*,*sha*,*h*,*inih*,
[([*Intg* (*the-Intg* (*the* (*S'* (*nat* (*n* − *int off*)))) − *int v*)],
vs, *safeP*, *sigSafeMain*, *the* (*cdm* (*l*, *i*)) + *1* + *1* + *1* + *1* + *1* + *1* + *1*, *ag*, *bd*)]))
apply (*unfold exec-all-def*)
apply (*erule rtrancl-trans*)
prefer *2*
apply (*rule r-into-rtrancl*)
apply (*clarify*)
apply (*unfold JVMExec.exec.simps*)
apply (*unfold Let-def*)
apply (*subgoal-tac* (*fst* (*snd* (*snd* (*snd* (*snd* (*the* (*method'* (*P'*, *safeP*) *sigSafeMain*))))))) !
(*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*the* (*cdm* (*l*, *i*)))))))))) =
BinOp Substract,*simp*)
apply (*erule nth-via-drop-append*)

apply (*drule drop-Suc-append*)
apply (*subgoal-tac P'* $\vdash$ (*None*,*sha*,*h*,*inih*,
[([*Intg* (*the-Intg* (*the* (*S'* (*nat* (*n* − *int off*)))) − *int v*)],
vs, *safeP*, *sigSafeMain*, *the* (*cdm* (*l*, *i*)) + *1* + *1* + *1* + *1* + *1* + *1* + *1*, *ag*, *bd*)])
−*jvm*→
(*None*,*sha*,*h*,*inih*,
[([],
vs, *safeP*, *sigSafeMain*,
if' ((*the-Intg* (*Intg* (*the-Intg* (*the* (*S'* (*nat* (*n* − *int off*)))) − *int v*)) < *0*) ∨
(*the-Intg* (*Intg* (*the-Intg* (*the* (*S'* (*nat* (*n* − *int off*)))) − *int v*)) > (*int* (*m* + *1*))))

```
                    (Suc (Suc (Suc (Suc (Suc (Suc (the (cdm (l, i)) +
                                        nat (map (λn. trAddr n (the (cdm (l, i)) +
incMatchN)) (map (λp. the (cdm' (p, 0))) ps) !
                                    nat (int (2 + m ))))))))))
                     (Suc (Suc (Suc (Suc (Suc (Suc (the (cdm (l, i)) +
                                        nat (map (λn. trAddr n (the (cdm (l, i)) +
incMatchN)) (map (λp. the (cdm' (p, 0))) ps) !
                                    (nat (the-Intg (Intg (the-Intg (the (S' (nat (n −
int off)))) − int v)) − 0)))))))))),
                      ag, bd)]))
apply (unfold exec-all-def)
apply (erule rtrancl-trans)
prefer 2
 apply (rule r-into-rtrancl)
 apply (clarify)
 apply (unfold JVMExec.exec.simps)
 apply (unfold Let-def)
 apply (subgoal-tac (fst (snd (snd (snd (snd (the (method' (P', safeP) sigSafe-
Main)))))) !
                   (Suc (Suc (Suc (Suc (Suc (Suc (the (cdm (l, i))))))))))) =
                    Tableswitch 0 (int (m + 1)) (map (λn. trAddr n (the (cdm (l,
i)) + incMatchN))
                                                (map (λp. the (cdm' (p, 0))) ps)),simp)
apply (simp add: if'-def)
apply (erule nth-via-drop-append)
apply simp


apply (rule-tac x=hm in exI)
apply (rule-tac x=k in exI)
apply (rule-tac x=k0 in exI)
apply (rule-tac x=(case S !+ off of
    Val (IntT i) ⇒ let r = nat i − v;
                       p = if 0 ≤ r ∧ r ≤ m − 1 then ps ! r else ps ! m
                   in (p,0)
  | Val (BoolT b) ⇒ let p = if ¬ b then ps ! 0 else ps ! 1
                    in (p,0)) in exI)
apply (rule-tac x=S in exI)

apply (rule-tac x=sha in exI)
apply (rule-tac x=h in exI)
apply (rule-tac x=inih in exI)
apply (rule-tac x=vs in exI)
apply (rule-tac x=if' ((the-Intg (Intg (the-Intg (the (S' (nat (n − int off))))) −
int v)) < 0) ∨
                      (the-Intg (Intg (the-Intg (the (S' (nat (n − int off))))) − int
v)) > (int (m + 1))))
                   (Suc (Suc (Suc (Suc (Suc (Suc (the (cdm (l, i)) +
                                        nat (map (λn. trAddr n (the (cdm (l, i)) +
```

```
incMatchN)) (map (λp. the (cdm' (p, 0))) ps) !
                                    nat (int (2 + m ))))))))))))
                  (Suc (Suc (Suc (Suc (Suc (Suc (the (cdm (l, i)) +
                                       nat (map (λn. trAddr n (the (cdm (l, i)) +
incMatchN)) (map (λp. the (cdm' (p, 0))) ps) !
                                    (nat (the-Intg (Intg (the-Intg (the (S' (nat (n −
int off)))) − int v)) − 0)))))))))))) in exI)
apply (rule-tac x=ref in exI)

apply (rule-tac x=nat (the-Intg (the (sha (heapC, kf)))) in exI)

apply (rule-tac x= nat (the-Intg (the (sha (heapC, k0f)))) in exI)

apply (rule-tac x=la in exI)
apply (rule-tac x=ty in exI)
apply (rule-tac x=ma in exI)
apply (rule-tac x=S' in exI)

apply (rule-tac x=n in exI)
apply (rule-tac x=l' in exI)
apply (rule-tac x=regS in exI)
apply (rule-tac x=l'' in exI)
apply (rule-tac x=m' in exI)
apply (rule-tac x=m'' in exI)
apply (rule-tac x=d in exI)
apply (rule-tac x=g in exI)

apply (rule conjI)
apply simp
apply (case-tac S!+off)
  apply simp
  apply (case-tac Vala)
   apply simp
   apply (insert RightNotUndefined)
   apply (erule-tac x= S2 in allE, force)
  apply clarsimp
 apply clarsimp
 apply clarsimp
 apply (insert RightNotUndefined)
 apply (erule-tac x= S2 in allE, force)
apply clarsimp
apply (insert RightNotUndefined)
apply (erule-tac x= S2 in allE, force)
apply (rule conjI, rule refl)
apply (rule conjI, rule refl)
apply (rule conjI, rule refl)
apply (rule conjI, assumption)
apply (rule conjI, assumption)
apply (rule conjI, simp)
```

apply (*rule conjI*, *assumption*)
apply (*rule conjI*, *assumption*)
apply (*rule conjI*, *assumption*)
apply (*rule conjI*, *assumption*)
apply (*rule conjI*, *assumption*)
apply (*rule conjI*, *assumption*)
apply (*rule conjI*, *simp*)
apply (*rule conjI*, *simp*)
apply (*rule conjI*, *simp*)
apply (*rule conjI*, *simp*)
apply (*rule conjI*, *simp*)
apply (*rule conjI*, *clarsimp*)
by (*rule pc-MATCHN*)

end

theory *dem-BUILDENV*
imports *../JVMSAFE/JVMExec SVM2JVM SVMSemantics CertifSVM2JVM*
dem-translation
begin

no-translations *Norm s == (None,s)*

no-translations *ex-table-of m == snd (snd (snd m))*

declare *trInstr.simps* [*simp del*]
declare *equivS.simps* [*simp del*]
declare *extractBytecode-def* [*simp del*]
declare *initConsTable-def* [*simp del*]

declare *exec-instr.simps* [*simp*]

fun *Item2val* :: *entries-* ⇒ *sheap* ⇒ *Item* ⇒ *val*
where
Item2val S sh (ItemConst v) = (*if* (*isBool v = True*)
then Bool (*the-BoolT v*)
else Intg (*the-IntT v*))
| *Item2val S sh* (*ItemVar l*) = *the* (*S* (*nat* (*the-Intg* (*the* (*sh* (*stackC*, *topf*)))
− *int l*)))
| *Item2val S sh* (*ItemRegSelf*) = *the* (*sh* (*heapC*, *kf*))

axioms *maxPush-BUILDENV*:
fst (*the* (*map-of svms l*)) ! *i* = *BUILDENV items* ⟹ *n* + *int* (*length items*) < *int m*

axioms *pushAux-instrs*:
P ⊢
(*None*,*sha*,*h*,*inih*,
[([],
vs[*Suc 0* := *Intg* (*n* + *int* (*length items*))],
safeP, *sigSafeMain*, *the* (*cdm* (*l*, *i*)) + *1* + *1* + *1* + *1*, *ag*, *bd*)])
−*jvm*→
(*None*,
sha,
(*let vitems* = *map* (*Item2val S*′ *sha*) *items*;
idxs = *decreasing* (*nat* (*the-Intg* (*the* (*sha* (*stackC*, *topf*))) + *int* (*length items*)));
S′′ = *S*′ ++ *map-of* (*zip idxs vitems*)
in *h*(*the-Addr* (*the* (*sha* (*stackC*, *Sf*))) ↦ *Arr ty m S*′′)),
inih,
[([],
vs[*Suc 0* := *Intg* (*n* + *int* (*length items*))],
safeP, *sigSafeMain*, *the* (*cdm* (*l*, *i*)) + *1* + *1* + *1* + *1* +
length (*concat* (*map pushAux* (*zip items* [*0*..<*length items*]))), *ag*, *bd*)])

axioms *equivS-BUILDENV*:
equivS S S′ *n ctm d g*
⟹ *equivS* (*map* (*item2Stack k S*) *items* @ *S*)
(*S*′ ++ *map-of* (*zip* (*decreasing* (*nat* (*n* + *int* (*length items*)))) (*map* (*Item2val S*′ *sha*) *items*)))
(*n* + *int* (*length items*)) *ctm d g*

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

lemma *activeCells-BUILDENV*:
⟦*the-Addr* (*the* (*sha* (*stackC*, *Sf*))) ∉ *activeCells regS h* (*nat* (*the-Intg* (*the* (*sha* (*heapC*, *kf*)))))⟧
⟹ *activeCells regS h* (*nat* (*the-Intg* (*the* (*sha* (*heapC*, *kf*))))) =
activeCells regS

```
      (h(the-Addr (the (sha (stackC, Sf))) ↦
       Arr ty m
        (S' ++ map-of (zip (decreasing (nat (the-Intg (the (sha (stackC, topf)))
+ int (length items))))
                      (map (Item2val S' sha) items))))) (nat (the-Intg (the (sha
(heapC, kf)))))
apply (frule l-not-in-cellReg)
apply (unfold activeCells-def,auto)
 apply (unfold region-def, simp add: Let-def, elim conjE)
 apply (rule-tac x=j in exI,simp)
 apply (rule cellReg-step)
  apply (rule cellReg-basic)
 apply (erule-tac x=j in allE)
 apply (rule cellsReg-monotone-1,assumption+)
 apply (erule l-not-in-regs)
apply (simp add: Let-def, elim conjE)
apply (rule-tac x=j in exI, simp)
apply (rule cellReg-step)
 apply (rule cellReg-basic)
 apply (erule-tac x=j in allE)
apply (erule cellsReg-monotone-2,assumption+)
by (erule l-not-in-regs)
```

```
lemma domH-BUILDENV:
  ⟦ la ∉ activeCells regS h (nat (the-Intg (the (sha (heapC, kf)))));
     ∀ l∈dom H. ∃ l'. l' = the (g l) ∧ equivC (the (H l)) h l' (the (h l')) (nat
(the-Intg (the (sha (heapC, kf))))) com regS d g ⟧
  ⟹ ∀ l∈dom H.
       ∃ l'. l' = the (g l) ∧
            equivC (the (H l))
             (h(la ↦
              Arr ty m
               (S' ++ map-of (zip (decreasing (nat (the-Intg (the (sha (stackC,
topf))) + int (length items))))
                             (map (Item2val S' sha) items)))))
             l' (the ((h(la ↦
                      Arr ty m
                       (S' ++ map-of (zip (decreasing (nat (the-Intg (the (sha
(stackC, topf))) + int (length items))))
                                     (map (Item2val S' sha) items)))))
                      l'))
             (nat (the-Intg (the (sha (heapC, kf)))))
             com regS d g
apply (frule l-not-in-cellReg)
apply (rule ballI)
apply (erule-tac x=l in ballE)
```

prefer *2* **apply** *simp*
apply (*erule exE*)
apply (*rule-tac x=l' in exI*) **apply** (*elim conjE*)
apply (*rule conjI*, *assumption*)
apply *clarsimp*
apply (*rule conjI*) **apply** (*rule impI*)
apply *clarsimp* **apply** (*simp add*: *region-def*)
apply *clarsimp*
apply (*rule-tac x=Obj in exI*)
apply (*rule-tac x=flds in exI*) **apply** *simp*
apply (*simp add*: *region-def*) **apply** (*elim conjE*)
apply (*rule cellReg-step*)
apply (*rule cellReg-basic*)
apply (*erule-tac x=j in allE*)
apply (*rule cellsReg-monotone-1*,*assumption*+)
by (*erule l-not-in-regs*)

lemma *equivH-BUILDENV*:
⟦ *k'* = *nat* (*the-Intg* (*the* (*sha* (*heapC*, *kf*)))); ((*hm*, *k*), *k0*, (*l*, *i*), *S*) = ((*H*, *ka*), *k0a*, *PC*, *Sa*); *equivH* (*H*, *ka*) *h k'* *com regS d g*; *sha* (*stackC*, *Sf*) = *Some* (*Addr la*); *activeCells regS h k'* ∩ {*la*, *l'*, *l''*} = {}⟧
⟹ *equivH* (*hm*, *k*)
(*h*(*the-Addr* (*the* (*sha* (*stackC*, *Sf*))) ↦
Arr ty m
(*S'* ++ *map-of* (*zip* (*decreasing* (*nat* (*the-Intg* (*the* (*sha* (*stackC*, *topf*))) + *int* (*length items*))))
(*map* (*Item2val S' sha*) *items*)))))
(*nat* (*the-Intg* (*the* ((*sha*((*stackC*, *topf*) ↦ *vs*[*Suc 0* := *Intg* (*n* + *int* (*length items*))] ! *Suc 0*)) (*heapC*, *kf*))))) *com regS d g*
apply (*simp add*: *heapC-def add*: *stackC-def*)
apply (*simp add*: *kf-def add*: *k0f-def*)
apply (*fold heapC-def*, *fold kf-def*)
apply (*unfold equivH.simps*, *elim conjE*)
apply (*rule conjI*, *assumption*)
apply (*fold stackC-def*)
apply (*rule conjI*)
apply (*subgoal-tac*
the-Addr (*the* (*sha* (*stackC*, *Sf*))) ∉ *activeCells regS h* (*nat* (*the-Intg* (*the* (*sha* (*heapC*, *kf*))))))
apply (*frule activeCells-BUILDENV*,*simp*)
apply *simp*
apply (*rule conjI*, *assumption*)
by (*rule domH-BUILDENV*,*simp*,*assumption*)

lemma *execSVMInstr-BUILDENV*:

$[\![(P', cdm, ctm, com) = trSVM2JVM ((svms, ctmap), ini, ct, ah, ai, bc);$
$cdm , ctm, com \vdash ((hm, k), k0, (l, i), S) \triangleq S1';$
$(fst (the (map\text{-}of\ svms\ l)) ! i) = BUILDENV\ items;$
$execSVMInst (BUILDENV\ items) (map\text{-}of\ ct) (hm, k) k0 (l, i) S = Either.Right\ S2;$
$drop (the (cdm (l, i))) (extractBytecode\ P') =$
$trInstr (the (cdm (l, i))) cdm'\ ctm'\ com\ pcc (BUILDENV\ items) @ bytecode'$
$]\!] \Longrightarrow \exists v'\ sh'\ dh'\ ih'\ fms'.$
$P' \vdash S1' -jvm\rightarrow (v',sh',dh',ih', fms') \wedge$
$cdm , ctm, com \vdash S2 \triangleq (v',sh',dh',ih', fms')$

apply (*case-tac S1'*)
apply (*rename-tac v tup*)
apply (*case-tac tup*)
apply (*rename-tac sh tup*)
apply (*case-tac tup*)
apply (*rename-tac dh tup*)
apply (*case-tac tup*)
apply (*rename-tac ih fms*)
apply (*simp*)

apply (*unfold equivState-def*)
apply (*elim exE,elim conjE*)

apply (*rule-tac x=None* **in** *exI*)
apply (*rule-tac x=sha((stackC, topf) ↦ vs[Suc 0 := Intg (n + int (length items))]*
! Suc 0) **in** *exI*)
apply (*rule-tac x=(let vitems = map (Item2val S' sha) items;*
idxs = decreasing (nat (the-Intg (the (sha (stackC, topf))) +
int (length items)));
S'' = S' ++ map-of (zip idxs vitems)
in h(the-Addr (the (sha (stackC, Sf))) ↦ Arr ty m S'')) **in** *exI*)
apply (*rule-tac x=inih* **in** *exI*)
apply (*rule-tac x=[([],*
vs[Suc 0 := Intg (n + int (length items))],
safeP, sigSafeMain,
the (cdm (l, i)) + 1 + 1 + 1 + 1 + 1 + 1 +
length (concat (map pushAux (zip items [0..<length items]))),
ref)] **in** *exI*)

apply (*subgoal-tac ∃ vs pc . fms=([],vs,safeP,sigSafeMain,pc,ref)#[]*)
prefer *2* **apply** *simp*
apply (*erule exE*)+

apply (*rule conjI*)

apply (*unfold exec-all-def*)
apply (*subgoal-tac PC = (l,i)*) **prefer** *2* **apply** *simp*
apply *simp* **apply** (*elim conjE*) **apply** *clarsimp*
apply (*subgoal-tac P'⊢ (None,sha,h,inih,[([], vs, safeP, sigSafeMain, the (cdm (l,i)), ag,bd)])*
−jvm→
(None,sha,h,inih,[([the (sha (stackC,topf))], vs, safeP, sigSafeMain, the (cdm (l,i)) + 1, ag,bd)]))
apply (*unfold exec-all-def*)
apply (*erule rtrancl-trans*)

prefer *2*
apply (*rule r-into-rtrancl*)
apply (*clarify*)
apply (*unfold JVMExec.exec.simps*)
apply (*unfold Let-def*)
apply (*unfold extractBytecode-def*)
apply (*unfold trInstr.simps*)
apply (*unfold Let-def*)
apply (*subgoal-tac (fst (snd (snd (snd (snd (the (method' (P', safeP) sigSafeMain))))))) !*
the (cdm (l, i))) = Getstatic topf stackC,simp)
apply (*erule nth-via-drop-append-2*)

apply (*drule drop-Suc-append-2*)
apply (*subgoal-tac P'⊢ (None, sha, h, inih,*
[([the (sha (stackC,topf))],
vs, safeP, sigSafeMain, the (cdm (l, i)) + 1, ag, bd)])
−jvm→
(None,sha,h,inih,
[([Intg (int (length items)),
the (sha (stackC,topf))],
vs, safeP, sigSafeMain, the (cdm (l, i)) + 1 + 1, ag, bd)]))
apply (*unfold exec-all-def*)
apply (*erule rtrancl-trans*)
prefer *2*
apply (*rule r-into-rtrancl*)
apply (*clarify*)
apply (*unfold JVMExec.exec.simps*)
apply (*unfold Let-def*)
apply (*subgoal-tac (fst (snd (snd (snd (snd (the (method' (P', safeP) sigSafeMain))))))) ! Suc (the (cdm (l, i)))) =*
LitPush (Intg (int (length items))),simp)
apply (*erule nth-via-drop-append*)

apply (*drule drop-Suc-append*)
apply (*subgoal-tac P'⊢* (*None*,*sha*,*h*,*inih*,
[([*Intg* (*int* (*length items*)),
the (*sha* (*stackC*,*topf*))],
vs, *safeP*, *sigSafeMain*, *the* (*cdm* (*l*, *i*)) + *1* + *1*, *ag*, *bd*)])
−*jvm*→
(*None*,*sha*,*h*,*inih*,
[([*Intg* (*n* + *int* (*length items*))],
vs, *safeP*, *sigSafeMain*, *the* (*cdm* (*l*, *i*)) + *1* + *1* + *1*, *ag*, *bd*)]))
apply (*unfold exec-all-def*)
apply (*erule rtrancl-trans*)
prefer *2*
apply (*rule r-into-rtrancl*)
apply (*clarify*)
apply (*unfold JVMExec.exec.simps*)
apply (*unfold Let-def*)
apply (*subgoal-tac* (*fst* (*snd* (*snd* (*snd* (*snd* (*the* (*method' P'*, *safeP*) *sigSafeMain*)))))) ! (*Suc* (*Suc* (*the* (*cdm* (*l*, *i*)))))) =
BinOp Add,*simp*)
apply (*erule nth-via-drop-append*)

apply (*drule drop-Suc-append*)
apply (*subgoal-tac P'⊢* (*None*,*sha*,*h*,*inih*,
[([*Intg* (*n* + *int* (*length items*))],
vs, *safeP*, *sigSafeMain*, *the* (*cdm* (*l*, *i*)) + *1* + *1* + *1*, *ag*, *bd*)])
−*jvm*→
(*None*,*sha*,*h*,*inih*,
[([],
vs[*Suc 0* := *Intg* (*n* + *int* (*length items*))],
safeP, *sigSafeMain*, *the* (*cdm* (*l*, *i*)) + *1* + *1* + *1* + *1*, *ag*, *bd*)]))
apply (*unfold exec-all-def*)
apply (*erule rtrancl-trans*)
prefer *2*
apply (*rule r-into-rtrancl*)
apply (*clarify*)
apply (*unfold JVMExec.exec.simps*)
apply (*unfold Let-def*)
apply (*subgoal-tac* (*fst* (*snd* (*snd* (*snd* (*snd* (*the* (*method' P'*, *safeP*) *sigSafeMain*)))))) ! (*Suc* (*Suc* (*Suc* (*the* (*cdm* (*l*, *i*))))))) =
Store 1,*simp*)
apply (*erule nth-via-drop-append*)

apply (*drule drop-Suc-append*)
apply (*subgoal-tac P'⊢* (*None*,*sha*,*h*,*inih*,

```
                    [([],
                     vs[Suc 0 := Intg (n + int (length items))],
                      safeP, sigSafeMain, the (cdm (l, i)) + 1 + 1 + 1 + 1, ag,
bd)])
                 −jvm→
                    (None, sha,
                     (let vitems = map (Item2val S' sha) items;
                            idxs   = decreasing (nat (the-Intg (the (sha (stackC,
topf))) + int (length items)));
                          S''    = S' ++ map-of (zip idxs vitems)
                      in  h(the-Addr (the (sha (stackC, Sf))) ↦ Arr ty m S'')),
                     inih,
                    [([],
                     vs[Suc 0 := Intg (n + int (length items))],
                     safeP, sigSafeMain, the (cdm (l, i)) + 1 + 1 + 1 + 1 +
                              length (concat (map pushAux (zip items [0..<length
items])))), ag, bd)]))
prefer 2 apply (rule pushAux-instrs)
apply (unfold exec-all-def)
apply (erule rtrancl-trans)


apply (drule drop-append-length)
apply (subgoal-tac P'⊢ (None,sha,
                      (let vitems = map (Item2val S' sha) items;
                            idxs   = decreasing (nat (the-Intg (the (sha (stackC,
topf))) + int (length items)));
                          S''    = S' ++ map-of (zip idxs vitems)
                      in  h(the-Addr (the (sha (stackC, Sf))) ↦ Arr ty m S'')),
                     inih,
                    [([],
                     vs[Suc 0 := Intg (n + int (length items))],
                     safeP, sigSafeMain, the (cdm (l, i)) + 1 + 1 + 1 + 1 +
                              length (concat (map pushAux (zip items [0..<length
items])))), ag, bd)])
                 −jvm→
                    (None,sha,
                     (let vitems = map (Item2val S' sha) items;
                            idxs   = decreasing (nat (the-Intg (the (sha (stackC,
topf))) + int (length items)));
                          S''    = S' ++ map-of (zip idxs vitems)
                      in  h(the-Addr (the (sha (stackC, Sf))) ↦ Arr ty m S'')),
                     inih,
                    [([vs[Suc 0 := Intg (n + int (length items))] ! Suc 0],
                     vs[Suc 0 := Intg (n + int (length items))],
                    safeP, sigSafeMain, the (cdm (l, i)) + 1 + 1 + 1 + 1 + 1 +
                              length (concat (map pushAux (zip items [0..<length
items])))), ag, bd)]))
apply (unfold exec-all-def)
```

apply (*erule rtrancl-trans*)
prefer *2*
apply (*rule r-into-rtrancl*)
apply (*clarify*)
apply (*unfold JVMExec.exec.simps*)
apply (*unfold Let-def*)
apply (*subgoal-tac* (*fst* (*snd* (*snd* (*snd* (*snd* (*the* (*method'* (*P'*, *safeP*) *sigSafeMain*)))))) !
(((*Suc* (*Suc* (*Suc* (*Suc* (*the* (*cdm* (*l*, *i*))))))) +
length (*concat* (*map pushAux* (*zip items* [*0*..<*length items*])))))) =
Load 1,*simp*)
apply (*erule nth-via-drop-append*)

apply (*drule drop-Suc-append*)
apply (*subgoal-tac P'*⊢ (*None*,*sha*,
(*let vitems* = *map* (*Item2val S' sha*) *items*;
idxs = *decreasing* (*nat* (*the-Intg* (*the* (*sha* (*stackC*, *topf*))) + *int* (*length items*)));
S'' = *S'* ++ *map-of* (*zip idxs vitems*)
in *h*(*the-Addr* (*the* (*sha* (*stackC*, *Sf*))) ↦ *Arr ty m S''*)),
inih,
[([*vs*[*Suc 0* := *Intg* (*n* + *int* (*length items*))] ! *Suc 0*],
vs[*Suc 0* := *Intg* (*n* + *int* (*length items*))],
safeP, *sigSafeMain*, *the* (*cdm* (*l*, *i*)) + *1* + *1* + *1* + *1* + *1* +
length (*concat* (*map pushAux* (*zip items* [*0*..<*length items*]))), *ag*, *bd*)])
−*jvm*→
(*None*,
sha((*stackC*, *topf*) ↦ *vs*[*Suc 0* := *Intg* (*n* + *int* (*length items*))] ! *Suc 0*),
(*let vitems* = *map* (*Item2val S' sha*) *items*;
idxs = *decreasing* (*nat* (*the-Intg* (*the* (*sha* (*stackC*, *topf*))) + *int* (*length items*)));
S'' = *S'* ++ *map-of* (*zip idxs vitems*)
in *h*(*the-Addr* (*the* (*sha* (*stackC*, *Sf*))) ↦ *Arr ty m S''*)),
inih,
[([],
vs[*Suc 0* := *Intg* (*n* + *int* (*length items*))],
safeP, *sigSafeMain*, *the* (*cdm* (*l*, *i*)) + *1* + *1* + *1* + *1* + *1* + *1* +
length (*concat* (*map pushAux* (*zip items* [*0*..<*length items*]))), *ag*, *bd*)]))
apply (*unfold exec-all-def*)
prefer *2*
apply (*rule r-into-rtrancl*)
apply (*clarify*)
apply (*unfold JVMExec.exec.simps*)

apply (*unfold Let-def*)
apply (*subgoal-tac* (*fst* (*snd* (*snd* (*snd* (*snd* (*the* (*method'* (*P'*, *safeP*) *sigSafeMain*))))))) !
(((*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*the* (*cdm* (*l*, *i*)))))))) +
length (*concat* (*map pushAux* (*zip items* [*0*..<*length items*])))) =
Putstatic topf stackC,*simp*)
apply (*rule nth-via-drop-append*, *force*)
apply *simp*

apply (*frule nonJumping-Suc-pc*)
apply (*erule-tac sym* [**where** *t*=*BUILDENV items*])
apply (*simp add*: *nonJumping.simps*)
apply (*simp add*: *trInstr.simps*)

apply (*rule-tac x*=*hm* **in** *exI*)
apply (*rule-tac x*=*k* **in** *exI*)
apply (*rule-tac x*=*k0* **in** *exI*)
apply (*rule-tac x*=(*l*, *Suc i*) **in** *exI*)
apply (*rule-tac x*=*map* (*item2Stack k S*) *items* @ *S* **in** *exI*)

apply (*rule-tac x*=*sha*((*stackC*, *topf*) ↦ *vs*[*Suc 0* := *Intg* (*n* + *int* (*length items*))] ! *Suc 0*) **in** *exI*)
apply (*rule-tac x*=*h*(*the-Addr* (*the* (*sha* (*stackC*, *Sf*))) ↦ *Arr ty m*
(*S'* ++ *map-of* (*zip* (*decreasing* (*nat* (*the-Intg* (*the* (*sha* (*stackC*, *topf*))) + *int* (*length items*))))
(*map* (*Item2val S' sha*) *items*)))) **in** *exI*)
apply (*rule-tac x*=*inih* **in** *exI*)
apply (*rule-tac x*=*vs*[*Suc 0* := *Intg* (*n* + *int* (*length items*))] **in** *exI*)
apply (*rule-tac x*=*the* (*cdm* (*l*, *i*)) + *1* + *1* + *1* + *1* + *1* + *1* + *length* (*concat* (*map pushAux* (*zip items* [*0*..<*length items*]))) **in** *exI*)
apply (*rule-tac x*=*ref* **in** *exI*)

apply (*rule-tac x*=*nat* (*the-Intg* (*the* ((*sha*((*stackC*, *topf*) ↦
vs[*Suc 0* := *Intg* (*n* + *int* (*length items*))] ! *Suc 0*)) (*heapC*, *kf*)))) **in** *exI*)

apply (*rule-tac x*= *nat* (*the-Intg* (*the* ((*sha*((*stackC*, *topf*) ↦
vs[*Suc 0* := *Intg* (*n* + *int* (*length items*))] ! *Suc 0*)) (*heapC*, *k0f*)))) **in** *exI*)

apply (*rule-tac x*=*la* **in** *exI*)
apply (*rule-tac x*=*ty* **in** *exI*)
apply (*rule-tac x*=*m* **in** *exI*)
apply (*rule-tac x*=*S'* ++ *map-of* (*zip* (*decreasing* (*nat* (*n* + *int* (*length items*)))) (*map* (*Item2val S' sha*) *items*)) **in** *exI*)

apply (*rule-tac x*=(*n* + *int* (*length items*)) **in** *exI*)
apply (*rule-tac x*=*l'* **in** *exI*)

apply (*rule-tac x=regS* **in** *exI*)
apply (*rule-tac x=l''* **in** *exI*)
apply (*rule-tac x=m'* **in** *exI*)
apply (*rule-tac x=m''* **in** *exI*)
apply (*rule-tac x=d* **in** *exI*)
apply (*rule-tac x=g* **in** *exI*)

apply (*rule conjI*, *erule sym*)
apply (*rule conjI*, *rule refl*)
apply (*rule conjI*, *rule refl*)
apply (*rule conjI*, *rule refl*)
apply (*rule conjI*) **apply** *clarsimp*
apply (*rule conjI*, *assumption*)
apply (*rule conjI*) **apply** *clarsimp*
apply (*simp add*: *heapC-def add*: *stackC-def*)
apply (*rule conjI*) **apply** (*erule activeCells*)
apply (*rule conjI*) **apply** (*erule activeCells-2*,*assumption*)
apply (*erule activeCells-2*,*assumption*)
apply (*rule conjI*) **apply** *clarsimp*
apply (*rule conjI*) **apply** *clarsimp*
apply (*subgoal-tac length vs = 10*)
apply *clarsimp*
apply (*rule length-vs*)
apply (*rule conjI*) **apply** *clarsimp*
apply (*rule conjI*) **apply** *clarsimp*
apply (*rule conjI*) **apply** *clarsimp*
apply (*rule conjI*) **apply** *clarsimp*
apply (*rule conjI*) **apply** *clarsimp*
apply (*rule conjI*) **apply** *clarsimp*
apply (*rule conjI*) **apply** (*rule equivH-BUILDENV*,*assumption+*)
apply (*rule conjI*) **apply** *clarsimp*
apply (*rule maxPush-BUILDENV*,*assumption+*)
apply (*rule conjI*) **apply** *clarsimp*
apply (*erule equivS-BUILDENV*)
apply (*rule conjI*) **apply** (*subgoal-tac length vs = 10*)
apply (*simp add*: *heapC-def add*: *stackC-def*)
apply (*rule length-vs*)
by *simp*

end

theory *dem-CALL*
imports *../JVMSAFE/JVMExec ../JVMSAFE/State SVM2JVM SVMSemantics CertifSVM2JVM*
dem-translation
begin

no-translations *Norm s == (None,s)*
no-translations *ex-table-of m == snd (snd (snd m))*

declare *trInstr.simps* [*simp del*]
declare *equivS.simps* [*simp del*]
declare *extractBytecode-def* [*simp del*]
declare *initConsTable-def* [*simp del*]

axioms *pushregion-method*:
⟦ *s = (None, shp, hp, ihp,*
([],vs,safeP, sigSafeMain,pc,z1,z2)#[]);
shp (stackC,topf) = Some (Intg k);
shp (stackC,Sf) = Some (Addr l);
shp (heapC,regionsf) = Some (Addr l′); hp l′ = Some (Arr ty m regS);
shp (dirCellC,safeDirf) = Some (Addr l″);
Invoke-static heapC pushRegion [] = fst(snd(snd(snd(snd(the(method′ (P′,safeP) sigSafeMain)))))) ! pc
⟧ ⟹
P′ |− s −jvm→ (None, shp ((heapC,kf)↦ Intg (k+1)),
hp(l′ ↦ Arr ty ma (regS (nat k + 1 ↦ Addr lnew)))
(lnew ↦ Obj cellC (empty ((izqf,cellC)↦ Addr lnew)
((derf,cellC)↦ Addr lnew))),
ihp, ([],loc,safeP,sigSafeMain,pc+1,z1,z2)#[]) ∧
lnew ∉ {l,l′,l″} ∧
lnew ∉ activeCells regS h (nat (the-Intg (the (sha (heapC, kf)))))

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

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

lemma *cellsReg-monotone-3* [*rule-format*]:
x ∈ cellsReg (h(l′ ↦ A, l″ ↦ B)) l
⟶ x ≠ l
⟶ l ≠ l′
⟶ l ≠ l″
⟶ x = nextCell h l
apply (*rule impI*)
apply (*erule cellsReg.induct*,*simp*)
by (*simp add*: *nextCell-def*)

lemma *cellsReg-monotone-4* [*rule-format*]:
x ∈ cellsReg h l

$\longrightarrow x \neq l$
$\longrightarrow l \neq l'$
$\longrightarrow l \neq l''$
$\longrightarrow x = nextCell\ (h(l' \mapsto A,\ l'' \mapsto B))\ l$
apply (*rule impI*)
apply (*erule cellsReg.induct*,*simp*)
by (*simp add*: *nextCell-def*)

lemma *activeCells-CALL*:
$[\![\ l' \notin activeCells\ regS\ dh\ (nat\ (the\text{-}Intg\ (the\ (sha\ (heapC,\ kf)))))$;
$lnew \notin activeCells\ regS\ dh\ (nat\ (the\text{-}Intg\ (the\ (sha\ (heapC,\ kf)))))$
$]\!]$
$\Longrightarrow activeCells\ regS\ dh\ (nat\ (the\text{-}Intg\ (the\ (sha\ (heapC,\ kf))))) =$
$activeCells\ (regS(Suc\ (nat\ (the\text{-}Intg\ (the\ (sha\ (heapC,\ kf)))))) \mapsto Addr\ lnew))$
$(dh(l' \mapsto Arr\ ty\ m'\ (regS(Suc\ (nat\ (the\text{-}Intg\ (the\ (sha\ (heapC,\ kf))))))) \mapsto$
$Addr\ lnew)),\ lnew \mapsto$
$Obj\ cellC\ [(izqf,\ cellC) \mapsto Addr\ lnew,\ (derf,\ cellC) \mapsto Addr\ lnew]))$
$(nat\ (the\text{-}Intg\ (the\ (sha\ (heapC,\ kf)))) + 1))$
apply (*frule l-not-in-cellReg*)
apply (*frule-tac l=lnew* **in** *l-not-in-cellReg*)
apply (*rule equalityI*)
apply (*rule subsetI*)
apply (*simp add*: *activeCells-def*,*clarsimp*)
apply (*unfold region-def*, *simp add*: *Let-def*, *elim conjE*)
apply (*rule-tac x=j* **in** *exI*,*simp*)
apply (*rule conjI*, *simp*)
apply (*rule cellReg-step*)
apply (*rule cellReg-basic*)
apply (*erule-tac x=j* **in** *allE*)
apply (*rule cellsReg-monotone-1*)
apply (*rule cellReg-step*)
apply (*rule cellReg-basic*)
apply (*rule cellsReg-monotone-1*,*assumption*+)
apply (*rule l-not-in-regs*)
apply (*erule-tac x=j* **in** *allE*,*simp*,*simp*)
apply (*rule l-not-in-regs*)
apply (*erule-tac x=j* **in** *allE*)+
apply *simp*
apply (*rule subsetI*)
apply (*unfold activeCells-def*)
apply (*unfold region-def*, *simp add*: *Let-def*)
apply (*erule exE*)
apply (*case-tac*
$j < Suc\ (nat\ (the\text{-}Intg\ (the\ (sha\ (heapC,\ kf)))))$,*simp*)
apply (*rule-tac x=j* **in** *exI*,*simp*)
apply (*elim conjE*)
apply (*rule cellReg-step*)

apply (*rule cellReg-basic*)
apply (*rule cellsReg-monotone-3*,*simp*,*simp*)
apply (*rule l-not-in-regs*)
apply (*erule-tac x=j* **in** *allE*)+ **apply** *simp*
apply (*rule l-not-in-regs*)
apply (*erule-tac x=j* **in** *allE*)+ **apply** *simp*
apply (*elim conjE*)
apply (*subgoal-tac*
j = *nat* (*the-Intg* (*the* (*sha* (*heapC*, *kf*))) + *1*),*simp*)
prefer *2* **apply** *simp*
apply (*subgoal-tac nat* (*the-Intg* (*the* (*sha* (*heapC*, *kf*))) + *1*) = *Suc* (*nat* (*the-Intg* (*the* (*sha* (*heapC*, *kf*)))))))
prefer *2* **apply** *arith*
apply *simp*
apply (*subgoal-tac x=lnew*,*simp*)
apply (*erule cellsReg.induct*)
apply *simp*
by (*simp add*: *nextCell-def*)

lemma *domH-CALL*:
⟦ *l′* ∉ *activeCells regS h* (*nat* (*the-Intg* (*the* (*sha* (*heapC*, *kf*))))));
lnew ∉ *activeCells regS h* (*nat* (*the-Intg* (*the* (*sha* (*heapC*, *kf*))))));
lnew ∉ {*la*,*l′*,*l′′*};
∀ *l*∈*dom H*. ∃ *l′*. *l′* = *the* (*g l*) ∧ *equivC* (*the* (*H l*)) *h l′* (*the* (*h l′*)) (*nat* (*the-Intg* (*the* (*sha* (*heapC*, *kf*)))))) *com regS d g* ⟧
⟹ ∀ *l*∈*dom H*.
∃ *l′a*. *l′a* = *the* (*g l*) ∧
equivC (*the* (*H l*))
(*h*(*l′* ↦ *Arr ty m′* (*regS*(*Suc* (*nat* (*the-Intg* (*the* (*sha* (*heapC*, *kf*))))))
↦ *Addr lnew*)), *lnew* ↦
Obj cellC [(*izqf*, *cellC*) ↦ *Addr lnew*, (*derf*, *cellC*) ↦ *Addr lnew*]))
l′a (*the* ((*h*(*l′* ↦ *Arr ty m′* (*regS*(*Suc* (*nat* (*the-Intg* (*the* (*sha* (*heapC*, *kf*))))))
↦ *Addr lnew*)), *lnew* ↦
Obj cellC [(*izqf*, *cellC*) ↦ *Addr lnew*, (*derf*, *cellC*) ↦ *Addr lnew*]))
l′a))
(*nat* (*the-Intg* (*the* (*sha* (*heapC*, *kf*))) + *1*))
com (*regS*(*Suc* (*nat* (*the-Intg* (*the* (*sha* (*heapC*, *kf*))))) ↦ *Addr lnew*)) *d g*
apply (*frule l-not-in-cellReg*)
apply (*frule-tac l=lnew* **in** *l-not-in-cellReg*)
apply (*rule ballI*)
apply (*erule-tac x=l* **in** *ballE*)
prefer *2* **apply** *simp*

apply (*erule exE*)
apply (*rule-tac x=l'a* **in** *exI*) **apply** (*elim conjE*)
apply (*rule conjI*, *assumption*)
apply *clarsimp*
apply (*rule conjI*, *rule impI*)
apply (*simp add*: *activeCells-def*)
apply (*simp add*: *activeCells-def*)
apply (*erule-tac x=j* **in** *allE*) **apply** *simp*
apply (*subgoal-tac the* (*g l*) ≠ *lnew*,*simp*)
prefer *2* **apply** *blast*
apply (*rule impI*)
apply (*rule conjI*, *simp*)
apply (*rule-tac x=Objа* **in** *exI*)
apply (*rule-tac x=flds* **in** *exI*) **apply** *simp*
apply (*simp add*: *region-def*)
apply *clarsimp*
apply (*rule cellReg-step*)
apply (*rule cellReg-basic*)
apply (*rule cellsReg-monotone-4*) **apply** *assumption*+
apply (*rule l-not-in-regs*)
apply (*erule-tac x=j* **in** *allE*) **apply** *simp*
apply (*rule l-not-in-regs*)
apply (*erule-tac x=j* **in** *allE*)
apply (*erule-tac x=j* **in** *allE*)
apply (*erule-tac x=j* **in** *allE*)
by *simp*

axioms *kf-ge-0*:
the-Intg (*the* (*sha* (*heapC*, *kf*))) >= *0*

lemma *equivH-CALL*:
⟦((*hm*, *k*), *k0*, (*l*, *i*), *S*) = ((*H*, *ka*), *k0a*, *PC*, *Sa*);
(*v*, *sh*, *dh*, *ih*, *fms*) = (*None*, *sha*, *h*, *inih*, [([], *vs*, *safeP*, *sigSafeMain*, *pc*, *ref*)]);
k' = *nat* (*the-Intg* (*the* (*sha* (*heapC*, *kf*))));
k0' = *nat* (*the-Intg* (*the* (*sha* (*heapC*, *k0f*))));
sha (*stackC*, *Sf*) = *Some* (*Addr la*);
lnew ∉ *activeCells regS h* (*nat* (*the-Intg* (*the* (*sha* (*heapC*, *kf*))))));
distinct [*la*, *l'*, *l''*]; *activeCells regS h k'* ∩ {*la*, *l'*, *l''*} = {}; *h la* = *Some* (*Arr ty m S'*);
sha (*stackC*, *topf*) = *Some* (*Intg n*); *sha* (*heapC*, *regionsf*) = *Some* (*Addr l'*); *h l'* = *Some* (*Arr ty m' regS*);
lnew ∉ {*la*,*l'*,*l''*};
inj-on g (*dom H*); *equivH* (*H*, *ka*) *h k'* *com regS d g*⟧
⟹ *equivH* (*hm*, *Suc k*)
(*dh*(*l'* ↦ *Arr ty m'* (*regS*(*k'* + *1* ↦ *Addr lnew*)), *lnew* ↦ *Obj cellC* [(*izqf*, *cellC*) ↦ *Addr lnew*, (*derf*, *cellC*) ↦ *Addr lnew*]))

(*nat* (*the-Intg* (*the* ((*sh*((*heapC*, *kf*) ↦ *Intg* (*the-Intg* (*the* (*sh* (*heapC*, *kf*))) + *1*))) (*heapC*, *kf*))))) *com*
(*regS*(*Suc* (*nat* (*the-Intg* (*the* (*sha* (*heapC*, *kf*)))))) ↦ *Addr lnew*)) *d g*
apply (*simp add*: *heapC-def add*: *stackC-def*)
apply (*fold heapC-def*)
apply (*unfold equivH.simps*, *elim conjE*)
apply (*rule conjI*, *clarsimp*)
apply (*subgoal-tac the-Intg* (*the* (*sha* (*heapC*, *kf*))) >= *0*,*simp*)
apply (*rule kf-ge-0*)
apply (*rule conjI*, *frule-tac l'=l'* **in** *activeCells-CALL*,*assumption+*,*simp*)
apply (*rule conjI*, *assumption*)
apply (*subgoal-tac lnew* ∉ {*la*,*l'*,*l''*})
apply (*rule domH-CALL*,*assumption+*)
by *clarsimp*

axioms *activeCells-3*:
⟦ *l'* ∉ *activeCells regS h k*; *l* ≠ *l'*; *l* ≠ *l''* ⟧
⟹ *l'* ∉ *activeCells regS* (*h*(*l* ↦ *A*, *l''*↦ *B*)) *k*

axioms *activeCells-3-1*:
⟦ *l'* ∉ *activeCells regS h k* ⟧
⟹ *l'* ∉ *activeCells regS* (*h*(*l'* ↦ *A*, *l''*↦ *B*)) *k*

axioms *activeCells-4*:
la ∉ *activeCells regS h k*
⟹ *la* ∉ *activeCells regS' h k'*

axioms *kf-Intg*:
sha (*heapC*, *kf*) = *Some* (*Intg j*)

axioms *p-in-dom-cdm'*:
(*p*, *0*) ∈ *dom cdm'*

axioms *lnew-notin-activeCells*:
lnew ∉ *activeCells regS h* (*nat* (*the-Intg* (*the* (*sha* (*heapC*, *kf*)))))

lemma *execSVMInstr-CALL* :
⟦ (*P'*, *cdm*, *ctm*, *com*) = *trSVM2JVM* ((*svms*, *ctmap*), *ini*, *ct*, *ah*, *ai*, *bc*);
cdm , *ctm*, *com* ⊢ ((*hm*, *k*), *k0*, (*l*, *i*), *S*) ≙ *S1'*;
(*fst* (*the* (*map-of svms l*)) ! *i*) = *CALL p*;
execSVMInst (*CALL p*) (*map-of ct*) (*hm*, *k*) *k0* (*l*, *i*) *S* = *Either.Right S2*;
drop (*the* (*cdm* (*l*, *i*))) (*extractBytecode P'*) =
trInstr (*the* (*cdm* (*l*, *i*))) *cdm' ctm' com pcc* (*CALL p*) @ *bytecode'*;
cdm' ⊆*m cdm*

$]\!] \Longrightarrow \ \exists\ v'\ sh'\ dh'\ ih'\ fms'\ .$
$P' \vdash\ S1'\ -jvm\rightarrow\ (v',sh',dh',ih',\ fms')\ \wedge$
$cdm\ ,\ ctm,\ com\ \vdash\ S2\ \triangleq\ (v',sh',dh',ih',\ fms')$

apply (*case-tac S1'*)
apply (*rename-tac v tup*)
apply (*case-tac tup*)
apply (*rename-tac sh tup*)
apply (*case-tac tup*)
apply (*rename-tac dh tup*)
apply (*case-tac tup*)
apply (*rename-tac ih fms*)
apply (*simp*)

apply (*unfold equivState-def*)
apply (*elim exE,elim conjE*)

apply (*rule-tac x=None* **in** *exI*)
apply (*rule-tac x=sh((heapC, kf) ↦ Intg (the-Intg (the (sh (heapC, kf)))+1))* **in** *exI*)
apply (*rule-tac x=dh(l' ↦ Arr ty m' (regS (k'+1 ↦ Addr lnew)))*
(*lnew ↦ Obj cellC (empty ((izqf,cellC)↦ Addr lnew)*
((derf,cellC)↦ Addr lnew)) **in** *exI*)
apply (*rule-tac x=ih* **in** *exI*)
apply (*rule-tac x=[([], vs, safeP, sigSafeMain, the (cdm (p, 0)), ref)]* **in** *exI*)
apply (*rule conjI*)
apply (*unfold extractBytecode-def*)

apply (*subgoal-tac P'⊢ (v,sh,dh,ih,fms) -jvm→*
(*None, sh ((heapC,kf)↦ Intg (the-Intg (the (sh (heapC, kf)))+1)),*
dh (l' ↦ Arr ty m' (regS (nat (the-Intg (the (sh (heapC, kf))))+1 ↦ Addr lnew)))
(lnew ↦ Obj cellC (empty ((izqf,cellC)↦ Addr lnew)
((derf,cellC)↦ Addr lnew))),
ih, ([],vs,safeP,sigSafeMain,pc+1,ref)#[]))
apply (*unfold exec-all-def*)
apply (*erule rtrancl-trans*)

prefer *2*
apply (*subgoal-tac (fst (snd (snd (snd (snd (the (method' (P', safeP) sigSafeMain))))))) !*
the (cdm (l, i))) = Invoke-static heapC pushRegion [])

apply (*fold exec-all-def*)
apply *clarify*
apply (*subgoal-tac*
P' |−
(*None*, *sha*, *h*, *inih*, [([], *vs*, *safeP*, *sigSafeMain*, *the* (*cdm* (*l*, *i*)), *ag*,*bd*)])
−*jvm*→
(*None*, *sha* ((*heapC*,*kf*)↦ *Intg* (*n*+*1*)),
h(*l'* ↦ *Arr ty m'* (*regS* (*nat n* + *1* ↦ *Addr lnew*)))
(*lnew* ↦ *Obj cellC* (*empty* ((*izqf*,*cellC*)↦ *Addr lnew*)
((*derf*,*cellC*)↦ *Addr lnew*))),
inih, [([],*vs*,*safeP*,*sigSafeMain*,*the* (*cdm* (*l*,*i*))+*1*,*ag*,*bd*)]) ∧
lnew ∉ {*la*,*l'*,*l''*} ∧
lnew ∉ *activeCells regS h* (*nat* (*the-Intg* (*the* (*sha* (*heapC*, *kf*))))))
prefer *2* **apply** (*rule pushregion-method*,*simp*,*assumption*+,*simp*)
apply *simp* **apply** (*subgoal-tac sha* (*heapC*, *kf*)=*Some* (*Intg n*))
prefer *2* **apply** (*rule kf-Intg*)
apply *simp*

apply (*unfold trInstr.simps*)
apply (*unfold Let-def*)
apply (*erule nth-via-drop-append*)

apply (*drule drop-Suc-append*)
apply (*unfold exec-all-def*)
apply (*rule r-into-rtrancl*)
apply (*clarify*)
apply (*unfold JVMExec.exec.simps*)
apply (*unfold Let-def*)
apply (*subgoal-tac* (*fst* (*snd* (*snd* (*snd* (*snd* (*the* (*method'* (*P'*, *safeP*) *sigSafeMain*)))))))) !
Suc (*the* (*cdm* (*l*, *i*)))) = *Goto* (*trAddr* (*the*(*cdm*(*p*,*0*))) (*the* (*cdm* (*l*, *i*)) + *incCall*))) **apply** (*simp*)
apply (*simp add*: *trAddr-def incCall-def*)
apply (*rule-tac ys*=[] **in** *nth-via-drop-append*)
apply *simp*
apply (*rule conjI*)
apply (*subgoal-tac cdm'* (*p*, *0*)=*cdm* (*p*, *0*),*simp*)
apply (*subgoal-tac* (*p*,*0*)∈ *dom cdm'*)
apply (*simp add*: *map-le-def*)
apply (*rule p-in-dom-cdm'*)
apply *simp*

apply (*rule-tac x*=*hm* **in** *exI*)
apply (*rule-tac x*=*Suc k* **in** *exI*)
apply (*rule-tac x*=*k0* **in** *exI*)

apply (*rule-tac x=(p,0)* **in** *exI*)
apply (*rule-tac x=S* **in** *exI*)

apply (*rule-tac x=sh((heapC, kf) ↦ Intg (the-Intg (the (sh (heapC, kf))) + 1))* **in** *exI*)
apply (*rule-tac x=dh(l′ ↦ Arr ty m′ (regS(k′ + 1 ↦ Addr lnew)),*
lnew ↦ Obj cellC [(izqf, cellC) ↦ Addr lnew,
(derf, cellC) ↦ Addr lnew]) **in** *exI*)
apply (*rule-tac x=ih* **in** *exI*)
apply (*rule-tac x=vs* **in** *exI*)
apply (*rule-tac x=the (cdm (p, 0))* **in** *exI*)
apply (*rule-tac x=ref* **in** *exI*)

apply (*rule-tac x=nat (the-Intg (the ((sh((heapC, kf) ↦*
Intg (the-Intg (the (sh (heapC, kf))) + 1))) (heapC, kf)))) **in** *exI*)
apply (*rule-tac x=nat (the-Intg (the ((sh((heapC, kf) ↦*
Intg (the-Intg (the (sh (heapC, kf))) + 1))) (heapC, k0f)))) **in** *exI*)

apply (*rule-tac x=la* **in** *exI*)
apply (*rule-tac x=ty* **in** *exI*)
apply (*rule-tac x=m* **in** *exI*)
apply (*rule-tac x=S′* **in** *exI*)

apply (*rule-tac x=n* **in** *exI*)
apply (*rule-tac x=l′* **in** *exI*)
apply (*rule-tac x= regS(Suc (nat (the-Intg (the (sha (heapC, kf)))))) ↦ Addr lnew*) **in** *exI*)
apply (*rule-tac x=l″* **in** *exI*)
apply (*rule-tac x=m′* **in** *exI*)
apply (*rule-tac x=m″* **in** *exI*)
apply (*rule-tac x=d* **in** *exI*)
apply (*rule-tac x=g* **in** *exI*)

apply (*subgoal-tac*
la ≠ lnew ∧ l′ ≠ lnew ∧ l″ ≠ lnew)
prefer *2*
apply (*case-tac ref,simp*)
apply (*frule-tac lnew=lnew*
in *pushregion-method, assumption+*)
apply (*clarsimp, rule sym*)
apply (*erule nth-via-drop*)
apply *clarsimp*

apply (*rule conjI, clarsimp*)
apply (*rule conjI, rule refl*)
apply (*rule conjI, rule refl*)
apply (*rule conjI, rule refl*)

apply (*rule conjI*) **apply** (*simp add*: *heapC-def add*: *stackC-def*)
apply (*rule conjI*, *assumption*)
apply (*rule conjI*) **apply** *clarsimp*
apply (*simp add*: *heapC-def add*: *stackC-def*)
apply (*rule conjI*) **apply** (*rule activeCells-3*)
apply (*rule activeCells-4*)
apply *simp* **apply** *force*
apply (*assumption*+)
apply (*rule conjI*) **apply** (*rule activeCells-3-1*)
apply (*rule activeCells-4*)
apply *simp*
apply (*rule activeCells-3*)
apply (*rule activeCells-4*)
apply *simp* **apply** *force*
apply (*assumption*+)
apply (*rule conjI*) **apply** *simp*
apply (*rule conjI*) **apply** (*simp add*: *heapC-def add*: *stackC-def*)
apply (*rule conjI*) **apply** (*simp add*: *regionsf-def add*: *kf-def*)
apply (*rule conjI*) **apply** *simp*
apply (*rule conjI*) **apply** (*simp add*: *safeDirf-def add*: *kf-def*)
apply (*rule conjI*) **apply** *clarsimp*
apply (*rule conjI*) **apply** *clarsimp*
apply (*rule conjI*) **apply** *simp*
apply (*rule conjI*) **apply** (*subgoal-tac*
lnew $\notin$ *activeCells regS h* (*nat* (*the-Intg* (*the* (*sha* (*heapC*, *kf*))))))
apply (*elim conjE*)
apply (*rule equivH-CALL*) **apply** *assumption*+
apply *simp* **apply** *assumption*+
apply (*rule lnew-notin-activeCells*)
apply (*rule conjI*) **apply** *simp*
apply (*rule conjI*) **apply** *simp*
apply (*rule conjI*) **apply** *simp*
apply (*simp add*: *k0f-def add*: *kf-def*)
by *simp*

end

theory *dem-DECREGION*
imports *../JVMSAFE/JVMExec SVM2JVM SVMSemantics CertifSVM2JVM*
dem-translation
begin

no-translations *Norm s* == (*None*,*s*)
no-translations *ex-table-of m* == *snd* (*snd* (*snd m*))

declare *trInstr.simps* [*simp del*]
declare *equivS.simps* [*simp del*]
declare *extractBytecode-def* [*simp del*]

declare *initConsTable-def* [*simp del*]

axioms *decregion-method*:
⟦ *s* = (*None*, *shp*, *hp*, *ihp*,([],*vs*,*safeP*, *sigSafeMain*,*pc*,*z*)#[]);
shp (*heapC*,*kf*) = *Some* (*Intg k*);
shp (*heapC*,*k0f*) = *Some* (*Intg k0*);
shp (*heapC*,*regionsf*) = *Some* (*Addr l*);
hp l = *Some* (*Arr ty m regS*);
Invoke-static heapC decregion [] = *fst*(*snd*(*snd*(*snd*(*snd*(*the*(*method'* (*P'*,*safeP*) *sigSafeMain*)))))) ! *pc*
⟧ ⟹
P' |− *s* −*jvm*→ (*None*, *shp* ((*heapC*,*kf*)↦ *Intg k0*),
hp(*l* ↦ *Arr ty ma regS'*),
ihp, ([],*loc*,*safeP*,*sigSafeMain*,*pc*+*1*,*z*)#[])
∧ (∀ *j* ≤ *nat k0* . *region regS' hp j* = *region regS hp j*)

axioms *kf-Some-i*:
sha (*heapC*, *kf*) = *Some* (*Intg jj*)

axioms *k0f-Some-i*:
sha (*heapC*, *k0f*) = *Some* (*Intg ii*)

axioms *activeCells-3*:
⟦ *l'* ∉ *activeCells regS h k*; *l* ≠ *l'*; *l* ≠ *l''* ⟧
⟹ *l'* ∉ *activeCells regS* (*h*(*l* ↦ *A*, *l''*↦ *B*)) *k*

axioms *activeCells-3-1*:
⟦ *l'* ∉ *activeCells regS h k* ⟧
⟹ *l'* ∉ *activeCells regS* (*h*(*l'* ↦ *A*, *l''*↦ *B*)) *k*

axioms *activeCells-5*:
la ∉ *activeCells regS h k*
⟹ *la* ∉ *activeCells regS' h' k'*

axioms *equivS-DECREGION*:
equivS S S' n ctm d g
⟹ *equivS S S' n ctm d* (*g* |‘ {*l* ∈ *dom hm*. *fst* (*the* (*hm l*)) ≤ *nat* (*the-Intg* (*the* (*sha* (*heapC*, *k0f*))))})

axioms *inj-on-DECREGION*:
⟦ ((*hm*, *k*), *k0*, (*l*, *i*), *S*) = ((*H*, *ka*), *k0a*, *PC*, *Sa*); *inj-on g* (*dom H*) ⟧
⟹ *inj-on* (*g* |‘ {*l* ∈ *dom hm*. *fst* (*the* (*hm l*)) ≤ *nat* (*the-Intg* (*the* (*sha* (*heapC*, *k0f*))))})
(*dom* (*hm* |‘ {*p* ∈ *dom hm*. *fst* (*the* (*hm p*)) ≤ *k0*}))

axioms *equivH-DECREGION*:
⟦ ((*hm*, *k*), *k0*, (*l*, *i*), *S*) = ((*H*, *ka*), *k0a*, *PC*, *Sa*); *equivH* (*H*, *ka*) *h k' com regS d g* ⟧
⟹ *equivH* (*hm* |‘ {*p* ∈ *dom hm*. *fst* (*the* (*hm p*)) ≤ *k0*}, *k0*) (*dh*(*l'* ↦ *Arr ty*

ma regS'))
(*nat* (*the-Intg* (*the* ((*sh*((*heapC*, *kf*) ↦ *the* (*sh* (*heapC*, *k0f*)))) (*heapC*, *kf*))))) *com regS' d*
(*g* |‘ {*l* ∈ *dom hm*. *fst* (*the* (*hm l*)) ≤ *nat* (*the-Intg* (*the* (*sha* (*heapC*, *k0f*))))})

lemma *execSVMInstr-DECREGION* :
⟦ (*P'*, *cdm*, *ctm*, *com*) = *trSVM2JVM* ((*svms*, *ctmap*), *ini*, *ct*, *ah*, *ai*, *bc*);
cdm , *ctm*, *com* ⊢ ((*hm*, *k*), *k0*, (*l*, *i*), *S*) ≜ *S1'*;
(*fst* (*the* (*map-of svms l*)) ! *i*) = *DECREGION*;
execSVMInst DECREGION (*map-of ct*) (*hm*, *k*) *k0* (*l*, *i*) *S* = *Either.Right S2*;
drop (*the* (*cdm* (*l*, *i*))) (*extractBytecode P'*) =
trInstr (*the* (*cdm* (*l*, *i*))) *cdm' ctm' com pcc DECREGION* @ *bytecode'*
⟧ ⟹ ∃ *v' sh' dh' ih' fms'* .
P' ⊢ *S1'* −*jvm*→ (*v'*,*sh'*,*dh'*,*ih'*, *fms'*) ∧
cdm , *ctm*, *com* ⊢ *S2* ≜ (*v'*,*sh'*,*dh'*,*ih'*, *fms'*)

apply (*case-tac S1'*)
apply (*rename-tac v tup*)
apply (*case-tac tup*)
apply (*rename-tac sh tup*)
apply (*case-tac tup*)
apply (*rename-tac dh tup*)
apply (*case-tac tup*)
apply (*rename-tac ih fms*)
apply (*simp*)

apply (*unfold equivState-def*)
apply (*elim exE*,*elim conjE*)

apply (*rule-tac x*=*None* **in** *exI*)
apply (*rule-tac x*=*sh*((*heapC*,*kf*)↦ *the* (*sh* (*heapC*, *k0f*))) **in** *exI*)
apply (*rule-tac x*=*dh*(*l'* ↦ *Arr ty ma regS'*) **in** *exI*)
apply (*rule-tac x*=*ih* **in** *exI*)
apply (*rule-tac x*=[([], *vs*, *safeP*, *sigSafeMain*, *pc*+*1*, *ref*)] **in** *exI*)

apply (*rule conjI*)

apply (*subgoal-tac sha* (*heapC*, *k0f*) = *Some* (*Intg ii*))
apply (*subgoal-tac sha* (*heapC*, *kf*) = *Some* (*Intg jj*))

apply (*subgoal-tac* $P' \vdash (v,\ sh,\ dh,\ ih,fms) -jvm\rightarrow$
$(None,\ sh\ ((heapC,kf)\mapsto Intg\ (the\text{-}Intg\ (the\ (sha\ (heapC,\ k0f))))),$
$dh\ (l' \mapsto Arr\ ty\ ma\ regS'),$
$ih,\ ([],vs,safeP,sigSafeMain,pc+1,ref)\#[])$
$\wedge\ (\forall\ j \leq nat\ (the\text{-}Intg\ (the\ (sha\ (heapC,\ k0f))))\ .\ region\ regS'\ dh\ j = region\ regS\ dh\ j))$
apply (*elim conjE*) **apply**(*simp*)

apply (*rule decregion-method*)
apply (*simp*,*simp*,*simp*,*simp*,*simp*)

apply (*unfold extractBytecode-def*)
apply (*unfold trInstr.simps*)
apply *clarify*
apply (*rule sym*)
apply (*rule-tac ys*=[] **in** *nth-via-drop-append*)
apply *simp*
apply (*rule kf-Some-i*)
apply (*rule k0f-Some-i*)

apply (*frule nonJumping-Suc-pc*)
apply (*erule-tac sym* [**where** *t*=*DECREGION*])
apply (*simp add*: *nonJumping.simps*)
apply (*simp add*: *trInstr.simps*)

apply (*rule-tac x*=*hm* |‘ $\{p \in dom\ hm.\ fst\ (the\ (hm\ p)) \leq k0\}$ **in** *exI*)
apply (*rule-tac x*=*k0* **in** *exI*)
apply (*rule-tac x*=*k0* **in** *exI*)
apply (*rule-tac x*=(*l*,*Suc i*) **in** *exI*)
apply (*rule-tac x*=*S* **in** *exI*)

apply (*rule-tac x*=*sh*((*heapC*, *kf*) $\mapsto$ *the* (*sh* (*heapC*, *k0f*))) **in** *exI*)
apply (*rule-tac x*=*dh*($l' \mapsto Arr\ ty\ ma\ regS'$) **in** *exI*)
apply (*rule-tac x*=*ih* **in** *exI*)
apply (*rule-tac x*=*vs* **in** *exI*)
apply (*rule-tac x*=*pc* + *1* **in** *exI*)
apply (*rule-tac x*=*ref* **in** *exI*)

apply (*rule-tac x*=*nat* (*the-Intg* (*the* ((*sh*((*heapC*, *kf*) $\mapsto$ *the* (*sh* (*heapC*, *k0f*))))
(*heapC*, *kf*)))) **in** *exI*)
apply (*rule-tac x*=*nat* (*the-Intg* (*the* ((*sh*((*heapC*, *kf*) $\mapsto$ *the* (*sh* (*heapC*, *k0f*))))
(*heapC*, *k0f*)))) **in** *exI*)

apply (*rule-tac x*=*la* **in** *exI*)
apply (*rule-tac x*=*ty* **in** *exI*)
apply (*rule-tac x*=*m* **in** *exI*)
apply (*rule-tac x*=*S'* **in** *exI*)

apply (*rule-tac x=n* **in** *exI*)
apply (*rule-tac x=l'* **in** *exI*)
apply (*rule-tac x=regS'* **in** *exI*)
apply (*rule-tac x=l''* **in** *exI*)
apply (*rule-tac x=ma* **in** *exI*)
apply (*rule-tac x=m''* **in** *exI*)
apply (*rule-tac x=d* **in** *exI*)
apply (*rule-tac x=g |‘ {l ∈ dom hm. fst (the (hm l)) ≤ nat (the-Intg (the (sha (heapC, k0f))))}* **in** *exI*)

apply (*rule conjI*, *clarsimp*)
apply (*rule conjI*, *rule refl*)
apply (*rule conjI*, *rule refl*)
apply (*rule conjI*, *rule refl*)
apply (*rule conjI*) **apply** (*simp add*: *stackC-def add*: *heapC-def*)
apply (*rule conjI*, *assumption*)
apply (*rule conjI*) **apply** *clarsimp*
apply (*rule conjI*) **apply** (*rule activeCells-5*,*assumption*)
apply (*rule conjI*) **apply** (*rule activeCells-5*,*assumption*)
apply (*rule activeCells-5*,*assumption*)
apply (*rule conjI*) **apply** *clarsimp*
apply (*rule conjI*) **apply** (*simp add*: *stackC-def add*: *heapC-def*)
apply (*rule conjI*) **apply** (*simp add*: *regionsf-def add*: *kf-def*)
apply (*rule conjI*) **apply** *clarsimp*
apply (*rule conjI*) **apply** (*simp add*: *dirCellC-def add*: *heapC-def*)
apply (*rule conjI*) **apply** *clarsimp*
apply (*rule conjI*) **apply** *clarsimp*
apply (*rule conjI*) **apply** (*simp add*: *inj-on-def*)
apply (*rule conjI*) **apply** (*rule equivH-DECREGION*, *assumption+*)
apply (*rule conjI*) **apply** *simp*
apply (*rule conjI*) **apply** (*clarsimp*, *rule equivS-DECREGION*,*assumption*)
apply (*rule conjI*) **apply** *clarsimp*
by *simp*

end

theory *dem-BUILDCLS*
imports *../JVMSAFE/JVMExec SVM2JVM SVMSemantics CertifSVM2JVM dem-translation*
begin

no-translations *Norm s == (None,s)*
no-translations *ex-table-of m == snd (snd (snd m))*

declare *trInstr.simps* [*simp del*]
declare *equivS.simps* [*simp del*]
declare *extractBytecode-def* [*simp del*]
declare *initConsTable-def* [*simp del*]
declare *exec-instr.simps* [*simp*]

lemma *drop-Suc-2*:
drop n xs = (*y* # *ys*) ⟹ *drop* (*Suc n*) *xs* = *ys*
apply (*induct xs arbitrary*: *n*, *simp*)
apply(*simp add*:*drop-Cons nth-Cons split*:*nat.splits*)
done

axioms *reserveCell-method*:
⟦ *s* = (*None*, *shp*, *hp*, *ihp*, [([],*loc*,*safeP*, *sigSafeMain*,*pc*,*ref*)]);
k' = *nat* (*the-Intg* (*the* (*sha* (*heapC*, *kf*))));
shp (*stackC*,*Sf*) = *Some* (*Addr l*);
shp (*heapC*,*regionsf*) = *Some* (*Addr l'*);
shp (*dirCellC*, *safeDirf*) = *Some* (*Addr l''*); *hp l''* = *Some* (*Arr ty m''* *d*);
fst(*snd*(*snd*(*snd*(*snd*(*the*(*method'* (*P'*,*safeP*) *sigSafeMain*)))))) ! *pc* =
Invoke-static dirCellC reserveCell []
⟧ ⟹
P' |− *s* −*jvm*→ (*None*, *shp*, *hp'*, *ihp*, [([*Intg ii*], *loc*, *safeP*,*sigSafeMain*,*pc*+*1*,*ref*)])
∧
d (*nat ii*) = *Some* (*Addr lobj*) ∧
hp' = *hp*(*lobj* ↦ *obj*) ∧
lobj ∉ *activeCells regS h k'* ∧
lobj ∉ {*l*,*l'*,*l''*} ∧
the-obj obj = (*cellC*, *empty*) ∧
ii < *int m''* ∧ *0* < *ii*

axioms *insertCell-method-jvm*:
⟦ *s* = (*None*, *shp*, *hp*(*lobj* ↦ *obj*), *ihp*, [([*Intg iloc*, *Intg* (*int j*)],*vs*,*safeP*, *sigSafe-Main*,*pc*,*ref*)]);
shp (*heapC*,*regionsf*) = *Some* (*Addr l'*); *hp l'* = *Some* (*Arr ty m'* *regS*);
shp (*dirCellC*,*safeDirf*) = *Some* (*Addr l''*); *hp l''* = *Some* (*Arr ty m''* *d*);
d (*nat iloc*) = *Some* (*Addr lobj*);
fst(*snd*(*snd*(*snd*(*snd*(*the*(*method'* (*P'*,*safeP*) *sigSafeMain*)))))) ! *pc* =
Invoke-static heapC insertCell [*PrimT Integer*, *PrimT Integer*]
⟧ ⟹
P' |− *s* −*jvm*→ (*None*, *shp*, *hp*(*lobj* ↦ *obj*, *l'* ↦ *Arr ty m' regS'*), *ihp*, [([], *vs*, *safeP*,*sigSafeMain*,*pc*+*1*,*ref*)])

axioms *insertCell-method-equivState*:
⟦ *sh* (*stackC*, *Sf*) = *Some* (*Addr l*); *dh l* = *Some* (*Arr ty m S'*);
sh (*stackC*, *topf*) = *Some* (*Intg n*);
sh (*heapC*,*regionsf*) = *Some* (*Addr l'*); *dh l'* = *Some* (*Arr ty m'* *regS*);

```
    sh (dirCellC,safeDirf) = Some (Addr l''); dh l'' = Some (Arr ty m'' d);
    d (nat ii) = Some (Addr lobj);
    fst(snd(snd(snd(snd(the(method' (P',safeP) sigSafeMain)))))) ! pc =
    Invoke-static heapC insertCell [PrimT Integer, PrimT Integer];
    P' |-
        (None, sh, dh(lobj ↦ obj), ih, [([ Intg ii, Intg (int j)],vs,safeP, sigSafe-
Main,pc,ref)])
    -jvm→
     (None, sh, dh(lobj ↦ obj, l' ↦ Arr ty m' regS'), ih, [([], vs, safeP,sigSafeMain,pc+1,ref)])
  ⟧ ⟹
  region regS' (dh(lobj ↦ obj,
                   l' ↦ Arr ty m' regS',
                   l ↦ Arr ty m (S'(nat (n + 1) ↦ Intg ii)))) j = region regS dh j
∪ {lobj} ∧
  (∀ j''. j'' ≠ j ⟶
        region regS'
              (dh(lobj ↦ obj,
                 l' ↦ Arr ty m' regS',
                 l ↦ Arr ty m (S'(nat (n + 1) ↦ Intg ii))))
              j'' = region regS dh j'')
```

```
fun Item2val :: entries- ⇒ sheap ⇒ Item ⇒ val
where
  Item2val S sh (ItemConst v)    = (if (isBool v = True)
                                  then (if (the-BoolT v)
                                        then Intg 1
                                        else Intg 0)
                                  else Intg (the-IntT v))
| Item2val S sh (ItemVar l)      = the (S (nat (the-Intg (the (sh (stackC, topf))))
-  l))
| Item2val S sh (ItemRegSelf)    = the (sh (heapC, kf))
```

```
axioms fillAux-instrs:
 P'⊢ (None, sha, h(lobj ↦ Obj cellC [(tagGf, cellC) ↦ Intg (int (the (com C)))]),

       inih,  [([],
      vs[Suc 0 := Intg ii,
           2 := Addr lobj,
           3 := k-fresh],
      safeP,sigSafeMain, pc, ref)])
  -jvm→
    (None,sha,
      (let vitems = Intg (int (the (com C))) # map (Item2val S' sha) items;
        cnames = (tagGf, cellC) # map (λ i. (VName (''arg''@ nat2Str i),cellC))
[0..<length items];
          objs   = Obj cellC (empty(cnames [↦] vitems))
```

```
      in  h(lobj ↦ objs )),
    inih,
    [([],
      vs[Suc 0 := Intg ii,
           2 := Addr lobj,
           3 := k-fresh],
       safeP, sigSafeMain, pc + length (concat (map fillAux (zip items [1..<length
items + 1]))), ref)])


axioms regAux-instrs:
  P⊢
    (None, sha, h, inih,
    [([],
      vs[Suc 0 := Intg ii, 2 := Addr lobj],
      safeP, sigSafeMain, pc, ref)])
 −jvm→
    (None, sha, h, inih,
    [([],
     vs[Suc 0 := Intg ii, 2 := Addr lobj, 3 := Intg j],
     safeP, sigSafeMain, pc + length (regAux item), ref)])



axioms arrayIndex-S′-valid:
  ⟦ h l = Some (Arr ty m S′);
    shp (stackC, topf) = Some (Intg n) ⟧
   ⟹ n + 1 < int m ∧ 0 ≤ n + 1


lemma S2-BUILDCLS:
  execSVMInst (BUILDCLS C items item) (map-of ct) (hm, k) k0
      (l, i) S = Either.Right S2
   ⟹ ∃ j′.
      ((hm(getFresh hm ↦ (j′, C, map (item2Val S) items)), k), k0,
       (l, Suc i), Val (Val.Loc (getFresh hm)) # S) = S2 ∧
       j′ ≤ k
apply (unfold execSVMInst.simps)
apply (case-tac item2Stack (snd (hm, k)) S item,simp-all)
  apply (insert RightNotUndefined)
  apply (erule-tac x= S2 in allE, force)
 apply (split split-if-asm,simp)
  apply (rename-tac j)
  apply (rule-tac x=j in exI,simp)
 apply simp
by (erule-tac x= S2 in allE, force)

lemma equivS-n-ge-minus-1:
  equivS S S′ n ctm d g ⟹ n >= −1
```

apply (*case-tac S*)
apply (*simp add*: *equivS.simps*)
apply (*case-tac a*)
apply (*simp add*: *equivS.simps*)
apply (*simp add*: *equivS.simps*)
apply (*simp add*: *equivS.simps*)
apply (*case-tac x*)
by (*simp add*: *equivS.simps*)

axioms *equivS-ge-n-distinct-g*:
equivS S S' n ctm d g $\Longrightarrow$ ($\forall$ $p > 0$. *equivS S* (*S'*(*nat* ($p + n$) $\mapsto$ *A*)) *n ctm d g'*)

lemma *equivS2-BUILDCLS*:
⟦ *equivS S S' n ctm d g*;
d (*nat ii*) = *Some* (*Addr lobj*) ⟧
$\Longrightarrow$ *equivS* (*Val* (*Val.Loc* (*getFresh hm*)) # *S*) (*S'*(*nat* ($n + 1$) $\mapsto$ *Intg ii*)) ($n + 1$)
ctm d (*g*(*getFresh hm* $\mapsto$ *lobj*))
apply (*frule equivS-n-ge-minus-1*)
apply (*simp add*: *equivS.simps*)
apply (*frule equivS-ge-n-distinct-g*)
apply (*erule-tac x=1* **in** *allE*) **apply** *clarsimp*
apply (*subgoal-tac nat* ($1 + n$) = *nat* ($n + 1$))
apply *force*
by *arith*

axioms *maxPush-BUILDCLS*:
$n + 1 <$ *int m*

axioms *getFresh-notin-dom-h*:
getFresh hm $\notin$ *dom hm*

axioms *lobj-notin-ran-g*:
lobj $\notin$ *ran g*

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

axioms *S-good*:
⟦ *equivS S S'* (*the-Intg* (*the* (*sh* (*stackC*, *topf*)))) *ctm d g*;
execSVMInst (*BUILDCLS C items item*) (*map-of ct*) (*H*, *k*)
(*nat* (*the-Intg* (*the* (*sh* (*heapC*, *k0f*))))) (*l*, *i*) *S* = *Either.Right S2* ⟧
$\Longrightarrow$ ($\forall$ $i <$ *length items*. $\forall$ *v* . *items*!*i* = *ItemConst v* $\longrightarrow$ ($\forall$ *v'*. *v* $\neq$ *Val.Loc v'*)) $\wedge$
($\forall$ $i <$ *length items*. $\forall$ *n* . *items*!*i* = *ItemVar n* $\longrightarrow$ ($\forall$ *z* . *S*!+*n* $\neq$ *Cont z*))

$\wedge$
$(\forall\ i < length\ items.\ \forall\ n\ .\ items!i = ItemVar\ n \longrightarrow (\forall\ r\ .\ S!+n \neq Reg\ r)) \wedge$
$(\forall\ i < length\ items.\ items!i \neq ItemRegSelf) \wedge$
$(\forall\ n < length\ S.\ \forall\ z\ .\ S!+n \neq Cont\ z) \wedge$
$(\forall\ i < length\ items.\ \forall\ n\ .\ items!i = ItemVar\ n \longrightarrow n < nat\ (the\text{-}Intg\ (the\ (sh\ (stackC,\ topf)))) \wedge n < length\ S)$

axioms *map-argCell*:
$argCell\ [(tagGf,\ cellC) \mapsto Intg\ (int\ (the\ (com\ C))),\ map\ (\lambda i.\ (VName\ (''arg''\ @\ nat2Str\ i),\ cellC))\ [0..<length\ items]\ [\mapsto]\ map\ (Item2val\ S'\ sha)\ items]\ i =$
$argCell\ [(VName\ (''arg''\ @\ nat2Str\ i),\ cellC) \mapsto Item2val\ S'\ sha\ (items!i)]\ i$

axioms *argCell-i*:
$argCell\ [(VName\ (''arg''\ @\ nat2Str\ i),\ cellC) \mapsto A]\ i = A$

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

axioms *equivS-nth-2*:
$[\![\ equivS\ S\ S'\ ntop\ ctm\ d\ g;$
$\forall\ i < length\ S\ .\ \forall\ z\ .\ S!+i \neq Cont\ z;$
$i < length\ S;$
$S!+i = Val\ v]\!] \Longrightarrow equivV\ v\ (the\ (S'\ (nat\ ntop - i)))\ d\ g$

axioms *equivV-distinct-g*:
$equivV\ v\ (the\ (S'\ (nat\ (the\text{-}Intg\ (the\ (sh\ (stackC,\ topf)))) - n)))\ d\ g$
$\Longrightarrow equivV\ v\ (the\ (S'\ (nat\ (the\text{-}Intg\ (the\ (sh\ (stackC,\ topf)))) - n)))\ d\ g'$

lemma *equivV-items*:
$[\![\ i < length\ items;$
$equivS\ S\ S'\ (the\text{-}Intg\ (the\ (sh\ (stackC,\ topf))))\ ctm\ d\ g;$
$\forall\ i < length\ items.\ \forall\ v\ .\ items!i = ItemConst\ v \longrightarrow (\forall\ v'.\ v \neq Val.Loc\ v');$
$\forall\ i < length\ items.\ \forall\ n\ .\ items!i = ItemVar\ n \longrightarrow (\forall\ z\ .\ S!+n \neq Cont\ z);$
$\forall\ i < length\ items.\ \forall\ n\ .\ items!i = ItemVar\ n \longrightarrow (\forall\ r\ .\ S!+n \neq Reg\ r);$
$\forall\ i < length\ items.\ items!i \neq ItemRegSelf;$
$\forall\ n < length\ S.\ \forall\ z\ .\ S!+n \neq Cont\ z;$
$\forall\ i < length\ items.\ \forall\ n\ .\ items!i = ItemVar\ n \longrightarrow n < nat\ (the\text{-}Intg\ (the\ (sh\ (stackC,\ topf)))) \wedge n < length\ S]\!]$
$\Longrightarrow$
$equivV\ (item2Val\ S\ (items\ !\ i))$
$(argCell$
$[(tagGf,\ cellC) \mapsto Intg\ (int\ (the\ (com\ C))),\ map\ (\lambda i.\ (VName\ (''arg''\ @\ nat2Str\ i),\ cellC))\ [0..<length\ items]\ [\mapsto]$
$map\ (Item2val\ S'\ sh)\ items]$
$i)$
d
$(g(getFresh\ H \mapsto lobj))$
apply $(case\text{-}tac\ (items\ !\ i))$

apply *simp*
apply (*case-tac Val*)

apply *force*

apply (*simp add*: *equivV.simps*)
apply (*subst map-argCell*,*simp*)
apply (*simp add*: *isBool-def*)
apply (*subst argCell-i*,*simp*)

apply (*simp add*: *equivV.simps*)
apply (*subst map-argCell*,*simp*)
apply (*simp add*: *isBool-def*)
apply (*subst argCell-i*,*simp*)
apply (*rule impI*)
apply (*subst argCell-i*,*simp*)

apply (*rename-tac n*)
apply *simp*
apply (*case-tac S* !+ *n*)

apply *simp*
apply (*rotate-tac 6*)
apply (*erule-tac x*=*i* **in** *allE*) **apply** *simp* **apply** (*elim conjE*)
apply (*frule equivS-nth-2*)
apply *force*
apply *assumption*
apply *assumption*
apply (*subst map-argCell*)
apply *clarsimp*
apply (*subst argCell-i*)
apply (*erule equivV-distinct-g*)

apply *force*

apply *force*

by *force*

declare *equivV.simps* [*simp*]

axioms *com-C*:
com C = *Some* (*nat* (*the-Intg*
(*the* ([(*tagGf*, *cellC*) ↦ *Intg* (*int* (*the* (*com C*))),
map (λ*i*. (*VName* (*''arg''* @ *nat2Str i*), *cellC*)) [*0*..<*length items*] [↦]
map (*Item2val S' sha*) *items*]
(*tagGf*, *cellC*)))))

lemma *domH-BUILDCLS*:
[[*ran g = activeCells regS h (nat (the-Intg (the (sha (heapC, kf)))))*; *finite (dom H)*; *dom g = dom H*;
j ≤ (nat (the-Intg (the (sha (heapC, kf)))));
activeCells regS h (nat (the-Intg (the (sha (heapC, kf))))) ∩ {la, l', l''} = {};
lobj ∉ activeCells regS h (nat (the-Intg (the (sha (heapC, kf))))) ∧ lobj ∉ {la,l',l''};
equivS Sa S' (the-Intg (the (sha (stackC, topf)))) ctm d g;
∀ i < length items. ∀ v . items!i = ItemConst v ⟶ (∀ v'. v ≠ Val.Loc v');
∀ i < length items. ∀ n . items!i = ItemVar n ⟶ (∀ z . Sa!+n ≠ Cont z);
∀ i < length items. ∀ n . items!i = ItemVar n ⟶ (∀ r . Sa!+n ≠ Reg r);
∀ i < length items. items!i ≠ ItemRegSelf;
∀ n < length Sa. ∀ z . Sa!+n ≠ Cont z;
∀ i < length items. ∀ n . items!i = ItemVar n ⟶ n < nat (the-Intg (the (sha (stackC, topf)))) ∧ n < length Sa;
∀ j''. j'' ≠ j ⟶
region regS' (h(lobj ↦ Obj cellC
[(tagGf, cellC) ↦ Intg (int (the (com C))),
map (λi. (VName (''arg'' @ nat2Str i), cellC)) [0..<length items]
[↦]
map (Item2val S' sha) items],
l' ↦ Arr ty m' regS', la ↦ Arr ty m (S'(nat (n + 1) ↦ Intg ii)))) j'' = region regS h j'';
region regS' (h(lobj ↦ Obj cellC
[(tagGf, cellC) ↦ Intg (int (the (com C))),
map (λi. (VName (''arg'' @ nat2Str i), cellC)) [0..<length items] [↦]
map (Item2val S' sha) items],
l' ↦ Arr ty m' regS', la ↦ Arr ty m (S'(nat (n + 1) ↦ Intg ii)))) j = region regS h j ∪ {lobj};
∀ l∈dom H. ∃ l'. l' = the (g l) ∧ equivC (the (H l)) h l' (the (h l')) (nat (the-Intg (the (sha (heapC, kf))))) com regS d g]]
⟹ ∀ l∈dom (H(getFresh H ↦ (j, C, map (item2Val Sa) items))).
∃ l'a. l'a = the ((g(getFresh H ↦ lobj)) l) ∧
equivC (the ((H(getFresh H ↦ (j, C, map (item2Val Sa) items))) l))
(h(lobj ↦
Obj cellC
[(tagGf, cellC) ↦ Intg (int (the (com C))),
map (λi. (VName (''arg'' @ nat2Str i), cellC)) [0..<length items] [↦]
map (Item2val S' sha) items],
l' ↦ Arr ty m' regS', la ↦ Arr ty m (S'(nat (n + 1) ↦ Intg ii))))
l'a (the ((h(lobj ↦
Obj cellC
[(tagGf, cellC) ↦ Intg (int (the (com C))),
map (λi. (VName (''arg'' @ nat2Str i), cellC)) [0..<length items] [↦]
map (Item2val S' sha) items],

$l' \mapsto Arr\ ty\ m'\ regS'$, $la \mapsto Arr\ ty\ m\ (S'(nat\ (n + 1) \mapsto Intg\ ii))))$
$l'a))$
$(nat\ (the\text{-}Intg\ (the\ (sha\ (heapC, kf))))))\ com\ regS'\ d\ (g(getFresh\ H \mapsto lobj))$
apply (*rule ballI*)
apply (*erule-tac x=l* **in** *ballE*)
apply (*elim exE*, *elim conjE*)
apply (*case-tac* $l \neq getFresh\ H$)
apply (*rule-tac x=l'a* **in** *exI*)
apply (*rule conjI*)
apply *clarsimp*
apply *simp*
apply (*elim exE*) **apply** (*elim conjE*)
apply (*elim exE*) **apply** (*elim conjE*)
apply (*subgoal-tac the* $(g\ l) \neq lobj \wedge the\ (g\ l) \neq l' \wedge the\ (g\ l) \neq la$,*simp*)
prefer *2*
apply (*simp add*: *activeCells-def*)
apply (*erule-tac x=ja* **in** *allE*)+ **apply** *simp*
apply *blast*
apply (*rule-tac x=Obja* **in** *exI*)
apply (*rule-tac x=flds* **in** *exI*)
apply (*rule conjI*, *rule refl*)
apply (*rule conjI*, *rule refl*)
apply (*rule conjI*)
apply *clarsimp*
apply (*erule-tac x=ja* **in** *allE*)
apply (*case-tac* $ja \neq j$)
apply *clarsimp*
apply *clarsimp*
apply (*rule allI*)
apply (*erule-tac x=i* **in** *allE*)+
apply (*rule impI*) **apply** *simp*
apply (*case-tac* (*vs* ! *i*))
apply *simp*
apply (*elim conjE*)
apply (*subgoal-tac nata* $\neq getFresh\ H$,*simp*)
apply (*subgoal-tac getFresh H* $\notin$ *dom g*) **apply** *force*
apply (*subgoal-tac getFresh H* $\notin$ *dom H*) **apply** *simp*
apply (*rule getFresh-notin-dom-h*)
apply *simp*
apply *simp*

apply *simp*
apply (*rule-tac x=Obj* **in** *exI*)
apply (*rule-tac x=*[(*tagGf*, *cellC*) $\mapsto$ *Intg* (*int* (*the* (*com C*))), *map* (λi. (*VName* (''*arg*'' @ *nat2Str i*), *cellC*)) [*0..<length items*] [$\mapsto$]
map (*Item2val S' sha*) *items*] **in** *exI*)

apply (*rule conjI*, *rule refl*)
apply (*rule conjI*)
apply *clarsimp*

apply (*subgoal-tac com C* =
Some (*nat* (*the-Intg*
(*the* ([(*tagGf*, *cellC*) ↦ *Intg* (*int* (*the* (*com C*))),
map (λ*i*. (*VName* (*''arg''* @ *nat2Str i*), *cellC*))
[*0..<length items*] [↦] *map* (*Item2val S' sha*) *items*]
(*tagGf*, *cellC*))))),*simp*)
apply (*rule com-C*)
apply (*rule allI*)
apply (*rule impI*)
apply (*subgoal-tac equivV* (*item2Val Sa* (*items* ! *i*))
(*argCell*
[(*tagGf*, *cellC*) ↦ *Intg* (*int* (*the* (*com C*))),
map (λ*i*. (*VName* (*''arg''* @ *nat2Str i*), *cellC*)) [*0..<length items*] [↦]
map (*Item2val S' sha*) *items*]
i)
d (*g*(*getFresh H* ↦ *lobj*)),*simp*)
apply (*rule equivV-items*) **apply** *assumption*+ **apply** *clarsimp* **apply** *assumption*+ **apply** *clarsimp*
apply *simp*
apply *simp*
apply (*rule-tac x*=*Obj* **in** *exI*)
apply (*rule-tac x*=[(*tagGf*, *cellC*) ↦ *Intg* (*int* (*the* (*com C*))),
map (λ*i*. (*VName* (*''arg''* @ *nat2Str i*), *cellC*)) [*0..<length items*] [↦]
map (*Item2val S' sha*) *items*] **in** *exI*)
apply (*rule conjI*)
apply (*rule refl*)
apply (*rule conjI*)
apply (*rule com-C*)
apply (*rule allI*) **apply** (*rule impI*)
apply (*rule equivV-items*) **apply** *assumption*+ **apply** *clarsimp* **apply** *assumption*+ **apply** *clarsimp*
by *simp*

lemma *ran-upd-getFresh*:
⟦ *lobj* ∉ *ran g*; *x* ∈ *ran g*; *getFresh H* ∉ *dom g*⟧
⟹ *x* ∈ *ran* (*g*(*getFresh H* ↦ *lobj*))
apply (*simp add*: *ran-def*)
apply (*erule exE*)

apply (*erule-tac x=a* **in** *allE*,*simp*)
apply (*rule-tac x=a* **in** *exI*)
apply (*rule conjI*)
apply *blast*
by *blast*

lemma *disjoint-getFresh*:
⟦ *x* ∈ *ran* (*g*(*getFresh H* ↦ *lobj*)); *lobj* ∉ *ran g* ⟧
⟹ *x*∈ *ran g* ∨ *x* = *lobj*
apply (*simp add*: *ran-def*)
apply (*split split-if-asm*,*clarsimp*)
by (*rule-tac x=a* **in** *exI*, *simp*)

lemma *disjoint-getFresh-2*:
⟦ *x*∈ *ran g* ∨ *x* = *lobj*; *lobj* ∉ *ran g*; *getFresh H* ∉ *dom g* ⟧
⟹ *x* ∈ *ran* (*g*(*getFresh H* ↦ *lobj*))
apply (*erule disjE*)
apply (*simp add*: *ran-def*)
apply (*erule exE*)
apply (*rule-tac x=a* **in** *exI*)
apply (*rule conjI*,*clarsimp*)
apply *clarsimp*
by (*simp add*: *ran-def*)

axioms *getFresh-notin-dom-g*:
getFresh hm ∉ *dom g*

lemma *activeCells-BUILDCLS*:
⟦ *l′* ∉ *activeCells regS dh* (*nat* (*the-Intg* (*the* (*sha* (*heapC*, *kf*))))));
j ≤ (*nat* (*the-Intg* (*the* (*sha* (*heapC*, *kf*))))));
∀ *j′*. *j′*≠ *j* ⟶ *region regS′* (*h*(*lobj* ↦
Obj cellC
[(*tagGf*, *cellC*) ↦ *Intg* (*int* (*the* (*com C*))),
map (λ*i*. (*VName* (*′′arg′′* @ *nat2Str i*), *cellC*)) [*0..<length items*] [↦]
map (*Item2val S′ sha*) *items*],
l′ ↦ *Arr ty m′ regS′*, *la* ↦ *Arr ty m* (*S′*(*nat* (*n* + *1*) ↦ *Intg ii*)))) *j′* = *region regS h j′*;
ran g = *activeCells regS h* (*nat* (*the-Intg* (*the* (*sha* (*heapC*, *kf*))))));
region regS′ (*h*(*lobj* ↦
Obj cellC
[(*tagGf*, *cellC*) ↦ *Intg* (*int* (*the* (*com C*))),
map (λ*i*. (*VName* (*′′arg′′* @ *nat2Str i*), *cellC*)) [*0..<length items*] [↦]
map (*Item2val S′ sha*) *items*],
l′ ↦ *Arr ty m′ regS′*, *la* ↦ *Arr ty m* (*S′*(*nat* (*n* + *1*) ↦ *Intg ii*)))) *j* = *insert lobj* (*region regS h j*) ⟧
⟹ *ran* (*g*(*getFresh hm* ↦ *lobj*)) =

```
      activeCells regS'
       (h(lobj ↦
        Obj cellC
         [(tagGf, cellC) ↦ Intg (int (the (com C))),
                    map (λi. (VName (''arg'' @ nat2Str i), cellC)) [0..<length
items] [↦]
                    map (Item2val S' sha) items],
        l' ↦ Arr ty m' regS', la ↦ Arr ty m (S'(nat (n + 1) ↦ Intg ii))))
       (nat (the-Intg (the (sha (heapC, kf))))))
apply (rule equalityI)
 apply (rule subsetI)
 apply (subgoal-tac lobj ∉ ran g)
  prefer 2 apply (rule lobj-notin-ran-g)
 apply (subgoal-tac x∈ ran g ∨ x = lobj,simp)
  prefer 2 apply (rule disjoint-getFresh,assumption+)
 apply (erule disjE)

  apply (simp add: activeCells-def)
  apply (elim exE)
  apply (rule-tac x=ja in exI,simp,elim conjE)
  apply (erule-tac x=ja in allE)+
  apply (case-tac ja = j) apply simp
  apply simp
 apply simp

 apply (simp add: activeCells-def)
 apply (rule-tac x=j in exI)
 apply  (rule conjI)
 apply (erule-tac x=getFresh hm in allE)
 apply assumption
 apply simp

apply (rule subsetI)
apply (subgoal-tac lobj ∉ ran g)
 prefer 2 apply (rule lobj-notin-ran-g)
apply (subgoal-tac getFresh hm ∉ dom g)
 prefer 2 apply (rule getFresh-notin-dom-g)
apply (simp add: activeCells-def)
apply (erule exE) apply (elim conjE)
apply (erule-tac x=ja in allE)+
apply simp
apply (case-tac ja = j)
 apply simp
apply (erule disjE)
 apply simp defer
 apply (subgoal-tac lobj ∉ ran g)
  prefer 2 apply (rule lobj-notin-ran-g)
apply (subgoal-tac x ∈ ran g)
apply (rule ran-upd-getFresh) apply assumption+
```

apply (*subgoal-tac* $x \in \{p.\ \exists j{\leq}nat\ (the\text{-}Intg\ (the\ (sha\ (heapC,\ kf))))).\ p \in region\ regS\ h\ j\}$)
apply *simp*
apply *clarsimp*
apply (*subgoal-tac* $x \in region\ regS\ h\ ja$)
prefer 2 **apply** *simp*
apply (*subgoal-tac* $x \in ran\ g$)
apply (*subgoal-tac* $lobj \notin ran\ g$)
prefer 2 **apply** (*rule lobj-notin-ran-g*)
apply (*subgoal-tac* $getFresh\ hm \notin dom\ g$)
prefer 2 **apply** (*rule getFresh-notin-dom-g*)
apply (*rule disjoint-getFresh-2*) **apply** *simp* **apply** *simp* **apply** *simp*
apply (*subgoal-tac* $x \in \{p.\ \exists j{\leq}nat\ (the\text{-}Intg\ (the\ (sha\ (heapC,\ kf))))).\ p \in region\ regS\ h\ j\}$)
apply *simp*
apply *simp*
apply (*rule-tac* $x{=}ja$ **in** *exI*)
apply *simp*
apply (*subgoal-tac* $lobj \notin ran\ g$)
prefer 2 **apply** (*rule lobj-notin-ran-g*)
apply (*simp add*: *ran-def*)
by *force*

lemma *equivH-BUILDCLS*:
$[\![\ equivH\ (H,\ ka)\ h\ (nat\ (the\text{-}Intg\ (the\ (sha\ (heapC,\ kf)))))\ com\ regS\ d\ g;\ j \leq ka;$
$dom\ g = dom\ H;$
$activeCells\ regS\ h\ (nat\ (the\text{-}Intg\ (the\ (sha\ (heapC,\ kf))))) \cap \{la,\ l',\ l''\} = \{\};$
$lobj \notin activeCells\ regS\ h\ (nat\ (the\text{-}Intg\ (the\ (sha\ (heapC,\ kf)))))$; $lobj \notin \{la,\ l',\ l''\}$;
$equivS\ Sa\ S'\ (the\text{-}Intg\ (the\ (sha\ (stackC,\ topf))))\ ctm\ d\ g$;
$\forall\ i < length\ items.\ \forall\ v\ .\ items!i = ItemConst\ v \longrightarrow (\forall\ v'.\ v \neq Val.Loc\ v')$;
$\forall\ i < length\ items.\ \forall\ n\ .\ items!i = ItemVar\ n \longrightarrow (\forall\ z\ .\ Sa!{+}n \neq Cont\ z)$;
$\forall\ i < length\ items.\ \forall\ n\ .\ items!i = ItemVar\ n \longrightarrow (\forall\ r\ .\ Sa!{+}n \neq Reg\ r)$;
$\forall\ i < length\ items.\ items!i \neq ItemRegSelf$;
$\forall\ n < length\ Sa.\ \forall\ z\ .\ Sa!{+}n \neq Cont\ z$;
$\forall\ i < length\ items.\ \forall\ n\ .\ items!i = ItemVar\ n \longrightarrow n < nat\ (the\text{-}Intg\ (the\ (sha\ (stackC,\ topf)))) \wedge n < length\ Sa$;
$\forall\ j''.\ j'' \neq j \longrightarrow$
$region\ regS'$
$(h(lobj \mapsto\ Obj\ cellC\ [(tagGf,\ cellC) \mapsto Intg\ (int\ (the\ (com\ C))),$
$map\ (\lambda i.\ (VName\ (''arg''\ @\ nat2Str\ i),\ cellC))$
$[0..{<}length\ items]\ [\mapsto]$
$map\ (Item2val\ S'\ sha)\ items],$
$l' \mapsto Arr\ ty\ m'\ regS',$
$la \mapsto Arr\ ty\ m\ (S'(nat\ (n + 1) \mapsto Intg\ ii))))$

```
                j'' = region regS h j'';
    region regS'
          (h(lobj ↦  Obj cellC [(tagGf, cellC) ↦ Intg (int (the (com C))),
                                  map (λi. (VName (''arg'' @ nat2Str i), cellC))
[0..<length items] [↦]
                                  map (Item2val S' sha) items],
             l' ↦ Arr ty m' regS',
              la ↦ Arr ty m (S'(nat (n + 1) ↦ Intg ii)))) j = region regS h j ∪
{lobj}
   ⟧
  ⟹ equivH ((H(getFresh H ↦ (j, C, map (item2Val Sa) items))), ka)
          (h(lobj ↦ Obj cellC [(tagGf, cellC) ↦ Intg (int (the (com C))),
                                  map (λi. (VName (''arg'' @ nat2Str i), cellC))
[0..<length items] [↦]
                                  map (Item2val S' sha) items],
            l' ↦ Arr ty m' regS',
            la ↦ Arr ty m (S'(nat (n + 1) ↦ Intg ii))))
          (nat (the-Intg (the (sha (heapC, kf)))))
          com regS' d (g(getFresh H ↦ lobj))
apply (simp add: heapC-def add: stackC-def)
apply (fold heapC-def)
apply (unfold equivH.simps, elim conjE)
apply (rule conjI, clarsimp)
apply (rule conjI)
apply (rule activeCells-BUILDCLS,assumption+,simp,assumption+)
apply (rule conjI, simp)
apply (fold stackC-def)
apply (frule-tac la=la and l'=l' and l''=l'' and
               lobj=lobj and C=C and Sa=Sa in domH-BUILDCLS)
by (assumption+,simp+)
```

```
lemma notin-ran:
  x ∉ ran g ⟹ ∀ y . g y ≠ Some x
by (clarsimp, simp add: ran-def)

lemma inj-on-BUILDCLS:
  ⟦ ((hm, k), k0, (l, i), S) = ((H, ka), k0a, PC, Sa);
      inj-on g (dom H) ⟧
   ⟹ inj-on (g(getFresh hm ↦ lobj)) (dom (hm(getFresh hm ↦ (j', C, map
(item2Val S) items))))
apply (subgoal-tac getFresh hm ∉ dom g)
 prefer 2 apply (rule getFresh-notin-dom-g)
apply (subgoal-tac getFresh hm ∉ dom hm)
 prefer 2 apply (rule getFresh-notin-dom-h)
apply (subgoal-tac lobj ∉ ran g)
```

prefer *2* **apply** (*rule lobj-notin-ran-g*)
apply *clarsimp*
apply (*simp add*: *inj-on-def*)
apply (*split split-if-asm*)
apply (*simp add*: *getFresh-def add*: *fresh-def*,*auto*)
apply (*split split-if-asm*,*auto*)
apply (*frule notin-ran*)
apply (*erule-tac x*=*x* **in** *allE*)
by *simp*

axioms *activeCells-3*:
⟦ $l' \notin$ *activeCells regS h k*; $l \neq l'$; $l \neq l''$ ⟧
$\Longrightarrow l' \notin$ *activeCells regS* (*h*($l \mapsto A$, $l'' \mapsto B$)) *k*

axioms *activeCells-3-1*:
⟦ $l' \notin$ *activeCells regS h k* ⟧
$\Longrightarrow l' \notin$ *activeCells regS* (*h*($l' \mapsto A$, $l'' \mapsto B$)) *k*

axioms *activeCells-4*:
la $\notin$ *activeCells regS h k*
$\Longrightarrow$ *la* $\notin$ *activeCells regS' h k'*

axioms *activeCells-5*:
la $\notin$ *activeCells regS h k*
$\Longrightarrow$ *la* $\notin$ *activeCells regS' h' k'*

axioms *drop-append-length-2*:
drop n xs = (*ys* @ *zs*) @ *ms* $\Longrightarrow$ *drop* (*n* + *length ys*) *xs* = *zs* @ *ms*

declare *List.upt.upt-Suc* [*simp del*]
declare *execSVMInst.simps* [*simp del*]
declare *equivH.simps* [*simp del*]

thm *drop-append-length*

axioms *drop-append-length-3*:
drop n xs = *ys* @ *zs* @ *ms* $\Longrightarrow$ *drop* (*n* + *length ys*) *xs* = *zs* @ *ms*

lemma *execSVMInstr-BUILDCLS-reserveCell* :
⟦ (*P'*, *cdm*, *ctm*, *com*) = *trSVM2JVM* ((*svms*, *ctmap*), *ini*, *ct*, *ah*, *ai*, *bc*);
S1' = (*None*, *sh*, *dh*, *ih*, [([], *vs*, *safeP*, *sigSafeMain*, *pc*,*ref*)]);
k' = *nat* (*the-Intg* (*the* (*sha* (*heapC*, *kf*))));
(*fst* (*the* (*map-of svms p*)) ! *i*) = *BUILDCLS C items item*;
execSVMInst (*BUILDCLS C items item*) (*map-of ct*) (*hm*, *k*) *k0* (*p*, *i*) *S* = *Either.Right S2*;
drop (*the* (*cdm* (*p*, *i*))) (*extractBytecode P'*) =
trInstr (*the* (*cdm* (*p*, *i*))) *cdm' ctm' com pcc* (*BUILDCLS C items item*) @

bytecode';
sh (*stackC*, *Sf*) = *Some* (*Addr l*);
distinct [*l*, *l'*, *l''*];
sh (*heapC*, *regionsf*) = *Some* (*Addr l'*); *dh l'* = *Some* (*Arr ty m' regS*);
sh (*dirCellC*, *safeDirf*) = *Some* (*Addr l''*); *dh l''* = *Some* (*Arr ty m'' d*);
pc = *the* (*cdm* (*p*,*i*))

⟧ ⟹
P' ⊢ *S1'* −*jvm*→
(*None*,*sh*,*dh'*,*ih*, [([*Intg ii*], *vs*, *safeP*, *sigSafeMain*, *pc* + *1*, *ref*)]) ∧
d (*nat ii*) = *Some* (*Addr lobj*) ∧
dh' = *dh*(*lobj* ↦ *obj*) ∧
lobj ∉ *activeCells regS dh k'* ∧
lobj ∉ {*l*,*l'*,*l''*} ∧
the-obj obj = (*cellC*, *empty*) ∧
ii < *int m''* ∧ *0* < *ii*

apply (*subgoal-tac*
P' |−
(*None*,*sh*,*dh*,*ih*,[([], *vs*, *safeP*, *sigSafeMain*, *the* (*cdm* (*p*, *i*)), *ref*)])
−*jvm*→
(*None*, *sh*, *dh'*, *ih*, [([*Intg ii*], *vs*, *safeP*,*sigSafeMain*,*the* (*cdm* (*p*, *i*))+*1*,*ref*)]) ∧
d (*nat ii*) = *Some* (*Addr lobj*) ∧
dh' = *dh*(*lobj* ↦ *obj*) ∧
lobj ∉ *activeCells regS dh k'* ∧
lobj ∉ {*l*, *l'*,*l''*} ∧
the-obj obj = (*cellC*, *empty*) ∧
ii < *int m''* ∧ *0* < *ii*)
prefer *2* **apply** (*rule reserveCell-method*) **apply** (*simp*,*assumption*+)
apply (*unfold extractBytecode-def*)
apply (*unfold trInstr.simps*)
apply (*unfold Let-def*)
apply (*simp*,*erule nth-via-drop-append*)
by *clarsimp*

lemma *execSVMInstr-BUILDCLS* :
⟦ (*P'*, *cdm*, *ctm*, *com*) = *trSVM2JVM* ((*svms*, *ctmap*), *ini*, *ct*, *ah*, *ai*, *bc*);
cdm , *ctm*, *com* ⊢ ((*hm*, *k*), *k0*, (*l*, *i*), *S*) ≜ *S1'*;
(*fst* (*the* (*map-of svms l*)) ! *i*) = *BUILDCLS C items item*;
execSVMInst (*BUILDCLS C items item*) (*map-of ct*) (*hm*, *k*) *k0* (*l*, *i*) *S* = *Either.Right S2*;
drop (*the* (*cdm* (*l*, *i*))) (*extractBytecode P'*) =
trInstr (*the* (*cdm* (*l*, *i*))) *cdm' ctm' com pcc* (*BUILDCLS C items item*) @ *bytecode'*
⟧ ⟹ ∃ *v' sh' dh' ih' fms'* .
P' ⊢ *S1'* −*jvm*→ (*v'*,*sh'*,*dh'*,*ih'*, *fms'*) ∧
cdm , *ctm*, *com* ⊢ *S2* ≜ (*v'*,*sh'*,*dh'*,*ih'*, *fms'*)

apply (*case-tac S1′*)
apply (*rename-tac v tup*)
apply (*case-tac tup*)
apply (*rename-tac sh tup*)
apply (*case-tac tup*)
apply (*rename-tac dh tup*)
apply (*case-tac tup*)
apply (*rename-tac ih fms*)
apply (*simp*)

apply (*subgoal-tac*
  ∃ *j*.
    ((*hm*(*getFresh hm* ↦ (*j*, *C*, *map* (*item2Val S*) *items*)), *k*), *k0*,
     (*l*, *Suc i*), *Val* (*Val.Loc* (*getFresh hm*)) # *S*) = *S2* ∧ *j* ≤ *k*)
prefer *2* **apply** (*erule S2-BUILDCLS*)
apply (*elim exE*, *elim conjE*)

apply (*unfold equivState-def*)
apply (*elim exE*,*elim conjE*)

apply (*rule-tac x=None* **in** *exI*)
apply (*rule-tac x=sha*((*stackC*, *topf*) ↦ *Intg* (*n* + *1*)) **in** *exI*)
apply (*rule-tac x=h*(*lobj* ↦ *Obj cellC* [(*tagGf*, *cellC*) ↦ *Intg* (*int* (*the* (*com C*))),
map (λ*i*. (*VName* (*''arg''* @ *nat2Str i*), *cellC*))
[*0*..<*length items*] [↦]
map (*Item2val S′ sha*) *items*],
l′ ↦ *Arr ty m′ regS′*,
la ↦ *Arr ty m* (*S′*(*nat* (*n* + *1*) ↦ *Intg ii*))) **in** *exI*)
apply (*rule-tac x=inih* **in** *exI*)
apply (*rule-tac x=*[([],
vs[*Suc 0* := *Intg ii*, *2* := *Addr lobj*, *3* := *Intg* (*int j*)],
safeP, *sigSafeMain*,
Suc (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc*
(*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*the* (*cdm* (*l*, *i*)) +
length (*regAux item*) +
length (*concat* (*map fillAux* (*zip items* [*Suc 0*..<*Suc* (*length items*)])))))))))))))))))))))), *ref*)] **in** *exI*)

apply (*subgoal-tac* ∃ *vs pc* . *fms*=[([],*vs*,*safeP*,*sigSafeMain*,*pc*,*ref*)])
prefer *2* **apply** *simp*
apply (*erule exE*)+

apply (*subgoal-tac length vs* = *10*)
prefer *2* **apply** (*rule length-vs*)

apply (*subgoal-tac PC* = (*l*,*i*))
prefer *2* **apply** *simp*

apply (*drule-tac r=S1'* **in** *trans*,*assumption*)

apply (*rule conjI*)

apply (*frule-tac dh'=h'* **and** *lobj=lobj* **and** *obj=obj* **and** *ii=ii* **in** *execSVMInstr-BUILDCLS-reserveCell*)
apply *assumption*+ **apply** *simp*
apply (*elim conjE*)

apply (*unfold exec-all-def*)
apply (*erule rtrancl-trans*)
apply (*unfold extractBytecode-def*)
apply (*unfold trInstr.simps*)
apply (*unfold Let-def*)

apply (*drule drop-Suc-append-2*)
apply (*subgoal-tac*
P'⊢ (*None*,*sh*,*h'*,*ih*,[([*Intg ii*], *vs*, *safeP*, *sigSafeMain*, *pc* + *1*, *ref*)])
−*jvm*→
(*None*,*sh*,*h'*,*ih*,[([], *vs*[*Suc 0* := *Intg ii*], *safeP*, *sigSafeMain*, *pc* + *1* + *1*, *ref*)]))
apply (*unfold exec-all-def*)
apply (*simp*,*erule rtrancl-trans*)
prefer *2*
apply (*rule r-into-rtrancl*)
apply (*clarify*)
apply (*unfold JVMExec.exec.simps*)
apply (*unfold Let-def*)
apply (*subgoal-tac*
(*fst* (*snd* (*snd* (*snd* (*snd* (*the* (*method'* (*P'*, *safeP*) *sigSafeMain*)))))) !
Suc (*the* (*cdm* (*l*, *i*)))) =
Store 1,*simp*)
apply (*simp*, *erule nth-via-drop-append*)

apply (*drule drop-Suc-append*)
apply (*subgoal-tac*
P'⊢ (*None*,*sh*,*h'*,*ih*,[([], *vs*[*Suc 0* := *Intg ii*], *safeP*, *sigSafeMain*, *pc* + *1* + *1*, *ref*)])
−*jvm*→
(*None*,*sh*,*h'*,*ih*,
[([*Addr l''*],
vs[*Suc 0* := *Intg ii*],
safeP, *sigSafeMain*, *pc* + *1* + *1* + *1*, *ref*)]))

apply (*unfold exec-all-def*)
apply (*simp*,*erule rtrancl-trans*)
prefer *2*
apply (*rule r-into-rtrancl*)
apply (*clarify*)
apply (*unfold JVMExec.exec.simps*)
apply (*unfold Let-def*)
apply (*subgoal-tac*
(*fst* (*snd* (*snd* (*snd* (*snd* (*the* (*method′* (*P′*, *safeP*) *sigSafeMain*)))))) !
(*Suc* (*Suc* (*the* (*cdm* (*l*, *i*)))))) =
Getstatic safeDirf dirCellC,*simp*)
apply (*erule nth-via-drop-append*)

apply (*drule drop-Suc-append*)
apply (*subgoal-tac*
P⊢ (*None*,*sh*,*h′*,*ih*,
[([*Addr l′′*],
vs[*Suc 0* := *Intg ii*],
safeP, *sigSafeMain*, *pc* + *1* + *1* + *1*, *ref*)])
−*jvm*→
(*None*,*sh*,*h′*,*ih*,
[([*Intg ii*,
Addr l′′],
vs[*Suc 0* := *Intg ii*],
safeP, *sigSafeMain*, *pc* + *1* + *1* + *1* + *1*, *ref*)]))
apply (*unfold exec-all-def*)
apply (*simp*,*erule rtrancl-trans*)
prefer *2*
apply (*rule r-into-rtrancl*)
apply (*clarify*)
apply (*unfold JVMExec.exec.simps*)
apply (*unfold Let-def*)
apply (*subgoal-tac*
(*fst* (*snd* (*snd* (*snd* (*snd* (*the* (*method′* (*P′*, *safeP*) *sigSafeMain*)))))) !
(*Suc* (*Suc* (*Suc* (*the* (*cdm* (*l*, *i*))))))) =
Load (*Suc 0*),*simp*)
apply (*erule nth-via-drop-append*)

apply (*drule drop-Suc-append*)
apply (*subgoal-tac*
P⊢ (*None*,*sh*,*h′*,*ih*,
[([*Intg ii*,
Addr l′′],
vs[*Suc 0* := *Intg ii*],
safeP, *sigSafeMain*, *pc* + *1* + *1* + *1* + *1*, *ref*)])
−*jvm*→
(*None*,*sh*,*h′*,*ih*,

```
      [([Addr lobj],
      vs[Suc 0 := Intg ii],
      safeP, sigSafeMain, pc + 1 + 1 + 1 + 1 + 1, ref)]))
apply (unfold exec-all-def)
apply (simp,erule rtrancl-trans)
prefer 2
 apply (rule r-into-rtrancl)
 apply (clarify)
 apply (unfold JVMExec.exec.simps)
 apply (unfold Let-def)
 apply (subgoal-tac
   (fst (snd (snd (snd (snd (the (method' (P', safeP) sigSafeMain)))))) !
    (Suc (Suc (Suc (Suc (the (cdm (l, i))))))) =
     ArrLoad,simp)
apply (simp add: raise-system-xcpt-def)
apply (erule nth-via-drop-append)


apply (drule drop-Suc-append)
apply (subgoal-tac
  P⊢ (None,sh,h',ih,
       [([Addr lobj],
       vs[Suc 0 := Intg ii],
       safeP, sigSafeMain, pc + 1 + 1 + 1 + 1 + 1, ref)])
  −jvm→
       (None,sh,h',ih,
       [([],
       vs[Suc 0 := Intg ii,
             2 := Addr lobj],
       safeP, sigSafeMain, pc + 1 + 1 + 1 + 1 + 1 + 1, ref)]))
apply (unfold exec-all-def)
apply (simp,erule rtrancl-trans)
prefer 2
 apply (rule r-into-rtrancl)
 apply (clarify)
 apply (unfold JVMExec.exec.simps)
 apply (unfold Let-def)
 apply (subgoal-tac
   (fst (snd (snd (snd (snd (the (method' (P', safeP) sigSafeMain)))))) !
    (Suc (Suc (Suc (Suc (Suc (the (cdm (l, i)))))))) =
    Store 2,simp)
apply (erule nth-via-drop-append)


apply (drule drop-Suc-append)
apply (subgoal-tac
  P⊢ (None,sh,h',ih,
       [([],
       vs[Suc 0 := Intg ii,
```

```
            2 := Addr lobj],
       safeP, sigSafeMain, pc + 1 + 1 + 1 + 1 + 1 + 1, ref)])
  −jvm→
       (None,sh,h',ih,
       [([Addr lobj],
       vs[Suc 0 := Intg ii,
            2 := Addr lobj],
       safeP, sigSafeMain, pc + 1 + 1 + 1 + 1 + 1 + 1 + 1, ref)]))
apply (unfold exec-all-def)
apply (simp,erule rtrancl-trans)
prefer 2
 apply (rule r-into-rtrancl)
 apply (clarify)
 apply (unfold JVMExec.exec.simps)
 apply (unfold Let-def)
 apply (subgoal-tac
   (fst (snd (snd (snd (snd (the (method' (P', safeP) sigSafeMain)))))) !
    (Suc (Suc (Suc (Suc (Suc (Suc (the (cdm (l, i)))))))))) =
    Load 2,simp)
apply (erule nth-via-drop-append)


apply (drule drop-Suc-append)
apply (subgoal-tac
  P'⊢ (None,sh,h',ih,
       [([Addr lobj],
       vs[Suc 0 := Intg ii,
            2 := Addr lobj],
       safeP, sigSafeMain, pc + 1 + 1 + 1 + 1 + 1 + 1 + 1, ref)])
  −jvm→
       (None,sh,h',ih,
       [([Intg (int (the (com C))),
          Addr lobj],
       vs[Suc 0 := Intg ii,
            2 := Addr lobj],
       safeP, sigSafeMain, pc + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1, ref)]))
apply (unfold exec-all-def)
apply (simp,erule rtrancl-trans)
prefer 2
 apply (rule r-into-rtrancl)
 apply (clarify)
 apply (unfold JVMExec.exec.simps)
 apply (unfold Let-def)
 apply (subgoal-tac
   (fst (snd (snd (snd (snd (the (method' (P', safeP) sigSafeMain)))))) !
    (Suc (Suc (Suc (Suc (Suc (Suc (Suc (the (cdm (l, i)))))))))) =
    LitPush (Intg (int (the (com C)))),simp)
apply (erule nth-via-drop-append)
```

apply (*drule drop-Suc-append*)
apply (*subgoal-tac*
  P ⊢ (*None*,*sh*,*h'*,*ih*,
        [([*Intg* (*int* (*the* (*com C*))),
           Addr lobj],
        vs[*Suc 0* := *Intg ii*,
              2 := *Addr lobj*],
        safeP, *sigSafeMain*, *pc* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1*, *ref*)])
  −*jvm*→
      (*None*,*sh*, *h'*(*lobj* ↦ *Obj cellC* (*empty*((*tagGf*, *cellC*) ↦ *Intg* (*int* (*the* (*com* *C*))))))),*ih*,
       [([],
       vs[*Suc 0* := *Intg ii*,
             2 := *Addr lobj*],
       safeP, *sigSafeMain*, *pc* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1*, *ref*)]))
apply (*unfold exec-all-def*)
apply (*simp*,*erule rtrancl-trans*)
prefer *2*
 apply (*rule r-into-rtrancl*)
 apply (*clarify*)
 apply (*unfold JVMExec.exec.simps*)
 apply (*unfold Let-def*)
 apply (*subgoal-tac*
   (*fst* (*snd* (*snd* (*snd* (*snd* (*the* (*method'* (*P'*, *safeP*) *sigSafeMain*)))))) !
    (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*the* (*cdm* (*l*, *i*))))))))))) =
    Putfield tagGf cellC,*simp*)
apply (*simp add*: *raise-system-xcpt-def*)
apply (*erule nth-via-drop-append*)

apply (*drule drop-Suc-append*)
apply (*subgoal-tac*
  P ⊢
    (*None*, *sh*,
    h'(*lobj* ↦ *Obj cellC* (*empty*((*tagGf*, *cellC*) ↦ *Intg* (*int* (*the* (*com* *C*)))))),
    ih,
    [([],
      vs[*Suc 0* := *Intg ii*, *2* := *Addr lobj*],
      safeP, *sigSafeMain*, *pc* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1*, *ref*)])
 −*jvm*→
    (*None*, *sh*,
    h'(*lobj* ↦ *Obj cellC* [(*tagGf*, *cellC*) ↦ *Intg* (*int* (*the* (*com* *C*)))]),
    ih,
    [([],
     vs[*Suc 0* := *Intg ii*, *2* := *Addr lobj*, *3* := *Intg* (*int j*)],
     safeP, *sigSafeMain*,
     pc + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *length* (*regAux item*), *ref*)]))
prefer *2* **apply** (*rule regAux-instrs*)

apply (*unfold exec-all-def*)
apply (*simp*, *erule rtrancl-trans*)

apply (*drule drop-append-length-3*)
apply (*subgoal-tac*
  P ⊢ (*None*, *sh*,
    h'(*lobj* ↦ *Obj cellC* [(*tagGf*, *cellC*) ↦ *Intg* (*int* (*the* (*com C*)))]),
    ih, [([],
    vs[*Suc 0* := *Intg ii*,
      2 := *Addr lobj*,
      3 := *Intg* (*int j*)],
    safeP,*sigSafeMain*,
    pc + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *length* (*regAux item*), *ref*)])
  −*jvm*→
    (*None*,*sh*,
      (*let vitems* = *Intg* (*int* (*the* (*com C*))) # *map* (*Item2val S' sh*) *items*;
        cnames = (*tagGf*, *cellC*) # *map* (λ *i*. (*VName* (*''arg''*@ *nat2Str i*),*cellC*))
[*0*..<*length items*];
          objs = *Obj cellC* (*empty*(*cnames* [↦] *vitems*))
       in *h'*(*lobj* ↦ *objs*)),
    ih,
    [([],
      vs[*Suc 0* := *Intg ii*,
        2 := *Addr lobj*,
        3 := *Intg* (*int j*)],
      safeP, *sigSafeMain*,
      pc + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *length* (*regAux item*) +
      length (*concat* (*map fillAux* (*zip items* [*1*..<*length items* + *1*]))), *ref*)]))
prefer *2* **apply** (*rule fillAux-instrs*)
apply (*unfold exec-all-def*)
apply (*simp*,*erule rtrancl-trans*)

apply (*drule drop-append-length-3*)
apply (*subgoal-tac*
  P ⊢
  (*None*,*sh*,
      (*let vitems* = *Intg* (*int* (*the* (*com C*))) # *map* (*Item2val S' sh*) *items*;
        cnames = (*tagGf*, *cellC*) # *map* (λ *i*. (*VName* (*''arg''*@ *nat2Str i*),*cellC*))
[*0*..<*length items*];
          objs = *Obj cellC* (*empty*(*cnames* [↦] *vitems*))
       in *h'*(*lobj* ↦ *objs*)),
    ih,
    [([],
      vs[*Suc 0* := *Intg ii*,
        2 := *Addr lobj*,
        3 := *Intg* (*int j*)],
      safeP, *sigSafeMain*,

```
      pc + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + length (regAux item) +
      length (concat (map fillAux (zip items [1..<length items + 1]))), ref)])
 −jvm→
   (None,sh,
      (let vitems = Intg (int (the (com C))) # map (Item2val S' sh) items;
        cnames = (tagGf, cellC) # map (λ i. (VName (''arg''@ nat2Str i),cellC))
[0..<length items];
           objs   = Obj cellC (empty(cnames [↦] vitems))
       in  h'(lobj ↦ objs )),
     ih,
     [([Intg (int j)],
      vs[Suc 0 := Intg ii,
            2 := Addr lobj,
            3 := Intg (int j)],
      safeP, sigSafeMain,
      pc + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + length (regAux item) +
      length (concat (map fillAux (zip items [1..<length items + 1]))) + 1, ref)]))
apply (unfold exec-all-def)
apply (simp,erule rtrancl-trans)
prefer 2
 apply (rule r-into-rtrancl)
 apply (clarify)
 apply (unfold JVMExec.exec.simps)
 apply (unfold Let-def)
 apply (subgoal-tac
   (fst (snd (snd (snd (snd (the (method' (P', safeP) sigSafeMain)))))) !
    (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (the (cdm (l, i))))))))))) +
length (regAux item) +
    length (concat (map fillAux (zip items [1..<length items + 1]))))) =
    Load 3,simp)
apply (simp,erule nth-via-drop-append)


apply (drule drop-Suc-append)
apply (subgoal-tac
  P'⊢
  (None,sh,
       h'(lobj ↦ Obj cellC [(tagGf, cellC) ↦ Intg (int (the (com C))),
                            map (λi. (VName (''arg'' @ nat2Str i), cellC)) [0..<length
items] [↦]
                              map (Item2val S' sh) items]),
     ih,
     [([Intg (int j)],
      vs[Suc 0 := Intg ii,
            2 := Addr lobj,
            3 := Intg (int j)],
      safeP, sigSafeMain,
      pc + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + length (regAux item) +
      length (concat (map fillAux (zip items [1..<length items + 1]))) + 1, ref)])
```

```
 −jvm→
    (None,sh,
       h'(lobj ↦ Obj cellC [(tagGf, cellC) ↦ Intg (int (the (com C))),
                            map (λi. (VName (''arg'' @ nat2Str i), cellC)) [0..<length
items] [↦]
                              map (Item2val S' sh) items]),
     ih,
     [([Intg ii,
        Intg (int j)],
       vs[Suc 0 := Intg ii,
            2 := Addr lobj,
            3 := Intg (int j)],
       safeP, sigSafeMain,
       pc + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + length (regAux item) +
        length (concat (map fillAux (zip items [1..<length items + 1]))) + 1 + 1,
ref)]))
apply (unfold exec-all-def)
apply (simp, erule rtrancl-trans)
prefer 2
 apply (rule r-into-rtrancl)
 apply (clarify)
 apply (unfold JVMExec.exec.simps)
 apply (unfold Let-def)
 apply (subgoal-tac
   (fst (snd (snd (snd (snd (the (method' (P', safeP) sigSafeMain)))))) !
    (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (the (cdm (l, i))))))))))))
+ length (regAux item) +
    length (concat (map fillAux (zip items [1..<length items + 1])))) =
    Load 1,simp)
apply (simp,erule nth-via-drop-append)


apply (drule drop-Suc-append)
apply (subgoal-tac
  P' |−
  (None,sha,
       h(lobj ↦
           Obj cellC
            [(tagGf, cellC) ↦ Intg (int (the (com C))),
                            map (λi. (VName (''arg'' @ nat2Str i), cellC)) [0..<length
items] [↦]
                              map (Item2val S' sha) items]),
     inih,
     [([Intg ii,
        Intg (int j)],
       vs[Suc 0 := Intg ii,
            2 := Addr lobj,
            3 := Intg (int j)],
       safeP, sigSafeMain,
```

```
      the (cdm (l, i)) + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + length (regAux
item) +
       length (concat (map fillAux (zip items [1..<length items + 1]))) + 1 + 1,
ref)])
 −jvm→
    (None, sha,
    h(lobj ↦
          Obj cellC
           [(tagGf, cellC) ↦ Intg (int (the (com C))),
                       map (λi. (VName (''arg'' @ nat2Str i), cellC)) [0..<length
items] [↦]
                         map (Item2val S' sha) items], l' ↦ Arr ty m' regS'),
    inih,
    [([],
      vs[Suc 0 := Intg ii,
           2 := Addr lobj,
           3 := Intg (int j)],
    safeP,sigSafeMain,
    the (cdm (l, i)) + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + length (regAux
item) +
    length (concat (map fillAux (zip items [1..<length items + 1]))) + 1 + 1 +
1, ref)]))
prefer 2
apply (rule-tac obj=Obj cellC [(tagGf, cellC) ↦ Intg (int (the (com C))),
                       map (λi. (VName (''arg'' @ nat2Str i), cellC)) [0..<length
items] [↦]
                                  map (Item2val S' sha) items] and iloc=ii in
insertCell-method-jvm)
apply (simp, assumption+)
apply (simp, erule nth-via-drop-append)
apply (unfold exec-all-def)
apply (elim conjE)
apply (simp, erule rtrancl-trans)



apply (drule drop-Suc-append)
apply (subgoal-tac
  P ⊢
  (None, sha,
    h(lobj ↦
          Obj cellC
           [(tagGf, cellC) ↦ Intg (int (the (com C))),
                       map (λi. (VName (''arg'' @ nat2Str i), cellC)) [0..<length
items] [↦]
                         map (Item2val S' sha) items], l' ↦ Arr ty m' regS'),
    inih,
    [([],
      vs[Suc 0 := Intg ii,
```

```
            2 := Addr lobj,
            3 := Intg (int j)],
     safeP,sigSafeMain,
     pc + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + length (regAux item) +
     length (concat (map fillAux (zip items [1..<length items + 1]))) + 1 + 1 +
1, ref)])
 −jvm→
    (None, sha,
     h(lobj ↦
           Obj cellC
            [(tagGf, cellC) ↦ Intg (int (the (com C))),
                        map (λi. (VName (''arg'' @ nat2Str i), cellC)) [0..<length
items] [↦]
                          map (Item2val S' sha) items], l' ↦ Arr ty m' regS'),
     inih,
     [([Intg n],
       vs[Suc 0 := Intg ii,
            2 := Addr lobj,
            3 := Intg (int j)],
     safeP,sigSafeMain,
     pc + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + length (regAux item) +
     length (concat (map fillAux (zip items [1..<length items + 1]))) + 1 + 1 +
1 + 1, ref)]))
apply (unfold exec-all-def)
apply (simp, erule rtrancl-trans)
prefer 2
 apply (rule r-into-rtrancl)
 apply (clarify)
 apply (unfold JVMExec.exec.simps)
 apply (unfold Let-def)
 apply (subgoal-tac
   (fst (snd (snd (snd (snd (the (method' (P', safeP) sigSafeMain)))))))) !
    (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (the (cdm (l,
i))))))))))) + length (regAux item) +
    length (concat (map fillAux (zip items [1..<length items + 1])))))) =
    Getstatic topf stackC,simp)
apply (simp, erule nth-via-drop-append)


apply (drule drop-Suc-append)
apply (subgoal-tac
  P'⊢
  (None, sha,
     h(lobj ↦
           Obj cellC
            [(tagGf, cellC) ↦ Intg (int (the (com C))),
                        map (λi. (VName (''arg'' @ nat2Str i), cellC)) [0..<length
items] [↦]
                          map (Item2val S' sha) items], l' ↦ Arr ty m' regS'),
```

```
    inih,
    [([Intg n],
      vs[Suc 0 := Intg ii,
            2 := Addr lobj,
            3 := Intg (int j)],
    safeP,sigSafeMain,
    pc + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + length (regAux item) +
    length (concat (map fillAux (zip items [1..<length items + 1]))) + 1 + 1 +
1 + 1, ref)])
 −jvm→
   (None, sha,
    h(lobj ↦
          Obj cellC
           [(tagGf, cellC) ↦ Intg (int (the (com C))),
                       map (λi. (VName (''arg'' @ nat2Str i), cellC)) [0..<length
items] [↦]
                         map (Item2val S' sha) items], l' ↦ Arr ty m' regS'),
    inih,
    [([Intg 1,
       Intg n],
      vs[Suc 0 := Intg ii,
            2 := Addr lobj,
            3 := Intg (int j)],
    safeP,sigSafeMain,
    pc + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + length (regAux item) +
    length (concat (map fillAux (zip items [1..<length items + 1]))) + 1 + 1 +
1 + 1 + 1, ref)]))
apply (unfold exec-all-def)
apply (simp, erule rtrancl-trans)
prefer 2
 apply (rule r-into-rtrancl)
 apply (clarify)
 apply (unfold JVMExec.exec.simps)
 apply (unfold Let-def)
 apply (subgoal-tac
   (fst (snd (snd (snd (snd (the (method' (P', safeP) sigSafeMain)))))))) !
    (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (the (cdm
(l, i))))))))))))) + length (regAux item) +
    length (concat (map fillAux (zip items [1..<length items + 1])))))))) =
    LitPush (Intg 1),simp)
apply (simp, erule nth-via-drop-append)


apply (drule drop-Suc-append)
apply (subgoal-tac
  P'⊢
  (None, sha,
    h(lobj ↦
          Obj cellC
```

```
          [(tagGf, cellC) ↦ Intg (int (the (com C))),
                        map (λi. (VName (''arg'' @ nat2Str i), cellC)) [0..<length
items] [↦]
                          map (Item2val S' sha) items], l' ↦ Arr ty m' regS'),
    inih,
    [([Intg 1,
       Intg n],
      vs[Suc 0 := Intg ii,
            2 := Addr lobj,
            3 := Intg (int j)],
    safeP,sigSafeMain,
    pc + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + length (regAux item) +
    length (concat (map fillAux (zip items [1..<length items + 1]))) + 1 + 1 +
1 + 1 + 1, ref)])
 −jvm→
   (None, sha,
    h(lobj ↦
          Obj cellC
           [(tagGf, cellC) ↦ Intg (int (the (com C))),
                        map (λi. (VName (''arg'' @ nat2Str i), cellC)) [0..<length
items] [↦]
                          map (Item2val S' sha) items], l' ↦ Arr ty m' regS'),
    inih,
    [([Intg (n + 1)],
      vs[Suc 0 := Intg ii,
            2 := Addr lobj,
            3 := Intg (int j)],
    safeP,sigSafeMain,
    pc + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + length (regAux item) +
    length (concat (map fillAux (zip items [1..<length items + 1]))) + 1 + 1 +
1 + 1 + 1 +1, ref)]))
apply (unfold exec-all-def)
apply (simp, erule rtrancl-trans)
prefer 2
 apply (rule r-into-rtrancl)
 apply (clarify)
 apply (unfold JVMExec.exec.simps)
 apply (unfold Let-def)
 apply (subgoal-tac
   (fst (snd (snd (snd (snd (the (method' (P', safeP) sigSafeMain)))))))) !
    (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (the
(cdm (l, i))))))))))))) +
   length (regAux item) + length (concat (map fillAux (zip items [1..<length items
+ 1])))))))))) =
    BinOp Add,simp)
apply (simp, erule nth-via-drop-append)


apply (drule drop-Suc-append)
```

```
apply (subgoal-tac
  P ⊢
  (None, sha,
     h(lobj ↦
           Obj cellC
            [(tagGf, cellC) ↦ Intg (int (the (com C))),
                        map (λi. (VName (''arg'' @ nat2Str i), cellC)) [0..<length
items] [↦]
                          map (Item2val S' sha) items], l' ↦ Arr ty m' regS'),
     inih,
     [([Intg (n + 1)],
       vs[Suc 0 := Intg ii,
            2 := Addr lobj,
            3 := Intg (int j)],
     safeP,sigSafeMain,
     pc + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + length (regAux item) +
     length (concat (map fillAux (zip items [1..<length items + 1]))) + 1 + 1 +
1 + 1 + 1 + 1, ref)])
 −jvm→
    (None,  sha((stackC, topf) ↦ Intg (n + 1)),
     h(lobj ↦
           Obj cellC
            [(tagGf, cellC) ↦ Intg (int (the (com C))),
                        map (λi. (VName (''arg'' @ nat2Str i), cellC)) [0..<length
items] [↦]
                          map (Item2val S' sha) items], l' ↦ Arr ty m' regS'),
     inih,
     [([],
       vs[Suc 0 := Intg ii,
            2 := Addr lobj,
            3 := Intg (int j)],
     safeP,sigSafeMain,
     pc + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + length (regAux item) +
     length (concat (map fillAux (zip items [1..<length items + 1]))) + 1 + 1 +
1 + 1 + 1 + 1 +1, ref)]))
apply (unfold exec-all-def)
apply (simp, erule rtrancl-trans)
prefer 2
 apply (rule r-into-rtrancl)
 apply (clarify)
 apply (unfold JVMExec.exec.simps)
 apply (unfold Let-def)
 apply (subgoal-tac
   (fst (snd (snd (snd (snd (the (method' (P', safeP) sigSafeMain)))))))) !
    (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc
(the (cdm (l, i))))))))))))) +
   length (regAux item) + length (concat (map fillAux (zip items [1..<length items
+ 1]))))))))))) =
    Putstatic topf stackC,simp)
```

apply (*simp*, *erule nth-via-drop-append*)

apply (*drule drop-Suc-append*)
apply (*subgoal-tac*
 P'⊢
 (*None*, *sha*((*stackC*, *topf*) ↦ *Intg* (*n* + *1*)),
  h(*lobj* ↦
   Obj cellC
    [(*tagGf*, *cellC*) ↦ *Intg* (*int* (*the* (*com C*))),
     map (λ*i*. (*VName* (''*arg*'' @ *nat2Str i*), *cellC*)) [*0*..<*length items*] [↦]
      map (*Item2val S' sha*) *items*], *l'* ↦ *Arr ty m' regS'*),
  inih,
  [([],
   vs[*Suc 0* := *Intg ii*,
    2 := *Addr lobj*,
    3 := *Intg* (*int j*)],
  safeP,*sigSafeMain*,
  pc + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *length* (*regAux item*) +
  length (*concat* (*map fillAux* (*zip items* [*1*..<*length items* + *1*]))) + *1* + *1* + *1* + *1* + *1* + *1* + *1*, *ref*)])
 −*jvm*→
  (*None*, *sha*((*stackC*, *topf*) ↦ *Intg* (*n* + *1*)),
  h(*lobj* ↦
   Obj cellC
    [(*tagGf*, *cellC*) ↦ *Intg* (*int* (*the* (*com C*))),
     map (λ*i*. (*VName* (''*arg*'' @ *nat2Str i*), *cellC*)) [*0*..<*length items*] [↦]
      map (*Item2val S' sha*) *items*], *l'* ↦ *Arr ty m' regS'*),
  inih,
  [([*Addr la*],
   vs[*Suc 0* := *Intg ii*,
    2 := *Addr lobj*,
    3 := *Intg* (*int j*)],
  safeP,*sigSafeMain*,
  pc + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *length* (*regAux item*) +
  length (*concat* (*map fillAux* (*zip items* [*1*..<*length items* + *1*]))) + *1* + *1* + *1* + *1* + *1* + *1* + *1* +*1*, *ref*)]))
apply (*unfold exec-all-def*)
apply (*simp*, *erule rtrancl-trans*)
prefer *2*
 apply (*rule r-into-rtrancl*)
 apply (*clarify*)
 apply (*unfold JVMExec.exec.simps*)
 apply (*unfold Let-def*)
 apply (*subgoal-tac*
  (*fst* (*snd* (*snd* (*snd* (*snd* (*the* (*method'* (*P'*, *safeP*) *sigSafeMain*)))))))) !
   (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc*

```
(Suc (the (cdm (l, i)))))))))))) +
   length (regAux item) + length (concat (map fillAux (zip items [1..<length items
+ 1])))))))))))) =
    Getstatic Sf stackC,simp)
apply (simp add: Sf-def add: topf-def)
apply (simp, erule nth-via-drop-append)


apply (drule drop-Suc-append)
apply (subgoal-tac
  P ⊢
  (None,  sha((stackC, topf) ↦ Intg (n + 1)),
     h(lobj ↦
           Obj cellC
            [(tagGf, cellC) ↦ Intg (int (the (com C))),
                        map (λi. (VName (''arg'' @ nat2Str i), cellC)) [0..<length
items] [↦]
                          map (Item2val S' sha) items], l'  ↦ Arr ty m' regS'),
     inih,
     [([Addr la],
       vs[Suc 0 := Intg ii,
            2 := Addr lobj,
            3 := Intg (int j)],
     safeP,sigSafeMain,
     pc + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + length (regAux item) +
     length (concat (map fillAux (zip items [1..<length items + 1]))) + 1 + 1 +
1 + 1 + 1 + 1 + 1 + 1, ref)])
 −jvm→
    (None,  sha((stackC, topf) ↦ Intg (n + 1)),
     h(lobj ↦
           Obj cellC
            [(tagGf, cellC) ↦ Intg (int (the (com C))),
                        map (λi. (VName (''arg'' @ nat2Str i), cellC)) [0..<length
items] [↦]
                          map (Item2val S' sha) items], l'  ↦ Arr ty m' regS'),
     inih,
     [([Intg (n + 1),
        Addr la],
       vs[Suc 0 := Intg ii,
            2 := Addr lobj,
            3 := Intg (int j)],
     safeP,sigSafeMain,
     pc + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + length (regAux item) +
     length (concat (map fillAux (zip items [1..<length items + 1]))) + 1 + 1 +
1 + 1 + 1 + 1 + 1 + 1 + 1, ref)]))
apply (unfold exec-all-def)
apply (simp, erule rtrancl-trans)
prefer 2
 apply (rule r-into-rtrancl)
```

```
 apply (clarify)
 apply (unfold JVMExec.exec.simps)
 apply (unfold Let-def)
 apply (subgoal-tac
   (fst (snd (snd (snd (snd (the (method' (P', safeP) sigSafeMain)))))))) !
   (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc
(Suc (Suc (the (cdm (l, i))))))))))))) +
   length (regAux item) + length (concat (map fillAux (zip items [1..<length items
+ 1])))))))))))) =
    Getstatic topf stackC,simp)
apply (simp, erule nth-via-drop-append)


apply (drule drop-Suc-append)
apply (subgoal-tac
  P'⊢
  (None,  sha((stackC, topf) ↦ Intg (n + 1)),
    h(lobj ↦
          Obj cellC
           [(tagGf, cellC) ↦ Intg (int (the (com C))),
                      map (λi. (VName (''arg'' @ nat2Str i), cellC)) [0..<length
items] [↦]
                        map (Item2val S' sha) items], l'  ↦ Arr ty m' regS'),
    inih,
    [([Intg (n + 1),
       Addr la],
      vs[Suc 0 := Intg ii,
            2 := Addr lobj,
            3 := Intg (int j)],
    safeP,sigSafeMain,
    pc + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + length (regAux item) +
    length (concat (map fillAux (zip items [1..<length items + 1]))) + 1 + 1 +
1 + 1 + 1 + 1 + 1 + 1 + 1, ref)])
 −jvm→
   (None,  sha((stackC, topf) ↦ Intg (n + 1)),
    h(lobj ↦
          Obj cellC
           [(tagGf, cellC) ↦ Intg (int (the (com C))),
                      map (λi. (VName (''arg'' @ nat2Str i), cellC)) [0..<length
items] [↦]
                        map (Item2val S' sha) items], l'  ↦ Arr ty m' regS'),
    inih,
    [([Intg ii,
       Intg (n + 1),
       Addr la],
      vs[Suc 0 := Intg ii,
            2 := Addr lobj,
            3 := Intg (int j)],
    safeP,sigSafeMain,
```

```
    pc + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + length (regAux item) +
    length (concat (map fillAux (zip items [1..<length items + 1]))) + 1 + 1 +
1 + 1 + 1 + 1 + 1 + 1 + 1 + 1, ref)]))
apply (unfold exec-all-def)
apply (simp, erule rtrancl-trans)
prefer 2
 apply (rule r-into-rtrancl)
 apply (clarify)
 apply (unfold JVMExec.exec.simps)
 apply (unfold Let-def)
 apply (subgoal-tac
   (fst (snd (snd (snd (snd (the (method' (P', safeP) sigSafeMain)))))))) !
   (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc
(Suc (Suc (Suc (the (cdm (l, i))))))))))) +
   length (regAux item) + length (concat (map fillAux (zip items [1..<length items
+ 1])))))))))))))) =
   Load (Suc 0),simp)
apply (simp, erule nth-via-drop-append)


apply (drule drop-Suc-append)
apply (subgoal-tac
  P'⊢
  (None,  sha((stackC, topf) ↦ Intg (n + 1)),
    h(lobj ↦
          Obj cellC
           [(tagGf, cellC) ↦ Intg (int (the (com C))),
                       map (λi. (VName (''arg'' @ nat2Str i), cellC)) [0..<length
items] [↦]
                         map (Item2val S' sha) items], l' ↦ Arr ty m' regS'),
    inih,
    [([Intg ii,
       Intg (n + 1),
       Addr la],
      vs[Suc 0 := Intg ii,
           2 := Addr lobj,
           3 := Intg (int j)],
    safeP,sigSafeMain,
    pc + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + length (regAux item) +
    length (concat (map fillAux (zip items [1..<length items + 1]))) + 1 + 1 +
1 + 1 + 1 + 1 + 1 + 1 + 1 + 1, ref)])
 −jvm→
   (None,  sha((stackC, topf) ↦ Intg (n + 1)),
     h(lobj ↦
         Obj cellC
          [(tagGf, cellC) ↦ Intg (int (the (com C))),
                       map (λi. (VName (''arg'' @ nat2Str i), cellC)) [0..<length
items] [↦]
                         map (Item2val S' sha) items],
```

```
        l' ↦ Arr ty m' regS', la ↦ Arr ty m (S'(nat (n + 1) ↦ Intg ii))),
    inih,
    [([],
      vs[Suc 0 := Intg ii,
            2 := Addr lobj,
            3 := Intg (int j)],
    safeP,sigSafeMain,
    pc + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + length (regAux item) +
    length (concat (map fillAux (zip items [1..<length items + 1]))) + 1 + 1 +
1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1, ref)]))
apply (unfold exec-all-def)
prefer 2
 apply (rule r-into-rtrancl)
 apply (clarify)
 apply (unfold JVMExec.exec.simps)
 apply (unfold Let-def)
 apply (subgoal-tac
   (fst (snd (snd (snd (snd (the (method' (P', safeP) sigSafeMain)))))))) !
    (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc
(Suc (Suc (Suc (Suc (Suc (the (cdm (l, i))))))))))) +
   length (regAux item) + length (concat (map fillAux (zip items [1..<length items
+ 1])))))))))))))) =
    ArrStore,simp)
apply (simp add: raise-system-xcpt-def)
apply (frule arrayIndex-S'-valid,assumption+)
apply simp
apply (simp, erule nth-via-drop-append)
apply simp



apply (frule nonJumping-Suc-pc)
apply (erule-tac sym [where t=BUILDCLS C items item])
apply (simp add: nonJumping.simps)
apply (simp add: trInstr.simps)

apply (rule-tac x=hm(getFresh hm ↦ (j, C, map (item2Val S) items)) in exI)
apply (rule-tac x=k in exI)
apply (rule-tac x=nat (the-Intg (the (sha (heapC, k0f)))) in exI)
apply (rule-tac x=(l, Suc i) in exI)
apply (rule-tac x=Val (Val.Loc (getFresh hm)) # S in exI)

apply (rule-tac x=sha((stackC, topf) ↦ Intg (n + 1)) in exI)
apply (rule-tac x=h(lobj ↦ Obj cellC [(tagGf, cellC) ↦ Intg (int (the (com C))),
                        map (λi. (VName (''arg'' @ nat2Str i), cellC)) [0..<length
items] [↦]
                            map (Item2val S' sha) items],
                    l' ↦ Arr ty m' regS',
                    la ↦ Arr ty m (S'(nat (n + 1) ↦ Intg ii))) in exI)
```

apply (*rule-tac x=inih* **in** *exI*)
apply (*rule-tac x=vs*[*Suc 0* := *Intg ii*, *2* := *Addr lobj*, *3* := *Intg* (*int j*)] **in** *exI*)
apply (*rule-tac x=Suc* (*the* (*cdm* (*l*, *i*)) + *length* (*regAux item*) + *length* (*concat* (*map fillAux* (*zip items* [*Suc 0*..<*Suc* (*length items*)])))))))))))))))))))))))) **in** *exI*)
apply (*rule-tac x=ref* **in** *exI*)

apply (*rule-tac x=nat* (*the-Intg* (*the* ((*sha*((*stackC*, *topf*) ↦ *Intg* (*n* + *1*))) (*heapC*, *kf*)))) **in** *exI*)

apply (*rule-tac x=nat* (*the-Intg* (*the* ((*sha*((*stackC*, *topf*) ↦ *Intg* (*n* + *1*))) (*heapC*, *k0f*)))) **in** *exI*)

apply (*rule-tac x=la* **in** *exI*)
apply (*rule-tac x=ty* **in** *exI*)
apply (*rule-tac x=m* **in** *exI*)
apply (*rule-tac x= S'*(*nat* (*n* + *1*) ↦ *Intg ii*) **in** *exI*)

apply (*rule-tac x=*(*n* + *1*) **in** *exI*)
apply (*rule-tac x=l'* **in** *exI*)
apply (*rule-tac x=regS'* **in** *exI*)
apply (*rule-tac x=l''* **in** *exI*)
apply (*rule-tac x=m'* **in** *exI*)
apply (*rule-tac x=m''* **in** *exI*)
apply (*rule-tac x=d* **in** *exI*)
apply (*rule-tac x=*(*g*(*getFresh hm* ↦ *lobj*)) **in** *exI*)

apply (*fold extractBytecode-def*)
apply (*frule-tac dh'=h'* **and** *lobj=lobj* **and** *obj=obj* **and** *ii=ii*
in *execSVMInstr-BUILDCLS-reserveCell*)
apply (*assumption+*, *simp add*: *trInstr.simps*, *assumption+*,*simp*)
apply (*elim conjE*)

apply (*thin-tac P'* ⊢ *?s1* −*jvm*→ *?s2*)

apply (*subgoal-tac*
P' |−
(*None*, *sha*, *dh*(*lobj* ↦ *Obj cellC* [(*tagGf*, *cellC*) ↦ *Intg* (*int* (*the* (*com C*))),
map (λ*i*. (*VName* (*''arg''* @ *nat2Str i*), *cellC*)) [*0*..<*length items*] [↦]
map (*Item2val S' sh*) *items*]),
ih,
[([*Intg ii*, *Intg* (*int j*)],
vs,*safeP*, *sigSafeMain*,
the (*cdm* (*l*, *i*)) + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *length* (*regAux*

```
item) +
       length (concat (map fillAux (zip items [1..<length items + 1]))) + 1 +
1,ref)])
  −jvm→
   (None, sha, dh(lobj ↦ Obj cellC [(tagGf, cellC) ↦ Intg (int (the (com C))),
                  map (λi. (VName (''arg'' @ nat2Str i), cellC)) [0..<length items]
[↦]
                    map (Item2val S' sh) items], l' ↦ Arr ty m' regS'),
    ih,
    [([], vs, safeP,sigSafeMain,
    the (cdm (l, i)) + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + length (regAux
item) +
      length (concat (map fillAux (zip items [1..<length items + 1]))) + 1 + 1 +
1,ref)]))
prefer 2 apply (rule-tac
   obj=Obj cellC [(tagGf, cellC) ↦ Intg (int (the (com C))),
       map (λi. (VName (''arg'' @ nat2Str i), cellC)) [0..<length items] [↦]
       map (Item2val S' sh) items] and
       iloc=ii and j=j and m''=m'' and d=d and
        pc=the (cdm (l, i)) + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + length
(regAux item) +
      length (concat (map fillAux (zip items [1..<length items + 1]))) + 1 + 1
        in insertCell-method-jvm)
apply (simp, assumption+, simp, simp, simp, assumption+)
apply (drule drop-Suc-append-2)
apply (drule drop-Suc-append)+
apply (drule drop-append-length)
apply (drule drop-append-length-2)
apply (drule drop-Suc-append)
apply (drule drop-Suc-append)
apply (simp add: extractBytecode-def)
apply (rule-tac zs=bytecode' in nth-via-drop-append, force)

apply (frule-tac
  obj=Obj cellC [(tagGf, cellC) ↦ Intg (int (the (com C))),
          map (λi. (VName (''arg'' @ nat2Str i), cellC)) [0..<length items] [↦]
          map (Item2val S' sh) items] and
  pc=the (cdm (l, i)) + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + length (regAux
item) +
       length (concat (map fillAux (zip items [1..<length items + 1]))) + 1 + 1
and
  P'=P' in  insertCell-method-equivState)
apply assumption+
apply (drule drop-Suc-append-2)
apply (drule drop-Suc-append)+
apply (drule drop-append-length)
apply (drule drop-append-length-2)
apply (drule drop-Suc-append)
apply (drule drop-Suc-append)
```

apply (*simp add*: *extractBytecode-def*)
apply (*rule-tac zs=bytecode′* **in** *nth-via-drop-append*, *force*)
apply *simp*
apply (*elim conjE*)

apply (*thin-tac* $P' \vdash$ *?s1* $-jvm\rightarrow$ *?s2*)

apply (*rule conjI*, *simp*)
apply (*rule conjI*, *simp*)
apply (*rule conjI*, *rule refl*)
apply (*rule conjI*, *rule refl*)
apply (*rule conjI*) **apply** *clarsimp*
apply (*rule conjI*, *assumption*)
apply (*rule conjI*) **apply** *clarsimp*
apply (*simp add*: *heapC-def add*: *stackC-def*)
apply (*rule conjI*) **apply** (*erule activeCells-5*)
apply (*rule conjI*) **apply** (*erule activeCells-5*)
apply (*erule activeCells-5*)
apply (*rule conjI*) **apply** *clarsimp*
apply (*rule conjI*) **apply** *clarsimp*
apply (*rule conjI*) **apply** *clarsimp*
apply (*rule conjI*) **apply** *clarsimp*
apply (*rule conjI*) **apply** *clarsimp*
apply (*rule conjI*) **apply** *clarsimp*
apply (*rule conjI*) **apply** *clarsimp*
apply (*rule conjI*) **apply** (*rule inj-on-BUILDCLS*,*assumption*+)
apply (*rule conjI*) **apply** *simp*
apply (*simp add*: *heapC-def add*: *stackC-def*)
apply (*fold heapC-def*)
apply (*subgoal-tac equivS Sa S′* (*the-Intg* (*the* (*sh* (*stackC*, *topf*))))
ctm d g)
apply (*frule S-good*) **apply** *simp*
apply (*rule-tac l′′=l′′* **in** *equivH-BUILDCLS*)
apply (*assumption*+,*simp*,*assumption*+,*simp*,*simp*,*simp*,*simp*,*simp*)
apply (*simp*,*simp*,*simp*,*assumption*+,*simp*,*simp*,*fold stackC-def*,*simp*)
apply (*rule conjI*) **apply** *clarsimp*
apply (*rule maxPush-BUILDCLS*)
apply (*rule conjI*) **apply** *clarsimp*
apply (*rule equivS2-BUILDCLS*,*assumption*+)
apply (*rule conjI*) **apply** (*simp add*: *heapC-def add*: *stackC-def*)
by *simp*

end

theory *dem-REUSE*
imports *../JVMSAFE/JVMExec SVM2JVM SVMSemantics CertifSVM2JVM dem-translation*
begin

no-translations *Norm s == (None,s)*

no-translations *ex-table-of m == snd (snd (snd m))*

declare *trInstr.simps* [*simp del*]
declare *equivS.simps* [*simp del*]

axioms *equivS-ge-n-distinct-g*:
equivS S S' n ctm d g $\Longrightarrow$ ($\forall$ *p* > *0*. *equivS S* (*S'*(*nat* (*p* + *n*) $\mapsto$ *A*)) *n ctm d g'*)

lemma *S2-REUSE*:
execSVMInst (*REUSE*) (*map-of ct*) (*hm*, *k*) *k0* (*l*, *i*) *S* = *Either.Right S2*
$\Longrightarrow\exists$ *b S'*. ((*hm*(*b* := *None*)(*getFresh hm* $\mapsto$ *the* (*hm b*)), *k*), *k0*, *incrPC* (*l*, *i*), *Val* (*Val.Loc* (*getFresh hm*)) # *S'*) = *S2* $\wedge$
S = *Val* (*Val.Loc b*) # *S'*
apply (*unfold execSVMInst.simps*)
apply (*case-tac S*,*simp-all*)
apply (*insert RightNotUndefined*)
apply (*erule-tac x*= *S2* **in** *allE*, *force*)
apply (*case-tac a*,*simp-all*)
apply (*case-tac Vala*,*simp-all*)
apply (*erule-tac x*= *S2* **in** *allE*, *force*)
apply *simp*
apply *simp*
by *simp*

lemma *equivS-Val-Loc*:
equivS (*Val* (*Val.Loc b*) # *S*) *S'* *n ctm d g*
$\Longrightarrow$ ($\exists$ *i*. *the* (*S'* (*nat n*)) = *Intg i* $\wedge$ ($\exists$ *l'*. *d* (*nat i*) = *Some* (*Addr l'*) $\wedge$ *g b* = *Some l'*))
by (*simp add*: *equivS.simps*)

lemma *equivS2-REUSE*:
⟦ *equivS* (*Val* (*Val.Loc b*) # *S*) *S′ n ctm d g*; *d* (*nat ii*) = *Some* (*Addr l′*); *g b′* = *Some l′*⟧
⟹ *equivS* (*Val* (*Val.Loc* (*getFresh hm*)) # *S*) (*S′*(*nat n* ↦ *Intg ii*)) *n ctm d* (*g*(*b*:=*None*)(*getFresh hm* ↦ *l′*))
apply (*frule equivS-Val-Loc*)
apply (*erule exE*, *elim conjE*)
apply (*erule exE*, *elim conjE*)
apply (*simp add*: *equivS.simps*)
apply (*elim conjE*)
apply (*frule equivS-ge-n-distinct-g*)
apply (*erule-tac x=1* **in** *allE*)
by *simp*

axioms *getFresh-notin-dom-h*:
getFresh hm ∉ *dom hm*

lemma *inj-on-REUSE*:
⟦ *inj-on g* (*dom hm*); *g b* = *Some l*; *dom g* = *dom hm*;
getFresh hm ∉ *dom g*; *getFresh hm* ∉ *dom hm*⟧
⟹ *inj-on* (*g*(*b* := *None*)(*getFresh hm* ↦ *l*)) (*dom* (*hm*(*b* := *None*)(*getFresh hm* ↦ *the* (*hm b*))))
apply *simp*
apply (*rule conjI*)
apply (*simp add*: *inj-on-def*)
apply (*simp add*: *inj-on-def*,*clarsimp*)
apply (*split split-if-asm*,*simp*)
apply (*split split-if-asm*,*simp*,*simp*)
apply (*erule-tac x=b* **in** *ballE*)
apply (*erule-tac x=x* **in** *ballE*)
apply *simp*
apply (*simp add*: *dom-def*)
apply (*subgoal-tac b* ∈ *dom hm*,*simp*)
apply (*subgoal-tac g b* = *Some l*)
apply (*subgoal-tac b* ∈ *dom g*)
apply (*simp add*: *dom-def*)
apply (*rule domI*,*assumption*)
by *simp*

lemma *x-ran-update*:
⟦ *g b* = *Some l′a*; *x* ∈ *ran* (*g*(*b* := *None*)(*getFresh hm* ↦ *l′a*))⟧
⟹ *x* ∈ *ran g*
apply (*simp add*: *ran-def*,*clarsimp*)

apply (*split split-if-asm*,*simp*)
 apply (*rule-tac x*=*b* **in** *exI*,*simp*)
apply (*simp*,*split split-if-asm*,*simp*)
by (*rule-tac x*=*a* **in** *exI*,*simp*)

lemma *x-ran-update-2*:
  ⟦ *g b* = *Some l*′*a*; *x* ∈ *ran g*; *getFresh hm* ∉ *dom g*; *inj-on g* (*dom h*)⟧
  ⟹ *x* ∈ *ran* (*g*(*b* := *None*)(*getFresh hm* ↦ *l*′*a*))
apply (*case-tac x*= *l*′*a*)
 apply (*simp add*: *ran-def*,*force*)
apply (*simp add*: *ran-def*)
apply (*elim exE*)
apply (*rule-tac x*=*a* **in** *exI*,*simp*)
apply (*rule conjI*)
 apply (*simp add*: *inj-on-def*)
 apply (*clarsimp*, *rule impI*)
by *clarsimp*

lemma *activeCells-REUSE*:
  ⟦ *la* ∉ *activeCells regS h* (*nat* (*the-Intg* (*the* (*sha* (*heapC*, *kf*))))));
    ran g = *activeCells regS h* (*nat* (*the-Intg* (*the* (*sha* (*heapC*, *kf*))))));
    g b = *Some l*′*a*;
    getFresh hm ∉ *dom g*;
    inj-on g (*dom hm*) ⟧
  ⟹ *ran* (*g*(*b* := *None*)(*getFresh hm* ↦ *l*′*a*)) =
      activeCells regS (*h*(*la* ↦ *Arr ty m* (*S*′*a*(*nat n* ↦ *vb*′)))) (*nat* (*the-Intg* (*the* (*sha* (*heapC*, *kf*))))))
apply (*frule l-not-in-cellReg*)
apply (*rule equalityI*)
 apply (*rule subsetI*)
 apply (*subgoal-tac x* ∈ *ran g*,*simp*)
  apply (*simp add*: *activeCells-def*)
  apply (*elim exE*, *elim conjE*)
  apply (*rule-tac x*=*j* **in** *exI*,*simp*)
  apply (*simp add*: *region-def*)
  apply (*elim conjE*)
  apply (*rule cellReg-step*)
   apply (*rule cellReg-basic*)
  apply (*erule-tac x*=*j* **in** *allE*)
  apply (*rule cellsReg-monotone-1*,*assumption*+)
  apply (*erule l-not-in-regs*)
 apply (*rule x-ran-update*,*assumption*+)
apply (*rule subsetI*)
apply (*subgoal-tac x* ∈ *activeCells regS h* (*nat* (*the-Intg* (*the* (*sha* (*heapC*, *kf*)))))))
 apply (*subgoal-tac x* ∈ *ran g*)
  prefer *2* **apply** *simp*
 apply (*rule x-ran-update-2*,*assumption*+)

apply (*simp add*: *activeCells-def*)
apply (*elim exE*, *elim conjE*)
apply (*rule-tac x=j* **in** *exI*,*simp*)
apply (*simp add*: *region-def*)
apply (*elim conjE*)
apply (*rule cellReg-step*)
apply (*rule cellReg-basic*)
apply (*erule-tac x=j* **in** *allE*)
apply (*rule cellsReg-monotone-2*,*assumption*+)
by (*erule l-not-in-regs*)

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

axioms *l-not-getFresh*:
$l \neq getFresh\ hm$

lemma *same-region-REUSE*:
$la \notin activeCells\ regS\ h\ (nat\ (the\text{-}Intg\ (the\ (sha\ (heapC,\ kf)))))$
$\Longrightarrow region\ regS\ h\ j = region\ regS\ (h(la \mapsto Arr\ ty\ m\ (S'a(nat\ n \mapsto vb'))))\ j$
apply (*frule l-not-in-cellReg*)
apply (*simp add*: *region-def*)
apply (*rule equalityI*)
apply (*rule subsetI*)
apply *clarsimp*
apply (*rule cellReg-step*)
apply (*rule cellReg-basic*)
apply (*erule-tac x=j* **in** *allE*)
apply (*rule cellsReg-monotone-1*,*assumption*+)
apply (*erule l-not-in-regs*)
apply (*rule subsetI*)
apply *clarsimp*
apply (*rule cellReg-step*)
apply (*rule cellReg-basic*)
apply (*erule-tac x=j* **in** *allE*)
apply (*rule cellsReg-monotone-2*,*assumption*+)
by (*erule l-not-in-regs*)

lemma *equivV-REUSE*:
$[\![\ i < length\ vs;$
$\forall\ i < length\ vs.\ \forall\ l''.\ vs!i = Val.Loc\ l'' \longrightarrow l'' \neq b \wedge l'' \in dom\ hm;$
$getFresh\ hm \notin dom\ g;$
$b \neq getFresh\ hm;$
$equivV\ (vs\ !\ i)\ (argCell\ flds\ i)\ d\ g\ ]\!]$
$\Longrightarrow equivV\ (vs\ !\ i)\ (argCell\ flds\ i)\ d\ (g(b := None)(getFresh\ hm \mapsto l'a))$
apply (*case-tac vs*!*i*)

apply *clarsimp*
apply *simp*
by *simp*

axioms *vs-i-good*:
$\forall\ i < length\ vs.\ \forall\ l''.\ vs!i = Val.Loc\ l'' \longrightarrow l'' \neq b \wedge l'' \in dom\ hm$

lemma *domH-REUSE*:
$[\![\ la \notin activeCells\ regS\ h\ (nat\ (the\text{-}Intg\ (the\ (sha\ (heapC, kf)))))$; $dom\ g = dom\ hm$;
$\forall l{\in}dom\ hm.\ \exists l'.\ l' = the\ (g\ l) \wedge equivC\ (the\ (hm\ l))\ h\ l'\ (the\ (h\ l'))\ (nat\ (the\text{-}Intg\ (the\ (sha\ (heapC, kf))))))\ com\ regS\ d\ g$;
$g\ b = Some\ l'a\ ]\!]$
$\Longrightarrow \forall l{\in}dom\ (hm(b := None)(getFresh\ hm \mapsto the\ (hm\ b))).$
$\exists l'.\ l' = the\ ((g(b := None)(getFresh\ hm \mapsto l'a))\ l) \wedge$
$equivC\ (the\ ((hm(b := None)(getFresh\ hm \mapsto the\ (hm\ b)))\ l))\ (h(la \mapsto Arr\ ty\ m\ (S'a(nat\ n \mapsto vb'))))\ l'$
$(the\ ((h(la \mapsto Arr\ ty\ m\ (S'a(nat\ n \mapsto vb'))))\ l'))\ (nat\ (the\text{-}Intg\ (the\ (sha\ (heapC, kf))))))\ com\ regS\ d$
$(g(b := None)(getFresh\ hm \mapsto l'a))$
apply (*rule ballI*)
apply (*erule-tac x=l* **in** *ballE*)
prefer *2* **apply** (*subgoal-tac* $l \neq getFresh\ hm$)
prefer *2* **apply** (*rule l-not-getFresh*)
apply *simp*
apply (*elim exE*)
apply (*rule-tac x=l'* **in** *exI*)
apply (*rule conjI*)
apply (*subgoal-tac* $l \neq getFresh\ hm$)
prefer *2* **apply** (*rule l-not-getFresh*)
apply *simp* **apply** (*elim conjE*)
apply (*unfold equivC.simps*)
apply (*elim exE*) **apply** (*elim conjE*)
apply (*rule-tac x=j* **in** *exI*)
apply (*rule-tac x=C* **in** *exI*)
apply (*rule-tac x=vs* **in** *exI*)
apply (*rule-tac x=Obj* **in** *exI*)
apply (*rule-tac x=cname* **in** *exI*)
apply (*rule-tac x=flds* **in** *exI*)
apply (*rule-tac x=tag* **in** *exI*)
apply (*rule-tac x=na* **in** *exI*)
apply (*rule-tac x=reg-j* **in** *exI*)
apply (*rule-tac x=tag'* **in** *exI*)

apply (*rule conjI*)
apply (*subgoal-tac* $l \neq getFresh\ hm$,*simp*)
apply (*rule l-not-getFresh*)
apply (*rule conjI*, *assumption*)

apply (*rule conjI*)
 apply (*subgoal-tac* $l' \neq la$,*simp*)
 apply (*simp add*: *activeCells-def*)
 apply (*erule-tac* $x{=}j$ **in** *allE*,*clarsimp*)
apply (*rule conjI*, *assumption*)
apply (*rule conjI*, *assumption*)
apply (*rule conjI*, *clarsimp*)
 apply (*rule same-region-REUSE*, *assumption*+)
apply (*rule conjI*, *assumption*)
apply (*rule conjI*, *assumption*)
apply (*rule conjI*, *assumption*)
apply (*rule conjI*, *assumption*)
apply *simp*
apply (*rule allI*, *rule impI*)
apply (*erule-tac* $x{=}i$ **in** *allE*,*simp*)
apply *clarsimp*
apply (*subgoal-tac* $\forall\ i < length\ vs.\ \forall\ l''.\ vs!i = Val.Loc\ l'' \longrightarrow l'' \neq b \wedge l'' \in dom\ hm$)
prefer *2* **apply** (*rule vs-i-good*)
apply (*subgoal-tac getFresh hm* $\notin$ *dom g*)
 prefer *2* **apply** (*subgoal-tac getFresh hm* $\notin$ *dom hm*,*simp*)
 apply (*rule getFresh-notin-dom-h*)
by (*rule equivV-REUSE*,*assumption*+,*blast*,*assumption*)

lemma *equivH-REUSE*:
  $[\![\ la \notin activeCells\ regS\ h\ (nat\ (the\text{-}Intg\ (the\ (sha\ (heapC,\ kf)))))$;
    $dom\ g = dom\ hm$; $g\ b = Some\ l'a$; $getFresh\ hm \notin dom\ g$; $inj\text{-}on\ g\ (dom\ hm)$;
    $equivH\ (hm,\ k)\ h\ (nat\ (the\text{-}Intg\ (the\ (sha\ (heapC,\ kf)))))\ com\ regS\ d\ g\ ]\!]$
  $\Longrightarrow equivH\ (hm(b := None)(getFresh\ hm \mapsto the\ (hm\ b)),\ k)\ (h(la \mapsto Arr\ ty\ m\ (S'a(nat\ n \mapsto vb'))))$
        $(nat\ (the\text{-}Intg\ (the\ (sha\ (heapC,\ kf)))))\ com\ regS\ d\ (g(b := None)(getFresh\ hm \mapsto l'a))$
apply (*unfold equivH.simps*, *elim conjE*)
apply (*rule conjI*, *clarsimp*)
apply (*rule conjI*)
apply (*rule activeCells-REUSE*, *assumption*+)
apply (*rule conjI*, *clarsimp*)
by (*rule domH-REUSE*,*assumption*+)

declare *extractBytecode-def* [*simp del*]
declare *initConsTable-def* [*simp del*]
declare *equivS.simps* [*simp del*]

axioms *vs-length*:

length vs = 10

axioms *copyCell-method-jvm*:
⟦ *s* = (*None*, *shp*, *hp*, *ihp*, [([*vb*],*loc*,*safeP*, *sigSafeMain*,*pc*,*z1*,*z2*)]);
shp (*stackC*,*Sf*) = *Some* (*Addr l*);
shp (*StackC*,*topf*) = *Some* (*Intg ntop*);
hp l = *Some* (*Arr ty ma S'*);
vb = *Intg i*;
d (*nat i*) = *Somde* (*Addr l'*);
Invoke-static heapC copyCell [*PrimT Integer*] =
fst(*snd*(*snd*(*snd*(*snd*(*the*(*method'* (*P'*,*safeP*) *sigSafeMain*)))))) !
pc
⟧ ⟹
P' |− *s* −*jvm*→ (*None*, *shp*, *hp*, *ihp*, [([*Intg ii*],*loc*,*safeP*,*sigSafeMain*,*pc*+*1*,*z1*,*z2*)])

axioms *copyCell-method-equivState*:
⟦ *s* = (*None*, *shp*, *hp*, *ihp*,
([*vb*],*loc*,*safeP*, *sigSafeMain*,*pc*,*z1*,*z2*)#[]);
shp (*stackC*,*Sf*) = *Some* (*Addr l*);
shp (*StackC*,*topf*) = *Some* (*Intg ntop*);
hp l = *Some* (*Arr ty ma S'*);
vb = *Intg i*;
d (*nat i*) = *Somde* (*Addr l'*);
Invoke-static heapC copyCell [*PrimT Integer*] =
fst(*snd*(*snd*(*snd*(*snd*(*the*(*method'* (*P'*,*safeP*) *sigSafeMain*)))))) !
pc
⟧ ⟹
P' |− *s* −*jvm*→ (*None*, *shp*,
hp,
ihp, ([*vb'*],*loc*,*safeP*,*sigSafeMain*,*pc*+*1*,*z1*,*z2*)#[]) ∧
vb' = *Intg ii* ∧
d (*nat i*) = *None* ∧
d (*nat ii*) = *Some* (*Addr l'*)

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

lemma *execSVMInstr-REUSE* :
⟦ (*P'*, *cdm*, *ctm*, *com*) = *trSVM2JVM* ((*svms*, *ctmap*), *ini*, *ct*, *ah*, *ai*, *bc*);
cdm , *ctm*, *com* ⊢ ((*hm*, *k*), *k0*, (*l*, *i*), *S*) ≙ *S1'*;
(*fst* (*the* (*map-of svms l*)) ! *i*) = *REUSE*;
execSVMInst REUSE (*map-of ct*) (*hm*, *k*) *k0* (*l*, *i*) *S* = *Either.Right S2*;
drop (*the* (*cdm* (*l*, *i*))) (*extractBytecode P'*) =
trInstr (*the* (*cdm* (*l*, *i*))) *cdm' ctm' com pcc REUSE* @ *bytecode'*
⟧ ⟹ ∃ *v' sh' dh' ih' fms'* .
P' ⊢ *S1'* −*jvm*→ (*v'*,*sh'*,*dh'*,*ih'*, *fms'*) ∧

cdm , ctm, com $\vdash$ $S2$ $\triangleq$ (v',sh',dh',ih', fms')

apply (*case-tac S1'*)
apply (*rename-tac v tup*)
apply (*case-tac tup*)
apply (*rename-tac sh tup*)
apply (*case-tac tup*)
apply (*rename-tac dh tup*)
apply (*case-tac tup*)
apply (*rename-tac ih fms*)
apply (*simp*)

apply (*subgoal-tac*
$\exists$ b S'. ((*hm*(*b* := *None*)(*getFresh hm* $\mapsto$ *the* (*hm b*)), *k*), *k0*, *incrPC* (*l*, *i*), *Val*
(*Val.Loc* (*getFresh hm*)) # S') = $S2$ $\wedge$
S = *Val* (*Val.Loc b*) # S')
prefer *2* **apply** (*rule S2-REUSE*,*assumption*)
apply (*elim exE*)

apply (*unfold equivState-def*)
apply (*elim exE*,*elim conjE*)

apply (*subgoal-tac equivS* (*Val* (*Val.Loc b*) # S') $S'a$ *n ctm d g*)
prefer *2* **apply** *clarsimp*
apply (*frule equivS-Val-Loc*)
apply (*elim exE*, *elim conjE*)
apply (*elim exE*, *elim conjE*)

apply (*subgoal-tac length vs = 10*)
prefer *2* **apply** (*rule vs-length*)

apply (*rule-tac x=None* **in** *exI*)
apply (*rule-tac x=sha* **in** *exI*)
apply (*rule-tac x= h*(*la* $\mapsto$ *Arr ty m* ($S'a$(*nat n* $\mapsto$ *Intg ia*))) **in** *exI*)
apply (*rule-tac x=inih* **in** *exI*)
apply (*rule-tac x=*[([],
vs[*Suc 0* := *Intg ia*],
safeP, *sigSafeMain*,
the (*cdm* (*l*, *i*)) + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1*,
ref)] **in** *exI*)

apply (*subgoal-tac* $\exists$ *vs pc* . *fms*=([],*vs*,*safeP*,*sigSafeMain*,*pc*,*ref*)#[])
prefer *2* **apply** *simp*
apply (*erule exE*)+
apply (*unfold exec-all-def*)

apply (*rule conjI*)

apply (*subgoal-tac PC = (l,i)*) **prefer** *2* **apply** *simp*
apply *simp* **apply** (*elim conjE*) **apply** *clarsimp*
apply (*subgoal-tac*
P ↾⊢
(*None*,*sha*,*h*,*inih*,[([], *vs*, *safeP*, *sigSafeMain*, *the* (*cdm* (*l*,*i*)), *ag*,*be*)])
−*jvm*→
(*None*,*sha*,*h*,*inih*,[([*the* (*sha* (*stackC*,*Sf*))], *vs*, *safeP*, *sigSafeMain*, *the* (*cdm* (*l*,*i*)) + *1*, *ag*,*be*)]))
apply (*unfold exec-all-def*)
apply (*erule rtrancl-trans*)

prefer *2*
apply (*rule r-into-rtrancl*)
apply (*clarify*)
apply (*unfold JVMExec.exec.simps*)
apply (*unfold Let-def*)
apply (*unfold extractBytecode-def*)
apply (*subgoal-tac*
(*fst* (*snd* (*snd* (*snd* (*snd* (*the* (*method′* (*P′*, *safeP*) *sigSafeMain*)))))) !
the (*cdm* (*l*, *i*))) = *Getstatic Sf stackC*,*simp*)
apply (*unfold trInstr.simps*, *erule nth-via-drop-append*)

apply (*drule drop-Suc-append*)
apply (*subgoal-tac*
P ↾⊢
(*None*, *sha*, *h*, *inih*,
[([*the* (*sha* (*stackC*, *Sf*))],
vs, *safeP*, *sigSafeMain*, *the* (*cdm* (*l*, *i*)) + *1*, *ag*, *be*)])
−*jvm*→
(*None*,*sha*,*h*,*inih*,
[([*the* (*sha* (*stackC*,*topf*)),
the (*sha* (*stackC*,*Sf*))],
vs, *safeP*, *sigSafeMain*, *the* (*cdm* (*l*, *i*)) + *1* + *1*, *ag*, *be*)]))
apply (*unfold exec-all-def*)
apply (*erule rtrancl-trans*)
prefer *2*
apply (*rule r-into-rtrancl*)
apply (*clarify*)
apply (*unfold JVMExec.exec.simps*)
apply (*unfold Let-def*)
apply (*subgoal-tac*
(*fst* (*snd* (*snd* (*snd* (*snd* (*the* (*method′* (*P′*, *safeP*) *sigSafeMain*)))))) ! *Suc* (*the* (*cdm* (*l*, *i*)))) =
Getstatic topf stackC,*simp*)

apply (*erule nth-via-drop-append*)

apply (*drule drop-Suc-append*)
apply (*subgoal-tac*
  P'⊢
    (*None*,*sha*,*h*,*inih*,
    [([*the* (*sha* (*stackC*,*topf*)),
      the (*sha* (*stackC*, *Sf*))],
    vs, *safeP*, *sigSafeMain*, *the* (*cdm* (*l*, *i*)) + *1* + *1*, *ag*, *be*)])
  −jvm→
    (*None*,*sha*,*h*,*inih*,
    [([*the* (*S'a* (*nat n*))],
    vs, *safeP*, *sigSafeMain*, *the* (*cdm* (*l*, *i*)) + *1* + *1* + *1*, *ag*, *be*)]))
apply (*unfold exec-all-def*)
apply (*erule rtrancl-trans*)
prefer *2*
 apply (*rule r-into-rtrancl*)
 apply (*clarify*)
 apply (*unfold JVMExec.exec.simps*)
 apply (*unfold Let-def*)
 apply (*subgoal-tac*
  (*fst* (*snd* (*snd* (*snd* (*snd* (*the* (*method'* (*P'*, *safeP*) *sigSafeMain*)))))) ! (*Suc* (*Suc* (*the* (*cdm* (*l*, *i*))))) =
    ArrLoad,*simp*)
 apply (*simp add*: *raise-system-xcpt-def*)
 apply (*subgoal-tac n >= 0*,*simp*)
 apply (*simp add*: *equivS.simps*)
apply (*erule nth-via-drop-append*)

apply (*drule drop-Suc-append*)
 apply (*frule equivS-Val-Loc*)
apply (*elim exE*, *elim conjE*)
apply (*elim exE*, *elim conjE*)
apply (*subgoal-tac*
  P'⊢
    (*None*,*sha*,*h*,*inih*,
    [([*the* (*S'a* (*nat n*))],
    vs, *safeP*, *sigSafeMain*, *the* (*cdm* (*l*, *i*)) + *1* + *1* + *1*, *ag*, *be*)])
  −jvm→
    (*None*,*sha*,*h*,*inih*,
    [([*Intg ia*],
    vs, *safeP*, *sigSafeMain*, *the* (*cdm* (*l*, *i*)) + *1* + *1* + *1* + *1*, *ag*, *be*)]))
prefer *2*
 apply (*rule-tac vb=the* (*S'a* (*nat n*)) **in** *copyCell-method-jvm*)
apply (*simp*, *assumption+*)
apply (*rule sym*)
apply (*simp*, *erule nth-via-drop*)

apply (*unfold exec-all-def*)
apply (*erule rtrancl-trans*)

apply (*drule drop-Suc-append*)
apply (*subgoal-tac*
  P'⊢
    (*None*,*sha*,*h*,*inih*,
    [([*Intg ia*],
    vs, *safeP*, *sigSafeMain*, *the* (*cdm* (*l*, *i*)) + *1* + *1* + *1* + *1*, *ag*, *be*)])
  −*jvm*→
    (*None*,*sha*,*h*,*inih*,
    [([],
    vs[*Suc 0* := *Intg ia*],
    safeP, *sigSafeMain*, *the* (*cdm* (*l*, *i*)) + *1* + *1* + *1* + *1* + *1*, *ag*, *be*)]))
apply (*unfold exec-all-def*)
apply (*erule rtrancl-trans*)
prefer *2*
 apply (*rule r-into-rtrancl*)
 apply (*clarify*)
 apply (*unfold JVMExec.exec.simps*)
 apply (*unfold Let-def*)
 apply (*subgoal-tac*
  (*fst* (*snd* (*snd* (*snd* (*snd* (*the* (*method'* (*P'*, *safeP*) *sigSafeMain*)))))) ! (*Suc* (*Suc*
(*Suc* (*Suc* (*the* (*cdm* (*l*, *i*)))))))) =
    Store 1,*simp*)
apply (*erule nth-via-drop-append*)

apply (*drule drop-Suc-append*)
apply (*subgoal-tac*
  P'⊢
    (*None*,*sha*,*h*,*inih*,
    [([],
    vs[*Suc 0* := *Intg ia*],
    safeP, *sigSafeMain*, *the* (*cdm* (*l*, *i*)) + *1* + *1* + *1* + *1* + *1*, *ag*, *be*)])
  −*jvm*→
    (*None*,*sha*,*h*,*inih*,
    [([*the* (*sha* (*stackC*, *Sf*))],
    vs[*Suc 0* := *Intg ia*],
    safeP, *sigSafeMain*, *the* (*cdm* (*l*, *i*)) + *1* + *1* + *1* + *1* + *1* + *1*, *ag*, *be*)]))
apply (*unfold exec-all-def*)
apply (*erule rtrancl-trans*)
prefer *2*
 apply (*rule r-into-rtrancl*)
 apply (*clarify*)
 apply (*unfold JVMExec.exec.simps*)

apply (*unfold Let-def*)
apply (*subgoal-tac*
(*fst* (*snd* (*snd* (*snd* (*snd* (*the* (*method'* (*P'*, *safeP*) *sigSafeMain*)))))) !
(*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*the* (*cdm* (*l*, *i*)))))))) =
Getstatic Sf stackC,*simp*)
apply (*erule nth-via-drop-append*)

apply (*drule drop-Suc-append*)
apply (*subgoal-tac*
P ⊢
(*None*,*sha*,*h*,*inih*,
[([*the* (*sha* (*stackC*, *Sf*))],
vs[*Suc 0* := *Intg ia*],
safeP, *sigSafeMain*, *the* (*cdm* (*l*, *i*)) + *1* + *1* + *1* + *1* + *1* + *1*, *ag*, *be*)])
−*jvm*→
(*None*,*sha*,*h*,*inih*,
[([*the* (*sha* (*stackC*,*topf*)),
the (*sha* (*stackC*, *Sf*))],
vs[*Suc 0* := *Intg ia*],
safeP, *sigSafeMain*, *the* (*cdm* (*l*, *i*)) + *1* + *1* + *1* + *1* + *1* + *1* + *1*, *ag*,
be)]))
apply (*unfold exec-all-def*)
apply (*erule rtrancl-trans*)
prefer *2*
apply (*rule r-into-rtrancl*)
apply (*clarify*)
apply (*unfold JVMExec.exec.simps*)
apply (*unfold Let-def*)
apply (*subgoal-tac*
(*fst* (*snd* (*snd* (*snd* (*snd* (*the* (*method'* (*P'*, *safeP*) *sigSafeMain*)))))) !
(*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*the* (*cdm* (*l*, *i*))))))))) =
Getstatic topf stackC,*simp*)
apply (*erule nth-via-drop-append*)

apply (*drule drop-Suc-append*)
apply (*subgoal-tac*
P ⊢
(*None*,*sha*,*h*,*inih*,
[([*the* (*sha* (*stackC*,*topf*)),
the (*sha* (*stackC*, *Sf*))],
vs[*Suc 0* := *Intg ia*],
safeP, *sigSafeMain*, *the* (*cdm* (*l*, *i*)) + *1* + *1* + *1* + *1* + *1* + *1* + *1*, *ag*, *be*)])
−*jvm*→
(*None*,*sha*,*h*,*inih*,
[([*Intg ia*,
the (*sha* (*stackC*,*topf*)),
the (*sha* (*stackC*, *Sf*))],

```
    vs[Suc 0 := Intg ia],
    safeP, sigSafeMain, the (cdm (l, i)) + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1,
ag, be)]))
apply (unfold exec-all-def)
apply (erule rtrancl-trans)
prefer 2
 apply (rule r-into-rtrancl)
 apply (clarify)
 apply (unfold JVMExec.exec.simps)
 apply (unfold Let-def)
 apply (subgoal-tac
   (fst (snd (snd (snd (snd (the (method' (P', safeP) sigSafeMain)))))) !
    (Suc (Suc (Suc (Suc (Suc (Suc (Suc (the (cdm (l, i))))))))))) =
     Load 1 ,simp)
apply (erule nth-via-drop-append)
```

```
apply (drule drop-Suc-append)
apply (subgoal-tac
  P'⊢
    (None,sha,h,inih,
    [([Intg ia,
       the (sha (stackC,topf)),
       the (sha (stackC, Sf))],
    vs[Suc 0 := Intg ia],
    safeP, sigSafeMain, the (cdm (l, i)) + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1,
ag, be)])
  −jvm→
    (None,sha,
    h(la ↦ Arr ty m (S'a(nat n ↦ Intg ia))),
    inih,
    [([],
    vs[Suc 0 := Intg ia],
    safeP, sigSafeMain, the (cdm (l, i)) + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 +
1, ag, be)]))
apply (unfold exec-all-def)
apply (erule rtrancl-trans)
prefer 2
 apply (rule r-into-rtrancl)
 apply (clarify)
 apply (unfold JVMExec.exec.simps)
 apply (unfold Let-def)
 apply (subgoal-tac
   (fst (snd (snd (snd (snd (the (method' (P', safeP) sigSafeMain)))))) !
    (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (the (cdm (l, i))))))))))) =
     ArrStore,simp)
 apply (simp add: raise-system-xcpt-def)
```

apply (*subgoal-tac* $n \geq 0$,*simp*)
apply (*simp add*: *equivS.simps*)
apply (*erule nth-via-drop-append*)
apply *simp*

apply (*frule nonJumping-Suc-pc*)
apply (*erule-tac sym* [**where** *t*=*REUSE*])
apply (*simp add*: *nonJumping.simps*)
apply (*simp add*: *trInstr.simps*)

apply (*rule-tac x*=*hm*(*b* := *None*)(*getFresh hm* $\mapsto$ *the* (*hm b*)) **in** *exI*)
apply (*rule-tac x*=*k* **in** *exI*)
apply (*rule-tac x*=*k0* **in** *exI*)
apply (*rule-tac x*=(*l*,*Suc i*) **in** *exI*)
apply (*rule-tac x*=*Val* (*Val.Loc* (*getFresh hm*)) # *S'* **in** *exI*)

apply (*rule-tac x*=*sha* **in** *exI*)
apply (*rule-tac x*=*h*(*la* $\mapsto$ *Arr ty m* (*S'a*(*nat n* $\mapsto$ *Intg ia*))) **in** *exI*)
apply (*rule-tac x*=*inih* **in** *exI*)
apply (*rule-tac x*=*vs*[*Suc 0* := *Intg ia*] **in** *exI*)
apply (*rule-tac x*=*the* (*cdm* (*l*, *i*)) + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* **in** *exI*)
apply (*rule-tac x*=*ref* **in** *exI*)

apply (*rule-tac x*=*nat* (*the-Intg* (*the* (*sha* (*heapC*, *kf*)))) **in** *exI*)
apply (*rule-tac x*=*nat* (*the-Intg* (*the* (*sha* (*heapC*, *k0f*)))) **in** *exI*)

apply (*rule-tac x*=*la* **in** *exI*)
apply (*rule-tac x*=*ty* **in** *exI*)
apply (*rule-tac x*=*m* **in** *exI*)
apply (*rule-tac x*=(*S'a*(*nat n* $\mapsto$ *Intg ia*)) **in** *exI*)

apply (*rule-tac x*=*n* **in** *exI*)
apply (*rule-tac x*=*l'* **in** *exI*)
apply (*rule-tac x*=*regS* **in** *exI*)
apply (*rule-tac x*=*l''* **in** *exI*)
apply (*rule-tac x*=*m'* **in** *exI*)
apply (*rule-tac x*=*m''* **in** *exI*)
apply (*rule-tac x*=*d* **in** *exI*)
apply (*rule-tac x*=*g*(*b* := *None*)(*getFresh hm* $\mapsto$ *l'a*) **in** *exI*)

apply (*rule conjI*, *clarsimp*)
apply (*rule conjI*, *rule refl*)
apply (*rule conjI*, *rule refl*)
apply (*rule conjI*, *rule refl*)
apply (*rule conjI*) **apply** *clarsimp*

apply (*rule conjI*, *assumption*)
apply (*rule conjI*) **apply** *clarsimp*
apply (*simp add*: *stackC-def add*: *heapC-def*)
apply (*simp add*: *kf-def add*: *k0f-def*)
apply (*rule conjI*) **apply** (*erule activeCells*)
apply (*rule conjI*) **apply** (*erule activeCells-2*,*assumption*)
apply (*erule activeCells-2*,*assumption*)
apply (*rule conjI*, *simp*)
apply (*rule conjI*) **apply** (*simp add*: *stackC-def add*: *heapC-def*)
apply (*rule conjI*) **apply** *clarsimp*
apply (*rule conjI*) **apply** *clarsimp*
apply (*rule conjI*) **apply** *clarsimp*
apply (*rule conjI*) **apply** *clarsimp*
apply (*rule conjI*) **apply** *clarsimp*
apply (*rule conjI*) **apply** (*subgoal-tac getFresh hm* $\notin$ *dom hm*)
prefer *2* **apply** (*rule getFresh-notin-dom-h*)
apply (*subgoal-tac getFresh hm* $\notin$ *dom g*)
prefer *2* **apply** *simp*
apply (*rule inj-on-REUSE*,*simp*,*assumption*,*simp*,*simp*,*assumption*+)
apply (*rule conjI*) **apply** *clarsimp*
apply (*subgoal-tac getFresh hm* $\notin$ *dom hm*)
prefer *2* **apply** (*rule getFresh-notin-dom-h*)
apply (*subgoal-tac getFresh hm* $\notin$ *dom g*)
prefer *2* **apply** *simp*
apply (*rule equivH-REUSE*,*assumption*+)
apply (*rule conjI*) **apply** *clarsimp*
apply (*rule conjI*) **apply** *clarsimp*
apply (*frule-tac hm=hm* **in** *equivS2-REUSE*,*assumption*+)
apply (*rule conjI*) **apply** *clarsimp*
by *simp*

end

theory *dem-MATCH*
imports *../JVMSAFE/JVMExec SVM2JVM SVMSemantics CertifSVM2JVM*
begin

no-translations *Norm s* == (*None*,*s*)
no-translations *ex-table-of m* == *snd* (*snd* (*snd m*))

declare *trInstr.simps* [*simp del*]
declare *equivS.simps* [*simp del*]
declare *extractBytecode-def* [*simp del*]
declare *initConsTable-def* [*simp del*]

consts

upt-rev :: *nat* => *nat* => *nat list* ((*1*[->/..-']))

primrec
upt-0-rev: [*0*>..*i*] = []
upt-Suc-rev: [(*Suc j*)>..*i*] = (*if i* <= *j then j* # [*j*>..*i*] *else* [])

lemma *length-upt-rev*: *length* [*i*>..*j*] = *i* − *j*
by (*induct i*) (*auto simp add*: *Suc-diff-le*)

lemma *nth-upt-2*: *k* < *j* ∧ *k* >= *i* ==> [*i*..<*j*] ! (*k* − *i*) = *k*
apply (*induct j*)
apply (*auto simp add*: *less-Suc-eq nth-append split*: *nat-diff-split*)
done

constdefs *diff* :: *jvm-state* ⇒ *nat*
diff s ≡ (*case s of* (*none*, *sh*, *h*, *inih*,[([], *vs*, *C*, *M*, *pc*, *ref*)])
⇒ *case vs*!*5 of Intg i* ⇒
case vs!*4 of Intg nargs* ⇒
nat (*nargs* − *i*))

constdefs *oneStep* :: *jvm-prog* ⇒ *jvm-state* ⇒ *jvm-state*
oneStep P s ≡ (*THE s'*. *P* ⊢ *s* −*jvm*→ *s'* ∧
(*case s of* (*none*, *sh*, *h*, *inih*,[([], *vs*, *C*, *M*, *pc*, *ref*)])
⇒ (*case s' of* (*none'*, *sh'*, *h'*, *inih'*,[([], *vs'*, *C'*, *M'*, *pc'*, *ref'*)])
⇒ *case vs*!*5 of Intg i*
⇒ *case sh* (*stackC*, *Sf*) *of Some* (*Addr la*)
⇒ *case h la of Some* (*Arr ty m S'*)
⇒ *case vs*!*6 of Addr l*
⇒ *case vs*!*4 of Intg nargs*
⇒ *case h l of Some* (*Obj cell fs*) ⇒
vs' = *vs*[*5* := *Intg* (*i* + *1*)] ∧
none' = *none* ∧
sh' = *sh* ∧
h' = *h*(*the-Addr* (*the* (*sh* (*stackC*, *Sf*))) ↦ *Arr ty m*
(*S'*(*nat* (*the-Intg* (*the* (*sh* (*stackC*, *topf*))) − *i*)
↦ *argCell fs* (*nat i* + *1*)))) ∧
inih' = *inih* ∧
C' = *C* ∧
M' = *M* ∧
pc' = *pc* ∧
ref' = *ref*)))

consts
steps :: *nat* ⇒ *jvm-prog* ⇒ *jvm-state* ⇒ *jvm-state*
primrec
steps 0 *P s* = *s*

steps (*Suc n*) *P s* = *steps n P* (*oneStep P s*)

constdefs *lastState* :: *jvm-prog* ⇒ *jvm-state* ⇒ *jvm-state*
lastState P s ≡ *steps* (*diff s*) *P s*

axioms *undefined-false*:
undefined ⟹ *False*

axioms *i-0*:
i = *0*

axioms *top-good*:
(¬ *int m* ≤ *the-Intg* (*the* (*sh* (*stackC*, *topf*))) − *i* ∧ ¬ *the-Intg* (*the* (*sh* (*stackC*, *topf*))) < *i*)

lemma *drop-append-3*:
drop n xs = [] @ (*ys* @ *zs*) ==> *drop n xs* = *ys* @ *zs*
by *simp*

lemma *drop-length-append*:
drop n xs = (*ys* @ *zs*) ⟹ *drop* (*n* + *length ys*) *xs* = *zs*
by (*induct n arbitrary*: *xs*) (*auto*, *case-tac xs*, *auto*)

lemma *drop-endlabel1*:
⟦ *drop* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc pc*))))))))))
(*fst* (*snd* (*snd* (*snd* (*snd* (*the* (*method′* (*P′*, *safeP*) *sigSafeMain*)))))))) =
[*Goto endlabel1*] @ *InstLabel2* @ *InstLabel3* @ *InstLabel4* @ *InstLabel5* @ *InstLabel6* @
InstLabel7 @ *InstLabel8* @ *Match2* @ *bytecode* ⟧
⟹ *drop* (*nat* (*12* + *int pc* + *endlabel1*)) (*fst* (*snd* (*snd* (*snd* (*snd* (*the* (*method′* (*P′*, *safeP*) *sigSafeMain*)))))))) =
Match2 @ *bytecode*
apply (*drule drop-Suc-append*)
apply (*drule drop-append-3*)
apply (*drule drop-length-append*)
apply (*drule drop-length-append*)
apply (*drule drop-length-append*)
apply (*drule drop-length-append*)
apply (*drule drop-length-append*)
apply (*drule drop-length-append*)
apply (*drule drop-length-append*)
apply (*simp add*: *endlabel1-def*)
apply (*simp add*: *nlab2-def*)
apply (*simp add*: *nlab3-def*)
apply (*simp add*: *nlab4-def*)
apply (*simp add*: *nlab5-def*)

apply (*simp add*: *nlab6-def*)
apply (*simp add*: *nlab7-def*)
apply (*simp add*: *nlab8-def*)
apply (*simp add*: *InstLabel2-def*)
apply (*simp add*: *InstLabel3-def*)
apply (*simp add*: *InstLabel4-def*)
apply (*simp add*: *InstLabel5-def*)
apply (*simp add*: *InstLabel6-def*)
apply (*simp add*: *InstLabel7-def*)
apply (*simp add*: *InstLabel8-def*)
apply (*subgoal-tac*
(*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc*
(*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc*
(*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc*
(*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc*
(*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc*
(*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc*
pc)) = *nat* (*69* + (*int*
pc)))
by *auto*

lemma *oneStep-MATCH*:
⟦ (*P′*, *cdm*, *ctm*, *com*) = *trSVM2JVM* ((*svms*, *ctmap*), *ini*, *ct*, *ah*, *ai*, *bc*);
length vs = *10*;
vs ! *6* = *Addr l*;
l ≠ *la*;
sh (*stackC*, *Sf*) = *Some* (*Addr la*);
h la = *Some* (*Arr ty m S′*);
h l = *Some* (*Obj cellC fs*);
drop pc (*fst* (*snd* (*snd* (*snd* (*snd* (*the* (*method′* (*P′*, *safeP*) *sigSafeMain*))))))) =
Match12 @ *InstLabel1* @ *InstLabel2* @ *InstLabel3* @ *InstLabel4* @ *InstLabel5* @
InstLabel6 @ *InstLabel7* @ *InstLabel8* @ *Match2* @ *bytecode*;
s = (*None*, *sh*, *h*, *inih*, [([], *vs*[*4*:= *Intg nargs*, *5*:= *Intg i*], *safeP*, *sigSafeMain*, *pc*, *ref*)]);
i < *nargs* ⟧
⟹ *P′* ⊢ (*None*, *sh*, *h*, *inih*, [([], *vs*[*4*:= *Intg nargs*, *5*:= *Intg i*], *safeP*, *sigSafe-Main*, *pc*, *ref*)])
−*jvm*→ (*None*, *sh*, *h*(*the-Addr* (*the* (*sh* (*stackC*, *Sf*))) ↦ *Arr ty m*
(*S′*(*nat* (*the-Intg* (*the* (*sh* (*stackC*, *topf*))) − *i*) ↦ *argCell fs* (*nat i* + *1*)))),
inih, [([], *vs*[*4*:= *Intg nargs*, *5*:= *Intg* (*i* + *1*)], *safeP*, *sigSafeMain*, *pc*, *ref*)])
apply (*unfold Match12-def*)

apply (*subgoal-tac*

P ⊢ (*None*,*sh*,*h*,*inih*,[([], *vs*[4:= *Intg nargs*, 5:= *Intg i*], *safeP*, *sigSafeMain*, *pc*, *ref*)])
−*jvm*→
(*None*,*sh*,*h*,*inih*,[([*Intg nargs*],*vs*[4:= *Intg nargs*, 5:= *Intg i*], *safeP*, *sigSafeMain*, *pc* + 1, *ref*)]))
apply (*unfold exec-all-def*)
apply (*erule rtrancl-trans*)
prefer 2
apply (*rule r-into-rtrancl*)
apply *clarify*
apply (*unfold JVMExec.exec.simps*)
apply (*unfold Let-def*)
apply (*subgoal-tac* (*fst* (*snd* (*snd* (*snd* (*snd* (*the* (*method'* (*P'*, *safeP*) *sigSafeMain*)))))) !
pc) = *Load 4*) **apply** *simp*
apply (*rule nth-via-drop-append*) **apply** *force*

apply (*drule drop-Suc-append*)
apply (*subgoal-tac*
P ⊢ (*None*,*sh*,*h*,*inih*,[([*Intg nargs*], *vs*[4:= *Intg nargs*, 5:= *Intg i*], *safeP*, *sigSafeMain*, *pc* + 1, *ref*)])
−*jvm*→
(*None*, *sh*, *h*, *inih*, [([*Intg i*, *Intg nargs*], *vs*[4:= *Intg nargs*, 5:= *Intg i*], *safeP*, *sigSafeMain*, *pc* + 1 + 1, *ref*)]))
apply (*unfold exec-all-def*)
apply (*erule rtrancl-trans*)
prefer 2
apply (*rule r-into-rtrancl*)
apply *clarify*
apply (*unfold JVMExec.exec.simps*)
apply (*unfold Let-def*)
apply (*subgoal-tac* (*fst* (*snd* (*snd* (*snd* (*snd* (*the* (*method'* (*P'*, *safeP*) *sigSafeMain*)))))) !
Suc pc) = *Load 5*) **apply** *simp*
apply (*rule nth-via-drop-append*) **apply** *force*

apply (*drule drop-Suc-append*)
apply (*subgoal-tac*
P ⊢ (*None*,*sh*,*h*,*inih*,[([*Intg i*, *Intg nargs*], *vs*[4:= *Intg nargs*, 5:= *Intg i*], *safeP*, *sigSafeMain*, *pc* + 1 + 1, *ref*)])
−*jvm*→
(*None*, *sh*, *h*, *inih*, [([], *vs*[4:= *Intg nargs*, 5:= *Intg i*], *safeP*, *sigSafeMain*, *pc* + 1 + 1 + 1, *ref*)]))
apply (*unfold exec-all-def*)
apply (*erule rtrancl-trans*)
prefer 2
apply (*rule r-into-rtrancl*)

apply *clarify*
apply (*unfold JVMExec.exec.simps*)
apply (*unfold Let-def*)
apply (*subgoal-tac* (*fst* (*snd* (*snd* (*snd* (*snd* (*the* (*method'* (*P'*, *safeP*) *sigSafeMain*))))))) !
Suc (*Suc pc*)) = *Ifcmp GreaterEqual labelEndLoop*) **apply** *simp*
apply (*rule nth-via-drop-append*) **apply** *force*

apply (*drule drop-Suc-append*)
apply (*subgoal-tac*
P ⊢ (*None*,*sh*,*h*,*inih*,[([], *vs*[*4* := *Intg nargs*, *5* := *Intg i*], *safeP*, *sigSafeMain*, *pc* + *1* + *1* + *1*, *ref*)])
−*jvm*→
(*None*, *sh*, *h*, *inih*, [([*Intg i*], *vs*[*4* := *Intg nargs*, *5* := *Intg i*], *safeP*, *sigSafeMain*, *pc* + *1* + *1* + *1* + *1*, *ref*)]))
apply (*unfold exec-all-def*)
apply (*erule rtrancl-trans*)
prefer *2*
apply (*rule r-into-rtrancl*)
apply *clarify*
apply (*unfold JVMExec.exec.simps*)
apply (*unfold Let-def*)
apply (*subgoal-tac* (*fst* (*snd* (*snd* (*snd* (*snd* (*the* (*method'* (*P'*, *safeP*) *sigSafeMain*))))))) !
Suc (*Suc* (*Suc pc*))) = *Load 5*) **apply** *simp*
apply (*rule nth-via-drop-append*) **apply** *force*

apply (*drule drop-Suc-append*)
apply (*subgoal-tac*
P ⊢ (*None*,*sh*,*h*,*inih*,[([*Intg i*], *vs*[*4* := *Intg nargs*, *5* := *Intg i*], *safeP*, *sigSafeMain*, *pc* + *1* + *1* + *1* + *1*, *ref*)])
−*jvm*→
(*None*, *sh*, *h*, *inih*, [([], *vs*[*4* := *Intg nargs*, *5* := *Intg i*], *safeP*, *sigSafeMain*, *pc* + *1* + *1* + *1* + *1* + *1*, *ref*)]))
apply (*unfold exec-all-def*)
apply (*erule rtrancl-trans*)
prefer *2*
apply (*rule r-into-rtrancl*)
apply *clarify*
apply (*unfold JVMExec.exec.simps*)
apply (*unfold Let-def*)
apply (*subgoal-tac* (*fst* (*snd* (*snd* (*snd* (*snd* (*the* (*method'* (*P'*, *safeP*) *sigSafeMain*))))))) !
Suc (*Suc* (*Suc* (*Suc pc*)))) =
Tableswitch 0 7 [*label1*, *label2*, *label3*, *label4*, *label5*, *label6*, *label7*, *label8*]) **apply** *simp*

apply (*subgoal-tac i = 0*)
prefer *2* **apply** (*rule i-0*) **apply** *simp* **apply** (*simp add*: *label1-def*)
apply (*rule nth-via-drop-append*) **apply** *force*

apply (*drule drop-Suc-append*)
apply (*unfold InstLabel1-def*)
apply (*drule drop-append-3*)
apply (*subgoal-tac*
  P ⊢ (*None*,*sh*,*h*,*inih*,[([], *vs*[*4*:= *Intg nargs*, *5*:= *Intg i*], *safeP*, *sigSafeMain*, *pc* + *1* + *1* + *1* + *1* + *1*, *ref*)])
  −*jvm*→
    (*None*, *sh*, *h*, *inih*, [([*the* (*sh* (*stackC*, *Sf*))],
    vs[*4*:= *Intg nargs*, *5*:= *Intg i*], *safeP*, *sigSafeMain*, *Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc pc*))))), *ref*)]))
apply (*unfold exec-all-def*)
apply (*erule rtrancl-trans*)
prefer *2*
 apply (*rule r-into-rtrancl*)
 apply *clarify*
 apply (*unfold JVMExec.exec.simps*)
 apply (*unfold Let-def*)
 apply (*subgoal-tac* (*fst* (*snd* (*snd* (*snd* (*snd* (*the* (*method′* (*P′*, *safeP*) *sigSafeMain*))))))) !
                  Suc (*Suc* (*Suc* (*Suc* (*Suc pc*)))) =
                  Getstatic Sf stackC) **apply** *simp*
apply (*rule nth-via-drop-append*) **apply** *force*

apply (*drule drop-Suc-append*)
apply (*subgoal-tac*
  P ⊢ (*None*, *sh*, *h*, *inih*,
    [([*the* (*sh* (*stackC*, *Sf*))],
    vs[*4*:= *Intg nargs*, *5*:= *Intg i*], *safeP*, *sigSafeMain*,
    Suc (*Suc* (*Suc* (*Suc* (*Suc* (*Suc pc*))))), *ref*)])
  −*jvm*→
    (*None*, *sh*, *h*, *inih*,
    [([*the* (*sh* (*stackC*, *topf*)), *the* (*sh* (*stackC*, *Sf*))],
    vs[*4*:= *Intg nargs*, *5*:= *Intg i*], *safeP*, *sigSafeMain*,
    Suc (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc pc*)))))), *ref*)]))
apply (*unfold exec-all-def*)
apply (*erule rtrancl-trans*)
prefer *2*
 apply (*rule r-into-rtrancl*)
 apply *clarify*
 apply (*unfold JVMExec.exec.simps*)
 apply (*unfold Let-def*)
 apply (*subgoal-tac* (*fst* (*snd* (*snd* (*snd* (*snd* (*the* (*method′* (*P′*, *safeP*) *sigSafeMain*))))))) !

Suc (Suc (Suc (Suc (Suc (Suc pc)))))) =
Getstatic topf stackC) **apply** *simp*
apply (*rule nth-via-drop-append*) **apply** *force*

apply (*drule drop-Suc-append*)
apply (*subgoal-tac*
P ⊢ (*None*, *sh*, *h*, *inih*,
[([*the* (*sh* (*stackC*, *topf*)), *the* (*sh* (*stackC*, *Sf*))],
vs[4 := *Intg nargs*, 5 := *Intg i*], *safeP*, *sigSafeMain*,
Suc (*Suc* (*Suc* (*Suc* (*Suc* (*Suc pc*))))), *ref*)])
−*jvm*→
(*None*, *sh*, *h*, *inih*,
[([*Intg i*, *the* (*sh* (*stackC*, *topf*)), *the* (*sh* (*stackC*, *Sf*))],
vs[4 := *Intg nargs*, 5 := *Intg i*], *safeP*, *sigSafeMain*,
Suc (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc pc*)))))), *ref*)]))
apply (*unfold exec-all-def*)
apply (*erule rtrancl-trans*)
prefer 2
apply (*rule r-into-rtrancl*)
apply *clarify*
apply (*unfold JVMExec.exec.simps*)
apply (*unfold Let-def*)
apply (*subgoal-tac* (*fst* (*snd* (*snd* (*snd* (*the* (*method′* (*P′*, *safeP*) *sigSafeMain*))))))) !
Suc (*Suc* (*Suc* (*Suc* (*Suc* (*Suc pc*)))))) =
Load 5) **apply** *simp*
apply (*rule nth-via-drop-append*) **apply** *force*

apply (*drule drop-Suc-append*)
apply (*subgoal-tac*
P ⊢ (*None*, *sh*, *h*, *inih*,
[([*Intg i*, *the* (*sh* (*stackC*, *topf*)), *the* (*sh* (*stackC*, *Sf*))],
vs[4 := *Intg nargs*, 5 := *Intg i*], *safeP*, *sigSafeMain*,
Suc (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc pc*)))))), *ref*)])
−*jvm*→
(*None*, *sh*, *h*, *inih*,
[([*Intg* (*the-Intg* (*the* (*sh* (*stackC*, *topf*))) − *i*), *the* (*sh* (*stackC*, *Sf*))],
vs[4 := *Intg nargs*, 5 := *Intg i*], *safeP*, *sigSafeMain*,
Suc (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc pc*))))))), *ref*)]))
apply (*unfold exec-all-def*)
apply (*erule rtrancl-trans*)
prefer 2
apply (*rule r-into-rtrancl*)
apply *clarify*
apply (*unfold JVMExec.exec.simps*)
apply (*unfold Let-def*)

apply (*subgoal-tac* (*fst* (*snd* (*snd* (*snd* (*snd* (*the* (*method'* (*P'*, *safeP*) *sigSafe-Main*))))))) !
Suc (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc pc*)))))))) =
BinOp Substract) **apply** *simp*
apply (*rule nth-via-drop-append*) **apply** *force*

apply (*drule drop-Suc-append*)
apply (*subgoal-tac*
P' ⊢ (*None*, *sh*, *h*, *inih*,
[([*Intg* (*the-Intg* (*the* (*sh* (*stackC*, *topf*))) − *i*), *the* (*sh* (*stackC*, *Sf*))],
vs[*4* := *Intg nargs*, *5* := *Intg i*], *safeP*, *sigSafeMain*,
Suc (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc pc*)))))))), *ref*)])
−*jvm*→
(*None*, *sh*, *h*, *inih*,
[([*Addr l*, *Intg* (*the-Intg* (*the* (*sh* (*stackC*, *topf*))) − *i*), *the* (*sh* (*stackC*, *Sf*))],
vs[*4* := *Intg nargs*, *5* := *Intg i*], *safeP*, *sigSafeMain*,
Suc (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc pc*)))))))), *ref*)]))
apply (*unfold exec-all-def*)
apply (*erule rtrancl-trans*)
prefer *2*
apply (*rule r-into-rtrancl*)
apply *clarify*
apply (*unfold JVMExec.exec.simps*)
apply (*unfold Let-def*)
apply (*subgoal-tac* (*fst* (*snd* (*snd* (*snd* (*snd* (*the* (*method'* (*P'*, *safeP*) *sigSafe-Main*))))))) !
Suc (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc pc*)))))))) =
Load 6) **apply** *simp*
apply (*rule nth-via-drop-append*) **apply** *force*

apply (*drule drop-Suc-append*)
apply (*subgoal-tac*
P' ⊢ (*None*, *sh*, *h*, *inih*,
[([*Addr l*, *Intg* (*the-Intg* (*the* (*sh* (*stackC*, *topf*))) − *i*), *the* (*sh* (*stackC*, *Sf*))],
vs[*4* := *Intg nargs*, *5* := *Intg i*], *safeP*, *sigSafeMain*,
Suc (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc pc*)))))))), *ref*)])
−*jvm*→
(*None*, *sh*, *h*, *inih*,
[([*the* (*fs* (*arg1*, *cellC*)), *Intg* (*the-Intg* (*the* (*sh* (*stackC*, *topf*))) − *i*), *the* (*sh* (*stackC*, *Sf*))],
vs[*4* := *Intg nargs*, *5* := *Intg i*], *safeP*, *sigSafeMain*,
Suc (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc pc*)))))))))), *ref*)]))
apply (*unfold exec-all-def*)

apply (*erule rtrancl-trans*)
prefer *2*
apply (*rule r-into-rtrancl*)
apply *clarify*
apply (*unfold JVMExec.exec.simps*)
apply (*unfold Let-def*)
apply (*subgoal-tac* (*fst* (*snd* (*snd* (*snd* (*snd* (*the* (*method'* (*P'*, *safeP*) *sigSafe-Main*))))))) !
Suc (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc pc*)))))))))
=
Getfield arg1 cellC) **apply** *simp*
apply (*simp add*: *raise-system-xcpt-def*)
apply (*rule nth-via-drop-append*) **apply** *force*

apply (*drule drop-Suc-append*)
apply (*subgoal-tac*
P ⊢ (*None*, *sh*, *h*, *inih*,
[([*the* (*fs* (*arg1*, *cellC*)), *Intg* (*the-Intg* (*the* (*sh* (*stackC*, *topf*))) − *i*), *the* (*sh* (*stackC*, *Sf*))],
vs[*4*:= *Intg nargs*, *5*:= *Intg i*], *safeP*, *sigSafeMain*,
Suc (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc pc*)))))))))), *ref*)])
−*jvm*→
(*None*, *sh*,
h(*the-Addr* (*the* (*sh* (*stackC*, *Sf*))) ↦ *Arr ty m* (*S'*(*nat* (*the-Intg* (*the* (*sh* (*stackC*, *topf*))) − *i*)
↦ *argCell fs* (*nat i* + *1*)))),
inih,
[([], *vs*[*4*:= *Intg nargs*, *5*:= *Intg i*], *safeP*, *sigSafeMain*,
Suc (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc pc*)))))))))), *ref*)]))
apply (*unfold exec-all-def*)
apply (*erule rtrancl-trans*)
prefer *2*
apply (*rule r-into-rtrancl*)
apply *clarify*
apply (*unfold JVMExec.exec.simps*)
apply (*unfold Let-def*)
apply (*subgoal-tac* (*fst* (*snd* (*snd* (*snd* (*snd* (*the* (*method'* (*P'*, *safeP*) *sigSafe-Main*))))))) !
Suc (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc pc*)))))))))) =
ArrStore) **apply** *simp*
apply (*subgoal-tac* (¬ *int m* ≤ *the-Intg* (*the* (*sh* (*stackC*, *topf*))) − *i* ∧ ¬ *the-Intg* (*the* (*sh* (*stackC*, *topf*))) < *i*))
apply (*simp add*: *raise-system-xcpt-def*)
apply (*subgoal-tac i=0*)
prefer *2* **apply** (*rule i-0*)
apply (*subgoal-tac the* (*fs* (*arg1*, *cellC*)) = *argCell fs* (*Suc* (*nat i*)),*simp*)

apply (*simp add*: *argCell-def*)
apply (*rule top-good*)
apply (*rule nth-via-drop-append*) **apply** *force*

apply (*drule drop-Suc-append*)
apply (*subgoal-tac*
  P⊢ (*None*, *sh*,
       h(*the-Addr* (*the* (*sh* (*stackC*, *Sf*))) ↦ *Arr ty m* (*S'*(*nat* (*the-Intg* (*the* (*sh* (*stackC*, *topf*))) − *i*)
                                        ↦ *argCell fs* (*nat i* + *1*)))),
       inih,
       [([], *vs*[*4*:= *Intg nargs*, *5*:= *Intg i*], *safeP*, *sigSafeMain*,
       Suc (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc pc*))))))))))),
ref)])
  −*jvm*→
     (*None*, *sh*,
       h(*the-Addr* (*the* (*sh* (*stackC*, *Sf*))) ↦ *Arr ty m* (*S'*(*nat* (*the-Intg* (*the* (*sh* (*stackC*, *topf*))) − *i*)
                                        ↦ *argCell fs* (*nat i* + *1*)))),
       inih,
       [([], *vs*[*4*:= *Intg nargs*, *5*:= *Intg i*], *safeP*, *sigSafeMain*, *nat* (*12* + *int pc* + *endlabel1*), *ref*)]))
apply (*unfold exec-all-def*)
apply (*erule rtrancl-trans*)
prefer *2*
 apply (*rule r-into-rtrancl*)
 apply *clarify*
 apply (*unfold JVMExec.exec.simps*)
 apply (*unfold Let-def*)
 apply (*subgoal-tac* (*fst* (*snd* (*snd* (*snd* (*snd* (*the* (*method'* (*P'*, *safeP*) *sigSafeMain*))))))) !
                  Suc (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc pc*)))))))))) =
                  Goto endlabel1) **apply** *simp*
apply (*rule nth-via-drop-append*) **apply** *force*

apply (*frule drop-endlabel1*)
apply (*unfold Match2-def*)
apply (*subgoal-tac*
  P⊢ (*None*, *sh*,
       h(*the-Addr* (*the* (*sh* (*stackC*, *Sf*))) ↦ *Arr ty m* (*S'*(*nat* (*the-Intg* (*the* (*sh* (*stackC*, *topf*))) − *i*)
                                        ↦ *argCell fs* (*nat i* + *1*)))),
       inih,
       [([], *vs*[*4*:= *Intg nargs*, *5*:= *Intg i*], *safeP*, *sigSafeMain*, *nat* (*12* + *int pc* + *endlabel1*), *ref*)])
  −*jvm*→

```
      (None, sh,
       h(the-Addr (the (sh (stackC, Sf))) ↦ Arr ty m (S'(nat (the-Intg (the (sh
(stackC, topf))) − i)
                                        ↦ argCell fs (nat i + 1)))),
       inih,
       [([Intg i],
       vs[4:= Intg nargs, 5:= Intg i], safeP, sigSafeMain,
       Suc (nat (12 + int pc + endlabel1)), ref)]))
apply (unfold exec-all-def)
apply (erule rtrancl-trans)
prefer 2
 apply (rule r-into-rtrancl)
 apply clarify
 apply (unfold JVMExec.exec.simps)
 apply (unfold Let-def)
 apply (subgoal-tac (fst (snd (snd (snd (snd (the (method' (P', safeP) sigSafe-
Main)))))) !
                    (nat (12 + int pc + endlabel1))) =
                    Load 5) apply simp
apply (rule nth-via-drop-append) apply force


apply (drule drop-Suc-append)
apply (drule drop-Suc-append)
apply (subgoal-tac
  P'⊢ (None, sh,
       h(the-Addr (the (sh (stackC, Sf))) ↦ Arr ty m (S'(nat (the-Intg (the (sh
(stackC, topf))) − i)
                                        ↦ argCell fs (nat i + 1)))),
       inih,
       [([Intg i],
       vs[4:= Intg nargs, 5:= Intg i], safeP, sigSafeMain,
       Suc (nat (12 + int pc + endlabel1)), ref)])
  −jvm→
      (None, sh,
       h(the-Addr (the (sh (stackC, Sf))) ↦ Arr ty m (S'(nat (the-Intg (the (sh
(stackC, topf))) − i)
                                        ↦ argCell fs (nat i + 1)))),
       inih,
       [([Intg 1, Intg i],
       vs[4:= Intg nargs, 5:= Intg i], safeP, sigSafeMain,
       Suc (Suc (nat (12 + int pc + endlabel1))), ref)]))
apply (unfold exec-all-def)
apply (erule rtrancl-trans)
prefer 2
 apply (rule r-into-rtrancl)
 apply clarify
 apply (unfold JVMExec.exec.simps)
 apply (unfold Let-def)
```

apply (*subgoal-tac* (*fst* (*snd* (*snd* (*snd* (*snd* (*the* (*method'* (*P'*, *safeP*) *sigSafe-Main*))))))) !
Suc (*nat* (*12* + *int pc* + *endlabel1*))) =
LitPush (*Intg 1*)) **apply** *simp*
apply (*rule nth-via-drop-append*) **apply** *force*

apply (*drule drop-Suc-append*)
apply (*subgoal-tac*
P'⊢ (*None*, *sh*,
h(*the-Addr* (*the* (*sh* (*stackC*, *Sf*))) ↦ *Arr ty m* (*S'*(*nat* (*the-Intg* (*the* (*sh* (*stackC*, *topf*))) − *i*)
↦ *argCell fs* (*nat i* + *1*)))),
inih,
[([*Intg 1*, *Intg i*],
vs[*4*:= *Intg nargs*, *5*:= *Intg i*], *safeP*, *sigSafeMain*,
Suc (*Suc* (*nat* (*12* + *int pc* + *endlabel1*))), *ref*)])
−*jvm*→
(*None*, *sh*,
h(*the-Addr* (*the* (*sh* (*stackC*, *Sf*))) ↦ *Arr ty m* (*S'*(*nat* (*the-Intg* (*the* (*sh* (*stackC*, *topf*))) − *i*)
↦ *argCell fs* (*nat i* + *1*)))),
inih,
[([*Intg* (*i* + *1*)],
vs[*4*:= *Intg nargs*, *5*:= *Intg i*], *safeP*, *sigSafeMain*,
Suc (*Suc* (*Suc* (*nat* (*12* + *int pc* + *endlabel1*)))), *ref*)]))
apply (*unfold exec-all-def*)
apply (*erule rtrancl-trans*)
prefer *2*
apply (*rule r-into-rtrancl*)
apply *clarify*
apply (*unfold JVMExec.exec.simps*)
apply (*unfold Let-def*)
apply (*subgoal-tac* (*fst* (*snd* (*snd* (*snd* (*snd* (*the* (*method'* (*P'*, *safeP*) *sigSafe-Main*))))))) !
Suc (*Suc* (*nat* (*12* + *int pc* + *endlabel1*)))) =
BinOp Add) **apply** *simp*
apply (*rule nth-via-drop-append*) **apply** *force*

apply (*drule drop-Suc-append*)
apply (*subgoal-tac*
P'⊢ (*None*, *sh*,
h(*the-Addr* (*the* (*sh* (*stackC*, *Sf*))) ↦ *Arr ty m* (*S'*(*nat* (*the-Intg* (*the* (*sh* (*stackC*, *topf*))) − *i*)
↦ *argCell fs* (*nat i* + *1*)))),
inih,
[([*Intg* (*i* + *1*)],

```
      vs[4:= Intg nargs, 5:= Intg i], safeP, sigSafeMain,
      Suc (Suc (Suc (nat (12 + int pc + endlabel1)))), ref)])
  −jvm→
      (None, sh, h(the-Addr (the (sh (stackC, Sf))) ↦ Arr ty m (S'(nat (the-Intg
(the (sh (stackC, topf))) − i)
                                                  ↦ argCell fs (nat i + 1)))),
      inih,
      [([],
      vs[4:= Intg nargs, 5:= Intg (i + 1)], safeP, sigSafeMain,
      Suc (Suc (Suc (Suc (nat (12 + int pc + endlabel1))))), ref)]))
apply (unfold exec-all-def)
apply (erule rtrancl-trans)
prefer 2
 apply (rule r-into-rtrancl)
 apply clarify
 apply (unfold JVMExec.exec.simps)
 apply (unfold Let-def)
 apply (subgoal-tac (fst (snd (snd (snd (snd (the (method' (P', safeP) sigSafe-
Main)))))) !
                    Suc (Suc (Suc (nat (12 + int pc + endlabel1))))) =
                    Store 5) apply simp
apply (rule nth-via-drop-append) apply force
```

```
apply (drule drop-Suc-append)
apply (subgoal-tac
  P'⊢ (None, sh, h(the-Addr (the (sh (stackC, Sf))) ↦ Arr ty m (S'(nat (the-Intg
(the (sh (stackC, topf))) − i)
                                                  ↦ argCell fs (nat i + 1)))),
      inih,
      [([],
      vs[4:= Intg nargs, 5:= Intg (i + 1)], safeP, sigSafeMain,
      Suc (Suc (Suc (Suc (nat (12 + int pc + endlabel1))))), ref)])
  −jvm→
      (None, sh,
       h(the-Addr (the (sh (stackC, Sf))) ↦  Arr ty m (S'(nat (the-Intg (the (sh
(stackC, topf))) − i)
                                        ↦ argCell fs (nat i + 1)))),
      inih,
      [([],
      vs[4:= Intg nargs, 5:= Intg (i + 1)],
      safeP, sigSafeMain,
      nat (16 + (int pc + endlabel1) + labelLoop), ref)]))
apply (unfold exec-all-def)
apply (erule rtrancl-trans)
prefer 2
 apply (rule r-into-rtrancl)
```

apply *clarify*
apply (*unfold JVMExec.exec.simps*)
apply (*unfold Let-def*)
apply (*subgoal-tac* (*fst* (*snd* (*snd* (*snd* (*snd* (*the* (*method'* (*P'*, *safeP*) *sigSafe-Main*))))))) !
Suc (*Suc* (*Suc* (*Suc* (*nat* (*12* + *int pc* + *endlabel1*))))) =
Goto labelLoop) **apply** *simp*
apply (*simp add*: *endlabel1-def*)
apply (*rule nth-via-drop-append*) **apply** *force*
apply (*simp add*: *endlabel1-def*)
apply (*simp add*: *labelLoop-def*)
apply (*simp add*: *nlab1-def*)
apply (*simp add*: *InstLabel1-def*)
apply (*simp add*: *nMatch2-def*)
apply (*simp add*: *nMatch12-def*)
by (*simp add*: *Match12-def*)

lemma *oneStep-state*:
⟦ (*P'*, *cdm*, *ctm*, *com*) = *trSVM2JVM* ((*svms*, *ctmap*), *ini*, *ct*, *ah*, *ai*, *bc*);
length vs = *10*;
vs ! *6* = *Addr l*;
sh (*stackC*, *Sf*) = *Some* (*Addr la*);
l ≠ *la*;
h la = *Some* (*Arr ty m S'*);
h l = *Some* (*Obj cellC fs*);
drop pc (*fst* (*snd* (*snd* (*snd* (*snd* (*the* (*method'* (*P'*, *safeP*) *sigSafeMain*)))))))
=
Match12 @ *InstLabel1* @ *InstLabel2* @ *InstLabel3* @ *InstLabel4* @ *InstLabel5* @
InstLabel6 @ *InstLabel7* @ *InstLabel8* @ *Match2* @ *bytecode*;
i < *nargs* ⟧
⟹ *oneStep P'* (*None*, *sh*, *h*, *inih*, [([], *vs*[*4* := *Intg nargs*, *5* := *Intg i*], *safeP*, *sigSafeMain*, *pc*, *ref*)])
= (*None*, *sh*, *h*(*the-Addr* (*the* (*sh* (*stackC*, *Sf*))) ↦
Arr ty m (*S'*(*nat* (*the-Intg* (*the* (*sh* (*stackC*, *topf*))) − *i*) ↦ *argCell fs* (*nat i* + *1*)))),
inih, [([], *vs*[*4* := *Intg nargs*, *5* := *Intg* (*i* + *1*)], *safeP*, *sigSafeMain*, *pc*, *ref*)])
apply (*simp* (*no-asm*) *only*: *oneStep-def*)
apply (*rule the-equality*)
apply (*rule conjI*)
apply (*rule oneStep-MATCH*,*assumption*+,*simp*)
apply *auto*
apply (*rename-tac v1' sh' dh' ih' frms'*)
apply (*case-tac frms'*,*simp*)
apply (*frule undefined-false*,*simp*)
apply *simp*
apply (*case-tac a*,*simp*)

apply (*case-tac aa*,*simp*)
apply (*case-tac b*,*simp*)
apply (*case-tac ba*,*simp*)
apply (*case-tac bb*,*simp*)
apply (*case-tac bca*,*simp*)
apply (*case-tac list*,*simp*)
apply (*simp*,*frule undefined-false*,*simp*)
apply (*simp*,*frule undefined-false*,*simp*)
apply (*case-tac y*,*simp*)
apply (*frule undefined-false*,*simp*)
apply *simp*
apply (*case-tac a*,*simp*)
apply (*case-tac aa*,*simp*)
apply (*case-tac b*,*simp*)
apply (*case-tac ba*,*simp*)
apply (*case-tac bb*,*simp*)
apply (*case-tac bca*,*simp*)
apply (*case-tac list*,*simp*)
apply (*simp*,*frule undefined-false*,*simp*)
apply (*simp*,*frule undefined-false*,*simp*)
apply (*case-tac y*,*simp*)
apply (*frule undefined-false*,*simp*)
apply *simp*
apply (*case-tac a*,*simp*)
apply (*case-tac aa*,*simp*)
apply (*case-tac b*,*simp*)
apply (*case-tac ba*,*simp*)
apply (*case-tac bb*,*simp*)
apply (*case-tac bca*,*simp*)
apply (*case-tac list*,*simp*)
apply (*simp*,*frule undefined-false*,*simp*)
apply (*simp*,*frule undefined-false*,*simp*)
apply (*rename-tac v1' sh' dh' ih' frms'*)
apply (*case-tac frms'*)
apply *simp*
apply (*frule undefined-false*,*simp*)
apply *simp*
apply (*case-tac a*,*simp*)
apply (*case-tac aa*,*simp*)
apply (*case-tac b*,*simp*)
apply (*case-tac ba*,*simp*)
apply (*case-tac bb*,*simp*)
apply (*case-tac bca*,*simp*)
apply (*case-tac list*,*simp*)
apply (*simp*,*frule undefined-false*,*simp*)
apply (*simp*,*frule undefined-false*,*simp*)
apply (*case-tac y*)
apply *simp*
apply (*frule undefined-false*,*simp*)

apply *simp*
apply (*case-tac a*,*simp*)
apply (*case-tac aa*,*simp*)
apply (*case-tac b*,*simp*)
apply (*case-tac ba*,*simp*)
apply (*case-tac bb*,*simp*)
apply (*case-tac bca*,*simp*)
apply (*case-tac list*,*simp*)
apply (*simp*,*frule undefined-false*,*simp*)
by (*simp*,*frule undefined-false*,*simp*)

lemma *diff-monotone*:
⟦ (*P'*, *cdm*, *ctm*, *com*) = *trSVM2JVM* ((*svms*, *ctmap*), *ini*, *ct*, *ah*, *ai*, *bc*);
length vs = *10*;
vs ! *6* = *Addr l*;
sh (*stackC*, *Sf*) = *Some* (*Addr la*);
h la = *Some* (*Arr ty m S'*);
h l = *Some* (*Obj cellC fs*);
l ≠ *la*;
i < *nargs*;
drop pc (*fst* (*snd* (*snd* (*snd* (*snd* (*the* (*method'* (*P'*, *safeP*) *sigSafeMain*)))))))
=
Match12 @ *InstLabel1* @ *InstLabel2* @ *InstLabel3* @ *InstLabel4* @ *InstLabel5* @
InstLabel6 @ *InstLabel7* @ *InstLabel8* @ *Match2* @ *bytecode*;
Suc n = *diff* (*None*, *sh*, *h*, *inih*, [([], *vs*[*4*:= *Intg nargs*, *5* := *Intg i*], *safeP*, *sigSafeMain*, *pc*, *ref*)]) ⟧
⟹ *n* = *diff* (*oneStep P'* (*None*, *sh*, *h*, *inih*, [([], *vs*[*4*:= *Intg nargs*, *5* := *Intg i*], *safeP*, *sigSafeMain*, *pc*, *ref*)]))
apply (*simp add*: *diff-def*)
apply (*subgoal-tac*
oneStep P' (*None*, *sh*, *h*, *inih*, [([], *vs*[*4*:= *Intg nargs*, *5* := *Intg i*], *safeP*, *sigSafeMain*, *pc*, *ref*)]) =
(*None*, *sh*, *h*(*the-Addr* (*the* (*sh* (*stackC*, *Sf*))) ↦
Arr ty m (*S'*(*nat* (*the-Intg* (*the* (*sh* (*stackC*, *topf*))) − *i*) ↦ *argCell fs* (*nat i* + *1*)))),
inih, [([], *vs*[*4*:= *Intg nargs*, *5*:= *Intg* (*i* + *1*)], *safeP*, *sigSafeMain*, *pc*, *ref*)]),*simp*)
by (*rule oneStep-state*,*assumption*+)

lemma *steps-monotone*:
⟦ (*P'*, *cdm*, *ctm*, *com*) = *trSVM2JVM* ((*svms*, *ctmap*), *ini*, *ct*, *ah*, *ai*, *bc*);
length vs = *10*;
vs ! *6* = *Addr l*;
sh (*stackC*, *Sf*) = *Some* (*Addr la*);
h la = *Some* (*Arr ty m S'*);
h l = *Some* (*Obj cellC fs*);
l ≠ *la*;

$i < nargs$;
$drop\ pc\ (fst\ (snd\ (snd\ (snd\ (snd\ (the\ (method'\ (P', safeP)\ sigSafeMain)))))))$
=
$Match12\ @\ InstLabel1\ @\ InstLabel2\ @\ InstLabel3\ @\ InstLabel4\ @\ InstLabel5$
@
$InstLabel6\ @\ InstLabel7\ @\ InstLabel8\ @\ Match2\ @\ bytecode$;
$Suc\ n = diff\ (None, sh, h, inih, [([], vs[4 := Intg\ nargs, 5 := Intg\ i], safeP, sigSafeMain, pc, ref)])\ ]\!]$
$\Longrightarrow steps\ (diff\ (None, sh, h, inih, [([], vs[4 := Intg\ nargs, 5 := Intg\ i], safeP, sigSafeMain, pc, ref)]))\ P'$
$(None, sh, h, inih, [([], vs[4 := Intg\ nargs, 5 := Intg\ i], safeP, sigSafeMain, pc, ref)]) =$
$steps\ (diff\ (oneStep\ P'\ (None, sh, h, inih, [([], vs[4 := Intg\ nargs, 5 := Intg\ i], safeP, sigSafeMain, pc, ref)])))\ P'$
$(oneStep\ P'\ (None, sh, h, inih, [([], vs[4 := Intg\ nargs, 5 := Intg\ i], safeP, sigSafeMain, pc, ref)]))$
apply $(frule\ diff\text{-}monotone, assumption+)$
by $(case\text{-}tac\ (diff\ (None, sh, h, inih, [([], vs[4 := Intg\ nargs, 5 := Intg\ i], safeP, sigSafeMain, pc, ref)])), simp, simp)$

lemma *MATCH-loop* [*rule-format*]:
$\forall\ sh\ h\ inih\ vs\ i\ pc\ ref\ S'.$
$n = diff\ (None, sh, h, inih, [([], vs[4 := Intg\ nargs, 5 := Intg\ i], safeP, sigSafeMain, pc, ref)])$
$\longrightarrow (P', cdm, ctm, com) = trSVM2JVM\ ((svms, ctmap), ini, ct, ah, ai, bc)$
$\longrightarrow length\ vs = 10$
$\longrightarrow vs\ !\ 6 = Addr\ l$
$\longrightarrow sh\ (stackC, Sf) = Some\ (Addr\ la)$
$\longrightarrow h\ la = Some\ (Arr\ ty\ m\ S')$
$\longrightarrow h\ l = Some\ (Obj\ cellC\ fs)$
$\longrightarrow l \neq la$
$\longrightarrow drop\ pc\ (fst\ (snd\ (snd\ (snd\ (snd\ (the\ (method'\ (P', safeP)\ sigSafeMain)))))))) =$
$Match12\ @\ InstLabel1\ @\ InstLabel2\ @\ InstLabel3\ @\ InstLabel4\ @\ InstLabel5$
@
$InstLabel6\ @\ InstLabel7\ @\ InstLabel8\ @\ Match2\ @\ bytecode$
$\longrightarrow P' \vdash (None, sh, h, inih, [([], vs[4 := Intg\ nargs, 5 := Intg\ i], safeP, sigSafeMain, pc, ref)])$
$-jvm\rightarrow steps\ n\ P'\ (None, sh, h, inih, [([], vs[4 := Intg\ nargs, 5 := Intg\ i], safeP, sigSafeMain, pc, ref)])$
apply $(induct\ n)$
apply $(simp\ add: steps.simps)$
apply $(simp\ add: exec\text{-}all\text{-}def)$
apply $(rule\ allI)+$
apply $(rule\ impI)$

apply $(case\text{-}tac\ i < nargs)$

apply (*erule-tac x=sh* **in** *allE*)
apply (*erule-tac x=h(the-Addr (the (sh (stackC, Sf))) ↦*
Arr ty m (S'(nat (the-Intg (the (sh (stackC, topf))) − i) ↦
*argCell fs (nat i + 1)))) **in** allE*)
apply (*erule-tac x=inih* **in** *allE*)
apply (*erule-tac x=vs* **in** *allE*)
apply (*erule-tac x=i + 1* **in** *allE*)
apply (*erule-tac x=pc* **in** *allE*)
apply (*erule-tac x=ref* **in** *allE*)
apply (*erule-tac x=S'(nat (the-Intg (the (sh (stackC, topf))) − i) ↦ argCell fs*
(nat i + 1)) **in** *allE*)
apply (*rule impI*)+
apply (*drule mp*)
apply (*frule diff-monotone,assumption+*)
apply (*frule-tac inih=inih* **and** *nargs=nargs* **and** *i=i* **and** *ref=ref* **in** *oneStep-state,assumption+*)

apply (*simp add*: *diff-def*)
apply (*drule mp,force*)+
apply (*frule diff-monotone,assumption+*)
apply (*frule-tac*
inih=inih **and** *nargs=nargs* **and** *i=i* **and** *ref=ref*
in *oneStep-MATCH,assumption+,simp,assumption*)
apply (*frule-tac*
inih=inih **and** *nargs=nargs* **and** *i=i* **and** *ref=ref*
in *oneStep-state,assumption+*)
apply (*drule-tac*
s=oneStep P' (None, sh, h, inih, [([], vs[4 := Intg nargs, 5 := Intg i],
*safeP, sigSafeMain, pc, ref)]) **in** sym*)
apply *clarify*
apply *simp*
apply (*frule steps-monotone, assumption+*)
apply (*drule-tac*
s=steps (diff (None, sh, h, inih, [([], vs[4 := Intg nargs, 5 := Intg i], safeP,
sigSafeMain, pc, a, b)])) P'
(None, sh, h, inih, [([], vs[4 := Intg nargs, 5 := Intg i], safeP,
*sigSafeMain, pc, a, b)]) **in** sym,simp*)
apply (*simp add*: *exec-all-def*)

apply (*rule impI*)+
by (*simp add*: *diff-def*)

lemma *execSVMInstr-MATCH-loop*:
⟦ (*P'*, *cdm*, *ctm*, *com*) = *trSVM2JVM* ((*svms*, *ctmap*), *ini*, *ct*, *ah*, *ai*, *bc*);
length vs = 10;
vs ! 6 = Addr l;
sh (*stackC*, *Sf*) = *Some* (*Addr la*);
h la = Some (*Arr ty m S'*);

$h\ l\ =\ Some\ (Obj\ cellC\ fs)$;
$l \neq la$;
$drop\ pc\ (fst\ (snd\ (snd\ (snd\ (snd\ (the\ (method'\ (P',\ safeP)\ sigSafeMain)))))))$
=
$Match12\ @\ InstLabel1\ @\ InstLabel2\ @\ InstLabel3\ @\ InstLabel4\ @\ InstLabel5$
@
$InstLabel6\ @\ InstLabel7\ @\ InstLabel8\ @\ Match2\ @\ bytecode$;
$s =\ (None,\ sh,\ h,\ inih,\ [([],\ vs[\ 4:=\ Intg\ nargs,\ 5:=\ Intg\ i],\ safeP,\ sigSafeMain,$
$pc,\ ref)])\ ⟧$
$\Longrightarrow\ P' \vdash\ s\ -jvm\rightarrow\ lastState\ P'\ s$
apply (*simp add*: *lastState-def*)
apply (*intro MATCH-loop*)
apply (*simp add*: *diff-def*)
by *assumption+*

axioms *map-of-update-add*:
$x \notin set\ xs$
$\Longrightarrow S(x \mapsto y)\ ++\ map\text{-}of\ (zip\ xs\ ys) = S\ ++\ map\text{-}of\ (zip\ (xs@[x])\ (ys@[y]))$

axioms *map-upt-rev*:
$m < n \Longrightarrow map\ f\ [n>..Suc\ m]\ @\ [f\ m] = map\ f\ [n>..m]$

lemma *map-argCell-upt-rev*:
$⟦\ i < nargs;\ i \geq 0\ ⟧$
$\Longrightarrow (map\ (\lambda j.\ argCell\ fs\ (Suc\ j))\ [nat\ nargs>..nat\ (i\ +\ 1)]\ @\ [argCell\ fs\ (Suc$
$(nat\ i))])$
$= (map\ (\lambda j.\ argCell\ fs\ (Suc\ j))\ [nat\ nargs>..nat\ i])$
apply (*subgoal-tac nat* $(i\ +\ 1) = Suc\ (nat\ i)$,*simp*)
apply (*subgoal-tac nat* $i < nat\ nargs$)
apply (*rule-tac* $f=(\lambda j.\ argCell\ fs\ (Suc\ j))$ **in** *map-upt-rev*,*assumption*)
by *force+*

lemma *empty-map-add-2*: $m\ ++\ empty = m$
by (*rule ext*) (*simp add*: *map-add-def split*: *option.split*)

lemma *upt-Suc-append-topf*:
$⟦\ i < nargs;\ i >= 0;\ (the\text{-}Intg\ (the\ (sh\ (stackC,\ topf)))) - nargs) >= 0\ ⟧$
$\Longrightarrow [Suc\ (nat\ (the\text{-}Intg\ (the\ (sh\ (stackC,\ topf)))) - nargs))..<nat\ (the\text{-}Intg\ (the$
$(sh\ (stackC,\ topf))) - i)]\ @$
$[nat\ (the\text{-}Intg\ (the\ (sh\ (stackC,\ topf))) - i)]$
$= [Suc\ (nat\ (the\text{-}Intg\ (the\ (sh\ (stackC,\ topf)))) - nargs))..<nat\ (the\text{-}Intg\ (the$
$(sh\ (stackC,\ topf))) - i + 1)]$
apply (*rule sym*)
apply (*subgoal-tac*

```
  nat (the-Intg (the (sh (stackC, topf))) − i + 1) =
   Suc (nat (the-Intg (the (sh (stackC, topf))) − i)),simp)
by arith+


lemma execSVMInstr-MATCH-loop-post [rule-format]:
  ∀ sh h inih vs i pc ref S'.
      n = diff (None, sh, h, inih, [([], vs[ 4:= Intg nargs, 5:= Intg i], safeP,
sigSafeMain, pc, ref)])
    ⟶ (P', cdm, ctm, com) = trSVM2JVM ((svms, ctmap), ini, ct, ah, ai, bc)
    ⟶ length vs = 10
    ⟶ i >= 0
    ⟶ (the-Intg (the (sh (stackC, topf))) − nargs) >= 0
    ⟶ nargs ≥ i
    ⟶ vs ! 6 = Addr l
    ⟶ sh (stackC, Sf) = Some (Addr la)
    ⟶ h la = Some (Arr ty m S')
    ⟶ h l  = Some (Obj cellC fs)
    ⟶ l ≠ la
     ⟶ drop pc (fst (snd (snd (snd (snd (the (method' (P', safeP) sigSafe-
Main)))))))) =
       Match12 @ InstLabel1 @ InstLabel2 @ InstLabel3 @ InstLabel4 @ InstLabel5
@
                 InstLabel6 @ InstLabel7 @ InstLabel8 @ Match2 @ bytecode
    ⟶ steps n P' (None, sh, h, inih, [([], vs[ 4:= Intg nargs, 5:= Intg i], safeP,
sigSafeMain, pc, ref)]) =
         (None, sh,
          (let vitems = map (%j. argCell fs (j + 1)) [nat nargs>..nat i];
               idxs   = map (%j. j)
                          [nat (the-Intg (the (sh (stackC, topf))) − nargs) + 1..<nat
(the-Intg (the (sh (stackC, topf))) − i + 1) ];
               S''    = S' ++ map-of (zip idxs vitems)
           in h(the-Addr (the (sh (stackC, Sf))) ↦  Arr ty m S'')),
          inih, [([], vs[ 4:= Intg nargs, 5:= Intg nargs],
          safeP, sigSafeMain, pc, ref)])
apply (induct n)
 apply (simp add: steps.simps,clarsimp)
 apply (simp add: diff-def)
 apply (subgoal-tac nargs = i,clarsimp)
  apply (subst upt-conv-Nil,simp)+
  apply (subst zip-Nil)
  apply (subst map-of.simps, subst empty-map-add-2)
  apply (rule ext,force)
 apply simp

apply (rule allI)+
apply (rule impI)+

apply (case-tac i < nargs)
```

apply (*erule-tac x=sh* **in** *allE*)
apply (*erule-tac x=h*(*the-Addr* (*the* (*sh* (*stackC*, *Sf*))) ↦ *Arr ty m*
(*S'*(*nat* (*the-Intg* (*the* (*sh* (*stackC*, *topf*))) − *i*) ↦ *argCell fs*
(*nat i* + *1*)))) **in** *allE*)
apply (*erule-tac x=inih* **in** *allE*)
apply (*erule-tac x=vs* **in** *allE*)
apply (*erule-tac x=i* + *1* **in** *allE*)
apply (*erule-tac x=pc* **in** *allE*)
apply (*erule-tac x=ref* **in** *allE*)
apply (*erule-tac x=*(*S'*(*nat* (*the-Intg* (*the* (*sh* (*stackC*, *topf*))) − *i*) ↦ *argCell fs*
(*nat i* + *1*))) **in** *allE*)
apply (*drule mp*)
apply (*frule diff-monotone*,*assumption*+)
apply (*frule-tac inih=inih* **and** *nargs=nargs* **and** *i=i* **and** *ref=ref* **in** *oneStep-state*,*assumption*+)

apply (*simp add*: *diff-def*)
apply (*drule mp*,*force*)+

apply (*frule diff-monotone*,*assumption*+)
apply (*frule-tac*
inih=inih **and** *nargs=nargs* **and** *i=i* **and** *ref=ref*
in *oneStep-MATCH*,*assumption*+,*simp*,*assumption*)
apply (*frule-tac*
inih=inih **and** *nargs=nargs* **and** *i=i* **and** *ref=ref*
in *oneStep-state*,*assumption*+)
apply (*drule-tac*
s=oneStep P' (*None*, *sh*, *h*, *inih*, [([], *vs*[*4* := *Intg nargs*, *5* := *Intg i*],
safeP, *sigSafeMain*, *pc*, *ref*)]) **in** *sym*)
apply *clarify*
apply *simp*
apply (*frule steps-monotone*, *assumption*+)
apply (*drule-tac*
s=steps (*diff* (*None*, *sh*, *h*, *inih*, [([], *vs*[*4* := *Intg nargs*, *5* := *Intg i*], *safeP*,
sigSafeMain, *pc*, *a*, *b*)])) *P'*
(*None*, *sh*, *h*, *inih*, [([], *vs*[*4* := *Intg nargs*, *5* := *Intg i*], *safeP*,
sigSafeMain, *pc*, *a*, *b*)]) **in** *sym*,*simp*)
apply (*drule-tac*
t=oneStep P' (*None*, *sh*, *h*, *inih*, [([], *vs*[*4* := *Intg nargs*, *5* := *Intg i*],
safeP, *sigSafeMain*, *pc*, *a*, *b*)]) **in** *sym*)
apply (*simp add*: *diff-def*)
apply (*subst map-of-update-add*,*simp*)
apply (*subst upt-Suc-append-topf*,*assumption*+)
apply (*subst map-argCell-upt-rev*,*assumption*+)
apply (*rule refl*)

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

lemma *execSVMInstr-MATCH-loop-state*:
⟦ (*P'*, *cdm*, *ctm*, *com*) = *trSVM2JVM* ((*svms*, *ctmap*), *ini*, *ct*, *ah*, *ai*, *bc*);
length vs = *10*;
vs ! *6* = *Addr l*;
sh (*stackC*, *Sf*) = *Some* (*Addr la*);
h la = *Some* (*Arr ty m S'*);
h l = *Some* (*Obj cellC fs*);
l ≠ *la*;
0 ≤ *i*;
0 ≤ *the-Intg* (*the* (*sh* (*stackC*, *topf*))) − *nargs*;
i ≤ *nargs*;
drop pc (*fst* (*snd* (*snd* (*snd* (*snd* (*the* (*method'* (*P'*, *safeP*) *sigSafeMain*)))))))
=
Match12 @ *InstLabel1* @ *InstLabel2* @ *InstLabel3* @ *InstLabel4* @ *InstLabel5* @
InstLabel6 @ *InstLabel7* @ *InstLabel8* @ *Match2* @ *bytecode*;
s = (*None*, *sh*, *h*, *inih*, [([], *vs*[*4* := *Intg nargs*, *5* := *Intg i*], *safeP*, *sigSafeMain*, *pc*, *ref*)]) ⟧
⟹ *P'* ⊢ *s* −*jvm*→ (*None*, *sh*,
(*let vitems* = *map* (%*j*. *argCell fs* (*j* + *1*)) [*nat nargs*>..*nat i*];
idxs = *map* (%*j*. *j*)
[*nat* (*the-Intg* (*the* (*sh* (*stackC*, *topf*))) − *nargs*) + *1*..<*nat* (*the-Intg* (*the* (*sh* (*stackC*, *topf*))) − *i* + *1*)];
S'' = *S'* ++ *map-of* (*zip idxs vitems*)
in h(*the-Addr* (*the* (*sh* (*stackC*, *Sf*))) ↦ *Arr ty m S''*)),
inih, [([], *vs*[*4* := *Intg nargs*, *5* := *Intg nargs*],
safeP, *sigSafeMain*, *pc*, *ref*)])
apply (*frule execSVMInstr-MATCH-loop*,*assumption*+)
apply (*simp add*: *lastState-def*)
apply (*subgoal-tac*
(*nat* (*nargs* − *i*)) = *diff* (*None*, *sh*, *h*, *inih*, [([], *vs*[*4* := *Intg nargs*, *5* := *Intg i*], *safeP*, *sigSafeMain*, *pc*, *ref*)]))
apply (*frule execSVMInstr-MATCH-loop-post*,*assumption*+)
apply (*simp only*: *diff-def*,*force*)
by (*simp add*: *diff-def*)

axioms *equivS-MATCH-bound*:
⟦ (*fst* (*the* (*map-of svms l*)) ! *i*) = *MATCH off ps*;
sh (*stackC*,*topf*) = *Some* (*Intg n*) ⟧ ⟹ *int off* ≤ *n*

axioms *maxPush-MATCH*:
fst (*the* (*map-of svms l*)) ! *i* = *MATCH off ps* ⟹ *n* + *nargs'* < *int m*

axioms *equivS-S′-n-minus-off*:
⟦ *equivS S S′ n ctm d g*; *int off* ≤ *n*; *h l′* = *Some* (*Arr ty m S′*) ⟧
⟹ ∃ *l fs-S′-n-minus-off*.
the (*d* (*nat* (*the-Intg* (*the* (*S′* (*nat* (*n* − *int off*)))))))) = *Addr l* ∧
h l = *Some* (*Obj cellC fs-S′-n-minus-off*) ∧
l′ ≠ *l*

axioms *consTableC-def*:
sha (*consTableC*, *tablef*) = *Some* (*Addr l‴*) ∧
h l‴ = *Some* (*Arr ty m‴ consTableS*)

axioms *consTableC-n-minus-off*:
⟦ *sha* (*consTableC*, *tablef*) = *Some* (*Addr l‴*);
h l‴ = *Some* (*Arr ty m‴ consTableS*)⟧
⟹ ∃ *l′ fs-consTableS-S′-n-minus-off-tagGf*.
the (*consTableS* (*nat*
(*the-Intg* (*the* (*fs-S′-n-minus-off* (*tagGf*, *cellC*)))))) = *Addr l′* ∧
h l′ = *Some* (*Obj cellC fs-consTableS-S′-n-minus-off-tagGf*) ∧
the (*fs-consTableS-S′-n-minus-off-tagGf* (*nargsf*, *consDataC*))
= *Intg nargs′* ∧
nargs′ ≥ *0*

axioms *consTable-bounds*:
int m‴ > *the-Intg* (*the* (*fs-S′-n-minus-off* (*tagGf*, *cellC*))) ∧
the-Intg (*the* (*fs-S′-n-minus-off* (*tagGf*, *cellC*))) ≥ *0*

axioms *MATCH-bounds*:
0 ≤ *i* ∧
0 ≤ *the-Intg* (*the* (*sh* (*stackC*, *topf*))) − *nargs* ∧
i ≤ *nargs*

axioms *tag-bounds*:
¬ *the-Intg* (*the* (*fs-consTableS-S′-n-minus-off-tagGf*
(*tagLf*, *consDataC*))) < *0* ∧
¬ *int* (*length ps* − *Suc 0*) <
the-Intg (*the* (*fs-consTableS-S′-n-minus-off-tagGf*
(*tagLf*, *consDataC*)))

lemma *drop-append-3*:
drop n xs = [] @ (*ys* @ *zs*) ==> *drop n xs* = *ys* @ *zs*
by *simp*

lemma *drop-Suc-2*:
drop n xs = (*y* # *ys*) ⟹ *drop* (*Suc n*) *xs* = *ys*
apply (*induct xs arbitrary*: *n*, *simp*)
apply(*simp add*:*drop-Cons nth-Cons split*:*nat.splits*)
done

lemma *drop-length-append*:
drop n xs = (*ys* @ *zs*) ⟹ *drop* (*n* + *length ys*) *xs* = *zs*
by (*induct n arbitrary*: *xs*) (*auto*, *case-tac xs*, *auto*)

lemma *go-End*:
drop (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc*
(*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc*
(*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*the* (*cdm* (*l*, *i*))))))))))))))))))))))))))
(*fst* (*snd* (*snd* (*snd* (*snd* (*the* (*method'* (*P'*, *safeP*) *sigSafeMain*))))))) =
[*Ifcmp GreaterEqual labelEndLoop*, *Load 5*, *Tableswitch 0 7*
[*label1*, *label2*, *label3*, *label4*, *label5*, *label6*, *label7*, *label8*]] @
InstLabel1 @
InstLabel2 @
InstLabel3 @
InstLabel4 @
InstLabel5 @
InstLabel6 @
InstLabel7 @
InstLabel8 @
Match2 @
Load 3 #
Tableswitch 0 (*int* (*length ps* − *Suc 0*))
(*map* (λ*n*. *trAddr n* (*the* (*cdm* (*l*, *i*)) + *incMatch*))
(*map* (λ*p*. *the* (*cdm'* (*p*, *0*))) *ps*)) #
bytecode' ⟹
drop (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc*
(*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc*
(*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*the* (*cdm* (*l*, *i*)) + *nat labelEndLoop*))))))))))))))))))))))))))
(*fst* (*snd* (*snd* (*snd* (*snd* (*the* (*method'* (*P'*, *safeP*) *sigSafeMain*)))))))
= *Load 3* #
Tableswitch 0 (*int* (*length ps* − *Suc 0*))
(*map* (λ*n*. *trAddr n* (*the* (*cdm* (*l*, *i*)) + *incMatch*))
(*map* (λ*p*. *the* (*cdm'* (*p*, *0*))) *ps*)) #
bytecode'
apply (*drule drop-Suc-append*)+
apply (*drule drop-append-3*)
apply (*drule drop-length-append*)
apply (*drule drop-length-append*)
apply (*drule drop-length-append*)
apply (*drule drop-length-append*)
apply (*drule drop-length-append*)
apply (*drule drop-length-append*)
apply (*drule drop-length-append*)
apply (*drule drop-length-append*)
apply (*simp add*: *labelEndLoop-def*)
apply (*simp add*: *nlab1-def*)

```
apply (simp add: nlab2-def)
apply (simp add: nlab3-def)
apply (simp add: nlab4-def)
apply (simp add: nlab5-def)
apply (simp add: nlab6-def)
apply (simp add: nlab7-def)
apply (simp add: nlab8-def)
apply (simp add: nMatch2-def)
apply (simp add: InstLabel1-def)
apply (simp add: InstLabel2-def)
apply (simp add: InstLabel3-def)
apply (simp add: InstLabel4-def)
apply (simp add: InstLabel5-def)
apply (simp add: InstLabel6-def)
apply (simp add: InstLabel7-def)
apply (simp add: InstLabel8-def)
apply (drule drop-length-append)
apply (simp add: Match2-def)
apply (subgoal-tac
  (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc
   (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc
   (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc
   (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc
   (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc
   (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc
   (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc
   (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (the (cdm (l, i))
   )))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))
   )))))))))))))))))))))))))))))))))))
      = (101 + the (cdm (l, i))))
by simp+


axioms equivS-MATCH:
  equivS S S′ n ctm d g
  ⟹ equivS (map Val vsa @ S)
         (S′ ++ map-of (zip [Suc (nat n)..<nat (n + nargs′ + 1)]
         (map (λj. argCell fs-S′-n-minus-off (Suc j)) [nat nargs′>..0])))
         (n + nargs′) ctm d g

axioms pc-MATCH:
  Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc
   (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc
   (Suc (Suc (Suc (Suc (the (cdm (l, i)) + nat labelEndLoop +
    nat (map (λn. trAddr n (the (cdm (l, i)) + incMatch)) (map (λp. the (cdm′
(p, 0))) ps) !
  nat (the-Intg (the (fs-consTableS-S′-n-minus-off-tagGf (tagLf, consDataC)))))))))))))))
   ))))))))))))))))))) =
   the (cdm (ps ! r, 0))
```

axioms *Obj-cells-equals*:
Obj cellC fs-S′-n-minus-off = Obja cellC flds
⟹ flds = fs-S′-n-minus-off

lemma *equivV-MATCH*:
⟦ hm b = Some (j,C,vs);
equivH (hm, k) h (nat (the-Intg (the (sha (heapC, kf)))))) com regS d g;
g b = Some l;
h l = Some (Obj cellC fs-S′-n-minus-off) ⟧
⟹ ∀ i<length vs. equivV (vs ! i) (argCell fs-S′-n-minus-off i) d g
apply (*simp add*: *equivH.simps*)
apply *clarsimp*
apply (*erule-tac x=b* **in** *ballE*)
apply *clarsimp*
apply (*frule Obj-cells-equals,simp*)
by (*simp add*: *dom-def*)

axioms *equivS-extend*:
equivS (map Val vs @ S)
(S′ ++ map-of (zip [Suc (nat n)..<nat (2 + (n + int (length vs)))]
(argCell fs-S′-n-minus-off (Suc (length vs)) #
map (λj. argCell fs-S′-n-minus-off (Suc j)) [length vs>..0])))
(n + int (length vs)) ctm d g

axioms *argCell-Suc-j*:
argCell fs-S′-n-minus-off 0 =
(argCell fs-S′-n-minus-off (Suc (length vs)) # map (λj. argCell fs-S′-n-minus-off (Suc j)) [length vs>..0]) !
(nat (1 + (n + int (length vs))) − Suc (nat n))

lemma *S′-map-of-0*:
n ≥ 0
⟹ (S′ ++ map-of (zip [Suc (nat n)..<nat (2 + (n + int (length vs)))]
(argCell fs-S′-n-minus-off (Suc (length vs)) #
map (λj. argCell fs-S′-n-minus-off (Suc j)) [length vs>..0])))
(nat (n + (1 + int (length vs)))) = Some (argCell fs-S′-n-minus-off 0)
apply (*subst map-add-Some-iff*)
apply (*rule disjI1*)
apply (*rule map-of-is-SomeI*)
apply (*subst map-fst-zip*)
apply (*subst length-upt,simp*)

apply (*subst length-upt-rev*)
apply *arith*
apply *simp*
apply (*subst set-zip*)
apply *simp*
apply (*rule-tac x=nat (1 + (n + int (length vs))) − Suc (nat n)* **in** *exI*)
apply (*rule conjI*)
apply (*subst nth-upt-2*)
apply *simp* **apply** *simp*
apply (*rule conjI*)
apply (*rule argCell-Suc-j*)
apply (*rule conjI*)
apply *simp*
by (*subst length-upt-rev*,*simp*)

axioms *equivV-HI*:
⟦ ∀ *i*<*Suc* (*length vs*). *equivV* ((*a* # *vs*) ! *i*) (*argCell fs-S′-n-minus-off i*) *d g* ⟧
⟹ (∀ *i*<*length vs*. *equivV* (*vs*! *i*) (*argCell fs-S′-n-minus-off i*) *d g*)

lemma *equivS-MATCH* [*rule-format*]:
equivS S S′ n ctm d g
⟶ *n* ≥ *0*
⟶ (∀ *i*<*length vs*. *equivV* (*vs* ! *i*) (*argCell fs-S′-n-minus-off i*) *d g*)
⟶ *equivS* (*map Val vs* @ *S*)
(*S′* ++ *map-of* (*zip* [*Suc* (*nat n*)..<*nat* (*n* + *int* (*length vs*) + *1*)]
(*map* (λ*j*. *argCell fs-S′-n-minus-off* (*Suc j*)) [*length vs*>..*0*])))
(*n* + *int* (*length vs*)) *ctm d g*
apply (*induct vs*)
apply (*simp add*: *equivS.simps*)
apply *clarsimp*
apply (*simp add*: *equivS.simps*)
apply (*rule conjI*)
apply (*subgoal-tac*
equivV a (*argCell fs-S′-n-minus-off 0*) *d g*)
apply (*subgoal-tac*
the ((*S′* ++ *map-of* (*zip* [*Suc* (*nat n*)..<*nat* (*2* + (*n* + *int* (*length vs*)))]
(*argCell fs-S′-n-minus-off* (*Suc* (*length vs*)) #
map (λ*j*. *argCell fs-S′-n-minus-off* (*Suc j*)) [*length vs*>..*0*])))
(*nat* (*n* + (*1* + *int* (*length vs*))))) = *argCell fs-S′-n-minus-off 0*,*simp*)
apply (*subst S′-map-of-0*,*assumption*,*simp*)
apply (*erule-tac x=0* **in** *allE*,*simp*)
apply (*frule equivV-HI*,*simp*)
by (*rule equivS-extend*)

axioms *equivS-MATCH-axiom*:
equivS (*map Val vsa* @ *S*)
(*S′* ++ *map-of* (*zip* [*Suc* (*nat n*)..<*nat* (*n* + *nargs′* + *1*)]
(*map* (λ*j*. *argCell fs-S′-n-minus-off* (*Suc j*)) [*nat nargs′*>..*0*])))

```
        (n + nargs') ctm d g


declare equivH.simps [simp del]

lemma activeCells-MATCH:
  ⟦la ∉ activeCells regS h (nat (the-Intg (the (sha (heapC, kf)))))⟧
   ⟹ activeCells regS h (nat (the-Intg (the (sha (heapC, kf))))) =
      activeCells regS
       (h(la ↦
        Arr ty m
          (S' ++ map-of (zip [Suc (nat n)..<nat (n + nargs' + 1)] (map (λj.
argCell fs-S'-n-minus-off (Suc j)) [nat nargs'>..0])))))
       (nat (the-Intg (the (sha (heapC, kf)))))
apply (frule l-not-in-cellReg)
apply (unfold activeCells-def,auto)
 apply (unfold region-def, simp add: Let-def, elim conjE)
 apply (rule-tac x=j in exI,simp)
 apply (rule cellReg-step)
  apply (rule cellReg-basic)
 apply (erule-tac x=j in allE)
 apply (rule cellsReg-monotone-1,assumption+)
 apply (erule l-not-in-regs)
apply (simp add: Let-def, elim conjE)
apply (rule-tac x=j in exI, simp)
apply (rule cellReg-step)
 apply (rule cellReg-basic)
 apply (erule-tac x=j in allE)
apply (erule cellsReg-monotone-2,assumption+)
by (erule l-not-in-regs)



lemma domH-MATCH:
  ⟦ la ∉ activeCells regS h (nat (the-Intg (the (sha (heapC, kf)))));
     ∀ l∈dom H. ∃ l'. l' = the (g l) ∧ equivC (the (H l)) h l' (the (h l')) (nat
(the-Intg (the (sha (heapC, kf))))) com regS d g ⟧
   ⟹ ∀ l∈dom H.
        ∃ l'. l' = the (g l) ∧
             equivC (the (H l))
              (h(la ↦
               Arr ty m
                (S' ++ map-of (zip [Suc (nat n)..<nat (n + nargs' + 1)]
                                   (map (λj. argCell fs-S'-n-minus-off (Suc j)) [nat
nargs'>..0])))))
              l' (the ((h(la ↦
                       Arr ty m
                        (S' ++ map-of (zip [Suc (nat n)..<nat (n + nargs' + 1)]
```

(*map* (λ*j*. *argCell fs-S′-n-minus-off* (*Suc j*))
[*nat nargs′*>..*0*])))))
l′))
(*nat* (*the-Intg* (*the* (*sha* (*heapC*, *kf*))))) *com regS d g*
apply (*frule l-not-in-cellReg*)
apply (*rule ballI*)
apply (*erule-tac x*=*l* **in** *ballE*)
prefer *2* **apply** *simp*
apply (*erule exE*)
apply (*rule-tac x*=*l′* **in** *exI*) **apply** (*elim conjE*)
apply (*rule conjI*, *assumption*)
apply *clarsimp*
apply (*rule conjI*) **apply** (*rule impI*)
apply *clarsimp* **apply** (*simp add*: *region-def*)
apply *clarsimp*
apply (*rule-tac x*=*Obj* **in** *exI*)
apply (*rule-tac x*=*flds* **in** *exI*) **apply** *simp*
apply (*simp add*: *region-def*) **apply** (*elim conjE*)
apply (*rule cellReg-step*)
apply (*rule cellReg-basic*)
apply (*erule-tac x*=*j* **in** *allE*)
apply (*rule cellsReg-monotone-1*,*assumption*+)
by (*erule l-not-in-regs*)

lemma *equivH-MATCH*:
⟦ *k′* = *nat* (*the-Intg* (*the* (*sha* (*heapC*, *kf*))));
equivH (*hm*, *k*) *h* (*nat* (*the-Intg* (*the* (*sha* (*heapC*, *kf*))))) *com regS d g*;
sha (*stackC*, *Sf*) = *Some* (*Addr la*);
activeCells regS h k′ ∩ {*la*, *l′*, *l′′*} = {}⟧
⟹ *equivH* (*hm*, *k*)
(*h*(*la* ↦
Arr ty m
(*S′* ++ *map-of* (*zip* [*Suc* (*nat n*)..<*nat* (*n* + *nargs′* + *1*)] (*map* (λ*j*.
argCell fs-S′-n-minus-off (*Suc j*)) [*nat nargs′*>..*0*])))))
(*nat* (*the-Intg* (*the* (*sha* (*heapC*, *kf*))))) *com regS d g*
apply (*simp add*: *heapC-def add*: *stackC-def*)
apply (*simp add*: *kf-def add*: *k0f-def*)
apply (*fold heapC-def*, *fold kf-def*)
apply (*unfold equivH.simps*, *elim conjE*)
apply (*rule conjI*, *assumption*)
apply (*fold stackC-def*)
apply (*rule conjI*)
apply (*subgoal-tac*
la ∉ *activeCells regS h* (*nat* (*the-Intg* (*the* (*sha* (*heapC*, *kf*))))))
apply (*frule activeCells-MATCH*,*simp*,*simp*)
apply (*rule conjI*, *assumption*)
by (*rule domH-MATCH*,*assumption*,*simp*)

lemma *execSVMInstr-MATCH* :
⟦ (*P′*, *cdm*, *ctm*, *com*) = *trSVM2JVM* ((*svms*, *ctmap*), *ini*, *ct*, *ah*, *ai*, *bc*);
cdm , *ctm*, *com* ⊢ ((*hm*, *k*), *k0*, (*l*, *i*), *S*) ≙ *S1′*;
(*fst* (*the* (*map-of svms l*)) ! *i*) = *MATCH off ps*;
execSVMInst (*MATCH off ps*) (*map-of ct*) (*hm*, *k*) *k0* (*l*, *i*) *S* =
Either.Right S2;
drop (*the* (*cdm* (*l*, *i*))) (*extractBytecode P′*) =
trInstr (*the* (*cdm* (*l*, *i*))) *cdm′ ctm′ com pcc* (*MATCH off ps*) @ *bytecode′*;
cdm′ ⊆*m* *cdm*
⟧ ⟹ ∃ *v′ sh′ dh′ ih′ fms′* .
P′ ⊢ *S1′* −*jvm*→ (*v′*,*sh′*,*dh′*,*ih′*, *fms′*) ∧
cdm , *ctm*, *com* ⊢ *S2* ≙ (*v′*,*sh′*,*dh′*,*ih′*, *fms′*)

apply (*case-tac S1′*)
apply (*rename-tac v1 tup*)
apply (*case-tac tup*)
apply (*rename-tac sh tup*)
apply (*case-tac tup*)
apply (*rename-tac dh tup*)
apply (*case-tac tup*)
apply (*rename-tac ih fms*)
apply (*simp del*: *execSVMInst.simps*)
apply (*unfold equivState-def*)
apply (*elim exE*,*elim conjE*)
apply (*subgoal-tac int off* ≤ *n*)
prefer *2* **apply** (*erule equivS-MATCH-bound*, *simp*)
apply *clarify*

apply (*simp only*: *execSVMInst.simps*)
apply (*insert RightNotUndefined*)
apply (*erule-tac x=S2* **in** *allE*)
apply (*case-tac S*!+ *off*)
defer apply (*force*,*force*)
apply (*case-tac Vala*)
defer apply (*force*, *force*)
apply (*case-tac hm nata*)
apply *force*
apply (*case-tac aha*,*rename-tac j rest*,*case-tac rest*,*rename-tac C vsa*)
apply (*case-tac map-of ct C*)
apply *force*
apply (*rename-tac tup*, *case-tac tup*,
rename-tac nargs rest′, *case-tac rest′*, *rename-tac r xa*)
apply *clarsimp*

apply (*frule equivS-S′-n-minus-off*,*assumption*+)

```
apply (elim exE, elim conjE)
apply (subgoal-tac
  sha (consTableC, tablef) = Some (Addr l''') ∧
   h l''' = Some (Arr ty m''' consTableS))
prefer 2 apply (rule consTableC-def)
apply (erule conjE)
apply (subgoal-tac
  ∃ l' fs-consTableS-S'-n-minus-off-tagGf.
     the (consTableS (nat (the-Intg (the (fs-S'-n-minus-off (tagGf, cellC)))))) =
Addr l' ∧
     h l' = Some (Obj cellC fs-consTableS-S'-n-minus-off-tagGf) ∧
     the (fs-consTableS-S'-n-minus-off-tagGf (nargsf, consDataC)) = Intg nargs'
∧
     nargs' ≥ 0)
apply (elim exE, elim conjE)
prefer 2
  apply (rule-tac l'''=l''' in consTableC-n-minus-off,assumption+)


apply (rule-tac x=sha((stackC, topf) ↦ Intg (n + nargs')) in exI)
apply (rule-tac x=h(la ↦ Arr ty m (S' ++ map-of (zip [Suc (nat n)..<nat (n +
nargs' + 1)]
                                                (map (λj. argCell fs-S'-n-minus-off (Suc j))
                                                    [nat nargs'>..0])))) in exI)
apply (rule-tac x=inih in exI)
apply (rule-tac x=[([],vs[Suc 0 := the (S' (nat (n − int off))),
                          6 := Addr laa,
                          2 := the (fs-S'-n-minus-off (tagGf, cellC)),
                             3 := the (fs-consTableS-S'-n-minus-off-tagGf (tagLf,
consDataC)),
                          4 := Intg nargs',
                          5 := Intg nargs'],
                  safeP, sigSafeMain,
                   Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc
(Suc
                   (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc
(Suc
                  (Suc (Suc (Suc (Suc (the (cdm (l, i)) + nat labelEndLoop +
                   nat (map (λn. trAddr n (the (cdm (l, i)) + incMatch))
                      (map (λp. the (cdm' (p, 0))) ps) !
                    nat (the-Intg (the (fs-consTableS-S'-n-minus-off-tagGf (tagLf,
consDataC))
                   ))))))))))))))))))))))))))))))),
                  ag,bd)] in exI)


apply (subgoal-tac length vs = 10)
prefer 2 apply (rule length-vs)
apply (rule conjI)
```

apply (*subgoal-tac*
  P'⊢ (*None*,*sha*,*h*,*inih*,[([],
      vs, *safeP*, *sigSafeMain*, *the* (*cdm* (*l*,*i*)), *ag*,*bd*)])
  −jvm→
      (*None*,*sha*,*h*,*inih*,[([*Addr la*],
      vs, *safeP*, *sigSafeMain*, *the* (*cdm* (*l*,*i*)) + *1*, *ag*,*bd*)]))
apply (*unfold exec-all-def*)
apply (*erule rtrancl-trans*)

prefer *2*
 apply (*rule r-into-rtrancl*)
 apply *clarify*
 apply (*unfold JVMExec.exec.simps*)
 apply (*unfold Let-def*)
 apply (*unfold extractBytecode-def*)
 apply (*subgoal-tac* (*fst* (*snd* (*snd* (*snd* (*snd* (*the* (*method'* (*P'*, *safeP*) *sigSafe-Main*))))))) !
                the (*cdm* (*l*, *i*))) = *Getstatic Sf stackC*) **apply** *simp*
 apply (*unfold trInstr.simps*, *unfold Let-def*)
apply (*unfold Match11-def*)
apply (*rule nth-via-drop-append*) **apply** *force*

apply (*drule-tac ?ms=bytecode'* **and** *?y= Getstatic Sf stackC*
          in *drop-Suc-append-2*)
apply (*subgoal-tac*
  P'⊢ (*None*, *sha*, *h*, *inih*,
      [([*Addr la*],
      vs, *safeP*, *sigSafeMain*, *the* (*cdm* (*l*, *i*)) + *1*, *ag*, *bd*)])
  −jvm→
      (*None*,*sha*,*h*,*inih*,
      [([*Intg n*,*Addr la*],
      vs, *safeP*, *sigSafeMain*, *the* (*cdm* (*l*, *i*)) + *1* + *1*, *ag*, *bd*)]))
apply (*unfold exec-all-def*)
apply (*simp del*:*append.append-Cons*)
apply (*erule rtrancl-trans*)
prefer *2*
 apply (*rule r-into-rtrancl*)
 apply (*clarify*)
 apply (*unfold JVMExec.exec.simps*)
 apply (*unfold Let-def*)
 apply (*subgoal-tac* (*fst* (*snd* (*snd* (*snd* (*snd* (*the* (*method'* (*P'*, *safeP*) *sigSafe-Main*))))))
      ! *Suc* (*the* (*cdm* (*l*, *i*)))) = *Getstatic topf stackC*,*simp*)
 apply (*erule nth-via-drop-append*)

apply (*drule drop-Suc-append*)

```
apply (subgoal-tac
  P'⊢ (None,sha,h,inih,
       [([Intg n,Addr la],
       vs, safeP, sigSafeMain, the (cdm (l, i)) + 1 + 1, ag, bd)])
  −jvm→
       (None,sha,h,inih,
       [([Intg (int off),
          Intg n,Addr la],
       vs, safeP, sigSafeMain, the (cdm (l, i)) + 1 + 1 + 1, ag, bd)]))
apply (unfold exec-all-def)
apply (simp del:append.append-Cons)
apply (erule rtrancl-trans)
prefer 2
 apply (rule r-into-rtrancl)
 apply (clarify)
 apply (unfold JVMExec.exec.simps)
 apply (unfold Let-def)
 apply (subgoal-tac (fst (snd (snd (snd (snd (the (method' (P', safeP) sigSafe-
Main))))))
      ! (Suc (Suc (the (cdm (l, i))))) = LitPush (Intg (int off)),simp)
apply (erule nth-via-drop-append)


apply (drule drop-Suc-append)
apply (subgoal-tac P'⊢ (None,sha,h,inih,
                        [([Intg (int off),
                           Intg n,Addr la],
                      vs, safeP, sigSafeMain, the (cdm (l, i)) + 1 + 1 + 1, ag, bd)])
                    −jvm→
                        (None,sha,h,inih,
                        [([Intg ( n − int off),
                           Addr la],
                         vs, safeP, sigSafeMain, the (cdm (l, i)) + 1 + 1 + 1 + 1,
ag, bd)]))
apply (unfold exec-all-def)
apply (simp del:append.append-Cons)
apply (erule rtrancl-trans)
prefer 2
 apply (rule r-into-rtrancl)
 apply (clarify)
 apply (unfold JVMExec.exec.simps)
 apply (unfold Let-def)
 apply (subgoal-tac (fst (snd (snd (snd (snd (the (method' (P', safeP) sigSafe-
Main))))))
      ! (Suc (Suc (Suc (the (cdm (l, i)))))) = BinOp Substract,simp)
apply (erule nth-via-drop-append)


apply (drule drop-Suc-append)
```

apply (*subgoal-tac P'⊢* (*None*,*sha*,*h*,*inih*,
[([*Intg* (*n* − *int off*),
Addr la],
vs, *safeP*, *sigSafeMain*, *the* (*cdm* (*l*, *i*)) + *1* + *1* + *1* + *1*,
ag, *bd*)])
−jvm→
(*None*,*sha*,*h*,*inih*,
[([*the* (*S'* (*nat* (*n* − *int off*)))],
vs, *safeP*, *sigSafeMain*, *the* (*cdm* (*l*, *i*)) + *1* + *1* + *1* + *1* +
1, *ag*, *bd*)]))
apply (*unfold exec-all-def*)
apply (*simp del*:*append.append-Cons*)
apply (*erule rtrancl-trans*)
prefer *2*
apply (*rule r-into-rtrancl*)
apply (*clarify*)
apply (*unfold JVMExec.exec.simps*)
apply (*unfold Let-def*)
apply (*subgoal-tac* (*fst* (*snd* (*snd* (*snd* (*snd* (*the* (*method'* (*P'*, *safeP*) *sigSafe-Main*)))))) !
(*Suc* (*Suc* (*Suc* (*Suc* (*the* (*cdm* (*l*, *i*)))))))) = *ArrLoad*)
apply (*simp del*: *Let-def execSVMInst.simps*)
apply (*simp* (*no-asm*) *add*: *raise-system-xcpt-def*)
apply (*rule conjI*) **apply** (*force*,*force*)
apply (*erule nth-via-drop-append*)

apply (*drule drop-Suc-append*)
apply (*subgoal-tac*
P'⊢ (*None*,*sha*,*h*,*inih*,[([*the* (*S'* (*nat* (*n* − *int off*)))],
vs, *safeP*, *sigSafeMain*, *the* (*cdm* (*l*, *i*)) + *1* + *1* + *1* + *1* + *1*, *ag*, *bd*)])
−jvm→
(*None*,*sha*,*h*,*inih*,[([],*vs*[*Suc 0* := *the* (*S'* (*nat* (*n* − *int off*)))],
safeP, *sigSafeMain*, *the* (*cdm* (*l*, *i*)) + *1* + *1* + *1* + *1* + *1* + *1*, *ag*, *bd*)]))
apply (*unfold exec-all-def*)
apply (*simp del*:*append.append-Cons*)
apply (*erule rtrancl-trans*)
prefer *2*
apply (*rule r-into-rtrancl*)
apply (*clarify*)
apply (*unfold JVMExec.exec.simps*)
apply (*unfold Let-def*)
apply (*subgoal-tac* (*fst* (*snd* (*snd* (*snd* (*snd* (*the* (*method'* (*P'*, *safeP*) *sigSafe-Main*)))))) !
(*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*the* (*cdm* (*l*, *i*))))))))) =
Store (*Suc 0*)) **apply** (*simp*)
apply (*erule nth-via-drop-append*)

apply (*drule drop-Suc-append*)
apply (*subgoal-tac P'⊢* (*None*, *sha*, *h*, *inih*,
[([], *vs*[*Suc 0* := *the* (*S'* (*nat* (*n* − *int off*)))], *safeP*, *sigSafeMain*,
the (*cdm* (*l*, *i*)) + *1* + *1* + *1* + *1* + *1* + *1*, *ag*, *bd*)])
−jvm→
(*None*,*sha*,*h*,*inih*,
[([*Addr l'*], *vs*[*Suc 0* := *the* (*S'* (*nat* (*n* − *int off*)))],
safeP, *sigSafeMain*, *the* (*cdm* (*l*, *i*)) + *1* + *1* + *1* + *1* + *1* + *1* + *1*, *ag*,
bd)]))
apply (*unfold exec-all-def*)
apply (*simp del*:*append.append-Cons*)
apply (*erule rtrancl-trans*)
prefer *2*
apply (*rule r-into-rtrancl*)
apply (*clarify*)
apply (*unfold JVMExec.exec.simps*)
apply (*unfold Let-def*)
apply (*subgoal-tac* (*fst* (*snd* (*snd* (*snd* (*snd* (*the* (*method'* (*P'*, *safeP*) *sigSafe-Main*)))))) !
(*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*the* (*cdm* (*l*, *i*))))))))) =
Getstatic safeDirf dirCellC,*simp*)
apply (*erule nth-via-drop-append*)

apply (*drule drop-Suc-append*)
apply (*subgoal-tac P'⊢* (*None*,*sha*,*h*,*inih*,
[([*Addr l'*], *vs*[*Suc 0* := *the* (*S'* (*nat* (*n* − *int off*)))],
safeP, *sigSafeMain*, *the* (*cdm* (*l*, *i*)) + *1* + *1* + *1* + *1* + *1* + *1* + *1*, *ag*,
bd)])
−jvm→
(*None*,*sha*,*h*,*inih*,
[([*the* (*S'* (*nat* (*n* − *int off*))), *Addr l'*],
vs[*Suc 0* := *the* (*S'* (*nat* (*n* − *int off*)))],
safeP, *sigSafeMain*, *the* (*cdm* (*l*, *i*)) + *1* + *1* + *1* + *1* + *1* + *1* + *1* +
1, *ag*, *bd*)]))
apply (*unfold exec-all-def*)
apply (*simp del*: *append.append-Cons*)
apply (*erule rtrancl-trans*)
prefer *2*
apply (*rule r-into-rtrancl*)
apply (*clarify*)
apply (*unfold JVMExec.exec.simps*)
apply (*unfold Let-def*)
apply (*subgoal-tac fst* (*snd* (*snd* (*snd* (*snd* (*the* (*method'* (*P'*, *safeP*) *sigSafe-Main*)))))) !
Suc (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*the* (*cdm* (*l*, *i*))))))))) =
Load (*Suc 0*)) **apply** (*simp* (*no-asm-simp*))
apply (*erule nth-via-drop-append*)

apply (*drule drop-Suc-append*)

apply (*subgoal-tac P* ⊢ (*None*,*sha*,*h*,*inih*,
[([*the* (*S'* (*nat* (*n* − *int off*))), *Addr l''*],
vs[*Suc 0* := *the* (*S'* (*nat* (*n* − *int off*)))],
safeP, *sigSafeMain*, *the* (*cdm* (*l*, *i*)) + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1*, *ag*, *bd*)])
−*jvm*→
(*None*,*sha*,*h*,*inih*,
[([*Addr laa*],
vs[*Suc 0* := *the* (*S'* (*nat* (*n* − *int off*)))],
safeP, *sigSafeMain*, *the* (*cdm* (*l*, *i*)) + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1*, *ag*, *bd*)]))
apply (*unfold exec-all-def*)
apply (*simp del*: *append.append-Cons*)
apply (*erule rtrancl-trans*)
prefer *2*
apply (*rule r-into-rtrancl*)
apply (*clarify*)
apply (*unfold JVMExec.exec.simps*)
apply (*unfold Let-def*)
apply (*subgoal-tac fst* (*snd* (*snd* (*snd* (*snd* (*the* (*method'* (*P'*, *safeP*) *sigSafeMain*)))))) !
Suc (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*the* (*cdm* (*l*, *i*))))))))))
=
ArrLoad) **apply** (*simp* (*no-asm-simp*) *add*: *raise-system-xcpt-def*)
apply (*subgoal-tac the-Intg* (*the* (*S'* (*nat* (*n* − *int off*)))) < *int m''*
∧ *0* ≤ *the-Intg* (*the* (*S'* (*nat* (*n* − *int off*)))))
prefer *2* **apply** (*rule safeDir-bounds*)
apply (*rule conjI*) **apply** (*force*,*force*)
apply (*erule nth-via-drop-append*)

apply (*drule drop-Suc-append*)
apply (*subgoal-tac P* ⊢ (*None*,*sha*,*h*,*inih*,
[([*Addr laa*],
vs[*Suc 0* := *the* (*S'* (*nat* (*n* − *int off*)))],
safeP, *sigSafeMain*, *the* (*cdm* (*l*, *i*)) + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1*, *ag*, *bd*)])
−*jvm*→
(*None*,*sha*,*h*,*inih*,
[([],
vs[*Suc 0* := *the* (*S'* (*nat* (*n* − *int off*))),
6 := *Addr laa*],
safeP, *sigSafeMain*, *the* (*cdm* (*l*, *i*)) + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1*, *ag*, *bd*)]))
apply (*unfold exec-all-def*)
apply (*simp del*: *append.append-Cons*)

apply (*erule rtrancl-trans*)
prefer *2*
apply (*rule r-into-rtrancl*)
apply (*clarify*)
apply (*unfold JVMExec.exec.simps*)
apply (*unfold Let-def*)
apply (*subgoal-tac fst* (*snd* (*snd* (*snd* (*snd* (*the* (*method'* (*P'*, *safeP*) *sigSafeMain*)))))) !
(*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*the* (*cdm* (*l*, *i*))))))))))) =
Store 6) **apply** (*simp* (*no-asm-simp*))
apply (*erule nth-via-drop-append*)

apply (*drule drop-Suc-append*)
apply (*subgoal-tac P'*⊢ (*None*,*sha*,*h*,*inih*,
[([],
vs[*Suc 0* := *the* (*S'* (*nat* (*n* − *int off*))),
6 := *Addr laa*],
safeP, *sigSafeMain*, *the* (*cdm* (*l*, *i*)) + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1*, *ag*, *bd*)])
−*jvm*→
(*None*,*sha*,*h*,*inih*,
[([*Addr laa*],
vs[*Suc 0* := *the* (*S'* (*nat* (*n* − *int off*))),
6 := *Addr laa*],
safeP, *sigSafeMain*, *the* (*cdm* (*l*, *i*)) + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1*, *ag*, *bd*)]))
apply (*unfold exec-all-def*)
apply (*simp del*: *append.append-Cons*)
apply (*erule rtrancl-trans*)
prefer *2*
apply (*rule r-into-rtrancl*)
apply (*clarify*)
apply (*unfold JVMExec.exec.simps*)
apply (*unfold Let-def*)
apply (*subgoal-tac fst* (*snd* (*snd* (*snd* (*snd* (*the* (*method'* (*P'*, *safeP*) *sigSafeMain*)))))) !
(*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*the* (*cdm* (*l*, *i*))))))))))) =
Load 6) **apply** (*simp* (*no-asm-simp*))
apply (*erule nth-via-drop-append*)

apply (*drule drop-Suc-append*)
apply (*subgoal-tac P'*⊢ (*None*,*sha*,*h*,*inih*,
[([*Addr laa*],
vs[*Suc 0* := *the* (*S'* (*nat* (*n* − *int off*))),
6 := *Addr laa*],

```
        safeP, sigSafeMain, the (cdm (l, i)) + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1
+ 1 + 1 + 1, ag, bd)])
      −jvm→
        (None,sha,h,inih,
        [([the (fs-S'-n-minus-off (tagGf, cellC))],
         vs[Suc 0 := the (S' (nat (n − int off))),
               6 := Addr laa],
         safeP, sigSafeMain, the (cdm (l, i)) + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1
+ 1 + 1 + 1 + 1, ag, bd)]))
apply (unfold exec-all-def)
apply (simp del: append.append-Cons)
apply (erule rtrancl-trans)
prefer 2
 apply (rule r-into-rtrancl)
 apply (clarify)
 apply (unfold JVMExec.exec.simps)
 apply (Let-def)
 apply (subgoal-tac fst (snd (snd (snd (snd (the (method' (P', safeP) sigSafe-
Main)))))) !
                    (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (the
(cdm (l, i))))))))))))))) =
                          Getfield tagGf cellC) apply (simp (no-asm-simp) add:
raise-system-xcpt-def)
apply (simp (no-asm-simp))
apply (erule nth-via-drop-append)


apply (drule drop-Suc-append)
apply (subgoal-tac P'⊢ (None,sha,h,inih,
        [([the (fs-S'-n-minus-off (tagGf, cellC))],
         vs[Suc 0 := the (S' (nat (n − int off))),
               6 := Addr laa],
         safeP, sigSafeMain, the (cdm (l, i)) + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1
+ 1 + 1 + 1 + 1, ag, bd)])
      −jvm→
        (None,sha,h,inih,
        [([],
         vs[Suc 0 := the (S' (nat (n − int off))),
               6 := Addr laa,
               2 := the (fs-S'-n-minus-off (tagGf, cellC))],
         safeP, sigSafeMain, the (cdm (l, i)) + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1
+ 1 + 1 + 1 + 1 + 1, ag, bd)]))
apply (unfold exec-all-def)
apply (simp del: append.append-Cons)
apply (erule rtrancl-trans)
prefer 2
 apply (rule r-into-rtrancl)
 apply (clarify)
 apply (unfold JVMExec.exec.simps)
```

apply (*unfold Let-def*)
apply (*subgoal-tac fst* (*snd* (*snd* (*snd* (*snd* (*the* (*method'* (*P'*, *safeP*) *sigSafe-Main*))))))) !
(*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*the* (*cdm* (*l*, *i*))))))))))))))) =
Store 2) **apply** (*simp* (*no-asm-simp*))
apply (*erule nth-via-drop-append*)

apply (*drule drop-Suc-append*)
apply (*subgoal-tac*
sha (*consTableC*, *tablef*) = *Some* (*Addr l'''*) ∧
h l''' = *Some* (*Arr ty m'''* *consTableS*))
apply (*elim conjE*)
prefer *2* **apply** (*rule consTableC-def*)
apply (*subgoal-tac P'*⊢ (*None*,*sha*,*h*,*inih*,
[([],
vs[*Suc 0* := *the* (*S'* (*nat* (*n* − *int off*))),
6 := *Addr laa*,
2 := *the* (*fs-S'-n-minus-off* (*tagGf*, *cellC*))],
safeP, *sigSafeMain*, *the* (*cdm* (*l*, *i*)) + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1*, *ag*, *bd*)])
−*jvm*→
(*None*,*sha*,*h*,*inih*,
[([*Addr l'''*],
vs[*Suc 0* := *the* (*S'* (*nat* (*n* − *int off*))),
6 := *Addr laa*,
2 := *the* (*fs-S'-n-minus-off* (*tagGf*, *cellC*))],
safeP, *sigSafeMain*, *the* (*cdm* (*l*, *i*)) + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1*, *ag*, *bd*)]))
apply (*unfold exec-all-def*)
apply (*simp del*: *append.append-Cons*)
apply (*erule rtrancl-trans*)
prefer *2*
apply (*rule r-into-rtrancl*)
apply (*clarify*)
apply (*unfold JVMExec.exec.simps*)
apply (*unfold Let-def*)
apply (*subgoal-tac fst* (*snd* (*snd* (*snd* (*snd* (*the* (*method'* (*P'*, *safeP*) *sigSafe-Main*))))))) !
(*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*the* (*cdm* (*l*, *i*))))))))))))))) =
Getstatic tablef consTableC) **apply** (*simp* (*no-asm-simp*))
apply (*erule nth-via-drop-append*)

apply (*drule drop-Suc-append*)
apply (*subgoal-tac P'*⊢ (*None*,*sha*,*h*,*inih*,
[([*Addr l'''*],

```
        vs[Suc 0 := the (S' (nat (n − int off))),
              6 := Addr laa,
              2 := the (fs-S'-n-minus-off (tagGf, cellC))],
        safeP, sigSafeMain, the (cdm (l, i)) + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1
+ 1 + 1 + 1 + 1 + 1 + 1 + 1, ag, bd)])
      −jvm→
        (None,sha,h,inih,
        [([the (fs-S'-n-minus-off (tagGf, cellC)),
           Addr l'']],
        vs[Suc 0 := the (S' (nat (n − int off))),
              6 := Addr laa,
              2 := the (fs-S'-n-minus-off (tagGf, cellC))],
        safeP, sigSafeMain, the (cdm (l, i)) + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1
+ 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1, ag, bd)]))
apply (unfold exec-all-def)
apply (simp del: append.append-Cons)
apply (erule rtrancl-trans)
prefer 2
 apply (rule r-into-rtrancl)
 apply (clarify)
 apply (unfold JVMExec.exec.simps)
 apply (unfold Let-def)
 apply (subgoal-tac fst (snd (snd (snd (snd (the (method' (P', safeP) sigSafe-
Main)))))) !
                    (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc
(Suc (Suc (Suc (the (cdm (l, i)))))))))))))))))) =
                  Load 2) apply (simp (no-asm-simp))
apply (erule nth-via-drop-append)


apply (drule drop-Suc-append)
apply (subgoal-tac P'⊢ (None,sha,h,inih,
        [([the (fs-S'-n-minus-off (tagGf, cellC)),
           Addr l'']],
        vs[Suc 0 := the (S' (nat (n − int off))),
              6 := Addr laa,
              2 := the (fs-S'-n-minus-off (tagGf, cellC))],
        safeP, sigSafeMain, the (cdm (l, i)) + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1
+ 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1, ag, bd)])
      −jvm→
        (None,sha,h,inih,
        [([Addr l'a],
        vs[Suc 0 := the (S' (nat (n − int off))),
              6 := Addr laa,
              2 := the (fs-S'-n-minus-off (tagGf, cellC))],
        safeP, sigSafeMain, the (cdm (l, i)) + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1
+ 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1, ag, bd)]))
apply (unfold exec-all-def)
apply (simp del: append.append-Cons)
```

apply (*erule rtrancl-trans*)
prefer *2*
apply (*rule r-into-rtrancl*)
apply (*clarify*)
apply (*unfold JVMExec.exec.simps*)
apply (*unfold Let-def*)
apply (*subgoal-tac fst* (*snd* (*snd* (*snd* (*snd* (*the* (*method'* (*P'*, *safeP*) *sigSafeMain*)))))) !
(*Suc* (*the* (*cdm* (*l*, *i*))))))))))))))))))) =
ArrLoad) **apply** (*simp* (*no-asm-simp*) *add*: *raise-system-xcpt-def*)
apply (*subgoal-tac*
int m''' > *the-Intg* (*the* (*fs-S'-n-minus-off* (*tagGf*, *cellC*))) ∧
the-Intg (*the* (*fs-S'-n-minus-off* (*tagGf*, *cellC*))) ≥ *0*)
apply (*simp* (*no-asm-simp*))
apply *simp*
apply (*rule consTable-bounds*)
apply (*erule nth-via-drop-append*)

apply (*drule drop-Suc-append*)
apply (*subgoal-tac P*⊢ (*None*,*sha*,*h*,*inih*,
[([*Addr l'a*],
vs[*Suc 0* := *the* (*S'* (*nat* (*n* − *int off*))),
6 := *Addr laa*,
2 := *the* (*fs-S'-n-minus-off* (*tagGf*, *cellC*))],
safeP, *sigSafeMain*, *the* (*cdm* (*l*, *i*)) + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1*, *ag*, *bd*)])
−*jvm*→
(*None*,*sha*,*h*,*inih*,
[([*Addr l'a*,
Addr l'a],
vs[*Suc 0* := *the* (*S'* (*nat* (*n* − *int off*))),
6 := *Addr laa*,
2 := *the* (*fs-S'-n-minus-off* (*tagGf*, *cellC*))],
safeP, *sigSafeMain*, *the* (*cdm* (*l*, *i*)) + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1*, *ag*, *bd*)]))
apply (*unfold exec-all-def*)
apply (*simp del*: *append.append-Cons*)
apply (*erule rtrancl-trans*)
prefer *2*
apply (*rule r-into-rtrancl*)
apply (*clarify*)
apply (*unfold JVMExec.exec.simps*)
apply (*unfold Let-def*)
apply (*subgoal-tac fst* (*snd* (*snd* (*snd* (*snd* (*the* (*method'* (*P'*, *safeP*) *sigSafeMain*)))))) !
(*Suc* (*the* (*cdm* (*l*, *i*))))))))))))))))))) =

```
                Dup) apply (simp (no-asm-simp))
apply (erule nth-via-drop-append)


apply (drule drop-Suc-append)
apply (subgoal-tac P'⊢ (None,sha,h,inih,
       [([Addr l'a,
          Addr l'a],
         vs[Suc 0 := the (S' (nat (n − int off))),
               6 := Addr laa,
               2 := the (fs-S'-n-minus-off (tagGf, cellC))],
         safeP, sigSafeMain, the (cdm (l, i)) + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1
+ 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1, ag, bd)])
     −jvm→
       (None,sha,h,inih,
       [([the (fs-consTableS-S'-n-minus-off-tagGf (tagLf, consDataC)),
          Addr l'a],
         vs[Suc 0 := the (S' (nat (n − int off))),
               6 := Addr laa,
               2 := the (fs-S'-n-minus-off (tagGf, cellC))],
         safeP, sigSafeMain, the (cdm (l, i)) + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1
+ 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1, ag, bd)]))
apply (unfold exec-all-def)
apply (simp del: append.append-Cons)
apply (erule rtrancl-trans)
prefer 2
 apply (rule r-into-rtrancl)
 apply (clarify)
 apply (unfold JVMExec.exec.simps)
 apply (unfold Let-def)
 apply (subgoal-tac fst (snd (snd (snd (snd (the (method' (P', safeP) sigSafe-
Main)))))) !
                  (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc
(Suc (Suc (Suc (Suc
                  (Suc (the (cdm (l, i))))))))))))))))))) =
                     Getfield tagLf consDataC) apply (simp (no-asm-simp) add:
raise-system-xcpt-def)
apply (erule nth-via-drop-append)


apply (drule drop-Suc-append)
apply (subgoal-tac P'⊢ (None,sha,h,inih,
       [([the (fs-consTableS-S'-n-minus-off-tagGf (tagLf, consDataC)),
          Addr l'a],
         vs[Suc 0 := the (S' (nat (n − int off))),
               6 := Addr laa,
               2 := the (fs-S'-n-minus-off (tagGf, cellC))],
         safeP, sigSafeMain, the (cdm (l, i)) + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1
+ 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1, ag, bd)])
```

```
    −jvm→
      (None,sha,h,inih,
      [([Addr l'a],
       vs[Suc 0 := the (S' (nat (n − int off))),
            6 := Addr laa,
            2 := the (fs-S'-n-minus-off (tagGf, cellC)),
            3 := the (fs-consTableS-S'-n-minus-off-tagGf (tagLf, consDataC))],
       safeP, sigSafeMain,
       the (cdm (l, i)) + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1
+ 1 + 1 + 1 + 1 + 1 + 1 + 1,
       ag, bd)]))
apply (unfold exec-all-def)
apply (simp del: append.append-Cons)
apply (erule rtrancl-trans)
prefer 2
 apply (rule r-into-rtrancl)
 apply (clarify)
 apply (unfold JVMExec.exec.simps)
 apply (unfold Let-def)
 apply (subgoal-tac fst (snd (snd (snd (snd (the (method' (P', safeP) sigSafe-
Main)))))) !
                  (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc
(Suc (Suc (Suc (Suc
                  (Suc (Suc (the (cdm (l, i))))))))))))))))))))) =
                  Store 3) apply (simp (no-asm-simp))
apply (erule nth-via-drop-append)


apply (drule drop-Suc-append)
apply (subgoal-tac P'⊢ (None,sha,h,inih,
      [([Addr l'a],
       vs[Suc 0 := the (S' (nat (n − int off))),
            6 := Addr laa,
            2 := the (fs-S'-n-minus-off (tagGf, cellC)),
            3 := the (fs-consTableS-S'-n-minus-off-tagGf (tagLf, consDataC))],
       safeP, sigSafeMain,
       the (cdm (l, i)) + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1
+ 1 + 1 + 1 + 1 + 1 + 1 + 1,
       ag, bd)])
    −jvm→
      (None,sha,h,inih,
      [([Intg nargs'],
       vs[Suc 0 := the (S' (nat (n − int off))),
            6 := Addr laa,
            2 := the (fs-S'-n-minus-off (tagGf, cellC)),
            3 := the (fs-consTableS-S'-n-minus-off-tagGf (tagLf, consDataC))],
       safeP, sigSafeMain,
       the (cdm (l, i)) + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1
+ 1 + 1 + 1 + 1 + 1 + 1 + 1,
```

ag, *bd*)]))
apply (*unfold exec-all-def*)
apply (*simp del*: *append.append-Cons*)
apply (*erule rtrancl-trans*)
prefer *2*
apply (*rule r-into-rtrancl*)
apply (*clarify*)
apply (*unfold JVMExec.exec.simps*)
apply (*unfold Let-def*)
apply (*subgoal-tac fst* (*snd* (*snd* (*snd* (*snd* (*the* (*method'* (*P'*, *safeP*) *sigSafeMain*)))))) !
(*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc*
(*Suc* (*Suc* (*Suc* (the (cdm (*l*, *i*))))))))))))))))))) =
Getfield nargsf consDataC) **apply** (*simp* (*no-asm-simp*) *add*: *raise-system-xcpt-def*)
apply (*erule nth-via-drop-append*)

apply (*drule drop-Suc-append*)
apply (*subgoal-tac P'*⊢ (*None*,*sha*,*h*,*inih*,
[([*Intg nargs'*],
vs[*Suc 0* := *the* (*S'* (*nat* (*n* − *int off*))),
6 := *Addr laa*,
2 := *the* (*fs-S'-n-minus-off* (*tagGf*, *cellC*)),
3 := *the* (*fs-consTableS-S'-n-minus-off-tagGf* (*tagLf*, *consDataC*))],
safeP, *sigSafeMain*,
the (*cdm* (*l*, *i*)) + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1*,
ag, *bd*)])
−*jvm*→
(*None*,*sha*,*h*,*inih*,
[([],
vs[*Suc 0* := *the* (*S'* (*nat* (*n* − *int off*))),
6 := *Addr laa*,
2 := *the* (*fs-S'-n-minus-off* (*tagGf*, *cellC*)),
3 := *the* (*fs-consTableS-S'-n-minus-off-tagGf* (*tagLf*, *consDataC*)),
4 := *the* (*fs-consTableS-S'-n-minus-off-tagGf* (*nargsf*, *consDataC*))],
safeP, *sigSafeMain*,
the (*cdm* (*l*, *i*)) + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1*,
ag, *bd*)]))
apply (*unfold exec-all-def*)
apply (*simp del*: *append.append-Cons*)
apply (*erule rtrancl-trans*)
prefer *2*
apply (*rule r-into-rtrancl*)
apply (*clarify*)
apply (*unfold JVMExec.exec.simps*)

apply (*unfold Let-def*)
apply (*subgoal-tac fst* (*snd* (*snd* (*snd* (*snd* (*the* (*method′* (*P′*, *safeP*) *sigSafe-Main*)))))) !
(*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc*
(*Suc* (*Suc* (*Suc* (*Suc* (*the* (*cdm* (*l*, *i*)))))))))))))))))))))))) =
Store 4) **apply** (*simp* (*no-asm-simp*))
apply (*erule nth-via-drop-append*)

apply (*drule drop-Suc-append*)
apply (*subgoal-tac P′⊢* (*None*,*sha*,*h*,*inih*,
[([],
vs[*Suc 0* := *the* (*S′* (*nat* (*n* − *int off*))),
6 := *Addr laa*,
2 := *the* (*fs-S′-n-minus-off* (*tagGf*, *cellC*)),
3 := *the* (*fs-consTableS-S′-n-minus-off-tagGf* (*tagLf*, *consDataC*)),
4 := *Intg nargs′*],
safeP, *sigSafeMain*,
the (*cdm* (*l*, *i*)) + *1*,
ag, *bd*)])
−*jvm*→
(*None*,*sha*,*h*,*inih*,
[([*Intg 0*],
vs[*Suc 0* := *the* (*S′* (*nat* (*n* − *int off*))),
6 := *Addr laa*,
2 := *the* (*fs-S′-n-minus-off* (*tagGf*, *cellC*)),
3 := *the* (*fs-consTableS-S′-n-minus-off-tagGf* (*tagLf*, *consDataC*)),
4 := *Intg nargs′*],
safeP, *sigSafeMain*,
the (*cdm* (*l*, *i*)) + *1*,
ag, *bd*)]))
apply (*unfold exec-all-def*)
apply (*simp del*: *append.append-Cons*)
apply (*erule rtrancl-trans*)
prefer *2*
apply (*rule r-into-rtrancl*)
apply (*clarify*)
apply (*unfold JVMExec.exec.simps*)
apply (*unfold Let-def*)
apply (*subgoal-tac fst* (*snd* (*snd* (*snd* (*snd* (*the* (*method′* (*P′*, *safeP*) *sigSafe-Main*)))))) !
(*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc*
(*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*the* (*cdm* (*l*, *i*))))))))))))))))))))))))) =
LitPush (*Intg 0*)) **apply** (*simp* (*no-asm-simp*))
apply (*erule nth-via-drop-append*)

apply (*drule drop-Suc-append*)
apply (*subgoal-tac P'⊢ (None,sha,h,inih,*
[([*Intg 0*],
vs[*Suc 0* := *the* (*S'* (*nat* (*n* − *int off*))),
6 := *Addr laa*,
2 := *the* (*fs-S'-n-minus-off* (*tagGf*, *cellC*)),
3 := *the* (*fs-consTableS-S'-n-minus-off-tagGf* (*tagLf*, *consDataC*)),
4 := *Intg nargs'*],
safeP, *sigSafeMain*,
the (*cdm* (*l*, *i*)) + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1*
+ *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1*,
ag, *bd*)])
−*jvm*→
(*None*,*sha*,*h*,*inih*,
[([],
vs[*Suc 0* := *the* (*S'* (*nat* (*n* − *int off*))),
6 := *Addr laa*,
2 := *the* (*fs-S'-n-minus-off* (*tagGf*, *cellC*)),
3 := *the* (*fs-consTableS-S'-n-minus-off-tagGf* (*tagLf*, *consDataC*)),
4 := *Intg nargs'*,
5 := *Intg 0*],
safeP, *sigSafeMain*,
the (*cdm* (*l*, *i*)) + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1*
+ *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1*,
ag, *bd*)]))
apply (*unfold exec-all-def*)
apply (*simp del*: *append.append-Cons*)
apply (*erule rtrancl-trans*)
prefer *2*
apply (*rule r-into-rtrancl*)
apply (*clarify*)
apply (*unfold JVMExec.exec.simps*)
apply (*unfold Let-def*)
apply (*subgoal-tac fst* (*snd* (*snd* (*snd* (*snd* (*the* (*method'* (*P'*, *safeP*) *sigSafe-Main*)))))) !
(*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc*
(*Suc* (*Suc* (*Suc* (*Suc*
(*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*the* (*cdm* (*l*, *i*))))))))))))))))))))))))))
=
Store 5) **apply** (*simp* (*no-asm-simp*))
apply (*erule nth-via-drop-append*)

apply (*drule drop-Suc-append*)
apply (*subgoal-tac P'⊢ (None,sha,h,inih,*
[([],
vs[*Suc 0* := *the* (*S'* (*nat* (*n* − *int off*))),

```
            6 := Addr laa,
            2 := the (fs-S'-n-minus-off (tagGf, cellC)),
            3 := the (fs-consTableS-S'-n-minus-off-tagGf (tagLf, consDataC)),
            4 := Intg nargs',
            5 := Intg 0],
        safeP, sigSafeMain,
        the (cdm (l, i)) + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1
+ 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1,
        ag, bd)])
     −jvm→
       (None,sha,h,inih,
       [([Intg n],
        vs[Suc 0 := the (S' (nat (n − int off))),
            6 := Addr laa,
            2 := the (fs-S'-n-minus-off (tagGf, cellC)),
            3 := the (fs-consTableS-S'-n-minus-off-tagGf (tagLf, consDataC)),
            4 := Intg nargs',
            5 := Intg 0],
        safeP, sigSafeMain,
        the (cdm (l, i)) + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1
+ 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1,
        ag, bd)]))
apply (unfold exec-all-def)
apply (simp del: append.append-Cons)
apply (erule rtrancl-trans)
prefer 2
 apply (rule r-into-rtrancl)
 apply (clarify)
 apply (unfold JVMExec.exec.simps)
 apply (unfold Let-def)
 apply (subgoal-tac fst (snd (snd (snd (snd (the (method' (P', safeP) sigSafe-
Main)))))) !
                    (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc
(Suc (Suc (Suc (Suc
              (Suc (Suc (Suc (Suc (Suc (Suc (Suc (the (cdm (l, i))))))))))))))))))))))))
=
                       Getstatic topf stackC) apply (simp (no-asm-simp))
apply (erule nth-via-drop-append)


apply (drule drop-Suc-append)
apply (subgoal-tac P'⊢ (None,sha,h,inih,
       [([Intg n],
        vs[Suc 0 := the (S' (nat (n − int off))),
            6 := Addr laa,
            2 := the (fs-S'-n-minus-off (tagGf, cellC)),
            3 := the (fs-consTableS-S'-n-minus-off-tagGf (tagLf, consDataC)),
            4 := Intg nargs',
            5 := Intg 0],
```

```
        safeP, sigSafeMain,
        the (cdm (l, i)) + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1
+ 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1,
        ag, bd)])
      −jvm→
        (None,sha,h,inih,
        [([Intg nargs′,
          Intg n],
        vs[Suc 0 := the (S′ (nat (n − int off))),
              6 := Addr laa,
              2 := the (fs-S′-n-minus-off (tagGf, cellC)),
              3 := the (fs-consTableS-S′-n-minus-off-tagGf (tagLf, consDataC)),
              4 := Intg nargs′,
              5 := Intg 0],
        safeP, sigSafeMain,
        the (cdm (l, i)) + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1
+ 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1,
        ag, bd)]))
apply (unfold exec-all-def)
apply (simp del: append.append-Cons)
apply (erule rtrancl-trans)
prefer 2
 apply (rule r-into-rtrancl)
 apply (clarify)
 apply (unfold JVMExec.exec.simps)
 apply (unfold Let-def)
 apply (subgoal-tac fst (snd (snd (snd (snd (the (method′ (P′, safeP) sigSafe-
Main)))))) !
                    (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc
(Suc (Suc (Suc (Suc
                                (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (the (cdm (l,
i))))))))))))))))))))))))))) =
                   Load 4) apply (simp (no-asm-simp))
apply (erule nth-via-drop-append)


apply (drule drop-Suc-append)
apply (subgoal-tac P′⊢ (None,sha,h,inih,
       [([Intg nargs′,
          Intg n],
        vs[Suc 0 := the (S′ (nat (n − int off))),
              6 := Addr laa,
              2 := the (fs-S′-n-minus-off (tagGf, cellC)),
              3 := the (fs-consTableS-S′-n-minus-off-tagGf (tagLf, consDataC)),
              4 := Intg nargs′,
              5 := Intg 0],
        safeP, sigSafeMain,
        the (cdm (l, i)) + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1
+ 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1,
```

```
        ag, bd)])
      −jvm→
        (None,sha,h,inih,
        [([Intg (n + nargs')],
         vs[Suc 0 := the (S' (nat (n − int off))),
               6 := Addr laa,
               2 := the (fs-S'-n-minus-off (tagGf, cellC)),
               3 := the (fs-consTableS-S'-n-minus-off-tagGf (tagLf, consDataC)),
               4 := Intg nargs',
               5 := Intg 0],
         safeP, sigSafeMain,
         the (cdm (l, i)) + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1
+ 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1,
         ag, bd)]))
apply (unfold exec-all-def)
apply (simp del: append.append-Cons)
apply (erule rtrancl-trans)
prefer 2
 apply (rule r-into-rtrancl)
 apply (clarify)
 apply (unfold JVMExec.exec.simps)
 apply (unfold Let-def)
 apply (subgoal-tac fst (snd (snd (snd (snd (the (method' (P', safeP) sigSafe-
Main)))))) !
                    (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc
(Suc (Suc (Suc (Suc
                     (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (the (cdm (l,
i))))))))))))))))))))))))))) =
                   BinOp Add) apply (simp (no-asm-simp))
apply (erule nth-via-drop-append)


apply (drule drop-Suc-append)
apply (subgoal-tac P'⊢ (None,sha,h,inih,
       [([Intg (n + nargs')],
        vs[Suc 0 := the (S' (nat (n − int off))),
               6 := Addr laa,
               2 := the (fs-S'-n-minus-off (tagGf, cellC)),
               3 := the (fs-consTableS-S'-n-minus-off-tagGf (tagLf, consDataC)),
               4 := Intg nargs',
               5 := Intg 0],
        safeP, sigSafeMain,
        the (cdm (l, i)) + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1
+ 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1,
        ag, bd)])
      −jvm→
        (None,
        sha((stackC, topf) ↦ Intg (n + nargs')),
        h,inih,
```

```
        [([],
         vs[Suc 0 := the (S' (nat (n − int off))),
               6 := Addr laa,
               2 := the (fs-S'-n-minus-off (tagGf, cellC)),
               3 := the (fs-consTableS-S'-n-minus-off-tagGf (tagLf, consDataC)),
               4 := Intg nargs',
               5 := Intg 0],
         safeP, sigSafeMain,
         the (cdm (l, i)) + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1
+ 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1,
         ag, bd)]))
apply (unfold exec-all-def)
apply (simp del: append.append-Cons)
apply (erule rtrancl-trans)
prefer 2
 apply (rule r-into-rtrancl)
 apply (clarify)
 apply (unfold JVMExec.exec.simps)
 apply (unfold Let-def)
 apply (subgoal-tac fst (snd (snd (snd (snd (the (method' (P', safeP) sigSafe-
Main)))))) !
                    (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc
(Suc (Suc (Suc (Suc
                    (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (the (cdm
(l, i))))))))))))))))))))))))))) =
                   Putstatic topf stackC) apply (simp (no-asm-simp))
apply (erule nth-via-drop-append)



apply (drule drop-Suc-append)
apply (drule drop-append-3)
apply (subgoal-tac
   0 ≤ the-Intg (vs!5) ∧
   0 ≤ the-Intg (the (sh (stackC, topf))) − the-Intg (vs!4) ∧
   the-Intg (vs!5) ≤ the-Intg (vs!4))
apply (elim conjE)
prefer 2 apply (rule MATCH-bounds)
apply (frule-tac inih=inih and
     ref=(ag,bd) and
     nargs=nargs' and
     i =0 and
     l =laa and
     h = h and
     ty = ty and
     m = m and
     S' = S' and
     vs=vs[Suc 0 := the (S' (nat (n − int off))),
               6 := Addr laa,
```

```
            2 := the (fs-S'-n-minus-off (tagGf, cellC)),
             3 := the (fs-consTableS-S'-n-minus-off-tagGf (tagLf, consDataC))]
and
    sh=sha((stackC, topf) ↦ Intg (n + nargs'))
    in execSVMInstr-MATCH-loop-state)
apply (simp,simp)
apply (simp add: Sf-def add: topf-def)
apply (simp,simp,simp,simp,simp,simp,simp,simp)
apply (unfold exec-all-def)
apply (simp,erule rtrancl-trans)


apply (simp del: append.append-Cons
          add: Match12-def add: Sf-def add: topf-def)
apply (fold topf-def)
apply (subgoal-tac
  P'⊢ (None,
       sha((stackC, topf) ↦ Intg (n + nargs')),
        h(la ↦
          Arr ty m
           (S' ++ map-of (zip [Suc (nat n)..<nat (n + nargs' + 1)]
                (map (λj. argCell fs-S'-n-minus-off (Suc j)) [nat nargs'>..0])))),
       inih,
       [([],
        vs[Suc 0 := the (S' (nat (n − int off))),
              6 := Addr laa,
              2 := the (fs-S'-n-minus-off (tagGf, cellC)),
              3 := the (fs-consTableS-S'-n-minus-off-tagGf (tagLf, consDataC)),
              4 := Intg nargs',
              5 := Intg nargs'],
        safeP, sigSafeMain,
        the (cdm (l, i)) + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1
+ 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1,
        ag, bd)])
     −jvm→
       (None,
       sha((stackC, topf) ↦ Intg (n + nargs')),
        h(la ↦
          Arr ty m
           (S' ++ map-of (zip [Suc (nat n)..<nat (n + nargs' + 1)]
                (map (λj. argCell fs-S'-n-minus-off (Suc j)) [nat nargs'>..0])))),
       inih,
       [([Intg nargs'],
        vs[Suc 0 := the (S' (nat (n − int off))),
              6 := Addr laa,
              2 := the (fs-S'-n-minus-off (tagGf, cellC)),
              3 := the (fs-consTableS-S'-n-minus-off-tagGf (tagLf, consDataC)),
              4 := Intg nargs',
              5 := Intg nargs'],
```

safeP, *sigSafeMain*,
the (*cdm* (*l*, *i*)) + *1*
+ *1*,
ag, *bd*)]))
apply (*unfold exec-all-def*)
apply (*simp del*: *append.append-Cons*)
apply (*erule rtrancl-trans*)
prefer *2*
apply (*rule r-into-rtrancl*)
apply (*clarify*)
apply (*unfold JVMExec.exec.simps*)
apply (*unfold Let-def*)
apply (*subgoal-tac fst* (*snd* (*snd* (*snd* (*snd* (*the* (*method′* (*P′*, *safeP*) *sigSafeMain*)))))) !
(*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc*
(*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*the* (*cdm* (*l*, *i*))))))))))))))))))))))))))))))) =
Load 4) **apply** (*simp* (*no-asm-simp*))
apply (*simp*, *erule nth-via-drop*)

apply (*drule drop-Suc-append*)
apply (*subgoal-tac*
P′⊢ (*None*,
sha((*stackC*, *topf*) ↦ *Intg* (*n* + *nargs′*)),
h(*la* ↦
Arr ty m
(*S′* ++ *map-of* (*zip* [*Suc* (*nat n*)..<*nat* (*n* + *nargs′* + *1*)]
(*map* (λ*j*. *argCell fs-S′-n-minus-off* (*Suc j*)) [*nat nargs′*>..*0*])))),
inih,
[([*Intg nargs′*],
vs[*Suc 0* := *the* (*S′* (*nat* (*n* − *int off*))),
6 := *Addr laa*,
2 := *the* (*fs-S′-n-minus-off* (*tagGf*, *cellC*)),
3 := *the* (*fs-consTableS-S′-n-minus-off-tagGf* (*tagLf*, *consDataC*)),
4 := *Intg nargs′*,
5 := *Intg nargs′*],
safeP, *sigSafeMain*,
the (*cdm* (*l*, *i*)) + *1*
+ *1*,
ag, *bd*)])
−*jvm*→
(*None*,
sha((*stackC*, *topf*) ↦ *Intg* (*n* + *nargs′*)),
h(*la* ↦
Arr ty m

```
          (S' ++ map-of (zip [Suc (nat n)..<nat (n + nargs' + 1)]
               (map (λj. argCell fs-S'-n-minus-off (Suc j)) [nat nargs'>..0])))),
       inih,
       [([Intg nargs',
          Intg nargs'],
        vs[Suc 0 := the (S' (nat (n − int off))),
             6 := Addr laa,
             2 := the (fs-S'-n-minus-off (tagGf, cellC)),
             3 := the (fs-consTableS-S'-n-minus-off-tagGf (tagLf, consDataC)),
             4 := Intg nargs',
             5 := Intg nargs'],
        safeP, sigSafeMain,
        the (cdm (l, i)) + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1
+ 1 + 1 + 1 + 1 + 1 +
        1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1,
        ag, bd)]))
apply (unfold exec-all-def)
apply (simp del: append.append-Cons)
apply (erule rtrancl-trans)
prefer 2
 apply (rule r-into-rtrancl)
 apply (clarify)
 apply (unfold JVMExec.exec.simps)
 apply (unfold Let-def)
 apply (subgoal-tac fst (snd (snd (snd (snd (the (method' (P', safeP) sigSafe-
Main)))))) !
                    (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc
(Suc (Suc (Suc (Suc
                    (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc
(the (cdm (l, i))))))))))))))))))))))))))) =
                   Load 5) apply (simp (no-asm-simp))
apply (simp, erule nth-via-drop)


apply (drule drop-Suc-append)
apply (subgoal-tac
  P'⊢ (None,
       sha((stackC, topf) ↦ Intg (n + nargs')),
        h(la ↦
          Arr ty m
           (S' ++ map-of (zip [Suc (nat n)..<nat (n + nargs' + 1)]
                (map (λj. argCell fs-S'-n-minus-off (Suc j)) [nat nargs'>..0])))),
       inih,
       [([Intg nargs',
          Intg nargs'],
        vs[Suc 0 := the (S' (nat (n − int off))),
             6 := Addr laa,
             2 := the (fs-S'-n-minus-off (tagGf, cellC)),
             3 := the (fs-consTableS-S'-n-minus-off-tagGf (tagLf, consDataC)),
```

```
            4 := Intg nargs',
            5 := Intg nargs'],
       safeP, sigSafeMain,
       the (cdm (l, i)) + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1
+ 1 + 1 + 1 + 1 + 1 + 1 +
       1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1,
       ag, bd)])
    −jvm→
      (None,
       sha((stackC, topf) ↦ Intg (n + nargs')),
       h(la ↦
          Arr ty m
           (S' ++ map-of (zip [Suc (nat n)..<nat (n + nargs' + 1)]
                (map (λj. argCell fs-S'-n-minus-off (Suc j)) [nat nargs'>..0])))),
       inih,
       [([],
        vs[Suc 0 := the (S' (nat (n − int off))),
            6 := Addr laa,
            2 := the (fs-S'-n-minus-off (tagGf, cellC)),
            3 := the (fs-consTableS-S'-n-minus-off-tagGf (tagLf, consDataC)),
            4 := Intg nargs',
            5 := Intg nargs'],
        safeP, sigSafeMain,
        the (cdm (l, i)) + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1
+ 1 + 1 + 1 + 1 + 1 + 1 +
          1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + nat
labelEndLoop,
        ag, bd)]))
apply (unfold exec-all-def)
apply (simp del: append.append-Cons)
apply (erule rtrancl-trans)
prefer 2
 apply (rule r-into-rtrancl)
 apply (clarify)
 apply (unfold JVMExec.exec.simps)
 apply (unfold Let-def)
 apply (subgoal-tac fst (snd (snd (snd (snd (the (method' (P', safeP) sigSafe-
Main)))))) !
                    (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc
(Suc (Suc (Suc (Suc (Suc
                    (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc
(the (cdm (l, i))))))))))))))))))))))))))))) =
                    Ifcmp GreaterEqual labelEndLoop) apply (simp (no-asm-simp))
apply (simp add: labelEndLoop-def)
apply (simp, erule nth-via-drop)


apply (frule go-End)
apply (subgoal-tac
```

```
  P⊢ (None,
       sha((stackC, topf) ↦ Intg (n + nargs′)),
        h(la ↦
          Arr ty m
           (S′ ++ map-of (zip [Suc (nat n)..<nat (n + nargs′ + 1)]
                 (map (λj. argCell fs-S′-n-minus-off (Suc j)) [nat nargs′>..0])))),
       inih,
       [([],
        vs[Suc 0 := the (S′ (nat (n − int off))),
             6 := Addr laa,
             2 := the (fs-S′-n-minus-off (tagGf, cellC)),
             3 := the (fs-consTableS-S′-n-minus-off-tagGf (tagLf, consDataC)),
             4 := Intg nargs′,
             5 := Intg nargs′],
        safeP, sigSafeMain,
        the (cdm (l, i)) + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1
+ 1 + 1 + 1 +
         1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + nat
labelEndLoop,
        ag, bd)])
     −jvm→
       (None,
       sha((stackC, topf) ↦ Intg (n + nargs′)),
        h(la ↦
          Arr ty m
           (S′ ++ map-of (zip [Suc (nat n)..<nat (n + nargs′ + 1)]
                 (map (λj. argCell fs-S′-n-minus-off (Suc j)) [nat nargs′>..0])))),
       inih,
       [([the (fs-consTableS-S′-n-minus-off-tagGf (tagLf, consDataC))],
        vs[Suc 0 := the (S′ (nat (n − int off))),
             6 := Addr laa,
             2 := the (fs-S′-n-minus-off (tagGf, cellC)),
             3 := the (fs-consTableS-S′-n-minus-off-tagGf (tagLf, consDataC)),
             4 := Intg nargs′,
             5 := Intg nargs′],
        safeP, sigSafeMain,
        the (cdm (l, i)) + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1
+ 1 + 1 + 1 +
         1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + nat
labelEndLoop + 1,
        ag, bd)]))
apply (unfold exec-all-def)
apply (simp del: append.append-Cons)
apply (erule rtrancl-trans)
prefer 2
 apply (rule r-into-rtrancl)
 apply (clarify)
 apply (unfold JVMExec.exec.simps)
 apply (unfold Let-def)
```

apply (*subgoal-tac* (*fst* (*snd* (*snd* (*snd* (*snd* (*the* (*method'* (*P'*, *safeP*) *sigSafe-Main*)))))) !
Suc (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc*
(*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc*
(*Suc* (*Suc* (*Suc* (*the* (*cdm* (*l*, *i*)) + *nat labelEndLoop*))))))))))))))))))))))))))))))
=
Load 3) **apply** (*simp* (*no-asm-simp*))
apply (*simp*, *erule nth-via-drop*)

apply (*thin-tac drop ?x ?xs = ?ys*) **thm** *drop-Suc-2*
apply (*drule drop-Suc-2*)

apply (*subgoal-tac*
P'⊢ (*None*,
sha((*stackC*, *topf*) ↦ *Intg* (*n* + *nargs'*)),
h(*la* ↦
Arr ty m
(*S'* ++ *map-of* (*zip* [*Suc* (*nat n*)..<*nat* (*n* + *nargs'* + *1*)]
(*map* (λ*j*. *argCell fs-S'-n-minus-off* (*Suc j*)) [*nat nargs'*>..*0*])))),
inih,
[([*the* (*fs-consTableS-S'-n-minus-off-tagGf* (*tagLf*, *consDataC*))],
vs[*Suc 0* := *the* (*S'* (*nat* (*n* − *int off*))),
6 := *Addr laa*,
2 := *the* (*fs-S'-n-minus-off* (*tagGf*, *cellC*)),
3 := *the* (*fs-consTableS-S'-n-minus-off-tagGf* (*tagLf*, *consDataC*)),
4 := *Intg nargs'*,
5 := *Intg nargs'*],
safeP, *sigSafeMain*,
the (*cdm* (*l*, *i*)) + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* +
1 + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *1* + *nat labelEndLoop* + *1*,
ag, *bd*)])
−*jvm*→
(*None*, *sha*((*stackC*, *topf*) ↦ *Intg* (*n* + *nargs'*)), *h*(*la* ↦
Arr ty m
(*S'* ++ *map-of* (*zip* [*Suc* (*nat n*)..<*nat* (*n* + *nargs'* + *1*)] (*map* (λ*j*. *argCell fs-S'-n-minus-off* (*Suc j*)) [*nat nargs'*>..*0*])))),
inih, [([], *vs*[*Suc 0* := *the* (*S'* (*nat* (*n* − *int off*))), *6* := *Addr laa*, *2* := *the* (*fs-S'-n-minus-off* (*tagGf*, *cellC*)),
3 := *the* (*fs-consTableS-S'-n-minus-off-tagGf* (*tagLf*, *consDataC*)), *4* := *Intg nargs'*, *5* := *Intg nargs'*],
safeP, *sigSafeMain*,
Suc (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc*
(*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc*
(*Suc* (*Suc* (*Suc* (*the* (*cdm* (*l*, *i*)) + *nat labelEndLoop* +
nat (*map* (λ*n*. *trAddr n* (*the* (*cdm* (*l*, *i*)) + *incMatch*)) (*map* (λ*p*. *the* (*cdm'* (*p*, *0*))) *ps*) !

```
                    nat (the-Intg (the (fs-consTableS-S'-n-minus-off-tagGf (tagLf,
consDataC))))))))))))))))))))))))))))))))))),
                ag, bd)]))
apply (unfold exec-all-def)
apply (simp del: append.append-Cons)
apply simp

apply (rule r-into-rtrancl)
 apply (clarify)
 apply (unfold JVMExec.exec.simps)
 apply (unfold Let-def)
 apply (subgoal-tac (fst (snd (snd (snd (snd (the (method' (P', safeP) sigSafe-
Main)))))) !
            Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc
             (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc
(Suc
        (Suc (Suc (Suc (Suc (the (cdm (l, i)) + nat labelEndLoop))))))))))))))))))))))))))))
=
            Tableswitch 0 (int (length ps - Suc 0))
                    (map (λn. trAddr n (the (cdm (l, i)) + incMatch))
                  (map (λp. the (cdm' (p, 0))) ps))) apply (simp (no-asm-simp))
apply (subgoal-tac
  ¬ the-Intg (the (fs-consTableS-S'-n-minus-off-tagGf (tagLf, consDataC))) < 0
∧
  ¬ int (length ps - Suc 0) < the-Intg (the (fs-consTableS-S'-n-minus-off-tagGf
(tagLf, consDataC))),simp)
apply (rule tag-bounds)
apply (simp, erule nth-via-drop)



apply (rule-tac x=vs[Suc 0 := the (S' (nat (n - int off))),
                     6 := Addr laa,
                     2 := the (fs-S'-n-minus-off (tagGf, cellC)),
                          3 := the (fs-consTableS-S'-n-minus-off-tagGf (tagLf,
consDataC)),
                     4 := Intg nargs',
                     5 := Intg nargs'] in exI)
apply (rule-tac x=Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc
(Suc
                  (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc
(Suc
                 (Suc (Suc (Suc (Suc (the (cdm (l, i)) + nat labelEndLoop +
                  nat (map (λn. trAddr n (the (cdm (l, i)) + incMatch))
                     (map (λp. the (cdm' (p, 0))) ps) !
                   nat (the-Intg (the (fs-consTableS-S'-n-minus-off-tagGf (tagLf,
consDataC))
                  ))))))))))))))))))))))))))))))))) in exI)
apply (rule conjI)
```

apply (*rule-tac x=ag* **in** *exI*)
apply (*rule-tac x=bd* **in** *exI*)
apply (*rule refl*)
apply (*rule-tac x=la* **in** *exI*)
apply (*rule conjI*, *simp add*: *Sf-def add*: *topf-def*)
apply (*rule-tac x=ty* **in** *exI*)
apply (*rule-tac x=m* **in** *exI*)
apply (*rule-tac x=S' ++ map-of* (*zip* [*Suc* (*nat n*)..<*nat* (*n* + *nargs'* + *1*)]
(*map* (λ*j*. *argCell fs-S'-n-minus-off* (*Suc j*))
[*nat nargs'*>..*0*])) **in** *exI*)
apply (*rule-tac x=*(*n* + *nargs'*) **in** *exI*)
apply (*rule-tac x=l'* **in** *exI*)
apply (*rule conjI*, *assumption*)
apply (*rule-tac x=regS* **in** *exI*)
apply (*rule-tac x=l''* **in** *exI*)
apply (*rule conjI*, *assumption*)
apply (*rule conjI*, *assumption*)
apply (*rule conjI*) **apply** (*rule activeCells*)
apply (*simp add*: *heapC-def add*: *stackC-def*)
apply (*rule conjI*) **apply** (*rule activeCells-2*)
apply (*simp add*: *heapC-def add*: *stackC-def*,*simp*)
apply (*rule conjI*) **apply** (*rule activeCells-2*)
apply (*simp add*: *heapC-def add*: *stackC-def*,*simp*)
apply (*rule conjI*, *simp*)
apply (*rule conjI*, *simp*)
apply (*rule conjI*, *simp add*: *heapC-def add*: *stackC-def*)
apply (*rule conjI*)
apply (*rule-tac x=m'* **in** *exI*,*simp*)
apply (*rule conjI*, *simp add*: *dirCellC-def add*: *stackC-def*)
apply (*rule-tac x=m''* **in** *exI*)
apply (*rule-tac x=d* **in** *exI*)
apply (*rule conjI*,*simp*)
apply (*rule-tac x=g* **in** *exI*,*simp add*: *heapC-def add*: *stackC-def*)
apply (*fold heapC-def*, *fold stackC-def*)
apply (*rule conjI*) **apply** (*rule equivH-MATCH*,*simp*,*assumption+*,*force*)
apply (*rule conjI*) **apply** (*rule maxPush-MATCH*,*assumption*)
apply (*rule conjI*) **apply** (*rule equivS-MATCH-axiom*)
apply (*rule pc-MATCH*)
done

end

26 Main correctness theorem of the translation SVM to JVM

theory *mainTheoremSVM2JVM*
imports *CertifSVM2JVM dem-translation dem-PUSHCONT dem-POPCONT dem-PRIMOP*

dem-SLIDE dem-MATCHN dem-BUILDENV dem-CALL dem-DECREGION
dem-BUILDCLS dem-REUSE dem-MATCH
begin

This theorem certifies that, whenever the SVM and the JVM are started in equivalent states, the SVM executes its next instruction, and the JVM executes its translation to bytecode, then both machines arive at equivalent states.

theorem *correctSVM2JVM* :
⟦ (*P'*,*cdm*,*ctm*,*com*) = *trSVM2JVM P*;
cdm, *ctm*, *com* ⊢ *S1* ≜ *S1'*;
execSVM P S1 = *Either.Right S2*
⟧ ⟹
∃ *S2'* . *P'* ⊢ *S1'* −*jvm*→ *S2'* ∧ *cdm*, *ctm*, *com* ⊢ *S2* ≜ *S2'*

apply (*case-tac P*)
apply (*rename-tac cs triple*)
apply (*case-tac cs*)
apply (*rename-tac svms ctmap*)
apply (*case-tac triple*)
apply (*rename-tac ini pair*)
apply (*case-tac pair*)
apply (*rename-tac ct st*)
apply (*unfold execSVM-def*)
apply (*case-tac S1*)
apply (*rename-tac h rest1*)
apply (*case-tac rest1*)
apply (*rename-tac k0 rest2*)
apply (*case-tac rest2*)
apply (*rename-tac pc S*)
apply (*case-tac pc*)
apply (*rename-tac l i*)
apply (*case-tac h*)
apply (*rename-tac hm k*)
apply (*clarsimp*)

apply (*subgoal-tac l* : *dom* (*map-of svms*)
∧ *i* < *length* (*fst* (*the* (*map-of svms l*))))
prefer *2* **apply** (*erule PC-good*,*simp*)
apply (*elim conjE*)
apply (*subgoal-tac* ∃ *j* .
j < *length svms* ∧ (*l*,*the* (*map-of svms l*)) = *svms* ! *j*)
prefer *2* **apply** (*rule map-of-distinct2*) **apply** *force*
apply (*erule exE*,*elim conjE*)
apply (*case-tac the* (*map-of svms l*),*rename-tac seq fn*,*clarify*)

apply (*subgoal-tac* (∃ *cdm' ctm' pcc inss bytecode'* .
inss = *trInstr* (*the* (*cdm* (*l*,*i*))) *cdm' ctm' com pcc*
(*fst* (*the* (*map-of svms l*)) ! *i*)
∧ *drop* (*the* (*cdm* (*l*,*i*))) (*extractBytecode P'*) = *inss* @ *bytecode'*
∧ *cdm'* ⊆$_m$ *cdm*))
prefer *2* **apply** (*rule fun-SVM2JVM*)
apply *simp*
apply *simp*
apply (*rule sym*,*simp*)
apply *simp*
apply *simp*
apply *simp*
apply *simp*

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

apply (*case-tac fst* (*the* (*map-of svms l*)) ! *i*)

apply (*erule execSVMInstr-DECREGION*)
apply (*simp*,*simp*,*simp*,*force*)

apply (*erule execSVMInstr-POPCONT*)
apply (*simp*,*simp*,*simp*,*force*)

apply (*erule execSVMInstr-PUSHCONT*)
apply (*simp*,*simp*,*simp*,*force*)

apply (*erule execSVMInstr-COPY*)
apply (*simp*,*simp*,*simp*,*force*)

apply (*erule execSVMInstr-REUSE*)
apply (*simp*,*simp*,*simp*,*force*)

apply (*erule execSVMInstr-CALL*)
apply (*simp*,*simp*,*simp*,*force*,*simp*)

apply (*erule execSVMInstr-PRIMOP*)
apply (*simp*,*simp*,*simp*,*force*)

apply (*erule execSVMInstr-MATCH*)
apply (*simp*,*force*,*simp*,*force*,*simp*)

apply (*erule execSVMInstr-MATCHD*)
apply (*simp*,*force*,*simp*,*force*)

apply (*erule execSVMInstr-MATCHN*)
apply (*simp*,*force*,*simp*,*force*,*simp*)

apply (*erule execSVMInstr-BUILDENV*)
apply (*simp*,*simp*,*simp*,*force*)

apply (*erule execSVMInstr-BUILDCLS*)
apply (*simp*,*force*,*simp*,*force*)

apply (*frule-tac m*=*nat1* **and**
n=*nat2* **in** *execSVMInstr-SLIDE*)
apply (*simp*+)
done

end