

WebAssembly

Conrad Watt

February 6, 2026

Abstract

This is a mechanised specification of the WebAssembly language, drawn mainly from the previously published paper formalisation [1]. Also included is a full proof of soundness of the type system, together with a verified type checker and interpreter. We include only a partial procedure for the extraction of the type checker and interpreter here. For more details, please see our paper [2].

Contents

1	WebAssembly Core AST	2
2	Syntactic Typeclasses	5
3	WebAssembly Base Definitions	7
4	Host Properties	24
5	Auxiliary Type System Properties	25
6	Lemmas for Soundness Proof	54
	6.1 Preservation	54
	6.2 Progress	92
7	Soundness Theorems	126
8	Augmented Type Syntax for Concrete Checker	127
9	Executable Type Checker	150
10	Correctness of Type Checker	154
	10.1 Soundness	154
	10.2 Completeness	171
11	WebAssembly Interpreter	189

1 WebAssembly Core AST

```

theory Wasm-Ast
  imports
    Main
    Native-Word.Uint8
    Word-Lib.Reversed-Bit-Lists
  begin

  type-synonym — immediate
    i = nat
  type-synonym — static offset
    off = nat
  type-synonym — alignment exponent
    a = nat

  — primitive types
  typedecl i32
  typedecl i64
  typedecl f32
  typedecl f64

  — memory
  type-synonym byte = uint8

  typedef bytes = UNIV :: (byte list) set ..
  setup-lifting type-definition-bytes
  declare Quotient-bytes[transfer-rule]

  lift-definition bytes-takefill :: byte  $\Rightarrow$  nat  $\Rightarrow$  bytes  $\Rightarrow$  bytes is ( $\lambda a\ n\ as.$  takefill
(Abs-uint8 a) n as) .
  lift-definition bytes-replicate :: nat  $\Rightarrow$  byte  $\Rightarrow$  bytes is ( $\lambda n\ b.$  replicate n (Abs-uint8
b)) .
  definition msbyte :: bytes  $\Rightarrow$  byte where
    msbyte bs = last (Rep-bytes bs)

  typedef mem = UNIV :: (byte list) set ..
  setup-lifting type-definition-mem
  declare Quotient-mem[transfer-rule]

  lift-definition read-bytes :: mem  $\Rightarrow$  nat  $\Rightarrow$  nat  $\Rightarrow$  bytes is ( $\lambda m\ n\ l.$  take l (drop
n m)) .
  lift-definition write-bytes :: mem  $\Rightarrow$  nat  $\Rightarrow$  bytes  $\Rightarrow$  mem is ( $\lambda m\ n\ bs.$  (take n
m) @ bs @ (drop (n + length bs) m)) .
  lift-definition mem-append :: mem  $\Rightarrow$  bytes  $\Rightarrow$  mem is append .

  — host

```

typedecl *host*
typedecl *host-state*

datatype — value types
t = *T-i32* | *T-i64* | *T-f32* | *T-f64*

datatype — packed types
tp = *Tp-i8* | *Tp-i16* | *Tp-i32*

datatype — mutability
mut = *T-immut* | *T-mut*

record *tg* = — global types
tg-mut :: *mut*
tg-t :: *t*

datatype — function types
tf = *Tf t list t list* ($\langle - \ ' \! \rightarrow \ - \rangle$ 60)

record *t-context* =
types-t :: *tf list*
func-t :: *tf list*
global :: *tg list*
table :: *nat option*
memory :: *nat option*
local :: *t list*
label :: (*t list*) *list*
return :: (*t list*) *option*

record *s-context* =
s-inst :: *t-context list*
s-funcs :: *tf list*
s-tab :: *nat list*
s-mem :: *nat list*
s-globs :: *tg list*

datatype
sx = *S* | *U*

datatype
unop-i = *Clz* | *Ctz* | *Popcnt*

datatype
unop-f = *Neg* | *Abs* | *Ceil* | *Floor* | *Trunc* | *Nearest* | *Sqrt*

datatype
binop-i = *Add* | *Sub* | *Mul* | *Div sx* | *Rem sx* | *And* | *Or* | *Xor* | *Shl* | *Shr sx* |
Rotl | *Rotr*

datatype*binop-f = Addf | Subf | Mulf | Divf | Min | Max | Copysign***datatype***testop = Eqz***datatype***relop-i = Eq | Ne | Lt sx | Gt sx | Le sx | Ge sx***datatype***relop-f = Eqf | Nef | Ltf | Gtf | Lef | Gef***datatype***cvtop = Convert | Reinterpret***datatype** — values

v =
ConstInt32 i32
| ConstInt64 i64
| ConstFloat32 f32
| ConstFloat64 f64

datatype — basic instructions

b-e =
Unreachable
| Nop
| Drop
| Select
| Block tf b-e list
| Loop tf b-e list
| If tf b-e list b-e list
| Br i
| Br-if i
| Br-table i list i
| Return
| Call i
| Call-indirect i
| Get-local i
| Set-local i
| Tee-local i
| Get-global i
| Set-global i
| Load t (tp × sx) option a off
| Store t tp option a off
| Current-memory
| Grow-memory
| EConst v (⊢ C → 60)
| Unop-i t unop-i

```

| Unop-f t unop-f
| Binop-i t binop-i
| Binop-f t binop-f
| Testop t testop
| Relop-i t relop-i
| Relop-f t relop-f
| Cvtop t cvtop t sz option

```

datatype *cl* = — function closures

```

  Func-native i tf t list b-e list
| Func-host tf host

```

record *inst* = — instances

```

types :: tf list
funcs :: i list
tab :: i option
mem :: i option
globs :: i list

```

type-synonym *tabinst* = (*cl option*) *list*

record *global* =

```

g-mut :: mut
g-val :: v

```

record *s* = — store

```

inst :: inst list
funcs :: cl list
tab :: tabinst list
mem :: mem list
globs :: global list

```

datatype *e* = — administrative instruction

```

Basic b-e (↪ 60)
| Trap
| Callcl cl
| Label nat e list e list
| Local nat i v list e list

```

datatype *Lholed* =

```

— L0 = v* [<hole>] e*
  LBase e list e list
— L(i+1) = v* (label n e* Li) e*
| LRec e list nat e list Lholed e list

```

end

2 Syntactic Typeclasses

theory *Wasm-Type-Abs* **imports** *Main* **begin**

```

class wasm-base = zero

class wasm-int = wasm-base +

  fixes int-clz :: 'a ⇒ 'a
  fixes int-ctz :: 'a ⇒ 'a
  fixes int-popcnt :: 'a ⇒ 'a

  fixes int-add :: 'a ⇒ 'a ⇒ 'a
  fixes int-sub :: 'a ⇒ 'a ⇒ 'a
  fixes int-mul :: 'a ⇒ 'a ⇒ 'a
  fixes int-div-u :: 'a ⇒ 'a ⇒ 'a option
  fixes int-div-s :: 'a ⇒ 'a ⇒ 'a option
  fixes int-rem-u :: 'a ⇒ 'a ⇒ 'a option
  fixes int-rem-s :: 'a ⇒ 'a ⇒ 'a option
  fixes int-and :: 'a ⇒ 'a ⇒ 'a
  fixes int-or :: 'a ⇒ 'a ⇒ 'a
  fixes int-xor :: 'a ⇒ 'a ⇒ 'a
  fixes int-shl :: 'a ⇒ 'a ⇒ 'a
  fixes int-shr-u :: 'a ⇒ 'a ⇒ 'a
  fixes int-shr-s :: 'a ⇒ 'a ⇒ 'a
  fixes int-rotl :: 'a ⇒ 'a ⇒ 'a
  fixes int-rotr :: 'a ⇒ 'a ⇒ 'a

  fixes int-eqz :: 'a ⇒ bool

  fixes int-eq :: 'a ⇒ 'a ⇒ bool
  fixes int-lt-u :: 'a ⇒ 'a ⇒ bool
  fixes int-lt-s :: 'a ⇒ 'a ⇒ bool
  fixes int-gt-u :: 'a ⇒ 'a ⇒ bool
  fixes int-gt-s :: 'a ⇒ 'a ⇒ bool
  fixes int-le-u :: 'a ⇒ 'a ⇒ bool
  fixes int-le-s :: 'a ⇒ 'a ⇒ bool
  fixes int-ge-u :: 'a ⇒ 'a ⇒ bool
  fixes int-ge-s :: 'a ⇒ 'a ⇒ bool

  fixes int-of-nat :: nat ⇒ 'a
  fixes nat-of-int :: 'a ⇒ nat
begin
  abbreviation (input)
    int-ne where
      int-ne x y ≡ ¬ (int-eq x y)
end

class wasm-float = wasm-base +

  fixes float-neg    :: 'a ⇒ 'a
  fixes float-abs    :: 'a ⇒ 'a

```

```

fixes float-ceil :: 'a ⇒ 'a
fixes float-floor :: 'a ⇒ 'a
fixes float-trunc :: 'a ⇒ 'a
fixes float-nearest :: 'a ⇒ 'a
fixes float-sqrt :: 'a ⇒ 'a

fixes float-add :: 'a ⇒ 'a ⇒ 'a
fixes float-sub :: 'a ⇒ 'a ⇒ 'a
fixes float-mul :: 'a ⇒ 'a ⇒ 'a
fixes float-div :: 'a ⇒ 'a ⇒ 'a
fixes float-min :: 'a ⇒ 'a ⇒ 'a
fixes float-max :: 'a ⇒ 'a ⇒ 'a
fixes float-copysign :: 'a ⇒ 'a ⇒ 'a

fixes float-eq :: 'a ⇒ 'a ⇒ bool
fixes float-lt :: 'a ⇒ 'a ⇒ bool
fixes float-gt :: 'a ⇒ 'a ⇒ bool
fixes float-le :: 'a ⇒ 'a ⇒ bool
fixes float-ge :: 'a ⇒ 'a ⇒ bool
begin
  abbreviation (input)
    float-ne where
      float-ne x y ≡ ¬ (float-eq x y)
end
end

```

3 WebAssembly Base Definitions

```

theory Wasm-Base-Defs imports Wasm-Ast Wasm-Type-Abs begin

```

```

instantiation i32 :: wasm-int begin instance .. end
instantiation i64 :: wasm-int begin instance .. end
instantiation f32 :: wasm-float begin instance .. end
instantiation f64 :: wasm-float begin instance .. end

```

```

consts

```

```

ui32-trunc-f32 :: f32 ⇒ i32 option
si32-trunc-f32 :: f32 ⇒ i32 option
ui32-trunc-f64 :: f64 ⇒ i32 option
si32-trunc-f64 :: f64 ⇒ i32 option

ui64-trunc-f32 :: f32 ⇒ i64 option
si64-trunc-f32 :: f32 ⇒ i64 option
ui64-trunc-f64 :: f64 ⇒ i64 option
si64-trunc-f64 :: f64 ⇒ i64 option

f32-convert-ui32 :: i32 ⇒ f32

```

f32-convert-si32 :: *i32* ⇒ *f32*
f32-convert-ui64 :: *i64* ⇒ *f32*
f32-convert-si64 :: *i64* ⇒ *f32*

f64-convert-ui32 :: *i32* ⇒ *f64*
f64-convert-si32 :: *i32* ⇒ *f64*
f64-convert-ui64 :: *i64* ⇒ *f64*
f64-convert-si64 :: *i64* ⇒ *f64*

wasm-wrap :: *i64* ⇒ *i32*
wasm-extend-u :: *i32* ⇒ *i64*
wasm-extend-s :: *i32* ⇒ *i64*
wasm-demote :: *f64* ⇒ *f32*
wasm-promote :: *f32* ⇒ *f64*

serialise-i32 :: *i32* ⇒ *bytes*
serialise-i64 :: *i64* ⇒ *bytes*
serialise-f32 :: *f32* ⇒ *bytes*
serialise-f64 :: *f64* ⇒ *bytes*
wasm-bool :: *bool* ⇒ *i32*
int32-minus-one :: *i32*

definition *mem-size* :: *mem* ⇒ *nat* **where**
mem-size *m* = *length* (*Rep-mem* *m*)

definition *mem-grow* :: *mem* ⇒ *nat* ⇒ *mem* **where**
mem-grow *m* *n* = *mem-append* *m* (*bytes-replicate* (*n* * 64000) 0)

definition *load* :: *mem* ⇒ *nat* ⇒ *off* ⇒ *nat* ⇒ *bytes option* **where**
load *m* *n* *off* *l* = (*if* (*mem-size* *m* ≥ (*n*+*off*+*l*))
then *Some* (*read-bytes* *m* (*n*+*off*) *l*)
else *None*)

definition *sign-extend* :: *sx* ⇒ *nat* ⇒ *bytes* ⇒ *bytes* **where**
sign-extend *sx* *l* *bytes* = (*let* *msb* = *msb* (*msbyte* *bytes*) *in*
let *byte* = (*case* *sx* of *U* ⇒ 0 | *S* ⇒ *if* *msb* then -1 else 0) *in*
bytes-takefill *byte* *l* *bytes*)

definition *load-packed* :: *sx* ⇒ *mem* ⇒ *nat* ⇒ *off* ⇒ *nat* ⇒ *nat* ⇒ *bytes option*
where
load-packed *sx* *m* *n* *off* *lp* *l* = *map-option* (*sign-extend* *sx* *l*) (*load* *m* *n* *off* *lp*)

definition *store* :: *mem* ⇒ *nat* ⇒ *off* ⇒ *bytes* ⇒ *nat* ⇒ *mem option* **where**
store *m* *n* *off* *bs* *l* = (*if* (*mem-size* *m* ≥ (*n*+*off*+*l*))
then *Some* (*write-bytes* *m* (*n*+*off*) (*bytes-takefill* 0 *l* *bs*))
else *None*)

definition *store-packed* :: *mem* ⇒ *nat* ⇒ *off* ⇒ *bytes* ⇒ *nat* ⇒ *mem option*

where

store-packed = *store*

consts

wasm-deserialise :: *bytes* ⇒ *t* ⇒ *v*

host-apply :: *s* ⇒ *tf* ⇒ *host* ⇒ *v list* ⇒ *host-state* ⇒ (*s* × *v list*) *option*

definition *typeof* :: *v* ⇒ *t* **where**

typeof v = (case *v* of
 ConstInt32 - ⇒ *T-i32*
 | *ConstInt64* - ⇒ *T-i64*
 | *ConstFloat32* - ⇒ *T-f32*
 | *ConstFloat64* - ⇒ *T-f64*)

definition *option-projl* :: (*'a* × *'b*) *option* ⇒ *'a option* **where**

option-projl x = *map-option fst x*

definition *option-projr* :: (*'a* × *'b*) *option* ⇒ *'b option* **where**

option-projr x = *map-option snd x*

definition *t-length* :: *t* ⇒ *nat* **where**

t-length t = (case *t* of
 T-i32 ⇒ 4
 | *T-i64* ⇒ 8
 | *T-f32* ⇒ 4
 | *T-f64* ⇒ 8)

definition *tp-length* :: *tp* ⇒ *nat* **where**

tp-length tp = (case *tp* of
 Tp-i8 ⇒ 1
 | *Tp-i16* ⇒ 2
 | *Tp-i32* ⇒ 4)

definition *is-int-t* :: *t* ⇒ *bool* **where**

is-int-t t = (case *t* of
 T-i32 ⇒ *True*
 | *T-i64* ⇒ *True*
 | *T-f32* ⇒ *False*
 | *T-f64* ⇒ *False*)

definition *is-float-t* :: *t* ⇒ *bool* **where**

is-float-t t = (case *t* of
 T-i32 ⇒ *False*
 | *T-i64* ⇒ *False*
 | *T-f32* ⇒ *True*
 | *T-f64* ⇒ *True*)

definition *is-mut* :: *tg* ⇒ *bool* **where**

$is\text{-mut } tg = (tg\text{-mut } tg = T\text{-mut})$

definition $app\text{-unop}\text{-}i :: unop\text{-}i \Rightarrow 'i::wasm\text{-}int \Rightarrow 'i::wasm\text{-}int$ **where**

$app\text{-unop}\text{-}i\ iop\ c =$
 (case iop of
 $Ctz \Rightarrow int\text{-}ctz\ c$
 | $Clz \Rightarrow int\text{-}clz\ c$
 | $Popcnt \Rightarrow int\text{-}popcnt\ c$)

definition $app\text{-unop}\text{-}f :: unop\text{-}f \Rightarrow 'f::wasm\text{-}float \Rightarrow 'f::wasm\text{-}float$ **where**

$app\text{-unop}\text{-}f\ fop\ c =$
 (case fop of
 $Neg \Rightarrow float\text{-}neg\ c$
 | $Abs \Rightarrow float\text{-}abs\ c$
 | $Ceil \Rightarrow float\text{-}ceil\ c$
 | $Floor \Rightarrow float\text{-}floor\ c$
 | $Trunc \Rightarrow float\text{-}trunc\ c$
 | $Nearest \Rightarrow float\text{-}nearest\ c$
 | $Sqrt \Rightarrow float\text{-}sqrt\ c$)

definition $app\text{-binop}\text{-}i :: binop\text{-}i \Rightarrow 'i::wasm\text{-}int \Rightarrow 'i::wasm\text{-}int \Rightarrow ('i::wasm\text{-}int)$

option where

$app\text{-binop}\text{-}i\ iop\ c1\ c2 =$ (case iop of
 $Add \Rightarrow Some\ (int\text{-}add\ c1\ c2)$
 | $Sub \Rightarrow Some\ (int\text{-}sub\ c1\ c2)$
 | $Mul \Rightarrow Some\ (int\text{-}mul\ c1\ c2)$
 | $Div\ U \Rightarrow int\text{-}div\text{-}u\ c1\ c2$
 | $Div\ S \Rightarrow int\text{-}div\text{-}s\ c1\ c2$
 | $Rem\ U \Rightarrow int\text{-}rem\text{-}u\ c1\ c2$
 | $Rem\ S \Rightarrow int\text{-}rem\text{-}s\ c1\ c2$
 | $And \Rightarrow Some\ (int\text{-}and\ c1\ c2)$
 | $Or \Rightarrow Some\ (int\text{-}or\ c1\ c2)$
 | $Xor \Rightarrow Some\ (int\text{-}xor\ c1\ c2)$
 | $Shl \Rightarrow Some\ (int\text{-}shl\ c1\ c2)$
 | $Shr\ U \Rightarrow Some\ (int\text{-}shr\text{-}u\ c1\ c2)$
 | $Shr\ S \Rightarrow Some\ (int\text{-}shr\text{-}s\ c1\ c2)$
 | $Rotl \Rightarrow Some\ (int\text{-}rotl\ c1\ c2)$
 | $Rotr \Rightarrow Some\ (int\text{-}rotr\ c1\ c2)$)

definition $app\text{-binop}\text{-}f :: binop\text{-}f \Rightarrow 'f::wasm\text{-}float \Rightarrow 'f::wasm\text{-}float \Rightarrow ('f::wasm\text{-}float)$

option where

$app\text{-binop}\text{-}f\ fop\ c1\ c2 =$ (case fop of
 $Addf \Rightarrow Some\ (float\text{-}add\ c1\ c2)$
 | $Subf \Rightarrow Some\ (float\text{-}sub\ c1\ c2)$
 | $Mulf \Rightarrow Some\ (float\text{-}mul\ c1\ c2)$
 | $Divf \Rightarrow Some\ (float\text{-}div\ c1\ c2)$
 | $Min \Rightarrow Some\ (float\text{-}min\ c1\ c2)$
 | $Max \Rightarrow Some\ (float\text{-}max\ c1\ c2)$
 | $Copysign \Rightarrow Some\ (float\text{-}copysign\ c1\ c2)$)

definition *app-testop-i* :: *testop* ⇒ 'i::wasm-int ⇒ bool **where**

app-testop-i testop c = (case *testop* of *Eqz* ⇒ *int-eqz c*)

definition *app-relop-i* :: *relop-i* ⇒ 'i::wasm-int ⇒ 'i::wasm-int ⇒ bool **where**

app-relop-i rop c1 c2 = (case *rop* of
| *Eq* ⇒ *int-eq c1 c2*
| *Ne* ⇒ *int-ne c1 c2*
| *Lt U* ⇒ *int-lt-u c1 c2*
| *Lt S* ⇒ *int-lt-s c1 c2*
| *Gt U* ⇒ *int-gt-u c1 c2*
| *Gt S* ⇒ *int-gt-s c1 c2*
| *Le U* ⇒ *int-le-u c1 c2*
| *Le S* ⇒ *int-le-s c1 c2*
| *Ge U* ⇒ *int-ge-u c1 c2*
| *Ge S* ⇒ *int-ge-s c1 c2*)

definition *app-relop-f* :: *relop-f* ⇒ 'f::wasm-float ⇒ 'f::wasm-float ⇒ bool **where**

app-relop-f rop c1 c2 = (case *rop* of
| *Eqf* ⇒ *float-eq c1 c2*
| *Nef* ⇒ *float-ne c1 c2*
| *Ltf* ⇒ *float-lt c1 c2*
| *Gtf* ⇒ *float-gt c1 c2*
| *Lef* ⇒ *float-le c1 c2*
| *Gef* ⇒ *float-ge c1 c2*)

definition *types-agree* :: *t* ⇒ *v* ⇒ bool **where**

types-agree t v = (*typeof v* = *t*)

definition *cl-type* :: *cl* ⇒ *tf* **where**

cl-type cl = (case *cl* of *Func-native* - *tf* - - ⇒ *tf* | *Func-host* *tf* - ⇒ *tf*)

definition *rglob-is-mut* :: *global* ⇒ bool **where**

rglob-is-mut g = (*g-mut g* = *T-mut*)

definition *stypes* :: *s* ⇒ *nat* ⇒ *nat* ⇒ *tf* **where**

stypes s i j = ((*types* ((*inst s*)!i))!j)

definition *sfunc-ind* :: *s* ⇒ *nat* ⇒ *nat* ⇒ *nat* **where**

sfunc-ind s i j = ((*inst.funcs* ((*inst s*)!i))!j)

definition *sfunc* :: *s* ⇒ *nat* ⇒ *nat* ⇒ *cl* **where**

sfunc s i j = (*funcs s*)!(*sfunc-ind s i j*)

definition *sglob-ind* :: *s* ⇒ *nat* ⇒ *nat* ⇒ *nat* **where**

sglob-ind s i j = ((*inst.globs* ((*inst s*)!i))!j)

definition *sglob* :: *s* ⇒ *nat* ⇒ *nat* ⇒ *global* **where**

sglob s i j = (*globs s*)!(*sglob-ind s i j*)

definition *sglob-val* :: $s \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow v$ **where**

sglob-val $s\ i\ j = g\text{-val}\ (sglob\ s\ i\ j)$

definition *smem-ind* :: $s \Rightarrow \text{nat} \Rightarrow \text{nat option}$ **where**

smem-ind $s\ i = (inst.\text{mem}\ ((inst\ s)!i))$

definition *stab-s* :: $s \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow cl\ option$ **where**

stab-s $s\ i\ j = (let\ stabinst = ((tab\ s)!i)\ in\ (if\ (length\ (stabinst) > j)\ then\ (stabinst!j)\ else\ None))$

definition *stab* :: $s \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow cl\ option$ **where**

stab $s\ i\ j = (case\ (inst.\text{tab}\ ((inst\ s)!i))\ of\ Some\ k \Rightarrow\ stab\text{-}s\ s\ k\ j\ | \ None \Rightarrow\ None)$

definition *supdate-glob-s* :: $s \Rightarrow \text{nat} \Rightarrow v \Rightarrow s$ **where**

supdate-glob-s $s\ k\ v = s(globs := (globs\ s)[k := ((globs\ s)!k)(g\text{-val} := v)])$

definition *supdate-glob* :: $s \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow v \Rightarrow s$ **where**

supdate-glob $s\ i\ j\ v = (let\ k = sglob\text{-}ind\ s\ i\ j\ in\ supdate\text{-}glob\text{-}s\ s\ k\ v)$

definition *is-const* :: $e \Rightarrow bool$ **where**

is-const $e = (case\ e\ of\ Basic\ (C\ -) \Rightarrow\ True\ | \ - \Rightarrow\ False)$

definition *const-list* :: $e\ list \Rightarrow bool$ **where**

const-list $xs = list\text{-}all\ is\text{-}const\ xs$

inductive *store-extension* :: $s \Rightarrow s \Rightarrow bool$ **where**

$\llbracket insts = insts'; fs = fs'; tclss = tclss'; list\text{-}all2\ (\lambda bs\ bs'.\ mem\text{-}size\ bs \leq mem\text{-}size\ bs')\ bss\ bss'; gs = gs \rrbracket \implies$

$store\text{-}extension\ (\llbracket s.\text{inst} = insts,\ s.\text{funcs} = fs,\ s.\text{tab} = tclss,\ s.\text{mem} = bss,\ s.\text{globs} = gs \rrbracket)$

$= gs'$ $(\llbracket s.\text{inst} = insts',\ s.\text{funcs} = fs',\ s.\text{tab} = tclss',\ s.\text{mem} = bss',\ s.\text{globs} = gs' \rrbracket)$

abbreviation *to-e-list* :: $b\text{-}e\ list \Rightarrow e\ list$ ($\langle \$* \rightarrow 60 \rangle$) **where**

to-e-list $b\text{-}es \equiv map\ Basic\ b\text{-}es$

abbreviation *v-to-e-list* :: $v\ list \Rightarrow e\ list$ ($\langle \$\$* \rightarrow 60 \rangle$) **where**

v-to-e-list $ves \equiv map\ (\lambda v.\ \$C\ v)\ ves$

inductive *Lfilled* :: $\text{nat} \Rightarrow Lholed \Rightarrow e\ list \Rightarrow e\ list \Rightarrow bool$ **where**

$L0:\llbracket const\text{-}list\ vs;\ lholed = (LBase\ vs\ es') \rrbracket \implies Lfilled\ 0\ lholed\ es\ (vs\ @\ es\ @\ es')$

$| LN:\llbracket const\text{-}list\ vs;\ lholed = (LRec\ vs\ n\ es'\ l\ es'')\ ;\ Lfilled\ k\ l\ es\ lfilledk \rrbracket \implies Lfilled\ (k+1)\ lholed\ es\ (vs\ @\ [Label\ n\ es'\ lfilledk]\ @\ es'')$

inductive *Lfilled-exact* :: nat ⇒ Lholed ⇒ e list ⇒ e list ⇒ bool **where**

L0: $[[\text{lholed} = (\text{LBase } [] [])] \implies \text{Lfilled-exact } 0 \text{ lholed es es}$

| *LN*: $[[\text{const-list vs}; \text{lholed} = (\text{LRec vs n es' l es'})]; \text{Lfilled-exact } k \text{ l es lfilledk}] \implies \text{Lfilled-exact } (k+1) \text{ lholed es (vs @ [Label n es' lfilledk] @ es')$

definition *load-store-t-bounds* :: a ⇒ tp option ⇒ t ⇒ bool **where**

load-store-t-bounds a tp t = (case tp of
 None ⇒ $2^a \leq \text{t-length } t$
 | Some tp ⇒ $2^a \leq \text{tp-length } tp \wedge \text{tp-length } tp < \text{t-length } t \wedge \text{is-int-t } t$)

definition *cvt-i32* :: sx option ⇒ v ⇒ i32 option **where**

cvt-i32 sx v = (case v of
 ConstInt32 c ⇒ None
 | ConstInt64 c ⇒ Some (wasm-wrap c)
 | ConstFloat32 c ⇒ (case sx of
 Some U ⇒ ui32-trunc-f32 c
 | Some S ⇒ si32-trunc-f32 c
 | None ⇒ None)
 | ConstFloat64 c ⇒ (case sx of
 Some U ⇒ ui32-trunc-f64 c
 | Some S ⇒ si32-trunc-f64 c
 | None ⇒ None))

definition *cvt-i64* :: sx option ⇒ v ⇒ i64 option **where**

cvt-i64 sx v = (case v of
 ConstInt32 c ⇒ (case sx of
 Some U ⇒ Some (wasm-extend-u c)
 | Some S ⇒ Some (wasm-extend-s c)
 | None ⇒ None)
 | ConstInt64 c ⇒ None
 | ConstFloat32 c ⇒ (case sx of
 Some U ⇒ ui64-trunc-f32 c
 | Some S ⇒ si64-trunc-f32 c
 | None ⇒ None)
 | ConstFloat64 c ⇒ (case sx of
 Some U ⇒ ui64-trunc-f64 c
 | Some S ⇒ si64-trunc-f64 c
 | None ⇒ None))

definition *cvt-f32* :: sx option ⇒ v ⇒ f32 option **where**

cvt-f32 sx v = (case v of
 ConstInt32 c ⇒ (case sx of
 Some U ⇒ Some (f32-convert-ui32 c)
 | Some S ⇒ Some (f32-convert-si32 c)
 | - ⇒ None)

$| \text{ConstInt64 } c \Rightarrow (\text{case } sx \text{ of}$
 $\quad \text{Some } U \Rightarrow \text{Some } (f32\text{-convert-ui64 } c)$
 $\quad | \text{Some } S \Rightarrow \text{Some } (f32\text{-convert-si64 } c)$
 $\quad | - \Rightarrow \text{None})$
 $| \text{ConstFloat32 } c \Rightarrow \text{None}$
 $| \text{ConstFloat64 } c \Rightarrow \text{Some } (\text{wasm-demote } c)$

definition $\text{cvt-f64} :: sx \text{ option} \Rightarrow v \Rightarrow f64 \text{ option}$ **where**

$\text{cvt-f64 } sx \ v = (\text{case } v \text{ of}$
 $\quad \text{ConstInt32 } c \Rightarrow (\text{case } sx \text{ of}$
 $\quad \quad \text{Some } U \Rightarrow \text{Some } (f64\text{-convert-ui32 } c)$
 $\quad \quad | \text{Some } S \Rightarrow \text{Some } (f64\text{-convert-si32 } c)$
 $\quad \quad | - \Rightarrow \text{None})$
 $\quad | \text{ConstInt64 } c \Rightarrow (\text{case } sx \text{ of}$
 $\quad \quad \text{Some } U \Rightarrow \text{Some } (f64\text{-convert-ui64 } c)$
 $\quad \quad | \text{Some } S \Rightarrow \text{Some } (f64\text{-convert-si64 } c)$
 $\quad \quad | - \Rightarrow \text{None})$
 $\quad | \text{ConstFloat32 } c \Rightarrow \text{Some } (\text{wasm-promote } c)$
 $\quad | \text{ConstFloat64 } c \Rightarrow \text{None})$

definition $\text{cvt} :: t \Rightarrow sx \text{ option} \Rightarrow v \Rightarrow v \text{ option}$ **where**

$\text{cvt } t \ sx \ v = (\text{case } t \text{ of}$
 $\quad T\text{-i32} \Rightarrow (\text{case } (\text{cvt-i32 } sx \ v) \text{ of } \text{Some } c \Rightarrow \text{Some } (\text{ConstInt32 } c) |$
 $\quad \text{None} \Rightarrow \text{None})$
 $\quad | T\text{-i64} \Rightarrow (\text{case } (\text{cvt-i64 } sx \ v) \text{ of } \text{Some } c \Rightarrow \text{Some } (\text{ConstInt64 } c) |$
 $\quad \text{None} \Rightarrow \text{None})$
 $\quad | T\text{-f32} \Rightarrow (\text{case } (\text{cvt-f32 } sx \ v) \text{ of } \text{Some } c \Rightarrow \text{Some } (\text{ConstFloat32 } c) |$
 $\quad \text{None} \Rightarrow \text{None})$
 $\quad | T\text{-f64} \Rightarrow (\text{case } (\text{cvt-f64 } sx \ v) \text{ of } \text{Some } c \Rightarrow \text{Some } (\text{ConstFloat64 } c) |$
 $\quad \text{None} \Rightarrow \text{None}))$

definition $\text{bits} :: v \Rightarrow \text{bytes}$ **where**

$\text{bits } v = (\text{case } v \text{ of}$
 $\quad \text{ConstInt32 } c \Rightarrow (\text{serialise-i32 } c)$
 $\quad | \text{ConstInt64 } c \Rightarrow (\text{serialise-i64 } c)$
 $\quad | \text{ConstFloat32 } c \Rightarrow (\text{serialise-f32 } c)$
 $\quad | \text{ConstFloat64 } c \Rightarrow (\text{serialise-f64 } c))$

definition $\text{bitzero} :: t \Rightarrow v$ **where**

$\text{bitzero } t = (\text{case } t \text{ of}$
 $\quad T\text{-i32} \Rightarrow \text{ConstInt32 } 0$
 $\quad | T\text{-i64} \Rightarrow \text{ConstInt64 } 0$
 $\quad | T\text{-f32} \Rightarrow \text{ConstFloat32 } 0$
 $\quad | T\text{-f64} \Rightarrow \text{ConstFloat64 } 0)$

definition $n\text{-zeros} :: t \text{ list} \Rightarrow v \text{ list}$ **where**

$n\text{-zeros } ts = (\text{map } (\lambda t. \text{bitzero } t) \ ts)$

lemma is-int-t-exists :

assumes *is-int-t* *t*
shows $t = T-i32 \vee t = T-i64$
using *assms*
by (*cases t*) (*auto simp add: is-int-t-def*)

lemma *is-float-t-exists*:
assumes *is-float-t* *t*
shows $t = T-f32 \vee t = T-f64$
using *assms*
by (*cases t*) (*auto simp add: is-float-t-def*)

lemma *int-float-disjoint*: $is-int-t\ t = \neg(is-float-t\ t)$
by *simp (metis is-float-t-def is-int-t-def t.exhaust t.simps(13-16))*

lemma *stab-unfold*:
assumes $stab\ s\ i\ j = Some\ cl$
shows $\exists k. inst.tab\ ((inst\ s)!i) = Some\ k \wedge length\ ((tab\ s)!k) > j \wedge ((tab\ s)!k)!j = Some\ cl$
proof –
obtain *k* **where** $have-k:(inst.tab\ ((inst\ s)!i)) = Some\ k$
using *assms*
unfolding *stab-def*
by *fastforce*
hence $s-o:stab\ s\ i\ j = stab-s\ s\ k\ j$
using *assms*
unfolding *stab-def*
by *simp*
then obtain *stabinst* **where** $stabinst-def:stabinst = ((tab\ s)!k)$
by *blast*
hence $stab-s\ s\ k\ j = (stabinst)!j \wedge (length\ stabinst > j)$
using *assms s-o*
unfolding *stab-s-def*
by (*cases (length stabinst > j), auto*)
thus *?thesis*
using *have-k stabinst-def assms s-o*
by *auto*
qed

lemma *inj-basic*: *inj Basic*
by (*meson e.inject(1) injI*)

lemma *inj-basic-econst*: *inj* ($\lambda v. \$C\ v$)
by (*meson b-e.inject(16) e.inject(1) injI*)

lemma *to-e-list-1*: $[\$ a] = \$* [a]$
by *simp*

lemma *to-e-list-2*: $[\$ a, \$ b] = \$* [a, b]$

```

by simp

lemma to-e-list-3:[$ a, $ b, $ c] = $* [a, b, c]
  by simp

lemma v-exists-b-e:∃ ves. ($$*vs) = ($*ves)
proof (induction vs)
  case (Cons a vs)
  thus ?case
  by (metis list.simps(9))
qed auto

lemma Lfilled-exact-imp-Lfilled:
  assumes Lfilled-exact n lholed es LI
  shows Lfilled n lholed es LI
  using assms
proof (induction rule: Lfilled-exact.induct)
  case (L0 lholed es)
  thus ?case
  using const-list-def Lfilled.intros(1)
  by fastforce
next
  case (LN vs lholed n es' l es'' k es lfilledk)
  thus ?case
  using Lfilled.intros(2)
  by fastforce
qed

lemma Lfilled-exact-app-imp-exists-Lfilled:
  assumes const-list ves
  Lfilled-exact n lholed (ves@es) LI
  shows ∃ lholed'. Lfilled n lholed' es LI
  using assms(2,1)
proof (induction (ves@es) LI rule: Lfilled-exact.induct)
  case (L0 lholed)
  show ?case
  using Lfilled.intros(1)[OF L0(2), of - []]
  by fastforce
next
  case (LN vs lholed n es' l es'' k lfilledk)
  thus ?case
  using Lfilled.intros(2)
  by fastforce
qed

lemma Lfilled-imp-exists-Lfilled-exact:
  assumes Lfilled n lholed es LI
  shows ∃ lholed' ves es-c. const-list ves ∧ Lfilled-exact n lholed' (ves@es@es-c) LI
  using assms Lfilled-exact.intros

```

```

by (induction rule: Lfilled.induct) fastforce+

lemma n-zeros-typeof:
  n-zeros ts = vs  $\implies$  (ts = map typeof vs)
proof (induction ts arbitrary: vs)
  case Nil
  thus ?case
  unfolding n-zeros-def
  by simp
next
  case (Cons t ts)
  obtain vs' where n-zeros ts = vs'
  using n-zeros-def
  by blast
  moreover
  have typeof (bitzero t) = t
  unfolding typeof-def bitzero-def
  by (cases t, simp-all)
  ultimately
  show ?case
  using Cons
  unfolding n-zeros-def
  by auto
qed

end
theory Wasm imports Wasm-Base-Defs begin

inductive b-e-typing :: [t-context, b-e list, tf]  $\implies$  bool ( $\langle \cdot \vdash \cdot \rangle$  60) where
  — num ops
  const:  $\mathcal{C} \vdash [C\ v] : ([\ ] \rightarrow [(typeof\ v)])$ 
| unop-i:is-int-t t  $\implies \mathcal{C} \vdash [Unop-i\ t\ -] : ([t] \rightarrow [t])$ 
| unop-f:is-float-t t  $\implies \mathcal{C} \vdash [Unop-f\ t\ -] : ([t] \rightarrow [t])$ 
| binop-i:is-int-t t  $\implies \mathcal{C} \vdash [Binop-i\ t\ iop] : ([t, t] \rightarrow [t])$ 
| binop-f:is-float-t t  $\implies \mathcal{C} \vdash [Binop-f\ t\ -] : ([t, t] \rightarrow [t])$ 
| testop:is-int-t t  $\implies \mathcal{C} \vdash [Testop\ t\ -] : ([t] \rightarrow [T-i32])$ 
| relop-i:is-int-t t  $\implies \mathcal{C} \vdash [Relop-i\ t\ -] : ([t, t] \rightarrow [T-i32])$ 
| relop-f:is-float-t t  $\implies \mathcal{C} \vdash [Relop-f\ t\ -] : ([t, t] \rightarrow [T-i32])$ 
  — convert
| convert:  $[(t1 \neq t2); (sx = None) = ((is-float-t\ t1 \wedge is-float-t\ t2) \vee (is-int-t\ t1 \wedge is-int-t\ t2 \wedge (t-length\ t1 < t-length\ t2)))] \implies \mathcal{C} \vdash [Cvtop\ t1\ Convert\ t2\ sx] : ([t2] \rightarrow [t1])$ 
  — reinterpret
| reinterpret:  $[(t1 \neq t2); t-length\ t1 = t-length\ t2] \implies \mathcal{C} \vdash [Cvtop\ t1\ Reinterpret\ t2\ None] : ([t2] \rightarrow [t1])$ 
  — unreachable, nop, drop, select
| unreachable:  $\mathcal{C} \vdash [Unreachable] : (ts \rightarrow ts')$ 
| nop:  $\mathcal{C} \vdash [Nop] : ([\ ] \rightarrow [\ ])$ 

```

$| \text{drop}:\mathcal{C} \vdash [\text{Drop}] : ([t] \rightarrow [])$
 $| \text{select}:\mathcal{C} \vdash [\text{Select}] : ([t, t, T\text{-i32}] \rightarrow [t])$
— *block*
 $| \text{block}:\llbracket \text{tf} = (tn \rightarrow tm); \mathcal{C}(\text{label} := ([tm] @ (\text{label } \mathcal{C}))) \vdash \text{es} : (tn \rightarrow tm) \rrbracket \Longrightarrow \mathcal{C} \vdash$
 $[\text{Block } \text{tf } \text{es}] : (tn \rightarrow tm)$
— *loop*
 $| \text{loop}:\llbracket \text{tf} = (tn \rightarrow tm); \mathcal{C}(\text{label} := ([tn] @ (\text{label } \mathcal{C}))) \vdash \text{es} : (tn \rightarrow tm) \rrbracket \Longrightarrow \mathcal{C} \vdash$
 $[\text{Loop } \text{tf } \text{es}] : (tn \rightarrow tm)$
— *if then else*
 $| \text{if-wasm}:\llbracket \text{tf} = (tn \rightarrow tm); \mathcal{C}(\text{label} := ([tm] @ (\text{label } \mathcal{C}))) \vdash \text{es1} : (tn \rightarrow tm);$
 $\mathcal{C}(\text{label} := ([tm] @ (\text{label } \mathcal{C}))) \vdash \text{es2} : (tn \rightarrow tm) \rrbracket \Longrightarrow \mathcal{C} \vdash [\text{If } \text{tf } \text{es1 } \text{es2}] : (tn @$
 $[T\text{-i32}] \rightarrow tm)$
— *br*
 $| \text{br}:\llbracket i < \text{length}(\text{label } \mathcal{C}); (\text{label } \mathcal{C})!i = \text{ts} \rrbracket \Longrightarrow \mathcal{C} \vdash [\text{Br } i] : (t1s @ ts \rightarrow t2s)$
— *br-if*
 $| \text{br-if}:\llbracket i < \text{length}(\text{label } \mathcal{C}); (\text{label } \mathcal{C})!i = \text{ts} \rrbracket \Longrightarrow \mathcal{C} \vdash [\text{Br-if } i] : (ts @ [T\text{-i32}] \rightarrow ts)$
— *br-table*
 $| \text{br-table}:\llbracket \text{list-all } (\lambda i. i < \text{length}(\text{label } \mathcal{C}) \wedge (\text{label } \mathcal{C})!i = \text{ts}) (\text{is}@[i]) \rrbracket \Longrightarrow \mathcal{C} \vdash$
 $[\text{Br-table } \text{is } i] : (t1s @ ts @ [T\text{-i32}] \rightarrow t2s)$
— *return*
 $| \text{return}:\llbracket (\text{return } \mathcal{C}) = \text{Some } \text{ts} \rrbracket \Longrightarrow \mathcal{C} \vdash [\text{Return}] : (t1s @ ts \rightarrow t2s)$
— *call*
 $| \text{call}:\llbracket i < \text{length}(\text{func-t } \mathcal{C}); (\text{func-t } \mathcal{C})!i = \text{tf} \rrbracket \Longrightarrow \mathcal{C} \vdash [\text{Call } i] : \text{tf}$
— *call-indirect*
 $| \text{call-indirect}:\llbracket i < \text{length}(\text{types-t } \mathcal{C}); (\text{types-t } \mathcal{C})!i = (t1s \rightarrow t2s); (\text{table } \mathcal{C}) \neq \text{None} \rrbracket$
 $\Longrightarrow \mathcal{C} \vdash [\text{Call-indirect } i] : (t1s @ [T\text{-i32}] \rightarrow t2s)$
— *get-local*
 $| \text{get-local}:\llbracket i < \text{length}(\text{local } \mathcal{C}); (\text{local } \mathcal{C})!i = t \rrbracket \Longrightarrow \mathcal{C} \vdash [\text{Get-local } i] : ([] \rightarrow [t])$
— *set-local*
 $| \text{set-local}:\llbracket i < \text{length}(\text{local } \mathcal{C}); (\text{local } \mathcal{C})!i = t \rrbracket \Longrightarrow \mathcal{C} \vdash [\text{Set-local } i] : ([t] \rightarrow [])$
— *tee-local*
 $| \text{tee-local}:\llbracket i < \text{length}(\text{local } \mathcal{C}); (\text{local } \mathcal{C})!i = t \rrbracket \Longrightarrow \mathcal{C} \vdash [\text{Tee-local } i] : ([t] \rightarrow [t])$
— *get-global*
 $| \text{get-global}:\llbracket i < \text{length}(\text{global } \mathcal{C}); \text{tg-t } ((\text{global } \mathcal{C})!i) = t \rrbracket \Longrightarrow \mathcal{C} \vdash [\text{Get-global } i] : ([]$
 $\rightarrow [t])$
— *set-global*
 $| \text{set-global}:\llbracket i < \text{length}(\text{global } \mathcal{C}); \text{tg-t } ((\text{global } \mathcal{C})!i) = t; \text{is-mut } ((\text{global } \mathcal{C})!i) \rrbracket \Longrightarrow$
 $\mathcal{C} \vdash [\text{Set-global } i] : ([t] \rightarrow [])$
— *load*
 $| \text{load}:\llbracket (\text{memory } \mathcal{C}) = \text{Some } n; \text{load-store-t-bounds } a (\text{option-projl } \text{tp-sx } t) \rrbracket \Longrightarrow \mathcal{C}$
 $\vdash [\text{Load } t \text{ tp-sx } a \text{ off}] : ([T\text{-i32}] \rightarrow [t])$
— *store*
 $| \text{store}:\llbracket (\text{memory } \mathcal{C}) = \text{Some } n; \text{load-store-t-bounds } a \text{ tp } t \rrbracket \Longrightarrow \mathcal{C} \vdash [\text{Store } t \text{ tp } a$
 $\text{off}] : ([T\text{-i32}, t] \rightarrow [])$
— *current-memory*
 $| \text{current-memory}:(\text{memory } \mathcal{C}) = \text{Some } n \Longrightarrow \mathcal{C} \vdash [\text{Current-memory}] : ([] \rightarrow [T\text{-i32}])$
— *Grow-memory*
 $| \text{grow-memory}:(\text{memory } \mathcal{C}) = \text{Some } n \Longrightarrow \mathcal{C} \vdash [\text{Grow-memory}] : ([T\text{-i32}] \rightarrow$
 $[T\text{-i32}])$

— *empty program*
 $| \text{empty}:\mathcal{C} \vdash [] : ([] \rightarrow [])$
 — *composition*
 $| \text{composition}:\llbracket \mathcal{C} \vdash es : (t1s \rightarrow t2s); \mathcal{C} \vdash [e] : (t2s \rightarrow t3s) \rrbracket \implies \mathcal{C} \vdash es @ [e] : (t1s \rightarrow t3s)$
 — *weakening*
 $| \text{weakening}:\mathcal{C} \vdash es : (t1s \rightarrow t2s) \implies \mathcal{C} \vdash es : (ts @ t1s \rightarrow ts @ t2s)$

inductive *cl-typing* :: $[s\text{-context}, cl, tf] \Rightarrow \text{bool}$ **where**
 $| i < \text{length } (s\text{-inst } \mathcal{S}); ((s\text{-inst } \mathcal{S})!i) = \mathcal{C}; tf = (t1s \rightarrow t2s); \mathcal{C}(\text{local} := (\text{local } \mathcal{C}) @ t1s @ ts, \text{label} := ([t2s] @ (\text{label } \mathcal{C})), \text{return} := \text{Some } t2s) \vdash es : ([] \rightarrow t2s) \implies \text{cl-typing } \mathcal{S} (\text{Func-native } i \text{ } tf \text{ } ts \text{ } es) (t1s \rightarrow t2s)$
 $| \text{cl-typing } \mathcal{S} (\text{Func-host } tf \text{ } h) \text{ } tf$

inductive *e-typing* :: $[s\text{-context}, t\text{-context}, e \text{ list}, tf] \Rightarrow \text{bool}$ ($\langle \dots \vdash - : \rightarrow 60$)
and *s-typing* :: $[s\text{-context}, (t \text{ list}) \text{ option}, \text{nat}, v \text{ list}, e \text{ list}, t \text{ list}] \Rightarrow \text{bool}$ ($\langle \dots \vdash' - -; - : \rightarrow 60$) **where**

$\mathcal{C} \vdash b\text{-es} : tf \implies \mathcal{S} \cdot \mathcal{C} \vdash \$*b\text{-es} : tf$

$| \llbracket \mathcal{S} \cdot \mathcal{C} \vdash es : (t1s \rightarrow t2s); \mathcal{S} \cdot \mathcal{C} \vdash [e] : (t2s \rightarrow t3s) \rrbracket \implies \mathcal{S} \cdot \mathcal{C} \vdash es @ [e] : (t1s \rightarrow t3s)$

$| \mathcal{S} \cdot \mathcal{C} \vdash es : (t1s \rightarrow t2s) \implies \mathcal{S} \cdot \mathcal{C} \vdash es : (ts @ t1s \rightarrow ts @ t2s)$

$| \mathcal{S} \cdot \mathcal{C} \vdash [\text{Trap}] : tf$

$| \llbracket \mathcal{S} \cdot \text{Some } ts \vdash\text{-}i \text{ } vs; es : ts; \text{length } ts = n \rrbracket \implies \mathcal{S} \cdot \mathcal{C} \vdash [\text{Local } n \text{ } i \text{ } vs \text{ } es] : ([] \rightarrow ts)$

$| \llbracket \text{cl-typing } \mathcal{S} \text{ } cl \text{ } tf \rrbracket \implies \mathcal{S} \cdot \mathcal{C} \vdash [\text{Callcl } cl] : tf$

$| \llbracket \mathcal{S} \cdot \mathcal{C} \vdash e0s : (ts \rightarrow t2s); \mathcal{S} \cdot \mathcal{C}(\text{label} := ([ts] @ (\text{label } \mathcal{C}))) \vdash es : ([] \rightarrow t2s); \text{length } ts = n \rrbracket \implies \mathcal{S} \cdot \mathcal{C} \vdash [\text{Label } n \text{ } e0s \text{ } es] : ([] \rightarrow t2s)$

$| \llbracket i < (\text{length } (s\text{-inst } \mathcal{S})); tvs = \text{map } \text{typeof } vs; \mathcal{C} = ((s\text{-inst } \mathcal{S})!i)(\text{local} := (\text{local } ((s\text{-inst } \mathcal{S})!i) @ tvs), \text{return} := rs); \mathcal{S} \cdot \mathcal{C} \vdash es : ([] \rightarrow ts); (rs = \text{Some } ts) \vee rs = \text{None} \rrbracket \implies \mathcal{S} \cdot rs \vdash\text{-}i \text{ } vs; es : ts$

definition *globi-agree* $gs \ n \ g = (n < \text{length } gs \wedge gs!n = g)$

definition *memi-agree* $sm \ j \ m = ((\exists j' \ m'. j = \text{Some } j' \wedge j' < \text{length } sm \wedge m = \text{Some } m' \wedge sm!j' = m') \vee j = \text{None} \wedge m = \text{None})$

definition *funci-agree* $fs \ n \ f = (n < \text{length } fs \wedge fs!n = f)$

inductive *inst-typing* :: $[s\text{-context}, \text{inst}, t\text{-context}] \Rightarrow \text{bool}$ **where**
 $| \text{list-all2 } (\text{funci-agree } (s\text{-funcs } \mathcal{S})) \ fs \ tfs; \text{list-all2 } (\text{globi-agree } (s\text{-globs } \mathcal{S})) \ gs \ tgs;$

$(i = \text{Some } i' \wedge i' < \text{length } (s\text{-tab } \mathcal{S}) \wedge (s\text{-tab } \mathcal{S})!i' = (\text{the } n)) \vee (i = \text{None} \wedge n = \text{None}); \text{memi-agree } (s\text{-mem } \mathcal{S}) j m \Longrightarrow \text{inst-typing } \mathcal{S} (\text{types} = ts, \text{funcs} = fs, \text{tab} = i, \text{mem} = j, \text{globs} = gs) (\text{types-t} = ts, \text{func-t} = tfs, \text{global} = tgs, \text{table} = n, \text{memory} = m, \text{local} = [], \text{label} = [], \text{return} = \text{None})$

definition $\text{glob-agree } g \text{ tg} = (\text{tg-mut } \text{tg} = g\text{-mut } g \wedge \text{tg-t } \text{tg} = \text{typeof } (g\text{-val } g))$

definition $\text{tab-agree } \mathcal{S} \text{ tcl} = (\text{case } \text{tcl} \text{ of } \text{None} \Rightarrow \text{True} \mid \text{Some } \text{cl} \Rightarrow \exists \text{tf. } \text{cl-typing } \mathcal{S} \text{ cl } \text{tf})$

definition $\text{mem-agree } bs \text{ m} = (\lambda bs \text{ m. } m \leq \text{mem-size } bs) bs \text{ m}$

inductive $\text{store-typing} :: [s, s\text{-context}] \Rightarrow \text{bool}$ **where**

$\llbracket \mathcal{S} = (\text{inst} = Cs, \text{s-funcs} = tfs, \text{s-tab} = ns, \text{s-mem} = ms, \text{s-globs} = tgs); \text{list-all2 } (\text{inst-typing } \mathcal{S}) \text{ insts } Cs; \text{list-all2 } (\text{cl-typing } \mathcal{S}) fs \text{ tfs}; \text{list-all } (\text{tab-agree } \mathcal{S}) (\text{concat } \text{tclss}); \text{list-all2 } (\lambda \text{tcls } n. n \leq \text{length } \text{tcls}) \text{tclss } ns; \text{list-all2 } \text{mem-agree } bss \text{ ms}; \text{list-all2 } \text{glob-agree } gs \text{ tgs} \rrbracket \Longrightarrow \text{store-typing } (\text{s.inst} = \text{insts}, \text{s.funcs} = fs, \text{s.tab} = \text{tclss}, \text{s.mem} = bss, \text{s.globs} = gs) \mathcal{S}$

inductive $\text{config-typing} :: [\text{nat}, s, v \text{ list}, e \text{ list}, t \text{ list}] \Rightarrow \text{bool}$ ($\langle \vdash' \text{ - } -; -; - : - \rangle 60$) **where**

$\llbracket \text{store-typing } s \mathcal{S}; \mathcal{S} \cdot \text{None} \Vdash\text{-i } vs; es : ts \rrbracket \Longrightarrow \vdash\text{-i } s; vs; es : ts$

inductive $\text{reduce-simple} :: [e \text{ list}, e \text{ list}] \Rightarrow \text{bool}$ ($\langle \langle \text{-} \rangle \rightsquigarrow \langle \text{-} \rangle \rangle 60$) **where**

— *integer unary ops*

$\text{unop-i32} : \llbracket [\mathcal{C} (\text{ConstInt32 } c), \mathcal{S} (\text{Unop-i } T\text{-i32 } \text{iop})] \rrbracket \rightsquigarrow \llbracket [\mathcal{C} (\text{ConstInt32 } (\text{app-unop-i } \text{iop } c))] \rrbracket$

$\mid \text{unop-i64} : \llbracket [\mathcal{C} (\text{ConstInt64 } c), \mathcal{S} (\text{Unop-i } T\text{-i64 } \text{iop})] \rrbracket \rightsquigarrow \llbracket [\mathcal{C} (\text{ConstInt64 } (\text{app-unop-i } \text{iop } c))] \rrbracket$

— *float unary ops*

$\mid \text{unop-f32} : \llbracket [\mathcal{C} (\text{ConstFloat32 } c), \mathcal{S} (\text{Unop-f } T\text{-f32 } \text{fop})] \rrbracket \rightsquigarrow \llbracket [\mathcal{C} (\text{ConstFloat32 } (\text{app-unop-f } \text{fop } c))] \rrbracket$

$\mid \text{unop-f64} : \llbracket [\mathcal{C} (\text{ConstFloat64 } c), \mathcal{S} (\text{Unop-f } T\text{-f64 } \text{fop})] \rrbracket \rightsquigarrow \llbracket [\mathcal{C} (\text{ConstFloat64 } (\text{app-unop-f } \text{fop } c))] \rrbracket$

— *int32 binary ops*

$\mid \text{binop-i32-Some} : \llbracket [\text{app-binop-i } \text{iop } c1 \text{ } c2 = (\text{Some } c)] \rrbracket \Longrightarrow \llbracket [\mathcal{C} (\text{ConstInt32 } c1), \mathcal{C} (\text{ConstInt32 } c2), \mathcal{S} (\text{Binop-i } T\text{-i32 } \text{iop})] \rrbracket \rightsquigarrow \llbracket [\mathcal{C} (\text{ConstInt32 } c)] \rrbracket$

$\mid \text{binop-i32-None} : \llbracket [\text{app-binop-i } \text{iop } c1 \text{ } c2 = \text{None}] \rrbracket \Longrightarrow \llbracket [\mathcal{C} (\text{ConstInt32 } c1), \mathcal{C} (\text{ConstInt32 } c2), \mathcal{S} (\text{Binop-i } T\text{-i32 } \text{iop})] \rrbracket \rightsquigarrow \llbracket [\text{Trap}] \rrbracket$

— *int64 binary ops*

$\mid \text{binop-i64-Some} : \llbracket [\text{app-binop-i } \text{iop } c1 \text{ } c2 = (\text{Some } c)] \rrbracket \Longrightarrow \llbracket [\mathcal{C} (\text{ConstInt64 } c1), \mathcal{C} (\text{ConstInt64 } c2), \mathcal{S} (\text{Binop-i } T\text{-i64 } \text{iop})] \rrbracket \rightsquigarrow \llbracket [\mathcal{C} (\text{ConstInt64 } c)] \rrbracket$

$\mid \text{binop-i64-None} : \llbracket [\text{app-binop-i } \text{iop } c1 \text{ } c2 = \text{None}] \rrbracket \Longrightarrow \llbracket [\mathcal{C} (\text{ConstInt64 } c1), \mathcal{C} (\text{ConstInt64 } c2), \mathcal{S} (\text{Binop-i } T\text{-i64 } \text{iop})] \rrbracket \rightsquigarrow \llbracket [\text{Trap}] \rrbracket$

— *float32 binary ops*

$\mid \text{binop-f32-Some} : \llbracket [\text{app-binop-f } \text{fop } c1 \text{ } c2 = (\text{Some } c)] \rrbracket \Longrightarrow \llbracket [\mathcal{C} (\text{ConstFloat32 } c1), \mathcal{C} (\text{ConstFloat32 } c2), \mathcal{S} (\text{Binop-f } T\text{-f32 } \text{fop})] \rrbracket \rightsquigarrow \llbracket [\mathcal{C} (\text{ConstFloat32 } c)] \rrbracket$

| *binop-f32-None*: $[[app\text{-}binop\text{-}f\ fop\ c1\ c2 = None]] \implies ([\$C\ (ConstFloat32\ c1),\ \$C\ (ConstFloat32\ c2),\ \$(Binop\text{-}f\ T\text{-}f32\ fop)]) \rightsquigarrow ([Trap])$
— *float64 binary ops*
| *binop-f64-Some*: $[[app\text{-}binop\text{-}f\ fop\ c1\ c2 = (Some\ c)]] \implies ([\$C\ (ConstFloat64\ c1),\ \$C\ (ConstFloat64\ c2),\ \$(Binop\text{-}f\ T\text{-}f64\ fop)]) \rightsquigarrow ([\$C\ (ConstFloat64\ c)])$
| *binop-f64-None*: $[[app\text{-}binop\text{-}f\ fop\ c1\ c2 = None]] \implies ([\$C\ (ConstFloat64\ c1),\ \$C\ (ConstFloat64\ c2),\ \$(Binop\text{-}f\ T\text{-}f64\ fop)]) \rightsquigarrow ([Trap])$
— *testops*
| *testop-i32*: $([\$C\ (ConstInt32\ c),\ \$(Testop\ T\text{-}i32\ testop)]) \rightsquigarrow ([\$C\ ConstInt32\ (wasm\text{-}bool\ (app\text{-}testop\text{-}i\ testop\ c))])$
| *testop-i64*: $([\$C\ (ConstInt64\ c),\ \$(Testop\ T\text{-}i64\ testop)]) \rightsquigarrow ([\$C\ ConstInt32\ (wasm\text{-}bool\ (app\text{-}testop\text{-}i\ testop\ c))])$
— *int relops*
| *relop-i32*: $([\$C\ (ConstInt32\ c1),\ \$C\ (ConstInt32\ c2),\ \$(Relop\text{-}i\ T\text{-}i32\ iop)]) \rightsquigarrow ([\$C\ (ConstInt32\ (wasm\text{-}bool\ (app\text{-}relop\text{-}i\ iop\ c1\ c2))])$
| *relop-i64*: $([\$C\ (ConstInt64\ c1),\ \$C\ (ConstInt64\ c2),\ \$(Relop\text{-}i\ T\text{-}i64\ iop)]) \rightsquigarrow ([\$C\ (ConstInt32\ (wasm\text{-}bool\ (app\text{-}relop\text{-}i\ iop\ c1\ c2))])$
— *float relops*
| *relop-f32*: $([\$C\ (ConstFloat32\ c1),\ \$C\ (ConstFloat32\ c2),\ \$(Relop\text{-}f\ T\text{-}f32\ fop)]) \rightsquigarrow ([\$C\ (ConstInt32\ (wasm\text{-}bool\ (app\text{-}relop\text{-}f\ fop\ c1\ c2))])$
| *relop-f64*: $([\$C\ (ConstFloat64\ c1),\ \$C\ (ConstFloat64\ c2),\ \$(Relop\text{-}f\ T\text{-}f64\ fop)]) \rightsquigarrow ([\$C\ (ConstInt32\ (wasm\text{-}bool\ (app\text{-}relop\text{-}f\ fop\ c1\ c2))])$
— *convert*
| *convert-Some*: $[[types\text{-}agree\ t1\ v;\ cut\ t2\ sx\ v = (Some\ v')]] \implies ([\$C\ v),\ \$(Cvtop\ t2\ Convert\ t1\ sx)]) \rightsquigarrow ([\$C\ v'])$
| *convert-None*: $[[types\text{-}agree\ t1\ v;\ cut\ t2\ sx\ v = None]] \implies ([\$C\ v),\ \$(Cvtop\ t2\ Convert\ t1\ sx)]) \rightsquigarrow ([Trap])$
— *reinterpret*
| *reinterpret*: $types\text{-}agree\ t1\ v \implies ([\$C\ v),\ \$(Cvtop\ t2\ Reinterpret\ t1\ None)]) \rightsquigarrow ([\$C\ (wasm\text{-}deserialise\ (bits\ v)\ t2)])$
— *unreachable*
| *unreachable*: $([\$ Unreachable]) \rightsquigarrow ([Trap])$
— *nop*
| *nop*: $([\$ Nop]) \rightsquigarrow ([\])$
— *drop*
| *drop*: $([\$ C\ v),\ (\$ Drop)]) \rightsquigarrow ([\])$
— *select*
| *select-false*: $int\text{-}eq\ n\ 0 \implies ([\$ C\ v1),\ \$C\ v2),\ \$C\ (ConstInt32\ n),\ (\$ Select)]) \rightsquigarrow ([\$ C\ v2])$
| *select-true*: $int\text{-}ne\ n\ 0 \implies ([\$ C\ v1),\ \$C\ v2),\ \$C\ (ConstInt32\ n),\ (\$ Select)]) \rightsquigarrow ([\$ C\ v1])$
— *block*
| *block*: $[[const\text{-}list\ vs;\ length\ vs = n;\ length\ t1s = n;\ length\ t2s = m]] \implies ([vs\ @\ [\$ (Block\ (t1s\ \text{-}>\ t2s)\ es)]) \rightsquigarrow ([Label\ m\ []\ (vs\ @\ (\$*\ es))])$
— *loop*
| *loop*: $[[const\text{-}list\ vs;\ length\ vs = n;\ length\ t1s = n;\ length\ t2s = m]] \implies ([vs\ @\ [\$ (Loop\ (t1s\ \text{-}>\ t2s)\ es)]) \rightsquigarrow ([Label\ n\ [\$ (Loop\ (t1s\ \text{-}>\ t2s)\ es)]\ (vs\ @\ (\$*\ es))])$
— *if*
| *if-false*: $int\text{-}eq\ n\ 0 \implies ([\$ C\ (ConstInt32\ n),\ \$(If\ tf\ e1s\ e2s)]) \rightsquigarrow ([\$ (Block\ tf\ e2s)])$

| *if-true:int-ne* $n \ 0 \implies \langle \langle [\$C \ (ConstInt32 \ n), \ \$(\text{If} \ \text{tf} \ e1s \ e2s)] \rangle \rangle \rightsquigarrow \langle \langle [\$(\text{Block} \ \text{tf} \ e1s)] \rangle \rangle$
— *label*
| *label-const:const-list* $vs \implies \langle \langle [Label \ n \ es \ vs] \rangle \rangle \rightsquigarrow \langle \langle vs \rangle \rangle$
| *label-trap*: $\langle \langle [Label \ n \ es \ [Trap]] \rangle \rangle \rightsquigarrow \langle \langle [Trap] \rangle \rangle$
— *br*
| *br*: $\langle \langle [const-list \ vs; \ length \ vs = n; \ Lfilled \ i \ lholed \ (vs \ @ \ [\$(\text{Br} \ i)] \ LI)] \rangle \rangle \implies \langle \langle [Label \ n \ es \ LI] \rangle \rangle \rightsquigarrow \langle \langle vs \ @ \ es \rangle \rangle$
— *br-if*
| *br-if-false:int-eq* $n \ 0 \implies \langle \langle [\$C \ (ConstInt32 \ n), \ \$(\text{Br-if} \ i)] \rangle \rangle \rightsquigarrow \langle \langle [] \rangle \rangle$
| *br-if-true:int-ne* $n \ 0 \implies \langle \langle [\$C \ (ConstInt32 \ n), \ \$(\text{Br-if} \ i)] \rangle \rangle \rightsquigarrow \langle \langle [\$(\text{Br} \ i)] \rangle \rangle$
— *br-table*
| *br-table*: $\langle \langle [length \ is > (nat-of-int \ c)] \rangle \rangle \implies \langle \langle [\$C \ (ConstInt32 \ c), \ \$(\text{Br-table} \ is \ i)] \rangle \rangle \rightsquigarrow \langle \langle [\$(\text{Br} \ (is!(nat-of-int \ c)))] \rangle \rangle$
| *br-table-length*: $\langle \langle [length \ is \leq (nat-of-int \ c)] \rangle \rangle \implies \langle \langle [\$C \ (ConstInt32 \ c), \ \$(\text{Br-table} \ is \ i)] \rangle \rangle \rightsquigarrow \langle \langle [\$(\text{Br} \ i)] \rangle \rangle$
— *local*
| *local-const*: $\langle \langle [const-list \ es; \ length \ es = n] \rangle \rangle \implies \langle \langle [Local \ n \ i \ vs \ es] \rangle \rangle \rightsquigarrow \langle \langle es \rangle \rangle$
| *local-trap*: $\langle \langle [Local \ n \ i \ vs \ [Trap]] \rangle \rangle \rightsquigarrow \langle \langle [Trap] \rangle \rangle$
— *return*
| *return*: $\langle \langle [const-list \ vs; \ length \ vs = n; \ Lfilled \ j \ lholed \ (vs \ @ \ [\$Return]) \ es] \rangle \rangle \implies \langle \langle [Local \ n \ i \ vs \ es] \rangle \rangle \rightsquigarrow \langle \langle vs \rangle \rangle$
— *tee-local*
| *tee-local:is-const* $v \implies \langle \langle [v, \ \$(\text{Tee-local} \ i)] \rangle \rangle \rightsquigarrow \langle \langle [v, \ v, \ \$(\text{Set-local} \ i)] \rangle \rangle$
| *trap*: $\langle \langle [es \neq [Trap]; \ Lfilled \ 0 \ lholed \ [Trap] \ es] \rangle \rangle \implies \langle \langle es \rangle \rangle \rightsquigarrow \langle \langle [Trap] \rangle \rangle$

inductive *reduce* :: $[s, \ v \ list, \ e \ list, \ nat, \ s, \ v \ list, \ e \ list] \Rightarrow \text{bool} \ (\langle \langle [-;-;-] \rangle \rangle \rightsquigarrow' \ - \ \langle \langle [-;-;-] \rangle \rangle \succ 60)$ **where**
— *lifting basic reduction*
basic: $\langle \langle e \rangle \rangle \rightsquigarrow \langle \langle e' \rangle \rangle \implies \langle \langle [s;vs;e] \rangle \rangle \rightsquigarrow-i \langle \langle [s;vs;e'] \rangle \rangle$
— *call*
| *call*: $\langle \langle [s;vs;[\$(\text{Call} \ j)]] \rangle \rangle \rightsquigarrow-i \langle \langle [s;vs;[\text{Callcl} \ (sfunc \ s \ i \ j)]] \rangle \rangle$
— *call-indirect*
| *call-indirect-Some*: $\langle \langle [stab \ s \ i \ (nat-of-int \ c) = \text{Some} \ cl; \ stypes \ s \ i \ j = \text{tf}; \ cl\text{-type} \ cl = \text{tf}] \rangle \rangle \implies \langle \langle [s;vs;[\$C \ (ConstInt32 \ c), \ \$(\text{Call-indirect} \ j)]] \rangle \rangle \rightsquigarrow-i \langle \langle [s;vs;[\text{Callcl} \ cl]] \rangle \rangle$
| *call-indirect-None*: $\langle \langle [stab \ s \ i \ (nat-of-int \ c) = \text{Some} \ cl \wedge \text{stypes} \ s \ i \ j \neq \text{cl-type} \ cl] \vee \text{stab} \ s \ i \ (nat-of-int \ c) = \text{None}] \rangle \rangle \implies \langle \langle [s;vs;[\$C \ (ConstInt32 \ c), \ \$(\text{Call-indirect} \ j)]] \rangle \rangle \rightsquigarrow-i \langle \langle [s;vs;[\text{Trap}]] \rangle \rangle$
— *call*
| *callcl-native*: $\langle \langle [cl = \text{Func-native} \ j \ (t1s \ -> \ t2s) \ ts \ es; \ ves = (\$\$* \ vcs); \ length \ vcs = n; \ length \ ts = k; \ length \ t1s = n; \ length \ t2s = m; \ (n\text{-zeros} \ ts = zs)] \rangle \rangle \implies \langle \langle [s;vs;ves \ @ \ [\text{Callcl} \ cl]] \rangle \rangle \rightsquigarrow-i \langle \langle [s;vs;[\text{Local} \ m \ j \ (vcs@zs) \ [\$(\text{Block} \ ([] \ -> \ t2s) \ es)]] \rangle \rangle$
| *callcl-host-Some*: $\langle \langle [cl = \text{Func-host} \ (t1s \ -> \ t2s) \ f; \ ves = (\$\$* \ vcs); \ length \ vcs = n; \ length \ t1s = n; \ length \ t2s = m; \ \text{host-apply} \ s \ (t1s \ -> \ t2s) \ f \ vcs \ hs = \text{Some} \ (s', \ vcs')] \rangle \rangle \implies \langle \langle [s;vs;ves \ @ \ [\text{Callcl} \ cl]] \rangle \rangle \rightsquigarrow-i \langle \langle [s';vs;(\$\$* \ vcs')] \rangle \rangle$
| *callcl-host-None*: $\langle \langle [cl = \text{Func-host} \ (t1s \ -> \ t2s) \ f; \ ves = (\$\$* \ vcs); \ length \ vcs = n; \ length \ t1s = n; \ length \ t2s = m] \rangle \rangle \implies \langle \langle [s;vs;ves \ @ \ [\text{Callcl} \ cl]] \rangle \rangle \rightsquigarrow-i \langle \langle [s;vs;[\text{Trap}]] \rangle \rangle$
— *get-local*
| *get-local*: $\langle \langle [length \ vi = j] \rangle \rangle \implies \langle \langle [s;(vi \ @ \ [v] \ @ \ vs);[\$(\text{Get-local} \ j)]] \rangle \rangle \rightsquigarrow-i \langle \langle [s;(vi \ @ \ [v]) \rangle \rangle$

$\text{@ } vs;[\$(C v)]$
— *set-local*
| *set-local*: $[\text{length } vi = j] \implies (\!s;vi \text{ @ } [v] \text{ @ } vs;[\$(C v'), \$(Set-local j)]) \rightsquigarrow\text{-}i (\!s;(vi \text{ @ } [v'] \text{ @ } vs);[])$
— *get-global*
| *get-global*: $(\!s;vs;[\$(Get-global j)]) \rightsquigarrow\text{-}i (\!s;vs;[\$ C (sglob-val s i j)])$
— *set-global*
| *set-global*:*supdate-glob* $s i j v = s' \implies (\!s;vs;[\$(C v), \$(Set-global j)]) \rightsquigarrow\text{-}i (\!s';vs;[])$
— *load*
| *load-Some*: $[\text{smem-ind } s i = \text{Some } j; ((mem s)!j) = m; \text{load } m \text{ (nat-of-int } k) \text{ off (t-length } t) = \text{Some } bs] \implies (\!s;vs;[\$C (ConstInt32 k), \$(Load t None a off)]) \rightsquigarrow\text{-}i (\!s;vs;[\$C (wasm-deserialise bs t)])$
| *load-None*: $[\text{smem-ind } s i = \text{Some } j; ((mem s)!j) = m; \text{load } m \text{ (nat-of-int } k) \text{ off (t-length } t) = \text{None}] \implies (\!s;vs;[\$C (ConstInt32 k), \$(Load t None a off)]) \rightsquigarrow\text{-}i (\!s;vs;[Trap])$
— *load packed*
| *load-packed-Some*: $[\text{smem-ind } s i = \text{Some } j; ((mem s)!j) = m; \text{load-packed } sx m \text{ (nat-of-int } k) \text{ off (tp-length } tp) \text{ (t-length } t) = \text{Some } bs] \implies (\!s;vs;[\$C (ConstInt32 k), \$(Load t (Some (tp, sx)) a off)]) \rightsquigarrow\text{-}i (\!s;vs;[\$C (wasm-deserialise bs t)])$
| *load-packed-None*: $[\text{smem-ind } s i = \text{Some } j; ((mem s)!j) = m; \text{load-packed } sx m \text{ (nat-of-int } k) \text{ off (tp-length } tp) \text{ (t-length } t) = \text{None}] \implies (\!s;vs;[\$C (ConstInt32 k), \$(Load t (Some (tp, sx)) a off)]) \rightsquigarrow\text{-}i (\!s;vs;[Trap])$
— *store*
| *store-Some*: $[\text{types-agree } t v; \text{smem-ind } s i = \text{Some } j; ((mem s)!j) = m; \text{store } m \text{ (nat-of-int } k) \text{ off (bits } v) \text{ (t-length } t) = \text{Some } mem] \implies (\!s;vs;[\$C (ConstInt32 k), \$C v, \$(Store t None a off)]) \rightsquigarrow\text{-}i (\!s(\!mem:= ((mem s)[j := mem']));vs;[])$
| *store-None*: $[\text{types-agree } t v; \text{smem-ind } s i = \text{Some } j; ((mem s)!j) = m; \text{store } m \text{ (nat-of-int } k) \text{ off (bits } v) \text{ (t-length } t) = \text{None}] \implies (\!s;vs;[\$C (ConstInt32 k), \$C v, \$(Store t None a off)]) \rightsquigarrow\text{-}i (\!s;vs;[Trap])$
— *store packed*
| *store-packed-Some*: $[\text{types-agree } t v; \text{smem-ind } s i = \text{Some } j; ((mem s)!j) = m; \text{store-packed } m \text{ (nat-of-int } k) \text{ off (bits } v) \text{ (tp-length } tp) = \text{Some } mem] \implies (\!s;vs;[\$C (ConstInt32 k), \$C v, \$(Store t (Some tp) a off)]) \rightsquigarrow\text{-}i (\!s(\!mem:= ((mem s)[j := mem]));vs;[])$
| *store-packed-None*: $[\text{types-agree } t v; \text{smem-ind } s i = \text{Some } j; ((mem s)!j) = m; \text{store-packed } m \text{ (nat-of-int } k) \text{ off (bits } v) \text{ (tp-length } tp) = \text{None}] \implies (\!s;vs;[\$C (ConstInt32 k), \$C v, \$(Store t (Some tp) a off)]) \rightsquigarrow\text{-}i (\!s;vs;[Trap])$
— *current-memory*
| *current-memory*: $[\text{smem-ind } s i = \text{Some } j; ((mem s)!j) = m; \text{mem-size } m = n] \implies (\!s;vs;[\$(Current-memory)]) \rightsquigarrow\text{-}i (\!s;vs;[\$C (ConstInt32 (int-of-nat n))])$
— *grow-memory*
| *grow-memory*: $[\text{smem-ind } s i = \text{Some } j; ((mem s)!j) = m; \text{mem-size } m = n; \text{mem-grow } m \text{ (nat-of-int } c) = \text{Some } mem] \implies (\!s;vs;[\$C (ConstInt32 c), \$(Grow-memory)]) \rightsquigarrow\text{-}i (\!s(\!mem:= ((mem s)[j := mem]));vs;[\$C (ConstInt32 (int-of-nat n))])$
— *grow-memory fail*
| *grow-memory-fail*: $[\text{smem-ind } s i = \text{Some } j; ((mem s)!j) = m; \text{mem-size } m = n] \implies (\!s;vs;[\$C (ConstInt32 c), \$(Grow-memory)]) \rightsquigarrow\text{-}i (\!s;vs;[\$C (ConstInt32 (int32-minus-one))])$

— *inductive label reduction*
 $| \text{label}::[(\langle s;vs;es \rangle \rightsquigarrow\text{-}i \langle s';vs';es' \rangle); \text{Lfilled } k \text{ lholed } es \text{ les}; \text{Lfilled } k \text{ lholed } es' \text{ les}'] \implies$
 $(\langle s;vs;les \rangle \rightsquigarrow\text{-}i \langle s';vs';les' \rangle)$
 — *inductive local reduction*
 $| \text{local}::[(\langle s;vs;es \rangle \rightsquigarrow\text{-}i \langle s';vs';es' \rangle)] \implies (\langle s;v0s;[\text{Local } n \text{ } i \text{ } vs \text{ } es] \rangle \rightsquigarrow\text{-}j \langle s';v0s;[\text{Local } n$
 $i \text{ } vs' \text{ } es'] \rangle)$

end

4 Host Properties

theory *Wasm-Axioms* **imports** *Wasm* **begin**

lemma *mem-grow-size*:

assumes *mem-grow* $m \ n = m'$

shows $(\text{mem-size } m + (64000 * n)) = \text{mem-size } m'$

using *assms Abs-mem-inverse Abs-bytes-inverse*

unfolding *mem-grow-def mem-size-def mem-append-def bytes-replicate-def*

by *auto*

lemma *load-size*:

$(\text{load } m \ n \ \text{off } l = \text{None}) = (\text{mem-size } m < (\text{off} + n + l))$

unfolding *load-def*

by $(\text{cases } n + \text{off} + l \leq \text{mem-size } m)$ *auto*

lemma *load-packed-size*:

$(\text{load-packed } sx \ m \ n \ \text{off } lp \ l = \text{None}) = (\text{mem-size } m < (\text{off} + n + lp))$

using *load-size*

unfolding *load-packed-def*

by $(\text{cases } n + \text{off} + l \leq \text{mem-size } m)$ *auto*

lemma *store-size1*:

$(\text{store } m \ n \ \text{off } v \ l = \text{None}) = (\text{mem-size } m < (\text{off} + n + l))$

unfolding *store-def*

by $(\text{cases } n + \text{off} + l \leq \text{mem-size } m)$ *auto*

lemma *store-size*:

assumes $(\text{store } m \ n \ \text{off } v \ l = \text{Some } m')$

shows $\text{mem-size } m = \text{mem-size } m'$

using *assms Abs-mem-inverse Abs-bytes-inverse*

unfolding *store-def write-bytes-def bytes-takefill-def*

by $(\text{cases } n + \text{off} + l \leq \text{mem-size } m)$ *(auto simp add: mem-size-def)*

lemma *store-packed-size1*:

$(\text{store-packed } m \ n \ \text{off } v \ l = \text{None}) = (\text{mem-size } m < (\text{off} + n + l))$

using *store-size1*

unfolding *store-packed-def*

by *simp*

lemma *store-packed-size*:
assumes (*store-packed* *m n* *off v l* = *Some m'*)
shows *mem-size m* = *mem-size m'*
using *assms store-size*
unfolding *store-packed-def*
by *simp*

axiomatization where

wasm-deserialise-type:typeof (*wasm-deserialise bs t*) = *t*

axiomatization where

host-apply-preserve-store: *list-all2 types-agree t1s vs* \implies *host-apply s* (*t1s* \rightarrow *t2s*) *f vs hs* = *Some (s', vs')* \implies *store-extension s s'*
and *host-apply-respect-type*: *list-all2 types-agree t1s vs* \implies *host-apply s* (*t1s* \rightarrow *t2s*) *f vs hs* = *Some (s', vs')* \implies *list-all2 types-agree t2s vs'*
end

5 Auxiliary Type System Properties

theory *Wasm-Properties-Aux* **imports** *Wasm-Axioms* **begin**

lemma *typeof-i32*:
assumes *typeof v* = *T-i32*
shows $\exists c. v$ = *ConstInt32 c*
using *assms*
unfolding *typeof-def*
by (*cases v*) *auto*

lemma *typeof-i64*:
assumes *typeof v* = *T-i64*
shows $\exists c. v$ = *ConstInt64 c*
using *assms*
unfolding *typeof-def*
by (*cases v*) *auto*

lemma *typeof-f32*:
assumes *typeof v* = *T-f32*
shows $\exists c. v$ = *ConstFloat32 c*
using *assms*
unfolding *typeof-def*
by (*cases v*) *auto*

lemma *typeof-f64*:
assumes *typeof v* = *T-f64*
shows $\exists c. v$ = *ConstFloat64 c*
using *assms*
unfolding *typeof-def*
by (*cases v*) *auto*

```

lemma exists-v-typeof:  $\exists v v. \text{typeof } v = t$ 
proof (cases t)
  case T-i32
    fix v
    have typeof (ConstInt32 v) = t
      using T-i32
      unfolding typeof-def
      by simp
    thus ?thesis
      using T-i32
      by fastforce
  next
    case T-i64
      fix v
      have typeof (ConstInt64 v) = t
        using T-i64
        unfolding typeof-def
        by simp
      thus ?thesis
        using T-i64
        by fastforce
  next
    case T-f32
      fix v
      have typeof (ConstFloat32 v) = t
        using T-f32
        unfolding typeof-def
        by simp
      thus ?thesis
        using T-f32
        by fastforce
  next
    case T-f64
      fix v
      have typeof (ConstFloat64 v) = t
        using T-f64
        unfolding typeof-def
        by simp
      thus ?thesis
        using T-f64
        by fastforce
qed

lemma lfilled-collapse1:
  assumes Lfilled n lholed (vs@es) LI
    const-list vs
    length vs  $\geq$  l
  shows  $\exists \text{lholed}'. \text{Lfilled } n \text{lholed}' ((\text{drop } (\text{length } vs - l) \text{vs})@es) \text{LI}$ 

```

```

using assms(1)
proof (induction vs@es LI rule: Lfilled.induct)
  case (L0 vs' lholed es')
  obtain vs1 vs2 where vs = vs1@vs2 length vs2 = l
    using assms(3)
    by (metis append-take-drop-id diff-diff-cancel length-drop)
  moreover
  hence const-list (vs'@vs1)
    using L0(1) assms(2)
    unfolding const-list-def
    by simp
  ultimately
  show ?case
    using Lfilled.intros(1)[of vs'@vs1 - es' vs2@es]
    by fastforce
next
  case (LN vs lholed n es' l es'' k lfilledk)
  thus ?case
    using Lfilled.intros(2)
    by fastforce
qed

```

```

lemma lfilled-collapse2:
  assumes Lfilled n lholed (es@es') LI
  shows  $\exists$  lholed' vs'. Lfilled n lholed' es LI
  using assms
proof (induction es@es' LI rule: Lfilled.induct)
  case (L0 vs lholed es')
  thus ?case
    using Lfilled.intros(1)
    by fastforce
next
  case (LN vs lholed n es' l es'' k lfilledk)
  thus ?case
    using Lfilled.intros(2)
    by fastforce
qed

```

```

lemma lfilled-collapse3:
  assumes Lfilled k lholed [Label n les es] LI
  shows  $\exists$  lholed'. Lfilled (Suc k) lholed' es LI
  using assms
proof (induction [Label n les es] LI rule: Lfilled.induct)
  case (L0 vs lholed es')
  have Lfilled 0 (LBase [] []) es es
    using Lfilled.intros(1) [of <[]> <LBase [] []> <[]> es]
    by (simp add: const-list-def)
  thus ?case
    using Lfilled.intros(2) L0

```

```

    by fastforce
next
case (LN vs lholed n es' l es'' k lfilledk)
thus ?case
    using Lfilled.intros(2)
    by fastforce
qed

```

```

lemma unlift-b-e: assumes  $\mathcal{S} \cdot \mathcal{C} \vdash \$*b\text{-es} : tf$  shows  $\mathcal{C} \vdash b\text{-es} : tf$ 
using assms proof (induction  $\mathcal{S} \mathcal{C} (\$*b\text{-es}) tf$  arbitrary: b-es)
    case (1  $\mathcal{C} b\text{-es} tf \mathcal{S}$ )
    then show ?case
        using inj-basic map-injective
        by auto
next
case (2  $\mathcal{S} \mathcal{C} es\ t1s\ t2s\ e\ t3s$ )
obtain  $es' e'$  where  $es' @ [e'] = b\text{-es}$ 
    using 2(5)
    by (simp add: snoc-eq-iff-butlast)
then show ?case using 2
    using b-e-typing.composition
    by fastforce
next
case (3  $\mathcal{S} \mathcal{C} t1s\ t2s\ ts$ )
then show ?case
    using b-e-typing.weakening
    by blast
qed auto

```

```

lemma store-typing-imp-inst-length-eq:
    assumes store-typing  $s \mathcal{S}$ 
    shows  $length (inst\ s) = length (s\text{-inst } \mathcal{S})$ 
    using assms list-all2-lengthD
    unfolding store-typing.simps
    by fastforce

```

```

lemma store-typing-imp-func-length-eq:
    assumes store-typing  $s \mathcal{S}$ 
    shows  $length (funcs\ s) = length (s\text{-funcs } \mathcal{S})$ 
    using assms list-all2-lengthD
    unfolding store-typing.simps
    by fastforce

```

```

lemma store-typing-imp-mem-length-eq:
    assumes store-typing  $s \mathcal{S}$ 
    shows  $length (s.\text{mem } s) = length (s\text{-mem } \mathcal{S})$ 
    using assms list-all2-lengthD
    unfolding store-typing.simps

```

by *fastforce*

lemma *store-typing-imp-glob-length-eq*:
assumes *store-typing s S*
shows $\text{length } (\text{globs } s) = \text{length } (s\text{-globs } S)$
using *assms list-all2-lengthD*
unfolding *store-typing.simps*
by *fastforce*

lemma *store-typing-imp-inst-typing*:
assumes *store-typing s S*
 $i < \text{length } (\text{inst } s)$
shows *inst-typing S ((inst s)!i) ((s-inst S)!i)*
using *assms*
unfolding *list-all2-conv-all-nth store-typing.simps*
by *fastforce*

lemma *stab-typed-some-imp-member*:
assumes *stab s i c = Some cl*
 store-typing s S
 $i < \text{length } (\text{inst } s)$
shows $\text{Some } cl \in \text{set } (\text{concat } (s.\text{tab } s))$
proof –
 obtain *k' where k-def:inst.tab ((inst s)!i) = Some k'*
 $\text{length } ((s.\text{tab } s)!k') > c$
 $((s.\text{tab } s)!k')!c = \text{Some } cl$
 using *stab-unfold assms(1,3)*
 by *fastforce*
 hence $\text{Some } cl \in \text{set } ((s.\text{tab } s)!k')$
 using *nth-mem*
 by *fastforce*
 moreover
 have *inst-typing S ((inst s)!i) ((s-inst S)!i)*
 using *assms(2,3) store-typing-imp-inst-typing*
 by *blast*
 hence $k' < \text{length } (s.\text{tab } S)$
 using *k-def(1)*
 unfolding *inst-typing.simps stypes-def*
 by *auto*
 hence $k' < \text{length } (s.\text{tab } s)$
 using *assms(2) list-all2-lengthD*
 unfolding *store-typing.simps*
 by *fastforce*
 ultimately
 show *?thesis*
 using *k-def*
 by *auto*
qed

lemma *stab-typed-some-imp-cl-typed*:
assumes $stab\ s\ i\ c = Some\ cl$
store-typing $s\ \mathcal{S}$
 $i < length\ (inst\ s)$
shows $\exists\ tf.\ cl\text{-typing}\ \mathcal{S}\ cl\ tf$
proof –
have $Some\ cl \in set\ (concat\ (s.tab\ s))$
using *assms stab-typed-some-imp-member*
by *auto*
moreover
have *list-all (tab-agree \mathcal{S}) (concat (s.tab s))*
using *assms(2)*
unfolding *store-typing.simps*
by *auto*
ultimately
show *?thesis*
unfolding *in-set-conv-nth list-all-length tab-agree-def*
by *fastforce*
qed

lemma *b-e-type-empty1[dest]*: **assumes** $\mathcal{C} \vdash [] : (ts \rightarrow ts')$ **shows** $ts = ts'$
using *assms*
by (*induction []:(b-e list) (ts -> ts') arbitrary: ts ts' rule: b-e-typing.induct, simp-all*)

lemma *b-e-type-empty*: $(\mathcal{C} \vdash [] : (ts \rightarrow ts')) = (ts = ts')$
proof (*safe*)
assume $\mathcal{C} \vdash [] : (ts \rightarrow ts')$
thus $ts = ts'$
by *blast*
next
assume $ts = ts'$
thus $\mathcal{C} \vdash [] : (ts' \rightarrow ts')$
using *b-e-typing.empty b-e-typing.weakening*
by *fastforce*
qed

lemma *b-e-type-value*:
assumes $\mathcal{C} \vdash [e] : (ts \rightarrow ts')$
 $e = C\ v$
shows $ts' = ts @ [typeof\ v]$
using *assms*
by (*induction [e] (ts -> ts') arbitrary: ts ts' rule: b-e-typing.induct, auto*)

lemma *b-e-type-load*:
assumes $\mathcal{C} \vdash [e] : (ts \rightarrow ts')$
 $e = Load\ t\ tp\text{-sx}\ a\ off$
shows $\exists\ ts''\ sec\ n.\ ts = ts''@[T\text{-i32}] \wedge ts' = ts''@[t] \wedge (memory\ \mathcal{C}) = Some\ n$
load-store-t-bounds a (option-projl tp-sx) t

using *assms*
by (*induction* [e] (ts -> ts') *arbitrary: ts ts' rule: b-e-typing.induct, auto*)

lemma *b-e-type-store:*

assumes $\mathcal{C} \vdash [e] : (ts \rightarrow ts')$
 $e = \text{Store } t \text{ tp } a \text{ off}$
shows $ts = ts'@[T-i32, t]$
 $\exists \text{sec } n. (\text{memory } \mathcal{C}) = \text{Some } n$
load-store-t-bounds a tp t
using *assms*
by (*induction* [e] (ts -> ts') *arbitrary: ts ts' rule: b-e-typing.induct, auto*)

lemma *b-e-type-current-memory:*

assumes $\mathcal{C} \vdash [e] : (ts \rightarrow ts')$
 $e = \text{Current-memory}$
shows $\exists \text{sec } n. ts' = ts @ [T-i32] \wedge (\text{memory } \mathcal{C}) = \text{Some } n$
using *assms*
by (*induction* [e] (ts -> ts') *arbitrary: ts ts' rule: b-e-typing.induct, auto*)

lemma *b-e-type-grow-memory:*

assumes $\mathcal{C} \vdash [e] : (ts \rightarrow ts')$
 $e = \text{Grow-memory}$
shows $\exists ts''. ts = ts''@[T-i32] \wedge ts = ts' \wedge (\exists n. (\text{memory } \mathcal{C}) = \text{Some } n)$
using *assms*
by (*induction* [e] (ts -> ts') *arbitrary: ts ts' rule: b-e-typing.induct*) *auto*

lemma *b-e-type-nop:*

assumes $\mathcal{C} \vdash [e] : (ts \rightarrow ts')$
 $e = \text{Nop}$
shows $ts = ts'$
using *assms*
by (*induction* [e] (ts -> ts') *arbitrary: ts ts' rule: b-e-typing.induct, auto*)

definition *arity-2-result* :: $b-e \Rightarrow t$ **where**

arity-2-result op2 = (case op2 of
 $\text{Binop-i } t \Rightarrow t$
 $\text{Binop-f } t \Rightarrow t$
 $\text{Relop-i } t \Rightarrow T-i32$
 $\text{Relop-f } t \Rightarrow T-i32)$

lemma *b-e-type-binop-relop:*

assumes $\mathcal{C} \vdash [e] : (ts \rightarrow ts')$
 $e = \text{Binop-i } t \text{ iop} \vee e = \text{Binop-f } t \text{ fop} \vee e = \text{Relop-i } t \text{ irop} \vee e = \text{Relop-f}$
 $t \text{ frop}$
shows $\exists ts''. ts = ts''@[t,t] \wedge ts' = ts''@[arity-2-result(e)]$
 $e = \text{Binop-f } t \text{ fop} \Longrightarrow \text{is-float-t } t$
 $e = \text{Relop-f } t \text{ frop} \Longrightarrow \text{is-float-t } t$
using *assms*
unfolding *arity-2-result-def*

by (induction [e] (ts -> ts') arbitrary: ts ts' rule: b-e-typing.induct, auto)

lemma *b-e-type-testop-drop-cvt0*:

assumes $\mathcal{C} \vdash [e] : (ts \rightarrow ts')$

$e = \text{Testop } t \text{ testop} \vee e = \text{Drop} \vee e = \text{Cvtop } t1 \text{ cvtop } t2 \text{ sx}$

shows $ts \neq []$

using *assms*

by (induction [e] ts -> ts' arbitrary: ts' rule: b-e-typing.induct, auto)

definition *arity-1-result* :: $b-e \Rightarrow t$ where

arity-1-result op1 = (case op1 of
 Unop-i t - \Rightarrow t
 | *Unop-f* t - \Rightarrow t
 | *Testop* t - \Rightarrow *T-i32*
 | *Cvtop* t1 *Convert* - - \Rightarrow t1
 | *Cvtop* t1 *Reinterpret* - - \Rightarrow t1)

lemma *b-e-type-unop-testop*:

assumes $\mathcal{C} \vdash [e] : (ts \rightarrow ts')$

$e = \text{Unop-i } t \text{ iop} \vee e = \text{Unop-f } t \text{ fop} \vee e = \text{Testop } t \text{ testop}$

shows $\exists ts''. ts = ts''@[t] \wedge ts' = ts''@[arity-1-result e]$

$e = \text{Unop-f } t \text{ fop} \Longrightarrow \text{is-float-t } t$

using *assms int-float-disjoint*

unfolding *arity-1-result-def*

by (induction [e] (ts -> ts') arbitrary: ts ts' rule: b-e-typing.induct) fastforce+

lemma *b-e-type-cvtop*:

assumes $\mathcal{C} \vdash [e] : (ts \rightarrow ts')$

$e = \text{Cvtop } t1 \text{ cvtop } t \text{ sx}$

shows $\exists ts''. ts = ts''@[t] \wedge ts' = ts''@[arity-1-result e]$

$\text{cvtop} = \text{Convert} \Longrightarrow (t1 \neq t) \wedge (sx = \text{None}) = ((\text{is-float-t } t1 \wedge \text{is-float-t } t)$

$\vee (\text{is-int-t } t1 \wedge \text{is-int-t } t \wedge (t\text{-length } t1 < t\text{-length } t)))$

$\text{cvtop} = \text{Reinterpret} \Longrightarrow (t1 \neq t) \wedge t\text{-length } t1 = t\text{-length } t$

using *assms*

unfolding *arity-1-result-def*

by (induction [e] (ts -> ts') arbitrary: ts ts' rule: b-e-typing.induct, auto)

lemma *b-e-type-drop*:

assumes $\mathcal{C} \vdash [e] : (ts \rightarrow ts')$

$e = \text{Drop}$

shows $\exists t. ts = ts'@[t]$

using *assms b-e-type-testop-drop-cvt0*

by (induction [e] (ts -> ts') arbitrary: ts ts' rule: b-e-typing.induct, auto)

lemma *b-e-type-select*:

assumes $\mathcal{C} \vdash [e] : (ts \rightarrow ts')$

$e = \text{Select}$

shows $\exists ts'' t. ts = ts''@[t, t, T-i32] \wedge ts' = ts''@[t]$

using *assms*

by (induction [e] (ts -> ts') arbitrary: ts ts' rule: b-e-typing.induct, auto)

lemma *b-e-type-call*:

assumes $\mathcal{C} \vdash [e] : (ts \rightarrow ts')$

$e = \text{Call } i$

shows $i < \text{length}(\text{func-t } \mathcal{C})$

$\exists ts'' \text{ tf1 tf2. } ts = ts''@tf1 \wedge ts' = ts''@tf2 \wedge (\text{func-t } \mathcal{C})!i = (tf1 \rightarrow tf2)$

using *assms*

by (induction [e] (ts -> ts') arbitrary: ts ts' rule: b-e-typing.induct, auto)

lemma *b-e-type-call-indirect*:

assumes $\mathcal{C} \vdash [e] : (ts \rightarrow ts')$

$e = \text{Call-indirect } i$

shows $i < \text{length}(\text{types-t } \mathcal{C})$

$\exists ts'' \text{ tf1 tf2. } ts = ts''@tf1@[T-i32] \wedge ts' = ts''@tf2 \wedge (\text{types-t } \mathcal{C})!i = (tf1 \rightarrow tf2)$

using *assms*

by (induction [e] (ts -> ts') arbitrary: ts ts' rule: b-e-typing.induct, auto)

lemma *b-e-type-get-local*:

assumes $\mathcal{C} \vdash [e] : (ts \rightarrow ts')$

$e = \text{Get-local } i$

shows $\exists t. ts' = ts@[t] \wedge (\text{local } \mathcal{C})!i = t \wedge i < \text{length}(\text{local } \mathcal{C})$

using *assms*

by (induction [e] (ts -> ts') arbitrary: ts ts' rule: b-e-typing.induct, auto)

lemma *b-e-type-set-local*:

assumes $\mathcal{C} \vdash [e] : (ts \rightarrow ts')$

$e = \text{Set-local } i$

shows $\exists t. ts = ts'@[t] \wedge (\text{local } \mathcal{C})!i = t \wedge i < \text{length}(\text{local } \mathcal{C})$

using *assms*

by (induction [e] (ts -> ts') arbitrary: ts ts' rule: b-e-typing.induct, auto)

lemma *b-e-type-tee-local*:

assumes $\mathcal{C} \vdash [e] : (ts \rightarrow ts')$

$e = \text{Tee-local } i$

shows $\exists ts'' t. ts = ts''@[t] \wedge ts' = ts''@[t] \wedge (\text{local } \mathcal{C})!i = t \wedge i < \text{length}(\text{local } \mathcal{C})$

using *assms*

by (induction [e] (ts -> ts') arbitrary: ts ts' rule: b-e-typing.induct, auto)

lemma *b-e-type-get-global*:

assumes $\mathcal{C} \vdash [e] : (ts \rightarrow ts')$

$e = \text{Get-global } i$

shows $\exists t. ts' = ts@[t] \wedge \text{tg-t}((\text{global } \mathcal{C})!i) = t \wedge i < \text{length}(\text{global } \mathcal{C})$

using *assms*

by (induction [e] (ts -> ts') arbitrary: ts ts' rule: b-e-typing.induct, auto)

lemma *b-e-type-set-global*:

assumes $\mathcal{C} \vdash [e] : (ts \rightarrow ts')$

$e = \text{Set-global } i$
shows $\exists t. ts = ts'@[t] \wedge (\text{global } C)!i = (\text{tg-mut} = T\text{-mut}, \text{tg-t} = t) \wedge i < \text{length}(\text{global } C)$
using *assms is-mut-def*
by (*induction [e] (ts -> ts')* arbitrary: *ts ts' rule: b-e-typing.induct*) *auto*

lemma *b-e-type-block*:
assumes $C \vdash [e] : (ts \rightarrow ts')$
 $e = \text{Block } tf \ es$
shows $\exists ts'' \ tfn \ tfm. tf = (tfn \rightarrow tfm) \wedge (ts = ts''@tfn) \wedge (ts' = ts''@tfm) \wedge (C(\text{label} := [tfm] @ \text{label } C) \vdash es : tf)$
using *assms*
by (*induction [e] (ts -> ts')* arbitrary: *ts ts' rule: b-e-typing.induct, auto*)

lemma *b-e-type-loop*:
assumes $C \vdash [e] : (ts \rightarrow ts')$
 $e = \text{Loop } tf \ es$
shows $\exists ts'' \ tfn \ tfm. tf = (tfn \rightarrow tfm) \wedge (ts = ts''@tfn) \wedge (ts' = ts''@tfm) \wedge (C(\text{label} := [tfn] @ \text{label } C) \vdash es : tf)$
using *assms*
by (*induction [e] (ts -> ts')* arbitrary: *ts ts' rule: b-e-typing.induct, auto*)

lemma *b-e-type-if*:
assumes $C \vdash [e] : (ts \rightarrow ts')$
 $e = \text{If } tf \ es1 \ es2$
shows $\exists ts'' \ tfn \ tfm. tf = (tfn \rightarrow tfm) \wedge (ts = ts''@tfn @ [T-i32]) \wedge (ts' = ts''@tfm) \wedge (C(\text{label} := [tfm] @ \text{label } C) \vdash es1 : tf) \wedge (C(\text{label} := [tfm] @ \text{label } C) \vdash es2 : tf)$
using *assms*
by (*induction [e] (ts -> ts')* arbitrary: *ts ts' rule: b-e-typing.induct, auto*)

lemma *b-e-type-br*:
assumes $C \vdash [e] : (ts \rightarrow ts')$
 $e = \text{Br } i$
shows $i < \text{length}(\text{label } C)$
 $\exists ts\text{-}c \ ts''. ts = ts\text{-}c @ ts'' \wedge (\text{label } C)!i = ts''$
using *assms*
by (*induction [e] (ts -> ts')* arbitrary: *ts ts' rule: b-e-typing.induct, auto*)

lemma *b-e-type-br-if*:
assumes $C \vdash [e] : (ts \rightarrow ts')$
 $e = \text{Br-if } i$
shows $i < \text{length}(\text{label } C)$
 $\exists ts\text{-}c \ ts''. ts = ts\text{-}c @ ts'' @ [T-i32] \wedge ts' = ts\text{-}c @ ts'' \wedge (\text{label } C)!i = ts''$
using *assms*
by (*induction [e] (ts -> ts')* arbitrary: *ts ts' rule: b-e-typing.induct, auto*)

lemma *b-e-type-br-table*:
assumes $\mathcal{C} \vdash [e] : (ts \rightarrow ts')$
 $e = \text{Br-table } i$
shows $\exists ts\text{-}c \ ts''. \text{list-all } (\lambda i. i < \text{length}(\text{label } \mathcal{C}) \wedge (\text{label } \mathcal{C})!i = ts'') (is@[i]) \wedge$
 $ts = ts\text{-}c @ ts''@[T\text{-}i32]$
using *assms*
by (*induction* $[e] (ts \rightarrow ts')$ *arbitrary: ts ts' rule: b-e-typing.induct, fastforce+*)

lemma *b-e-type-return*:
assumes $\mathcal{C} \vdash [e] : (ts \rightarrow ts')$
 $e = \text{Return}$
shows $\exists ts\text{-}c \ ts''. ts = ts\text{-}c @ ts'' \wedge (\text{return } \mathcal{C}) = \text{Some } ts''$
using *assms*
by (*induction* $[e] (ts \rightarrow ts')$ *arbitrary: ts ts' rule: b-e-typing.induct, auto*)

lemma *b-e-type-comp*:
assumes $\mathcal{C} \vdash es@[e] : (t1s \rightarrow t4s)$
shows $\exists ts'. (\mathcal{C} \vdash es : (t1s \rightarrow ts')) \wedge (\mathcal{C} \vdash [e] : (ts' \rightarrow t4s))$
proof (*cases es rule: List.rev-cases*)
case *Nil*
then show *?thesis*
using *assms b-e-typing.empty b-e-typing.weakening*
by *fastforce*
next
case (*snoc es' e'*)
show *?thesis using assms snoc b-e-typing.weakening*
by (*induction es@[e] (t1s \rightarrow t4s) arbitrary: t1s t4s, fastforce+*)
qed

lemma *b-e-type-comp2-unlift*:
assumes $\mathcal{S}\cdot\mathcal{C} \vdash [\$e1, \$e2] : (t1s \rightarrow t2s)$
shows $\exists ts'. (\mathcal{C} \vdash [e1] : (t1s \rightarrow ts')) \wedge (\mathcal{C} \vdash [e2] : (ts' \rightarrow t2s))$
using *assms*
 $\text{unlift-b-e}[of \ \mathcal{S} \ \mathcal{C} \ ([e1, e2]) \ (t1s \rightarrow t2s)]$
 $\text{b-e-type-comp}[of \ \mathcal{C} \ [e1] \ e2 \ t1s \ t2s]$
by *simp*

lemma *b-e-type-comp2-relift*:
assumes $\mathcal{C} \vdash [e1] : (t1s \rightarrow ts') \ \mathcal{C} \vdash [e2] : (ts' \rightarrow t2s)$
shows $\mathcal{S}\cdot\mathcal{C} \vdash [\$e1, \$e2] : (ts@t1s \rightarrow ts@t2s)$
using *assms*
 $\text{b-e-typing.composition}[OF \ \text{assms}]$
 $\text{e-typing-s-typing.intros}(1)[of \ \mathcal{C} \ [e1, e2] \ (t1s \rightarrow t2s)]$
 $\text{e-typing-s-typing.intros}(3)[of \ \mathcal{S} \ \mathcal{C} \ ([\$e1, \$e2]) \ t1s \ t2s \ ts]$
by *simp*

lemma *b-e-type-value2*:
assumes $\mathcal{C} \vdash [C \ v1, C \ v2] : (t1s \rightarrow t2s)$

shows $t2s = t1s @ [typeof\ v1, typeof\ v2]$
proof –
obtain ts' **where** $ts'\text{-def}:\mathcal{C} \vdash [C\ v1] : (t1s \rightarrow ts')$
 $\mathcal{C} \vdash [C\ v2] : (ts' \rightarrow t2s)$
using *b-e-type-comp assms*
by (*metis append-butlast-last-id butlast.simps(2) last-ConsL last-ConsR list.distinct(1)*)
have $ts' = t1s @ [typeof\ v1]$
using *b-e-type-value ts'-def(1)*
by *fastforce*
thus *?thesis*
using *b-e-type-value ts'-def(2)*
by *fastforce*
qed

lemma *e-type-comp*:
assumes $\mathcal{S}\cdot\mathcal{C} \vdash es@[e] : (t1s \rightarrow t3s)$
shows $\exists ts'. (\mathcal{S}\cdot\mathcal{C} \vdash es : (t1s \rightarrow ts')) \wedge (\mathcal{S}\cdot\mathcal{C} \vdash [e] : (ts' \rightarrow t3s))$
proof (*cases es rule: List.rev-cases*)
case *Nil*
thus *?thesis*
using *assms e-typing-s-typing.intros(1)*
by (*metis append-Nil b-e-type-empty list.simps(8)*)
next
case (*snoc es' e'*)
show *?thesis using assms snoc*
proof (*induction es@[e] (t1s -> t3s) arbitrary: t1s t3s*)
case (*1 C b-es S*)
obtain $es''\ e''$ **where** $b\text{-e-defs}:(\$* (es'' @ [e''])) = (\$* b\text{-es})$
using *1(1,2)*
by (*metis Nil-is-map-conv append-is-Nil-conv not-Cons-self2 rev-exhaust*)
hence $(\$*es'') = es (\$e'') = e$
using *1(2) inj-basic map-injective*
by *auto*
moreover
have $\mathcal{C} \vdash (es'' @ [e'']) : (t1s \rightarrow t3s)$ **using** *1(1)*
using *inj-basic map-injective b-e-defs*
by *blast*
then obtain $t2s$ **where** $\mathcal{C} \vdash es'' : (t1s \rightarrow t2s)\ \mathcal{C} \vdash [e''] : (t2s \rightarrow t3s)$
using *b-e-type-comp*
by *blast*
ultimately
show *?case*
using *e-typing-s-typing.intros(1)*
by *fastforce*
next
case (*3 S C t1s t2s ts*)
thus *?case*
using *e-typing-s-typing.intros(3)*

by *fastforce*
qed *auto*
qed

lemma *e-type-comp-conc*:
assumes $\mathcal{S}\cdot\mathcal{C} \vdash es : (t1s \rightarrow t2s)$
 $\mathcal{S}\cdot\mathcal{C} \vdash es' : (t2s \rightarrow t3s)$
shows $\mathcal{S}\cdot\mathcal{C} \vdash es@es' : (t1s \rightarrow t3s)$
using *assms(2)*
proof (*induction es' arbitrary: t3s rule: List.rev-induct*)
case *Nil*
hence $t2s = t3s$
using *unlift-b-e[of - - []] b-e-type-empty[of - t2s t3s]*
by *fastforce*
then show *?case*
using *Nil assms(1) e-typing-s-typing.intros(2)*
by *fastforce*
next
case (*snoc x xs*)
then obtain ts' **where** $\mathcal{S}\cdot\mathcal{C} \vdash xs : (t2s \rightarrow ts')$ $\mathcal{S}\cdot\mathcal{C} \vdash [x] : (ts' \rightarrow t3s)$
using *e-type-comp[of - - xs x]*
by *fastforce*
then show *?case*
using *snoc(1)[of ts'] e-typing-s-typing.intros(2)[of - - es @ xs t1s ts' x t3s]*
by *simp*
qed

lemma *b-e-type-comp-conc*:
assumes $\mathcal{C} \vdash es : (t1s \rightarrow t2s)$
 $\mathcal{C} \vdash es' : (t2s \rightarrow t3s)$
shows $\mathcal{C} \vdash es@es' : (t1s \rightarrow t3s)$
proof –
fix \mathcal{S}
have $1:\mathcal{S}\cdot\mathcal{C} \vdash \$*es : (t1s \rightarrow t2s)$
using *e-typing-s-typing.intros(1)[OF assms(1)]*
by *fastforce*
have $2:\mathcal{S}\cdot\mathcal{C} \vdash \$*es' : (t2s \rightarrow t3s)$
using *e-typing-s-typing.intros(1)[OF assms(2)]*
by *fastforce*
show *?thesis*
using *e-type-comp-conc[OF 1 2]*
by (*simp add: unlift-b-e*)
qed

lemma *e-type-comp-conc1*:
assumes $\mathcal{S}\cdot\mathcal{C} \vdash es@es' : (ts \rightarrow ts')$
shows $\exists ts'' . (\mathcal{S}\cdot\mathcal{C} \vdash es : (ts \rightarrow ts'')) \wedge (\mathcal{S}\cdot\mathcal{C} \vdash es' : (ts'' \rightarrow ts'))$
using *assms*

```

proof (induction es' arbitrary: ts ts' rule: List.rev-induct)
  case Nil
  thus ?case
  using b-e-type-empty[of - ts' ts'] e-typing-s-typing.intros(1)
  by fastforce
next
  case (snoc x xs)
  then show ?case
  using e-type-comp[of  $\mathcal{S} \mathcal{C}$  es @ xs x ts ts'] e-typing-s-typing.intros(2)[of  $\mathcal{S} \mathcal{C}$  xs
- - x ts']
  by fastforce
qed

```

```

lemma e-type-comp-conc2:
  assumes  $\mathcal{S} \cdot \mathcal{C} \vdash es @ es' @ es'' : (t1s \rightarrow t2s)$ 
  shows  $\exists ts' ts''.$  ( $\mathcal{S} \cdot \mathcal{C} \vdash es : (t1s \rightarrow ts')$ )
     $\wedge$  ( $\mathcal{S} \cdot \mathcal{C} \vdash es' : (ts' \rightarrow ts'')$ )
     $\wedge$  ( $\mathcal{S} \cdot \mathcal{C} \vdash es'' : (ts'' \rightarrow t2s)$ )

```

```

proof -
  obtain ts' where  $\mathcal{S} \cdot \mathcal{C} \vdash es : (t1s \rightarrow ts')$   $\mathcal{S} \cdot \mathcal{C} \vdash es' @ es'' : (ts' \rightarrow t2s)$ 
  using assms(1) e-type-comp-conc1
  by fastforce
  moreover
  then obtain ts'' where  $\mathcal{S} \cdot \mathcal{C} \vdash es' : (ts' \rightarrow ts'')$   $\mathcal{S} \cdot \mathcal{C} \vdash es'' : (ts'' \rightarrow t2s)$ 
  using e-type-comp-conc1
  by fastforce
  ultimately
  show ?thesis
  by fastforce
qed

```

```

lemma b-e-type-value-list:
  assumes ( $\mathcal{C} \vdash es @ [C v] : (ts \rightarrow ts' @ [t])$ )
  shows ( $\mathcal{C} \vdash es : (ts \rightarrow ts')$ )
    ( $typeof v = t$ )

```

```

proof -
  obtain ts'' where ( $\mathcal{C} \vdash es : (ts \rightarrow ts'')$ ) ( $\mathcal{C} \vdash [C v] : (ts'' \rightarrow ts' @ [t])$ )
  using b-e-type-comp assms
  by blast
  thus ( $\mathcal{C} \vdash es : (ts \rightarrow ts')$ ) ( $typeof v = t$ )
  using b-e-type-value[of  $\mathcal{C}$  C v ts'' ts' @ [t]]
  by auto
qed

```

```

lemma e-type-label:
  assumes  $\mathcal{S} \cdot \mathcal{C} \vdash [Label n es0 es] : (ts \rightarrow ts')$ 
  shows  $\exists t1s t2s.$  ( $ts' = (ts @ t2s)$ )
     $\wedge$  length t1s = n
     $\wedge$  ( $\mathcal{S} \cdot \mathcal{C} \vdash es0 : (t1s \rightarrow t2s)$ )

```

```

       $\wedge (\mathcal{S}\cdot\mathcal{C}(\text{label} := [t1s] @ (\text{label } \mathcal{C})) \vdash es : ([ ] \rightarrow t2s))$ 
using assms
proof (induction  $\mathcal{S} \mathcal{C} [\text{Label } n \text{ es0 } es] (ts \rightarrow ts')$  arbitrary: ts ts')
  case (1  $\mathcal{C} \text{ b-es } \mathcal{S}$ )
  then show ?case
    by (simp add: map-eq-Cons-conv)
next
  case (2  $\mathcal{S} \mathcal{C} \text{ es } t1s \ t2s \ e \ t3s$ )
  then show ?case
    by (metis append-self-conv2 b-e-type-empty last-snoc list.simps(8) unlift-b-e)
next
  case (3  $\mathcal{S} \mathcal{C} \ t1s \ t2s \ ts$ )
  then show ?case
    by simp
next
  case (7  $\mathcal{S} \mathcal{C} \ t2s$ )
  then show ?case
    by fastforce
qed

lemma e-type-callcl-native:
  assumes  $\mathcal{S}\cdot\mathcal{C} \vdash [\text{Callcl } cl] : (t1s' \rightarrow t2s')$ 
     $cl = \text{Func-native } i \ \text{tf} \ ts \ es$ 
  shows  $\exists t1s \ t2s \ ts\text{-c}. (t1s' = ts\text{-c} @ t1s)$ 
     $\wedge (t2s' = ts\text{-c} @ t2s)$ 
     $\wedge \text{tf} = (t1s \rightarrow t2s)$ 
     $\wedge i < \text{length } (s\text{-inst } \mathcal{S})$ 
     $\wedge (((s\text{-inst } \mathcal{S})!i)(\text{local} := (\text{local } ((s\text{-inst } \mathcal{S})!i)) @ t1s @ ts, \text{label}$ 
  :=  $([t2s] @ (\text{label } ((s\text{-inst } \mathcal{S})!i))), \text{return} := \text{Some } t2s) \vdash es : ([ ] \rightarrow t2s))$ 
  using assms
proof (induction  $\mathcal{S} \mathcal{C} [\text{Callcl } cl] (t1s' \rightarrow t2s')$  arbitrary: t1s' t2s')
  case (1  $\mathcal{C} \text{ b-es } \mathcal{S}$ )
  thus ?case
    by auto
next
  case (2  $\mathcal{S} \mathcal{C} \text{ es } t1s \ t2s \ e \ t3s$ )
  have  $\mathcal{C} \vdash [ ] : (t1s \rightarrow t2s)$ 
    using 2(1,5) unlift-b-e
    by (metis Nil-is-map-conv append-Nil butlast-snoc)
  thus ?case
    using 2(4,5,6)
    by fastforce
next
  case (3  $\mathcal{S} \mathcal{C} \ t1s \ t2s \ ts$ )
  thus ?case
    by fastforce
next
  case (6  $\mathcal{S} \mathcal{C}$ )
  thus ?case

```

```

    unfolding cl-typing.simps
    by fastforce
qed

lemma e-type-callcl-host:
  assumes  $\mathcal{S}\cdot\mathcal{C} \vdash [\text{Callcl } cl] : (t1s' \rightarrow t2s')$ 
     $cl = \text{Func-host } tf$ 
  shows  $\exists t1s \ t2s \ ts\text{-}c. (t1s' = ts\text{-}c @ t1s)$ 
     $\wedge (t2s' = ts\text{-}c @ t2s)$ 
     $\wedge tf = (t1s \rightarrow t2s)$ 

  using assms
proof (induction  $\mathcal{S} \ \mathcal{C} \ [\text{Callcl } cl] \ (t1s' \rightarrow t2s')$  arbitrary:  $t1s' \ t2s'$ )
  case (1  $\mathcal{C} \ b\text{-}es \ \mathcal{S}$ )
    thus ?case
      by auto
  next
  case (2  $\mathcal{S} \ \mathcal{C} \ es \ t1s \ t2s \ e \ t3s$ )
    have  $\mathcal{C} \vdash [] : (t1s \rightarrow t2s)$ 
      using 2(1,5) unlift-b-e
    by (metis Nil-is-map-conv append-Nil butlast-snoc)
    thus ?case
      using 2(4,5,6)
      by fastforce
  next
  case (3  $\mathcal{S} \ \mathcal{C} \ t1s \ t2s \ ts$ )
    thus ?case
      by fastforce
  next
  case (6  $\mathcal{S} \ \mathcal{C}$ )
    thus ?case
      unfolding cl-typing.simps
      by fastforce
qed

```

```

lemma e-type-callcl:
  assumes  $\mathcal{S}\cdot\mathcal{C} \vdash [\text{Callcl } cl] : (t1s' \rightarrow t2s')$ 
  shows  $\exists t1s \ t2s \ ts\text{-}c. (t1s' = ts\text{-}c @ t1s)$ 
     $\wedge (t2s' = ts\text{-}c @ t2s)$ 
     $\wedge cl\text{-}type \ cl = (t1s \rightarrow t2s)$ 

proof (cases cl)
  case (Func-native  $x11 \ x12 \ x13 \ x14$ )
    thus ?thesis
      using e-type-callcl-native[OF assms]
      unfolding cl-type-def
      by (cases  $x12$ ) fastforce
  next
  case (Func-host  $x21 \ x22$ )
    thus ?thesis
      using e-type-callcl-host[OF assms]

```

unfolding *cl-type-def*
by *fastforce*
qed

lemma *s-type-unfold*:

assumes $\mathcal{S} \cdot rs \Vdash -i \text{ vs}; es : ts$
shows $i < \text{length } (s\text{-inst } \mathcal{S})$
 $(rs = \text{Some } ts) \vee rs = \text{None}$
 $(\mathcal{S} \cdot ((s\text{-inst } \mathcal{S})!i) \text{ local} := (\text{local } ((s\text{-inst } \mathcal{S})!i)) @ (\text{map typeof } vs), \text{return} :=$
 $rs) \vdash es : ([\] \rightarrow ts)$
using *assms*
by (*induction vs es ts, auto*)

lemma *e-type-local*:

assumes $\mathcal{S} \cdot \mathcal{C} \vdash [\text{Local } n \ i \ \text{vs } es] : (ts \rightarrow ts')$
shows $\exists t1s. i < \text{length } (s\text{-inst } \mathcal{S})$
 $\wedge \text{length } t1s = n$
 $\wedge (\mathcal{S} \cdot ((s\text{-inst } \mathcal{S})!i) \text{ local} := (\text{local } ((s\text{-inst } \mathcal{S})!i)) @ (\text{map typeof } vs),$
 $\text{return} := \text{Some } t1s) \vdash es : ([\] \rightarrow t1s)$
 $\wedge ts' = ts @ t1s$

using *assms*

proof (*induction* $\mathcal{S} \ \mathcal{C} \ [\text{Local } n \ i \ \text{vs } es] \ (ts \rightarrow ts') \ \text{arbitrary: } ts \ ts'$)

case ($2 \ \mathcal{S} \ \mathcal{C} \ es' \ t1s \ t2s \ e \ t3s$)

have $t1s = t2s$

using $2 \ \text{unlift-b-e}$

by *force*

thus $?case$

using 2

by *simp*

qed (*auto simp add: unlift-b-e s-typing.simps*)

lemma *e-type-local-shallow*:

assumes $\mathcal{S} \cdot \mathcal{C} \vdash [\text{Local } n \ i \ \text{vs } es] : (ts \rightarrow ts')$
shows $\exists t1s. \text{length } t1s = n \wedge ts' = ts @ t1s \wedge (\mathcal{S} \cdot (\text{Some } t1s) \Vdash -i \ \text{vs}; es : t1s)$
using *assms*

proof (*induction* $\mathcal{S} \ \mathcal{C} \ [\text{Local } n \ i \ \text{vs } es] \ (ts \rightarrow ts') \ \text{arbitrary: } ts \ ts'$)

case ($1 \ \mathcal{C} \ b\text{-es } \mathcal{S}$)

thus $?case$

by (*metis e.distinct(7) map-eq-Cons-D*)

next

case ($2 \ \mathcal{S} \ \mathcal{C} \ es \ t1s \ t2s \ e \ t3s$)

thus $?case$

by *simp (metis append-Nil append-eq-append-conv e-type-comp-conc e-type-local)*

qed *simp-all*

lemma *e-type-const-unwrap*:

assumes *is-const e*

shows $\exists v. e = \$C \ v$

```

using assms
proof (cases e)
  case (Basic x1)
  then show ?thesis
    using assms
  proof (cases x1)
    case (EConst x23)
    thus ?thesis
      using Basic e-typing-s-typing.intros(1,3)
      by fastforce
  qed (simp-all add: is-const-def)
qed (simp-all add: is-const-def)

lemma is-const-list1:
  assumes ves = map (Basic ∘ EConst) vs
  shows const-list ves
  using assms
proof (induction vs arbitrary: ves)
  case Nil
  then show ?case
    unfolding const-list-def
    by simp
next
  case (Cons a vs)
  then obtain ves' where ves' = map (Basic ∘ EConst) vs
    by blast
  moreover
  have is-const ((Basic ∘ EConst) a)
    unfolding is-const-def
    by simp
  ultimately
  show ?case
    using Cons
    unfolding const-list-def

    by auto
qed

lemma is-const-list:
  assumes ves = $$* vs
  shows const-list ves
  using assms is-const-list1
  unfolding comp-def
  by auto

lemma const-list-cons-last:
  assumes const-list (es@[e])
  shows const-list es
    is-const e

```

```

using assms list-all-append[of is-const es [e]]
unfolding const-list-def
by auto

lemma e-type-const1:
  assumes is-const e
  shows  $\exists t. (\mathcal{S}\cdot\mathcal{C} \vdash [e] : (ts \rightarrow ts@[t]))$ 
  using assms
proof (cases e)
  case (Basic x1)
  then show ?thesis
    using assms
  proof (cases x1)
  case (EConst x23)
    hence  $\mathcal{C} \vdash [x1] : ([\ ] \rightarrow [typeof\ x23])$ 
    by (simp add: b-e-typing.intros(1))
    thus ?thesis
    using Basic e-typing-s-typing.intros(1,3)
    by (metis append-Nil2 to-e-list-1)
  qed (simp-all add: is-const-def)
qed (simp-all add: is-const-def)

lemma e-type-const:
  assumes is-const e
     $\mathcal{S}\cdot\mathcal{C} \vdash [e] : (ts \rightarrow ts')$ 
  shows  $\exists t. (ts' = ts@[t]) \wedge (\mathcal{S}\cdot\mathcal{C}' \vdash [e] : ([\ ] \rightarrow [t]))$ 
  using assms
proof (cases e)
  case (Basic x1)
  then show ?thesis
    using assms
  proof (cases x1)
  case (EConst x23)
    then have  $ts' = ts @ [typeof\ x23]$ 
    by (metis (no-types) Basic assms(2) b-e-type-value list.simps(8,9) unlift-b-e)
    moreover
    have  $\mathcal{S}\cdot\mathcal{C}' \vdash [e] : ([\ ] \rightarrow [typeof\ x23])$ 
    using Basic EConst b-e-typing.intros(1) e-typing-s-typing.intros(1)
    by fastforce
    ultimately
    show ?thesis
    by simp
  qed (simp-all add: is-const-def)
qed (simp-all add: is-const-def)

lemma const-typeof:
  assumes  $\mathcal{S}\cdot\mathcal{C} \vdash [\$C\ v] : ([\ ] \rightarrow [t])$ 
  shows  $typeof\ v = t$ 
  using assms

```

```

proof –
  have  $\mathcal{C} \vdash [C\ v] : ([\ ] \rightarrow [t])$ 
    using unlift-b-e assms
    by fastforce
  thus ?thesis
    by (induction  $[C\ v] ([\ ] \rightarrow [t])$  rule: b-e-typing.induct, auto)
qed

lemma e-type-const-list:
  assumes const-list vs
     $\mathcal{S}\cdot\mathcal{C} \vdash vs : (ts \rightarrow ts')$ 
  shows  $\exists tvs. ts' = ts @ tvs \wedge \text{length } vs = \text{length } tvs \wedge (\mathcal{S}\cdot\mathcal{C}' \vdash vs : ([\ ] \rightarrow tvs))$ 
  using assms
proof (induction vs arbitrary: ts ts' rule: List.rev-induct)
  case Nil
  have  $\mathcal{S}\cdot\mathcal{C}' \vdash [\ ] : ([\ ] \rightarrow [\ ])$ 
    using b-e-type-empty[of C' [] [] e-typing-s-typing.intros(1)]
    by fastforce
  thus ?case
    using Nil
    by (metis append-Nil2 b-e-type-empty list.map(1) list.size(3) unlift-b-e)
next
  case (snoc x xs)
  hence v-lists:list-all is-const xs is-const x
  unfolding const-list-def
  by simp-all
  obtain ts'' where ts''-def:  $\mathcal{S}\cdot\mathcal{C} \vdash xs : (ts \rightarrow ts'')$   $\mathcal{S}\cdot\mathcal{C} \vdash [x] : (ts'' \rightarrow ts')$ 
    using snoc(3) e-type-comp
    by fastforce
  then obtain ts-b where ts-b-def:  $ts'' = ts @ ts-b$   $\text{length } xs = \text{length } ts-b$   $\mathcal{S}\cdot\mathcal{C}' \vdash$ 
  xs : ([\ ] \rightarrow ts-b)
    using snoc(1) v-lists(1)
    unfolding const-list-def
    by fastforce
  then obtain t where t-def:  $ts' = ts @ ts-b @ [t]$   $\mathcal{S}\cdot\mathcal{C}' \vdash [x] : ([\ ] \rightarrow [t])$ 
    using e-type-const v-lists(2) ts''-def
    by fastforce
  moreover
  then have  $\text{length } (ts-b@[t]) = \text{length } (xs@[x])$ 
    using ts-b-def(2)
    by simp
  moreover
  have  $\mathcal{S}\cdot\mathcal{C}' \vdash (xs@[x]) : ([\ ] \rightarrow ts-b@[t])$ 
    using ts-b-def(3) t-def e-typing-s-typing.intros(2,3)
    by fastforce
  ultimately
  show ?case
    by simp
qed

```

lemma *e-type-const-list-snoc*:

assumes *const-list vs*

$\mathcal{S}\cdot\mathcal{C} \vdash vs : (\square \rightarrow ts@[t])$

shows $\exists vs1\ v2. (\mathcal{S}\cdot\mathcal{C} \vdash vs1 : (\square \rightarrow ts))$
 $\wedge (\mathcal{S}\cdot\mathcal{C} \vdash [v2] : (ts \rightarrow ts@[t]))$
 $\wedge (vs = vs1@[v2])$
 $\wedge \text{const-list } vs1$
 $\wedge \text{is-const } v2$

using *assms*

proof –

obtain *vs' v* **where** *vs-def:vs = vs'@[v]*

using *e-type-const-list[OF assms(1,2)]*

by (*metis append-Nil append-eq-append-conv list.size(3) snoc-eq-iff-butlast*)

hence *consts-def:const-list vs' is-const v*

using *assms(1)*

unfolding *const-list-def*

by *auto*

obtain *ts'* **where** *ts'-def:mathcal{S}\cdot\mathcal{C} \vdash vs' : (\square \rightarrow ts')* $\mathcal{S}\cdot\mathcal{C} \vdash [v] : (ts' \rightarrow ts@[t])$

using *vs-def assms(2) e-type-comp[of mathcal{S} mathcal{C} vs' v [] ts@[t]]*

by *fastforce*

obtain *c* **where** $v = \$C\ c$

using *e-type-const-unwrap consts-def(2)*

by *fastforce*

hence $ts' = ts$

using *ts'-def(2) unlift-b-e[of mathcal{S} mathcal{C} [C c] b-e-type-value*

by *fastforce*

thus *?thesis using ts'-def vs-def consts-def*

by *simp*

qed

lemma *e-type-const-list-cons*:

assumes *const-list vs*

$\mathcal{S}\cdot\mathcal{C} \vdash vs : (\square \rightarrow (ts1@ts2))$

shows $\exists vs1\ vs2. (\mathcal{S}\cdot\mathcal{C} \vdash vs1 : (\square \rightarrow ts1))$
 $\wedge (\mathcal{S}\cdot\mathcal{C} \vdash vs2 : (ts1 \rightarrow (ts1@ts2)))$
 $\wedge vs = vs1@vs2$
 $\wedge \text{const-list } vs1$
 $\wedge \text{const-list } vs2$

using *assms*

proof (*induction ts1@ts2 arbitrary: vs ts1 ts2 rule: List.rev-induct*)

case *Nil*

thus *?case*

using *e-type-const-list*

by *fastforce*

next

case (*snoc t ts*)

note *snoc-outer = snoc*

show *?case*

```

proof (cases ts2 rule: List.rev-cases)
  case Nil
  have  $\mathcal{S}\cdot\mathcal{C} \vdash [] : (ts1 \rightarrow ts1 @ [])$ 
    using b-e-typing.empty b-e-typing.weakening e-typing-s-typing.intros(1)
    by fastforce
  then show ?thesis
    using snoc(3,4) Nil
    unfolding const-list-def
    by auto
  next
  case (snoc ts2' a)
  obtain vs1 v2 where vs1-def:( $\mathcal{S}\cdot\mathcal{C} \vdash vs1 : ([] \rightarrow ts1 @ ts2')$ )
    ( $\mathcal{S}\cdot\mathcal{C} \vdash [v2] : (ts1 @ ts2' \rightarrow ts1 @ ts2' @[t])$ )
    ( $vs = vs1@[v2]$ )
    const-list vs1
    is-const v2
     $ts = ts1 @ ts2'$ 
  using e-type-const-list-snoc[OF snoc-outer(3), of  $\mathcal{S} \ \mathcal{C} \ ts1@ts2' \ t$ ]
    snoc-outer(2,4) snoc
  by fastforce
  show ?thesis
  using snoc-outer(1)[OF vs1-def(6,4,1)] snoc-outer(2) vs1-def(3,5)
    e-typing-s-typing.intros(2)[OF - vs1-def(2), of - ts1]
    snoc
  unfolding const-list-def
  by fastforce
  qed
qed

```

```

lemma e-type-const-conv-vs:
  assumes const-list ves
  shows  $\exists vs. ves = \$\$* \ vs$ 
  using assms
proof (induction ves)
  case Nil
  thus ?case
    by simp
  next
  case (Cons a ves)
  thus ?case
  using e-type-const-unwrap
  unfolding const-list-def
  by (metis (no-types, lifting) list.pred-inject(2) list.simps(9))
qed

```

```

lemma types-exist-lfilled:
  assumes Lfilled k lholed es lfilled
     $\mathcal{S}\cdot\mathcal{C} \vdash \text{lfilled} : (ts \rightarrow ts')$ 
  shows  $\exists t1s \ t2s \ C' \text{ arb-label. } (\mathcal{S}\cdot\mathcal{C}(\text{label} := \text{arb-label}@(\text{label } C))) \vdash es : (t1s \rightarrow$ 

```

```

t2s))
  using assms
proof (induction arbitrary: C ts ts' rule: Lfilled.induct)
  case (L0 vs lholed es' es)
  hence  $\mathcal{S}\cdot\mathcal{C}(\text{label} := \text{label } C) \vdash \text{vs} @ \text{es} @ \text{es}' : (ts \rightarrow ts')$ 
  by simp
  thus ?case
  using e-type-comp-conc2
  by (metis append-Nil)
next
  case (LN vs lholed n es' l es'' k es lfilledk)
  obtain  $ts'' ts'''$  where  $\mathcal{S}\cdot\mathcal{C} \vdash [\text{Label } n \text{ es}' \text{ lfilledk}] : (ts'' \rightarrow ts''')$ 
  using e-type-comp-conc2[OF LN(5)]
  by fastforce
  then obtain  $t1s \ t2s \ ts$  where  $\text{test}:\mathcal{S}\cdot\mathcal{C}(\text{label} := [ts] @ (\text{label } C)) \vdash \text{lfilledk} : (t1s \rightarrow t2s)$ 
  using e-type-label
  by metis
  show ?case
  using LN(4)[OF test(1)]
  by simp (metis append.assoc append-Cons append-Nil)
qed

```

```

lemma types-exist-lfilled-weak:
  assumes Lfilled k lholed es lfilled
   $\mathcal{S}\cdot\mathcal{C} \vdash \text{lfilled} : (ts \rightarrow ts')$ 
  shows  $\exists t1s \ t2s \ C' \text{ arb-label } \text{arb-return}. (\mathcal{S}\cdot\mathcal{C}(\text{label} := \text{arb-label}, \text{return} := \text{arb-return})) \vdash \text{es} : (t1s \rightarrow t2s)$ 
proof -
  have  $\exists t1s \ t2s \ C' \text{ arb-label}. (\mathcal{S}\cdot\mathcal{C}(\text{label} := \text{arb-label}, \text{return} := (\text{return } C))) \vdash \text{es} : (t1s \rightarrow t2s)$ 
  using types-exist-lfilled[OF assms]
  by fastforce
  thus ?thesis
  by fastforce
qed

```

```

lemma store-typing-imp-func-agree:
  assumes store-typing s S
   $i < \text{length } (s\text{-inst } S)$ 
   $j < \text{length } (\text{func-t } ((s\text{-inst } S)!i))$ 
  shows  $(\text{sfunc-ind } s \ i \ j) < \text{length } (s\text{-funcs } S)$ 
   $\text{cl-typing } S \ (\text{sfunc } s \ i \ j) \ ((s\text{-funcs } S)!(\text{sfunc-ind } s \ i \ j))$ 
   $((s\text{-funcs } S)!(\text{sfunc-ind } s \ i \ j)) = (\text{func-t } ((s\text{-inst } S)!i))!j$ 
proof -
  have  $\text{funcs-agree:list-all2 } (\text{cl-typing } S) \ (\text{funcs } s) \ (s\text{-funcs } S)$ 
  using assms(1)
  unfolding store-typing.simps
  by auto

```

have $list\text{-}all2$ ($funci\text{-}agree$ ($s\text{-}funcs$ \mathcal{S})) ($inst.funcs$ ($(inst\ s)!i$)) ($func\text{-}t$ ($(s\text{-}inst\ \mathcal{S})!i$))
using $assms(1,2)$ $store\text{-}typing\text{-}imp\text{-}inst\text{-}length\text{-}eq$ $store\text{-}typing\text{-}imp\text{-}inst\text{-}typing$
by ($fastforce$ $simp$ $add: inst\text{-}typing.simps$)
hence $funci\text{-}agree$ ($s\text{-}funcs$ \mathcal{S}) ($(inst.funcs$ ($(inst\ s)!i$))! j) ($(func\text{-}t$ ($(s\text{-}inst\ \mathcal{S})!i$))! j)
using $assms(3)$ $list\text{-}all2\text{-}nthD2$
by $blast$
thus ($sfunc\text{-}ind$ s i j) $<$ $length$ ($s\text{-}funcs$ \mathcal{S})
 $((s\text{-}funcs\ \mathcal{S})!(sfunc\text{-}ind\ s\ i\ j)) = (func\text{-}t$ ($(s\text{-}inst\ \mathcal{S})!i$))! j
unfolding $funci\text{-}agree\text{-}def$ $sfunc\text{-}ind\text{-}def$
by $auto$
thus $cl\text{-}typing$ \mathcal{S} ($sfunc$ s i j) ($(s\text{-}funcs\ \mathcal{S})!(sfunc\text{-}ind\ s\ i\ j)$)
using $funcs\text{-}agree$ $list\text{-}all2\text{-}nthD2$
unfolding $sfunc\text{-}def$
by $fastforce$
qed

lemma $store\text{-}typing\text{-}imp\text{-}glob\text{-}agree$:

assumes $store\text{-}typing$ s \mathcal{S}
 $i < length$ ($s\text{-}inst\ \mathcal{S}$)
 $j < length$ ($global$ ($(s\text{-}inst\ \mathcal{S})!i$))
shows ($sglob\text{-}ind$ s i j) $<$ $length$ ($s\text{-}globs$ \mathcal{S})
 $glob\text{-}agree$ ($sglob$ s i j) ($(s\text{-}globs\ \mathcal{S})!(sglob\text{-}ind\ s\ i\ j)$)
 $((s\text{-}globs\ \mathcal{S})!(sglob\text{-}ind\ s\ i\ j)) = (global$ ($(s\text{-}inst\ \mathcal{S})!i$))! j

proof –

have $globs\text{-}agree: list\text{-}all2$ $glob\text{-}agree$ ($globs$ s) ($s\text{-}globs$ \mathcal{S})
using $assms(1)$
unfolding $store\text{-}typing.simps$
by $auto$
have $list\text{-}all2$ ($globi\text{-}agree$ ($s\text{-}globs$ \mathcal{S})) ($inst.globs$ ($(inst\ s)!i$)) ($global$ ($(s\text{-}inst\ \mathcal{S})!i$))
using $assms(1,2)$ $store\text{-}typing\text{-}imp\text{-}inst\text{-}length\text{-}eq$ $store\text{-}typing\text{-}imp\text{-}inst\text{-}typing$
by ($fastforce$ $simp$ $add: inst\text{-}typing.simps$)
hence $globi\text{-}agree$ ($s\text{-}globs$ \mathcal{S}) ($(inst.globs$ ($(inst\ s)!i$))! j) ($(global$ ($(s\text{-}inst\ \mathcal{S})!i$))! j)
using $assms(3)$ $list\text{-}all2\text{-}nthD2$
by $blast$
thus ($sglob\text{-}ind$ s i j) $<$ $length$ ($s\text{-}globs$ \mathcal{S})
 $((s\text{-}globs\ \mathcal{S})!(sglob\text{-}ind\ s\ i\ j)) = (global$ ($(s\text{-}inst\ \mathcal{S})!i$))! j
unfolding $globi\text{-}agree\text{-}def$ $sglob\text{-}ind\text{-}def$
by $auto$
thus $glob\text{-}agree$ ($sglob$ s i j) ($(s\text{-}globs\ \mathcal{S})!(sglob\text{-}ind\ s\ i\ j)$)
using $globs\text{-}agree$ $list\text{-}all2\text{-}nthD2$
unfolding $sglob\text{-}def$
by $fastforce$
qed

lemma $store\text{-}typing\text{-}imp\text{-}mem\text{-}agree\text{-}Some$:

assumes $store\text{-}typing$ s \mathcal{S}
 $i < length$ ($s\text{-}inst\ \mathcal{S}$)

```

      smem-ind s i = Some j
shows j < length (s-mem S)
      mem-agree ((mem s)!j) ((s-mem S)!j)
       $\exists x. ((s-mem S)!j) = x \wedge (\text{memory } ((s-inst S)!i)) = \text{Some } x$ 
proof –
  have mems-agree:list-all2 mem-agree (mem s) (s-mem S)
  using assms(1)
  unfolding store-typing.simps
  by auto
  hence memi-agree (s-mem S) ((inst.mem ((inst s)!i))) ((memory ((s-inst S)!i)))
    using assms(1,2) store-typing-imp-inst-length-eq store-typing-imp-inst-typing
    by (fastforce simp add: inst-typing.simps)
  thus j < length (s-mem S)
     $\exists x. ((s-mem S)!j) = x \wedge (\text{memory } ((s-inst S)!i)) = \text{Some } x$ 
    using assms(3)
    unfolding memi-agree-def smem-ind-def
    by auto
  thus mem-agree ((mem s)!j) ((s-mem S)!j)
    using mems-agree list-all2-nthD2
    unfolding sglob-def
    by fastforce
qed

```

```

lemma store-typing-imp-mem-agree-None:
  assumes store-typing s S
    i < length (s-inst S)
    smem-ind s i = None
  shows (memory ((s-inst S)!i)) = None
proof –
  have mems-agree:list-all2 mem-agree (mem s) (s-mem S)
  using assms(1)
  unfolding store-typing.simps
  by auto
  hence memi-agree (s-mem S) ((inst.mem ((inst s)!i))) ((memory ((s-inst S)!i)))
    using assms(1,2) store-typing-imp-inst-length-eq store-typing-imp-inst-typing
    by (fastforce simp add: inst-typing.simps)
  thus ?thesis
    using assms(3)
    unfolding memi-agree-def smem-ind-def
    by auto
qed

```

```

lemma store-mem-exists:
  assumes i < length (s-inst S)
    store-typing s S
  shows Option.is-none (memory ((s-inst S)!i)) = Option.is-none (inst.mem ((inst s)!i))
proof –
  obtain j where j-def:j = (inst.mem ((inst s)!i))

```

by *blast*
obtain m **where** $m\text{-def}:m = (\text{memory } ((s\text{-inst } \mathcal{S})!i))$
 by *blast*
have $\text{inst-typ}:\text{inst-typing } \mathcal{S} ((\text{inst } s)!i) ((s\text{-inst } \mathcal{S})!i)$
using *assms*
unfolding *store-typing.simps list-all2-conv-all-nth*
by *auto*
thus *?thesis*
unfolding *inst-typing.simps memi-agree-def*
by *auto*
qed

lemma *store-preserved-mem:*

assumes *store-typing s S*
 $s' = s(\text{s.mem} := (\text{s.mem } s)[i := \text{mem}'])$
 $\text{mem-size } \text{mem}' \geq \text{mem-size } \text{orig-mem}$
 $((\text{s.mem } s)!i) = \text{orig-mem}$
shows *store-typing s' S*
proof –
obtain $\text{insts } fs \text{ clss } bss \text{ gs}$ **where** $s = (\text{inst} = \text{insts}, \text{funcs} = fs, \text{tab} = \text{clss}, \text{mem}$
 $= bss, \text{globs} = gs)$
using *s.cases*
by *blast*
moreover
obtain $\text{insts}' fs' \text{ clss}' bss' \text{ gs}'$ **where** $s' = (\text{inst} = \text{insts}', \text{funcs} = fs', \text{tab} = \text{clss}'$
 $\text{mem} = bss', \text{globs} = gs')$
using *s.cases*
by *blast*
moreover
obtain $Cs \text{ tfs } ns \text{ ms } tgs$ **where** $\mathcal{S} = (\text{s-inst} = Cs, \text{s-funcs} = tfs, \text{s-tab} = ns,$
 $\text{s-mem} = ms, \text{s-globs} = tgs)$
using *s-context.cases*
by *blast*
moreover
note $s\text{-S-defs} = \text{calculation}$
hence
 $\text{insts} = \text{insts}'$
 $fs = fs'$
 $\text{clss} = \text{clss}'$
 $gs = gs'$
using *assms(2)*
by *simp-all*
hence
 $\text{list-all2 } (\text{inst-typing } \mathcal{S}) \text{ insts}' Cs$
 $\text{list-all2 } (\text{cl-typing } \mathcal{S}) fs' tfs$
 $\text{list-all } (\text{tab-agree } \mathcal{S}) (\text{concat } \text{clss}')$
 $\text{list-all2 } (\lambda \text{cls } n. n \leq \text{length } \text{cls}) \text{ clss}' ns$
 $\text{list-all2 } \text{glob-agree } gs' tgs$
using *s-S-defs assms(1)*

```

    unfolding store-typing.simps
  by auto
moreover
have list-all2 ( $\lambda bs m. m \leq \text{mem-size } bs$ ) bss' ms
proof -
  have length bss = length bss'
    using assms(2) s-S-defs
    by (simp)
  moreover

  have initial-mem:list-all2 ( $\lambda bs m. m \leq \text{mem-size } bs$ ) bss ms
    using assms(1) s-S-defs
    unfolding store-typing.simps mem-agree-def
    by blast
  have  $\bigwedge n. n < \text{length } bss \implies (\lambda bs m. m \leq \text{mem-size } bs) (bss!n) (ms!n)$ 
  proof -
    fix n
    assume local-assms:n < length bss
    obtain C-m where cmdef:C-m = Cs ! n
      by blast
    hence ( $\lambda bs m. m \leq \text{mem-size } bs$ ) (bss!n) (ms!n)
      using initial-mem local-assms
      unfolding list-all2-conv-all-nth
      by simp
    thus ( $\lambda bs m. m \leq \text{mem-size } bs$ ) (bss!n) (ms!n)
      using assms(2,3,4) s-S-defs local-assms
      by (cases n=i, auto)
  qed
ultimately
show ?thesis
  by (metis initial-mem list-all2-all-nthI list-all2-lengthD)
qed
ultimately
show ?thesis
  unfolding store-typing.simps mem-agree-def
  by simp
qed

lemma types-agree-imp-e-typing:
  assumes types-agree t v
  shows  $\mathcal{S}\cdot\mathcal{C} \vdash [\$C v] : ([\ ] \rightarrow [t])$ 
  using assms e-typing-s-typing.intros(1)[OF b-e-typing.intros(1)]
  unfolding types-agree-def
  by fastforce

lemma list-types-agree-imp-e-typing:
  assumes list-all2 types-agree ts vs
  shows  $\mathcal{S}\cdot\mathcal{C} \vdash \$\$* vs : ([\ ] \rightarrow ts)$ 
  using assms

```

```

proof (induction rule: list-all2-induct)
  case Nil
  thus ?case
  using b-e-typing.empty e-typing-s-typing.intros(1)
  by fastforce
next
  case (Cons t ts v vs)
  hence  $\mathcal{S}\cdot\mathcal{C} \vdash [\mathcal{S} C v] : ([\ ] \rightarrow [t])$ 
  using types-agree-imp-e-typing
  by fastforce
  thus ?case
  using e-typing-s-typing.intros(3)[OF Cons(3), of [t]] e-type-comp-conc
  by fastforce
qed

lemma b-e-typing-imp-list-types-agree:
  assumes  $\mathcal{C} \vdash (\text{map } (\lambda v. C v) vs) : (ts' \rightarrow ts'@ts)$ 
  shows list-all2 types-agree ts vs
  using assms
proof (induction (map ( $\lambda v. C v$ ) vs) (ts'  $\rightarrow$  ts'@ts) arbitrary: ts ts' vs rule:
b-e-typing.induct)
  case (composition C es t1s t2s e)
  obtain vs1 vs2 where es-e-def:es = map EConst vs1 [e] = map EConst vs2
vs1@vs2=vs
  using composition(5)
  by (metis (no-types) last-map list.simps(8,9) map-butlast snoc-eq-iff-butlast)
  have const-list ($*es)
  using es-e-def(1) is-const-list1
  by auto
  then obtain tvs1 where t2s = t1s@tvs1
  using e-type-const-list e-typing-s-typing.intros(1)[OF composition(1)]
  by fastforce
  moreover
  have const-list ($*[e])
  using es-e-def(2) is-const-list1
  by auto
  then obtain tvs2 where t1s @ ts = t2s @ tvs2
  using e-type-const-list e-typing-s-typing.intros(1)[OF composition(3)]
  by fastforce
  ultimately
  show ?case
  using composition(2,4,5) es-e-def
  by (auto simp add: list-all2-appendI)
qed (auto simp add: types-agree-def)

lemma e-typing-imp-list-types-agree:
  assumes  $\mathcal{S}\cdot\mathcal{C} \vdash (\mathcal{S}\$* vs) : (ts' \rightarrow ts'@ts)$ 
  shows list-all2 types-agree ts vs
proof –

```

```

have ( $\$ \$ * vs$ ) =  $\$ * (map (\lambda v. C v) vs)$ 
  by simp
thus ?thesis
  using assms unlift-b-e b-e-typing-imp-list-types-agree
  by (fastforce simp del: map-map)
qed

lemma store-extension-imp-store-typing:
  assumes store-extension s s'
    store-typing s S
  shows store-typing s' S
proof –
  obtain insts fs cls bss gs where  $s = (inst = insts, funcs = fs, tab = cls, mem$ 
=  $bss, globs = gs)$ 
  using s.cases
  by blast
  moreover
  obtain insts' fs' cls' bss' gs' where  $s' = (inst = insts', funcs = fs', tab = cls',$ 
mem = bss', globs = gs')
  using s.cases
  by blast
  moreover
  obtain Cs tfs ns ms tgs where  $S = (s-inst = Cs, s-funcs = tfs, s-tab = ns,$ 
s-mem = ms, s-globs = tgs)
  using s-context.cases
  by blast
  moreover
  note  $s-S-defs = calculation$ 
  hence
insts = insts'
fs = fs'
cls = cls'
gs = gs'
  using assms(1)
  unfolding store-extension.simps
  by simp-all
  hence
list-all2 (inst-typing S) insts' Cs
list-all2 (cl-typing S) fs' tfs
list-all (tab-agree S) (concat cls')
list-all2 ( $\lambda cls n. n \leq length\ cls$ ) cls' ns
list-all2 glob-agree gs' tgs
  using s-S-defs assms(2)
  unfolding store-typing.simps
  by auto
  moreover
  have list-all2 ( $\lambda bs m. m \leq mem-size\ bs$ ) bss ms
  using s-S-defs(1,3) assms(2)
  unfolding store-typing.simps mem-agree-def

```

```

    by simp
  hence list-all2 mem-agree bss' ms
    using assms(1) s-S-defs(1,2)
    unfolding store-extension.simps list-all2-conv-all-nth mem-agree-def
    by fastforce
  ultimately
  show ?thesis
    using store-typing.intros
    by fastforce
qed

```

```

lemma lfilled-deterministic:
  assumes Lfilled k lfilled es les
         Lfilled k lfilled es les'
  shows les = les'
  using assms
proof (induction arbitrary: les' rule: Lfilled.induct)
  case (L0 vs lholed es' es)
  thus ?case
    by (fastforce simp add: Lfilled.simps[of 0])
next
  case (LN vs lholed n es' l es'' k es lfilledk)
  thus ?case
    unfolding Lfilled.simps[of (k + 1)]
    by fastforce
qed
end

```

6 Lemmas for Soundness Proof

```

theory Wasm-Properties imports Wasm-Properties-Aux begin

```

6.1 Preservation

```

lemma t-cvt: assumes cvt t sx v = Some v' shows t = typeof v'
  using assms
  unfolding cvt-def typeof-def
  apply (cases t)
  apply (simp add: option.case-eq-if, metis option.discI option.inject v.simps(17))
  apply (simp add: option.case-eq-if, metis option.discI option.inject v.simps(18))
  apply (simp add: option.case-eq-if, metis option.discI option.inject v.simps(19))
  apply (simp add: option.case-eq-if, metis option.discI option.inject v.simps(20))
  done

```

```

lemma store-preserved1:
  assumes (|s;vs;es)  $\rightsquigarrow$ -i (|s';vs';es')
         store-typing s  $\mathcal{S}$ 
          $\mathcal{S}\cdot\mathcal{C} \vdash es : (ts \rightarrow ts')$ 
          $\mathcal{C} = ((s\text{-inst } \mathcal{S})!i)(\text{local} := \text{local } ((s\text{-inst } \mathcal{S})!i) \text{ @ } (\text{map } \text{typeof } vs), \text{label} :=$ 

```

```

arb-label, return := arb-return)
  i < length (s-inst S)
  shows store-typing s' S
  using assms
proof (induction arbitrary: C arb-label arb-return ts ts' rule: reduce.induct)
  case (callcl-host-Some cl t1s t2s f ves vcs n m s hs s' vcs' vs i)
  obtain ts'' where ts''-def: S.C ⊢ ves : (ts -> ts'') S.C ⊢ [Callcl cl] : (ts'' -> ts')
  using callcl-host-Some(8) e-type-comp
  by fastforce
  have ves-c:const-list ves
    using is-const-list[OF callcl-host-Some(2)]
    by simp
  then obtain tvs where tvs-def: ts'' = ts @ tvs
    length t1s = length tvs
    S.C ⊢ ves : ([] -> tvs)
    using ts''-def(1) e-type-const-list[of ves S C ts ts''] callcl-host-Some
    by fastforce
  hence ts'' = ts @ t1s
    ts' = ts @ t2s
    using e-type-callcl-host[OF ts''-def(2) callcl-host-Some(1)]
    by auto
  moreover
  hence list-all2 types-agree t1s vcs
    using e-typing-imp-list-types-agree[where ?ts' = []] callcl-host-Some(2) tvs-def(1,3)
    by fastforce
  thus ?case
    using store-extension-imp-store-typing
    host-apply-preserve-store[OF - callcl-host-Some(6)] callcl-host-Some(7)
    by fastforce
next
  case (set-global s i j v s' vs)
  obtain insts fs class bss gs where s = (inst = insts, funcs = fs, tab = class, mem
= bss, globs = gs)
    using s.cases
    by blast
  moreover
  obtain insts' fs' class' bss' gs' where s' = (inst = insts', funcs = fs', tab = class',
mem = bss', globs = gs')
    using s.cases
    by blast
  moreover
  obtain Cs tfs ns ms tgs where S = (s-inst = Cs, s-funcs = tfs, s-tab = ns,
s-mem = ms, s-globs = tgs)
    using s-context.cases
    by blast
  moreover

note s-S-defs = calculation

```

```

have
  insts = insts'
  fs = fs'
  clss = clss'
  bss = bss'
  using set-global(1) s-S-defs(1,2)
  unfolding supdate-glob-def supdate-glob-s-def
  by (metis s.ext-inject s.update-convs(5))+
hence
  list-all2 (inst-typing  $\mathcal{S}$ ) insts' Cs
  list-all2 (cl-typing  $\mathcal{S}$ ) fs' tfs
  list-all (tab-agree  $\mathcal{S}$ ) (concat clss')
  list-all2 ( $\lambda cl\ n. n \leq \text{length } cl$ ) clss' ns
  list-all2 mem-agree bss' ms
  using set-global(2) s-S-defs
  unfolding store-typing.simps
  by auto
moreover
have list-all2 glob-agree gs' tgs
proof –
  have gs-agree:list-all2 glob-agree gs tgs
  using set-global(2) s-S-defs
  unfolding store-typing.simps
  by auto
have length gs = length gs'
  using s-S-defs(1,2) set-global(1)
  unfolding supdate-glob-def supdate-glob-s-def
  by (metis length-list-update s.select-convs(5) s.update-convs(5))
moreover
obtain k where k-def:(sglob-ind s i j) = k
  by blast
hence  $\bigwedge j'. \llbracket j' \neq k; j' < \text{length } gs \rrbracket \implies gs!j' = gs!j'$ 
  using s-S-defs(1,2) set-global(1)
  unfolding supdate-glob-def supdate-glob-s-def
  by auto
hence  $\bigwedge j'. \llbracket j' \neq k; j' < \text{length } gs \rrbracket \implies \text{glob-agree } (gs!j') (tgs!j')$ 
  using gs-agree
  by (metis list-all2-conv-all-nth)
moreover
have glob-agree (gs!k) (tgs!k)
proof –
  obtain ts'' where ts''-def:C  $\vdash [C\ v] : (ts \rightarrow ts'')\ C \vdash [\text{Set-global } j] : (ts'' \rightarrow$ 
ts')
  by (metis b-e-type-comp2-unlift set-global.prem(2))
have b-es:ts'' = ts@[typeof v]
  ts = ts'
  global C ! j = (!tg-mut = T-mut, tg-t = typeof v)
  j < length (global C)

```

```

    using b-e-type-value[OF ts''-def(1)] b-e-type-set-global[OF ts''-def(2)]
  by auto
hence j < length (global ((s-inst S)!i))
  using set-global(4)
  by fastforce
hence globs-agree:k < length (s-globs S)
      glob-agree (gs!k) (tgs!k)
      (tgs!k) = (global C)!j
  using store-typing-imp-glob-agree[OF set-global(2,5)] b-es(4) s-S-defs(1,3)
k-def set-global(4)
  unfolding sglob-def
  by auto
hence g-mut (gs!k) = T-mut
      typeof (g-val (gs!k)) = typeof v
  using b-es(3)
  unfolding glob-agree-def
  by auto
hence g-mut (gs^!k) = T-mut
      typeof (g-val (gs^!k)) = typeof v
  using set-global(1) k-def globs-agree(1) store-typing-imp-glob-length-eq[OF
set-global(2)] s-S-defs(1,2)
  unfolding supdate-glob-def supdate-glob-s-def
  by auto
thus ?thesis
  using globs-agree(3) b-es(3)
  unfolding glob-agree-def
  by fastforce
qed
ultimately
show ?thesis
  using gs-agree
  unfolding list-all2-conv-all-nth
  by fastforce
qed
ultimately
show ?case
  using store-typing.intros
  by simp
next

case (store-Some t v s i j m k off mem' vs a)
show ?case
  using store-preserved-mem[OF store-Some(5) - - store-Some(3)] store-size[OF
store-Some(4)]
  by fastforce
next
case (store-packed-Some t v s i j m k off tp mem' vs a)
thus ?case
  using store-preserved-mem[OF store-packed-Some(5) - - store-packed-Some(3)]

```

```

store-packed-size[OF store-packed-Some(4)]
  by simp
next
case (grow-memory s i j n mem c mem' vs)
show ?case
  using store-preserved-mem[OF grow-memory(5) - grow-memory(2)] mem-grow-size[OF
grow-memory(4)]
  by simp
next
case (label s vs es i s' vs' es' k lholed les les')
obtain C' t1s t2s arb-label' arb-return' where es-def:C' = C(label := arb-label',
return := arb-return') S.C' ⊢ es : (t1s -> t2s)
  using types-exist-lfilled-weak[OF label(2,6)]
  by fastforce
thus ?case
  using label(4)[OF label(5) es-def(2) - label(8)] label(7)
  by fastforce
next
case (local s vs es i s' vs' es' v0s n j)
obtain t1s where t-local:(S((s-inst S)!i)(local := (local ((s-inst S)!i)) @ (map
typeof vs), return := Some t1s) ⊢ es : ([] -> t1s))
  ts' = ts @ t1s i < length (s-inst S)
  using e-type-local[OF local(4)]
  by blast+
show ?case
  using local(2)[OF local(3) t-local(1) - t-local(3), of (Some t1s) label ((s-inst
S)!i) ]
  by fastforce
qed (simp-all)

lemma store-preserved:
  assumes (|s;vs;es) ~-i (|s';vs';es')
    store-typing s S
    S.None ⊢-i vs;es : ts
  shows store-typing s' S
proof -
  show ?thesis
    using store-preserved1[OF assms(1,2), of - [] ts None label (s-inst S!i)]
      s-type-unfold[OF assms(3)]
    by fastforce
qed

lemma typeof-unop-testop:
  assumes S.C ⊢ [C v, $e] : (ts -> ts')
    (e = (Unop-i t iop)) ∨ (e = (Unop-f t fop)) ∨ (e = (Testop t testop))
  shows (typeof v) = t
    e = (Unop-f t fop) ⇒ is-float-t t
proof -
  have C ⊢ [C v, e] : (ts -> ts')

```

using *unlift-b-e* *assms*(1)
by *simp*
then obtain ts'' **where** $ts''\text{-def}:\mathcal{C} \vdash [C\ v] : (ts \rightarrow ts'') \mathcal{C} \vdash [e] : (ts'' \rightarrow ts')$
using *b-e-type-comp*[**where** $?e = e$ **and** $?es = [C\ v]$]
by *fastforce*
show $(\text{typeof } v) = t\ e = (\text{Unop-f } t\ \text{fop}) \implies \text{is-float-t } t$
using *b-e-type-value*[*OF* $ts''\text{-def}(1)$] *assms*(2) *b-e-type-unop-testop*[*OF* $ts''\text{-def}(2)$]
by *simp-all*
qed

lemma *typeof-cvtop*:
assumes $\mathcal{S}\cdot\mathcal{C} \vdash [\$C\ v, \$e] : (ts \rightarrow ts')$
 $e = \text{Cvtop } t1\ \text{cvtop } t\ \text{sx}$
shows $(\text{typeof } v) = t$
 $\text{cvtop} = \text{Convert} \implies (t1 \neq t) \wedge ((\text{sx} = \text{None}) = ((\text{is-float-t } t1 \wedge \text{is-float-t } t) \vee (\text{is-int-t } t1 \wedge \text{is-int-t } t \wedge (t\text{-length } t1 < t\text{-length } t))))$
 $\text{cvtop} = \text{Reinterpret} \implies (t1 \neq t) \wedge t\text{-length } t1 = t\text{-length } t$

proof –
have $\mathcal{C} \vdash [C\ v, e] : (ts \rightarrow ts')$
using *unlift-b-e* *assms*(1)
by *simp*
then obtain ts'' **where** $ts''\text{-def}:\mathcal{C} \vdash [C\ v] : (ts \rightarrow ts'') \mathcal{C} \vdash [e] : (ts'' \rightarrow ts')$
using *b-e-type-comp*[**where** $?e = e$ **and** $?es = [C\ v]$]
by *fastforce*
show $(\text{typeof } v) = t$
 $\text{cvtop} = \text{Convert} \implies (t1 \neq t) \wedge (\text{sx} = \text{None}) = ((\text{is-float-t } t1 \wedge \text{is-float-t } t) \vee (\text{is-int-t } t1 \wedge \text{is-int-t } t \wedge (t\text{-length } t1 < t\text{-length } t)))$
 $\text{cvtop} = \text{Reinterpret} \implies (t1 \neq t) \wedge t\text{-length } t1 = t\text{-length } t$
using *b-e-type-value*[*OF* $ts''\text{-def}(1)$] *b-e-type-cvtop*[*OF* $ts''\text{-def}(2)$] *assms*(2)]
by *simp-all*
qed

lemma *types-preserved-unop-testop-cvtop*:
assumes $(\llbracket \$C\ v, \$e \rrbracket \rightsquigarrow \llbracket \$C\ v' \rrbracket)$
 $\mathcal{S}\cdot\mathcal{C} \vdash [\$C\ v, \$e] : (ts \rightarrow ts')$
 $(e = (\text{Unop-i } t\ \text{iop})) \vee (e = (\text{Unop-f } t\ \text{fop})) \vee (e = (\text{Testop } t\ \text{testop})) \vee$
 $(e = (\text{Cvtop } t2\ \text{cvtop } t\ \text{sx}))$
shows $\mathcal{S}\cdot\mathcal{C} \vdash [\$C\ v'] : (ts \rightarrow ts')$
proof –
have $\mathcal{C} \vdash [C\ v, e] : (ts \rightarrow ts')$
using *unlift-b-e* *assms*(2)
by *simp*
then obtain ts'' **where** $ts''\text{-def}:\mathcal{C} \vdash [C\ v] : (ts \rightarrow ts'') \mathcal{C} \vdash [e] : (ts'' \rightarrow ts')$
using *b-e-type-comp*[**where** $?e = e$ **and** $?es = [C\ v]$]
by *fastforce*
have $ts@[arity-1\text{-result } e] = ts' (\text{typeof } v) = t$
using *b-e-type-value*[*OF* $ts''\text{-def}(1)$] *assms*(3) *b-e-type-unop-testop*(1)[*OF* $ts''\text{-def}(2)$]
 $b\text{-e-type-cvtop}$ (1)[*OF* $ts''\text{-def}(2)$]
by (*metis butlast-snoc*, *metis last-snoc*)

moreover
have *arity-1-result* $e = \text{typeof } (v')$
using *assms*(1,3)
apply (*cases rule: reduce-simple.cases*)
apply (*simp-all add: arity-1-result-def wasm-deserialise-type t-cvt*)
apply (*auto simp add: typeof-def*)
done
hence $\mathcal{C} \vdash [C \ v'] : ([\] \rightarrow [\text{arity-1-result } e])$
using *b-e-typing.const*
by *metis*
ultimately
show $\mathcal{S}\cdot\mathcal{C} \vdash [\$C \ v'] : (ts \rightarrow ts')$
using *e-typing-s-typing.intros*(1)
b-e-typing.weakening[of $\mathcal{C} [C \ v'] [\] [\text{arity-1-result } e] \ ts$]
by *fastforce*
qed

lemma *typeof-binop-relop*:
assumes $\mathcal{S}\cdot\mathcal{C} \vdash [\$C \ v1, \$C \ v2, \$e] : (ts \rightarrow ts')$
 $e = \text{Binop-i } t \ \text{iop} \vee e = \text{Binop-f } t \ \text{fop} \vee e = \text{Relop-i } t \ \text{irop} \vee e = \text{Relop-f}$
 $t \ \text{frop}$
shows $\text{typeof } v1 = t$
 $\text{typeof } v2 = t$
 $e = \text{Binop-f } t \ \text{fop} \implies \text{is-float-t } t$
 $e = \text{Relop-f } t \ \text{frop} \implies \text{is-float-t } t$
proof –
have $\mathcal{C} \vdash [C \ v1, C \ v2, e] : (ts \rightarrow ts')$
using *unlift-b-e assms*(1)
by *simp*
then obtain ts'' **where** $ts''\text{-def}:\mathcal{C} \vdash [C \ v1, C \ v2] : (ts \rightarrow ts'')$ $\mathcal{C} \vdash [e] : (ts'' \rightarrow$
 $ts')$
using *b-e-type-comp*[**where** $?e = e$ **and** $?es = [C \ v1, C \ v2]$]
by *fastforce*
then obtain $ts\text{-id}$ **where** $ts\text{-id-def}:\text{ts-id}@[t,t] = ts'' \ ts' = ts\text{-id} \ @ \ [\text{arity-2-result}$
 $e]$
 $e = \text{Binop-f } t \ \text{fop} \implies \text{is-float-t } t$
 $e = \text{Relop-f } t \ \text{frop} \implies \text{is-float-t } t$
using *assms*(2) *b-e-type-binop-relop*[of $\mathcal{C} \ e \ ts'' \ ts' \ t]$
by *blast*
thus $\text{typeof } v1 = t$
 $\text{typeof } v2 = t$
 $e = \text{Binop-f } t \ \text{fop} \implies \text{is-float-t } t$
 $e = \text{Relop-f } t \ \text{frop} \implies \text{is-float-t } t$
using $ts''\text{-def}$ *b-e-type-comp*[of $\mathcal{C} [C \ v1] \ C \ v2 \ ts \ ts'']$ *b-e-type-value2*
by *fastforce+*
qed

lemma *types-preserved-binop-relop*:
assumes $([\$C \ v1, \$C \ v2, \$e]) \rightsquigarrow ([\$C \ v'])$

$\mathcal{S}\cdot\mathcal{C} \vdash [\$C v1, \$C v2, \$e] : (ts \rightarrow ts')$
 $e = \text{Binop-}i\ t\ iop \vee e = \text{Binop-}f\ t\ fop \vee e = \text{Relop-}i\ t\ irop \vee e = \text{Relop-}f\ t\ frop$
shows $\mathcal{S}\cdot\mathcal{C} \vdash [\$C v'] : (ts \rightarrow ts')$
proof –
have $\mathcal{C} \vdash [C v1, C v2, e] : (ts \rightarrow ts')$
using *unlift-b-e assms(2)*
by *simp*
then obtain ts'' **where** $ts''\text{-def}:\mathcal{C} \vdash [C v1, C v2] : (ts \rightarrow ts'')$ $\mathcal{C} \vdash [e] : (ts'' \rightarrow ts')$
using *b-e-type-comp[where ?e = e and ?es = [C v1, C v2]]*
by *fastforce*
then obtain $ts\text{-id}$ **where** $ts\text{-id-def}:ts\text{-id}@[t,t] = ts''\ ts' = ts\text{-id} @ [arity\text{-}2\text{-result}\ e]$
using *assms(3) b-e-type-binop-relop[of C e ts'' ts' t]*
by *blast*
hence $\mathcal{C} \vdash [C v1] : (ts \rightarrow ts\text{-id}@[t])$
using $ts''\text{-def}$ *b-e-type-comp[of C [C v1] C v2 ts ts'']* *b-e-type-value*
by *fastforce*
hence $ts@[arity\text{-}2\text{-result}\ e] = ts'$
using *b-e-type-value ts-id-def(2)*
by *fastforce*
moreover
have $arity\text{-}2\text{-result}\ e = \text{typeof}\ (v')$
using *assms(1,3)*
by *(cases rule: reduce-simple.cases) (auto simp add: arity-2-result-def typeof-def)*
hence $\mathcal{C} \vdash [C v'] : ([] \rightarrow [arity\text{-}2\text{-result}\ e])$
using *b-e-typing.const*
by *metis*
ultimately show $?thesis$
using *e-typing-s-typing.intros(1)*
b-e-typing.weakening[of C [C v'] [] [arity-2-result e] ts]
by *fastforce*
qed

lemma *types-preserved-drop:*

assumes $(\llbracket \$C v, \$e \rrbracket) \rightsquigarrow (\llbracket [] \rrbracket)$
 $\mathcal{S}\cdot\mathcal{C} \vdash [\$C v, \$e] : (ts \rightarrow ts')$
 $(e = (\text{Drop}))$
shows $\mathcal{S}\cdot\mathcal{C} \vdash [] : (ts \rightarrow ts')$

proof –

have $\mathcal{C} \vdash [C v, e] : (ts \rightarrow ts')$
using *unlift-b-e assms(2)*
by *simp*
then obtain ts'' **where** $ts''\text{-def}:\mathcal{C} \vdash [C v] : (ts \rightarrow ts'')$ $\mathcal{C} \vdash [e] : (ts'' \rightarrow ts')$
using *b-e-type-comp[where ?e = e and ?es = [C v]]*
by *fastforce*
hence $ts'' = ts@[typeof\ v]$
using *b-e-type-value*

by *blast*
 hence $ts = ts'$
 using *ts''-def* *assms*(3) *b-e-type-drop*
 by *blast*
 hence $\mathcal{C} \vdash [] : (ts \rightarrow ts')$
 using *b-e-type-empty*
 by *simp*
 thus *?thesis*
 using *e-typing-s-typing.intros*(1)
 by *fastforce*
 qed

lemma *types-preserved-select*:

assumes $([\$C\ v1, \$C\ v2, \$C\ vn, \$e]) \rightsquigarrow ([\$C\ v3])$
 $\mathcal{S}\cdot\mathcal{C} \vdash [\$C\ v1, \$C\ v2, \$C\ vn, \$e] : (ts \rightarrow ts')$
 ($e = \text{Select}$)
 shows $\mathcal{S}\cdot\mathcal{C} \vdash [\$C\ v3] : (ts \rightarrow ts')$
proof –
 have $\mathcal{C} \vdash [C\ v1, C\ v2, C\ vn, e] : (ts \rightarrow ts')$
 using *unlift-b-e* *assms*(2)
 by *simp*
 then obtain *t1s* where *t1s-def*: $\mathcal{C} \vdash [C\ v1, C\ v2, C\ vn] : (ts \rightarrow t1s)$ $\mathcal{C} \vdash [e] :$
 ($t1s \rightarrow ts'$)
 using *b-e-type-comp*[**where** $?e = e$ **and** $?es = [C\ v1, C\ v2, C\ vn]$]
 by *fastforce*
 then obtain *t2s* *t* where *t2s-def*: $t1s = t2s @ [t, t, (T-i32)]$ $ts' = t2s@[t]$
 using *b-e-type-select*[*of* \mathcal{C} *e* *t1s*] *assms*
 by *fastforce*
 hence $\mathcal{C} \vdash [C\ v1, C\ v2] : (ts \rightarrow t2s@[t, t])$
 using *t1s-def* *t2s-def* *b-e-type-value-list*[*of* \mathcal{C} $[C\ v1, C\ v2]$ *vn* *ts* $t2s@[t, t]$]
 by *fastforce*
 hence *v2-t-def*: $\mathcal{C} \vdash [C\ v1] : (ts \rightarrow t2s@[t])$ *typeof* *v2* = *t*
 using *t1s-def* *t2s-def* *b-e-type-value-list*[*of* \mathcal{C} $[C\ v1]$ *v2* *ts* $t2s@[t]$]
 by *fastforce*+
 hence *v1-t-def*: $ts = t2s$ *typeof* *v1* = *t*
 using *b-e-type-value*
 by *fastforce*+
 have *typeof* *v3* = *t*
 using *assms*(1) *v2-t-def*(2) *v1-t-def*(2)
 by (*cases* *rule*: *reduce-simple.cases*, *simp-all*)
 hence $\mathcal{C} \vdash [C\ v3] : (ts \rightarrow ts')$
 using *b-e-typing.const* *b-e-typing.weakening* *t2s-def*(2) *v1-t-def*(1)
 by *fastforce*
 thus *?thesis*
 using *e-typing-s-typing.intros*(1)
 by *fastforce*
 qed

lemma *types-preserved-block*:

assumes $(vs @ [\$Block (tn \rightarrow tm) es]) \rightsquigarrow ([Label\ m\ \square] (vs @ (\$* es)))$
 $\mathcal{S}\cdot\mathcal{C} \vdash vs @ [\$Block (tn \rightarrow tm) es] : (ts \rightarrow ts')$
const-list vs
length vs = n
length tn = n
length tm = m
shows $\mathcal{S}\cdot\mathcal{C} \vdash [Label\ m\ \square] (vs @ (\$* es)) : (ts \rightarrow ts')$
proof –
obtain \mathcal{C}' **where** $c\text{-def}:\mathcal{C}' = \mathcal{C}(\text{label} := [tm] @ \text{label}\ \mathcal{C})$ **by** *blast*
obtain ts'' **where** $ts''\text{-def}:\mathcal{S}\cdot\mathcal{C} \vdash vs : (ts \rightarrow ts'')$ $\mathcal{S}\cdot\mathcal{C} \vdash [\$Block (tn \rightarrow tm) es] :$
 $(ts'' \rightarrow ts')$
using *assms(2)* *e-type-comp[of $\mathcal{S}\ \mathcal{C}\ vs\ \$Block (tn \rightarrow tm) es\ ts\ ts'$]*
by *fastforce*
hence $\mathcal{C} \vdash [Block (tn \rightarrow tm) es] : (ts'' \rightarrow ts')$
using *unlift-b-e*
by *auto*
then obtain $ts\text{-c}\ tfn\ tfm$ **where** $ts\text{-c}\text{-def}:(tn \rightarrow tm) = (tfn \rightarrow tfm)$ $ts'' = ts\text{-c}@tfn$
 $ts' = ts\text{-c}@tfn$ $(\mathcal{C}(\text{label} := [tfm] @ \text{label}\ \mathcal{C}) \vdash es : (tn \rightarrow tm))$
using *b-e-type-block[of $\mathcal{C}\ Block (tn \rightarrow tm) es\ ts''\ ts' (tn \rightarrow tm) es$]*
by *fastforce*
hence $tfn\text{-l}:\text{length}\ tfn = n$
using *assms(5)*
by *simp*
obtain tvs' **where** $tvs'\text{-def}:ts'' = ts@tvs'$ $\text{length}\ tvs' = n$ $\mathcal{S}\cdot\mathcal{C}' \vdash vs : (\square \rightarrow tvs')$
using *e-type-const-list assms(3,4)* $ts''\text{-def}(1)$
by *fastforce*
hence $\mathcal{S}\cdot\mathcal{C}' \vdash vs : (\square \rightarrow tn)$ $\mathcal{S}\cdot\mathcal{C}' \vdash \$*es : (tn \rightarrow tm)$
using *ts-c-def tvs'-def tfn-l ts''-def c-def e-typing-s-typing.intros(1)*
by *simp-all*
hence $\mathcal{S}\cdot\mathcal{C}' \vdash (vs @ (\$* es)) : (\square \rightarrow tm)$ **using** *e-type-comp-conc*
by *simp*
moreover
have $\mathcal{S}\cdot\mathcal{C} \vdash \square : (tm \rightarrow tm)$
using *b-e-type-empty[of $\mathcal{C}\ \square\ \square$]*
e-typing-s-typing.intros(1)[where ?b-es = \square]
e-typing-s-typing.intros(3)[of $\mathcal{S}\ \mathcal{C}\ \square\ \square\ tm$]
by *fastforce*
ultimately
show *?thesis*
using *e-typing-s-typing.intros(7)[of $\mathcal{S}\ \mathcal{C}\ \square\ tm - vs @ (\$* es) m$]*
ts-c-def tvs'-def assms(5,6) *e-typing-s-typing.intros(3)* *c-def*
by *fastforce*
qed

lemma *types-preserved-if:*

assumes $([\$C\ ConstInt32\ n,\ \$If\ tf\ e1s\ e2s]) \rightsquigarrow ([\$Block\ tf\ es'])$
 $\mathcal{S}\cdot\mathcal{C} \vdash [\$C\ ConstInt32\ n,\ \$If\ tf\ e1s\ e2s] : (ts \rightarrow ts')$
shows $\mathcal{S}\cdot\mathcal{C} \vdash [\$Block\ tf\ es'] : (ts \rightarrow ts')$
proof –

```

have  $C \vdash [C \text{ ConstInt32 } n, \text{If } tf \ e1s \ e2s] : (ts \rightarrow ts')$ 
  using unlift-b-e assms(2)
  by fastforce
then obtain  $ts\text{-}i$  where  $ts\text{-}i\text{-}def : C \vdash [C \text{ ConstInt32 } n] : (ts \rightarrow ts\text{-}i)$   $C \vdash [\text{If } tf \ e1s \ e2s] : (ts\text{-}i \rightarrow ts')$ 
  using b-e-type-comp
  by (metis append-Cons append-Nil)
then obtain  $ts''$   $tfn$   $tfm$  where  $ts\text{-}def : tf = (tfn \rightarrow tfm)$ 
   $ts\text{-}i = ts'' @ tfn @ [T\text{-}i32]$ 
   $ts' = ts'' @ tfm$ 
   $(C \langle label := [tfm] @ label \ C \rangle \vdash e1s : tf)$ 
   $(C \langle label := [tfm] @ label \ C \rangle \vdash e2s : tf)$ 
  using b-e-type-if[of C If tf e1s e2s]
  by fastforce
have  $ts\text{-}i = ts @ [(T\text{-}i32)]$ 
  using ts-i-def(1) b-e-type-value
  unfolding typeof-def
  by fastforce
moreover
have  $(C \langle label := [tfm] @ label \ C \rangle \vdash es' : (tfn \rightarrow tfm))$ 
  using assms(1) ts-def(4,5) ts-def(1)
  by (cases rule: reduce-simple.cases, simp-all)
hence  $C \vdash [\text{Block } tf \ es'] : (tfn \rightarrow tfm)$ 
  using ts-def(1) b-e-typing.block[of tf tfn tfm C es']
  by simp
ultimately
show ?thesis
  using ts-def(2,3) e-typing-s-typing.intros(1,3)
  by fastforce
qed

```

lemma *types-preserved-tee-local:*

```

assumes  $(\langle [v, \$Tee\text{-}local \ i] \rangle \rightsquigarrow \langle [v, v, \$Set\text{-}local \ i] \rangle)$ 
   $\mathcal{S} \cdot C \vdash [v, \$Tee\text{-}local \ i] : (ts \rightarrow ts')$ 
  is-const v
shows  $\mathcal{S} \cdot C \vdash [v, v, \$Set\text{-}local \ i] : (ts \rightarrow ts')$ 
proof –
obtain  $bv$  where  $bv\text{-}def : v = \$C \ bv$ 
  using e-type-const-unwrap assms(3)
  by fastforce
hence  $C \vdash [C \ bv, Tee\text{-}local \ i] : (ts \rightarrow ts')$ 
  using unlift-b-e assms(2)
  by fastforce
then obtain  $ts''$  where  $ts''\text{-}def : C \vdash [C \ bv] : (ts \rightarrow ts'')$   $C \vdash [Tee\text{-}local \ i] : (ts'' \rightarrow ts')$ 
  using b-e-type-comp[of - [C bv] Tee-local i]
  by fastforce
then obtain  $ts\text{-}c \ t$  where  $ts\text{-}c\text{-}def : ts'' = ts\text{-}c @ [t]$   $ts' = ts\text{-}c @ [t]$  (local C)!  $i = t \ i < \text{length}(\text{local } C)$ 

```

using $b\text{-}e\text{-type}\text{-tee}\text{-local}[of\ C\ Tee\text{-local}\ i\ ts''\ ts'\ i]$
by $fastforce$
hence $t\text{-}bv:t = typeof\ bv\ ts = ts\text{-}c$
using $b\text{-}e\text{-type}\text{-value}\ ts''\text{-}def$
by $fastforce+$
have $C \vdash [Set\text{-local}\ i] : ([t,t] \rightarrow [t])$
using $ts\text{-}c\text{-}def(3,4)\ b\text{-}e\text{-typing.set}\text{-local}[of\ i\ C\ t]$
 $b\text{-}e\text{-typing.weakening}[of\ C\ [Set\text{-local}\ i]\ [t]\ []\ [t]]$
by $fastforce$
moreover
have $C \vdash [C\ bv] : ([t] \rightarrow [t,t])$
using $t\text{-}bv\ b\text{-}e\text{-typing.const}[of\ C\ bv]\ b\text{-}e\text{-typing.weakening}[of\ C\ [C\ bv]\ []\ [t]\ [t]]$
by $fastforce$
hence $C \vdash [C\ bv,\ C\ bv] : ([] \rightarrow [t,t])$
using $t\text{-}bv\ b\text{-}e\text{-typing.const}[of\ C\ bv]\ b\text{-}e\text{-typing.composition}[of\ C\ [C\ bv]\ []\ [t]]$
by $fastforce$
ultimately
have $C \vdash [C\ bv,\ C\ bv,\ Set\text{-local}\ i] : (ts \rightarrow ts@[t])$
using $b\text{-}e\text{-typing.composition}\ b\text{-}e\text{-typing.weakening}[of\ C\ [C\ bv,\ C\ bv,\ Set\text{-local}\ i]]$
by $fastforce$
thus $?thesis$
using $t\text{-}bv(2)\ ts\text{-}c\text{-}def(2)\ bv\text{-}def\ e\text{-typing}\text{-}s\text{-}typing.intros(1)$
by $fastforce$
qed

lemma $types\text{-}preserved\text{-}loop$:

assumes $(vs\ @\ [\$Loop\ (t1s\ \rightarrow\ t2s)\ es]) \rightsquigarrow ([Label\ n\ [\$Loop\ (t1s\ \rightarrow\ t2s)\ es]\ (vs\ @\ (\$*\ es)))]$
 $\mathcal{S}\cdot\mathcal{C} \vdash vs\ @\ [\$Loop\ (t1s\ \rightarrow\ t2s)\ es] : (ts \rightarrow ts')$
 $const\text{-}list\ vs$
 $length\ vs = n$
 $length\ t1s = n$
 $length\ t2s = m$
shows $\mathcal{S}\cdot\mathcal{C} \vdash [Label\ n\ [\$Loop\ (t1s\ \rightarrow\ t2s)\ es]\ (vs\ @\ (\$*\ es))] : (ts \rightarrow ts')$
proof –
obtain ts'' **where** $ts''\text{-}def:\mathcal{S}\cdot\mathcal{C} \vdash vs : (ts \rightarrow ts'')$ $\mathcal{S}\cdot\mathcal{C} \vdash [\$Loop\ (t1s\ \rightarrow\ t2s)\ es] : (ts'' \rightarrow ts')$
using $assms(2)\ e\text{-type}\text{-}comp$
by $fastforce$
then **have** $C \vdash [Loop\ (t1s\ \rightarrow\ t2s)\ es] : (ts'' \rightarrow ts')$
using $unlift\text{-}b\text{-}e\ assms(2)$
by $fastforce$
then **obtain** $ts\text{-}c\ tfn\ tfm\ C'$ **where** $t\text{-}loop:(t1s \rightarrow t2s) = (tfn \rightarrow tfm)$
 $(ts'' = ts\text{-}c@tfn)$
 $(ts' = ts\text{-}c@tfm)$
 $C' = C(\text{label} := [t1s] @ \text{label}\ C)$
 $(C' \vdash es : (tfn \rightarrow tfm))$
using $b\text{-}e\text{-type}\text{-}loop[of\ C\ Loop\ (t1s\ \rightarrow\ t2s)\ es\ ts''\ ts']$

by *fastforce*
obtain tv s **where** tv s-def: $ts'' = ts @ tv$ s length $vs = length tv$ s $\mathcal{S}\cdot\mathcal{C}' \vdash vs : ([] \rightarrow tv$ s)
 using *e-type-const-list* *assms*(3) ts'' -def(1)
 by *fastforce*
then have tv s-eq: tv s = $t1$ s $tfn = t1$ s
 using *assms*(4,5) *t-loop*(1,2)
 by *simp-all*
have $\mathcal{S}\cdot\mathcal{C} \vdash [\$Loop (t1s \rightarrow t2s) es] : (t1s \rightarrow t2s)$
 using *t-loop* *b-e-typing.loop* *e-typing-s-typing.intros*(1)
 by *fastforce*
moreover
have $\mathcal{S}\cdot\mathcal{C}' \vdash \$*es : (t1s \rightarrow t2s)$
 using *t-loop* *e-typing-s-typing.intros*(1)
 by *fastforce*
then have $\mathcal{S}\cdot\mathcal{C}' \vdash vs @ (*es) : ([] \rightarrow t2s)$
 using *tvs-eq tv s-def(3) *e-type-comp-conc*
 by *blast*
ultimately
have $\mathcal{S}\cdot\mathcal{C} \vdash [Label\ n\ [\$Loop (t1s \rightarrow t2s) es] (vs @ (*es))] : ([] \rightarrow t2s)$
 using *e-typing-s-typing.intros*(7)[of $\mathcal{S}\ \mathcal{C}\ [\$Loop (t1s \rightarrow t2s) es]\ t1s\ t2s\ vs\ @\ (*es)$]
t-loop(4) *assms*(5)
 by *fastforce*
then show *?thesis*
 using *t-loop* *e-typing-s-typing.intros*(3) tv s-def(1) tv s-eq(1)
 by *fastforce*
qed*

lemma *types-preserved-label-value*:

assumes $([Label\ n\ es0\ vs]) \rightsquigarrow ([vs])$
 $\mathcal{S}\cdot\mathcal{C} \vdash [Label\ n\ es0\ vs] : (ts \rightarrow ts')$
const-list vs

shows $\mathcal{S}\cdot\mathcal{C} \vdash vs : (ts \rightarrow ts')$

proof –

obtain $t1s\ t2s$ **where** $t2s$ -def: $ts' = (ts @ t2s)$
 $(\mathcal{S}\cdot\mathcal{C} \vdash es0 : (t1s \rightarrow t2s))$
 $(\mathcal{S}\cdot\mathcal{C}(\text{label} := [t1s] @ (\text{label } \mathcal{C})) \vdash vs : ([] \rightarrow t2s))$
 using *assms* *e-type-label*
 by *fastforce*
thus *?thesis*
 using *e-type-const-list*[of $vs\ \mathcal{S}\ \mathcal{C}(\text{label} := [t1s] @ (\text{label } \mathcal{C}))\ []\ t2s]$
assms(3) *e-typing-s-typing.intros*(3)
 by *fastforce*

qed

lemma *types-preserved-br-if*:

assumes $([\$C\ ConstInt32\ n,\ \$Br\text{-if}\ i]) \rightsquigarrow ([e])$
 $\mathcal{S}\cdot\mathcal{C} \vdash [\$C\ ConstInt32\ n,\ \$Br\text{-if}\ i] : (ts \rightarrow ts')$

$e = [\$Br\ i] \vee e = []$
shows $\mathcal{S}\cdot\mathcal{C} \vdash e : (ts \rightarrow ts')$
proof –
have $\mathcal{C} \vdash [C\ ConstInt32\ n, Br\text{-}if\ i] : (ts \rightarrow ts')$
using *unlift-b-e assms(2)*
by *fastforce*
then obtain ts'' **where** $ts''\text{-}def:\mathcal{C} \vdash [C\ ConstInt32\ n] : (ts \rightarrow ts'')$ $\mathcal{C} \vdash [Br\text{-}if\ i]$
 $: (ts'' \rightarrow ts')$
using *b-e-type-comp[of - [C ConstInt32 n] Br-if i]*
by *fastforce*
then obtain $ts\text{-}c\ ts\text{-}b$ **where** $ts\text{-}bc\text{-}def:i < length(label\ \mathcal{C})$
 $ts'' = ts\text{-}c @ ts\text{-}b @ [T\text{-}i32]$
 $ts' = ts\text{-}c @ ts\text{-}b$
 $(label\ \mathcal{C})!i = ts\text{-}b$
using *b-e-type-br-if[of C Br-if i ts'' ts' i]*
by *fastforce*
hence $ts\text{-}def:ts = ts\text{-}c @ ts\text{-}b$
using $ts''\text{-}def(1)$ *b-e-type-value*
by *fastforce*
show *?thesis*
using *assms(3)*
proof (*rule disjE*)
assume $e = [\$Br\ i]$
thus *?thesis*
using $ts\text{-}def$ *e-typing-s-typing.intros(1)* *b-e-typing.br ts-bc-def*
by *fastforce*
next
assume $e = []$
thus *?thesis*
using $ts\text{-}def$ *b-e-type-empty ts-bc-def(3)*
 $e\text{-typing-s-typing.intros(1)[of - [] (ts \rightarrow ts')]$
by *fastforce*
qed
qed

lemma *types-preserved-br-table:*

assumes $(\llbracket [C\ ConstInt32\ c, \$Br\text{-}table\ is\ i] \rrbracket \rightsquigarrow \llbracket [\$Br\ i'] \rrbracket)$
 $\mathcal{S}\cdot\mathcal{C} \vdash [C\ ConstInt32\ c, \$Br\text{-}table\ is\ i] : (ts \rightarrow ts')$
 $(i' = (is\ !\ nat\text{-}of\text{-}int\ c) \wedge length\ is > nat\text{-}of\text{-}int\ c) \vee i' = i$
shows $\mathcal{S}\cdot\mathcal{C} \vdash [\$Br\ i'] : (ts \rightarrow ts')$

proof –

have $\mathcal{C} \vdash [C\ ConstInt32\ c, Br\text{-}table\ is\ i] : (ts \rightarrow ts')$
using *unlift-b-e assms(2)*
by *fastforce*
then obtain ts'' **where** $ts''\text{-}def:\mathcal{C} \vdash [C\ ConstInt32\ c] : (ts \rightarrow ts'')$ $\mathcal{C} \vdash [Br\text{-}table\ is\ i] : (ts'' \rightarrow ts')$
using *b-e-type-comp[of - [C ConstInt32 c] Br-table is i]*
by *fastforce*
then obtain $ts\text{-}l\ ts\text{-}c$ **where** $ts\text{-}c\text{-}def:list\text{-}all\ (\lambda i. i < length(label\ \mathcal{C}) \wedge (label\ \mathcal{C})!i$

```

= ts-l) (is@[i])
                                ts'' = ts-c @ ts-l@[T-i32]
  using b-e-type-br-table[of C Br-table is i ts'' ts']
  by fastforce
  hence ts-def:ts = ts-c @ ts-l
  using ts''-def(1) b-e-type-value
  by fastforce
  have C ⊢ [Br i'] : (ts -> ts')
  using assms(3) ts-c-def(1,2) b-e-typing.br[of i' C ts-l ts-c ts'] ts-def
  unfolding list-all-length
  by (fastforce simp add: less-Suc-eq nth-append)
  thus ?thesis
  using e-typing-s-typing.intros(1)
  by fastforce
qed

```

```

lemma types-preserved-local-const:
  assumes ([Local n i vs es]) ~ ([es])
          S.C ⊢ [Local n i vs es] : (ts -> ts')
          const-list es
  shows S.C ⊢ es: (ts -> ts')
proof -
  obtain tls where (S.((s-inst S)!i)(local := (local ((s-inst S)!i)) @ (map typeof
vs), return := Some tls) ⊢ es : ([ ] -> tls))
    ts' = ts @ tls
  using e-type-local[OF assms(2)]
  by blast+
  moreover
  then have S.C ⊢ es : ([ ] -> tls)
    using assms(3) e-type-const-list
  by fastforce
  ultimately
  show ?thesis
    using e-typing-s-typing.intros(3)
  by fastforce
qed

```

```

lemma typing-map-typeof:
  assumes ves = $$* vs
          S.C ⊢ ves : ([ ] -> tvs)
  shows tvs = map typeof vs
  using assms
proof (induction ves arbitrary: vs tvs rule: List.rev-induct)
  case Nil
  hence C ⊢ [ ] : ([ ] -> tvs)
  using unlift-b-e
  by auto
  thus ?case
  using Nil

```

```

    by auto
next
case (snoc a ves)
obtain vs' v' where vs'-def:ves @ [a] = $$* (vs@[v']) vs = vs@[v']
  using snoc(2)
  by (metis Nil-is-map-conv append-is-Nil-conv list.distinct(1) rev-exhaust)
obtain tvs' where tvs'-def: $\mathcal{S}\cdot\mathcal{C} \vdash ves: ([ ] \rightarrow tvs') \mathcal{S}\cdot\mathcal{C} \vdash [a]: (tvs' \rightarrow tvs)$ 
  using snoc(3) e-type-comp
  by fastforce
hence tvs' = map typeof vs'
  using snoc(1) vs'-def
  by fastforce
moreover
have is-const a
  using vs'-def
  unfolding is-const-def
  by auto
then obtain t where t-def:tv $s = tvs' @ [t] \mathcal{S}\cdot\mathcal{C} \vdash [a]: ([ ] \rightarrow [t])$ 
  using tvs'-def(2) e-type-const[of a  $\mathcal{S} \mathcal{C} tvs' tvs$ ]
  by fastforce
have a =  $\$ C v'$ 
  using vs'-def(1)
  by auto
hence t = typeof v'
  using t-def unlift-b-e[of  $\mathcal{S} \mathcal{C} [C v'] ([ ] \rightarrow [t])$ ] b-e-type-value[of  $\mathcal{C} C v' [ ] [t] v'$ ]
  by fastforce
ultimately
show ?case
  using vs'-def t-def
  by simp
qed

```

lemma *types-preserved-call-indirect-Some:*

```

assumes  $\mathcal{S}\cdot\mathcal{C} \vdash [\$C \text{ ConstInt32 } c, \$Call\text{-indirect } j]: (ts \rightarrow ts')$ 
  stab s i' (nat-of-int c) = Some cl
  stypes s i' j = tf
  cl-type cl = tf
  store-typing s  $\mathcal{S}$ 
  i' < length (inst s)
   $\mathcal{C} = (s\text{-inst } \mathcal{S} ! i') (\text{local} := \text{local } (s\text{-inst } \mathcal{S} ! i') @ tvs, \text{label} := \text{arb-labs},$ 
return := arb-return)
shows  $\mathcal{S}\cdot\mathcal{C} \vdash [Callcl \text{ cl}]: (ts \rightarrow ts')$ 
proof –
obtain t1s t2s where tf-def:tf = (t1s  $\rightarrow$  t2s)
  using tf.exhaust by blast
obtain ts'' where ts''-def: $\mathcal{C} \vdash [C \text{ ConstInt32 } c]: (ts \rightarrow ts'')$ 
   $\mathcal{C} \vdash [Call\text{-indirect } j]: (ts'' \rightarrow ts')$ 
  using e-type-comp[of  $\mathcal{S} \mathcal{C} [\$C \text{ ConstInt32 } c] \$Call\text{-indirect } j \text{ ts } ts'$ ]
  assms(1)

```

```

      unlift-b-e[of  $\mathcal{S} \mathcal{C}$  [ $C$   $ConstInt32$   $c$ ]]
      unlift-b-e[of  $\mathcal{S} \mathcal{C}$  [ $Call$ -indirect  $j$ ]]
    by fastforce
  hence  $ts'' = ts @ [(T-i32)]$ 
    using b-e-type-value
    unfolding typeof-def
    by fastforce
  moreover
  have  $i' < length (s-inst \mathcal{S})$ 
    using assms(5,6) store-typing-imp-inst-length-eq
    by fastforce
  hence  $stypes\text{-}eq:types\text{-}t (s-inst \mathcal{S} ! i') = types (inst s ! i')$ 
    using store-typing-imp-inst-typing[OF assms(5)] store-typing-imp-inst-length-eq[OF
  assms(5)]
    unfolding inst-typing.simps
    by fastforce
  obtain  $ts''a$  where  $ts''a\text{-}def:j < length (types\text{-}t \mathcal{C})$ 
     $ts'' = ts''a @ t1s @ [T-i32]$ 
     $ts' = ts''a @ t2s$ 
     $types\text{-}t \mathcal{C} ! j = (t1s \rightarrow t2s)$ 
    using b-e-type-call-indirect[OF  $ts''\text{-}def(2)$ , of  $j$ ] tf-def assms(3,7) stypes-eq
    unfolding stypes-def
    by fastforce
  moreover
  obtain  $tf'$  where  $tf'\text{-}def:cl\text{-}typing \mathcal{S} cl tf'$ 
    using assms(2,5,6) stab-typed-some-imp-cl-typed
    by blast
  hence  $cl\text{-}typing \mathcal{S} cl tf$ 
    using assms(4)
    unfolding cl-typing.simps cl-type-def
    by auto
  hence  $\mathcal{S}\cdot\mathcal{C} \vdash [Callcl\ cl] : tf$ 
    using e-typing-s-typing.intros(6) assms(6,7)  $ts''a\text{-}def(1)$ 
    by fastforce
  ultimately
  show  $\mathcal{S}\cdot\mathcal{C} \vdash [Callcl\ cl] : (ts \rightarrow ts')$ 
    using tf-def e-typing-s-typing.intros(3)
    by auto
qed

```

lemma *types-preserved-call-indirect-None*:

```

  assumes  $\mathcal{S}\cdot\mathcal{C} \vdash [\$C\ ConstInt32\ c, \$Call\text{-}indirect\ j] : (ts \rightarrow ts')$ 
  shows  $\mathcal{S}\cdot\mathcal{C} \vdash [Trap] : (ts \rightarrow ts')$ 
  using e-typing-s-typing.intros(4)
  by blast

```

lemma *types-preserved-callcl-native*:

```

  assumes  $\mathcal{S}\cdot\mathcal{C} \vdash ves @ [Callcl\ cl] : (ts \rightarrow ts')$ 
     $cl = Func\text{-}native\ i (t1s \rightarrow t2s) tfs\ es$ 

```

$ves = \$\$* \text{ vs}$
 $\text{length } \text{vs} = n$
 $\text{length } \text{tfs} = k$
 $\text{length } \text{t1s} = n$
 $\text{length } \text{t2s} = m$
 $n\text{-zeros } \text{tfs} = \text{zs}$
 $\text{store-typing } s \ \mathcal{S}$

shows $\mathcal{S}\cdot\mathcal{C} \vdash [\text{Local } m \ i \ (\text{vs} \ @ \ \text{zs}) \ [\text{\$Block} (\ [] \ \rightarrow \ \text{t2s}) \ \text{es}]] : (\text{ts} \ \rightarrow \ \text{ts}')$

proof –

obtain ts'' **where** $\text{ts}''\text{-def}:\mathcal{S}\cdot\mathcal{C} \vdash \text{ves} : (\text{ts} \ \rightarrow \ \text{ts}'')$ $\mathcal{S}\cdot\mathcal{C} \vdash [\text{Callcl } \text{cl}] : (\text{ts}'' \ \rightarrow \ \text{ts}')$
using $\text{assms}(1)$ $e\text{-type-comp}$
by fastforce
have $\text{ves-c}:\text{const-list } \text{ves}$
using $\text{is-const-list}[OF \ \text{assms}(3)]$
by simp
then obtain tvs **where** $\text{tvs-def}:\text{ts}'' = \text{ts} \ @ \ \text{tvs}$
 $\text{length } \text{t1s} = \text{length } \text{tvs}$
 $\mathcal{S}\cdot\mathcal{C} \vdash \text{ves} : (\ [] \ \rightarrow \ \text{tvs})$
using $\text{ts}''\text{-def}(1)$ $e\text{-type-const-list}[of \ \text{ves } \mathcal{S} \ \mathcal{C} \ \text{ts} \ \text{ts}']$ assms
by fastforce
obtain $\text{ts-c } \mathcal{C}'$ **where** $\text{ts-c-def}:(\text{ts}'' = \text{ts-c} \ @ \ \text{t1s})$
 $(\text{ts}' = \text{ts-c} \ @ \ \text{t2s})$
 $i < \text{length } (s\text{-inst } \mathcal{S})$
 $\mathcal{C}' = ((s\text{-inst } \mathcal{S})!i)$
 $(\mathcal{C}'(\text{local} := (\text{local } \mathcal{C}') \ @ \ \text{t1s} \ @ \ \text{tfs}, \text{label} := ([\text{t2s}] \ @ \ (\text{label}$
 $\mathcal{C}')), \text{return} := \text{Some } \text{t2s}) \vdash \text{es} : (\ [] \ \rightarrow \ \text{t2s}))$
using $e\text{-type-callcl-native}[OF \ \text{ts}''\text{-def}(2)]$ $\text{assms}(2)$
by fastforce
have $\text{inst-typing } \mathcal{S} \ (inst \ s \ ! \ i) \ (s\text{-inst } \mathcal{S} \ ! \ i)$
using $\text{store-typing-imp-inst-length-eq}[OF \ \text{assms}(9)]$ $\text{store-typing-imp-inst-typing}[OF \ \text{assms}(9)]$
 $\text{ts-c-def}(3)$
by simp
obtain \mathcal{C}'' **where** $\mathcal{C}''\text{-def}:\mathcal{C}'' = \mathcal{C}'(\text{local} := (\text{local } \mathcal{C}') \ @ \ \text{t1s} \ @ \ \text{tfs}, \text{return} := \text{Some}$
 $\text{t2s})$
by blast
hence $\mathcal{C}''(\text{label} := ([\text{t2s}] \ @ \ (\text{label } \mathcal{C}'')))) = \mathcal{C}'(\text{local} := (\text{local } \mathcal{C}') \ @ \ \text{t1s} \ @ \ \text{tfs},$
 $\text{label} := ([\text{t2s}] \ @ \ (\text{label } \mathcal{C}')), \text{return} := \text{Some } \text{t2s})$
by fastforce
hence $\mathcal{S}\cdot\mathcal{C}'' \vdash [\text{\$Block} (\ [] \ \rightarrow \ \text{t2s}) \ \text{es}] : (\ [] \ \rightarrow \ \text{t2s})$
using $\text{ts-c-def } b\text{-e-typing.block}[of \ (\ [] \ \rightarrow \ \text{t2s}) \ [] \ \text{t2s} - \ \text{es}]$ $e\text{-typing-s-typing.intros}(1)[of$
 $- [\text{Block} (\ [] \ \rightarrow \ \text{t2s}) \ \text{es}]$
by fastforce
moreover
have $t\text{-eqs}:\text{ts} = \text{ts-c } \text{t1s} = \text{tvs}$
using $\text{tvs-def}(1,2)$ $\text{ts-c-def}(1)$
by simp-all
have $1:\text{tfs} = \text{map } \text{typeof } \ \text{zs}$
using $n\text{-zeros-typeof } \ \text{assms}(8)$

by *auto*
 have $t1s = \text{map } \text{typeof } vs$
 using *typing-map-typeof* $\text{assms}(3)$ *tvs-def* *t-eqs*
 by *fastforce*
 hence $(t1s @ tfs) = \text{map } \text{typeof } (vs @ zs)$
 using *1*
 by *simp*
 ultimately
 have $\mathcal{S} \cdot \text{Some } t2s \vdash\text{-i } (vs @ zs); ([\$Block \ [] \ \rightarrow t2s] \text{ es}) : t2s$
 using *e-typing-s-typing.intros(8)* *ts-c-def* *c''-def*
 by *fastforce*
 thus *?thesis*
 using *e-typing-s-typing.intros(3,5)* *ts-c-def* *t-eqs(1)* *assms(2,7)*
 by *fastforce*
 qed

lemma *types-preserved-callcl-host-some*:
 assumes $\mathcal{S} \cdot \mathcal{C} \vdash \text{ves} @ [Callcl \ cl] : (ts \rightarrow ts')$
 $cl = \text{Func-host } (t1s \rightarrow t2s) f$
 $\text{ves} = \$\$* \ vcs$
 $\text{length } vcs = n$
 $\text{length } t1s = n$
 $\text{length } t2s = m$
 $\text{host-apply } s \ (t1s \rightarrow t2s) f \ vcs \ hs = \text{Some } (s', vcs')$
 $\text{store-typing } s \ \mathcal{S}$
 shows $\mathcal{S} \cdot \mathcal{C} \vdash \$\$* \ vcs' : (ts \rightarrow ts')$
proof –
 obtain ts'' where $ts''\text{-def} : \mathcal{S} \cdot \mathcal{C} \vdash \text{ves} : (ts \rightarrow ts'')$ $\mathcal{S} \cdot \mathcal{C} \vdash [Callcl \ cl] : (ts'' \rightarrow ts')$
 using *assms(1)* *e-type-comp*
 by *fastforce*
 have $\text{ves-c:const-list } \text{ves}$
 using *is-const-list[OF assms(3)]*
 by *simp*
 then obtain tvs where $tvs\text{-def} : ts'' = ts @ tvs$
 $\text{length } t1s = \text{length } tvs$
 $\mathcal{S} \cdot \mathcal{C} \vdash \text{ves} : ([\] \rightarrow tvs)$
 using $ts''\text{-def}(1)$ *e-type-const-list[of ves S C ts ts'']* *assms*
 by *fastforce*
 hence $ts'' = ts @ t1s$
 $ts' = ts @ t2s$
 using *e-type-callcl-host[OF ts''-def(2) assms(2)]*
 by *auto*
moreover
 hence *list-all2* *types-agree* $t1s \ vcs$
 using *e-typing-imp-list-types-agree[where ?ts' = []]* *assms(3)* *tvs-def(1,3)*
 by *fastforce*
 hence $\mathcal{S} \cdot \mathcal{C} \vdash \$\$* \ vcs' : ([\] \rightarrow t2s)$
 using *list-types-agree-imp-e-typing* *host-apply-respect-type[OF - assms(7)]*
 by *fastforce*

ultimately
show *?thesis*
using *e-typing-s-typing.intros(3)*
by *fastforce*
qed

lemma *types-imp-concat*:
assumes $\mathcal{S}\cdot\mathcal{C} \vdash es \ @ \ [e] \ @ \ es' : (ts \rightarrow ts')$
 $\wedge tes \ tes'. ((\mathcal{S}\cdot\mathcal{C} \vdash [e] : (tes \rightarrow tes'))) \implies (\mathcal{S}\cdot\mathcal{C} \vdash [e] : (tes \rightarrow tes'))$
shows $\mathcal{S}\cdot\mathcal{C} \vdash es \ @ \ [e] \ @ \ es' : (ts \rightarrow ts')$
proof –
obtain ts'' where $\mathcal{S}\cdot\mathcal{C} \vdash es : (ts \rightarrow ts'')$
 $\mathcal{S}\cdot\mathcal{C} \vdash [e] \ @ \ es' : (ts'' \rightarrow ts')$
using *e-type-comp-conc1[of - - es [e] @ es'] assms(1)*
by *fastforce*
moreover
then obtain ts''' where $\mathcal{S}\cdot\mathcal{C} \vdash [e] : (ts'' \rightarrow ts''')$ $\mathcal{S}\cdot\mathcal{C} \vdash es' : (ts''' \rightarrow ts')$
using *e-type-comp-conc1[of - - [e] es' ts'' ts'] assms*
by *fastforce*
ultimately
show *?thesis*
using *assms(2) e-type-comp-conc[of - - es ts ts'' [e] ts''']*
e-type-comp-conc[of - - es @ [e] ts ts''']
by *fastforce*
qed

lemma *type-const-return*:
assumes $Lfilled \ i \ lholed \ (vs \ @ \ [\$Return]) \ LI$
 $(return \ C) = Some \ tcs$
 $length \ tcs = length \ vs$
 $\mathcal{S}\cdot\mathcal{C} \vdash LI : (ts \rightarrow ts')$
 $const-list \ vs$
shows $\mathcal{S}\cdot\mathcal{C}' \vdash vs : ([] \rightarrow tcs)$
using *assms*
proof (*induction i arbitrary: ts ts' lholed C LI C'*)
case 0
obtain $vs' \ es'$ where $LI = (vs' \ @ \ (vs \ @ \ [\$Return])) \ @ \ es'$
using *Lfilled.simps[of 0 lholed (vs @ [Return]) LI] 0(1)*
by *fastforce*
then obtain $ts'' \ ts'''$ where $\mathcal{S}\cdot\mathcal{C} \vdash vs' : (ts \rightarrow ts'')$
 $\mathcal{S}\cdot\mathcal{C} \vdash (vs \ @ \ [\$Return]) : (ts'' \rightarrow ts''')$
 $\mathcal{S}\cdot\mathcal{C} \vdash es' : (ts''' \rightarrow ts')$
using *e-type-comp-conc2[of S C vs' (vs @ [Return]) es'] 0(4)*
by *fastforce*
then obtain $ts-b$ where $ts-b-def:\mathcal{S}\cdot\mathcal{C} \vdash vs : (ts'' \rightarrow ts-b)$ $\mathcal{S}\cdot\mathcal{C} \vdash [\$Return] : (ts-b \rightarrow ts''')$
using *e-type-comp-conc1*
by *fastforce*
then obtain $ts-c$ where $ts-c-def:ts-b = ts-c \ @ \ tcs \ (return \ C) = Some \ tcs$

```

    using 0(2) b-e-type-return[of C] unlift-b-e[of S C [Return] ts-b -> ts'']
    by fastforce
  obtain tcs' where ts-b = ts'' @ tcs' length vs = length tcs' S.C' ⊢ vs : ([] ->
tcs')
    using ts-b-def(1) e-type-const-list 0(5)
    by fastforce
  thus ?case
    using 0(3) ts-c-def
    by simp
next
case (Suc i)
obtain vs' n l les les' LK where es-def:lholed = (LRec vs' n les l les')
                                Lfilled i l (vs @ [$Return]) LK
                                LI = (vs' @ [Label n les LK] @ les')
    using Lfilled.simps[of (Suc i) lholed (vs @ [$Return]) LI] Suc(2)
    by fastforce
  then obtain ts'' ts''' where S.C ⊢ [Label n les LK] : (ts'' -> ts''')
    using e-type-comp-conc2[of S C vs' [Label n les LK] les'] Suc(5)
    by fastforce
  then obtain tls t2s where
    ts''' = ts'' @ t2s
    length tls = n
    S.C ⊢ les : (tls -> t2s)
    S.C(\label := [tls] @ label C) ⊢ LK : ([] -> t2s)
    return (C(\label := [tls] @ label C)) = Some tcs
    using e-type-label[of S C n les LK ts'' ts'''] Suc(3)
    by fastforce
  then show ?case
    using Suc(1)[OF es-def(2) - assms(3) - assms(5)]
    by fastforce
qed

```

lemma *types-preserved-return*:

```

  assumes (|[Local n i vls LI]) ~> (|ves)
          S.C ⊢ [Local n i vls LI] : (ts -> ts')
          const-list ves
          length ves = n
          Lfilled j lholed (ves @ [$Return]) LI
  shows S.C ⊢ ves : (ts -> ts')
proof -
  obtain tls C' where l-def:i < length (s-inst S)
    C' = ((s-inst S)!i)(\local := (local ((s-inst S)!i)) @ (map typeof
vls), return := Some tls)
    S.C' ⊢ LI : ([] -> tls)
    ts' = ts @ tls
    length tls = n
  using e-type-local[OF assms(2)]
  by blast
  hence S.C ⊢ ves : ([] -> tls)

```

```

    using type-const-return[OF assms(5) - - l-def(3)] assms(3-5)
    by fastforce
  thus ?thesis
    using e-typing-s-typing.intros(3) l-def(4)
    by fastforce
qed

lemma type-const-br:
  assumes Lfilled i lholed (vs @ [\$Br (i+k)]) LI
    length (label C) > k
    (label C)!k = tcs
    length tcs = length vs
    S.C ⊢ LI : (ts -> ts')
    const-list vs
  shows S.C' ⊢ vs : ([] -> tcs)
  using assms
proof (induction i arbitrary: k ts ts' lholed C LI C')
  case 0
  obtain vs' es' where LI = (vs' @ (vs @ [\$Br (0+k)]) @ es')
    using Lfilled.simps[of 0 lholed (vs @ [\$Br (0 + k)]) LI] 0(1)
    by fastforce
  then obtain ts'' ts''' where S.C ⊢ vs' : (ts -> ts'')
    S.C ⊢ (vs @ [\$Br (0+k)]) : (ts'' -> ts''')
    S.C ⊢ es' : (ts''' -> ts')
    using e-type-comp-conc2[of S C vs' (vs @ [\$Br (0+k)]) es'] 0(5)
    by fastforce
  then obtain ts-b where ts-b-def:S.C ⊢ vs : (ts'' -> ts-b) S.C ⊢ [\$Br (0+k)] :
    (ts-b -> ts''')
    using e-type-comp-conc1
    by fastforce
  then obtain ts-c where ts-c-def:ts-b = ts-c @ tcs (label C)!k = tcs
    using 0(3) b-e-type-br[of C Br (0 + k)] unlift-b-e[of S C [Br (0 + k)] ts-b ->
    ts''']
    by fastforce
  obtain tcs' where ts-b = ts'' @ tcs' length vs = length tcs' S.C' ⊢ vs : ([] ->
  tcs')
    using ts-b-def(1) e-type-const-list 0(6)
    by fastforce
  thus ?case
    using 0(4) ts-c-def
    by simp
next
  case (Suc i k ts ts' lholed C LI)
  obtain vs' n l les les' LK where es-def:lholed = (LRec vs' n les l les')
    Lfilled i l (vs @ [\$Br (i + (Suc k))]) LK
    LI = (vs' @ [Label n les LK] @ les')
    using Lfilled.simps[of (Suc i) lholed (vs @ [\$Br ((Suc i) + k)]) LI] Suc(2)
    by fastforce
  then obtain ts'' ts''' where S.C ⊢ [Label n les LK] : (ts'' -> ts''')

```

using *e-type-comp-conc2*[of $\mathcal{S} \mathcal{C} \text{ vs}'$ [Label $n \text{ les } LK$] les'] *Suc*(6)
by *fastforce*
moreover
then obtain $\text{tls } \mathcal{C}'' \text{ ts}''''$ **where** $\mathcal{S} \cdot \mathcal{C}'' \vdash LK : ([\] \rightarrow \text{ts}''')$ $\mathcal{C}'' = \mathcal{C}(\text{label} := [\text{tls}]$
 $\text{@ } (\text{label } \mathcal{C}))$
 $\text{length } (\text{label } \mathcal{C}'') > (\text{Suc } k)$
 $(\text{label } \mathcal{C}'')!(\text{Suc } k) = \text{tcs}$
using *e-type-label*[of $\mathcal{S} \mathcal{C} \text{ n les } LK \text{ ts}'' \text{ ts}'''$] *Suc*(3,4)
by *fastforce*
then show *?case*
using *Suc*(1) *es-def*(2) *assms*(4,6)
by *fastforce*
qed

lemma *types-preserved-br*:
assumes $([\text{Label } n \text{ es0 } LI]) \rightsquigarrow (\text{vs } \text{@ } \text{es0})$
 $\mathcal{S} \cdot \mathcal{C} \vdash [\text{Label } n \text{ es0 } LI] : (\text{ts} \rightarrow \text{ts}')$
const-list vs
 $\text{length } \text{vs} = n$
 $L\text{filled } i \text{ lholed } (\text{vs } \text{@ } [\text{Br } i]) \text{ LI}$
shows $\mathcal{S} \cdot \mathcal{C} \vdash (\text{vs } \text{@ } \text{es0}) : (\text{ts} \rightarrow \text{ts}')$
proof –
obtain $\text{tls } \text{t2s } \mathcal{C}'$ **where** *l-def*: $(\text{ts}' = (\text{ts} \text{@ } \text{t2s}))$
 $(\mathcal{S} \cdot \mathcal{C} \vdash \text{es0} : (\text{tls} \rightarrow \text{t2s}))$
 $\mathcal{C}' = \mathcal{C}(\text{label} := [\text{tls}] \text{@ } (\text{label } \mathcal{C}))$
 $\text{length } (\text{label } \mathcal{C}') > 0$
 $(\text{label } \mathcal{C}')!0 = \text{tls}$
 $\text{length } \text{tls} = n$
 $(\mathcal{S} \cdot \mathcal{C}(\text{label} := [\text{tls}] \text{@ } (\text{label } \mathcal{C})) \vdash LI : ([\] \rightarrow \text{t2s}))$
using *e-type-label*[of $\mathcal{S} \mathcal{C} \text{ n es0 } LI \text{ ts } \text{ts}'$] *assms*(2)
by *fastforce*
hence $\mathcal{S} \cdot \mathcal{C} \vdash \text{vs} : ([\] \rightarrow \text{tls})$
using *assms*(3–5) *type-const-br*[of $i \text{ lholed } \text{vs } 0 \text{ LI } \mathcal{C}' \text{ tls}$]
by *fastforce*
thus *?thesis*
using *l-def*(1,2) *e-type-comp-conc* *e-typing-s-typing.intros*(3)
by *fastforce*
qed

lemma *store-local-label-empty*:
assumes $i < \text{length } (\text{s-inst } \mathcal{S})$
store-typing s S
shows $\text{label } ((\text{s-inst } \mathcal{S})!i) = [\]$ $\text{local } ((\text{s-inst } \mathcal{S})!i) = [\]$
proof –
obtain *insts* **where** *inst-typ*: $\text{list-all2 } (\text{inst-typing } \mathcal{S}) \text{ insts } (\text{s-inst } \mathcal{S})$
using *assms*(2)
unfolding *store-typing.simps*
by *auto*
thus $\text{label } ((\text{s-inst } \mathcal{S})!i) = [\]$

```

    using assms(1)
    unfolding inst-typing.simps List.list-all2-conv-all-nth
    by fastforce
  show local ((s-inst  $\mathcal{S}$ )!i) = []
    using assms(1) inst-typ
    unfolding inst-typing.simps List.list-all2-conv-all-nth
    by fastforce
qed

lemma types-preserved-b-e1:
  assumes ((es)  $\rightsquigarrow$  (es'))
    store-typing s  $\mathcal{S}$ 
     $\mathcal{S} \cdot \mathcal{C} \vdash es : (ts \rightarrow ts')$ 
  shows  $\mathcal{S} \cdot \mathcal{C} \vdash es' : (ts \rightarrow ts')$ 
  using assms(1)
proof (cases rule: reduce-simple.cases)
  case (unop-i32 c iop)
  thus ?thesis
    using assms(1,3) types-preserved-unop-testop-cutop
    by simp
next
  case (unop-i64 c iop)
  thus ?thesis
    using assms(1, 3) types-preserved-unop-testop-cutop
    by simp
next
  case (unop-f32 c fop)
  thus ?thesis
    using assms(1, 3) types-preserved-unop-testop-cutop
    by simp
next
  case (unop-f64 c fop)
  thus ?thesis
    using assms(1, 3) types-preserved-unop-testop-cutop
    by simp
next
  case (binop-i32-Some iop c1 c2 c)
  thus ?thesis
    using assms(1, 3) types-preserved-binop-relop
    by simp
next
  case (binop-i32-None iop c1 c2)
  thus ?thesis
    by (simp add: e-typing-s-typing.intros(4))
next
  case (binop-i64-Some iop c1 c2 c)
  thus ?thesis
    using assms(1, 3) types-preserved-binop-relop
    by simp

```

```

next
  case (binop-i64-None iop c1 c2)
  thus ?thesis
    by (simp add: e-typing-s-typing.intros(4))
next
  case (binop-f32-Some fop c1 c2 c)
  thus ?thesis
    using assms(1, 3) types-preserved-binop-relop
    by simp
next
  case (binop-f32-None fop c1 c2)
  thus ?thesis
    by (simp add: e-typing-s-typing.intros(4))
next
  case (binop-f64-Some fop c1 c2 c)
  then show ?thesis
    using assms(1, 3) types-preserved-binop-relop
    by simp
next
  case (binop-f64-None fop c1 c2)
  then show ?thesis
    by (simp add: e-typing-s-typing.intros(4))
next
  case (testop-i32 c testop)
  then show ?thesis
    using assms(1, 3) types-preserved-unop-testop-cvtop
    by simp
next
  case (testop-i64 c testop)
  then show ?thesis
    using assms(1, 3) types-preserved-unop-testop-cvtop
    by simp
next
  case (relop-i32 c1 c2 iop)
  then show ?thesis
    using assms(1, 3) types-preserved-binop-relop
    by simp
next
  case (relop-i64 c1 c2 iop)
  then show ?thesis
    using assms(1, 3) types-preserved-binop-relop
    by simp
next
  case (relop-f32 c1 c2 fop)
  then show ?thesis
    using assms(1, 3) types-preserved-binop-relop
    by simp
next
  case (relop-f64 c1 c2 fop)

```

```

then show ?thesis
  using assms(1, 3) types-preserved-binop-relop
  by simp
next
  case (convert-Some t1 v t2 sx v')
  then show ?thesis
    using assms(1, 3) types-preserved-unop-testop-cvtop
    by simp
next
  case (convert-None t1 v t2 sx)
  then show ?thesis
    using e-typing-s-typing.intros(4)
    by simp
next
  case (reinterpret t1 v t2)
  then show ?thesis
    using assms(1, 3) types-preserved-unop-testop-cvtop
    by simp
next
  case unreachable
  then show ?thesis
    using e-typing-s-typing.intros(4)
    by simp
next
  case nop
  then have  $\mathcal{C} \vdash [Nop] : (ts \rightarrow ts')$ 
    using assms(3) unlift-b-e
    by simp
  then show ?thesis
    using nop b-e-typing.empty e-typing-s-typing.intros(1,3)
    apply (induction [Nop] ts -> ts' arbitrary: ts ts')
    apply simp-all
    apply (metis list.simps(8))
    apply blast
  done
next
  case (drop v)
  then show ?thesis
    using assms(1, 3) types-preserved-drop
    by simp
next
  case (select-false v1 v2)
  then show ?thesis
    using assms(1, 3) types-preserved-select
    by simp
next
  case (select-true n v1 v2)
  then show ?thesis
    using assms(1, 3) types-preserved-select

```

```

    by simp
next
case (block vs n t1s t2s m es)
then show ?thesis
  using assms(1, 3) types-preserved-block
  by simp
next
case (loop vs n t1s t2s m es)
then show ?thesis
  using assms(1, 3) types-preserved-loop
  by simp
next
case (if-false tf e1s e2s)
then show ?thesis
  using assms(1, 3) types-preserved-if
  by simp
next
case (if-true n tf e1s e2s)
then show ?thesis
  using assms(1, 3) types-preserved-if
  by simp
next
case (label-const ts es)
then show ?thesis
  using assms(1, 3) types-preserved-label-value
  by simp
next
case (label-trap ts es)
then show ?thesis
  by (simp add: e-typing-s-typing.intros(4))
next
case (br vs n i lholed LI es)
then show ?thesis
  using assms(1, 3) types-preserved-br
  by fastforce
next
case (br-if-false n i)
then show ?thesis
  using assms(1, 3) types-preserved-br-if
  by fastforce
next
case (br-if-true n i)
then show ?thesis
  using assms(1, 3) types-preserved-br-if
  by fastforce
next
case (br-table is' c i')
then show ?thesis
  using assms(1, 3) types-preserved-br-table

```

```

    by fastforce
next
case (br-table-length is' c i')
then show ?thesis
  using assms(1, 3) types-preserved-br-table
  by fastforce
next
case (local-const i vs)
then show ?thesis
  using assms(1, 3) types-preserved-local-const
  by fastforce
next
case (local-trap i vs)
then show ?thesis
  by (simp add: e-typing-s-typing.intros(4))
next
case (return n j lholed es i vls)
then show ?thesis
  using assms(1, 3) types-preserved-return
  by fastforce
next
case (tee-local v i)
then show ?thesis
  using assms(1, 3) types-preserved-tee-local
  by simp
next
case (trap lholed)
then show ?thesis
  by (simp add: e-typing-s-typing.intros(4))
qed

lemma types-preserved-b-e:
  assumes (|es| ~> |es'|)
    store-typing s S
    S.None ⊢-i vs;es : ts
  shows S.None ⊢-i vs;es' : ts
proof -
  have i < (length (s-inst S))
    using assms(3) s-typing.cases
    by blast
  moreover
  obtain tus C where defs: tus = map typeof vs C = ((s-inst S)!i)(|local := (local
((s-inst S)!i) @ tus), return := None) S.C ⊢ es : ([] -> ts)
    using assms(3)
    unfolding s-typing.simps
    by blast
  have S.C ⊢ es' : ([] -> ts)
    using assms(1,2) defs(3) types-preserved-b-e1
    by simp

```

ultimately show *?thesis*
using *defs*
unfolding *s-typing.simps*
by *auto*
qed

lemma *types-preserved-store:*

assumes $\mathcal{S}\cdot\mathcal{C} \vdash [\$C \text{ ConstInt32 } k, \$C v, \$Store t tp a \text{ off}] : (ts \rightarrow ts')$
shows $\mathcal{S}\cdot\mathcal{C} \vdash [] : (ts \rightarrow ts')$
types-agree t v

proof –

obtain $ts'' ts'''$ **where** $ts\text{-def}:\mathcal{S}\cdot\mathcal{C} \vdash [\$C \text{ ConstInt32 } k] : (ts \rightarrow ts'')$
 $\mathcal{S}\cdot\mathcal{C} \vdash [\$C v] : (ts'' \rightarrow ts''')$
 $\mathcal{S}\cdot\mathcal{C} \vdash [\$Store t tp a \text{ off}] : (ts''' \rightarrow ts')$
using *assms e-type-comp-conc2*[of $\mathcal{S} \mathcal{C} [\$C \text{ ConstInt32 } k] [\$C v] [\$Store t tp a \text{ off}]]
by *fastforce*
then have $ts'' = ts@[T-i32]$
using *b-e-type-value*[of $\mathcal{C} \mathcal{C} \text{ ConstInt32 } k ts ts''$]
unlift-b-e[of $\mathcal{S} \mathcal{C} [\mathcal{C} (\text{ConstInt32 } k)] (ts \rightarrow ts'')$]
unfolding *typeof-def*
by *fastforce*
hence $ts''' = ts@[T-i32], (\text{typeof } v)$
using *ts-def*(2) *b-e-type-value*[of $\mathcal{C} \mathcal{C} v ts'' ts'''$]
unlift-b-e[of $\mathcal{S} \mathcal{C} [\mathcal{C} v] (ts'' \rightarrow ts''')$]
by *fastforce*
hence $ts = ts'$ *types-agree t v*
using *ts-def*(3) *b-e-type-store*[of $\mathcal{C} \text{ Store } t tp a \text{ off } ts''' ts'$]
unlift-b-e[of $\mathcal{S} \mathcal{C} [\text{Store } t tp a \text{ off}] (ts''' \rightarrow ts')$]
unfolding *types-agree-def*
by *fastforce+*
thus $\mathcal{S}\cdot\mathcal{C} \vdash [] : (ts \rightarrow ts')$ *types-agree t v*
using *b-e-type-empty*[of $\mathcal{C} ts ts'$] *e-typing-s-typing.intros(1)*
by *fastforce+*$

qed

lemma *types-preserved-current-memory:*

assumes $\mathcal{S}\cdot\mathcal{C} \vdash [\$Current\text{-memory}] : (ts \rightarrow ts')$
shows $\mathcal{S}\cdot\mathcal{C} \vdash [\$C \text{ ConstInt32 } c] : (ts \rightarrow ts')$

proof –

have $ts' = ts@[T-i32]$
using *assms b-e-type-current-memory unlift-b-e*[of $\mathcal{S} \mathcal{C} [\text{Current-memory}]$]
by *fastforce*
thus *?thesis*
using *b-e-typing.const*[of $\mathcal{C} \text{ ConstInt32 } c$] *e-typing-s-typing.intros(1,3)*
unfolding *typeof-def*
by *fastforce*

qed

lemma *types-preserved-grow-memory*:
assumes $\mathcal{S}\cdot\mathcal{C} \vdash [\$C \text{ ConstInt32 } c, \$Grow\text{-memory}] : (ts \rightarrow ts')$
shows $\mathcal{S}\cdot\mathcal{C} \vdash [\$C \text{ ConstInt32 } c'] : (ts \rightarrow ts')$
proof –
obtain ts'' **where** $ts''\text{-def}:\mathcal{S}\cdot\mathcal{C} \vdash [\$C \text{ ConstInt32 } c] : (ts \rightarrow ts'')$
 $\mathcal{S}\cdot\mathcal{C} \vdash [\$Grow\text{-memory}] : (ts'' \rightarrow ts')$
using *e-type-comp assms*
by (*metis append-butlast-last-id butlast.simps(2) last.simps list.distinct(1)*)
have $ts'' = ts@[T-i32]$
using *b-e-type-value*[*of* $\mathcal{C} \text{ ConstInt32 } c \text{ ts } ts''$]
 $unlift\text{-b-e}$ [*of* $\mathcal{S} \mathcal{C} [\mathcal{C} \text{ ConstInt32 } c]$ $ts''\text{-def}(1)$]
unfolding *typeof-def*
by *fastforce*
moreover
hence $ts'' = ts'$
using $ts''\text{-def}$ *b-e-type-grow-memory*[*of* $ts'' \text{ ts}'$] $unlift\text{-b-e}$ [*of* $\mathcal{S} \mathcal{C} [Grow\text{-memory}]$]
by *fastforce*
ultimately
show $\mathcal{S}\cdot\mathcal{C} \vdash [\$C \text{ ConstInt32 } c'] : (ts \rightarrow ts')$
using *e-typing-s-typing.intros(1,3)*
 $b\text{-e-typing.const}$ [*of* $\mathcal{C} \text{ ConstInt32 } c'$]
unfolding *typeof-def*
by *fastforce*
qed

lemma *types-preserved-set-global*:
assumes $\mathcal{S}\cdot\mathcal{C} \vdash [\$C \text{ } v, \$Set\text{-global } j] : (ts \rightarrow ts')$
shows $\mathcal{S}\cdot\mathcal{C} \vdash [] : (ts \rightarrow ts')$
 $tg\text{-t } (global \mathcal{C} ! j) = \text{typeof } v$
proof –
obtain ts'' **where** $ts''\text{-def}:\mathcal{S}\cdot\mathcal{C} \vdash [\$C \text{ } v] : (ts \rightarrow ts'')$
 $\mathcal{S}\cdot\mathcal{C} \vdash [\$Set\text{-global } j] : (ts'' \rightarrow ts')$
using *e-type-comp assms*
by (*metis append-butlast-last-id butlast.simps(2) last.simps list.distinct(1)*)
hence $ts'' = ts@[typeof v]$
using *b-e-type-value* $unlift\text{-b-e}$ [*of* $\mathcal{S} \mathcal{C} [\mathcal{C} \text{ } v]$]
by *fastforce*
hence $ts = ts' \text{ tg-t } (global \mathcal{C} ! j) = \text{typeof } v$
using *b-e-type-set-global* $ts''\text{-def}(2)$ $unlift\text{-b-e}$ [*of* $\mathcal{S} \mathcal{C} [Set\text{-global } j]$]
by *fastforce+*
thus $\mathcal{S}\cdot\mathcal{C} \vdash [] : (ts \rightarrow ts')$ $tg\text{-t } (global \mathcal{C} ! j) = \text{typeof } v$
using *b-e-type-empty*[*of* $\mathcal{C} \text{ ts } ts'$] *e-typing-s-typing.intros(1)*
by *fastforce+*
qed

lemma *types-preserved-load*:
assumes $\mathcal{S}\cdot\mathcal{C} \vdash [\$C \text{ ConstInt32 } k, \$Load \text{ } t \text{ } tp \text{ } a \text{ } off] : (ts \rightarrow ts')$
 $\text{typeof } v = t$
shows $\mathcal{S}\cdot\mathcal{C} \vdash [\$C \text{ } v] : (ts \rightarrow ts')$

proof –

obtain ts'' **where** $ts''\text{-def}:\mathcal{S}\cdot\mathcal{C} \vdash [\$C \text{ ConstInt32 } k] : (ts \rightarrow ts')$
 $\mathcal{S}\cdot\mathcal{C} \vdash [\$Load \ t \ tp \ a \ off] : (ts'' \rightarrow ts')$

using *e-type-comp assms*
by (*metis append-butlast-last-id butlast.simps(2) last.simps list.distinct(1)*)

hence $ts'' = ts@[T-i32]$
using *b-e-type-value unlift-b-e[of $\mathcal{S} \ \mathcal{C} \ [C \ \text{ConstInt32 } k]$]*
unfolding *typeof-def*
by *fastforce*

hence $ts\text{-def}:ts' = ts@[t] \text{ load-store-t-bounds } a \ (\text{option-proj1 } tp) \ t$
using $ts''\text{-def}(2) \ \text{b-e-type-load unlift-b-e[of } \mathcal{S} \ \mathcal{C} \ [Load \ t \ tp \ a \ off]]$
by *fastforce+*

moreover
hence $\mathcal{C} \vdash [C \ v] : (ts \rightarrow ts@[t])$
using $assms(2) \ \text{b-e-typing.const b-e-typing.weakening}$
by *fastforce*

ultimately
show $\mathcal{S}\cdot\mathcal{C} \vdash [\$C \ v] : (ts \rightarrow ts')$
using *e-typing-s-typing.intros(1)*
by *fastforce*

qed

lemma *types-preserved-get-local:*

assumes $\mathcal{S}\cdot\mathcal{C} \vdash [\$Get\text{-local } i] : (ts \rightarrow ts')$
 $\text{length } vi = i$
 $(\text{local } \mathcal{C}) = \text{map } \text{typeof } (vi \ @ \ [v] \ @ \ vs)$
shows $\mathcal{S}\cdot\mathcal{C} \vdash [\$C \ v] : (ts \rightarrow ts')$

proof –

have $(\text{local } \mathcal{C})!i = \text{typeof } v$
using $assms(2,3)$
by (*metis (no-types, opaque-lifting) append-Cons length-map list.simps(9) map-append nth-append-length*)

hence $ts' = ts@[typeof \ v]$
using $assms(1) \ \text{unlift-b-e[of } \mathcal{S} \ \mathcal{C} \ [Get\text{-local } i]] \ \text{b-e-type-get-local}$
by *fastforce*

thus *?thesis*
using $\text{b-e-typing.const e-typing-s-typing.intros}(1,3)$
by *fastforce*

qed

lemma *types-preserved-set-local:*

assumes $\mathcal{S}\cdot\mathcal{C} \vdash [\$C \ v', \$Set\text{-local } i] : (ts \rightarrow ts')$
 $\text{length } vi = i$
 $(\text{local } \mathcal{C}) = \text{map } \text{typeof } (vi \ @ \ [v] \ @ \ vs)$
shows $(\mathcal{S}\cdot\mathcal{C} \vdash [] : (ts \rightarrow ts')) \wedge \text{map } \text{typeof } (vi \ @ \ [v] \ @ \ vs) = \text{map } \text{typeof } (vi \ @ \ [v'] \ @ \ vs)$

proof –

have $v\text{-type}:(\text{local } \mathcal{C})!i = \text{typeof } v$
using $assms(2,3)$

by (metis (no-types, opaque-lifting) append-Cons length-map list.simps(9) map-append nth-append-length)

obtain ts'' where $ts''\text{-def}:\mathcal{S}\cdot\mathcal{C} \vdash [\$C v^\wedge] : (ts \rightarrow ts')$
 $\mathcal{S}\cdot\mathcal{C} \vdash [\$Set\text{-local } i] : (ts'' \rightarrow ts')$

using *e-type-comp assms*

by (metis append-butlast-last-id butlast.simps(2) last.simps list.distinct(1))

hence $ts'' = ts@[typeof v^\wedge]$

using *b-e-type-value unlift-b-e[of $\mathcal{S} \mathcal{C} [C v^\wedge]$]*

by *fastforce*

hence $typeof v = typeof v' \ ts' = ts$

using *v-type b-e-type-set-local[of $\mathcal{C} \ Set\text{-local } i \ ts'' \ ts^\wedge \ ts''\text{-def}(2) \ unlift\text{-b-e[of } \mathcal{S} \mathcal{C} [Set\text{-local } i]]$]*

by *fastforce+*

thus *?thesis*

using *b-e-type-empty[of $\mathcal{C} \ ts \ ts^\wedge \ e\text{-typing-s-typing.intros}(1)$]*

by *fastforce*

qed

lemma *types-preserved-get-global:*

assumes $typeof (sglob\text{-val } s \ i \ j) = tg\text{-t } (global \ \mathcal{C} \ ! \ j)$

$\mathcal{S}\cdot\mathcal{C} \vdash [\$Get\text{-global } j] : (ts \rightarrow ts')$

shows $\mathcal{S}\cdot\mathcal{C} \vdash [\$C \ sglob\text{-val } s \ i \ j] : (ts \rightarrow ts')$

proof –

have $ts' = ts@[tg\text{-t } (global \ \mathcal{C} \ ! \ j)]$

using *b-e-type-get-global assms(2) unlift-b-e[of - - [Get-global j]]*

by *fastforce*

thus *?thesis*

using *b-e-typing.const[of $\mathcal{C} \ sglob\text{-val } s \ i \ j \ assms(1) \ e\text{-typing-s-typing.intros}(1,3)$]*

by *fastforce*

qed

lemma *lholed-same-type:*

assumes $Lfilled \ k \ lholed \ es \ les$

$Lfilled \ k \ lholed \ es' \ les'$

$\mathcal{S}\cdot\mathcal{C} \vdash les : (ts \rightarrow ts')$

$\bigwedge arb\text{-labs } ts \ ts'.$

$\mathcal{S}\cdot(\mathcal{C}(\backslash label := arb\text{-labs}@(\backslash label \ \mathcal{C}))) \vdash es : (ts \rightarrow ts')$

$\implies \mathcal{S}\cdot(\mathcal{C}(\backslash label := arb\text{-labs}@(\backslash label \ \mathcal{C}))) \vdash es' : (ts \rightarrow ts')$

shows $(\mathcal{S}\cdot\mathcal{C} \vdash les' : (ts \rightarrow ts'))$

using *assms*

proof (*induction arbitrary: $ts \ ts' \ es' \ \mathcal{C} \ les'$ rule: $Lfilled.induct$*)

case (*L0 vs lholed $es' \ es \ ts \ ts' \ es''$*)

obtain $ts'' \ ts'''$ where $\mathcal{S}\cdot\mathcal{C} \vdash vs : (ts \rightarrow ts')$

$\mathcal{S}\cdot\mathcal{C} \vdash es : (ts'' \rightarrow ts''')$

$\mathcal{S}\cdot\mathcal{C} \vdash es' : (ts''' \rightarrow ts')$

using *e-type-comp-conc2 L0(4)*

by *blast*

moreover

hence $(\mathcal{S}\cdot\mathcal{C} \vdash es'' : (ts'' \rightarrow ts'''))$

```

    using L0(5)[of [] ts'' ts''']
    by fastforce
  ultimately
  have (S.C ⊢ vs @ es'' @ es' : (ts -> ts'))
    using e-type-comp-conc
    by fastforce
  thus ?case
    using L0(2,3) Lfilled.simps[of 0 lholed es'' les']
    by fastforce
next
  case (LN vs lholed n es' l es'' k es lfilledk t1s t2s es''' C les')
  obtain lfilledk' where l'-def:Lfilled k l es''' lfilledk' les' = vs @ [Label n es'
lfilledk'] @ es''
    using LN Lfilled.simps[of k+1 lholed es''' les']
    by fastforce
  obtain ts' ts'' where lab-def:S.C ⊢ vs : (t1s -> ts')
    S.C ⊢ [Label n es' lfilledk] : (ts' -> ts'')
    S.C ⊢ es'' : (ts'' -> t2s)
  using e-type-comp-conc2[OF LN(6)]
  by blast
  obtain tls ts-c C-int where int-def: ts'' = ts' @ ts-c
    length tls = n
    S.C ⊢ es' : (tls -> ts-c)
    C-int = C(label := [tls] @ label C)
    S.C-int ⊢ lfilledk : ([] -> ts-c)
  using e-type-label[OF lab-def(2)]
  by blast
  have (∧ C' arb-labs' ts ts'.
    C' = C-int(label := arb-labs' @ label C-int) ⇒
    S.C' ⊢ es : (ts -> ts') ⇒
    (S.C' ⊢ es''' : (ts -> ts')))
  proof -
    fix C'' arb-labs'' tts tts'
    assume C'' = C-int(label := arb-labs'' @ label C-int)
      S.C'' ⊢ es : (tts -> tts')
    thus (S.C'' ⊢ es''' : (tts -> tts'))
      using LN(7)[of arb-labs'' @ [tls] tts tts'] int-def(4)
      by fastforce
  qed
  hence (S.C-int ⊢ lfilledk' : ([] -> ts-c))
    using LN(4)[OF l'-def(1) int-def(5)]
    by fastforce
  hence (S.C ⊢ [Label n es' lfilledk'] : (ts' -> ts''))
    using int-def e-typing-s-typing.intros(3,7)
    by (metis append.right-neutral)
  thus ?case
    using lab-def e-type-comp-conc l'-def(2)
    by blast
qed

```

```

lemma types-preserved-e1:
  assumes  $(\!|s;vs;es\!) \rightsquigarrow\!-\!i (\!|s';vs';es'\!)$ 
    store-typing  $s \mathcal{S}$ 
     $tvs = \text{map } \text{typeof } vs$ 
     $i < \text{length } (\text{inst } s)$ 
     $C = ((s\text{-inst } \mathcal{S})!i)(\text{local} := (\text{local } ((s\text{-inst } \mathcal{S})!i) @ tvs), \text{label} := \text{arb-labs},$ 
  return := arb-return)
     $\mathcal{S}\cdot C \vdash es : (ts \rightarrow ts')$ 
  shows  $(\mathcal{S}\cdot C \vdash es' : (ts \rightarrow ts')) \wedge (\text{map } \text{typeof } vs = \text{map } \text{typeof } vs')$ 
  using assms
proof (induction arbitrary: tvs C ts ts' arb-labs arb-return rule: reduce.induct)
  case (basic e e' s vs i)
  then show ?case
    using types-preserved-b-e1 [OF basic(1,2)]
    by fastforce
next
  case (call s vs j i)
  obtain  $ts'' \text{ tf1 } \text{ tf2}$  where l-func-t:  $\text{length } (\text{func-t } C) > j$ 
     $ts = ts'' @ \text{tf1}$ 
     $ts' = ts'' @ \text{tf2}$ 
     $((\text{func-t } C)!j) = (\text{tf1} \rightarrow \text{tf2})$ 
  using b-e-type-call[of C Call j ts ts' j] call(5)
    unlift-b-e[of - - [Call j] (ts -> ts')]
  by fastforce
  have  $i < \text{length } (s\text{-inst } \mathcal{S})$ 
  using call(3) store-typing-imp-inst-length-eq[OF call(1)]
  by simp
  moreover
  have  $j < \text{length } (\text{func-t } (s\text{-inst } \mathcal{S} ! i))$ 
  using l-func-t(1) call(4)
  by simp
  ultimately
  have cl-typing  $\mathcal{S} (\text{sfunc } s \ i \ j) (\text{tf1} \rightarrow \text{tf2})$ 
  using store-typing-imp-func-agree[OF call(1)] l-func-t(4) call(4)
  by fastforce
  thus ?case
  using e-typing-s-typing.intros(3,6) l-func-t
  by fastforce
next
  case (call-indirect-Some s i' c cl j tf vs)
  show ?case
  using types-preserved-call-indirect-Some[OF call-indirect-Some(8,1)]
    call-indirect-Some(2,3,4,6,7)
  by fastforce
next
  case (call-indirect-None s i c cl j vs)
  thus ?case
  using e-typing-s-typing.intros(4)

```

```

    by blast
next
case (callcl-native cl j t1s t2s ts es ves vcs n k m zs s vs i)
thus ?case
  using types-preserved-callcl-native
  by fastforce
next
case (callcl-host-Some cl t1s t2s f ves vcs n m s hs s' vcs' vs i)
thus ?case
  using types-preserved-callcl-host-some
  by fastforce
next
case (callcl-host-None cl t1s t2s f ves vcs n m s hs vs i)
thus ?case
  using e-typing-s-typing.intros(4)
  by blast
next
case (get-local vi j s v vs i)
hence i < length (s-inst S)
  unfolding list-all2-conv-all-nth store-typing.simps
  by fastforce
then have local C = tvs
  using store-local-label-empty assms(2) get-local
  by fastforce
then show ?case
  using types-preserved-get-local get-local
  by fastforce
next
case (set-local vi j s v vs v' i)
hence i < length (s-inst S)
  unfolding list-all2-conv-all-nth store-typing.simps
  by fastforce
hence local C = tvs
  using store-local-label-empty assms(2) set-local
  by fastforce
thus ?case
  using set-local types-preserved-set-local
  by simp
next
case (get-global s vs j i)
have length (global C) > j
  using b-e-type-get-global get-global(5) unlift-b-e[of - - [Get-global j] (ts -> ts')]
  by fastforce
hence glob-agree (sglob s i j) ((global C)!j)
  using get-global(3,4) store-typing-imp-glob-agree[OF get-global(1)] store-typing-imp-inst-length-eq[OF
get-global(1)]
  by fastforce
hence typeof (g-val (sglob s i j)) = tg-t (global C ! j)
  unfolding glob-agree-def

```

```

    by simp
  thus ?case
    using get-global(5) types-preserved-get-global
    unfolding glob-agree-def sglob-val-def
    by fastforce
next
  case (set-global s i j v s' vs)
  then show ?case
    using types-preserved-set-global
    by fastforce
next
  case (load-Some s i j m k off t bs vs a)
  then show ?case
    using types-preserved-load(1) wasm-deserialise-type
    by blast
next
  case (load-None s i j m k off t vs a)
  then show ?case
    using e-typing-s-typing.intros(4)
    by blast
next
  case (load-packed-Some s i j m sx k off tp bs vs t a)
  then show ?case
    using types-preserved-load(1) wasm-deserialise-type
    by blast
next
  case (load-packed-None s i j m sx k off tp vs t a)
  then show ?case
    using e-typing-s-typing.intros(4)
    by blast
next
  case (store-Some t v s i j m k off mem' vs a)
  then show ?case
    using types-preserved-store
    by blast
next
  case (store-None t v s i j m k off vs a)
  then show ?case
    using e-typing-s-typing.intros(4)
    by blast
next
  case (store-packed-Some t v s i j m k off tp mem' vs a)
  then show ?case
    using types-preserved-store
    by blast
next
  case (store-packed-None t v s i j m k off tp vs a)
  then show ?case
    using e-typing-s-typing.intros(4)

```

```

    by blast
next
case (current-memory s i j m n vs)
then show ?case
  using types-preserved-current-memory
  by fastforce
next
case (grow-memory s i j m n c mem' vs)
then show ?case
  using types-preserved-grow-memory
  by fastforce
next
case (grow-memory-fail s i j m n vs c)
thus ?case
  using types-preserved-grow-memory
  by blast
next
case (label s vs es i s' vs' es' k lholed les les')
{
  fix C' arb-labs' ts ts'
  assume local-assms:C' = C(|label := arb-labs'@(label C), return := (return C)|)
  hence (S.C' ⊢ es : (ts -> ts')) ⇒ (S.C' ⊢ es' : (ts -> ts')) ∧ map typeof vs =
map typeof vs'
  using label(4)[OF label(5,6,7)] label(8)
  by fastforce
  hence (S.C(|label := arb-labs'@(label C)|) ⊢ es : (ts -> ts'))
    ⇒ (S.C(|label := arb-labs'@(label C)|) ⊢ es' : (ts -> ts')) ∧
    map typeof vs = map typeof vs'
  using local-assms
  by simp
}
hence ∧arb-labs' ts ts'. S.C(|label := arb-labs'@(label C)|) ⊢ es : (ts -> ts')
  ⇒ (S.C(|label := arb-labs'@(label C)|) ⊢ es' : (ts -> ts'))
  map typeof vs = map typeof vs'
  using types-exist-lfilled[OF label(2,9)]
  by auto
thus ?case
  using lholed-same-type[OF label(2,3,9)]
  by fastforce
next
case (local s vls es i s' vs' es' vs n j)
obtain C' tls where es-def:i < length (s-inst S)
  length tls = n
  C' = (s-inst S ! i) (|local := local(s-inst S ! i) @ map typeof
vls, label := label (s-inst S ! i), return := Some tls)
  S.C' ⊢ es : ([] -> tls)
  ts' = ts @ tls
  using e-type-local[OF local(7)]
  by fastforce

```

moreover
obtain ts'' **where** $ts' = ts@ts''$ ($\mathcal{S} \cdot (\text{Some } ts'') \Vdash -i \text{ vls}; es : ts''$)
using *e-type-local-shallow local(7)*
by *fastforce*
moreover
have *inst-typing* $\mathcal{S} ((\text{inst } s)!i) ((s\text{-inst } \mathcal{S})!i) i < \text{length } (\text{inst } s)$
using *local es-def(1)*
unfolding *store-typing.simps list-all2-conv-all-nth*
by *fastforce+*
ultimately
have $\mathcal{S} \cdot \mathcal{C}' \vdash es' : ([\] \rightarrow \text{tls}) \text{ map typeof vls} = \text{map typeof vs}'$
using *local(2)[OF local(3) - - - es-def(4), of map typeof vls Some tls label*
(s-inst S ! i)]
by *fastforce+*
hence $\mathcal{S} \cdot (\text{Some tls}) \Vdash -i \text{ vs}'; es' : \text{tls}$
using *e-typing-s-typing.intros(8) es-def(1,3)*
by *fastforce*
thus *?case*
using *e-typing-s-typing.intros(3,5) es-def(2,5)*
by *fastforce*
qed

lemma *types-preserved-e:*
assumes $(s; vs; es) \rightsquigarrow -i (s'; vs'; es')$
store-typing s S
 $\mathcal{S} \cdot \text{None} \Vdash -i \text{ vs}; es : ts$
shows $\mathcal{S} \cdot \text{None} \Vdash -i \text{ vs}'; es' : ts$
using *assms*
proof –
have $i < (\text{length } (s\text{-inst } \mathcal{S}))$
using *assms(3) s-typing.cases*
by *blast*
moreover
hence *i-bound: i < length (inst s)*
using *assms(2)*
unfolding *list-all2-conv-all-nth store-typing.simps*
by *fastforce*
obtain $\text{tvs } \mathcal{C}$ **where** *defs: tvs = map typeof vs*
 $\mathcal{C} = ((s\text{-inst } \mathcal{S})!i)(\text{local} := (\text{local } ((s\text{-inst } \mathcal{S})!i) @ \text{tvs}), \text{label}$
 $:= (\text{label } ((s\text{-inst } \mathcal{S})!i), \text{return} := \text{None}))$
 $\mathcal{S} \cdot \mathcal{C} \vdash es : ([\] \rightarrow ts)$
using *assms(3)*
unfolding *s-typing.simps*
by *fastforce*
have $(\mathcal{S} \cdot \mathcal{C} \vdash es' : ([\] \rightarrow ts)) \wedge (\text{map typeof vs} = \text{map typeof vs}')$
using *types-preserved-e1[OF assms(1,2) defs(1) i-bound defs(2,3)]*
by *simp*
ultimately show *?thesis*
using *defs*

```

    unfolding s-typing.simps
  by auto
qed

```

6.2 Progress

```

lemma const-list-no-progress:
  assumes const-list es
  shows  $\neg(|s;vs;es| \rightsquigarrow - i (|s';vs';es'|))$ 
proof -
  {
    assume  $(|s;vs;es| \rightsquigarrow - i (|s';vs';es'|))$ 
    hence False
    using assms
  proof (induction rule: reduce.induct)
    case (basic e e' s vs i)
    thus ?thesis
  proof (induction rule: reduce-simple.induct)
    case (trap es lholed)
    show ?case
      using trap(2)
    proof (cases rule: Lfilled.cases)
      case (L0 vs es')
      thus ?thesis
        using trap(3) list-all-append const-list-cons-last(2)[of vs Trap]
        unfolding const-list-def
        by (simp add: is-const-def)
    next
      case (LN vs n es' l es'' k lfilledk)
      thus ?thesis
        by (simp add: is-const-def)
    qed
  qed (fastforce simp add: const-list-cons-last(2) is-const-def const-list-def)+
  next
  case (label s vs es i s' vs' es' k lholed les les')
  show ?case
    using label(2)
  proof (cases rule: Lfilled.cases)
    case (L0 vs es')
    thus ?thesis
      using label(4,5) list-all-append
      unfolding const-list-def
      by fastforce
  next
  case (LN vs n es' l es'' k lfilledk)
  thus ?thesis
    using label(4,5)
    unfolding const-list-def
    by (simp add: is-const-def)

```

```

    qed
  qed (fastforce simp add: const-list-cons-last(2) is-const-def const-list-def)+
}
thus ?thesis
  by blast
qed

```

```

lemma empty-no-progress:
  assumes es = []
  shows  $\neg(|s;vs;es| \rightsquigarrow\!-\! i(|s';vs';es'|))$ 
proof -
  {
    assume  $(|s;vs;es| \rightsquigarrow\!-\! i(|s';vs';es'|))$ 
    hence False
      using assms
    proof (induction rule: reduce.induct)
      case (basic e e' s vs i)
      thus ?thesis
    proof (induction rule: reduce-simple.induct)
      case (trap es lholed)
      thus ?case
        using Lfilled.simps[of 0 lholed [Trap] es]
        by auto
      qed auto
    next
      case (label s vs es i s' vs' es' k lholed les les')
      thus ?case
        using Lfilled.simps[of k lholed es []]
        by auto
      qed auto
    }
  thus ?thesis
    by blast
qed

```

```

lemma trap-no-progress:
  assumes es = [Trap]
  shows  $\neg(|s;vs;es| \rightsquigarrow\!-\! i(|s';vs';es'|))$ 
proof -
  {
    assume  $(|s;vs;es| \rightsquigarrow\!-\! i(|s';vs';es'|))$ 
    hence False
      using assms
    proof (induction rule: reduce.induct)
      case (basic e e' s vs i)
      thus ?case
        by (induction rule: reduce-simple.induct) auto
    next
      case (label s vs es i s' vs' es' k lholed les les')

```

```

show ?case
  using label(2)
  proof (cases rule: Lfilled.cases)
    case (L0 vs es')
      show ?thesis
        using L0(2) label(1,4,5) empty-no-progress
        by (auto simp add: Cons-eq-append-conv)
      next
        case (LN vs n es' l es'' k' lfilledk)
          show ?thesis
            using LN(2) label(5)
            by (simp add: Cons-eq-append-conv)
        qed
      qed auto
    }
  thus ?thesis
    by blast
qed

```

```

lemma terminal-no-progress:
  assumes const-list es  $\vee$  es = [Trap]
  shows  $\neg$ (s;vs;es)  $\rightsquigarrow$ - i (s';vs';es')
  using const-list-no-progress trap-no-progress assms
  by blast

```

```

lemma progress-L0:
  assumes (s;vs;es)  $\rightsquigarrow$ - i (s';vs';es')
    const-list cs
  shows (s;vs;cs@es@es-c)  $\rightsquigarrow$ - i (s';vs';cs@es'@es-c)
proof -
  have  $\bigwedge$ es. Lfilled 0 (LBase cs es-c) es (cs@es@es-c)
    using Lfilled.intros(1)[of cs (LBase cs es-c) es-c] assms(2)
    unfolding const-list-def
    by fastforce
  thus ?thesis
    using reduce.intros(23) assms(1)
    by blast
qed

```

```

lemma progress-L0-left:
  assumes (s;vs;es)  $\rightsquigarrow$ - i (s';vs';es')
    const-list cs
  shows (s;vs;cs@es)  $\rightsquigarrow$ - i (s';vs';cs@es')
  using assms progress-L0[where ?es-c = []]
  by fastforce

```

```

lemma progress-L0-trap:
  assumes const-list cs
    cs  $\neq$  []  $\vee$  es  $\neq$  []

```

shows $\exists a. (\{s; vs; cs @ [Trap] @ es\} \rightsquigarrow - i (\{s; vs; [Trap]\})$
proof –
have $cs @ [Trap] @ es \neq [Trap]$
using $assms(2)$
by $(cases\ cs = [])\ (auto\ simp\ add:\ append\ eq\ Cons\ conv)$
thus $?thesis$
using $reduce.intros(1)\ assms(2)\ reduce.simple.trap$
 $Lfilled.intros(1)[OF\ assms(1),\ of\ -\ es\ [Trap]]$
by $blast$
qed

lemma $progress-LN$:
assumes $(Lfilled\ j\ lholed\ [\$Br\ (j+k)]\ es)$
 $\mathcal{S}\cdot\mathcal{C} \vdash es : ([\] \rightarrow ts)$
 $(label\ C)!k = tvs$
shows $\exists\ lholed'\ vs\ \mathcal{C}'.\ (Lfilled\ j\ lholed'\ (vs @ [\$Br\ (j+k)])\ es)$
 $\wedge (\mathcal{S}\cdot\mathcal{C}' \vdash vs : ([\] \rightarrow tvs))$
 $\wedge\ const\ list\ vs$
using $assms$
proof $(induction\ [\$Br\ (j+k)]\ es\ arbitrary:\ k\ C\ ts\ rule:\ Lfilled.induct)$
case $(L0\ vs\ lholed\ es')$
obtain $ts'\ ts''$ **where** $ts\text{-}def:\mathcal{S}\cdot\mathcal{C} \vdash vs : ([\] \rightarrow ts')$
 $\mathcal{S}\cdot\mathcal{C} \vdash [\$Br\ k] : (ts' \rightarrow ts'')$
 $\mathcal{S}\cdot\mathcal{C} \vdash es' : (ts'' \rightarrow ts)$
using $e\text{-}type\text{-}comp\text{-}conc2[OF\ L0(3)]$
by $fastforce$
obtain $ts\text{-}c$ **where** $ts' = ts\text{-}c @ tvs$
using $b\text{-}e\text{-}type\text{-}br[of\ C\ Br\ k\ ts'\ ts'']\ L0(3,4)\ ts\text{-}def(2)\ unlift\ b\text{-}e$
by $fastforce$
then obtain $vs1\ vs2$ **where** $vs\text{-}def:\mathcal{S}\cdot\mathcal{C} \vdash vs1 : ([\] \rightarrow ts\text{-}c)$
 $\mathcal{S}\cdot\mathcal{C} \vdash vs2 : (ts\text{-}c \rightarrow (ts\text{-}c @ tvs))$
 $vs = vs1 @ vs2$
 $const\ list\ vs1$
 $const\ list\ vs2$
using $e\text{-}type\text{-}const\ list\ cons[OF\ L0(1)]\ ts\text{-}def(1)$
by $fastforce$
hence $\mathcal{S}\cdot\mathcal{C} \vdash vs2 : ([\] \rightarrow tvs)$
using $e\text{-}type\text{-}const\ list$ **by** $blast$
thus $?case$
using $Lfilled.intros(1)[OF\ vs\text{-}def(4),\ of\ -\ es'\ vs2 @ [\$Br\ k]]\ vs\text{-}def(3,5)$
by $fastforce$

next
case $(LN\ vs\ lholed\ n\ es'\ l\ es''\ j\ lfilledk)$
obtain $t1s\ t2s$ **where** $ts\text{-}def:\mathcal{S}\cdot\mathcal{C} \vdash vs : ([\] \rightarrow t1s)$
 $\mathcal{S}\cdot\mathcal{C} \vdash [Label\ n\ es'\ lfilledk] : (t1s \rightarrow t2s)$
 $\mathcal{S}\cdot\mathcal{C} \vdash es'' : (t2s \rightarrow ts)$
using $e\text{-}type\text{-}comp\text{-}conc2[OF\ LN(5)]$
by $fastforce$
obtain $ts'\ ts\text{-}l$ **where** $ts\text{-}l\text{-}def:\mathcal{S}\cdot\mathcal{C} (label := [ts'] @ label\ C) \vdash lfilledk : ([\] \rightarrow ts\text{-}l)$

using *e-type-label*[*OF ts-def*(2)]
by *fastforce*
obtain *lholed' vs' C' where lfilledk-def:Lfilled j lholed' (vs' @ [Br (j + (1 + k))]) lfilledk*

$$\mathcal{S}\cdot\mathcal{C}' \vdash vs' : ([\] \rightarrow tvs)$$

$$const\text{-list } vs'$$
using *LN*(4)[*OF - ts-l-def, of 1 + k*] *LN*(5,6)
by *fastforce*
thus *?case*
using *Lfilled.intros*(2)[*OF LN*(1) - *lfilledk-def*(1)]
by *fastforce*
qed

lemma *progress-LN-return*:
assumes (*Lfilled j lholed [Return] es*)

$$\mathcal{S}\cdot\mathcal{C} \vdash es : ([\] \rightarrow ts)$$

$$(return \mathcal{C}) = Some \ tvs$$
shows $\exists \ lholed' \ vs \ \mathcal{C}'. (Lfilled \ j \ lholed' \ (vs@[Return]) \ es)$

$$\wedge (\mathcal{S}\cdot\mathcal{C}' \vdash vs : ([\] \rightarrow tvs))$$

$$\wedge const\text{-list } vs$$
using *assms*
proof (*induction [Return] es arbitrary: k C ts rule: Lfilled.induct*)
case (*L0 vs lholed es'*)
obtain *ts' ts'' where ts-def:S.C \vdash vs : ([\] \rightarrow ts')*

$$\mathcal{S}\cdot\mathcal{C} \vdash [Return] : (ts' \rightarrow ts'')$$

$$\mathcal{S}\cdot\mathcal{C} \vdash es' : (ts'' \rightarrow ts)$$
using *e-type-comp-conc2*[*OF L0*(3)]
by *fastforce*
obtain *ts-c where ts' = ts-c @ tvs*
using *b-e-type-return*[*of C Return ts' ts''*] *L0*(3,4) *ts-def*(2) *unlift-b-e*
by *fastforce*
then obtain *vs1 vs2 where vs-def:S.C \vdash vs1 : ([\] \rightarrow ts-c)*

$$\mathcal{S}\cdot\mathcal{C} \vdash vs2 : (ts-c \rightarrow (ts-c@tvs))$$

$$vs = vs1@vs2$$

$$const\text{-list } vs1$$

$$const\text{-list } vs2$$
using *e-type-const-list-cons*[*OF L0*(1)] *ts-def*(1)
by *fastforce*
hence $\mathcal{S}\cdot\mathcal{C} \vdash vs2 : ([\] \rightarrow tvs)$
using *e-type-const-list* **by** *blast*
thus *?case*
using *Lfilled.intros*(1)[*OF vs-def*(4), *of - es' vs2@[Return]*] *vs-def*(3,5)
by *fastforce*
next
case (*LN vs lholed n es' l es'' j lfilledk*)
obtain *t1s t2s where ts-def:S.C \vdash vs : ([\] \rightarrow t1s)*

$$\mathcal{S}\cdot\mathcal{C} \vdash [Label \ n \ es' \ lfilledk] : (t1s \rightarrow t2s)$$

$$\mathcal{S}\cdot\mathcal{C} \vdash es'' : (t2s \rightarrow ts)$$
using *e-type-comp-conc2*[*OF LN*(5)]

by *fastforce*
obtain $ts' \text{ } ts\text{-}l$ **where** $ts\text{-}l\text{-}def : \mathcal{S} \cdot \mathcal{C} (\text{label} := [ts'] @ \text{label } \mathcal{C}) \vdash \text{lfilled}k : ([\] \text{ } \rightarrow \text{ts}\text{-}l)$
using $e\text{-type}\text{-label}$ [OF $ts\text{-}def(2)$]
by *fastforce*
obtain $\text{lholed}' \text{ } vs' \text{ } \mathcal{C}'$ **where** $\text{lfilled}k\text{-}def : \text{Lfilled } j \text{ } \text{lholed}' (vs' @ [\text{Return}]) \text{ } \text{lfilled}k$
 $\mathcal{S} \cdot \mathcal{C}' \vdash vs' : ([\] \text{ } \rightarrow \text{tus})$
 $\text{const}\text{-list } vs'$
using $LN(4)$ [OF $ts\text{-}l\text{-}def$] $LN(6)$
by *fastforce*
thus *?case*
using $\text{Lfilled.intros}(2)$ [OF $LN(1)$ - $\text{lfilled}k\text{-}def(1)$]
by *fastforce*
qed

lemma *progress-LN1*:
assumes $(\text{Lfilled } j \text{ } \text{lholed } [\text{\$Br } (j+k)] \text{ } es)$
 $\mathcal{S} \cdot \mathcal{C} \vdash es : (ts \text{ } \rightarrow \text{ts}')$
shows $\text{length } (\text{label } \mathcal{C}) > k$
using *assms*
proof (*induction* $[\text{\$Br } (j+k)] \text{ } es$ *arbitrary*: $k \text{ } \mathcal{C} \text{ } ts \text{ } ts'$ *rule*: Lfilled.induct)
case $(L0 \text{ } vs \text{ } \text{lholed } es')$
obtain $ts'' \text{ } ts'''$ **where** $ts\text{-}def : \mathcal{S} \cdot \mathcal{C} \vdash vs : (ts \text{ } \rightarrow \text{ts}'')$
 $\mathcal{S} \cdot \mathcal{C} \vdash [\text{\$Br } k] : (ts'' \text{ } \rightarrow \text{ts}''')$
 $\mathcal{S} \cdot \mathcal{C} \vdash es' : (ts''' \text{ } \rightarrow \text{ts}')$
using $e\text{-type}\text{-comp}\text{-conc}2$ [OF $L0(3)$]
by *fastforce*
thus *?case*
using $b\text{-e}\text{-type}\text{-br}(1)$ [of - $\text{Br } k \text{ } ts'' \text{ } ts'''$] *unlift-b-e*
by *fastforce*

next
case $(LN \text{ } vs \text{ } \text{lholed } n \text{ } es' \text{ } l \text{ } es'' \text{ } k' \text{ } \text{lfilled}k)$
obtain $t1s \text{ } t2s$ **where** $ts\text{-}def : \mathcal{S} \cdot \mathcal{C} \vdash vs : (ts \text{ } \rightarrow \text{t1s})$
 $\mathcal{S} \cdot \mathcal{C} \vdash [\text{Label } n \text{ } es' \text{ } \text{lfilled}k] : (t1s \text{ } \rightarrow \text{t2s})$
 $\mathcal{S} \cdot \mathcal{C} \vdash es'' : (t2s \text{ } \rightarrow \text{ts}')$
using $e\text{-type}\text{-comp}\text{-conc}2$ [OF $LN(5)$]
by *fastforce*
obtain $ts'' \text{ } ts\text{-}l$ **where** $ts\text{-}l\text{-}def : \mathcal{S} \cdot \mathcal{C} (\text{label} := [ts''] @ \text{label } \mathcal{C}) \vdash \text{lfilled}k : ([\] \text{ } \rightarrow \text{ts}\text{-}l)$
using $e\text{-type}\text{-label}$ [OF $ts\text{-}def(2)$]
by *fastforce*
thus *?case*
using $LN(4)$ [of $1+k$]
by *fastforce*
qed

lemma *progress-LN2*:
assumes $(\text{Lfilled } j \text{ } \text{lholed } e1s \text{ } \text{lfilled})$
shows $\exists \text{lfilled}' . (\text{Lfilled } j \text{ } \text{lholed } e2s \text{ } \text{lfilled}')$
using *assms*
proof (*induction rule*: Lfilled.induct)

```

case (L0 vs lholed es' es)
thus ?case
  using Lfilled.intros(1)
  by fastforce
next
case (LN vs lholed n es' l es'' k es lfilledk)
thus ?case
  using Lfilled.intros(2)
  by fastforce
qed

```

```

lemma const-of-const-list:
  assumes length cs = 1
    const-list cs
  shows  $\exists v. cs = [\$C v]$ 
  using e-type-const-unwrap assms
  unfolding const-list-def list-all-length
  by (metis append-butlast-last-id append-self-conv2 gr-zeroI last-conv-nth length-butlast
    length-greater-0-conv less-numeral-extra(1,4) zero-less-diff)

```

```

lemma const-of-i32:
  assumes const-list cs
     $\mathcal{S}\cdot\mathcal{C} \vdash cs : ([\ ] \rightarrow [(T-i32)])$ 
  shows  $\exists c. cs = [\$C \text{ConstInt32 } c]$ 
proof –
  obtain v where cs = [\$C v]
    using const-of-const-list assms(1) e-type-const-list[OF assms]
    by fastforce
  moreover
  hence  $\mathcal{C} \vdash [C v] : ([\ ] \rightarrow [(T-i32)])$ 
    using assms(2) unlift-b-e
    by fastforce
  hence  $\exists c. v = \text{ConstInt32 } c$ 
  proof (induction [C v] ([\ ] \rightarrow [(T-i32)]) rule: b-e-typing.induct)
    case (const C)
    then show ?case
      unfolding typeof-def
      by (cases v, auto)
  qed auto
  ultimately
  show ?thesis
    by fastforce
qed

```

```

lemma const-of-i64:
  assumes const-list cs
     $\mathcal{S}\cdot\mathcal{C} \vdash cs : ([\ ] \rightarrow [(T-i64)])$ 
  shows  $\exists c. cs = [\$C \text{ConstInt64 } c]$ 
proof –

```

```

obtain  $v$  where  $cs = [\$C v]$ 
  using const-of-const-list  $assms(1)$  e-type-const-list [ $OF\ assms$ ]
  by fastforce
moreover
hence  $\mathcal{C} \vdash [C v] : ([\ ] \rightarrow [(T-i64)])$ 
  using  $assms(2)$  unlift-b-e
  by fastforce
hence  $\exists c. v = ConstInt64\ c$ 
proof (induction  $[C v]$   $([\ ] \rightarrow [(T-i64)])$ ) rule: b-e-typing.induct
  case (const  $\mathcal{C}$ )
  then show ?case
    unfolding typeof-def
    by (cases  $v$ , auto)
qed auto
ultimately
show ?thesis
  by fastforce
qed

```

```

lemma const-of-f32:
  assumes const-list  $cs$ 
     $\mathcal{S}\cdot\mathcal{C} \vdash cs : ([\ ] \rightarrow [T-f32])$ 
  shows  $\exists c. cs = [\$C\ ConstFloat32\ c]$ 
proof –
  obtain  $v$  where  $cs = [\$C v]$ 
    using const-of-const-list  $assms(1)$  e-type-const-list [ $OF\ assms$ ]
    by fastforce
  moreover
hence  $\mathcal{C} \vdash [C v] : ([\ ] \rightarrow [T-f32])$ 
    using  $assms(2)$  unlift-b-e
    by fastforce
hence  $\exists c. v = ConstFloat32\ c$ 
proof (induction  $[C v]$   $([\ ] \rightarrow [T-f32])$ ) rule: b-e-typing.induct
  case (const  $\mathcal{C}$ )
  then show ?case
    unfolding typeof-def
    by (cases  $v$ , auto)
qed auto
ultimately
show ?thesis
  by fastforce
qed

```

```

lemma const-of-f64:
  assumes const-list  $cs$ 
     $\mathcal{S}\cdot\mathcal{C} \vdash cs : ([\ ] \rightarrow [T-f64])$ 
  shows  $\exists c. cs = [\$C\ ConstFloat64\ c]$ 
proof –
  obtain  $v$  where  $cs = [\$C v]$ 

```

```

    using const-of-const-list assms(1) e-type-const-list[OF assms]
    by fastforce
  moreover
  hence  $\mathcal{C} \vdash [C v] : ([\ ] \rightarrow [T-f64])$ 
    using assms(2) unlift-b-e
    by fastforce
  hence  $\exists c. v = \text{ConstFloat64 } c$ 
  proof (induction [C v] ([\ ]  $\rightarrow$  [T-f64]) rule: b-e-typing.induct)
    case (const C)
    then show ?case
      unfolding typeof-def
      by (cases v, auto)
  qed auto
  ultimately
  show ?thesis
    by fastforce
qed

lemma progress-unop-testop-i:
  assumes  $\mathcal{S}\cdot\mathcal{C} \vdash cs : ([\ ] \rightarrow [t])$ 
    is-int-t t
    const-list cs
     $e = \text{Unop-i } t \text{ iop} \vee e = \text{Testop } t \text{ testop}$ 
  shows  $\exists a s' vs' es'. (s;vs;cs@[e]) \rightsquigarrow\text{-i } (s';vs';es')$ 
  using assms(2)
  proof (cases t)
    case T-i32
    thus ?thesis
      using const-of-i32[OF assms(3)] assms(1,4)
        reduce.intros(1)[OF reduce-simple.intros(1)] reduce.intros(1)[OF re-
duce-simple.intros(13)]
      by fastforce
  next
  case T-i64
  thus ?thesis
    using const-of-i64[OF assms(3)] assms(1,4)
      reduce.intros(1)[OF reduce-simple.intros(2)] reduce.intros(1)[OF re-
duce-simple.intros(14)]
    by fastforce
  qed (simp-all add: is-int-t-def)

lemma progress-unop-f:
  assumes  $\mathcal{S}\cdot\mathcal{C} \vdash cs : ([\ ] \rightarrow [t])$ 
    is-float-t t
    const-list cs
     $e = \text{Unop-f } t \text{ iop}$ 
  shows  $\exists a s' vs' es'. (s;vs;cs@[e]) \rightsquigarrow\text{-i } (s';vs';es')$ 
  using assms(2)
  proof (cases t)

```

```

case T-f32
thus ?thesis
  using const-of-f32[OF assms(3)] assms(1,4)
    reduce.intros(1)[OF reduce-simple.intros(3)] reduce.intros(1)[OF re-
duce-simple.intros(13)]
  by fastforce
next
case T-f64
thus ?thesis
  using const-of-f64[OF assms(3)] assms(1,4)
    reduce.intros(1)[OF reduce-simple.intros(4)] reduce.intros(1)[OF re-
duce-simple.intros(14)]
  by fastforce
qed (simp-all add: is-float-t-def)

```

lemma *const-list-split-2*:

```

assumes const-list cs
   $\mathcal{S}\cdot\mathcal{C} \vdash cs : ([\ ] \rightarrow [t1, t2])$ 
shows  $\exists c1\ c2. (\mathcal{S}\cdot\mathcal{C} \vdash [c1] : ([\ ] \rightarrow [t1]))$ 
   $\wedge (\mathcal{S}\cdot\mathcal{C} \vdash [c2] : ([\ ] \rightarrow [t2]))$ 
   $\wedge cs = [c1, c2]$ 
   $\wedge \text{const-list } [c1]$ 
   $\wedge \text{const-list } [c2]$ 

```

proof –

```

have l-cs:length cs = 2
  using assms e-type-const-list[OF assms]
  by simp
then obtain c1 c2 where cs!0 = c1 cs!1 = c2
  by fastforce
hence cs = [c1] @ [c2]
  using assms e-type-const-conv-vs typing-map-typeof
  by fastforce
thus ?thesis
  using assms e-type-comp[of  $\mathcal{S}\ \mathcal{C}$  [c1] c2] e-type-const[of c2  $\mathcal{S}\ \mathcal{C}$  - [t1,t2]]
  unfolding const-list-def
  by fastforce
qed

```

lemma *const-list-split-3*:

```

assumes const-list cs
   $\mathcal{S}\cdot\mathcal{C} \vdash cs : ([\ ] \rightarrow [t1, t2, t3])$ 
shows  $\exists c1\ c2\ c3. (\mathcal{S}\cdot\mathcal{C} \vdash [c1] : ([\ ] \rightarrow [t1]))$ 
   $\wedge (\mathcal{S}\cdot\mathcal{C} \vdash [c2] : ([\ ] \rightarrow [t2]))$ 
   $\wedge (\mathcal{S}\cdot\mathcal{C} \vdash [c3] : ([\ ] \rightarrow [t3]))$ 
   $\wedge cs = [c1, c2, c3]$ 

```

proof –

```

have l-cs:length cs = 3
  using assms e-type-const-list[OF assms]
  by simp

```

then obtain $c1\ c2\ c3$ **where** $cs!0 = c1\ cs!1 = c2\ cs!2 = c3$
 by *fastforce*
hence $cs = [c1] @ [c2] @ [c3]$
 using *assms e-type-const-conv-vs typing-map-typeof*
 by *fastforce*
thus *?thesis*
 using *assms e-type-comp-conc2[of S C [c1] [c2] [c3] [] [t1,t2,t3]*
 e-type-const[of c1] e-type-const[of c2] e-type-const[of c3]
 unfolding *const-list-def*
 by *fastforce*
qed

lemma *progress-binop-relop-i:*

assumes $\mathcal{S}\cdot\mathcal{C} \vdash cs : ([\] \rightarrow [t, t])$
 is-int-t t
 const-list cs
 $e = \text{Binop-i } t\ iop \vee e = \text{Relop-i } t\ irop$
shows $\exists a\ s'\ vs'\ es'. (s;vs;cs@[e]) \rightsquigarrow\text{-i } (s';vs';es')$
 using *assms(2)*
proof (*cases t*)
 case (*T-i32*)
hence *cs-def*: $\exists c1\ c2. cs = [\$C\ ConstInt32\ c1, \$C\ ConstInt32\ c2]$
 using *const-list-split-2[OF assms(3,1)] assms(3) const-of-i32*
 unfolding *const-list-def*
 by *blast*
show *?thesis*
proof (*cases e = Binop-i t iop*)
 case *True*
obtain $c1\ c2$ **where** $cs = [\$C\ ConstInt32\ c1, \$C\ ConstInt32\ c2]$
 using *cs-def*
 by *blast*
thus *?thesis*
 apply (*cases app-binop-i iop c1 c2*)
 apply (*metis reduce-simple.intros(6) reduce.intros(1) T-i32 True append-Cons*
append-Nil)
 apply (*metis reduce-simple.intros(5) reduce.intros(1) T-i32 True append-Cons*
append-Nil)
 done
next
 case *False*
thus *?thesis*
 using *reduce-simple.intros(15) assms(4) reduce.intros(1) cs-def T-i32*
 by *fastforce*
qed
next
 case (*T-i64*)
hence *cs-def*: $\exists c1\ c2. cs = [\$C\ ConstInt64\ c1, \$C\ ConstInt64\ c2]$
 using *const-list-split-2[OF assms(3,1)] assms(3) const-of-i64*
 unfolding *const-list-def*

```

  by blast
show ?thesis
proof (cases e = Binop-i t iop)
  case True
  obtain c1 c2 where cs = [ $\$C$  ConstInt64 c1, $\$C$  ConstInt64 c2]
  using cs-def
  by blast
  thus ?thesis
  apply (cases app-binop-i iop c1 c2)
  apply (metis reduce-simple.intros(8) reduce.intros(1) T-i64 True append-Cons
append-Nil)
  apply (metis reduce-simple.intros(7) reduce.intros(1) T-i64 True append-Cons
append-Nil)
  done
next
  case False
  thus ?thesis
  using reduce-simple.intros(16) assms(4) reduce.intros(1) cs-def T-i64
  by fastforce
qed
qed (simp-all add: is-int-t-def)

```

```

lemma progress-binop-relop-f:
  assumes  $\mathcal{S}\cdot\mathcal{C} \vdash cs : ([\ ] \rightarrow [t, t])$ 
    is-float-t t
    const-list cs
     $e = \text{Binop-f } t \text{ fop} \vee e = \text{Relop-f } t \text{ frop}$ 
  shows  $\exists a s' vs' es'. (s;vs;cs@[e]) \rightsquigarrow\text{-i } (s';vs';es')$ 
  using assms(2)
proof (cases t)
  case T-f32
  hence cs-def: $\exists c1 c2. cs = [\$C$  ConstFloat32 c1, $\$C$  ConstFloat32 c2]
  using const-list-split-2[OF assms(3,1)] assms(3) const-of-f32
  unfolding const-list-def
  by blast
  show ?thesis
  proof (cases e = Binop-f t fop)
    case True
    obtain c1 c2 where cs-def:cs = [ $\$C$  ConstFloat32 c1, $\$C$  ConstFloat32 c2]
    using cs-def
    by blast
    thus ?thesis
    apply (cases app-binop-f fop c1 c2)
    apply (metis reduce-simple.intros(10) reduce.intros(1) T-f32 True ap-
pend-Cons append-Nil)
    apply (metis reduce-simple.intros(9) reduce.intros(1) T-f32 True append-Cons
append-Nil)
    done
  next

```

```

    case False
    thus ?thesis
    using reduce-simple.intros(17) assms(4) reduce.intros(1) cs-def T-f32
    by fastforce
  qed
next
case T-f64
hence cs-def:∃ c1 c2. cs = [C ConstFloat64 c1, C ConstFloat64 c2]
  using const-list-split-2[OF assms(3,1)] assms(3) const-of-f64
  unfolding const-list-def
  by blast
show ?thesis
proof (cases e = Binop-f t fop)
  case True
  obtain c1 c2 where cs = [C ConstFloat64 c1, C ConstFloat64 c2]
    using cs-def
    by blast
  thus ?thesis
    apply (cases app-binop-f fop c1 c2)
      apply (metis reduce-simple.intros(12) reduce.intros(1) T-f64 True append-Cons append-Nil)
      apply (metis reduce-simple.intros(11) reduce.intros(1) T-f64 True append-Cons append-Nil)
    done
  next
  case False
  thus ?thesis
    using reduce-simple.intros(18) assms(4) reduce.intros(1) cs-def T-f64
    by fastforce
  qed
qed (simp-all add: is-float-t-def)

lemma progress-b-e:
  assumes  $\mathcal{C} \vdash b\text{-es} : (ts \rightarrow ts')$ 
     $\mathcal{S} \cdot \mathcal{C} \vdash cs : ([\ ] \rightarrow ts)$ 
     $(\bigwedge l\text{holed}. \neg(L\text{filled } 0 \text{ lholed } [Return] (cs@[*b\text{-es}])))$ 
     $\bigwedge i \text{ lholed}. \neg(L\text{filled } 0 \text{ lholed } [Br \ (i)] (cs@[*b\text{-es}])))$ 
    const-list cs
     $\neg \text{const-list } (* b\text{-es})$ 
     $i < \text{length } (s\text{-inst } \mathcal{S})$ 
     $\text{length } (\text{local } \mathcal{C}) = \text{length } (vs)$ 
     $Option.is\text{-none } (\text{memory } \mathcal{C}) = Option.is\text{-none } (inst.mem ((inst \ s)!i))$ 
  shows  $\exists a \ s' \ vs' \ es'. (s;vs;cs@[*b\text{-es}]) \rightsquigarrow\text{-i } (s';vs';es')$ 
  using assms
proof (induction b-es (ts -> ts') arbitrary: ts ts' cs rule: b-e-typing.induct)
  case (const C v)
  then show ?case
    unfolding const-list-def is-const-def
    by simp

```

```

next
  case (unop-i t C uu)
  thus ?case
    using progress-unop-testop-i[OF unop-i(2,1)]
    by fastforce
next
  case (unop-f t C uv)
  thus ?case
    using progress-unop-f[OF unop-f(2,1,5)]
    by fastforce
next
  case (binop-i t C uw)
  thus ?case
    using progress-binop-relop-i[OF binop-i(2,1)]
    by fastforce
next
  case (binop-f t C ux)
  thus ?case
    using progress-binop-relop-f[OF binop-f(2,1,5)]
    by fastforce
next
  case (testop t C uy)
  thus ?case
    using progress-unop-testop-i[OF testop(2,1)]
    by fastforce
next
  case (relop-i t C uz)
  thus ?case
    using progress-binop-relop-i[OF relop-i(2,1)]
    by fastforce
next
  case (relop-f t C va)
  thus ?case
    using progress-binop-relop-f[OF relop-f(2,1,5)]
    by fastforce
next
  case (convert t1 t2 sx C)
  obtain v where cs-def:cs = [$ C v] typeof v = t2
  using const-typeof const-of-const-list[OF - convert(6)] e-type-const-list[OF con-
vert(6,3)]
  by fastforce
  thus ?case
  proof (cases cvt t1 sx v)
  case None
  thus ?thesis
    using reduce.intros(1)[OF reduce-simple.convert-None[OF - None]] cs-def
  unfolding types-agree-def
  by fastforce
next

```

```

    case (Some a)
  thus ?thesis
    using reduce.intros(1)[OF reduce-simple.convert-Some[OF - Some]] cs-def
    unfolding types-agree-def
    by fastforce
qed
next
case (reinterpret t1 t2 C)
obtain v where cs-def:cs = [$ C v] typeof v = t2
  using const-typeof const-of-const-list[OF - reinterpret(6)] e-type-const-list[OF
reinterpret(6,3)]
  by fastforce
  thus ?case
    using reduce.intros(1)[OF reduce-simple.reinterpret]
    unfolding types-agree-def
    by fastforce
next
case (unreachable C ts ts')
  thus ?case
    using reduce.intros(1)[OF reduce-simple.unreachable] progress-L0[OF - unreach-
able(4)]
    by fastforce
next
case (nop C)
  thus ?case
    using reduce.intros(1)[OF reduce-simple.nop] progress-L0[OF - nop(4)]
    by fastforce
next
case (drop C t)
obtain v where cs = [$ C v]
  using const-of-const-list drop(4) e-type-const-list[OF drop(4,1)]
  by fastforce
  thus ?case
    using reduce.intros(1)[OF reduce-simple.drop] progress-L0[OF - drop(4)]
    by fastforce
next
case (select C t)
obtain v1 v2 v3 where cs-def:S·C ⊢ [$ C v3] : ([ ] -> [T-i32])
  cs = [$ C v1, $ C v2, $ C v3]
  using const-list-split-3 [OF select(4,1)] select(4)
  by (auto simp add: const-list-def list-all-iff dest!: e-type-const-unwrap)
obtain c3 where c-def:v3 = ConstInt32 c3
  using cs-def select(4) const-of-i32[OF - cs-def(1)]
  unfolding const-list-def
  by fastforce
have ∃ a s' vs' es'. (|s;vs;[$ C v1, $ C v2, $ C ConstInt32 c3, $Select]) ~→-i
(|s';vs';es'|)
proof (cases int-eq c3 0)
  case True

```

```

thus ?thesis
  using reduce.intros(1)[OF reduce-simple.select-false]
  by fastforce
next
  case False
  thus ?thesis
    using reduce.intros(1)[OF reduce-simple.select-true]
    by fastforce
qed
thus ?case
  using c-def cs-def
  by fastforce
next
  case (block tf tn tm C es)
  show ?case
    using reduce-simple.block[OF block(7), of - tn tm - es]
      e-type-const-list[OF block(7,4)] reduce.intros(1) block(1)
    by fastforce
next
  case (loop tf tn tm C es)
  show ?case
    using reduce-simple.loop[OF loop(7), of - tn tm - es]
      e-type-const-list[OF loop(7,4)] reduce.intros(1) loop(1)
    by fastforce
next
  case (if-wasm tf tn tm C es1 es2)
  obtain c1s c2s where cs-def:  $\mathcal{S}\cdot\mathcal{C} \vdash c1s : ([\ ] \rightarrow tn)$ 
     $\mathcal{S}\cdot\mathcal{C} \vdash c2s : ([\ ] \rightarrow [T-i32])$ 
    const-list c1s
    const-list c2s
    cs = c1s @ c2s
    using e-type-const-list-cons[OF if-wasm(9,6)] e-type-const-list
    by fastforce
  obtain c where c-def: c2s = [ $\$ C (ConstInt32 c)$ ]
    using const-of-i32 cs-def
    by fastforce
  have  $\exists a s' vs' es'. (\$s;vs;[\$ C (ConstInt32 c), \$ If tf es1 es2]) \rightsquigarrow -i (\$s';vs';es')$ 
  proof (cases int-eq c 0)
    case True
    thus ?thesis
      using reduce.intros(1)[OF reduce-simple.if-false]
      by fastforce
  next
  case False
  thus ?thesis
    using reduce.intros(1)[OF reduce-simple.if-true]
    by fastforce
qed
thus ?case

```

```

    using c-def cs-def progress-L0
    by fastforce
next
case (br i C ts t1s t2s)
thus ?case
    using Lfilled.intros(1)[OF br(6), of - [] [$Br i]]
    by fastforce
next
case (br-if j C ts)
obtain cs1 cs2 where cs-def: $\mathcal{S}\cdot\mathcal{C} \vdash cs1 : ([\ ] \rightarrow ts)$ 
                     $\mathcal{S}\cdot\mathcal{C} \vdash cs2 : ([\ ] \rightarrow [T-i32])$ 
                    const-list cs1
                    const-list cs2
                    cs = cs1 @ cs2
    using e-type-const-list-cons[OF br-if(6,3)] e-type-const-list
    by fastforce
obtain c where c-def:cs2 = [$C ConstInt32 c]
    using const-of-i32[OF cs-def(4,2)]
    by blast
have  $\exists a s' vs' es'. (s;vs;cs2@(\$* [Br-if j])) \rightsquigarrow -i (s';vs';es')$ 
proof (cases int-eq c 0)
  case True
  thus ?thesis
    using c-def reduce.intros(1)[OF reduce-simple.br-if-false]
    by fastforce
next
case False
thus ?thesis
    using c-def reduce.intros(1)[OF reduce-simple.br-if-true]
    by fastforce
qed
thus ?case
    using cs-def(5) progress-L0[OF - cs-def(3), of s vs cs2 @ ($* [Br-if j]) - - - -]
    by fastforce
next
case (br-table C ts is i' t1s t2s)
obtain cs1 cs2 where cs-def: $\mathcal{S}\cdot\mathcal{C} \vdash cs1 : ([\ ] \rightarrow (t1s @ ts))$ 
                     $\mathcal{S}\cdot\mathcal{C} \vdash cs2 : ([\ ] \rightarrow [T-i32])$ 
                    const-list cs1
                    const-list cs2
                    cs = cs1 @ cs2
    using e-type-const-list-cons[OF br-table(5), of  $\mathcal{S} \mathcal{C} (t1s @ ts) [T-i32]$ ]
        e-type-const-list[of -  $\mathcal{S} \mathcal{C} t1s @ ts (t1s @ ts) @ [T-i32]$ ]
        br-table(2,5)
    unfolding const-list-def
    by fastforce
obtain c where c-def:cs2 = [$C ConstInt32 c]
    using const-of-i32[OF cs-def(4,2)]

```

```

  by blast
have  $\exists a s' vs' es'. (s;vs;[\$C \text{ ConstInt32 } c, \$Br\text{-table } is \ i']) \rightsquigarrow\text{-}i (s';vs';es')$ 
proof (cases (nat-of-int c) < length is)
  case True
  show ?thesis
  using reduce.intros(1)[OF reduce-simple.br-table[OF True]]
  by fastforce
next
  case False
  hence length is  $\leq$  nat-of-int c
  by fastforce
  thus ?thesis
  using reduce.intros(1)[OF reduce-simple.br-table-length]
  by fastforce
qed
thus ?case
  using c-def cs-def progress-L0
  by fastforce
next
  case (return C ts t1s t2s)
  thus ?case
  using Lfilled.intros(1)[OF return(5), of - [] [$Return]]
  by fastforce
next
  case (call j C)
  show ?case
  using progress-L0[OF reduce.intros(2) call(6)]
  by fastforce
next
  case (call-indirect j C t1s t2s)
  obtain cs1 cs2 where cs-def: $\mathcal{S}\cdot\mathcal{C} \vdash cs1 : ([\ ] \rightarrow t1s)$ 
     $\mathcal{S}\cdot\mathcal{C} \vdash cs2 : ([\ ] \rightarrow [T\text{-i32}])$ 
    const-list cs1
    const-list cs2
    cs = cs1 @ cs2
  using e-type-const-list-cons[OF call-indirect(7), of S C t1s [T-i32]]
    e-type-const-list[of - S C t1s t1s @ [T-i32]]
    call-indirect(4)
  by fastforce
  obtain c where c-def:cs2 = [ $\$C \text{ ConstInt32 } c$ ]
  using cs-def(2,4) const-of-i32
  by fastforce
consider
  (1)  $\exists cl \text{ } tf. \text{stab } s \ i \ (\text{nat-of-int } c) = \text{Some } cl \wedge \text{stypes } s \ i \ j = tf \wedge \text{cl-type } cl = tf$ 
| (2)  $\exists cl. \text{stab } s \ i \ (\text{nat-of-int } c) = \text{Some } cl \wedge \text{stypes } s \ i \ j \neq \text{cl-type } cl$ 
| (3)  $\text{stab } s \ i \ (\text{nat-of-int } c) = \text{None}$ 
  by (metis option.collapse)
  hence  $\exists a s' vs' es'. (s;vs;[\$C \text{ ConstInt32 } c, \$Call\text{-indirect } j]) \rightsquigarrow\text{-}i (s';vs';es')$ 
proof (cases)

```

```

    case 1
    thus ?thesis
      using reduce.intros(3)
      by blast
  next
  case 2
  thus ?thesis
    using reduce.intros(4)
    by blast
  next
  case 3
  thus ?thesis
    using reduce.intros(4)
    by blast
qed
then show ?case
  using c-def cs-def progress-L0
  by fastforce
next
case (get-local j C t)
obtain v vj vj' where v-def:v = vs ! j vj = (take j vs) vj' = (drop (j+1) vs)
  by blast
have j-def:j < length vs
  using get-local(1,9)
  by simp
hence vj-len:length vj = j
  using v-def(2)
  by fastforce
hence vs = vj @ [v] @ vj'
  using v-def id-take-nth-drop j-def
  by fastforce
thus ?case
  using progress-L0[OF reduce.intros(8)[OF vj-len, of s v vj] get-local(6)]
  by fastforce
next
case (set-local j C t)
obtain v vj vj' where v-def:v = vs ! j vj = (take j vs) vj' = (drop (j+1) vs)
  by blast
obtain v' where cs-def: cs = [$C v']
  using const-of-const-list set-local(3,6) e-type-const-list
  by fastforce
have j-def:j < length vs
  using set-local(1,9)
  by simp
hence vj-len:length vj = j
  using v-def(2)
  by fastforce
hence vs = vj @ [v] @ vj'
  using v-def id-take-nth-drop j-def

```

```

    by fastforce
  thus ?case
    using reduce.intros(9)[OF vj-len, of s v vj' v' i] cs-def
    by fastforce
next
case (tee-local i C t)
obtain v where cs = [C v]
using const-of-const-list tee-local(3,6) e-type-const-list
by fastforce
thus ?case
using reduce.intros(1)[OF reduce-simple.tee-local] tee-local(6)
unfolding const-list-def
by fastforce
next
case (get-global j C t)
thus ?case
using reduce.intros(10)[of s vs j i] progress-L0
by fastforce
next
case (set-global j C t)
obtain v where cs = [C v]
using const-of-const-list set-global(4,7) e-type-const-list
by fastforce
thus ?case
using reduce.intros(11)[of s i j v - vs]
by fastforce
next
case (load C n a tp-sx t off)
obtain c where c-def: cs = [C ConstInt32 c]
using const-of-i32 load(3,6) e-type-const-unwrap
unfolding const-list-def
by fastforce
obtain j where mem-some:smem-ind s i = Some j
using load(1,10)
unfolding smem-ind-def
by fastforce
have  $\exists a' s' vs' es'. (s;vs;[C ConstInt32 c, \$Load t tp-sx a off]) \rightsquigarrow -i (s';vs';es')$ 
proof (cases tp-sx)
case None
note tp-none = None
show ?thesis
proof (cases load ((mem s)!j) (nat-of-int c) off (t-length t))
case None
show ?thesis
using reduce.intros(13)[OF mem-some - None, of vs] tp-none load(2)
by fastforce
next
case (Some a)
show ?thesis

```

```

    using reduce.intros(12)[OF mem-some - Some, of vs] tp-none load(2)
    by fastforce
  qed
next
  case (Some a)
  obtain tp sx where tp-some:tp-sx = Some (tp, sx)
  using Some
  by fastforce
  show ?thesis
  proof (cases load-packed sx ((mem s)!j) (nat-of-int c) off (tp-length tp) (t-length
t))
    case None
    show ?thesis
    using reduce.intros(15)[OF mem-some - None, of vs] tp-some load(2)
    by fastforce
  next
  case (Some a)
  show ?thesis
  using reduce.intros(14)[OF mem-some - Some, of vs] tp-some load(2)
  by fastforce
  qed
  qed
  then show ?case
  using c-def progress-L0
  by fastforce
next
  case (store C n a tp t off)
  obtain cs' v where cs-def:S·C ⊢ [cs'] : ([ ] -> [T-i32])
    S·C ⊢ [$ C v] : ([ ] -> [t])
    cs = [cs', $ C v]
  using const-list-split-2[OF store(6,3)] e-type-const-unwrap
  unfolding const-list-def
  by fastforce
  have t-def:typeof v = t
  using cs-def(2) b-e-type-value[OF unlift-b-e[of S C [C v] ([ ] -> [t])]]
  by fastforce
  obtain j where mem-some:smem-ind s i = Some j
  using store(1,10)
  unfolding smem-ind-def
  by fastforce
  obtain c where c-def:cs' = $C ConstInt32 c
  using const-of-i32[OF - cs-def(1)] cs-def(3) store(6)
  unfolding const-list-def
  by fastforce
  have ∃ a' s' vs' es'. (|s;vs;[$C ConstInt32 c, $C v, $Store t tp a off]) ~→-i
(|s';vs';es'|)
  proof (cases tp)
  case None
  note tp-none = None

```

```

show ?thesis
proof (cases store (s.mem s ! j) (nat-of-int c) off (bits v) (t-length t))
  case None
  show ?thesis
  using reduce.intros(17)[OF - mem-some - None, of vs] t-def tp-none store(2)
  unfolding types-agree-def
  by fastforce
next
  case (Some a)
  show ?thesis
  using reduce.intros(16)[OF - mem-some - Some, of vs] t-def tp-none store(2)
  unfolding types-agree-def
  by fastforce
qed
next
  case (Some a)
  note tp-some = Some
  show ?thesis
  proof (cases store-packed (s.mem s ! j) (nat-of-int c) off (bits v) (tp-length a))
    case None
    show ?thesis
    using reduce.intros(19)[OF - mem-some - None, of t vs] t-def tp-some
store(2)
    unfolding types-agree-def
    by fastforce
  next
  case (Some a)
  show ?thesis
  using reduce.intros(18)[OF - mem-some - Some, of t vs] t-def tp-some
store(2)
  unfolding types-agree-def
  by fastforce
  qed
qed
then show ?case
  using c-def cs-def progress-L0
  by fastforce
next
  case (current-memory C n)
  obtain j where mem-some:smem-ind s i = Some j
  using current-memory(1,9)
  unfolding smem-ind-def
  by fastforce
  thus ?case
  using progress-L0[OF reduce.intros(20)[OF mem-some] current-memory(5), of
- - vs []]
  by fastforce
next
  case (grow-memory C n)

```

```

obtain  $c$  where  $c\text{-def}:cs = [\$C\ ConstInt32\ c]$ 
  using  $const\text{-of}\text{-}i32\ grow\text{-}memory(2,5)$ 
  by  $fastforce$ 
obtain  $j$  where  $mem\text{-}some:smem\text{-}ind\ s\ i = Some\ j$ 
  using  $grow\text{-}memory(1,9)$ 
  unfolding  $smem\text{-}ind\text{-}def$ 
  by  $fastforce$ 
show  $?case$ 
  using  $reduce.intros(22)[OF\ mem\text{-}some,\ of\ -]\ c\text{-}def$ 
  by  $fastforce$ 
next
  case ( $empty\ C$ )
  thus  $?case$ 
  unfolding  $const\text{-}list\text{-}def$ 
  by  $simp$ 
next
  case ( $composition\ C\ es\ t1s\ t2s\ e\ t3s$ )
  consider  $(1) \neg const\text{-}list\ (\$*\ es) \mid (2) const\text{-}list\ (\$*\ es) \neg const\text{-}list\ (\$*[e])$ 
  using  $composition(9)$ 
  unfolding  $const\text{-}list\text{-}def$ 
  by  $fastforce$ 
thus  $?case$ 
proof ( $cases$ )
  case  $1$ 
  have  $(\bigwedge lholed.\ \neg Lfilled\ 0\ lholed\ [\$Return]\ (cs\ @\ (\$*\ es)))$ 
     $(\bigwedge i\ lholed.\ \neg Lfilled\ 0\ lholed\ [\$Br\ i]\ (cs\ @\ (\$*\ es)))$ 
  proof  $safe$ 
  fix  $lholed$ 
  assume  $Lfilled\ 0\ lholed\ [\$Return]\ (cs\ @\ (\$*\ es))$ 
  hence  $\exists lholed'.\ Lfilled\ 0\ lholed'\ [\$Return]\ (cs\ @\ (\$*\ es\ @\ [e]))$ 
  proof ( $cases\ rule:\ Lfilled.cases$ )
  case ( $L0\ vs\ es'$ )
  thus  $?thesis$ 
  using  $Lfilled.intros(1)[of\ vs\ -\ es'\ @\ (\$*[e])\ [\$Return]]$ 
  by ( $metis\ append.assoc\ map\text{-}append$ )
  qed  $simp$ 
thus  $False$ 
  using  $composition(6)$ 
  by  $simp$ 
next
  fix  $i\ lholed$ 
  assume  $Lfilled\ 0\ lholed\ [\$Br\ i]\ (cs\ @\ (\$*\ es))$ 
  hence  $\exists lholed'.\ Lfilled\ 0\ lholed'\ [\$Br\ i]\ (cs\ @\ (\$*\ es\ @\ [e]))$ 
  proof ( $cases\ rule:\ Lfilled.cases$ )
  case ( $L0\ vs\ es'$ )
  thus  $?thesis$ 
  using  $Lfilled.intros(1)[of\ vs\ -\ es'\ @\ (\$*[e])\ [\$Br\ i]]$ 
  by ( $metis\ append.assoc\ map\text{-}append$ )
  qed  $simp$ 

```

```

thus False
  using composition(7)
  by simp
qed
thus ?thesis
  using composition(2)[OF composition(5) - - composition(8) 1 composition(10,11,12)] progress-L0[of s vs (cs @ ($* es)) i - - [] $*[e]]
  unfolding const-list-def
  by fastforce
next
  case 2
  hence const-list (cs@($* es))
  using composition(8)
  unfolding const-list-def
  by simp
  moreover
  have  $\mathcal{S}\cdot\mathcal{C} \vdash (cs@($* es)) : ([\ ] \rightarrow ts)$ 
  using composition(5) e-typing-s-typing.intros(1)[OF composition(1)] e-type-comp-conc
  by fastforce
  ultimately
  show ?thesis
  using composition(4)[of (cs@($* es))] 2(2) composition(6,7) composition(10-)
  by fastforce
qed
next
  case (weakening C es t1s t2s ts)
  obtain cs1 cs2 where cs-def:  $\mathcal{S}\cdot\mathcal{C} \vdash cs1 : ([\ ] \rightarrow ts)$ 
     $\mathcal{S}\cdot\mathcal{C} \vdash cs2 : ([\ ] \rightarrow t1s)$ 
     $cs = cs1 @ cs2$ 
    const-list cs1
    const-list cs2
  using e-type-const-list-cons[OF weakening(6,3)] e-type-const-list[of - S C ts ts @ t1s]
  by fastforce
  have ( $\bigwedge lholed. \neg Lfilled\ 0\ lholed\ [\$Return]\ (cs2\ @\ (\$*\ es))$ )
    ( $\bigwedge i\ lholed. \neg Lfilled\ 0\ lholed\ [\$Br\ i]\ (cs2\ @\ (\$*\ es))$ )
  proof safe
    fix lholed
    assume  $Lfilled\ 0\ lholed\ [\$Return]\ (cs2\ @\ (\$*\ es))$ 
    hence  $\exists lholed'. Lfilled\ 0\ lholed'\ [\$Return]\ (cs1\ @\ cs2\ @\ (\$*\ es))$ 
    proof (cases rule: Lfilled.cases)
      case (L0 vs es')
        thus ?thesis
          using Lfilled.intros(1)[of cs1 @ vs - es' [$Return]] cs-def(4)
          unfolding const-list-def
          by fastforce
    qed simp
  thus False
  using weakening(4) cs-def(3)

```

```

    by simp
next
  fix i lholed
  assume Lfilled 0 lholed [\$Br i] (cs2 @ (\$* es))
  hence  $\exists$  lholed'. Lfilled 0 lholed' [\$Br i] (cs1 @ cs2 @ (\$* es))
  proof (cases rule: Lfilled.cases)
    case (L0 vs es')
    thus ?thesis
      using Lfilled.intros(1)[of cs1 @ vs - es' [\$Br i]] cs-def(4)
      unfolding const-list-def
      by fastforce
  qed simp
  thus False
    using weakening(5) cs-def(3)
    by simp
  qed
  hence  $\exists$  a s' vs' es'. (|s;vs;cs2@(\$*es)|)  $\rightsquigarrow$ -i (|s';vs';es'|)
    using weakening(2)[OF cs-def(2) - - cs-def(5) weakening(7)] weakening(8-)
    by fastforce
  thus ?case
    using progress-L0[OF - cs-def(4), of s vs cs2 @ (\$* es) i - - - []] cs-def(3)
    by fastforce
  qed

```

lemma progress-e:

```

  assumes  $\mathcal{S} \cdot \text{None} \Vdash$ -i vs;cs-es : ts'
     $\bigwedge$  k lholed.  $\neg$ (Lfilled k lholed [\$Return] cs-es)
     $\bigwedge$  i k lholed. (Lfilled k lholed [\$Br (i)] cs-es)  $\implies$   $i < k$ 
    cs-es  $\neq$  [Trap]
     $\neg$  const-list (cs-es)
    store-typing s  $\mathcal{S}$ 
  shows  $\exists$  a s' vs' es'. (|s;vs;cs-es)|  $\rightsquigarrow$ -i (|s';vs';es'|)
  proof -
    fix C cs es ts-c
    have prems1:
       $\mathcal{S} \cdot \mathcal{C} \vdash$  es : (ts-c -> ts')  $\implies$ 
       $\mathcal{S} \cdot \mathcal{C} \vdash$  cs-es : ([ ] -> ts')  $\implies$ 
      cs-es = cs@es  $\implies$ 
      const-list cs  $\implies$ 
       $\mathcal{S} \cdot \mathcal{C} \vdash$  cs : ([ ] -> ts-c)  $\implies$ 
      ( $\bigwedge$  k lholed.  $\neg$ (Lfilled k lholed [\$Return] cs-es))  $\implies$ 
      ( $\bigwedge$  i k lholed. (Lfilled k lholed [\$Br (i)] cs-es)  $\implies$   $i < k$ )  $\implies$ 
      cs-es  $\neq$  [Trap]  $\implies$ 
       $\neg$  const-list (cs-es)  $\implies$ 
      store-typing s  $\mathcal{S}$   $\implies$ 
       $i < \text{length} (s\text{-inst } \mathcal{S}) \implies$ 
      length (local C) = length (vs)  $\implies$ 
      Option.is-none (memory C) = Option.is-none (inst.mem ((inst s)!i))  $\implies$ 
       $\exists$  a s' vs' cs-es'. (|s;vs;cs-es)|  $\rightsquigarrow$ -i (|s';vs';cs-es'|)

```

and *prems2*:
 $\mathcal{S} \cdot \text{None} \vdash -i \text{ vs}; \text{cs-es} : \text{ts}' \implies$
 $(\bigwedge k \text{ lholed. } \neg(\text{Lfilled } k \text{ lholed } [\text{\$Return}] \text{ cs-es})) \implies$
 $(\bigwedge i k \text{ lholed. } (\text{Lfilled } k \text{ lholed } [\text{\$Br } (i)] \text{ cs-es} \implies i < k) \implies$
 $\text{cs-es} \neq [\text{Trap}] \implies$
 $\neg \text{const-list } (\text{cs-es}) \implies$
 $\text{store-typing } s \mathcal{S} \implies$
 $\exists a \text{ s}' \text{ vs}' \text{ cs-es}'. (\text{s}; \text{vs}; \text{cs-es}) \rightsquigarrow -i (\text{s}'; \text{vs}'; \text{cs-es}')$

proof (*induction arbitrary: vs ts-c ts' i cs-es cs rule: e-typing-s-typing.inducts*)
case (1 C b-es tf S)
hence $\mathcal{C} \vdash \text{b-es} : (\text{ts-c} \rightarrow \text{ts}')$
using *e-type-comp-conc1*[of S C cs (\$* b-es) [] ts'] *unlift-b-e*
by (*metis e-type-const-conv-vs typing-map-typeof*)
then show ?*case*
using *progress-b-e*[OF - 1(5) - - 1(4)] 1(3,4,9) *list-all-append 1*
unfolding *const-list-def*
by *fastforce*

next
case (2 S C es t1s t2s e t3s)
show ?*case*
proof (*cases const-list es*)
case *True*
hence *const-list* (cs@es)
using 2(7)
unfolding *const-list-def*
by *simp*
moreover
have $\exists \text{ts}'' . (\mathcal{S} \cdot \mathcal{C} \vdash (\text{cs} @ \text{es}) : ([\] \rightarrow \text{ts}''))$
using 2(5,6)
by (*metis append.assoc e-type-comp-conc1*)
ultimately
show ?*thesis*
using 2(4)[OF 2(5) - - - 2(9,10,11,12,13,14,15), of (cs@es)] 2(6,16)
by *fastforce*

next
case *False*
hence $\neg \text{const-list } (\text{cs} @ \text{es})$
unfolding *const-list-def*
by *simp*
moreover
have $\exists \text{ts}'' . (\mathcal{S} \cdot \mathcal{C} \vdash (\text{cs} @ \text{es}) : ([\] \rightarrow \text{ts}''))$
using 2(5,6)
by (*metis append.assoc e-type-comp-conc1*)
moreover
have $\bigwedge k \text{ lholed. } \neg \text{Lfilled } k \text{ lholed } [\text{\$Return}] (\text{cs} @ \text{es})$
proof -
{
assume $\exists k \text{ lholed. } \text{Lfilled } k \text{ lholed } [\text{\$Return}] (\text{cs} @ \text{es})$
then obtain *k lholed where local-assms:Lfilled k lholed* [*\\$Return*] (cs @

```

es)
  by blast
  hence  $\exists \text{ lholed}'. \text{Lfilled } k \text{ lholed}' [\text{\$Return}] (\text{cs } @ \text{ es } @ [e])$ 
  proof (cases rule: Lfilled.cases)
    case (L0 vs es')
      obtain lholed' where lholed' = LBase vs (es'@[e])
        by blast
      thus ?thesis
        using L0
        by (metis Lfilled.intros(1) append.assoc)
    next
      case (LN vs ts es' l es'' k lfilledk)
      obtain lholed' where lholed' = LRec vs ts es' l (es''@[e])
        by blast
      thus ?thesis
        using LN
        by (metis Lfilled.intros(2) append.assoc)
  qed
  hence False
    using 2(6,9)
    by blast
}
thus  $\bigwedge k \text{ lholed}. \neg \text{Lfilled } k \text{ lholed} [\text{\$Return}] (\text{cs } @ \text{ es})$ 
  by blast
qed
moreover
have  $\bigwedge i k \text{ lholed}. \text{Lfilled } k \text{ lholed} [\text{\$Br } i] (\text{cs } @ \text{ es}) \implies i < k$ 
proof -
  {
    assume  $\exists i k \text{ lholed}. \text{Lfilled } k \text{ lholed} [\text{\$Br } i] (\text{cs } @ \text{ es}) \wedge \neg(i < k)$ 
    then obtain i k lholed where local-assms:Lfilled k lholed [\$Br i] (cs @
es)  $\neg(i < k)$ 
      by blast
    hence  $\exists \text{ lholed}'. \text{Lfilled } k \text{ lholed}' [\text{\$Br } i] (\text{cs } @ \text{ es } @ [e]) \wedge \neg(i < k)$ 
    proof (cases rule: Lfilled.cases)
      case (L0 vs es')
        obtain lholed' where lholed' = LBase vs (es'@[e])
          by blast
        thus ?thesis
          using L0 local-assms(2)
          by (metis Lfilled.intros(1) append.assoc)
      next
        case (LN vs ts es' l es'' k lfilledk)
        obtain lholed' where lholed' = LRec vs ts es' l (es''@[e])
          by blast
        thus ?thesis
          using LN local-assms(2)
          by (metis Lfilled.intros(2) append.assoc)
    qed
  }

```

```

    hence False
      using 2(6,10)
      by blast
  }
  thus  $\bigwedge i k \text{ lholed. } L\text{filled } k \text{ lholed } [\$Br\ i] (cs\ @\ es) \implies i < k$ 
    by blast
qed
moreover
note preds = calculation
show ?thesis
proof (cases cs @ es = [Trap])
  case True
  thus ?thesis
    using reduce-simple.trap[of - (LBase [] [e])]
      Lfilled.intros(1)[of [] LBase [] [e] [e] cs @ es]
      reduce.intros(1) 2(6,11)
    unfolding const-list-def
    by (metis append.assoc append-Nil list.pred-inject(1))
  next
  case False
  thus ?thesis
    using 2(3)[OF - - 2(7,8) - - - - 2(13,14,15)] preds 2(6,16)
      progress-L0[of s vs (cs @ es) - - - - [] [e]]
    unfolding const-list-def
    by (metis append.assoc append-Nil list.pred-inject(1))
  qed
qed
next
case (3 S C es t1s t2s ts)
  thus ?case
    by fastforce
next
case (4 S C)
  have cs-es-def: Lfilled 0 (LBase cs []) [Trap] cs-es
    using Lfilled.intros(1)[OF 4(3), of - [] [Trap]] 4(2)
    by fastforce
  thus ?case
    using reduce-simple.trap[OF 4(7) cs-es-def] reduce.intros(1)
    by blast
next
case (5 S ts j vls es n C)
  consider (1) ( $\bigwedge k \text{ lholed. } \neg L\text{filled } k \text{ lholed } [\$Return] es$ )
    ( $\bigwedge k \text{ lholed } i. (L\text{filled } k \text{ lholed } [\$Br\ i] es) \implies i < k$ )
    es  $\neq$  [Trap]
     $\neg \text{const-list } es$ 
    | (2)  $\exists k \text{ lholed. } L\text{filled } k \text{ lholed } [\$Return] es$ 
    | (3) const-list es  $\vee (es = [Trap])$ 
    | (4)  $\exists k \text{ lholed } i. (L\text{filled } k \text{ lholed } [\$Br\ i] es) \wedge i \geq k$ 
  using not-le-imp-less

```

```

    by blast
  thus ?case
proof (cases)
  case 1
  obtain  $s' vs'' a$  where  $temp1: (\!|s; vls; es\!) \rightsquigarrow - j (\!|s'; vs''; a\!)$ 
    using 5(3)[OF 1(1) - 1(3,4) 5(12)] 1(2)
    by fastforce
  show ?thesis
    using reduce.intros(24)[OF temp1, of vs] progress-L0[where ?cs = cs, OF
- 5(6)] 5(5)
    by fastforce
  next
  case 2
  then obtain  $k lholed$  where  $local-assms: (Lfilled\ k\ lholed\ [\!|\$Return\!] es)$ 
    by blast
  then obtain  $lholed' vs' C'$  where  $lholed'-def: (Lfilled\ k\ lholed'\ (vs'@[\!|\$Return\!]))$ 
es)

$$\mathcal{S} \cdot \mathcal{C}' \vdash vs' : (\!|\!| \rightarrow ts)$$


$$const-list\ vs'$$

    using progress-LN-return[OF local-assms, of  $\mathcal{S} - ts\ ts$ ] s-type-unfold[OF
5(1)]
    by fastforce
  hence  $temp1: \exists a. (\!|Local\ n\ j\ vls\ es\!) \rightsquigarrow (\!|vs'\!)$ 
    using reduce-simple.return[OF lholed'-def(3)]
    e-type-const-list[OF lholed'-def(3,2)] 5(2)
    by fastforce
  show ?thesis
    using temp1 progress-L0[OF reduce.intros(1) 5(6)] 5(5)
    by fastforce
  next
  case 3
  then consider (1) const-list es | (2) es = [Trap]
    by blast
  hence  $temp1: \exists a. (\!|s; vs; [Local\ n\ j\ vls\ es]\!) \rightsquigarrow - i (\!|s; vs; es\!)$ 
  proof (cases)
    case 1
    have length es = length ts
      using s-type-unfold[OF 5(1)] e-type-const-list[OF 1]
      by fastforce
    thus ?thesis
      using reduce-simple.local-const[OF 1] reduce.intros(1) 5(2)
      by fastforce
    next
    case 2
    thus ?thesis
      using reduce-simple.local-trap reduce.intros(1)
      by fastforce
  qed
  thus ?thesis

```

```

    using progress-L0[where ?cs = cs, OF - 5(6)] 5(5)
  by fastforce
next
case 4
then obtain k' l'holed' i' where temp1:Lfilled k' l'holed' [$Br (k'+i')] es
  using le-Suc-ex
  by blast
  obtain C' where c-def:C' = ((s-inst S)!j)(local := (local ((s-inst S)!j)) @
(map typeof vls), return := Some ts)
  by blast
  hence es-def:S.C' ⊢ es : ([] -> ts) j < length (s-inst S)
  using 5(1) s-type-unfold
  by fastforce+
  hence length (label C') = 0
  using c-def store-local-label-empty 5(12)
  by fastforce
  thus ?thesis
  using progress-LN1[OF temp1 es-def(1)]
  by linarith
qed
next
case (6 S cl tf C)
obtain ts'' where ts''-def:S.C ⊢ cs : ([] -> ts'') S.C ⊢ [Callcl cl] : (ts'' -> ts')
  using 6(2,3) e-type-comp-conc1
  by fastforce
obtain ts-c t1s t2s where cl-def:(ts'' = ts-c @ t1s)
  (ts' = ts-c @ t2s)
  cl-type cl = (t1s -> t2s)
  using e-type-callcl[OF ts''-def(2)]
  by fastforce
obtain vs1 vs2 where vs-def:S.C ⊢ vs1 : ([] -> ts-c)
  S.C ⊢ vs2 : (ts-c -> ts-c @ t1s)
  cs = vs1 @ vs2
  const-list vs1
  const-list vs2
  using e-type-const-list-cons[OF 6(4)] ts''-def(1) cl-def(1)
  by fastforce
have l:(length vs2) = (length t1s)
  using e-type-const-list vs-def(2,5)
  by fastforce
show ?case
proof (cases cl)
case (Func-native x11 x12 x13 x14)
  hence func-native-def:cl = Func-native x11 (t1s -> t2s) x13 x14
  using cl-def(3)
  unfolding cl-type-def
  by simp
  have ∃ a a'. (s;vs;vs2 @ [Callcl cl]) ~~- i (s;vs;a)
  using reduce.intros(5)[OF func-native-def] e-type-const-conv-vs[OF vs-def(5)]

```

```

l
  unfolding n-zeros-def
  by fastforce
thus ?thesis
  using progress-L0 vs-def(3,4) 6(3)
  by fastforce
next
case (Func-host x21 x22)
hence func-host-def:cl = Func-host (t1s -> t2s) x22
  using cl-def(3)
  unfolding cl-type-def
  by simp
obtain vcs where vcs-def:vs2 = $$* vcs
  using e-type-const-conv-vs[OF vs-def(5)]
  by blast
fix hs
have  $\exists s' a a'. (s;vs;vs2 @ [Callcl cl]) \rightsquigarrow - i (s';vs;a)$ 
proof (cases host-apply s (t1s -> t2s) x22 vcs hs)
  case None
  thus ?thesis
    using reduce.intros(7)[OF func-host-def] l vcs-def
    by fastforce
next
case (Some a)
  then obtain s' vcs' where ha-def:host-apply s (t1s -> t2s) x22 vcs hs =
Some (s', vcs')
  by (metis surj-pair)
  have list-all2 types-agree t1s vcs
  using e-typing-imp-list-types-agree vs-def(2,4) vcs-def
  by simp
  thus ?thesis
  using reduce.intros(6)[OF func-host-def - - - ha-def] l vcs-def
    host-apply-respect-type[OF - ha-def]
  by fastforce
qed
thus ?thesis
  using vs-def(3,4) 6(3) progress-L0
  by fastforce
qed
next
case (7 S C e0s ts t2s es n)
consider (1) ( $\bigwedge k$  lholed.  $\neg$  Lfilled k lholed [ $\$Return$ ] es)
  ( $\bigwedge k$  lholed i. (Lfilled k lholed [ $\$Br$  i] es)  $\implies$   $i < k$ )
  es  $\neq$  [Trap]
   $\neg$  const-list es
  | (2)  $\exists k$  lholed. Lfilled k lholed [ $\$Return$ ] es
  | (3) const-list es  $\vee$  (es = [Trap])
  | (4)  $\exists k$  lholed i. (Lfilled k lholed [ $\$Br$  i] es)  $\wedge$   $i = k$ 
  | (5)  $\exists k$  lholed i. (Lfilled k lholed [ $\$Br$  i] es)  $\wedge$   $i > k$ 

```

```

    using linorder-neqE-nat
  by blast
thus ?case
proof (cases)
  case 1
  have temp1: es = [] @ es const-list []
    unfolding const-list-def
  by auto
  have temp2:  $\mathcal{S}\cdot\mathcal{C}(\text{label} := [ts] @ \text{label } \mathcal{C}) \vdash [] : ([] \rightarrow [])$ 
    using b-e-typing.empty e-typing-s-typing.intros(1)
  by fastforce
  have  $\exists s' vs' a. (s; vs; es) \rightsquigarrow\!-\! i (s'; vs'; a)$ 
    using  $\gamma(5)[OF \gamma(2), \text{of } [] [], OF \text{temp1 temp2 } 1(1) - 1(3,4) \gamma(14,15)]$ 
     $1(2) \gamma(16,17)$ 
  unfolding const-list-def
  by fastforce
  then obtain  $s' vs' a$  where red-def:  $(s; vs; es) \rightsquigarrow\!-\! i (s'; vs'; a)$ 
  by blast
  have temp4:  $\bigwedge es. Lfilled\ 0 (LBase\ []\ [])\ es\ es$ 
    using Lfilled.intros(1)[of [] (LBase [] []) []]
  unfolding const-list-def
  by fastforce
  hence temp5:  $Lfilled\ 1 (LRec\ cs\ n\ e0s\ (LBase\ []\ [])\ [])\ es\ (cs@[Label\ n\ e0s\ es])$ 
    using Lfilled.intros(2)[of cs (LRec cs n e0s (LBase [] []) []) n e0s (LBase []
[]) [] 0 es es]  $\gamma(8)$ 
  unfolding const-list-def
  by fastforce
  have temp6:  $Lfilled\ 1 (LRec\ cs\ n\ e0s\ (LBase\ []\ [])\ [])\ a\ (cs@[Label\ n\ e0s\ a])$ 
    using temp4 Lfilled.intros(2)[of cs (LRec cs n e0s (LBase [] []) []) n e0s
(LBase [] []) [] 0 a a]  $\gamma(8)$ 
  unfolding const-list-def
  by fastforce
  show ?thesis
    using reduce.intros(23)[OF - temp5 temp6]  $\gamma(7)$  red-def
  by fastforce
next
  case 2
  then obtain  $k$  lholed where  $(Lfilled\ k\ lholed\ [\$Return]\ es)$ 
  by blast
  hence  $(Lfilled\ (k+1)\ (LRec\ cs\ n\ e0s\ lholed\ [])\ [\$Return]\ (cs@[Label\ n\ e0s\ es]))$ 
    using Lfilled.intros(2)  $\gamma(8)$ 
  by fastforce
  thus ?thesis
    using  $\gamma(10)[\text{of } k+1] \gamma(7)$ 
  by fastforce
next
  case 3
  hence temp1:  $\exists a. (s; vs; [Label\ n\ e0s\ es]) \rightsquigarrow\!-\! i (s; vs; es)$ 
    using reduce-simple.label-const reduce-simple.label-trap reduce.intros(1)

```

```

    by fastforce
  show ?thesis
    using progress-L0[OF - 7(8)] 7(7) temp1
    by fastforce
next
  case 4
  then obtain k lholed where lholed-def:(Lfilled k lholed [$Br (k+0)] es)
    by fastforce
  then obtain lholed' vs' C' where lholed'-def:(Lfilled k lholed' (vs'@[$Br (k)])
es)
    
$$\mathcal{S} \cdot \mathcal{C}' \vdash vs' : ([\ ] \rightarrow ts)$$

    const-list vs'
    using progress-LN[OF lholed-def 7(2), of ts]
    by fastforce
  have  $\exists es' a. ([\text{Label } n \text{ } e0s \text{ } es]) \rightsquigarrow ([vs'@e0s])$ 
    using reduce-simple.br[OF lholed'-def(3) - lholed'-def(1)] 7(3)
    e-type-const-list[OF lholed'-def(3,2)]
    by fastforce
  hence  $\exists es' a. ([s;vs;[\text{Label } n \text{ } e0s \text{ } es]) \rightsquigarrow - i ([s;vs;es'])$ 
    using reduce.intros(1)
    by fastforce
  thus ?thesis
    using progress-L0 7(7,8)
    by fastforce
next
  case 5
  then obtain i k lholed where lholed-def:(Lfilled k lholed [$Br i] es)  $i > k$ 
    using less-imp-add-positive
    by blast
  have k1-def:Lfilled (k+1) (LRec cs n e0s lholed []) [$Br i] cs-es
    using 7(7) Lfilled.intros(2)[OF 7(8) - lholed-def(1), of - n e0s []]
    by fastforce
  thus ?thesis
    using 7(11)[OF k1-def] lholed-def(2)
    by simp
qed
next
  case (8 i S tvs vs C rs es ts)
  have length (local C) = length vs
    using 8(2,3) store-local-label-empty[OF 8(1,11)]
    by fastforce
  moreover
  have Option.is-none (memory C) = Option.is-none (inst.mem ((inst s)!i))
    using store-mem-exists[OF 8(1,11)] 8(3)
    by simp
  ultimately show ?case
    using 8(6)[OF 8(4) - - - 8(7,8,9,10,11,1)]
    e-typing-s-typing.intros(1)[OF b-e-typing.empty[of C]]
    unfolding const-list-def

```

```

    by fastforce
  qed
  show ?thesis
    using prems2[OF assms]
    by fastforce
  qed

lemma progress-e1:
  assumes  $\mathcal{S} \cdot \text{None} \Vdash -i \text{ vs}; \text{ es} : \text{ ts}$ 
  shows  $\neg(\text{Lfilled } k \text{ lholed } [\text{\$Return}] \text{ es})$ 
  proof -
    {
      assume  $\exists k \text{ lholed. } (\text{Lfilled } k \text{ lholed } [\text{\$Return}] \text{ es})$ 
      then obtain  $k \text{ lholed}$  where  $\text{local-assms}:(\text{Lfilled } k \text{ lholed } [\text{\$Return}] \text{ es})$ 
        by blast
      obtain  $\mathcal{C}$  where  $c\text{-def}: i < \text{length } (s\text{-inst } \mathcal{S})$ 
         $\mathcal{C} = ((s\text{-inst } \mathcal{S})!i)(\text{local} := (\text{local } ((s\text{-inst } \mathcal{S})!i)) \text{ @ } (\text{map } \text{typeof } \text{vs}),$ 
         $\text{return} := \text{None})$ 
         $(\mathcal{S} \cdot \mathcal{C} \vdash \text{ es} : ([\ ] \text{ -> } \text{ ts}))$ 
      using  $\text{assms } s\text{-type-unfold}$ 
      by fastforce
      have  $\exists \text{ rs. } \text{return } \mathcal{C} = \text{Some } \text{rs}$ 
      using  $\text{local-assms } c\text{-def}(3)$ 
      proof (induction  $[\text{\$Return}] \text{ es}$  arbitrary:  $\mathcal{C} \text{ ts}$  rule:  $\text{Lfilled.induct}$ )
        case ( $L0 \text{ vs lholed es}'$ )
          thus ?case
          using  $e\text{-type-comp-conc2}[\text{OF } L0(3)] \text{ unlift-b-e[of } \mathcal{S} \mathcal{C} [\text{Return}]] \text{ b-e-type-return}$ 
          by fastforce
        next
          case ( $LN \text{ vs lholed tls es}' \text{ l es}'' \text{ k lfilledk}$ )
            thus ?case
            using  $e\text{-type-comp-conc2}[\text{OF } LN(5)] \text{ e-type-label[of } \mathcal{S} \mathcal{C} \text{ tls es}' \text{ lfilledk}]$ 
            by fastforce
      qed
      hence  $\text{False}$ 
      using  $c\text{-def}(2)$ 
      by fastforce
    }
    thus  $\bigwedge k \text{ lholed. } \neg(\text{Lfilled } k \text{ lholed } [\text{\$Return}] \text{ es})$ 
      by blast
  qed

```

```

lemma progress-e2:
  assumes  $\mathcal{S} \cdot \text{None} \Vdash -i \text{ vs}; \text{ es} : \text{ ts}$ 
     $\text{store-typing } s \mathcal{S}$ 
  shows  $(\text{Lfilled } k \text{ lholed } [\text{\$Br } (j)] \text{ es}) \implies j < k$ 
  proof -
    {
      assume  $(\exists i \text{ k lholed. } (\text{Lfilled } k \text{ lholed } [\text{\$Br } (i)] \text{ es}) \wedge i \geq k)$ 
    }
  
```

```

then obtain  $j\ k$  lholed where  $local\text{-}assms:(Lfilled\ k\ lholed\ [\$Br\ (k+j)]\ es)$ 
  by  $(metis\ le\text{-}iff\text{-}add)$ 
obtain  $\mathcal{C}$  where  $c\text{-}def:i < length\ (s\text{-}inst\ \mathcal{S})$ 
   $\mathcal{C} = ((s\text{-}inst\ \mathcal{S})!i)(local := (local\ ((s\text{-}inst\ \mathcal{S})!i))\ @\ (map\ typeof\ vs),$ 
 $return := None)$ 
   $(\mathcal{S}\cdot\mathcal{C} \vdash es : ([ ] \rightarrow ts))$ 
  using  $assms\ s\text{-}type\text{-}unfold$ 
  by  $fastforce$ 
have  $j < length\ (label\ \mathcal{C})$ 
  using  $progress\text{-}LN1[OF\ local\text{-}assms\ c\text{-}def(3)]$ 
  by  $-$ 
hence  $False$ 
  using  $store\text{-}local\text{-}label\text{-}empty(1)[OF\ c\text{-}def(1)\ assms(2)]\ c\text{-}def(2)$ 
  by  $fastforce$ 
}
thus  $(\bigwedge\ j\ k\ lholed.\ (Lfilled\ k\ lholed\ [\$Br\ (j)]\ es) \implies j < k)$ 
by  $fastforce$ 
qed

```

```

lemma  $progress\text{-}e3$ :
assumes  $\mathcal{S}\cdot None \Vdash\text{-}i\ vs;cs\text{-}es : ts'$ 
   $cs\text{-}es \neq [Trap]$ 
   $\neg\ const\text{-}list\ (cs\text{-}es)$ 
   $store\text{-}typing\ s\ \mathcal{S}$ 
shows  $\exists a\ s'\ vs'\ es'. (s;vs;cs\text{-}es) \rightsquigarrow\text{-}i\ (s';vs';es')$ 
using  $assms\ progress\text{-}e\ progress\text{-}e1\ progress\text{-}e2$ 
by  $fastforce$ 

```

end

7 Soundness Theorems

theory $Wasm\text{-}Soundness$ **imports** $Main\ Wasm\text{-}Properties$ **begin**

```

theorem  $preservation$ :
assumes  $\vdash\text{-}i\ s;vs;es : ts$ 
   $(s;vs;es) \rightsquigarrow\text{-}i\ (s';vs';es')$ 
shows  $\vdash\text{-}i\ s';vs';es' : ts$ 
proof  $-$ 
obtain  $\mathcal{S}$  where  $store\text{-}typing\ s\ \mathcal{S}\ \mathcal{S}\cdot None \Vdash\text{-}i\ vs;es : ts$ 
  using  $assms(1)\ config\text{-}typing.\text{simps}$ 
  by  $blast$ 
hence  $store\text{-}typing\ s'\ \mathcal{S}\ \mathcal{S}\cdot None \Vdash\text{-}i\ vs';es' : ts$ 
  using  $assms(2)\ store\text{-}preserved\ types\text{-}preserved\text{-}e$ 
  by  $simp\text{-}all$ 
thus  $?thesis$ 
  using  $config\text{-}typing.\text{intros}$ 
  by  $blast$ 
qed

```

```

theorem progress:
  assumes  $\vdash\text{-}i\ s;vs;es : ts$ 
  shows  $\text{const-list } es \vee es = [\text{Trap}] \vee (\exists a\ s'\ vs'\ es'. (\!s;vs;es\!) \rightsquigarrow\text{-}i\ (\!s';vs';es'\!))$ 
proof –
  obtain  $\mathcal{S}$  where store-typing  $s\ \mathcal{S}\ \mathcal{S}\cdot\text{None} \Vdash\text{-}i\ vs;es : ts$ 
    using assms config-typing.simps
    by blast
  thus ?thesis
    using progress-e3
    by blast
qed

end

```

8 Augmented Type Syntax for Concrete Checker

```

theory Wasm-Checker-Types imports Wasm HOL-Library.Sublist begin

```

```

datatype ct =
  TAny
  | TSome t

```

```

datatype checker-type =
  TopType ct list
  | Type t list
  | Bot

```

```

definition to-ct-list :: t list  $\Rightarrow$  ct list where
  to-ct-list ts = map TSome ts

```

```

fun ct-eq :: ct  $\Rightarrow$  ct  $\Rightarrow$  bool where
  ct-eq (TSome t) (TSome t') =  $(t = t')$ 
  | ct-eq TAny - = True
  | ct-eq - TAny = True

```

```

definition ct-list-eq :: ct list  $\Rightarrow$  ct list  $\Rightarrow$  bool where
  ct-list-eq ct1s ct2s = list-all2 ct-eq ct1s ct2s

```

```

definition ct-prefix :: ct list  $\Rightarrow$  ct list  $\Rightarrow$  bool where
  ct-prefix xs ys =  $(\exists as\ bs. ys = as@bs \wedge \text{ct-list-eq } as\ xs)$ 

```

```

definition ct-suffix :: ct list  $\Rightarrow$  ct list  $\Rightarrow$  bool where
  ct-suffix xs ys =  $(\exists as\ bs. ys = as@bs \wedge \text{ct-list-eq } bs\ xs)$ 

```

```

lemma ct-eq-commute:
  assumes ct-eq x y
  shows ct-eq y x
  using assms

```

by (*metis ct-eq.elims*(\exists) *ct-eq.simps*(1))

lemma *ct-eq-flip*: $ct\text{-}eq^{-1-1} = ct\text{-}eq$
using *ct-eq-commute*
by *fastforce*

lemma *ct-eq-common-tsome*: $ct\text{-}eq\ x\ y = (\exists t. ct\text{-}eq\ x\ (TSome\ t) \wedge ct\text{-}eq\ (TSome\ t)\ y)$
by (*metis ct-eq.elims*(\exists) *ct-eq.simps*(1))

lemma *ct-list-eq-commute*:
assumes *ct-list-eq xs ys*
shows *ct-list-eq ys xs*
using *assms ct-eq-commute List.List.list.rel-flip ct-eq-flip*
unfolding *ct-list-eq-def*
by *fastforce*

lemma *ct-list-eq-refl*: *ct-list-eq xs xs*
unfolding *ct-list-eq-def*
by (*metis ct-eq.elims*(\exists) *ct-eq.simps*(1) *list-all2-refl*)

lemma *ct-list-eq-length*:
assumes *ct-list-eq xs ys*
shows $length\ xs = length\ ys$
using *assms list-all2-lengthD*
unfolding *ct-list-eq-def*
by *blast*

lemma *ct-list-eq-concat*:
assumes *ct-list-eq xs ys*
ct-list-eq xs' ys'
shows $ct\text{-}list\text{-}eq\ (xs@xs')\ (ys@ys')$
using *assms*
unfolding *ct-list-eq-def*
by (*simp add: list-all2-appendI*)

lemma *ct-list-eq-ts-conv-eq*:
 $ct\text{-}list\text{-}eq\ (to\text{-}ct\text{-}list\ ts)\ (to\text{-}ct\text{-}list\ ts') = (ts = ts')$
unfolding *ct-list-eq-def to-ct-list-def*
list-all2-map1 list-all2-map2
ct-eq.simps(1)
by (*simp add: list-all2-eq*)

lemma *ct-list-eq-exists*: $\exists ys. ct\text{-}list\text{-}eq\ xs\ (to\text{-}ct\text{-}list\ ys)$
proof (*induction xs*)
case *Nil*
thus *?case*
unfolding *ct-list-eq-def to-ct-list-def*
by (*simp*)

```

next
  case (Cons a xs)
  thus ?case
    unfolding ct-list-eq-def to-ct-list-def
    apply (cases a)
    apply (metis ct-eq.simps(3) ct-eq-commute list.rel-intros(2) list.simps(9))
    apply (metis ct-eq.simps(1) list.rel-intros(2) list.simps(9))
    done
qed

lemma ct-list-eq-common-tsome-list:
  ct-list-eq xs ys = ( $\exists$  zs. ct-list-eq xs (to-ct-list zs)  $\wedge$  ct-list-eq (to-ct-list zs) ys)
proof (induction ys arbitrary: xs)
  case Nil
  thus ?case
    unfolding ct-list-eq-def to-ct-list-def
    by simp
next
  case (Cons a ys)
  show ?case
  proof (safe)
    assume assms:ct-list-eq xs (a # ys)
    then obtain x' xs' where xs-def:xs = x'#xs'
      by (meson ct-list-eq-def list-all2-Cons2)
    then obtain zs where zs-def:ct-eq x' a
      
$$ct\text{-list-eq } xs' (to\text{-ct-list } zs) \wedge ct\text{-list-eq } (to\text{-ct-list } zs) ys$$

    using Cons[of xs'] assms list-all2-Cons
    unfolding ct-list-eq-def
    by fastforce
    obtain z where ct-eq x' (TSome z) ct-eq (TSome z) a
      using ct-eq-common-tsome[of x' a] zs-def(1)
      by fastforce
    hence ct-list-eq (x'#xs') (to-ct-list (z#zs))  $\wedge$  ct-list-eq (to-ct-list (z#zs)) (a #
ys)
      using zs-def(2) list-all2-Cons
      unfolding ct-list-eq-def to-ct-list-def
      by simp
    thus  $\exists$  zs. ct-list-eq xs (to-ct-list zs)  $\wedge$  ct-list-eq (to-ct-list zs) (a # ys)
      using xs-def
      by fastforce
  next
  fix zs
  assume assms:ct-list-eq xs (to-ct-list zs) ct-list-eq (to-ct-list zs) (a # ys)
  then obtain x' xs' z' zs' where xs = x'#xs'
    
$$zs = z'\#zs'$$

    
$$ct\text{-list-eq } xs' (to\text{-ct-list } zs')$$

    
$$ct\text{-list-eq } (to\text{-ct-list } zs') (ys)$$

    using list-all2-Cons2
    unfolding ct-list-eq-def to-ct-list-def list-all2-map1 list-all2-map2

```

by (*metis* (*no-types*, *lifting*))
 thus *ct-list-eq* *xs* (*a* # *ys*)
 using *assms* *Cons* *ct-list-eq-def* *to-ct-list-def* *ct-eq-common-tsome*
 by (*metis* *list.simps(9)* *list-all2-Cons*)
 qed
 qed

lemma *ct-list-eq-cons-ct-list*:
 assumes *ct-list-eq* (*to-ct-list* *as*) (*xs* @ *ys*)
 shows $\exists bs\ bs'. as = bs @ bs' \wedge ct-list-eq (to-ct-list\ bs)\ xs \wedge ct-list-eq (to-ct-list\ bs')\ ys$
 using *assms*
proof (*induction* *xs* *arbitrary: as*)
 case *Nil*
 thus ?*case*
 by (*metis* *append-Nil* *ct-list-eq-ts-conv-eq* *list.simps(8)* *to-ct-list-def*)
next
 case (*Cons* *a* *xs*)
 thus ?*case*
 unfolding *ct-list-eq-def* *to-ct-list-def* *list-all2-map1*
 by (*meson* *list-all2-append2*)
 qed

lemma *ct-list-eq-cons-ct-list1*:
 assumes *ct-list-eq* (*to-ct-list* *as*) (*xs* @ (*to-ct-list* *ys*))
 shows $\exists bs. as = bs @ ys \wedge ct-list-eq (to-ct-list\ bs)\ xs$
 using *ct-list-eq-cons-ct-list[OF assms]* *ct-list-eq-ts-conv-eq*
 by *fastforce*

lemma *ct-list-eq-shared*:
 assumes *ct-list-eq* *xs* (*to-ct-list* *as*)
 ct-list-eq *ys* (*to-ct-list* *as*)
 shows *ct-list-eq* *xs* *ys*
 using *assms* *ct-list-eq-def*
 by (*meson* *ct-list-eq-common-tsome-list* *ct-list-eq-commute*)

lemma *ct-list-eq-take*:
 assumes *ct-list-eq* *xs* *ys*
 shows *ct-list-eq* (*take* *n* *xs*) (*take* *n* *ys*)
 using *assms* *list-all2-takeI*
 unfolding *ct-list-eq-def*
 by *blast*

lemma *ct-prefixI* [*intro?*]:
 assumes *ys* = *as* @ *zs*
 ct-list-eq *as* *xs*
 shows *ct-prefix* *xs* *ys*
 using *assms*
 unfolding *ct-prefix-def*

by *blast*

lemma *ct-prefixE* [*elim?*]:
assumes *ct-prefix xs ys*
obtains *as zs* **where** $ys = as @ zs$ *ct-list-eq as xs*
using *assms*
unfolding *ct-prefix-def*
by *blast*

lemma *ct-prefix-snoc* [*simp*]: $ct\text{-prefix } xs (ys @ [y]) = (ct\text{-list-eq } xs (ys@[y]) \vee ct\text{-prefix } xs ys)$
proof (*safe*)
assume $ct\text{-prefix } xs (ys @ [y]) \neg ct\text{-prefix } xs ys$
thus $ct\text{-list-eq } xs (ys @ [y])$
unfolding *ct-prefix-def ct-list-eq-def*
by (*metis butlast-append butlast-snoc ct-eq-flip list.rel-flip*)
next
assume $ct\text{-list-eq } xs (ys @ [y])$
thus $ct\text{-prefix } xs (ys @ [y])$
using *ct-list-eq-commute ct-prefixI*
by *fastforce*
next
assume $ct\text{-prefix } xs ys$
thus $ct\text{-prefix } xs (ys @ [y])$
using *append-assoc*
unfolding *ct-prefix-def*
by *blast*
qed

lemma *ct-prefix-nil:ct-prefix* $\square xs$
 $\neg ct\text{-prefix } (x \# xs) \square$
by (*simp-all add: ct-prefix-def ct-list-eq-def*)

lemma *Cons-ct-prefix-Cons*[*simp*]: $ct\text{-prefix } (x \# xs) (y \# ys) = ((ct\text{-eq } x y) \wedge ct\text{-prefix } xs ys)$
proof (*safe*)
assume $ct\text{-prefix } (x \# xs) (y \# ys)$
thus $ct\text{-eq } x y$
unfolding *ct-prefix-def ct-list-eq-def*
by (*metis ct-eq-commute hd-append2 list.sel(1) list.simps(3) list-all2-Cons2*)
next
assume $ct\text{-prefix } (x \# xs) (y \# ys)$
thus $ct\text{-prefix } xs ys$
unfolding *ct-prefix-def ct-list-eq-def*
by (*metis list.rel-distinct(1) list.sel(3) list-all2-Cons2 tl-append2*)
next
assume $ct\text{-eq } x y ct\text{-prefix } xs ys$
thus $ct\text{-prefix } (x \# xs) (y \# ys)$
unfolding *ct-prefix-def ct-list-eq-def*

by (*metis* (*full-types*) *append-Cons ct-list-eq-commute ct-list-eq-def list.rel-inject(2)*)
qed

lemma *ct-prefix-code* [*code*]:
ct-prefix [] *xs* \longleftrightarrow *True*
ct-prefix (*x # xs*) [] \longleftrightarrow *False*
ct-prefix (*x # xs*) (*y # ys*) \longleftrightarrow *ct-eq x y* \wedge *ct-prefix xs ys*
by (*simp-all add: ct-prefix-nil*)

lemma *ct-suffix-to-ct-prefix* [*code*]: *ct-suffix xs ys* = *ct-prefix (rev xs) (rev ys)*
unfolding *ct-suffix-def ct-prefix-def ct-list-eq-def*
by (*metis list-all2-rev1 rev-append rev-rev-ident*)

lemma *inj-TSome*: *inj TSome*
by (*meson ct.inject injI*)

lemma *to-ct-list-append*:
assumes *to-ct-list ts* = *as @ bs*
shows \exists *as'*. *to-ct-list as'* = *as*
 \exists *bs'*. *to-ct-list bs'* = *bs*
using *assms*

proof (*induct as arbitrary: ts*)

fix *ts*
assume *to-ct-list ts* = [] @ *bs*
thus \exists *as'*. *to-ct-list as'* = []
 \exists *bs'*. *to-ct-list bs'* = *bs*
unfolding *to-ct-list-def*
by *auto*

next

case (*Cons a as*)
fix *ts*
assume *local-assms:to-ct-list ts* = (*a # as*) @ *bs*
then obtain *t' ts'* **where** *ts* = *t' # ts'*
unfolding *to-ct-list-def*
by *auto*
thus \exists *as'*. *to-ct-list as'* = *a # as*
 \exists *as'*. *to-ct-list as'* = *bs*
using *Cons local-assms*
unfolding *to-ct-list-def*
apply *simp-all*
apply (*metis list.simps(9)*)
apply *blast*
done

qed

lemma *ct-suffixI* [*intro?*]:
assumes *ys* = *as @ zs*
 $\text{ct-list-eq } zs \ xs$
shows *ct-suffix xs ys*

```

using assms
unfolding ct-suffix-def
by blast

lemma ct-suffixE [elim?]:
assumes ct-suffix xs ys
obtains as zs where ys = as @ zs ct-list-eq zs xs
using assms
unfolding ct-suffix-def
by blast

lemma ct-suffix-nil: ct-suffix [] ts
unfolding ct-suffix-def
using ct-list-eq-refl
by auto

lemma ct-suffix-refl: ct-suffix ts ts
unfolding ct-suffix-def
using ct-list-eq-refl
by auto

lemma ct-suffix-length:
assumes ct-suffix ts ts'
shows length ts ≤ length ts'
using assms list-all2-lengthD
unfolding ct-suffix-def ct-list-eq-def
by fastforce

lemma ct-suffix-take:
assumes ct-suffix ts ts'
shows ct-suffix ((take (length ts - n) ts)) ((take (length ts' - n) ts'))
using assms ct-list-eq-take append-eq-conv-conj
unfolding ct-suffix-def
proof -
assume  $\exists as\ bs. ts' = as @ bs \wedge ct\text{-list-eq}\ bs\ ts$ 
then obtain ccs :: ct list and ccsa :: ct list where
  f1: ts' = ccs @ ccsa \wedge ct-list-eq ccsa ts
by moura
then have f2: length ccsa = length ts
by (meson ct-list-eq-length)
have  $\bigwedge n. ct\text{-list-eq}\ (take\ n\ ccsa)\ (take\ n\ ts)$ 
using f1 by (meson ct-list-eq-take)
then show  $\exists cs\ csa. take\ (length\ ts' - n)\ ts' = cs @ csa \wedge ct\text{-list-eq}\ csa\ (take\ (length\ ts - n)\ ts)$ 
using f2 f1 by auto
qed

lemma ct-suffix-ts-conv-suffix:
ct-suffix (to-ct-list ts) (to-ct-list ts') = suffix ts ts'

```


using *assms to-ct-list-append*[of *zs*]
unfolding *ct-suffix-def*
by *blast*
thus *?thesis*
using *assms ct-list-eq-cons-ct-list*[of *bs xs ys*]
unfolding *ct-suffix-def*
by (*metis append.assoc ct-list-eq-commute ct-list-eq-ts-conv-eq map-append to-ct-list-def*)
qed

lemma *ct-suffix-cons-ct-list1*:
assumes *ct-suffix (xs@(to-ct-list ys)) (to-ct-list zs)*
shows $\exists as. zs = as@ys \wedge ct-suffix\ xs\ (to-ct-list\ as)$
using *ct-suffix-cons-ct-list*[OF *assms*] *ct-list-eq-ts-conv-eq*
by *fastforce*

lemma *ct-suffix-cons2*:
assumes *ct-suffix (xs) (ys@zs)*
 $length\ xs = length\ zs$
shows *ct-list-eq xs zs*
using *assms*
by (*metis append-eq-append-conv ct-list-eq-commute ct-list-eq-def ct-suffix-def list-all2-lengthD*)

lemma *ct-suffix-imp-ct-list-eq*:
assumes *ct-suffix xs ys*
shows *ct-list-eq (drop (length ys - length xs) ys) xs*
using *assms ct-list-eq-def list-all2-lengthD*
unfolding *ct-suffix-def*
by *fastforce*

lemma *ct-suffix-extend-ct-list-eq*:
assumes *ct-suffix xs ys*
 $ct-list-eq\ xs'\ ys'$
shows *ct-suffix (xs@xs') (ys@ys')*
using *assms*
unfolding *ct-suffix-def ct-list-eq-def*
by (*meson append.assoc ct-list-eq-commute ct-list-eq-def list-all2-appendI*)

lemma *ct-suffix-extend-any1*:
assumes *ct-suffix xs ys*
 $length\ xs < length\ ys$
shows *ct-suffix (TAny#xs) ys*
proof –
obtain *as bs* **where** *ys-def:ys = as@bs*
 $ct-list-eq\ bs\ xs$
using *assms(1) ct-suffix-def*
by *fastforce*
hence $length\ as > 0$
using *list-all2-lengthD assms(2)*

unfolding *ct-list-eq-def*
by *fastforce*
then obtain *as' a* **where** *as-def:as = as'@[a]*
by (*metis append-butlast-last-id length-greater-0-conv*)
hence *ct-list-eq (a#bs) (TAny#xs)*
using *ys-def*
by (*meson ct-eq.simps(2) ct-list-eq-commute ct-list-eq-def list.rel-intros(2)*)
thus *?thesis*
using *as-def ys-def ct-suffix-def*
by *fastforce*
qed

lemma *ct-suffix-singleton-any: ct-suffix [TAny] [t]*
using *ct-suffix-extend-ct-list-eq[of [] [] [TAny] [t]] ct-suffix-nil*
by (*simp add: ct-list-eq-def*)

lemma *ct-suffix-cons-it: ct-suffix xs (xs'@xs)*
using *ct-list-eq-refl ct-suffix-def*
by *blast*

lemma *ct-suffix-singleton:*
assumes *length cts > 0*
shows *ct-suffix [TAny] cts*
proof –
have $\bigwedge c. ct\text{-prefix } [TAny] [c]$
using *ct-suffix-singleton-any ct-suffix-to-ct-prefix by force*
then show *?thesis*
by (*metis (no-types) Suc-leI append-butlast-last-id assms butlast.simps(2) ct-list-eq-commute*
ct-prefix-nil(2) ct-prefix-snoc ct-suffix-def impossible-Cons
length-Cons
list.size(3))

qed

lemma *ct-suffix-less:*
assumes *ct-suffix (xs@xs') ys*
shows *ct-suffix xs' ys*
using *assms*
unfolding *ct-suffix-def*
by (*metis append-eq-appendI ct-list-eq-def list-all2-append2*)

lemma *ct-suffix-unfold-one: ct-suffix (xs@[x]) (ys@[y]) = ((ct-eq x y) \wedge ct-suffix xs ys)*
using *ct-prefix-code(3)*
by (*simp add: ct-suffix-to-ct-prefix*)

lemma *ct-suffix-shared:*
assumes *ct-suffix cts (to-ct-list ts)*
ct-suffix cts' (to-ct-list ts)
shows *ct-suffix cts cts' \vee ct-suffix cts' cts*

```

proof (cases length cts > length cts')
  case True
  obtain as bs where cts-def:ts = as@bs
                        ct-list-eq cts (to-ct-list bs)
    using assms(1) ct-suffix-def to-ct-list-def
    by (metis append-Nil ct-suffix-cons-ct-list)
  obtain as' bs' where cts'-def:ts = as'@bs'
                        ct-list-eq cts' (to-ct-list bs')
    using assms(2) ct-suffix-def to-ct-list-def
    by (metis append-Nil ct-suffix-cons-ct-list)
  obtain ct1s ct2s where cts = ct1s@ct2s
                        length ct2s = length cts'

  using True
  by (metis add-diff-cancel-right' append-take-drop-id length-drop less-imp-le-nat
nat-le-iff-add)
  show ?thesis
  proof -
    obtain tts :: t list  $\Rightarrow$  ct list  $\Rightarrow$  ct list  $\Rightarrow$  t list and tsa :: t list  $\Rightarrow$  ct list  $\Rightarrow$  ct
list  $\Rightarrow$  t list where
       $\forall x0\ x1\ x2. (\exists v3\ v4. x0 = v3 @ v4 \wedge ct\text{-list-eq}\ x1\ (to\text{-ct-list}\ v4) \wedge ct\text{-suffix}$ 
 $x2\ (to\text{-ct-list}\ v3)) = (x0 = tts\ x0\ x1\ x2 @ tsa\ x0\ x1\ x2 \wedge ct\text{-list-eq}\ x1\ (to\text{-ct-list}$ 
 $(tsa\ x0\ x1\ x2)) \wedge ct\text{-suffix}\ x2\ (to\text{-ct-list}\ (tts\ x0\ x1\ x2)))$ 
      by moura
    then have f1: as' @ bs' = tts (as' @ bs') ct2s ct1s @ tsa (as' @ bs') ct2s ct1s
 $\wedge$  ct-list-eq ct2s (to-ct-list (tsa (as' @ bs') ct2s ct1s))  $\wedge$  ct-suffix ct1s (to-ct-list
(tts (as' @ bs') ct2s ct1s))
      using assms(1)  $\langle$ cts = ct1s @ ct2s $\rangle$  cts'-def(1) ct-suffix-cons-ct-list by force
    then have ct-list-eq cts' (to-ct-list (tsa (as' @ bs') ct2s ct1s))
      by (metis  $\langle$ ct-suffix cts' (to-ct-list ts) $\rangle$   $\langle$ length ct2s = length cts' $\rangle$  cts'-def(1)
ct-list-eq-length ct-suffix-cons2 map-append to-ct-list-def)
    then show ?thesis
      using f1 by (metis  $\langle$ cts = ct1s @ ct2s $\rangle$  ct-list-eq-shared ct-suffix-def)
  qed
next
  case False
  hence len:length cts'  $\geq$  length cts
    by linarith
  obtain as bs where cts-def:ts = as@bs
                        ct-list-eq cts (to-ct-list bs)
    using assms(1) ct-suffix-def to-ct-list-def
    by (metis append-Nil ct-suffix-cons-ct-list)
  obtain as' bs' where cts'-def:ts = as'@bs'
                        ct-list-eq cts' (to-ct-list bs')
    using assms(2) ct-suffix-def to-ct-list-def
    by (metis append-Nil ct-suffix-cons-ct-list)
  obtain ct1s ct2s where cts' = ct1s@ct2s
                        length ct2s = length cts

  using len
  by (metis add-diff-cancel-right' append-take-drop-id length-drop nat-le-iff-add)

```

```

show ?thesis
proof -
  obtain tts :: t list ⇒ ct list ⇒ ct list ⇒ t list and ttsa :: t list ⇒ ct list ⇒ ct list ⇒ t list where
    ∀ x0 x1 x2. (∃ v3 v4. x0 = v3 @ v4 ∧ ct-list-eq x1 (to-ct-list v4) ∧ ct-suffix
      x2 (to-ct-list v3)) = (x0 = tts x0 x1 x2 @ ttsa x0 x1 x2 ∧ ct-list-eq x1 (to-ct-list
      (ttsa x0 x1 x2)) ∧ ct-suffix x2 (to-ct-list (tts x0 x1 x2)))
    by moura
    then have f1: as @ bs = tts (as @ bs) ct2s ct1s @ ttsa (as @ bs) ct2s ct1s ∧
      ct-list-eq ct2s (to-ct-list (ttsa (as @ bs) ct2s ct1s)) ∧ ct-suffix ct1s (to-ct-list (tts
      (as @ bs) ct2s ct1s))
    using assms(2) ⟨cts' = ct1s @ ct2s⟩ cts-def(1) ct-suffix-cons-ct-list by force
    then have ct-list-eq cts (to-ct-list (ttsa (as @ bs) ct2s ct1s))
    by (metis ⟨ct-suffix cts (to-ct-list ts)⟩ ⟨length ct2s = length cts⟩ cts-def(1)
      ct-list-eq-length ct-suffix-cons2 map-append to-ct-list-def)
    then show ?thesis
    using f1 by (metis ⟨cts' = ct1s @ ct2s⟩ ct-list-eq-shared ct-suffix-def)
  qed
qed

```

```

fun checker-type-suffix::checker-type ⇒ checker-type ⇒ bool where
  checker-type-suffix (Type ts) (Type ts') = suffix ts ts'
| checker-type-suffix (Type ts) (TopType cts) = ct-suffix (to-ct-list ts) cts
| checker-type-suffix (TopType cts) (Type ts) = ct-suffix cts (to-ct-list ts)
| checker-type-suffix - - = False

```

```

fun consume :: checker-type ⇒ ct list ⇒ checker-type where
  consume (Type ts) cons = (if ct-suffix cons (to-ct-list ts)
    then Type (take (length ts - length cons) ts)
    else Bot)
| consume (TopType cts) cons = (if ct-suffix cons cts
  then TopType (take (length cts - length cons) cts)
  else (if ct-suffix cts cons
    then TopType []
    else Bot))
| consume - - = Bot

```

```

fun produce :: checker-type ⇒ checker-type ⇒ checker-type where
  produce (TopType ts) (Type ts') = TopType (ts@(to-ct-list ts'))
| produce (Type ts) (Type ts') = Type (ts@ts')
| produce (Type ts') (TopType ts) = TopType ts
| produce (TopType ts') (TopType ts) = TopType ts
| produce - - = Bot

```

```

fun type-update :: checker-type ⇒ ct list ⇒ checker-type ⇒ checker-type where
  type-update curr-type cons prods = produce (consume curr-type cons) prods

```

```

fun select-return-top :: [ct list] ⇒ ct ⇒ ct ⇒ checker-type where
  select-return-top ts ct1 TAny = TopType ((take (length ts - 3) ts) @ [ct1])

```

```

| select-return-top ts TAny ct2 = TopType ((take (length ts - 3) ts) @ [ct2])
| select-return-top ts (TSome t1) (TSome t2) = (if (t1 = t2)
    then (TopType ((take (length ts - 3) ts)
@ [TSome t1]))
    else Bot)

```

```

fun type-update-select :: checker-type ⇒ checker-type where
  type-update-select (Type ts) = (if (length ts ≥ 3 ∧ (ts!(length ts-2)) = (ts!(length
ts-3)))
    then consume (Type ts) [TAny, TSome T-i32]
    else Bot)
| type-update-select (TopType ts) = (case length ts of
  0 ⇒ TopType [TAny]
| Suc 0 ⇒ type-update (TopType ts) [TSome T-i32]
(TopType [TAny])
| Suc (Suc 0) ⇒ consume (TopType ts) [TSome
T-i32]
| - ⇒ type-update (TopType ts) [TAny, TAny,
TSome T-i32]
(select-return-top ts (ts!(length ts-2))
(ts!(length ts-3)))
| type-update-select - = Bot

```

```

fun c-types-agree :: checker-type ⇒ t list ⇒ bool where
  c-types-agree (Type ts) ts' = (ts = ts')
| c-types-agree (TopType ts) ts' = ct-suffix ts (to-ct-list ts')
| c-types-agree Bot - = False

```

lemma consume-type:

```

assumes consume (Type ts) ts' = c-t
  c-t ≠ Bot
shows ∃ ts''. ct-list-eq (to-ct-list ts) ((to-ct-list ts'')@ts') ∧ c-t = Type ts''
proof -
  {
    assume a1: (if ct-suffix ts' (map TSome ts) then Type (take (length ts - length
ts') ts) else Bot) = c-t
    assume a2: c-t ≠ Bot
    obtain ccs :: ct list ⇒ ct list ⇒ ct list and ccsa :: ct list ⇒ ct list ⇒ ct list
where
      f3: ∀ cs csa. ¬ ct-suffix cs csa ∨ csa = ccs cs csa @ ccsa cs csa ∧ ct-list-eq
(ccsa cs csa) cs
    using ct-suffixE by moura
    have f4: ct-suffix ts' (map TSome ts)
    using a2 a1 by metis
    then have f5: ct-list-eq (ccsa ts' (map TSome ts)) ts'
    using f3 by blast
    have f6: take (length (map TSome ts) - length (ccsa ts' (map TSome ts)))
(map TSome ts) @ ccsa ts' (map TSome ts) = map TSome ts
    using f4 f3 by (metis (full-types) suffixI suffix-take)
  }

```

```

have  $\bigwedge cs. ct\text{-list}\text{-eq} (cs @ csa\ ts' (map\ TSome\ ts)) (cs @ ts')$ 
  using f5 ct-list-eq-concat ct-list-eq-refl by blast
then have  $\exists tsa. ct\text{-list}\text{-eq} (map\ TSome\ ts) (map\ TSome\ tsa @ ts') \wedge c\text{-t} =$ 
Type tsa
  using f6 f5 f4 a1 by (metis (no-types) ct-list-eq-length length-map take-map)
}
thus ?thesis
  using assms to-ct-list-def
  by simp
qed

lemma consume-top-geq:
  assumes consume (TopType ts) ts' = c-t
    length ts  $\geq$  length ts'
    c-t  $\neq$  Bot
  shows ( $\exists as\ bs. ts = as@bs \wedge ct\text{-list}\text{-eq}\ bs\ ts' \wedge c\text{-t} = TopType\ as$ )
proof -
  consider (1) ct-suffix ts' ts
    | (2)  $\neg ct\text{-suffix}\ ts'\ ts\ ct\text{-suffix}\ ts\ ts'$ 
    | (3)  $\neg ct\text{-suffix}\ ts'\ ts\ \neg ct\text{-suffix}\ ts\ ts'$ 
  by blast
thus ?thesis
proof (cases)
  case 1
  hence TopType (take (length ts - length ts') ts) = c-t
    using assms
    by simp
  thus ?thesis
    using assms(2) 1 ct-list-eq-def
    unfolding ct-suffix-def
    by (metis (no-types, lifting) append-eq-append-conv append-take-drop-id diff-diff-cancel
length-drop list-all2-lengthD)
  next
  case 2
  thus ?thesis
    using assms append-eq-append-conv ct-list-eq-commute
    unfolding ct-suffix-def
    by (metis append.left-neutral ct-suffix-def ct-suffix-length le-antisym)
  next
  case 3
  thus ?thesis
    using assms
    by auto
qed
qed

lemma consume-top-leq:
  assumes consume (TopType ts) ts' = c-t
    length ts  $\leq$  length ts'

```

```

       $c-t \neq \text{Bot}$ 
shows  $c-t = \text{TopType } []$ 
using assms append-eq-conv-conj
by fastforce

lemma consume-type-type:
assumes  $\text{consume } xs \text{ cons} = (\text{Type } t\text{-int})$ 
shows  $\exists tn. xs = \text{Type } tn$ 
using assms
apply (cases xs)
  apply simp-all
apply (metis checker-type.distinct(1) checker-type.distinct(5))
done

lemma produce-type-type:
assumes  $\text{produce } xs \text{ cons} = (\text{Type } tm)$ 
shows  $\exists tn. xs = \text{Type } tn$ 
apply (cases xs; cases cons)
using assms
  apply simp-all
done

lemma consume-weaken-type:
assumes  $\text{consume } (\text{Type } tn) \text{ cons} = (\text{Type } t\text{-int})$ 
shows  $\text{consume } (\text{Type } (ts@tn)) \text{ cons} = (\text{Type } (ts@t\text{-int}))$ 
proof –
  obtain  $ts'$  where  $ct\text{-list-eq } (to\text{-ct-list } tn) (to\text{-ct-list } ts' @ \text{cons}) \wedge \text{Type } t\text{-int} =$ 
 $\text{Type } ts'$ 
    using consume-type[OF assms]
    by blast
  have  $cond: ct\text{-suffix } cons (to\text{-ct-list } tn)$ 
    using assms
    by (simp, metis checker-type.distinct(5))
  hence  $res:t\text{-int} = \text{take } (length \text{ tn} - length \text{ cons}) \text{ tn}$ 
    using assms
    by simp
  have  $ct\text{-suffix } cons (to\text{-ct-list } (ts@tn))$ 
    using cond
    unfolding to-ct-list-def
    by (metis append-assoc ct-suffix-def map-append)
  moreover
  have  $ts@t\text{-int} = \text{take } (length (ts@tn) - length \text{ cons}) (ts@tn)$ 
    using res take-append cond ct-suffix-length to-ct-list-def
    by fastforce
  ultimately
  show ?thesis
    by simp
qed

```

lemma *produce-weaken-type*:
assumes *produce* (Type *tn*) *cons* = (Type *tm*)
shows *produce* (Type (*ts*@*tn*)) *cons* = (Type (*ts*@*tm*))
using *assms*
by (*cases cons, simp-all*)

lemma *produce-nil*: *produce ts* (Type []) = *ts*
using *to-ct-list-def*
by (*cases ts, simp-all*)

lemma *c-types-agree-id*: *c-types-agree* (Type *ts*) *ts*
by *simp*

lemma *c-types-agree-top1*: *c-types-agree* (TopType []) *ts*
using *ct-suffix-ts-conv-suffix to-ct-list-def*
by (*simp add: ct-suffix-nil*)

lemma *c-types-agree-top2*:
assumes *ct-list-eq ts* (*to-ct-list ts'*)
shows *c-types-agree* (TopType *ts*) (*ts'*@*ts''*)
using *assms ct-list-eq-commute ct-suffix-def to-ct-list-def*
by *auto*

lemma *c-types-agree-imp-ct-list-eq*:
assumes *c-types-agree* (TopType *cts*) *ts*
shows $\exists ts' ts''. (ts = ts'@ts'') \wedge ct\text{-list-eq } cts (to\text{-ct-list } ts'')$
using *assms ct-suffix-def to-ct-list-def*
by (*simp, metis ct-list-eq-commute ct-list-eq-ts-conv-eq ct-suffix-ts-conv-suffix suffixE*
to-ct-list-append(2))

lemma *c-types-agree-not-bot-exists*:
assumes *ts* \neq Bot
shows $\exists ts\text{-c}. c\text{-types-agree } ts \text{ } ts\text{-c}$
using *assms ct-suffix-exists*
by (*cases ts, simp-all*)

lemma *consume-c-types-agree*:
assumes *consume* (Type *ts*) *cts* = (Type *ts'*)
c-types-agree ctn ts
shows $\exists c\text{-t}'. consume \text{ } ctn \text{ } cts = c\text{-t}' \wedge c\text{-types-agree } c\text{-t}' \text{ } ts'$
using *assms*
proof (*cases ctn*)
case (TopType *x1*)
have *1: ct-suffix cts* (*to-ct-list ts*)
using *assms*
by (*simp, metis checker-type.distinct(5)*)
hence *ct-suffix cts x1* \vee *ct-suffix x1 cts*
using TopType 1 *assms(2) ct-suffix-shared*

```

    by simp
  thus ?thesis
proof (rule disjE)
  assume local-assms:ct-suffix cts x1
  hence 2:consume (TopType x1) cts = TopType (take (length x1 - length cts)
x1)
    by simp
  have (take (length ts - length cts) ts) = ts'
    using assms 1
    by simp
  hence c-types-agree (TopType (take (length x1 - length cts) x1)) ts'
    using 2 assms local-assms TopType ct-suffix-take
    by (simp, metis length-map take-map to-ct-list-def)
  thus ?thesis
    using 2 TopType
    by simp
next
  assume local-assms:ct-suffix x1 cts
  hence 3:consume (TopType x1) cts = TopType []
    by (simp add: ct-suffix-length)
  thus ?thesis
    using TopType c-types-agree-top1
    by blast
qed
qed simp-all

```

lemma *type-update-type*:

```

  assumes type-update (Type ts) (to-ct-list cons) prods = ts'
    ts' ≠ Bot
  shows (ts' = prods ∧ (∃ ts-c. prods = (TopType ts-c)))
    ∨ (∃ ts-a ts-b. prods = Type ts-a ∧ ts = ts-b@cons ∧ ts' = Type
(ts-b@ts-a))
  using assms
  apply (cases prods)
  apply simp-all
  apply (metis (full-types) produce.simps(3) produce.simps(7))
  using ct-suffix-ts-conv-suffix suffix-take to-ct-list-def
  apply fastforce
  done

```

lemma *type-update-empty*: $\text{type-update } ts \text{ cons } (\text{Type } []) = \text{consume } ts \text{ cons}$

```

  using produce-nil
  by simp

```

lemma *type-update-top-top*:

```

  assumes type-update (TopType ts) (to-ct-list cons) (Type prods) = (TopType ts')
    c-types-agree (TopType ts') t-ag
  shows ct-suffix (to-ct-list prods) ts'

```

$\exists t\text{-ag}'. t\text{-ag} = t\text{-ag}'@prods \wedge c\text{-types-agree } (TopType\ ts) (t\text{-ag}'@cons)$

proof –

consider (1) $ct\text{-suffix } (to\text{-ct-list } cons) ts$
| (2) $\neg ct\text{-suffix } (to\text{-ct-list } cons) ts\ ct\text{-suffix } ts (to\text{-ct-list } cons)$
| (3) $\neg ct\text{-suffix } (to\text{-ct-list } cons) ts\ \neg ct\text{-suffix } ts (to\text{-ct-list } cons)$

by *blast*

hence $ct\text{-suffix } (to\text{-ct-list } prods) ts' \wedge (\exists t\text{-ag}'. t\text{-ag} = t\text{-ag}'@prods \wedge c\text{-types-agree } (TopType\ ts) (t\text{-ag}'@cons))$

proof (*cases*)

case 1

hence $ts' = (take\ (length\ ts - length\ cons)\ ts) @\ to\text{-ct-list } prods$
using *assms(1) to-ct-list-def*
by *simp*

moreover

then obtain $t\text{-ag}'$ **where** $t\text{-ag} = t\text{-ag}' @\ prods$
 $ct\text{-suffix } (take\ (length\ ts - length\ cons)\ ts) (to\text{-ct-list } t\text{-ag}')$
using *assms(2) ct-suffix-cons-ct-list1*
unfolding *c-types-agree.simps*
by *blast*

moreover

hence $ct\text{-suffix } ts (to\text{-ct-list } (t\text{-ag}'@cons))$
using 1 *ct-suffix-imp-ct-list-eq ct-suffix-extend-ct-list-eq to-ct-list-def*
by *fastforce*

ultimately

show *?thesis*
using *c-types-agree.simps(2) ct-list-eq-ts-conv-eq ct-suffix-def*
by *auto*

next

case 2

thus *?thesis*
using *assms*
by (*metis append.assoc c-types-agree.simps(2) checker-type.inject(1) consume.simps(2)*
 $ct\text{-list-eq-ts-conv-eq } ct\text{-suffix-cons-ct-list } ct\text{-suffix-def } map\text{-append } produce.simps(1) to\text{-ct-list-def } type\text{-update.simps}$)

next

case 3

thus *?thesis*
using *assms*
by *simp*

qed

thus $ct\text{-suffix } (to\text{-ct-list } prods) ts'$
 $\exists t\text{-ag}'. t\text{-ag} = t\text{-ag}'@prods \wedge c\text{-types-agree } (TopType\ ts) (t\text{-ag}'@cons)$

by *simp-all*

qed

lemma *type-update-select-length0*:
assumes $type\text{-update-select } (TopType\ cts) = tm$
 $length\ cts = 0$

```

       $tm \neq Bot$ 
shows  $tm = TopType [TAny]$ 
using assms
by simp

lemma type-update-select-length1:
assumes  $type\text{-}update\text{-}select (TopType\ cts) = tm$ 
       $length\ cts = 1$ 
       $tm \neq Bot$ 
shows  $ct\text{-}list\text{-}eq\ cts [TSome\ T\text{-}i32]$ 
       $tm = TopType [TAny]$ 
proof –
have  $1: type\text{-}update (TopType\ cts) [TSome\ T\text{-}i32] (TopType [TAny]) = tm$ 
      using assms(1,2)
      by simp
hence  $ct\text{-}suffix\ cts [TSome\ T\text{-}i32] \vee ct\text{-}suffix [TSome\ T\text{-}i32]\ cts$ 
      using assms(3)
      by (metis consume.simps(2) produce.simps(7) type-update.simps)
thus  $ct\text{-}list\text{-}eq\ cts [TSome\ T\text{-}i32]$ 
      using assms(2,3) ct-suffix-imp-ct-list-eq
      by (metis One-nat-def Suc-length-conv ct-list-eq-commute diff-Suc-1 drop-0
list.size(3))
show  $tm = TopType [TAny]$ 
      using  $1\ assms(3)\ consume\text{-}top\text{-}leq$ 
      by (metis One-nat-def assms(2) diff-Suc-1 diff-is-0-eq length-Cons list.size(3)
produce.simps(4,7) type-update.simps)
qed

```

```

lemma type-update-select-length2:
assumes  $type\text{-}update\text{-}select (TopType\ cts) = tm$ 
       $length\ cts = 2$ 
       $tm \neq Bot$ 
shows  $\exists t1\ t2. cts = [t1, t2] \wedge ct\text{-}eq\ t2 (TSome\ T\text{-}i32) \wedge tm = TopType [t1]$ 
proof –
obtain  $x\ y$  where  $cts\text{-}def: cts = [x, y]$ 
      using assms(2) List.length-Suc-conv[of cts Suc 0]
      by (metis length-0-conv length-Suc-conv numeral-2-eq-2)
moreover
hence  $consume (TopType [x, y]) [TSome\ T\text{-}i32] = tm$ 
      using assms(1,2)
      by simp
moreover
hence  $ct\text{-}suffix [x, y] [TSome\ T\text{-}i32] \vee ct\text{-}suffix [TSome\ T\text{-}i32] [x, y]$ 
      using assms(3)
      by (metis consume.simps(2))
hence  $ct\text{-}suffix [TSome\ T\text{-}i32] ([x]@[y])$ 
      using assms(2) ct-suffix-length
      by fastforce

```

moreover
hence $ct\text{-eq } y$ ($T\text{Some } T\text{-i32}$)
by ($metis\ ct\text{-eq}\text{-commute}\ ct\text{-list}\text{-eq}\text{-def}\ ct\text{-suffix}\text{-cons2}\ list.\text{rel}\text{-sel}\ list.\text{sel}(1)\ list.\text{simsps}(3)$
 $list.\text{size}(4)$)
ultimately
show $?thesis$
by $auto$
qed

lemma $type\text{-update}\text{-select}\text{-length3}$:
assumes $type\text{-update}\text{-select}\ (TopType\ cts) = (TopType\ ctm)$
 $length\ cts \geq 3$
shows $\exists\ cts'\ ct1\ ct2\ ct3.\ cts = cts'@[ct1, ct2, ct3] \wedge ct\text{-eq}\ ct3$ ($T\text{Some } T\text{-i32}$)
proof –
obtain $cts'\ cts''$ **where** $cts\text{-def}: cts = cts'@[ct1, ct2, ct3]$ $length\ cts'' = 3$
using $assms(2)$
by ($metis\ append\text{-take}\text{-drop}\text{-id}\ diff\text{-diff}\text{-cancel}\ length\text{-drop}$)
then obtain $ct1\ cts''2$ **where** $cts'' = ct1\#\ cts''2$ $length\ cts''2 = Suc\ (Suc\ 0)$
using $List.\text{length}\text{-Suc}\text{-conv}[of\ cts'\ Suc\ (Suc\ 0)]$
by ($metis\ length\text{-Suc}\text{-conv}\ numeral\text{-3}\text{-eq}\text{-3}$)
then obtain $ct2\ ct3$ **where** $cts'' = [ct1, ct2, ct3]$
using $List.\text{length}\text{-Suc}\text{-conv}[of\ cts''2\ Suc\ 0]$
by ($metis\ length\text{-0}\text{-conv}\ length\text{-Suc}\text{-conv}$)
hence $cts\text{-def}2: cts = cts'@[ct1, ct2, ct3]$
using $cts\text{-def}$
by $simp$
obtain nat **where** $length\ cts = Suc\ (Suc\ (Suc\ nat))$
using $assms(2)$
by ($simp\ add: cts\text{-def}$)
hence ($type\text{-update}\ (TopType\ cts)\ [TAny, TAny, TSome\ T\text{-i32}]\ (select\text{-return}\text{-top}\ cts\ (cts\ !\ (length\ cts\ -\ 2))\ (cts\ !\ (length\ cts\ -\ 3)))) = TopType\ ctm$
using $assms$
by $simp$
then obtain $c\text{-mid}$ **where** $consume\ (TopType\ cts)\ [TAny, TAny, TSome\ T\text{-i32}]$
 $= TopType\ c\text{-mid}$
by ($metis\ consume.\text{simsps}(2)\ produce.\text{simsps}(6)\ type\text{-update}.\text{simsps}$)
hence $ct\text{-suffix}\ [TAny, TAny, TSome\ T\text{-i32}]\ (cts'@[ct1, ct2, ct3])$
using $assms(2)\ consume\text{-top}\text{-geq}\ cts\text{-def}2$
by ($metis\ checker\text{-type}.\text{distinct}(3)\ ct\text{-suffix}\text{-def}\ length\text{-Cons}\ list.\text{size}(3)\ numeral\text{-3}\text{-eq}\text{-3}$)
hence $ct\text{-eq}\ ct3$ ($T\text{Some } T\text{-i32}$)
using $ct\text{-suffix}\text{-def}\ ct\text{-list}\text{-eq}\text{-def}$
by ($simp, metis\ append\text{-eq}\text{-append}\text{-conv}\ length\text{-Cons}\ list\text{-all}2\text{-Cons}\ list\text{-all}2\text{-length}D$)
thus $?thesis$
using $cts\text{-def}2$
by $simp$
qed

lemma $type\text{-update}\text{-select}\text{-type}\text{-length3}$:

```

assumes type-update-select (Type tn) = (Type tm)
shows  $\exists t ts'. tn = ts'@[t, t, T-i32]$ 
proof –
  have tn-cond:(length tn  $\geq$  3  $\wedge$  (tn!(length tn-2)) = (tn!(length tn-3)))
    using assms
    by (simp, metis checker-type.distinct(5))
  hence tm-def:consume (Type tn) [TAny, TSome T-i32] = Type tm
    using assms
    by simp
  obtain tn' tn'' where tn-split:tn = tn'@tn''
    length tn'' = 3
    using assms tn-cond
    by (metis append-take-drop-id diff-diff-cancel length-drop)
  then obtain t1 tn''2 where tn'' = t1#tn''2 length tn''2 = Suc (Suc 0)
    by (metis length-Suc-conv numeral-3-eq-3)
  then obtain t2 t3 where tn''-def:tn'' = [t1,t2,t3]
    using List.length-Suc-conv[of tn''2 Suc 0]
    by (metis length-0-conv length-Suc-conv)
  hence tn-def:tn = tn'@[t1,t2,t3]
    using tn-split
    by simp
  hence t1-t2-eq:t1 = t2
    using tn-cond
    by (metis (no-types, lifting) Suc-diff-Suc Suc-eq-plus1-left Suc-lessD tn''-def
      add-diff-cancel-right' diff-is-0-eq length-append neq0-conv
      not-less-eq-eq nth-Cons-0 nth-Cons-numeral
      nth-append numeral-2-eq-2 numeral-3-eq-3 numeral-One
      tn-split(2)
      zero-less-diff)
  have ct-suffix [TAny, TSome T-i32] (to-ct-list (tn' @ [t1, t2, t3]))
    using tn-def tm-def
    by (simp, metis checker-type.distinct(5))
  hence t3 = T-i32
    using ct-suffix-unfold-one[of [TAny] TSome T-i32 to-ct-list (tn' @ [t1, t2])
      TSome t3]
    ct-eq.simps(1)
    unfolding to-ct-list-def
    by simp
  thus ?thesis
    using t1-t2-eq tn-def
    by simp
qed

lemma select-return-top-exists:
assumes select-return-top cts c1 c2 = ctm
  ctm  $\neq$  Bot
shows  $\exists xs. ctm = TopType xs$ 
using assms
by (meson select-return-top.elims)

```

```

lemma type-update-select-top-exists:
  assumes type-update-select xs = (TopType tm)
  shows  $\exists tn. xs = TopType tn$ 
  using assms
proof (cases xs)
  case (Type x2)
  thus ?thesis
    using assms
    by (simp, metis checker-type.distinct(1) checker-type.distinct(3) consume.simps(1))
qed simp-all

```

```

lemma type-update-select-conv-select-return-top:
  assumes ct-suffix [TSome T-i32] cts
    length cts  $\geq 3$ 
  shows type-update-select (TopType cts) = (select-return-top cts (cts!(length cts-2))
    (cts!(length cts-3)))
proof -
  obtain nat where nat-def:length cts = Suc (Suc (Suc nat))
    using assms(2)
  by (metis add-eq-if diff-Suc-1 le-Suc-ex numeral-3-eq-3 nat.distinct(2))
  have ct-suffix [TAny, TAny, TSome T-i32] cts
    using assms ct-suffix-extend-any1
  by simp
  then obtain cts' where consume (TopType cts) [TAny, TAny, TSome T-i32] =
    TopType cts'
  by simp
  thus ?thesis
    using nat-def select-return-top-exists
    apply (cases select-return-top cts (cts ! Suc nat) (cts ! nat))
    apply simp-all
    apply (metis checker-type.distinct(1) checker-type.distinct(5))
  done
qed

```

```

lemma select-return-top-ct-eq:
  assumes select-return-top cts c1 c2 = TopType ctm
    length cts  $\geq 3$ 
    c-types-agree (TopType ctm) cm
  shows  $\exists c' cm'. cm = cm'@[c']$ 
     $\wedge$  ct-suffix (take (length cts - 3) cts) (to-ct-list cm')
     $\wedge$  ct-eq c1 (TSome c')
     $\wedge$  ct-eq c2 (TSome c')
proof (cases c1)
  case TAny
  note outer-TAny = TAny
  show ?thesis
  proof (cases c2)
  case TAny

```

```

hence take (length cts - 3) cts @ [TAny] = ctm
using outer-TAny assms ct-suffix-cons-ct-list
by simp
then obtain cm' c where cm = cm' @ [c]
                    ct-suffix (take (length cts - 3) cts) (to-ct-list cm')
using ct-suffix-cons-ct-list[of take (length cts - 3) cts [TAny]]
                    assms(3) c-types-agree.simps(2)[of ctm cm]
unfolding ct-list-eq-def to-ct-list-def
by (metis Suc-leI impossible-Cons length-Cons length-map list.exhaust list.size(3)
                    list-all2-conv-all-nth zero-less-Suc)
thus ?thesis
using TAny outer-TAny ct-eq.simps(2)
by blast
next
case (TSome x2)
hence take (length cts - 3) cts @ [TSome x2] = ctm
using outer-TAny assms ct-suffix-cons-ct-list
by simp
then obtain cm' where cm = cm' @ [x2]
                    ct-suffix (take (length cts - 3) cts) (to-ct-list cm')
using ct-suffix-cons-ct-list[of take (length cts - 3) cts [TSome x2]]
                    assms(3) c-types-agree.simps(2)[of ctm cm] ct-list-eq-ts-conv-eq
unfolding ct-list-eq-def to-ct-list-def
by (metis (no-types, opaque-lifting) list.simps(8,9))
thus ?thesis
using TSome outer-TAny
by simp
qed
next
case (TSome x2)
note outer-TSome = TSome
show ?thesis
proof (cases c2)
case TAny
hence take (length cts - 3) cts @ [TSome x2] = ctm
using TSome assms ct-suffix-cons-ct-list
by simp
then obtain cm' where cm = cm' @ [x2]
                    ct-suffix (take (length cts - 3) cts) (to-ct-list cm')
using ct-suffix-cons-ct-list[of take (length cts - 3) cts [TSome x2]]
                    assms(3) c-types-agree.simps(2)[of ctm cm] ct-list-eq-ts-conv-eq
unfolding ct-list-eq-def to-ct-list-def
by (metis (no-types, opaque-lifting) list.simps(8,9))
thus ?thesis
using TSome TAny
by simp
next
case (TSome x3)
hence x-eq:x2 = x3

```

```

    using outer-TSome assms ct-suffix-cons-ct-list
  by (metis checker-type.distinct(3) select-return-top.simps(3))
hence take (length cts - 3) cts @ [TSome x2] = ctm
  using TSome outer-TSome assms ct-suffix-cons-ct-list
  by (metis checker-type.inject(1) select-return-top.simps(3))
then obtain cm' where cm = cm' @ [x2]
      ct-suffix (take (length cts - 3) cts) (to-ct-list cm')
  using ct-suffix-cons-ct-list[of take (length cts - 3) cts [TSome x2]]
      assms(3) c-types-agree.simps(2)[of ctm cm] ct-list-eq-ts-conv-eq
  unfolding ct-list-eq-def to-ct-list-def
  by (metis (no-types, opaque-lifting) list.simps(8,9))
thus ?thesis
  using TSome outer-TSome x-eq
  by simp
qed
qed
end

```

9 Executable Type Checker

theory *Wasm-Checker* **imports** *Wasm-Checker-Types* **begin**

```

fun convert-cond :: t ⇒ t ⇒ sx option ⇒ bool where
  convert-cond t1 t2 sx = ((t1 ≠ t2) ∧ (sx = None) = ((is-float-t t1 ∧ is-float-t
t2)
                                                                    ∨ (is-int-t t1 ∧ is-int-t t2 ∧ (t-length
t1 < t-length t2))))

```

```

fun same-lab-h :: nat list ⇒ (t list) list ⇒ t list ⇒ (t list) option where
  same-lab-h [] - ts = Some ts
| same-lab-h (i#is) lab-c ts = (if i ≥ length lab-c
  then None
  else (if lab-c!i = ts
    then same-lab-h is lab-c (lab-c!i)
    else None))

```

```

fun same-lab :: nat list ⇒ (t list) list ⇒ (t list) option where
  same-lab [] lab-c = None
| same-lab (i#is) lab-c = (if i ≥ length lab-c
  then None
  else same-lab-h is lab-c (lab-c!i))

```

lemma *same-lab-h-conv-list-all*:

```

assumes same-lab-h ils ls ts' = Some ts
shows list-all (λi. i < length ls ∧ ls!i = ts) ils ∧ ts' = ts
using assms by (induction ils) (simp-all split: if-splits)

```

lemma *same-lab-conv-list-all*:

assumes *same-lab ils ls = Some ts*
shows *list-all* ($\lambda i. i < \text{length } ls \wedge ls!i = ts$) *ils*
using *assms* **by** (*induction rule: same-lab.induct*)
(auto dest: same-lab-h-conv-list-all split: if-splits)

lemma *list-all-conv-same-lab-h*:
assumes *list-all* ($\lambda i. i < \text{length } ls \wedge ls!i = ts$) *ils*
shows *same-lab-h ils ls ts = Some ts*
using *assms*
by (*induction ils, simp-all*)

lemma *list-all-conv-same-lab*:
assumes *list-all* ($\lambda i. i < \text{length } ls \wedge ls!i = ts$) (*is@[i]*)
shows *same-lab* (*is@[i]*) *ls = Some ts*
using *assms*
proof (*induction* (*is@[i]*))
case (*Cons a x*)
thus ?*case*
using *list-all-conv-same-lab-h[OF Cons(3)]*
by (*metis option.distinct(1) same-lab.simps(2) same-lab-h.simps(2)*)
qed *auto*

fun *b-e-type-checker* :: *t-context* \Rightarrow *b-e list* \Rightarrow *tf* \Rightarrow *bool*
and *check* :: *t-context* \Rightarrow *b-e list* \Rightarrow *checker-type* \Rightarrow *checker-type*
and *check-single* :: *t-context* \Rightarrow *b-e* \Rightarrow *checker-type* \Rightarrow *checker-type* **where**
b-e-type-checker *C es (tn -> tm) = c-types-agree (check C es (Type tn)) tm*
| *check C es ts = (case es of*
| $\square \Rightarrow ts$
| (*e#es*) \Rightarrow (*case ts of*
| *Bot* \Rightarrow *Bot*
| $- \Rightarrow$ *check C es (check-single C e ts))*)

| *check-single C (C v) ts = type-update ts [] (Type [typeof v])*
| *check-single C (Unop-i t -) ts = (if is-int-t t*
| *then type-update ts [TSome t] (Type [t])*
| *else Bot)*
| *check-single C (Unop-f t -) ts = (if is-float-t t*
| *then type-update ts [TSome t] (Type [t])*
| *else Bot)*
| *check-single C (Binop-i t t -) ts = (if is-int-t t*
| *then type-update ts [TSome t, TSome t] (Type [t])*
| *else Bot)*
| *check-single C (Binop-f t t -) ts = (if is-float-t t*
| *then type-update ts [TSome t, TSome t] (Type [t])*
| *else Bot)*
| *check-single C (Testop t -) ts = (if is-int-t t*
| *then type-update ts [TSome t] (Type [T-i32])*
| *else Bot)*

$| \text{check-single } \mathcal{C} (\text{Relop-}i \ t \ -) \ ts = (\text{if is-int-}t \ t$
 $\qquad\qquad\qquad \text{then type-update } ts \ [T\text{Some } t, \ T\text{Some } t] \ (\text{Type}$
 $[T\text{-}i32])$
 $\qquad\qquad\qquad \text{else Bot})$

$| \text{check-single } \mathcal{C} (\text{Relop-}f \ t \ -) \ ts = (\text{if is-float-}t \ t$
 $\qquad\qquad\qquad \text{then type-update } ts \ [T\text{Some } t, \ T\text{Some } t] \ (\text{Type}$
 $[T\text{-}i32])$
 $\qquad\qquad\qquad \text{else Bot})$

$| \text{check-single } \mathcal{C} (\text{Cvtop } t1 \ \text{Convert } t2 \ sx) \ ts = (\text{if } (\text{convert-cond } t1 \ t2 \ sx)$
 $\qquad\qquad\qquad \text{then type-update } ts \ [T\text{Some } t2] \ (\text{Type } [t1])$
 $\qquad\qquad\qquad \text{else Bot})$

$| \text{check-single } \mathcal{C} (\text{Cvtop } t1 \ \text{Reinterpret } t2 \ sx) \ ts = (\text{if } ((t1 \neq t2) \wedge t\text{-length } t1 =$
 $t\text{-length } t2 \wedge sx = \text{None})$
 $\qquad\qquad\qquad \text{then type-update } ts \ [T\text{Some } t2] \ (\text{Type}$
 $[t1])$
 $\qquad\qquad\qquad \text{else Bot})$

$| \text{check-single } \mathcal{C} (\text{Unreachable}) \ ts = \text{type-update } ts \ [] \ (\text{TopType } [])$
 $| \text{check-single } \mathcal{C} (\text{Nop}) \ ts = ts$
 $| \text{check-single } \mathcal{C} (\text{Drop}) \ ts = \text{type-update } ts \ [T\text{Any}] \ (\text{Type } [])$
 $| \text{check-single } \mathcal{C} (\text{Select}) \ ts = \text{type-update-select } ts$

$| \text{check-single } \mathcal{C} (\text{Block } (tn \ -> \ tm) \ es) \ ts = (\text{if } (\text{b-e-type-checker } (\mathcal{C}(\text{label} := ([tm]$
 $\text{@ } (\text{label } \mathcal{C})))) \ es \ (tn \ -> \ tm))$
 $\qquad\qquad\qquad \text{then type-update } ts \ (\text{to-ct-list } tn) \ (\text{Type } tm)$
 $\qquad\qquad\qquad \text{else Bot})$

$| \text{check-single } \mathcal{C} (\text{Loop } (tn \ -> \ tm) \ es) \ ts = (\text{if } (\text{b-e-type-checker } (\mathcal{C}(\text{label} := ([tn] \ \text{@}$
 $(\text{label } \mathcal{C})))) \ es \ (tn \ -> \ tm))$
 $\qquad\qquad\qquad \text{then type-update } ts \ (\text{to-ct-list } tn) \ (\text{Type } tm)$
 $\qquad\qquad\qquad \text{else Bot})$

$| \text{check-single } \mathcal{C} (\text{If } (tn \ -> \ tm) \ es1 \ es2) \ ts = (\text{if } (\text{b-e-type-checker } (\mathcal{C}(\text{label} := ([tm]$
 $\text{@ } (\text{label } \mathcal{C})))) \ es1 \ (tn \ -> \ tm))$
 $\qquad\qquad\qquad \wedge \ \text{b-e-type-checker } (\mathcal{C}(\text{label} := ([tm] \ \text{@}$
 $(\text{label } \mathcal{C})))) \ es2 \ (tn \ -> \ tm))$
 $\qquad\qquad\qquad \text{then type-update } ts \ (\text{to-ct-list } (tn \ \text{@} [T\text{-}i32]))$
 $(\text{Type } tm)$
 $\qquad\qquad\qquad \text{else Bot})$

$| \text{check-single } \mathcal{C} (\text{Br } i) \ ts = (\text{if } i < \text{length } (\text{label } \mathcal{C})$
 $\qquad\qquad\qquad \text{then type-update } ts \ (\text{to-ct-list } ((\text{label } \mathcal{C})!i)) \ (\text{TopType } [])$
 $\qquad\qquad\qquad \text{else Bot})$

$| \text{check-single } \mathcal{C} (\text{Br-if } i) \ ts = (\text{if } i < \text{length } (\text{label } \mathcal{C})$
 $\qquad\qquad\qquad \text{then type-update } ts \ (\text{to-ct-list } ((\text{label } \mathcal{C})!i \ \text{@} [T\text{-}i32]))$
 $(\text{Type } ((\text{label } \mathcal{C})!i))$

else Bot)

| *check-single* \mathcal{C} (*Br-table is i*) $ts =$ (*case* (*same-lab* ($is@[i]$) (*label* \mathcal{C})) of
 None \Rightarrow *Bot*
 | *Some* $tls \Rightarrow$ *type-update* ts (*to-ct-list* ($tls @ [T-i32]$)))
 (*TopType* $[]$))

| *check-single* \mathcal{C} (*Return*) $ts =$ (*case* (*return* \mathcal{C}) of
 None \Rightarrow *Bot*
 | *Some* $tls \Rightarrow$ *type-update* ts (*to-ct-list* tls) (*TopType* $[]$))

| *check-single* \mathcal{C} (*Call i*) $ts =$ (*if* $i < \text{length}(\text{func-t } \mathcal{C})$
 then (*case* ($(\text{func-t } \mathcal{C})!i$) of
 ($tn \rightarrow tm$) \Rightarrow *type-update* ts (*to-ct-list* tn) (*Type*
 tm))
 else *Bot*)

| *check-single* \mathcal{C} (*Call-indirect i*) $ts =$ (*if* ($\text{table } \mathcal{C} \neq \text{None} \wedge i < \text{length}(\text{types-t } \mathcal{C})$
 then (*case* ($(\text{types-t } \mathcal{C})!i$) of
 ($tn \rightarrow tm$) \Rightarrow *type-update* ts (*to-ct-list*
 ($tn@[T-i32]$)) (*Type* tm))
 else *Bot*)

| *check-single* \mathcal{C} (*Get-local i*) $ts =$ (*if* $i < \text{length}(\text{local } \mathcal{C})$
 then *type-update* $ts []$ (*Type* $[(\text{local } \mathcal{C})!i]$)
 else *Bot*)

| *check-single* \mathcal{C} (*Set-local i*) $ts =$ (*if* $i < \text{length}(\text{local } \mathcal{C})$
 then *type-update* $ts [T\text{Some } ((\text{local } \mathcal{C})!i)]$ (*Type* $[]$)
 else *Bot*)

| *check-single* \mathcal{C} (*Tee-local i*) $ts =$ (*if* $i < \text{length}(\text{local } \mathcal{C})$
 then *type-update* $ts [T\text{Some } ((\text{local } \mathcal{C})!i)]$ (*Type* $[(\text{local } \mathcal{C})!i]$)
 else *Bot*)

| *check-single* \mathcal{C} (*Get-global i*) $ts =$ (*if* $i < \text{length}(\text{global } \mathcal{C})$
 then *type-update* $ts []$ (*Type* $[tg-t ((\text{global } \mathcal{C})!i)]$)
 else *Bot*)

| *check-single* \mathcal{C} (*Set-global i*) $ts =$ (*if* $i < \text{length}(\text{global } \mathcal{C}) \wedge is\text{-mut}(\text{global } \mathcal{C} ! i)$
 then *type-update* $ts [T\text{Some } (tg-t ((\text{global } \mathcal{C})!i))]$
 (*Type* $[]$)
 else *Bot*)

| *check-single* \mathcal{C} (*Load t tp-sx a off*) $ts =$ (*if* ($\text{memory } \mathcal{C} \neq \text{None} \wedge \text{load-store-t-bounds}$
 a (*option-projl* $tp\text{-sx}$) t
 then *type-update* $ts [T\text{Some } T\text{-i32}]$ (*Type* $[t]$)
 else *Bot*)

```

| check-single C (Store t tp a off) ts = (if (memory C) ≠ None ∧ load-store-t-bounds
a tp t
                                     then type-update ts [TSome T-i32, TSome t]
(Type []))
                                     else Bot)

| check-single C Current-memory ts = (if (memory C) ≠ None
                                     then type-update ts [] (Type [T-i32])
                                     else Bot)

| check-single C Grow-memory ts = (if (memory C) ≠ None
                                     then type-update ts [TSome T-i32] (Type [T-i32])
                                     else Bot)

```

end

10 Correctness of Type Checker

theory *Wasm-Checker-Properties* **imports** *Wasm-Checker* *Wasm-Properties* **begin**

10.1 Soundness

lemma *b-e-check-single-type-sound*:

```

assumes type-update (Type x1) (to-ct-list t-in) (Type t-out) = Type x2
          c-types-agree (Type x2) tm
          C ⊢ [e] : (t-in -> t-out)
shows ∃ tn. c-types-agree (Type x1) tn ∧ C ⊢ [e] : (tn -> tm)
using assms(2) b-e-typing.intros(35)[OF assms(3)] type-update-type[OF assms(1)]
by auto

```

lemma *b-e-check-single-top-sound*:

```

assumes type-update (TopType x1) (to-ct-list t-in) (Type t-out) = TopType x2
          c-types-agree (TopType x2) tm
          C ⊢ [e] : (t-in -> t-out)
shows ∃ tn. c-types-agree (TopType x1) tn ∧ C ⊢ [e] : (tn -> tm)

```

proof –

```

obtain t-ag where t-ag-def:ct-suffix (to-ct-list t-out) x2
                  tm = t-ag @ t-out
                  c-types-agree (TopType x1) (t-ag @ t-in)
using type-update-top-top[OF assms(1,2)]
by fastforce
hence C ⊢ [e] : (t-ag@t-in -> t-ag@t-out)
using b-e-typing.intros(35)[OF assms(3)]
by fastforce
thus ?thesis
using t-ag-def
by fastforce

```

qed

lemma *b-e-check-single-top-not-bot-sound*:

assumes *type-update* *ts* (*to-ct-list* *t-in*) (*TopType* []) = *ts'*

ts \neq *Bot*

ts' \neq *Bot*

shows $\exists tn. c\text{-types-agree } ts \ tn \wedge \text{suffix } t\text{-in } tn$

proof (*cases* *ts*)

case (*TopType* *x1*)

then obtain *t-int* **where** *consume* (*TopType* *x1*) (*to-ct-list* *t-in*) = *t-int* *t-int* \neq *Bot*

using *assms*(1,2,3)

by *fastforce*

thus *?thesis*

using *TopType* *ct-suffix-ct-list-eq-exists* *ct-suffix-ts-conv-suffix*

unfolding *consume.simps*

by (*metis* *append-Nil* *c-types-agree.simps*(2) *ct-suffix-def*)

next

case (*Type* *x2*)

then obtain *t-int* **where** *consume* (*Type* *x2*) (*to-ct-list* *t-in*) = *t-int* *t-int* \neq *Bot*

using *assms*(1,2,3)

by *fastforce*

thus *?thesis*

using *c-types-agree-id* *Type* *consume-type* *suffixI* *ct-suffix-ts-conv-suffix*

by *fastforce*

next

case *Bot*

thus *?thesis*

using *assms*(2)

by *simp*

qed

lemma *b-e-check-single-type-not-bot-sound*:

assumes *type-update* *ts* (*to-ct-list* *t-in*) (*Type* *t-out*) = *ts'*

ts \neq *Bot*

ts' \neq *Bot*

c-types-agree *ts'* *tm*

$\mathcal{C} \vdash [e] : (t\text{-in} \rightarrow t\text{-out})$

shows $\exists tn. c\text{-types-agree } ts \ tn \wedge \mathcal{C} \vdash [e] : (tn \rightarrow tm)$

using *assms* *b-e-check-single-type-sound*

proof (*cases* *ts*)

case (*TopType* *x1*)

then obtain *x1'* **where** *x-def:TopType* *x1'* = *ts'*

using *assms*

by (*simp*, *metis* (*full-types*) *produce.simps*(1) *produce.simps*(6))

thus *?thesis*

using *assms* *b-e-check-single-top-sound* *TopType*

by *fastforce*

next

```

case (Type x2)
then obtain x2' where x-def:Type x2' = ts'
  using assms
  by (simp, metis (full-types) produce.simps(2) produce.simps(6))
thus ?thesis
  using assms b-e-check-single-type-sound Type
  by fastforce
next
case Bot
thus ?thesis
  using assms(2)
  by simp
qed

lemma b-e-check-single-sound-unop-testop-cvtop:
  assumes check-single C e tn' = tm'
    ((e = (Unop-i t uu) ∨ e = (Testop t uw)) ∧ is-int-t t)
    ∨ (e = (Unop-f t uw) ∧ is-float-t t)
    ∨ (e = (Cvtop t1 Convert t sx) ∧ convert-cond t1 t sx)
    ∨ (e = (Cvtop t1 Reinterpret t sx) ∧ ((t1 ≠ t) ∧ t-length t1 = t-length t
  ∧ sx = None))
    c-types-agree tm' tm
    tn' ≠ Bot
    tm' ≠ Bot
  shows ∃ tn. c-types-agree tn' tn ∧ C ⊢ [e] : (tn -> tm)
proof -
  have (e = (Cvtop t1 Convert t sx) ⇒ convert-cond t1 t sx)
    using assms(2)
    by simp
  hence temp0:(e = (Cvtop t1 Convert t sx) ⇒ (type-update tn' [TSome t] (Type
  [arity-1-result e]) = tm'))
    using assms(1,5) arity-1-result-def
    by (simp del: convert-cond.simps)
  have temp1:(e = (Cvtop t1 Reinterpret t sx) ⇒ (type-update tn' [TSome t]
  (Type [arity-1-result e]) = tm'))
    using assms(1,2,5) arity-1-result-def
    by simp
  have 1:type-update tn' (to-ct-list [t]) (Type [arity-1-result e]) = tm'
    using assms arity-1-result-def
    unfolding to-ct-list-def
    apply (simp del: convert-cond.simps)
  apply (metis (no-types, lifting) temp0 temp1 b-e.simps(978,979,982) check-single.simps(2)
  check-single.simps(3) check-single.simps(6) type-update.simps)
  done
  have C ⊢ [e] : ([t] -> [arity-1-result e])
    using assms(2) b-e-typing.intros(2,3,6,9,10)
    unfolding arity-1-result-def
    by fastforce
  thus ?thesis

```

using *b-e-check-single-type-not-bot-sound*[*OF 1 assms(4,5,3)*]
by *fastforce*
qed

lemma *b-e-check-single-sound-binop-relop*:

assumes *check-single* \mathcal{C} e $tn' = tm'$
 $((e = \text{Binop-i } t \text{ iop} \wedge \text{is-int-t } t)$
 $\vee (e = \text{Binop-f } t \text{ fop} \wedge \text{is-float-t } t)$
 $\vee (e = \text{Relop-i } t \text{ irop} \wedge \text{is-int-t } t)$
 $\vee (e = \text{Relop-f } t \text{ frop} \wedge \text{is-float-t } t))$
c-types-agree $tm' \text{ } tm$
 $tn' \neq \text{Bot}$
 $tm' \neq \text{Bot}$

shows $\exists tn. \text{c-types-agree } tn' \text{ } tn \wedge \mathcal{C} \vdash [e] : (tn \rightarrow tm)$

proof –

have *type-update* tn' (*to-ct-list* $[t,t]$) (*Type* [*arity-2-result* e]) = tm'
using *assms* *arity-2-result-def*
unfolding *to-ct-list-def*
by *auto*

moreover

have $\mathcal{C} \vdash [e] : ([t,t] \rightarrow [\text{arity-2-result } e])$
using *assms(2)* *b-e-typing.intros(4,5,7,8)*
unfolding *arity-2-result-def*
by *fastforce*

ultimately

show *?thesis*
using *b-e-check-single-type-not-bot-sound*[*OF - assms(4,5,3)*]
by *fastforce*

qed

lemma *b-e-type-checker-sound*:

assumes *b-e-type-checker* \mathcal{C} es $(tn \rightarrow tm)$
shows $\mathcal{C} \vdash es : (tn \rightarrow tm)$

proof –

fix $e \text{ } tn'$

have *b-e-type-checker* \mathcal{C} es $(tn \rightarrow tm) \implies$
 $\mathcal{C} \vdash es : (tn \rightarrow tm)$

and $\bigwedge tm' \text{ } tm.$

$\text{check } \mathcal{C} \text{ } es \text{ } tn' = tm' \implies$
 $\text{c-types-agree } tm' \text{ } tm \implies$
 $\exists tn. \text{c-types-agree } tn' \text{ } tn \wedge \mathcal{C} \vdash es : (tn \rightarrow tm)$

and $\bigwedge tm' \text{ } tm.$

$\text{check-single } \mathcal{C} \text{ } e \text{ } tn' = tm' \implies$
 $\text{c-types-agree } tm' \text{ } tm \implies$
 $tn' \neq \text{Bot} \implies$
 $tm' \neq \text{Bot} \implies$

$\exists tn. \text{c-types-agree } tn' \text{ } tn \wedge \mathcal{C} \vdash [e] : (tn \rightarrow tm)$

proof (*induction rule: b-e-type-checker-check-check-single.induct*)

case (*1* \mathcal{C} es $tn' \text{ } tm$)

```

thus ?case
  by simp
next
case (2 C es' ts)
show ?case
proof (cases es')
  case Nil
  thus ?thesis
  using 2(5,6)
  by (simp add: b-e-type-empty)
next
case (Cons e es)
  thus ?thesis
  proof (cases ts)
  case (TopType x1)
  have check-expand:check C es (check-single C e ts) = tm'
    using 2(5,6) TopType Cons
    by simp
  obtain ts' where ts'-def:check-single C e ts = ts'
    by blast
  obtain t-int where t-int-def:C ⊢ es : (t-int -> tm)
    using 2(2)[OF Cons TopType check-expand 2(6)] ts'-def
    by blast
  obtain t-int' where c-types-agree ts t-int' C ⊢ [e] : (t-int' -> t-int)
    using 2(1)[OF Cons - ts'-def] TopType c-types-agree.simps(3) t-int-def(2)
    by blast
  thus ?thesis
  using t-int-def(1) b-e-type-comp-conc Cons
  by fastforce
next
case (Type x2)
  have check-expand:check C es (check-single C e ts) = tm'
    using 2(5,6) Type Cons
    by simp
  obtain ts' where ts'-def:check-single C e ts = ts'
    by blast
  obtain t-int where t-int-def:C ⊢ es : (t-int -> tm)
    using 2(4)[OF Cons Type check-expand 2(6)] ts'-def
    by blast
  obtain t-int' where c-types-agree ts t-int' C ⊢ [e] : (t-int' -> t-int)
    using 2(3)[OF Cons - ts'-def] Type c-types-agree.simps(3) t-int-def(2)
    by blast
  thus ?thesis
  using t-int-def(1) b-e-type-comp-conc Cons
  by fastforce
next
case Bot

```

```

    then show ?thesis
      using 2(5,6) Cons
      by auto
  qed
qed
next
case (3 C v ts)
hence type-update ts [] (Type [typeof v]) = tm'
  by simp
moreover
have C ⊢ [C v] : ([] -> [typeof v])
  using b-e-typing.intros(1)
  by blast
ultimately
show ?case
  using b-e-check-single-type-not-bot-sound[OF - 3(3,4,2)]
  by (metis list.simps(8) to-ct-list-def)
next
case (4 C t uu ts)
hence is-int-t t
  by (simp, meson)
thus ?case
  using b-e-check-single-sound-unop-testop-cvtop 4
  by fastforce
next
case (5 C t uv ts)
hence is-float-t t
  by (simp, meson)
thus ?case
  using b-e-check-single-sound-unop-testop-cvtop 5
  by fastforce
next
case (6 C t uw ts)
hence is-int-t t
  by (simp, meson)
thus ?case
  using b-e-check-single-sound-binop-relop 6
  by fastforce
next
case (7 C t ux ts)
hence is-float-t t
  by (simp, meson)
thus ?case
  using b-e-check-single-sound-binop-relop 7
  by fastforce
next
case (8 C t uy ts)
hence is-int-t t
  by (simp, meson)

```

```

thus ?case
  using b-e-check-single-sound-unop-testop-cvtop 8
  by fastforce
next
  case (9 C t uz ts)
  hence is-int-t t
  by (simp, meson)
  thus ?case
  using b-e-check-single-sound-binop-relop 9
  by fastforce
next
  case (10 C t va ts)
  hence is-float-t t
  by (simp, meson)
  thus ?case
  using b-e-check-single-sound-binop-relop 10
  by fastforce
next
  case (11 C t1 t2 sx ts)
  hence convert-cond t1 t2 sx
  by (simp del: convert-cond.simps, meson)
  thus ?case
  using b-e-check-single-sound-unop-testop-cvtop 11
  by fastforce
next
  case (12 C t1 t2 sx ts)
  hence t1 ≠ t2 ∧ t-length t1 = t-length t2 ∧ sx = None
  by (simp, presburger)
  thus ?case
  using b-e-check-single-sound-unop-testop-cvtop 12
  by fastforce
next
  case (13 C ts)
  thus ?case
  using b-e-typing.intros(11) c-types-agree-not-bot-exists
  by blast
next
  case (14 C ts)
  thus ?case
  using b-e-typing.intros(12,35)
  by fastforce
next
  case (15 C ts)
  thus ?case
  proof (cases ts)
  case (TopType x1)
  thus ?thesis
  proof (cases x1 rule: List.rev-cases)
  case Nil

```

```

have C ⊢ [Drop] : (tm@[T-i32] -> tm)
  using b-e-typing.intros(13,35)
  by fastforce
thus ?thesis
  using c-types-agree-top1 Nil TopType
  by fastforce
next
case (snoc ys y)
  hence temp1:(consume (TopType (ys@[y])) [TAny]) = tm'
    using 15 TopType type-update-empty
    by (metis check-single.simps(13))
  hence temp2:c-types-agree (TopType ys) tm
    using consume-top-geq[OF temp1] 15(2,3,4)
    by (metis Suc-leI add-diff-cancel-right' append-eq-conv-conj consume.simps(2)
      ct-suffix-def length-Cons length-append list.size(3) trans-le-add2
      zero-less-Suc)
  obtain t where ct-list-eq [y] (to-ct-list [t])
    using ct-list-eq-exists
    unfolding ct-list-eq-def to-ct-list-def list-all2-map2
    by (metis list-all2-Cons1 list-all2-Nil)
  hence c-types-agree ts (tm@[t])
    using temp2 ct-suffix-extend-ct-list-eq snoc TopType
    by (simp add: to-ct-list-def)
  thus ?thesis
    using b-e-typing.intros(13,35)
    by fastforce
  qed
next
case (Type x2)
thus ?thesis
proof (cases x2 rule: List.rev-cases)
  case Nil
  hence (consume (Type []) [TAny]) = tm'
    using 15 Type type-update-empty
    by fastforce
  thus ?thesis
    using 15(4) ct-list-eq-def ct-suffix-def to-ct-list-def
    by simp
next
case (snoc ys y)
  hence temp1:(consume (Type (ys@[y])) [TAny]) = tm'
    using 15 Type type-update-empty
    by (metis check-single.simps(13))
  hence temp2:c-types-agree (Type ys) tm
    using 15(2,3,4) ct-suffix-def
  by (simp, metis One-nat-def butlast-conv-take butlast-snoc c-types-agree.simps(1)
    length-Cons list.size(3))
  obtain t where ct-list-eq [TSome y] (to-ct-list [t])

```

```

      using ct-list-eq-exists
      unfolding ct-list-eq-def to-ct-list-def list-all2-map2
      by (metis list-all2-Cons1 list-all2-Nil)
      hence c-types-agree ts (tm@[t])
      using temp2 ct-suffix-extend-ct-list-eq snoc Type
      by (simp add: ct-list-eq-def to-ct-list-def)
      thus ?thesis
      using b-e-typing.intros(13,35)
      by fastforce
    qed
  qed simp
next
case (16 C ts)
thus ?case
proof (cases ts)
  case (TopType x1)
  consider
    (1) length x1 = 0
  | (2) length x1 = 1
  | (3) length x1 = 2
  | (4) length x1 ≥ 3
  by linarith
thus ?thesis
proof (cases)
  case 1
  hence tm' = TopType [TAny]
  using TopType 16
  by simp
  then obtain t'' tm'' where tm-def:tm = tm''@[t'']
  using 16(2) ct-suffix-def
  by (simp, metis Nil-is-append-conv append-butlast-last-id checker-type.inject(1)
    ct-prefixI ct-prefix-nil(2) produce.simps(1) produce-nil)
  have C ⊢ [Select] : ([t'',t'',T-i32] -> [t''])
  using b-e-typing.intros(14)
  by blast
  thus ?thesis
  using TopType 16 1 tm-def b-e-typing.intros(35) c-types-agree.simps(2)
  c-types-agree-top1
  by fastforce
next
case 2
  have type-update-select (TopType x1) = tm'
  using 16 TopType
  unfolding check-single.simps
  by simp
  hence x1-def:ct-list-eq x1 [TSome T-i32] tm' = TopType [TAny]
  using type-update-select-length1[OF - 2 16(4)]
  by simp-all
  then obtain t'' tm'' where tm-def:tm = tm''@[t'']

```

```

    using 16(2) ct-suffix-def
  by (metis Nil-is-append-conv append-butlast-last-id c-types-agree.simps(2)
ct-prefixI
      ct-prefix-nil(2) list.simps(8) to-ct-list-def)
  have c-types-agree (TopType x1) ((tm''@[t'',t''])@[T-i32])
    using x1-def(1)
    by (metis c-types-agree-top2 list.simps(8,9) to-ct-list-def)
  thus ?thesis
    using TopType b-e-typing.intros(14,35) tm-def
    by auto
next
case 3
have type-update-select (TopType x1) = tm'
  using 16 TopType
  unfolding check-single.simps
  by simp
then obtain ct1 ct2 where x1-def:x1 = [ct1, ct2]
      ct-eq ct2 (TSome T-i32)
      tm' = TopType [ct1]
  using type-update-select-length2[OF - 3 16(4)]
  by blast
then obtain t'' tm'' where tm-def:tm = tm''@[t'']
      ct-list-eq [ct1] [(TSome t'')]
  using 16(2) c-types-agree-imp-ct-list-eq[of [ct1] tm]
  by (metis append-Nil2 append-butlast-last-id append-eq-append-conv-if
append-eq-conv-conj
      ct-list-eq-length diff-Suc-1 length-Cons length-butlast length-map
list.simps(8,9) list.size(3) nat.distinct(2) to-ct-list-def)
hence ct-list-eq x1 (to-ct-list [ t'', T-i32])
  using x1-def(1,2)
  unfolding ct-list-eq-def to-ct-list-def
  by fastforce
hence c-types-agree (TopType x1) ((tm''@[t''])@[t'',T-i32])
  using c-types-agree-top2
  by blast
thus ?thesis
  using TopType b-e-typing.intros(14,35) tm-def
  by auto
next
case 4
then obtain nat where nat-def:length x1 = Suc (Suc (Suc nat))
  by (metis add-eq-if diff-Suc-1 le-Suc-ex numeral-3-eq-3 nat.distinct(2))
hence tm'-def:type-update-select (TopType x1) = tm'
  using 16 TopType
  by simp
then obtain tm-int where (select-return-top x1
      (x1 ! (length x1 - 2))
      (x1 ! (length x1 - 3))) = tm-int
      tm-int ≠ Bot

```

```

using nat-def 16(4)
unfolding type-update-select.simps
by fastforce
then obtain x2 where x2-def:(select-return-top x1
  (x1 ! (length x1 - 2))
  (x1 ! (length x1 - 3))) = TopType x2

using select-return-top-exists
by fastforce
have ct-suffix x1 [TAny, TAny, TSome T-i32] ∨ ct-suffix [TAny, TAny,
TSome T-i32] x1
using tm'-def nat-def 16(4)
by (simp, metis (full-types) produce.simps(6))
hence tm'-eq:tm' = TopType x2
using tm'-def nat-def 16(4) x2-def
by force
then obtain cts' ct1 ct2 ct3 where cts'-def:x1 = cts'@[ct1, ct2, ct3]
  ct-eq ct3 (TSome T-i32)

using type-update-select-length3 tm'-def 4
by blast
then obtain c' cm' where tm-def:tm = cm'@[c']
  ct-suffix cts' (to-ct-list cm')
  ct-eq (x1 ! (length x1 - 2)) (TSome c')
  ct-eq (x1 ! (length x1 - 3)) (TSome c')

using select-return-top-ct-eq[OF x2-def 4] tm'-eq 4 16(2)
by fastforce
then obtain as bs where cm'-def:cm' = as@bs
  ct-list-eq (to-ct-list bs) cts'

using ct-list-eq-cons-ct-list1 ct-list-eq-ts-conv-eq
by (metis ct-suffix-def to-ct-list-append(2))
hence ct-eq ct1 (TSome c')
  ct-eq ct2 (TSome c')
using cts'-def tm-def
apply simp-all
apply (metis append.assoc append-Cons append-Nil length-append-singleton
nth-append-length)
done
hence c-types-agree ts (cm'@[c',c',T-i32])
using c-types-agree-top2[of - - as] cm'-def(1) TopType
  ct-list-eq-concat[OF ct-list-eq-commute[OF cm'-def(2)]] cts'-def
unfolding to-ct-list-def ct-list-eq-def
by fastforce
thus ?thesis
using b-e-typing.intros(14,35) tm-def
by auto
qed
next

case (Type x2)
hence x2-cond:(length x2 ≥ 3 ∧ (x2!(length x2-2)) = (x2!(length x2-3)))

```

```

    using 16
    by (simp, meson)
  hence tm'-def:consume (Type x2) [TAny, TSome T-i32] = tm'
    using 16 Type
    by simp
  obtain ts' ts'' where cts-def:x2 = ts'@ ts'' length ts'' = 3
    using x2-cond
    by (metis append-take-drop-id diff-diff-cancel length-drop)
  then obtain t1 ts''2 where ts'' = t1#ts''2 length ts''2 = Suc (Suc 0)
    using List.length-Suc-conv[of ts' Suc (Suc 0)]
    by (metis length-Suc-conv numeral-3-eq-3)
  then obtain t2 t3 where ts'' = [t1,t2,t3]
    using List.length-Suc-conv[of ts''2 Suc 0]
    by (metis length-0-conv length-Suc-conv)
  hence cts-def2:x2 = ts'@ [t1,t2,t3]
    using cts-def
    by simp
  have ts'-suffix:ct-suffix [TAny, TSome T-i32] (to-ct-list (ts' @ [t1, t2, t3]))
    using tm'-def 16(4)
    by (simp, metis cts-def2)
  hence tm'-def:tm' = Type (ts'@[t1])
    using tm'-def 16(4) cts-def2
    by simp
  obtain as bs where (to-ct-list (ts' @ [t1])) @ (to-ct-list ([t2, t3])) = as@bs
    ct-list-eq bs [TAny, TSome T-i32]
    using ts'-suffix
    unfolding ct-suffix-def to-ct-list-def
    by fastforce
  hence t3 = T-i32
    unfolding to-ct-list-def ct-list-eq-def
  by (metis (no-types, lifting) Nil-is-map-conv append-eq-append-conv ct-eq.simps(1)
    length-Cons list.sel(1,3) list.simps(9) list-all2-Cons2 list-all2-lengthD)
  moreover
  have t1 = t2
    using x2-cond cts-def2
  by (simp, metis append.left-neutral append-Cons append-assoc length-append-singleton
    nth-append-length)
  ultimately
  have c-types-agree (Type x2) ((ts'@[t1,t1])@[T-i32])
    using cts-def2
    by simp
  thus ?thesis
    using b-e-typing.intros(14,35) Type tm'-def 16(2)
    by fastforce
qed simp
next
case (17 C tn'' tm'' es ts)
hence type-update ts (to-ct-list tn'') (Type tm'') = tm'
  by auto

```

```

moreover
have (b-e-type-checker (C(label := ([tm''] @ (label C)))) es (tn'' -> tm''))
  using 17
  by (simp, meson)
hence C ⊢ [Block (tn'' -> tm'') es] : (tn'' -> tm'')
  using b-e-typing.intros(15)[OF - 17(1)]
  by blast
ultimately
show ?case
  using b-e-check-single-type-not-bot-sound[OF - 17(4,5,3)]
  by blast
next
case (18 C tn'' tm'' es ts)
hence type-update ts (to-ct-list tn'') (Type tm'') = tm'
  by auto
moreover
have (b-e-type-checker (C(label := ([tm''] @ (label C)))) es (tn'' -> tm''))
  using 18
  by (simp, meson)
hence C ⊢ [Loop (tn'' -> tm'') es] : (tn'' -> tm'')
  using b-e-typing.intros(16)[OF - 18(1)]
  by blast
ultimately
show ?case
  using b-e-check-single-type-not-bot-sound[OF - 18(4,5,3)]
  by blast
next
case (19 C tn'' tm'' es1 es2 ts)
hence type-update ts (to-ct-list (tn''@[T-i32])) (Type tm'') = tm'
  by auto
moreover
have (b-e-type-checker (C(label := ([tm''] @ (label C)))) es1 (tn'' -> tm''))
  (b-e-type-checker (C(label := ([tm''] @ (label C)))) es2 (tn'' -> tm''))
  using 19
  by (simp, meson)+
hence C ⊢ [If (tn'' -> tm'') es1 es2] : (tn''@[T-i32] -> tm'')
  using b-e-typing.intros(17)[OF - 19(1,2)]
  by blast
ultimately
show ?case
  using b-e-check-single-type-not-bot-sound[OF - 19(5,6,4)]
  by blast
next
case (20 C i ts)
hence type-update ts (to-ct-list ((label C)!i)) (TopType []) = tm'
  by auto
moreover
have i < length (label C)
  using 20

```

```

    by (simp, meson)
  ultimately
  show ?case
    using b-e-check-single-top-not-bot-sound[OF - 20(3,4)]
           b-e-typing.intros(18)
           b-e-typing.intros(35)
    by (metis suffix-def)
next
  case (21 C i ts)
  hence type-update ts (to-ct-list ((label C)!i @ [T-i32])) (Type ((label C)!i)) =
tm'
    by auto
  moreover
  have i < length (label C)
    using 21
    by (simp, meson)
  hence C ⊢ [Br-if i] : ((label C)!i @ [T-i32] -> (label C)!i)
    using b-e-typing.intros(19)
    by fastforce
  ultimately
  show ?case
    using b-e-check-single-type-not-bot-sound[OF - 21(3,4,2)]
    by fastforce
next
  case (22 C is i ts)
  then obtain tls where tls-def:(same-lab (is@[i]) (label C)) = Some tls
    by fastforce
  hence type-update ts (to-ct-list (tls @ [T-i32])) (TopType []) = tm'
    using 22
    by simp
  thus ?case
    using b-e-check-single-top-not-bot-sound[OF - 22(3,4)]
           b-e-typing.intros(20)[OF same-lab-conv-list-all[OF tls-def]]
           b-e-typing.intros(35)
    by (metis suffix-def)
next
  case (23 C ts)
  then obtain ts-r where (return C) = Some ts-r
    by fastforce
  moreover
  hence type-update ts (to-ct-list ts-r) (TopType []) = tm'
    using 23
    by simp
  ultimately
  show ?case
    using b-e-check-single-top-not-bot-sound[OF - 23(3,4)]
           b-e-typing.intros(21,35)
    by (metis suffix-def)
next

```

```

case (24 C i ts)
obtain tn'' tm'' where func-def:(func-t C)!i = (tn'' -> tm'')
  using tf.exhaust
  by blast
hence type-update ts (to-ct-list tn'') (Type tm'') = tm'
  using 24
  by auto
moreover
have i < length (func-t C)
  using 24
  by (simp, meson)
hence C ⊢ [Call i] : (tn'' -> tm'')
  using b-e-typing.intros(22) func-def
  by fastforce
ultimately
show ?case
  using b-e-check-single-type-not-bot-sound[OF - 24(3,4,2)]
  by fastforce
next
case (25 C i ts)
obtain tn'' tm'' where type-def:(types-t C)!i = (tn'' -> tm'')
  using tf.exhaust
  by blast
hence type-update ts (to-ct-list (tn''@[T-i32])) (Type tm'') = tm'
  using 25
  by auto
moreover
have (table C) ≠ None ∧ i < length (types-t C)
  using 25
  by (simp, meson)
hence C ⊢ [Call-indirect i] : (tn''@[T-i32] -> tm'')
  using b-e-typing.intros(23) type-def
  by fastforce
ultimately
show ?case
  using b-e-check-single-type-not-bot-sound[OF - 25(3,4,2)]
  by fastforce
next
case (26 C i ts)
hence type-update ts [] (Type [(local C)!i]) = tm'
  by auto
moreover
have i < length (local C)
  using 26
  by (simp, meson)
hence C ⊢ [Get-local i] : ([] -> [(local C)!i])
  using b-e-typing.intros(24)
  by fastforce
ultimately

```

```

show ?case
  using b-e-check-single-type-not-bot-sound[OF - 26(3,4,2)]
  unfolding to-ct-list-def
  by (metis list.map-disc-iff)
next
  case (27 C i ts)
  hence type-update ts (to-ct-list [(local C)!i]) (Type []) = tm'
    unfolding to-ct-list-def
    by auto
  moreover
  have i < length (local C)
    using 27
    by (simp, meson)
  hence C ⊢ [Set-local i] : ((local C)!i -> [])
    using b-e-typing.intros(25)
    by fastforce
  ultimately
  show ?case
    using b-e-check-single-type-not-bot-sound[OF - 27(3,4,2)]
    by fastforce
next
  case (28 C i ts)
  hence type-update ts (to-ct-list [(local C)!i]) (Type [(local C)!i]) = tm'
    unfolding to-ct-list-def
    by auto
  moreover
  have i < length (local C)
    using 28
    by (simp, meson)
  hence C ⊢ [Tee-local i] : ((local C)!i -> [(local C)!i])
    using b-e-typing.intros(26)
    by fastforce
  ultimately
  show ?case
    using b-e-check-single-type-not-bot-sound[OF - 28(3,4,2)]
    by fastforce
next
  case (29 C i ts)
  hence type-update ts [] (Type [tg-t ((global C)!i)]) = tm'
    by auto
  moreover
  have i < length (global C)
    using 29
    by (simp, meson)
  hence C ⊢ [Get-global i] : ([] -> [tg-t ((global C)!i)])
    using b-e-typing.intros(27)
    by fastforce
  ultimately
  show ?case

```

```

    using b-e-check-single-type-not-bot-sound[OF - 29(3,4,2)]
    unfolding to-ct-list-def
    by (metis list.map-disc-iff)
next
case (30 C i ts)
hence type-update ts (to-ct-list [tg-t ((global C)!i)]) (Type []) = tm'
  unfolding to-ct-list-def
  by auto
moreover
have i < length (global C) ∧ is-mut (global C ! i)
  using 30
  by (simp, meson)
then obtain t where (global C ! i) = (tg-mut = T-mut, tg-t = t) i < length
(global C)
  unfolding is-mut-def
  by (cases global C ! i, auto)
hence C ⊢ [Set-global i] : ([tg-t (global C ! i)] -> [])
  using b-e-typing.intros(28)[of i C tg-t (global C ! i)]
  unfolding is-mut-def tg-t-def
  by fastforce
ultimately
show ?case
  using b-e-check-single-type-not-bot-sound[OF - 30(3,4,2)]
  by fastforce
next
case (31 C t tp-sx a off ts)
hence type-update ts (to-ct-list [T-i32]) (Type [t]) = tm'
  unfolding to-ct-list-def
  by auto
moreover
have (memory C) ≠ None ∧ load-store-t-bounds a (option-proj1 tp-sx) t
  using 31
  by (simp, meson)
hence C ⊢ [Load t tp-sx a off] : ([T-i32] -> [t])
  using b-e-typing.intros(29)
  by fastforce
ultimately
show ?case
  using b-e-check-single-type-not-bot-sound[OF - 31(3,4,2)]
  by fastforce
next
case (32 C t tp a off ts)
hence type-update ts (to-ct-list [T-i32,t]) (Type []) = tm'
  unfolding to-ct-list-def
  by auto
moreover
have (memory C) ≠ None ∧ load-store-t-bounds a tp t
  using 32
  by (simp, meson)

```

```

hence  $\mathcal{C} \vdash [\text{Store } t \text{ tp } a \text{ off}] : ([T\text{-}i32, t] \rightarrow [])$ 
  using b-e-typing.intros(30)
  by fastforce
ultimately
show ?case
  using b-e-check-single-type-not-bot-sound[OF - 32(3,4,2)]
  by fastforce
next
case (33  $\mathcal{C}$   $ts$ )
hence type-update  $ts [] (\text{Type } [T\text{-}i32]) = tm'$ 
  by auto
moreover
have memory  $\mathcal{C} \neq \text{None}$ 
  using 33
  by (simp, meson)
hence  $\mathcal{C} \vdash [\text{Current-memory}] : ([[] \rightarrow [T\text{-}i32])$ 
  using b-e-typing.intros(31)
  by fastforce
ultimately
show ?case
  using b-e-check-single-type-not-bot-sound[OF - 33(3,4,2)]
  unfolding to-ct-list-def
  by (metis list.map-disc-iff)
next
case (34  $\mathcal{C}$   $ts$ )
hence type-update  $ts (\text{to-ct-list } [T\text{-}i32]) (\text{Type } [T\text{-}i32]) = tm'$ 
  unfolding to-ct-list-def
  by auto
moreover
have memory  $\mathcal{C} \neq \text{None}$ 
  using 34
  by (simp, meson)
hence  $\mathcal{C} \vdash [\text{Grow-memory}] : ([T\text{-}i32] \rightarrow [T\text{-}i32])$ 
  using b-e-typing.intros(32)
  by fastforce
ultimately
show ?case
  using b-e-check-single-type-not-bot-sound[OF - 34(3,4,2)]
  by fastforce
qed
thus ?thesis
  using assms
  by simp
qed

```

10.2 Completeness

lemma *check-single-imp*:
assumes *check-single* $\mathcal{C} \ e \ \text{ctn} = \text{ctm}$

```

       $ctm \neq Bot$ 
shows  $check\_single\ C\ e = id$ 
       $\vee check\_single\ C\ e = (\lambda ctn. type\_update\_select\ ctn)$ 
       $\vee (\exists\ cons\ prods. (check\_single\ C\ e = (\lambda ctn. type\_update\ ctn\ cons\ prods)))$ 
proof –
  have  $True$ 
  and  $True$ 
  and  $check\_single\ C\ e\ ctn = ctm \implies$ 
     $ctm \neq Bot \implies$ 
     $?thesis$ 
proof (induction rule: b-e-type-checker-check-check-single.induct)
  case (1  $C\ es\ tn\ tm$ )
  thus  $?case$ 
  by  $simp$ 
next
  case (2  $C\ es\ ts$ )
  thus  $?case$ 
  by  $simp$ 
next
  case (3  $C\ v\ ts$ )
  thus  $?case$ 
  by  $fastforce$ 
next
  case (4  $C\ t\ uu\ ts$ )
  thus  $?case$ 
  by ( $simp, meson\ assms(2)\ type\_update.simps$ )
next
  case (5  $C\ t\ uv\ ts$ )
  thus  $?case$ 
  by ( $simp, meson\ assms(2)\ type\_update.simps$ )
next
  case (6  $C\ t\ uw\ ts$ )
  thus  $?case$ 
  by ( $simp, meson\ assms(2)\ type\_update.simps$ )
next
  case (7  $C\ t\ ux\ ts$ )
  thus  $?case$ 
  by ( $simp, meson\ assms(2)\ type\_update.simps$ )
next
  case (8  $C\ t\ uy\ ts$ )
  thus  $?case$ 
  by ( $simp, meson\ assms(2)\ type\_update.simps$ )
next
  case (9  $C\ t\ uz\ ts$ )
  thus  $?case$ 
  by ( $simp, meson\ assms(2)\ type\_update.simps$ )
next
  case (10  $C\ t\ va\ ts$ )
  thus  $?case$ 

```

```

    by (simp, meson assms(2) type-update.simps)
next
  case (11 C t1 t2 sx ts)
  thus ?case
    by (simp del: convert-cond.simps, meson assms(2) type-update.simps)
next
  case (12 C t1 t2 sx ts)
  thus ?case
    by (simp, meson assms(2) type-update.simps)
next
  case (13 C ts)
  thus ?case
    by fastforce
next
  case (14 C ts)
  thus ?case
    by fastforce
next
  case (15 C ts)
  thus ?case
    by fastforce
next
  case (16 C ts)
  thus ?case
    by fastforce
next
  case (17 C tn tm es ts)
  thus ?case
    by (simp, meson assms(2) type-update.simps)
next
  case (18 C tn tm es ts)
  thus ?case
    by (simp, meson assms(2) type-update.simps)
next
  case (19 C tn tm es1 es2 ts)
  thus ?case
    by (simp, meson assms(2) type-update.simps)
next
  case (20 C i ts)
  thus ?case
    by (simp, meson assms(2) type-update.simps)
next
  case (21 C i ts)
  thus ?case
    by (simp, meson assms(2) type-update.simps)
next
  case (22 C is ts)
  thus ?case
    by (simp, metis assms(2) option.case-eq-if type-update.simps)

```

```

next
  case (23 C ts)
  thus ?case
    by (simp, metis assms(2) option.case-eq-if type-update.simps)
next
  case (24 C i ts)
  then show ?case
    by (simp, metis (no-types, lifting) assms(2) tf.case tf.exhaust type-update.simps)
next
  case (25 C i ts)
  thus ?case
    by (simp, metis (no-types, lifting) assms(2) tf.case tf.exhaust type-update.simps)
next
  case (26 C i ts)
  thus ?case
    by (simp, meson assms(2) type-update.simps)
next
  case (27 C i ts)
  thus ?case
    by (simp, meson assms(2) type-update.simps)
next
  case (28 C i ts)
  thus ?case
    by (simp, meson assms(2) type-update.simps)
next
  case (29 C i ts)
  thus ?case
    by (simp, meson assms(2) type-update.simps)
next
  case (30 C i ts)
  thus ?case
    by (simp, meson assms(2) type-update.simps)
next
  case (31 C t tp-sx a off ts)
  thus ?case
    by (simp, meson assms(2) type-update.simps)
next
  case (32 C t tp a off ts)
  thus ?case
    by (simp, meson assms(2) type-update.simps)
next
  case (33 C ts)
  thus ?case
    by (simp, meson assms(2) type-update.simps)
next
  case (34 C ts)
  thus ?case
    by (simp, meson assms(2) type-update.simps)
qed

```

```

thus ?thesis
  using assms
  by simp
qed

```

lemma *check-equiv-fold*:

```

check  $\mathcal{C}$  es ts = foldl ( $\lambda$  ts e. (case ts of Bot  $\Rightarrow$  Bot | -  $\Rightarrow$  check-single  $\mathcal{C}$  e ts))
ts es

```

proof (*induction es arbitrary: ts*)

```

case Nil
thus ?case
  by simp

```

next

```

case (Cons e es)
obtain ts' where ts'-def:check  $\mathcal{C}$  (e # es) ts = ts'
  by blast

```

show ?*case*

proof (*cases ts = Bot*)

```

case True
thus ?thesis
  using ts'-def
  by (induction es, simp-all)

```

next

```

case False
thus ?thesis
  using ts'-def Cons
  by (cases ts, simp-all)

```

qed

qed

lemma *check-neq-bot-snoc*:

assumes *check* \mathcal{C} (*es*@[*e*]) *ts* \neq *Bot*

shows *check* \mathcal{C} *es* *ts* \neq *Bot*

using *assms*

proof (*induction es arbitrary: ts*)

```

case Nil
thus ?case
  by (cases ts, simp-all)

```

next

```

case (Cons a es)
thus ?case
  by (cases ts, simp-all)

```

qed

lemma *check-unfold-snoc*:

assumes *check* \mathcal{C} *es* *ts* \neq *Bot*

shows *check* \mathcal{C} (*es*@[*e*]) *ts* = *check-single* \mathcal{C} *e* (*check* \mathcal{C} *es* *ts*)

proof –

obtain *f* **where** *f-def*:*f* = (λ *e* *ts*. (*case* *ts* of *Bot* \Rightarrow *Bot* | - \Rightarrow *check-single* \mathcal{C} *e*

```

ts))
  by blast
  have f-simp:  $\wedge ts. ts \neq \text{Bot} \implies (f\ e\ ts = \text{check-single } \mathcal{C}\ e\ ts)$ 
  proof -
    fix ts
    show  $ts \neq \text{Bot} \implies (f\ e\ ts = \text{check-single } \mathcal{C}\ e\ ts)$ 
      using f-def
      by (cases ts, simp-all)
    qed
  have check  $\mathcal{C}\ (es@[e])\ ts = \text{foldl } (\lambda\ ts\ e. (\text{case } ts\ \text{of } \text{Bot} \Rightarrow \text{Bot} \mid - \Rightarrow \text{check-single } \mathcal{C}\ e\ ts))\ ts\ (es@[e])$ 
    using check-equiv-fold
    by simp
  also
  have ... =  $\text{foldr } (\lambda\ e\ ts. (\text{case } ts\ \text{of } \text{Bot} \Rightarrow \text{Bot} \mid - \Rightarrow \text{check-single } \mathcal{C}\ e\ ts))\ (\text{rev } (es@[e]))\ ts$ 
    using foldl-conv-foldr
    by fastforce
  also
  have ... =  $f\ e\ (\text{foldr } (\lambda\ e\ ts. (\text{case } ts\ \text{of } \text{Bot} \Rightarrow \text{Bot} \mid - \Rightarrow \text{check-single } \mathcal{C}\ e\ ts))\ (\text{rev } es)\ ts)$ 
    using f-def
    by simp
  also
  have ... =  $f\ e\ (\text{check } \mathcal{C}\ es\ ts)$ 
    using foldr-conv-foldl[of - (rev es) ts] rev-rev-ident[of es] check-equiv-fold
    by simp
  also
  have ... =  $\text{check-single } \mathcal{C}\ e\ (\text{check } \mathcal{C}\ es\ ts)$ 
    using assms f-simp
    by simp
  finally
  show ?thesis .
  qed

```

lemma *check-single-imp-weakening*:

```

assumes check-single  $\mathcal{C}\ e\ (\text{Type } t1s) = \text{ctm}$ 
         $\text{ctm} \neq \text{Bot}$ 
        c-types-agree  $\text{ctn } t1s$ 
        c-types-agree  $\text{ctm } t2s$ 
shows  $\exists \text{ctm}'. \text{check-single } \mathcal{C}\ e\ \text{ctn} = \text{ctm}' \wedge \text{c-types-agree } \text{ctm}'\ t2s$ 
proof -
  consider (1)  $\text{check-single } \mathcal{C}\ e = \text{id}$ 
    | (2)  $\text{check-single } \mathcal{C}\ e = (\lambda \text{ctn}. \text{type-update-select } \text{ctn})$ 
    | (3)  $(\exists \text{cons } \text{prods}. (\text{check-single } \mathcal{C}\ e = (\lambda \text{ctn}. \text{type-update } \text{ctn } \text{cons } \text{prods})))$ 
  using check-single-imp assms
  by blast
  thus ?thesis
  proof (cases)

```

```

case 1
thus ?thesis
  using assms(1,3,4)
  by fastforce
next

case 2
note outer-2 = 2
hence t1s-cond:(length t1s ≥ 3 ∧ (t1s!(length t1s-2)) = (t1s!(length t1s-3)))
  using assms(1,2)
  by (simp, meson)
hence ctm-def:ctm = consume (Type t1s) [TAny, TSome T-i32]
  using assms(1,2) 2
  by simp
then obtain c-t where c-t-def:ctm = Type c-t
  using assms(2)
  by (meson consume.simps(1))
hence t2s-eq:t2s = c-t
  using assms(4)
  by simp
hence t2s-len:length t2s > 0
  using t1s-cond ctm-def c-t-def assms(2)
  by (metis Suc-leI Suc-n-not-le-n checker-type.inject(2) consume.simps(1)
    diff-is-0-eq dual-order.trans length-0-conv length-Cons length-greater-0-conv
    nat.simps(3) numeral-3-eq-3 take-eq-Nil)
have t1s-suffix-full:ct-suffix [TAny, TSome T-i32] (to-ct-list t1s)
  using assms(2) ctm-def ct-suffix-less
  by (metis consume.simps(1))
hence t1s-suffix:ct-suffix [TSome T-i32] (to-ct-list t1s)
  using assms(2) ctm-def ct-suffix-less
  by (metis append-butlast-last-id last.simps list.distinct(1))
obtain t t1s' where t1s-suffix2:t1s = t1s'@[t,t,T-i32]
  using type-update-select-type-length3 assms(1) c-t-def outer-2
  by fastforce
hence t2s-def:t2s = t1s'@[t]
  using ctm-def c-t-def t2s-eq t1s-suffix assms(2) t1s-suffix-full
  by simp
show ?thesis
  using assms(1,3,4)
proof (cases ctn)
  case (TopType x1)
  consider
    (1) length x1 = 0
    | (2) length x1 = 1
    | (3) length x1 = 2
    | (4) length x1 ≥ 3
    by linarith
  thus ?thesis
proof (cases)

```

```

case 1
hence check-single C e ctn = TopType [TAny]
  using 2 TopType
  by simp
thus ?thesis
  using ct-suffix-singleton to-ct-list-def t2s-len
  by auto
next
case 2
hence ct-suffix [TSome T-i32] x1
  using assms(3) TopType ct-suffix-imp-ct-list-eq ct-suffix-shared t1s-suffix
  by (metis One-nat-def append-Nil c-types-agree.simps(2) ct-list-eq-commute
ct-suffix-def
      diff-self-eq-0 drop-0 length-Cons list.size(3))
hence check-single C e ctn = TopType [TAny]
  using outer-2 TopType 2
  by simp
thus ?thesis
  using t2s-len ct-suffix-singleton
  by (simp add: to-ct-list-def)
next
case 3
have ct-list-eq (to-ct-list t1s) (to-ct-list (t1s' @ [t, t, T-i32]))
  using t1s-suffix2
  by (simp add: ct-list-eq-ts-conv-eq)
hence temp1:to-ct-list t1s = (to-ct-list (t1s' @ [t])) @ (to-ct-list [t, T-i32])
  using t1s-suffix2 to-ct-list-def
  by simp
hence ct-suffix (to-ct-list [t, T-i32]) (to-ct-list t1s)
  using ct-suffix-def[of (to-ct-list [t, T-i32]) (to-ct-list t1s)]
  by (simp add: ct-suffix-cons-it)
hence ct-suffix (to-ct-list [t, T-i32]) x1
  using assms(3) TopType 3
  by (simp, metis temp1 append-Nil ct-suffix-cons2 ct-suffix-def length-Cons
length-map
      list.size(3) numeral-2-eq-2 to-ct-list-def)
hence temp3:ct-list-eq (to-ct-list [t, T-i32]) x1
  using 3 ct-suffix-imp-ct-list-eq
  unfolding to-ct-list-def
  by (metis Suc-leI ct-list-eq-commute diff-is-0-eq drop-0 length-Cons
length-map lessI
      list.size(3) numeral-2-eq-2)
hence temp4:ct-suffix [TSome T-i32] x1
  using ct-suffix-less[of [TSome t] [TSome T-i32] x1]
  ct-suffix-extend-ct-list-eq[of [] []] ct-suffix-nil
  unfolding to-ct-list-def
  by fastforce
hence ct-suffix (take 1 x1) (to-ct-list [t])
  using temp3 ct-suffix-nil ct-list-eq-commute ct-suffix-extend-ct-list-eq[of []

```

```

[] (take 1 x1) (to-ct-list [t])
  unfolding to-ct-list-def
  by (simp, metis butlast.simps(2) butlast-conv-take ct-list-eq-take diff-Suc-1
length-Cons
      list.distinct(1) list.size(3))
thus ?thesis
  using TopType 2 3 ct-suffix-nil temp3 temp4 t2s-def to-ct-list-def
  apply (simp, safe)
  apply (metis append.assoc ct-suffix-def)
  done
next
case 4
then obtain nat where nat-def:length x1 = Suc (Suc (Suc nat))
  by (metis add-eq-if diff-Suc-1 le-Suc-ex numeral-3-eq-3 nat.distinct(2))
obtain x1' x x' x'' where x1-split:x1 = x1'@[x,x',x'']
proof -
  assume local-assms:( $\bigwedge x1' x x' x''. x1 = x1' @ [x, x', x''] \implies thesis$ )
  obtain x1' x1'' where tn-split:x1 = x1'@x1''
    length x1'' = 3
    using 4
    by (metis append-take-drop-id diff-diff-cancel length-drop)
  then obtain x x1''2 where x1'' = x#x1''2 length x1''2 = Suc (Suc 0)
    by (metis length-Suc-conv numeral-3-eq-3)
  then obtain x' x'' where tn''-def:x1'' = [x,x',x'']
    using List.length-Suc-conv[of x1''2 Suc 0]
    by (metis length-0-conv length-Suc-conv)
  thus ?thesis
    using tn-split local-assms
    by simp
qed
hence a:ct-suffix (x1'@[x,x',x'']) (to-ct-list (t1s' @ [t, t, T-i32]))
  using t1s-suffix2 assms(3) TopType
  by simp
hence b:ct-suffix (x1'@[x,x']) (to-ct-list (t1s' @ [t, t]))  $\wedge$  (ct-eq x'' (TSome
T-i32))
  using to-ct-list-def ct-suffix-unfold-one[of (x1'@[x,x']) x'' to-ct-list (t1s' @
[t, t])]
  by fastforce
hence c:ct-suffix (x1'@[x]) (to-ct-list (t1s' @ [t]))  $\wedge$  (ct-eq x' (TSome t))
  using to-ct-list-def ct-suffix-unfold-one[of (x1'@[x]) x' to-ct-list (t1s' @
[t])]
  by fastforce
hence d:ct-suffix x1' (to-ct-list t1s')  $\wedge$  (ct-eq x (TSome t))
  using to-ct-list-def ct-suffix-unfold-one[of (x1') x to-ct-list (t1s')]
  by fastforce
have (take (length x1 - 3) x1) = x1'
  using x1-split
  by simp
have x'-ind:(x1!(length x1-2)) = x'

```

```

    using x1-split List.nth-append-length[of x1' @ [x]]
  by simp
have x-ind:(x1!(length x1-3)) = x
  using x1-split
  by simp
have ct-suffix [TSome T-i32] x1
  using b x1-split ct-suffix-def ct-list-eq-def ct-suffixI[of x1 x1' @ [x, x']]
  by simp
hence check-single C e (TopType x1) = (select-return-top x1 (x1!(length
x1-2)) (x1!(length x1-3)))
  using type-update-select-conv-select-return-top[OF - 4]
  unfolding 2
  by blast
moreover
have ... = (TopType (x1'@[x])) ∨ ... = (TopType (x1'@[x']))
  apply (cases x; cases x')
  using x1-split 4 nat-def 2 x-ind x'-ind c d
  by simp-all
moreover
have ct-suffix (x1'@[x]) (to-ct-list t2s)
  by (simp add: c t2s-def)
moreover
have ct-suffix (x1'@[x']) (to-ct-list t2s)
  using ct-suffix-unfold-one[symmetric, of x' (TSome t) x1' (to-ct-list t1s')]
c d
    t2s-def
  unfolding to-ct-list-def
  by fastforce
ultimately
show ?thesis
  using TopType
  by auto
qed
qed simp-all
next
case 3
then obtain cons prods where c-s-def:check-single C e = (λctn. type-update
ctn cons prods)
  by blast
hence ctm-def:ctm = type-update (Type t1s) cons prods
  using assms(1)
  by fastforce
hence cons-suffix:ct-suffix cons (to-ct-list t1s)
  using assms
  by (simp, metis (full-types) produce.simps(6))
hence t-int-def:consume (Type t1s) cons = (Type (take (length t1s - length
cons) t1s))
  using ctm-def
  by simp

```

```

    hence ctm-def2:ctm = produce (Type (take (length t1s - length cons) t1s))
prods
  using ctm-def
  by simp
show ?thesis
proof (cases ctn)
  case (TopType x1)
  hence ct-suffix x1 (to-ct-list t1s)
    using assms(3)
    by simp
  thus ?thesis
    using assms(2) ctm-def2
  proof (cases prods)
    case (TopType x1)
    thus ?thesis
      using consume-c-types-agree[OF t-int-def assms(3)] ctm-def2 assms(4)
c-s-def
    by (metis c-types-agree.elims(2) produce.simps(3,4) type-update.simps)
  next
  case (Type x2)
  hence ctm-def3:ctm = Type ((take (length t1s - length cons) t1s)@ x2)
    using ctm-def2
    by simp
  have ct-suffix x1 cons ∨ ct-suffix cons x1
    using ct-suffix-shared assms(3) TopType cons-suffix
    by auto
  thus ?thesis
  proof (rule disjE)
    assume ct-suffix x1 cons
    hence consume (TopType x1) cons = TopType []
      by (simp add: ct-suffix-length)
    hence check-single C e ctn = TopType (to-ct-list x2)
      using c-s-def TopType Type
      by simp
    thus ?thesis
      using TopType ctm-def3 assms(4) c-types-agree-top2 ct-list-eq-refl
      by auto
  next
  assume ct-suffix cons x1
  hence 4:consume (TopType x1) cons = TopType (take (length x1 - length
cons) x1)
    by (simp add: ct-suffix-length)
  hence 3:check-single C e ctn = TopType ((take (length x1 - length cons
) x1) @ to-ct-list x2)
    using c-s-def TopType Type
    by simp
  have ((take (length t1s - length cons) t1s) @ x2) = t2s
    using assms(4) ctm-def3
    by simp

```

```

      have c-types-agree (TopType (take (length x1 - length cons ) x1)) (take
(length t1s - length cons) t1s)
      using consume-c-types-agree[OF t-int-def assms(3)] 4 TopType
      by simp
      hence c-types-agree (TopType (take (length x1 - length cons ) x1 @
to-ct-list x2)) (take (length t1s - length cons) t1s @ x2)
      unfolding c-types-agree.simps to-ct-list-def
      by (simp add: ct-suffix-cons2 ct-suffix-cons-it ct-suffix-extend-ct-list-eq)
      thus ?thesis
      using ctm-def3 assms 3
      by simp
    qed
  qed simp
next
case (Type x2)
thus ?thesis
  using assms
  by simp
next
case Bot
thus ?thesis
  using assms
  by simp
qed
qed
qed

```

lemma *b-e-type-checker-compose*:

```

  assumes b-e-type-checker C es (t1s -> t2s)
         b-e-type-checker C [e] (t2s -> t3s)
  shows b-e-type-checker C (es @ [e]) (t1s -> t3s)
proof -
  have c-types-agree (check-single C e (Type t2s)) t3s
    using assms(2)
    by simp
  then obtain ctm where ctm-def:check-single C e (Type t2s) = ctm
                        c-types-agree ctm t3s
                        ctm ≠ Bot
    by fastforce
  have c-types-agree (check C es (Type t1s)) t2s
    using assms(1)
    by simp
  then obtain ctn where ctn-def:check C es (Type t1s) = ctn
                        c-types-agree ctn t2s
                        ctn ≠ Bot
    by fastforce
  thus ?thesis
    using check-single-imp-weakening[OF ctm-def(1,3) ctn-def(2) ctm-def(2)]
      check-unfold-snoc[of C es (Type t1s) e]

```

by *simp*
qed

lemma *b-e-check-single-type-type*:

assumes *check-single* \mathcal{C} e $xs = (\text{Type } tm)$

shows $\exists tn. xs = (\text{Type } tn)$

proof –

consider (1) *check-single* \mathcal{C} $e = id$

| (2) *check-single* \mathcal{C} $e = (\lambda ct n. \text{type-update-select } ct n)$

| (3) $(\exists \text{cons prods. } (\text{check-single } \mathcal{C} \ e = (\lambda ct n. \text{type-update } ct n \ \text{cons prods})))$

using *check-single-imp assms*

by *blast*

thus *?thesis*

proof (*cases*)

case 1

thus *?thesis*

using *assms*

by *simp*

next

case 2

note *outer-2 = 2*

thus *?thesis*

using *assms*

proof (*cases xs*)

case (*TopType x1*)

consider

(1) *length x1 = 0*

| (2) *length x1 = Suc 0*

| (3) *length x1 = Suc (Suc 0)*

| (4) *length x1 \geq 3*

by *linarith*

thus *?thesis*

proof *cases*

case 1

thus *?thesis*

using *assms 2 TopType*

by *simp*

next

case 2

thus *?thesis*

using *assms outer-2 TopType produce-type-type*

by *fastforce*

next

case 3

thus *?thesis*

using *assms 2 TopType*

by (*simp, metis checker-type.distinct(1) checker-type.distinct(5)*)

next

case 4

```

then obtain nat where nat-def:length x1 = Suc (Suc (Suc nat))
  by (metis add-eq-if diff-Suc-1 le-Suc-ex numeral-3-eq-3 nat.distinct(2))
thus ?thesis
  using assms 2 TopType
proof –
  {
    assume a1: produce (if ct-suffix [TAny, TAny, TSome T-i32] x1 then
      TopType (take (length x1 – length [TAny, TAny, TSome T-i32]) x1) else if ct-suffix
      x1 [TAny, TAny, TSome T-i32] then TopType [] else Bot) (select-return-top x1 (x1
      ! Suc nat) (x1 ! nat)) = Type tm
    obtain tts :: checker-type ⇒ t list where
      f2: ∀ c. (∀ ca ts. produce c ca ≠ Type ts) ∨ c = Type (tts c)
    using produce-type-type by moura
    then have f3: ∧ts. ¬ ct-suffix [TAny, TAny, TSome T-i32] x1 ∨ Type
    tm ≠ Type ts
    using a1 by fastforce
    then have ∧ts. ¬ ct-suffix x1 [TAny, TAny, TSome T-i32] ∨ Type tm
    ≠ Type ts
    using f2 a1 by fastforce
    then have False
    using f3 a1 by fastforce
  }
thus ?thesis
  using assms 2 TopType nat-def
  by simp
qed
qed
qed simp-all
next
case 3
then obtain cons prods where check-def:check-single C e = (λctn. type-update
ctn cons prods)
  by blast
hence produce (consume xs cons) prods = (Type tm)
  using assms(1)
  by simp
thus ?thesis
  using assms check-def consume-type-type produce-type-type
  by blast
qed
qed

lemma b-e-check-single-weaken-type:
  assumes check-single C e (Type tn) = (Type tm)
  shows check-single C e (Type (ts@tn)) = Type (ts@tm)
proof –
  consider (1) check-single C e = id
    | (2) check-single C e = (λctn. type-update-select ctn)
    | (3) (∃ cons prods. (check-single C e = (λctn. type-update ctn cons prods)))

```

```

    using check-single-imp assms
  by blast
thus ?thesis
proof (cases)
  case 1
  thus ?thesis
    using assms(1)
    by simp
next
  case 2
  hence cond:(length tn ≥ 3 ∧ (tn!(length tn-2)) = (tn!(length tn-3)))
    using assms
    by (simp, metis checker-type.distinct(5))
  hence consume (Type tn) [TAny, TSome T-i32] = (Type tm)
    using assms 2
    by simp
  hence consume (Type (ts@tn)) [TAny, TSome T-i32] = (Type (ts@tm))
    using consume-weaken-type
    by blast
  moreover
  have (length (ts@tn) ≥ 3 ∧ ((ts@tn)!(length (ts@tn)-2)) = ((ts@tn)!(length
(ts@tn)-3)))
    using cond
    by (simp, metis add.commute add-leE nth-append-length-plus numeral-Bit1
numeral-One
one-add-one ordered-cancel-comm-monoid-diff-class.diff-add-assoc2)
  ultimately
  show ?thesis
    using 2
    by simp
next
  case 3
  then obtain cons prods where check-def:check-single C e = (λctn. type-update
ctn cons prods)
    by blast
  hence produce (consume (Type tn) cons) prods = (Type tm)
    using assms(1)
    by simp
  then obtain t-int where t-int-def:consume (Type tn) cons = (Type t-int)
    by (metis consume.simps(1) produce.simps(6))
  thus ?thesis
    using assms(1) check-def
      consume-weaken-type[OF t-int-def, of ts]
      produce-weaken-type[of t-int prods tm ts]
    by simp
qed
qed

```

lemma b-e-check-single-weaken-top:

```

assumes check-single  $\mathcal{C}$   $e$  (Type  $tn$ ) = TopType  $tm$ 
shows check-single  $\mathcal{C}$   $e$  (Type ( $ts@tn$ )) = TopType  $tm$ 
proof –
  consider (1) check-single  $\mathcal{C}$   $e$  = id
    | (2) check-single  $\mathcal{C}$   $e$  = ( $\lambda ctn.$  type-update-select  $ctn$ )
    | (3) ( $\exists$  cons prods. (check-single  $\mathcal{C}$   $e$  = ( $\lambda ctn.$  type-update  $ctn$  cons prods)))
  using check-single-imp assms
  by blast
thus ?thesis
proof (cases)
  case 1
    thus ?thesis
    using assms
    by simp
  next
  case 2
    thus ?thesis
    using assms
    by (simp, metis checker-type.distinct(1) checker-type.distinct(3) consume.simps(1))
  next
  case 3
    then obtain cons prods where check-def:check-single  $\mathcal{C}$   $e$  = ( $\lambda ctn.$  type-update
ctn cons prods)
    by blast
    hence produce (consume (Type  $tn$ ) cons) prods = (TopType  $tm$ )
    using assms(1)
    by simp
    moreover
    then obtain  $t\text{-int}$  where t-int-def:consume (Type  $tn$ ) cons = (Type  $t\text{-int}$ )
    by (metis checker-type.distinct(3) consume.simps(1) produce.simps(6))
    ultimately
    show ?thesis
    using check-def consume-weaken-type
    by (cases prods, auto)
  qed
qed

```

```

lemma b-e-check-weaken-type:
  assumes check  $\mathcal{C}$   $es$  (Type  $tn$ ) = (Type  $tm$ )
  shows check  $\mathcal{C}$   $es$  (Type ( $ts@tn$ )) = (Type ( $ts@tm$ ))
  using assms
proof (induction  $es$  arbitrary: tn tm rule: List.rev-induct)
  case Nil
    thus ?case
    by simp
  next
  case (snoc  $e$   $es$ )
    hence check-single  $\mathcal{C}$   $e$  (check  $\mathcal{C}$   $es$  (Type  $tn$ )) = Type  $tm$ 
    using check-unfold-snoc[OF check-neq-bot-snoc]

```

```

    by (metis checker-type.distinct(5))
  thus ?case
    using b-e-check-single-weaken-type b-e-check-single-type-type snoc
    by (metis check-unfold-snoc checker-type.distinct(5))
qed

```

```

lemma check-bot: check C es Bot = Bot
  by (simp add: list.case-eq-if)

```

```

lemma b-e-check-weaken-top:
  assumes check C es (Type tn) = (TopType tm)
  shows check C es (Type (ts@tn)) = (TopType tm)
  using assms
proof (induction es arbitrary: tn tm)
  case Nil
  thus ?case
    by simp
next
  case (Cons e es)
  show ?case
  proof (cases (check-single C e (Type tn)))
    case (TopType x1)
    hence check-single C e (Type (ts@tn)) = TopType x1
      using b-e-check-single-weaken-top
      by blast
    thus ?thesis
      using TopType Cons
      by simp
  next
    case (Type x2)
    hence check-single C e (Type (ts@tn)) = Type (ts@x2)
      using b-e-check-single-weaken-type
      by blast
    thus ?thesis
      using Cons Type
      by fastforce
  next
    case Bot
    thus ?thesis
      using check-bot Cons
      by simp
  qed
qed

```

qed

```

lemma b-e-type-checker-weaken:
  assumes b-e-type-checker C es (t1s -> t2s)
  shows b-e-type-checker C es (ts@t1s -> ts@t2s)
proof -

```

```

have c-types-agree (check C es (Type t1s)) t2s
  using assms(1)
  by simp
then obtain ctn where ctn-def:check C es (Type t1s) = ctn
  c-types-agree ctn t2s
  ctn ≠ Bot

  by fastforce
show ?thesis
proof (cases ctn)
  case (TopType x1)
  thus ?thesis
    using ctn-def(1,2) b-e-check-weaken-top[of C es t1s x1 ts]
    by (metis append-assoc b-e-type-checker.simps c-types-agree-imp-ct-list-eq
c-types-agree-top2)
  next
  case (Type x2)
  thus ?thesis
    using ctn-def(1,2) b-e-check-weaken-type[of C es t1s x2 ts]
    by simp
  next
  case Bot
  thus ?thesis
    using ctn-def(3)
    by simp
  qed
qed

lemma b-e-type-checker-complete:
  assumes C ⊢ es : (tn -> tm)
  shows b-e-type-checker C es (tn -> tm)
  using assms
proof (induction es (tn -> tm) arbitrary: tn tm rule: b-e-typing.induct)
  case (select C t)
  have ct-list-eq [TAny, TSome T-i32] [TSome t, TSome T-i32]
  by (simp add: to-ct-list-def ct-list-eq-def)
  thus ?case
    using ct-suffix-extend-ct-list-eq[OF ct-suffix-nil[of [TSome t]] to-ct-list-def]
    by auto
  next
  case (br-table C ts is i t1s t2s)
  show ?case
    using list-all-conv-same-lab[OF br-table]
    by (auto simp add: to-ct-list-def ct-suffix-nil ct-suffix-cons-it)
  next
  case (set-global i C t)
  thus ?case
    using to-ct-list-def ct-suffix-refl is-mut-def tq-t-def
    by auto
  next

```

```

case (composition  $C$  es  $t1s$   $t2s$   $e$   $t3s$ )
thus ?case
  using b-e-type-checker-compose
  by simp
next
case (weakening  $C$  es  $t1s$   $t2s$   $ts$ )
thus ?case
  using b-e-type-checker-weaken
  by simp
qed (auto simp add: to-ct-list-def ct-suffix-refl ct-suffix-nil ct-suffix-cons-it
      ct-suffix-singleton-any)

```

```

theorem b-e-typing-equiv-b-e-type-checker:
  shows ( $C \vdash es : (tn \rightarrow tm)$ ) = (b-e-type-checker  $C$  es ( $tn \rightarrow tm$ ))
  using b-e-type-checker-sound b-e-type-checker-complete
  by blast

```

end

11 WebAssembly Interpreter

theory *Wasm-Interpreter* **imports** *Wasm* **begin**

```

datatype res-crash =
  CError
| CExhaustion

```

```

datatype res =
  RCrash res-crash
| RTrap
| RValue v list

```

```

datatype res-step =
  RSCrash res-crash
| RSBreak nat v list
| RSReturn v list
| RSNormal e list

```

abbreviation crash-error **where** crash-error \equiv RSCrash CError

type-synonym depth = nat

type-synonym fuel = nat

type-synonym config-tuple = $s \times v \text{ list} \times e \text{ list}$

type-synonym config-one-tuple = $s \times v \text{ list} \times v \text{ list} \times e$

type-synonym res-tuple = $s \times v \text{ list} \times \text{res-step}$

```

fun split-vals :: b-e list ⇒ v list × b-e list where
  split-vals ((C v)#es) = (let (vs', es') = split-vals es in (v#vs', es'))
| split-vals es = ([], es)

fun split-vals-e :: e list ⇒ v list × e list where
  split-vals-e (($ C v)#es) = (let (vs', es') = split-vals-e es in (v#vs', es'))
| split-vals-e es = ([], es)

fun split-n :: v list ⇒ nat ⇒ v list × v list where
  split-n [] n = ([], [])
| split-n es 0 = ([], es)
| split-n (e#es) (Suc n) = (let (es', es'') = split-n es n in (e#es', es''))

lemma split-n-conv-take-drop: split-n es n = (take n es, drop n es)
  by (induction es n rule: split-n.induct, simp-all)

lemma split-n-length:
  assumes split-n es n = (es1, es2) length es ≥ n
  shows length es1 = n
  using assms
  unfolding split-n-conv-take-drop
  by fastforce

lemma split-n-conv-app:
  assumes split-n es n = (es1, es2)
  shows es = es1@es2
  using assms
  unfolding split-n-conv-take-drop
  by auto

lemma app-conv-split-n:
  assumes es = es1@es2
  shows split-n es (length es1) = (es1, es2)
  using assms
  unfolding split-n-conv-take-drop
  by auto

lemma split-vals-const-list: split-vals (map EConst vs) = (vs, [])
  by (induction vs, simp-all)

lemma split-vals-e-const-list: split-vals-e ($$* vs) = (vs, [])
  by (induction vs, simp-all)

lemma split-vals-e-conv-app:
  assumes split-vals-e xs = (as, bs)
  shows xs = ($$* as)@bs
  using assms
proof (induction xs arbitrary: as rule: split-vals-e.induct)
  case (1 v es)

```

```

obtain as' bs' where split-vals-e es = (as', bs')
  by (meson surj-pair)
thus ?case
  using 1
  by fastforce
qed simp-all

abbreviation expect :: 'a option ⇒ ('a ⇒ 'b) ⇒ 'b ⇒ 'b where
  expect a f b ≡ (case a of
    Some a' ⇒ f a'
    | None ⇒ b)

abbreviation vs-to-es :: v list ⇒ e list
where vs-to-es v ≡ $$$* (rev v)

definition e-is-trap :: e ⇒ bool where
  e-is-trap e = (case e of Trap ⇒ True | - ⇒ False)

definition es-is-trap :: e list ⇒ bool where
  es-is-trap es = (case es of [e] ⇒ e-is-trap e | - ⇒ False)

lemma[simp]: e-is-trap e = (e = Trap)
using e-is-trap-def
by (cases e) auto

lemma[simp]: es-is-trap es = (es = [Trap])
proof (cases es)
  case Nil
  thus ?thesis
    using es-is-trap-def
    by auto
next
  case outer-Cons:(Cons a list)
  thus ?thesis
  proof (cases list)
    case Nil
    thus ?thesis
      using outer-Cons es-is-trap-def
      by auto
    next
    case (Cons a' list')
    thus ?thesis
      using es-is-trap-def outer-Cons
      by auto
  qed
qed

axiomatization
  mem-grow-impl:: mem ⇒ nat ⇒ mem option where

```

mem-grow-impl-correct:(*mem-grow-impl* *m n* = *Some m'*) \implies (*mem-grow* *m n* = *m'*)

axiomatization

host-apply-impl:: *s* \implies *tf* \implies *host* \implies *v list* \implies (*s* \times *v list*) *option* **where**
host-apply-impl-correct:(*host-apply-impl* *s tf h vs* = *Some m'*) \implies (\exists *hs*. *host-apply* *s tf h vs hs* = *Some m'*)

function (*sequential*)

run-step :: *depth* \implies *nat* \implies *config-tuple* \implies *res-tuple*
and *run-one-step* :: *depth* \implies *nat* \implies *config-one-tuple* \implies *res-tuple* **where**
run-step *d i* (*s,vs,es*) = (let (*es'*, *es''*) = *split-vals-e es* in
 case *es'* of
 | [] \implies (*s,vs, crash-error*)
 | *e#es''* \implies
 if *e-is-trap e*
 then
 if (*es''* \neq [] \vee *ves* \neq [])
 then
 (*s, vs, RSNormal [Trap]*)
 else
 (*s, vs, crash-error*)
 else
 (let (*s',vs',r*) = *run-one-step d i* (*s,vs,(rev ves),e*) in
 case *r* of
 RSNormal res \implies (*s', vs', RSNormal (res@es')*)
 | - \implies (*s', vs', r*)))
| *run-one-step d i* (*s, vs, ves, e*) =
 (case *e* of
 — *B-E*
 — *UNOPS*
 \$(*Unop-i T-i32 iop*) \implies
 (case *ves* of
 (*ConstInt32 c*)#*ves'* \implies
 (*s, vs, RSNormal (vs-to-es ((ConstInt32 (app-unop-i iop c))#ves')*))
 | - \implies (*s, vs, crash-error*)
 | \$(*Unop-i T-i64 iop*) \implies
 (case *ves* of
 (*ConstInt64 c*)#*ves'* \implies
 (*s, vs, RSNormal (vs-to-es ((ConstInt64 (app-unop-i iop c))#ves')*))
 | - \implies (*s, vs, crash-error*)
 | \$(*Unop-i - iop*) \implies (*s, vs, crash-error*)
 | \$(*Unop-f T-f32 fop*) \implies
 (case *ves* of
 (*ConstFloat32 c*)#*ves'* \implies
 (*s, vs, RSNormal (vs-to-es ((ConstFloat32 (app-unop-f fop c))#ves')*))
 | - \implies (*s, vs, crash-error*)
 | \$(*Unop-f T-f64 fop*) \implies

```

      (case ves of
        (ConstFloat64 c)#ves' ⇒
          (s, vs, RSNormal (vs-to-es ((ConstFloat64 (app-unop-f fop c))#ves')))
        | - ⇒ (s, vs, crash-error))
    | $(Unop-f - fop) ⇒ (s, vs, crash-error)
  — BINOPS
    | $(Binop-i T-i32 iop) ⇒
      (case ves of
        (ConstInt32 c2)#(ConstInt32 c1)#ves' ⇒
          expect (app-binop-i iop c1 c2) (λc. (s, vs, RSNormal (vs-to-es
            ((ConstInt32 c)#ves')))) (s, vs, RSNormal ((vs-to-es ves')@[Trap]))
          | - ⇒ (s, vs, crash-error))
        | $(Binop-i T-i64 iop) ⇒
          (case ves of
            (ConstInt64 c2)#(ConstInt64 c1)#ves' ⇒
              expect (app-binop-i iop c1 c2) (λc. (s, vs, RSNormal (vs-to-es
                ((ConstInt64 c)#ves')))) (s, vs, RSNormal ((vs-to-es ves')@[Trap]))
              | - ⇒ (s, vs, crash-error))
            | $(Binop-i - iop) ⇒ (s, vs, crash-error)
            | $(Binop-f T-f32 fop) ⇒
              (case ves of
                (ConstFloat32 c2)#(ConstFloat32 c1)#ves' ⇒
                  expect (app-binop-f fop c1 c2) (λc. (s, vs, RSNormal (vs-to-es
                    ((ConstFloat32 c)#ves')))) (s, vs, RSNormal ((vs-to-es ves')@[Trap]))
                  | - ⇒ (s, vs, crash-error))
                | $(Binop-f T-f64 fop) ⇒
                  (case ves of
                    (ConstFloat64 c2)#(ConstFloat64 c1)#ves' ⇒
                      expect (app-binop-f fop c1 c2) (λc. (s, vs, RSNormal (vs-to-es
                        ((ConstFloat64 c)#ves')))) (s, vs, RSNormal ((vs-to-es ves')@[Trap]))
                      | - ⇒ (s, vs, crash-error))
                    | $(Binop-f - fop) ⇒ (s, vs, crash-error)
                  — TESTOPS
                    | $(Testop T-i32 testop) ⇒
                      (case ves of
                        (ConstInt32 c)#ves' ⇒
                          (s, vs, RSNormal (vs-to-es ((ConstInt32 (wasm-bool (app-testop-i testop
                            c))#ves'))))
                          | - ⇒ (s, vs, crash-error))
                        | $(Testop T-i64 testop) ⇒
                          (case ves of
                            (ConstInt64 c)#ves' ⇒
                              (s, vs, RSNormal (vs-to-es ((ConstInt32 (wasm-bool (app-testop-i testop
                                c))#ves'))))
                              | - ⇒ (s, vs, crash-error))
                            | $(Testop - testop) ⇒ (s, vs, crash-error)
                          — RELOPS
                            | $(Relop-i T-i32 iop) ⇒
                              (case ves of

```

```

      (ConstInt32 c2)#(ConstInt32 c1)#ves' ⇒
      (s, vs, RSNormal (vs-to-es ((ConstInt32 (wasm-bool (app-relop-i iop
c1 c2))))#ves'))
    | - ⇒ (s, vs, crash-error))
  | $(Relop-i T-i64 iop) ⇒
    (case ves of
      (ConstInt64 c2)#(ConstInt64 c1)#ves' ⇒
      (s, vs, RSNormal (vs-to-es ((ConstInt32 (wasm-bool (app-relop-i iop
c1 c2))))#ves'))
    | - ⇒ (s, vs, crash-error))
  | $(Relop-i - iop) ⇒ (s, vs, crash-error)
  | $(Relop-f T-f32 fop) ⇒
    (case ves of
      (ConstFloat32 c2)#(ConstFloat32 c1)#ves' ⇒
      (s, vs, RSNormal (vs-to-es ((ConstInt32 (wasm-bool (app-relop-f fop
c1 c2))))#ves'))
    | - ⇒ (s, vs, crash-error))
  | $(Relop-f T-f64 fop) ⇒
    (case ves of
      (ConstFloat64 c2)#(ConstFloat64 c1)#ves' ⇒
      (s, vs, RSNormal (vs-to-es ((ConstInt32 (wasm-bool (app-relop-f fop
c1 c2))))#ves'))
    | - ⇒ (s, vs, crash-error))
  | $(Relop-f - fop) ⇒ (s, vs, crash-error)
— CONVERT
  | $(Cvtop t2 Convert t1 sx) ⇒
    (case ves of
      v#ves' ⇒
      (if (types-agree t1 v)
        then
          expect (cut t2 sx v) (λv'. (s, vs, RSNormal (vs-to-es (v'#ves'))))
        (s, vs, RSNormal ((vs-to-es ves')@[Trap]))
      else
        (s, vs, crash-error))
    | - ⇒ (s, vs, crash-error))
  | $(Cvtop t2 Reinterpret t1 sx) ⇒
    (case ves of
      v#ves' ⇒
      (if (types-agree t1 v ∧ sx = None)
        then
          (s, vs, RSNormal (vs-to-es ((wasm-deserialise (bits v) t2)#ves'))
        else
          (s, vs, crash-error))
    | - ⇒ (s, vs, crash-error))
— UNREACHABLE
  | $Unreachable ⇒
    (s, vs, RSNormal ((vs-to-es ves)@[Trap]))
— NOP
  | $Nop ⇒

```

```

      (s, vs, RSNormal (vs-to-es ves))
— DROP
| $Drop ⇒
  (case ves of
    v#ves' ⇒
      (s, vs, RSNormal (vs-to-es ves'))
    | - ⇒ (s, vs, crash-error))
— SELECT
| $Select ⇒
  (case ves of
    (ConstInt32 c)#v2#v1#ves' ⇒
      (if int-eq c 0 then (s, vs, RSNormal (vs-to-es (v2#ves'))) else (s, vs,
RSNormal (vs-to-es (v1#ves'))))
    | - ⇒ (s, vs, crash-error))
— BLOCK
| $(Block (t1s -> t2s) es) ⇒
  (if length ves ≥ length t1s
    then
      let (ves', ves'') = split-n ves (length t1s) in
      (s, vs, RSNormal ((vs-to-es ves'') @ [Label (length t2s) [] ((vs-to-es
ves')@($* es))]))
    else
      (s, vs, crash-error))
— LOOP
| $(Loop (t1s -> t2s) es) ⇒
  (if length ves ≥ length t1s
    then
      let (ves', ves'') = split-n ves (length t1s) in
      (s, vs, RSNormal ((vs-to-es ves'') @ [Label (length t1s) $(Loop (t1s ->
t2s) es)] ((vs-to-es ves')@($* es)))))
    else
      (s, vs, crash-error))
— IF
| $(If tf es1 es2) ⇒
  (case ves of
    (ConstInt32 c)#ves' ⇒
      if int-eq c 0
      then
        (s, vs, RSNormal ((vs-to-es ves')@$(Block tf es2)))
      else
        (s, vs, RSNormal ((vs-to-es ves')@$(Block tf es1)))
    | - ⇒ (s, vs, crash-error))
— BR
| $Br j ⇒
  (s, vs, RSBreak j ves)
— BR-IF
| $Br-if j ⇒
  (case ves of
    (ConstInt32 c)#ves' ⇒

```

```

      if int-eq c 0
      then
        (s, vs, RSNormal (vs-to-es ves'))
      else
        (s, vs, RSNormal ((vs-to-es ves') @ [Br j]))
    | - => (s, vs, crash-error)
  — BR-TABLE
| $Br-table js j =>
  (case ves of
    (ConstInt32 c)#ves' =>
      let k = nat-of-int c in
      if k < length js
      then
        (s, vs, RSNormal ((vs-to-es ves') @ [Br (js!k)]))
      else
        (s, vs, RSNormal ((vs-to-es ves') @ [Br j]))
  | - => (s, vs, crash-error)
  — CALL
| $Call j =>
  (s, vs, RSNormal ((vs-to-es ves) @ [Callcl (sfunc s i j)]))
  — CALL-INDIRECT
| $Call-indirect j =>
  (case ves of
    (ConstInt32 c)#ves' =>
      (case (stab s i (nat-of-int c)) of
        Some cl =>
          if (stypes s i j = cl-type cl)
          then
            (s, vs, RSNormal ((vs-to-es ves') @ [Callcl cl]))
          else
            (s, vs, RSNormal ((vs-to-es ves')@[Trap]))
      | - => (s, vs, RSNormal ((vs-to-es ves')@[Trap])))
  | - => (s, vs, crash-error)
  — RETURN
| $Return =>
  (s, vs, RSReturn ves)
  — GET-LOCAL
| $Get-local j =>
  (if j < length vs
   then (s, vs, RSNormal (vs-to-es ((vs!j)#ves)))
   else (s, vs, crash-error))
  — SET-LOCAL
| $Set-local j =>
  (case ves of
    v#ves' =>
      if j < length vs
      then (s, vs[j := v], RSNormal (vs-to-es ves'))
      else (s, vs, crash-error)
  | - => (s, vs, crash-error))

```

```

— TEE-LOCAL
| $Tee-local j ⇒
  (case ves of
    v#ves' ⇒
      (s, vs, RSNormal ((vs-to-es (v#ves)) @ [$(Set-local j)]))
    | - ⇒ (s, vs, crash-error))
— GET-GLOBAL
| $Get-global j ⇒
  (s, vs, RSNormal (vs-to-es ((sglob-val s i j)#ves)))
— SET-GLOBAL
| $Set-global j ⇒
  (case ves of
    v#ves' ⇒ ((supdate-glob s i j v), vs, RSNormal (vs-to-es ves'))
    | - ⇒ (s, vs, crash-error))
— LOAD
| $(Load t None a off) ⇒
  (case ves of
    (ConstInt32 k)#ves' ⇒
      expect (smem-ind s i)
      (λj.
        expect (load ((mem s)!j) (nat-of-int k) off (t-length t))
        (λbs. (s, vs, RSNormal (vs-to-es ((wasm-deserialise bs t)#ves'))))
        (s, vs, RSNormal ((vs-to-es ves')@[Trap])))
      (s, vs, crash-error)
    | - ⇒ (s, vs, crash-error))
— LOAD PACKED
| $(Load t (Some (tp, sx)) a off) ⇒
  (case ves of
    (ConstInt32 k)#ves' ⇒
      expect (smem-ind s i)
      (λj.
        expect (load-packed sx ((mem s)!j) (nat-of-int k) off (tp-length tp)
          (t-length t))
        (λbs. (s, vs, RSNormal (vs-to-es ((wasm-deserialise bs t)#ves'))))
        (s, vs, RSNormal ((vs-to-es ves')@[Trap])))
      (s, vs, crash-error)
    | - ⇒ (s, vs, crash-error))
— STORE
| $(Store t None a off) ⇒
  (case ves of
    v#(ConstInt32 k)#ves' ⇒
      (if (types-agree t v)
        then
          expect (smem-ind s i)
          (λj.
            expect (store ((mem s)!j) (nat-of-int k) off (bits v) (t-length t))
            (λmem'. (s|mem:= ((mem s)[j := mem'])), vs, RSNormal
              (vs-to-es ves')))
          (s, vs, RSNormal ((vs-to-es ves')@[Trap])))
  )

```

```

      (s, vs, crash-error)
    else
      (s, vs, crash-error)
    | - ⇒ (s, vs, crash-error)
  — STORE-PACKED
  | $(Store t (Some tp) a off) ⇒
    (case ves of
      v#(ConstInt32 k)#ves' ⇒
        (if (types-agree t v)
          then
            expect (smem-ind s i)
              (λj.
                expect (store-packed ((mem s)!j) (nat-of-int k) off (bits v)
                  (tp-length tp))
                  (λmem'. (s{mem:= ((mem s)[j := mem']})), vs, RSNormal
                    (vs-to-es ves'))
                    (s, vs, RSNormal ((vs-to-es ves')@[Trap])))
                (s, vs, crash-error)
          else
            (s, vs, crash-error)
        | - ⇒ (s, vs, crash-error))
  — CURRENT-MEMORY
  | $Current-memory ⇒
    expect (smem-ind s i)
      (λj. (s, vs, RSNormal (vs-to-es ((ConstInt32 (int-of-nat (mem-size
        ((s.mem s)!j))))#ves))))
      (s, vs, crash-error)
  — GROW-MEMORY
  | $Grow-memory ⇒
    (case ves of
      (ConstInt32 c)#ves' ⇒
        expect (smem-ind s i)
          (λj.
            let l = (mem-size ((s.mem s)!j)) in
              (expect (mem-grow-impl ((mem s)!j) (nat-of-int c))
                (λmem'. (s{mem:= ((mem s)[j := mem']})), vs, RSNormal
                  (vs-to-es ((ConstInt32 (int-of-nat l))#ves'))
                  (s, vs, RSNormal (vs-to-es ((ConstInt32 int32-minus-one)#ves'))))
                (s, vs, crash-error)
            | - ⇒ (s, vs, crash-error))
  — VAL - should not be executed
  | $C v ⇒ (s, vs, crash-error)
  — E
  — CALLCL
  | Callcl cl ⇒
    (case cl of
      Func-native i' (t1s -> t2s) ts es ⇒
        let n = length t1s in
          let m = length t2s in

```

```

    if length ves ≥ n
    then
      let (ves', ves'') = split-n ves n in
      let zs = n-zeros ts in
      (s, vs, RSNormal ((vs-to-es ves'') @ ([Local m i' ((rev ves')@zs)
[$(Block ([ -> t2s) es])]))))
    else
      (s, vs, crash-error)
  | Func-host (t1s -> t2s) f ⇒
    let n = length t1s in
    let m = length t2s in
    if length ves ≥ n
    then
      let (ves', ves'') = split-n ves n in
      case host-apply-impl s (t1s -> t2s) f (rev ves') of
      Some (s', rves) ⇒
        if list-all2 types-agree t2s rves
        then
          (s', vs, RSNormal ((vs-to-es ves'') @ ($$* rves)))
        else
          (s', vs, crash-error)
      | None ⇒ (s, vs, RSNormal ((vs-to-es ves'')@[Trap]))
    else
      (s, vs, crash-error)
— LABEL
| Label ln les es ⇒
  if es-is-trap es
  then
    (s, vs, RSNormal ((vs-to-es ves)@[Trap]))
  else
    (if (const-list es)
    then
      (s, vs, RSNormal ((vs-to-es ves)@es))
    else
      let (s', vs', res) = run-step d i (s, vs, es) in
      (case res of
      RSBreak 0 bvs ⇒
        if (length bvs ≥ ln)
        then (s', vs', RSNormal ((vs-to-es ((take ln bvs)@ves))@les))
        else (s', vs', crash-error)
      | RSBreak (Suc n) bvs ⇒
        (s', vs', RSBreak n bvs)
      | RSReturn rvs ⇒
        (s', vs', RSReturn rvs)
      | RSNormal es' ⇒
        (s', vs', RSNormal ((vs-to-es ves)@[Label ln les es'])))
      | - ⇒ (s', vs', crash-error)))
— LOCAL
| Local ln j vls es ⇒

```

```

    if es-is-trap es
    then
      (s, vs, RSNormal ((vs-to-es ves)@[Trap]))
    else
      (if (const-list es)
       then
         if (length es = ln)
         then (s, vs, RSNormal ((vs-to-es ves)@es))
         else (s, vs, crash-error)
       else
         case d of
           0 ⇒ (s, vs, crash-error)
         | Suc d' ⇒
           let (s', vls', res) = run-step d' j (s, vls, es) in
           (case res of
            RSReturn rvs ⇒
              if (length rvs ≥ ln)
              then (s', vs, RSNormal (vs-to-es ((take ln rvs)@ves)))
              else (s', vs, crash-error)
            | RSNormal es' ⇒
              (s', vs, RSNormal ((vs-to-es ves)@[Local ln j vls' es']))
            | - ⇒ (s', vs, RSCrash CExhaustion)))
      — TRAP - should not be executed
      | Trap ⇒ (s, vs, crash-error))
  by pat-completeness auto
termination
proof -
{
  fix xs::e list and as b bs
  assume local-assms:(as, b#bs) = split-vals-e xs
  have 2*(size b) < 2*(size-list size xs) + 1
  using local-assms[symmetric] split-vals-e-conv-app
         size-list-estimation'[of b xs size b size]
  unfolding size-list-def
  by fastforce
}
thus ?thesis
  by (relation measure (case-sum
    (λp. 2 * (size-list size (snd (snd (snd (snd p)))))) + 1)
    (λp. 2 * size (snd (snd (snd (snd (snd p)))))))) auto
qed

fun run-v :: fuel ⇒ depth ⇒ nat ⇒ config-tuple ⇒ (s × res) where
  run-v (Suc n) d i (s,vs,es) = (if (es-is-trap es)
    then (s, RTrap)
    else if (const-list es)
      then (s, RValue (fst (split-vals-e es)))
      else (let (s',vs',res) = (run-step d i (s,vs,es)) in
        case res of

```

```

|  $run-v\ 0\ d\ i\ (s,vs,es) = (s, RCrash\ CExhaustion)$ 
|  $RSNormal\ es' \Rightarrow run-v\ n\ d\ i\ (s',vs',es')$ 
|  $RSCrash\ error \Rightarrow (s, RCrash\ error)$ 
|  $- \Rightarrow (s, RCrash\ CError))$ 

```

end

12 Soundness of Interpreter

theory *Wasm-Interpreter-Properties* **imports** *Wasm-Interpreter Wasm-Properties*
begin

lemma *is-const-list-vs-to-es-list: const-list* ($\$\$*$ *vs*)
using *is-const-list*
by *auto*

lemma *not-const-vs-to-es-list:*
assumes $\sim(is-const\ e)$
shows $vs1\ @\ [e]\ @\ vs2 \neq \$\$*\ vs$

proof –

```

fix vs
{
  assume  $vs1\ @\ [e]\ @\ vs2 = \$\$*\ vs$ 
  hence  $(\forall y \in set\ (vs1\ @\ [e]\ @\ vs2). \exists x. y = \$C\ x)$ 
  by simp
  hence False
  using assms
  unfolding is-const-def
  by fastforce
}
thus  $vs1\ @\ [e]\ @\ vs2 \neq \$\$*\ vs$ 
by fastforce

```

qed

lemma *neq-label-nested:[Label n les es] ≠ es*

proof –

```

have  $size-list\ size\ [Label\ n\ les\ es] > size-list\ size\ es$ 
by simp
thus ?thesis
by fastforce

```

qed

lemma *neq-local-nested:[Local n i lvs es] ≠ es*

proof –

```

have  $size-list\ size\ [Local\ n\ i\ lvs\ es] > size-list\ size\ es$ 
by simp
thus ?thesis
by fastforce

```

qed

```

lemma trap-not-value:[Trap] ≠ $$*es
  by fastforce

thm Lfilled.simps[of - - - [e], simplified]

lemma lfilled-single:
  assumes Lfilled k lholed es [e]
    ∧ a b c. e ≠ Label a b c
  shows (es = [e] ∧ lholed = LBase [] []) ∨ es = []
  using assms
proof (cases rule: Lfilled.cases)
  case (L0 vs es')
  thus ?thesis
    by (metis Nil-is-append-conv append-self-conv2 butlast-append butlast-snoc)
next
  case (LN vs n es' l es'' k lfilledk)
  assume (∧ a b c. e ≠ Label a b c)
  thus ?thesis
    using LN(2)
    unfolding Cons-eq-append-conv
    by fastforce
qed

lemma lfilled-eq:
  assumes Lfilled j lholed es LI
    Lfilled j lholed es' LI
  shows es = es'
  using assms
proof (induction arbitrary: es' rule: Lfilled.induct)
  case (L0 vs lholed es' es)
  thus ?case
    using Lfilled.simps[of 0, simplified]
    by auto
next
  case (LN vs lholed n les' l les'' k les lfilledk)
  thus ?case
    using Lfilled.simps[of (k+1) LRec vs n les' l les'' es' (vs @ [Label n les' lfilledk]
    @ les''), simplified]
    by auto
qed

lemma lfilled-size:
  assumes Lfilled j lholed es LI
  shows size-list size LI ≥ size-list size es
  using assms
  by (induction rule: Lfilled.induct) auto

thm Lfilled.simps[of - - es es, simplified]

```

```

lemma reduce-simple-not-eq:
  assumes  $(\text{es}) \rightsquigarrow (\text{es}')$ 
  shows  $\text{es} \neq \text{es}'$ 
  using assms
proof (induction es' rule: reduce-simple.induct)
  case (label-const vs n es)
  thus ?case
    using neq-label-nested
    by auto
next
  case (br vs n i lholed LI es)
  have  $\text{size-list size [Label } n \text{ es LI]} > \text{size-list size (vs @ es)}$ 
    using  $\text{lfilled-size[OF br(3)]}$ 
    by simp
  thus ?case
    by fastforce
next
  case (local-const es n i vs)
  thus ?case
    using neq-local-nested
    by auto
next
  case (return vs n j lholed es i vls)
  hence  $\text{size-list size [Local } n \text{ i vls es]} > \text{size-list size vs}$ 
    using  $\text{lfilled-size[OF return(3)]}$ 
    by simp
  thus ?case
    by auto
qed auto

lemma reduce-not-eq:
  assumes  $(\text{s}; \text{vs}; \text{es}) \rightsquigarrow\text{-i} (\text{s}'; \text{vs}'; \text{es}')$ 
  shows  $\text{es} \neq \text{es}'$ 
  using assms
proof (induction es' rule: reduce.induct)
  case (basic e e' s vs i)
  thus ?case
    using reduce-simple-not-eq
    by simp
next
  case (callcl-host-Some cl t1s t2s f ves vcs n m s hs s' vcs' vs i)
  thus ?case
    by (cases vcs' rule: rev-cases) auto
next
  case (label s vs es i s' vs' es' k lholed les les')
  thus ?case
    using lfilled-eq
    by fastforce

```

```

qed auto

lemma reduce-simple-not-value:
  assumes  $(\!|es|) \rightsquigarrow (\!|es'|)$ 
  shows  $es \neq \text{\$}\$* \text{ vs}$ 
  using assms
proof (induction rule: reduce-simple.induct)
  case (block vs n t1s t2s m es)
  have  $\neg(\text{is-const } (\text{\$Block } (t1s \rightarrow t2s) es))$ 
    unfolding is-const-def
    by simp
  thus ?case
    using not-const-vs-to-es-list
    by (metis append.right-neutral)
next
  case (loop vs n t1s t2s m es)
  have  $\neg(\text{is-const } (\text{\$Loop } (t1s \rightarrow t2s) es))$ 
    unfolding is-const-def
    by simp
  thus ?case
    using not-const-vs-to-es-list
    by (metis append.right-neutral)
next
  case (trap lholed es)
  show ?case
    using trap(2)
  proof (cases rule: Lfilled.cases)
    case L0
    have  $\neg(\text{is-const } \text{Trap})$ 
      unfolding is-const-def
      by simp
    thus ?thesis
      using L0 not-const-vs-to-es-list
      by fastforce
  qed auto
qed auto

lemma reduce-not-value:
  assumes  $(\!|s;vs;es|) \rightsquigarrow\text{-}i (\!|s';vs';es'|)$ 
  shows  $es \neq \text{\$}\$* \text{ ves}$ 
  using assms
proof (induction es' arbitrary: ves rule: reduce.induct)
  case (basic e e' s vs i)
  thus ?case
    using reduce-simple-not-value
    by fastforce
next
  case (callcl-native cl i' j ts es s t1s t2s ves vcs n k m zs vs i)
  have  $\neg(\text{is-const } (\text{Callcl } cl))$ 

```

```

    unfolding is-const-def
  by simp
thus ?case
  using not-const-vs-to-es-list
  by (metis append.right-neutral)
next
case (callcl-host-Some cl t1s t2s f ves vcs n m s i s' vcs' vs)
have ¬(is-const (Callcl cl))
  unfolding is-const-def
  by simp
thus ?case
  using not-const-vs-to-es-list
  by (metis append.right-neutral)
next
case (callcl-host-None cl t1s t2s f ves vcs n m s vs i)
have ¬(is-const (Callcl cl))
  unfolding is-const-def
  by simp
thus ?case
  using not-const-vs-to-es-list
  by (metis append.right-neutral)
next
case (label s vs es i s' vs' es' k lholed les les')
show ?case
  using label(2,4)
proof (induction rule: Lfilled.induct)
  case (L0 lvs lholed les' les)
  {
    assume lvs @ les @ les' = $$* ves
    hence (∀ y ∈ set (lvs @ les @ les')). ∃ x. y = $C x
      by simp
    hence (∀ y ∈ set les. ∃ x. y = $C x)
      by simp
    hence ∃ vs1. les = $$* vs1
      unfolding ex-map-conv.
  }
  thus ?case
    using L0(3)
    by fastforce
next
case (LN lvs lholed ln les' l les'' k les lfilledk)
have ¬(is-const (Label ln les' lfilledk))
  unfolding is-const-def
  by simp
thus ?case
  using not-const-vs-to-es-list
  by fastforce
qed
qed auto

```

```

lemma reduce-simple-not-nil:
  assumes  $(\text{es}) \rightsquigarrow (\text{es}')$ 
  shows  $\text{es} \neq []$ 
  using assms
proof (induction rule: reduce-simple.induct)
  case (trap es lholed)
  thus ?case
    using Lfilled.simps[of 0 lholed [Trap]]
    by auto
qed auto

lemma reduce-not-nil:
  assumes  $(\text{s}; \text{vs}; \text{es}) \rightsquigarrow\text{-i} (\text{s}'; \text{vs}'; \text{es}')$ 
  shows  $\text{es} \neq []$ 
  using assms
proof (induction rule: reduce.induct)
  case (basic e e' s vs i)
  thus ?case
    using reduce-simple-not-nil
    by simp
next
  case (label s vs es i s' vs' es' k lholed les les')
  show ?case
    using lfilled-size[OF label(2)] label(4)
    by (metis One-nat-def add-is-0 le-0-eq list.exhaust list.size(2) list.size-gen(1))
zero-neq-one
qed auto

lemma reduce-simple-not-trap:
  assumes  $(\text{es}) \rightsquigarrow (\text{es}')$ 
  shows  $\text{es} \neq [\text{Trap}]$ 
  using assms
  by (induction rule: reduce-simple.induct) auto

lemma reduce-not-trap:
  assumes  $(\text{s}; \text{vs}; \text{es}) \rightsquigarrow\text{-i} (\text{s}'; \text{vs}'; \text{es}')$ 
  shows  $\text{es} \neq [\text{Trap}]$ 
  using assms
proof (induction rule: reduce.induct)
  case (basic e e' s vs i)
  thus ?case
    using reduce-simple-not-trap
    by simp
next
  case (label s vs es i s' vs' es' k lholed les les')
  {
    assume  $\text{les} = [\text{Trap}]$ 
    hence Lfilled k lholed es [Trap]
  }

```

```

    using label(2)
    by simp
  hence False
    using lfilled-single reduce-not-nil[OF label(1)] label(4)
    by fastforce
}
thus ?case
  by auto
qed auto

```

```

lemma reduce-simple-call:  $\neg([\text{\$Call } j]) \rightsquigarrow (es')$ 
  using reduce-simple.simps[of [\text{\$Call } j], simplified] lfilled-single
  by fastforce

```

```

lemma reduce-call:
  assumes  $(s;vs;[\text{\$Call } j]) \rightsquigarrow i (s';vs';es')$ 
  shows  $s = s'$ 
     $vs = vs'$ 
     $es' = [\text{Callcl } (sfunc\ s\ i\ j)]$ 
  using assms
proof (induction [\text{\$Call } j]:: e list i s' vs' es' rule: reduce.induct)
  case (label s vs es i s' vs' es' k lholed les')
  have  $es = [\text{\$Call } j]$ 
     $lholed = LBase [] []$ 
    using reduce-not-nil[OF label(1)] lfilled-single[OF label(5)]
    by auto
  thus  $s = s'$ 
     $vs = vs'$ 
     $les' = [\text{Callcl } (sfunc\ s\ i\ j)]$ 
    using label(2,3,4,6) Lfilled.simps[of k LBase [] [] [\text{Callcl } (sfunc\ s\ i\ j)] les']
    by auto
qed (auto simp add: reduce-simple-call)

```

```

lemma run-one-step-basic-unreachable-result:
  assumes  $run-one-step\ d\ i\ (s,vs,ves,\$Unreachable) = (s', vs', res)$ 
  shows  $\exists r. res = RSNormal\ r$ 
  using assms
  by auto

```

```

lemma run-one-step-basic-nop-result:
  assumes  $run-one-step\ d\ i\ (s,vs,ves,\$Nop) = (s', vs', res)$ 
  shows  $\exists r. res = RSNormal\ r$ 
  using assms
  by auto

```

```

lemma run-one-step-basic-drop-result:
  assumes  $run-one-step\ d\ i\ (s,vs,ves,\$Drop) = (s', vs', res)$ 
  shows  $(\exists r. res = RSNormal\ r) \vee (\exists e. res = RSCrash\ e)$ 
  using assms

```

```

by (cases ves) auto

lemma run-one-step-basic-select-result:
  assumes run-one-step d i (s,vs,ves,$Select) = (s', vs', res)
  shows (∃ r. res = RSNormal r) ∨ (∃ e. res = RSCrash e)
  using assms
proof (cases ves)
  case (Cons a list)
  thus ?thesis
    using assms
  proof (cases a; cases list)
    fix x1a aa listaa
    assume a = ConstInt32 x1a and list = aa#listaa
    thus ?thesis
      using assms Cons
      by (cases listaa; cases int-eq x1a 0) auto
  qed auto
qed auto

lemma run-one-step-basic-block-result:
  assumes run-one-step d i (s,vs,ves,$(Block x51 x52)) = (s', vs', res)
  shows (∃ r. res = RSNormal r) ∨ (∃ e. res = RSCrash e)
  using assms
proof -
  obtain t1s t2s where x51 = (t1s -> t2s)
    using tf.exhaust
    by blast
  moreover obtain ves' ves'' where split-n ves (length t1s) = (ves', ves'')
    by (metis surj-pair)
  ultimately
  show ?thesis
    using assms
    by (cases length t1s ≤ length ves) auto
qed

lemma run-one-step-basic-loop-result:
  assumes run-one-step d i (s,vs,ves,$(Loop x61 x62)) = (s', vs', res)
  shows (∃ r. res = RSNormal r) ∨ (∃ e. res = RSCrash e)
  using assms
proof -
  obtain t1s t2s where x61 = (t1s -> t2s)
    using tf.exhaust
    by blast
  moreover obtain ves' ves'' where split-n ves (length t1s) = (ves', ves'')
    by (metis surj-pair)
  ultimately
  show ?thesis
    using assms
    by (cases length t1s ≤ length ves) auto

```

qed

lemma *run-one-step-basic-if-result:*

```
assumes run-one-step d i (s,vs,ves,$(If x71 x72 x73)) = (s', vs', res)
shows (∃ r. res = RSNormal r) ∨ (∃ e. res = RSCrash e)
using assms
proof (cases ves)
case (Cons a list)
thus ?thesis
  using assms
proof (cases a)
  fix x1a
  assume a = ConstInt32 x1a
  thus ?thesis
    using assms Cons
    by (cases int-eq x1a 0) auto
qed auto
qed auto
```

lemma *run-one-step-basic-br-result:*

```
assumes run-one-step d i (s,vs,ves,$Br x8) = (s', vs', res)
shows ∃ r vrs. res = RSBreak r vrs
using assms
by (cases ves) auto
```

lemma *run-one-step-basic-br-if-result:*

```
assumes run-one-step d i (s,vs,ves,$Br-if x9) = (s', vs', res)
shows (∃ r. res = RSNormal r) ∨ (∃ e. res = RSCrash e)
using assms
proof (cases ves)
case (Cons a list)
thus ?thesis
  using assms
proof (cases a)
  case (ConstInt32 x1)
  thus ?thesis
    using assms Cons
    by (cases int-eq x1 0) auto
qed auto
qed auto
```

lemma *run-one-step-basic-br-table-result:*

```
assumes run-one-step d i (s,vs,ves,$Br-table js j) = (s', vs', res)
shows (∃ r. res = RSNormal r) ∨ (∃ e. res = RSCrash e)
using assms
proof (cases ves)
case (Cons a list)
thus ?thesis
  using assms
```

```

proof (cases a)
  case (ConstInt32 x1)
  thus ?thesis
  using assms Cons
  by (cases nat-of-int x1 < length js) auto
qed auto
qed auto

```

```

lemma run-one-step-basic-return-result:
  assumes run-one-step d i (s,vs,ves,$Return) = (s', vs', res)
  shows  $\exists$  vrs. res = RSReturn vrs
  using assms
  by (cases ves) auto

```

```

lemma run-one-step-basic-call-result:
  assumes run-one-step d i (s,vs,ves,$Call x12) = (s', vs', res)
  shows  $\exists$  r. res = RSNormal r
  using assms
  by (cases ves) auto

```

```

lemma run-one-step-basic-call-indirect-result:
  assumes run-one-step d i (s,vs,ves,$Call-indirect x13) = (s', vs', res)
  shows ( $\exists$  r. res = RSNormal r)  $\vee$  ( $\exists$  e. res = RSCrash e)
  using assms
proof (cases ves)
  case (Cons a list)
  thus ?thesis
  using assms
proof (cases a)
  case (ConstInt32 x1)
  thus ?thesis
  using Cons assms
proof (cases stab s i (nat-of-int x1))
  case (Some cl)
  thus ?thesis
  using Cons assms ConstInt32
  by (cases cl; cases stypes s i x13 = cl-type cl) auto
qed auto
qed auto
qed auto

```

```

lemma run-one-step-basic-get-local-result:
  assumes run-one-step d i (s,vs,ves,$Get-local x14) = (s', vs', res)
  shows ( $\exists$  r. res = RSNormal r)  $\vee$  ( $\exists$  e. res = RSCrash e)
  using assms
  by (cases x14 < length vs) auto

```

```

lemma run-one-step-basic-set-local-result:
  assumes run-one-step d i (s,vs,ves,$Set-local x15) = (s', vs', res)

```

```

shows ( $\exists r. res = RSNormal\ r$ )  $\vee$  ( $\exists e. res = RSCrash\ e$ )
using assms
by (cases ves; cases x15 < length vs) auto

lemma run-one-step-basic-tee-local-result:
assumes run-one-step d i (s,vs,ves,$Tee-local x16) = (s', vs', res)
shows ( $\exists r. res = RSNormal\ r$ )  $\vee$  ( $\exists e. res = RSCrash\ e$ )
using assms
by (cases ves) auto

lemma run-one-step-basic-get-global-result:
assumes run-one-step d i (s,vs,ves,$Get-global x17) = (s', vs', res)
shows ( $\exists r. res = RSNormal\ r$ )  $\vee$  ( $\exists e. res = RSCrash\ e$ )
using assms
by auto

lemma run-one-step-basic-set-global-result:
assumes run-one-step d i (s,vs,ves,$Set-global x18) = (s', vs', res)
shows ( $\exists r. res = RSNormal\ r$ )  $\vee$  ( $\exists e. res = RSCrash\ e$ )
using assms
by (cases ves) auto

lemma run-one-step-basic-load-result:
assumes run-one-step d i (s,vs,ves,$Load x191 x192 x193 x194) = (s', vs', res)
shows ( $\exists r. res = RSNormal\ r$ )  $\vee$  ( $\exists e. res = RSCrash\ e$ )
proof (cases x192)
  case None
    thus ?thesis
      using assms
    proof (cases ves)
      case (Cons a list)
        thus ?thesis
          using assms None
        proof (cases smem-ind s i; cases a)
          fix aa x1
          assume smem-ind s i = Some aa and a = ConstInt32 x1
          thus ?thesis
            using assms None Cons
            by (cases load (s.mem s ! aa) (nat-of-int x1) x194 (t-length x191)) auto
          qed auto
        qed auto
      next
        case (Some a)
          thus ?thesis
            using assms
          proof (cases ves)
            case (Cons a' list)
              thus ?thesis
                using assms Some

```

```

proof (cases smem-ind s i; cases a; cases a')
  fix aa x y x1
  assume smem-ind s i = Some aa and a = (x, y) and a' = ConstInt32 x1
  thus ?thesis
  using assms Some Cons
  by (cases load-packed y (s.mem s ! aa) (nat-of-int x1) x194 (tp-length x)
(t-length x191)) auto
  qed auto
qed auto
qed

lemma run-one-step-basic-store-result:
  assumes run-one-step d i (s,vs,ves,$Store x201 x202 x203 x204) = (s', vs', res)
  shows ( $\exists r. res = RSNormal r$ )  $\vee$  ( $\exists e. res = RSCrash e$ )
proof (cases x202)
  case None
  thus ?thesis
  using assms
proof (cases ves)
  case (Cons a list)
  note outer-Cons = Cons
  thus ?thesis
  using assms None outer-Cons
proof (cases list)
  case (Cons a' list')
  thus ?thesis
  using assms None outer-Cons
proof (cases a'; cases types-agree x201 a; cases smem-ind s i)
  fix k aa
  assume a' = ConstInt32 k and types-agree x201 a and smem-ind s i =
Some aa
  thus ?thesis
  using assms None outer-Cons Cons
  by (cases store (s.mem s ! aa) (nat-of-int k) x204 (bits a) (t-length x201))
auto
  qed auto
  qed auto
qed auto
next
case (Some a'')
thus ?thesis
  using assms
proof (cases ves)
  case (Cons a list)
  note outer-Cons = Cons
  thus ?thesis
  using assms Some
proof (cases list)
  case (Cons a' list')

```

```

thus ?thesis
using assms Some outer-Cons
proof (cases a'; cases types-agree x201 a; cases smem-ind s i)
  fix k aa
    assume a' = ConstInt32 k and types-agree x201 a and smem-ind s i =
Some aa
    thus ?thesis
    using assms Some outer-Cons Cons
    by (cases store-packed (s.mem s ! aa) (nat-of-int k) x204 (bits a) (tp-length
a')) auto
  qed auto
qed auto
qed auto
qed

```

```

lemma run-one-step-basic-current-memory-result:
assumes run-one-step d i (s,vs,ves,$Current-memory) = (s', vs', res)
shows  $(\exists r. res = RSNormal\ r) \vee (\exists e. res = RSCrash\ e)$ 
using assms
by (cases smem-ind s i) auto

```

```

lemma run-one-step-basic-grow-memory-result:
assumes run-one-step d i (s,vs,ves,$Grow-memory) = (s', vs', res)
shows  $(\exists r. res = RSNormal\ r) \vee (\exists e. res = RSCrash\ e)$ 
using assms
proof (cases ves)
case (Cons a list)
thus ?thesis
  using assms
proof (cases a; cases smem-ind s i)
  fix c a'
    assume a = ConstInt32 c and smem-ind s i = Some a'
    thus ?thesis
    using assms Cons
    by (cases mem-grow-impl (s.mem s ! a') (nat-of-int c)) auto
  qed auto
qed auto

```

```

lemma run-one-step-basic-const-result:
assumes run-one-step d i (s,vs,ves,$EConst x23) = (s', vs', res)
shows  $\exists e. res = RSCrash\ e$ 
using assms
by auto

```

```

lemma run-one-step-basic-unop-i-result:
assumes run-one-step d i (s,vs,ves,$Unop-i x241 x242) = (s', vs', res)
shows  $(\exists r. res = RSNormal\ r) \vee (\exists e. res = RSCrash\ e)$ 
using assms
proof (cases ves)

```

```

case (Cons a list)
thus ?thesis
  using assms
  by (cases x241; cases a) auto
qed (cases x241; auto)

```

```

lemma run-one-step-basic-unop-f-result:
assumes run-one-step d i (s,vs,ves,$Unop-f x251 x252) = (s', vs', res)
shows  $(\exists r. \text{res} = \text{RSNormal } r) \vee (\exists e. \text{res} = \text{RSCrash } e)$ 
using assms
proof (cases ves)
  case (Cons a list)
  thus ?thesis
    using assms
    by (cases x251; cases a) auto
qed (cases x251; auto)

```

```

lemma run-one-step-basic-binop-i-result:
assumes run-one-step d i (s,vs,ves,$Binop-i x261 x262) = (s', vs', res)
shows  $(\exists r. \text{res} = \text{RSNormal } r) \vee (\exists e. \text{res} = \text{RSCrash } e)$ 
using assms
proof (cases ves)
  case (Cons a list)
  note outer-Cons = Cons
  thus ?thesis
    using assms
  proof (cases list)
    case (Cons a' list')
    thus ?thesis
      using assms outer-Cons
  proof (cases x261; cases a; cases a')
    fix x1 x2
    assume  $x261 = T-i32\ a = \text{ConstInt32 } x1$  and  $a' = \text{ConstInt32 } x2$ 
    thus ?thesis
      using assms outer-Cons Cons
      by (cases app-binop-i x262 x2 x1) auto
  next
    fix x1 x2
    assume  $x261 = T-i64\ a = \text{ConstInt64 } x1$  and  $a' = \text{ConstInt64 } x2$ 
    thus ?thesis
      using assms outer-Cons Cons
      by (cases app-binop-i x262 x2 x1) auto
  qed auto
qed (cases x261; cases a; auto)
qed (cases x261; auto)

```

```

lemma run-one-step-basic-binop-f-result:
assumes run-one-step d i (s,vs,ves,$Binop-f x271 x272) = (s', vs', res)
shows  $(\exists r. \text{res} = \text{RSNormal } r) \vee (\exists e. \text{res} = \text{RSCrash } e)$ 

```

```

using assms
proof (cases ves)
  case (Cons a list)
  note outer-Cons = Cons
  thus ?thesis
    using assms
  proof (cases list)
    case (Cons a' list')
    thus ?thesis
      using assms outer-Cons
  proof (cases x271; cases a; cases a')
    fix x1 x2
    assume x271 = T-f32 a = ConstFloat32 x1 and a' = ConstFloat32 x2
    thus ?thesis
      using assms outer-Cons Cons
      by (cases app-binop-f x272 x2 x1) auto
  next
    fix x1 x2
    assume x271 = T-f64 a = ConstFloat64 x1 and a' = ConstFloat64 x2
    thus ?thesis
      using assms outer-Cons Cons
      by (cases app-binop-f x272 x2 x1) auto
  qed auto
qed (cases x271; cases a; auto)
qed (cases x271; auto)

```

```

lemma run-one-step-basic-testop-result:
  assumes run-one-step d i (s,vs,ves,$Testop x281 x282) = (s', vs', res)
  shows  $(\exists r. res = RSNormal\ r) \vee (\exists e. res = RSCrash\ e)$ 
  using assms
proof (cases ves)
  case (Cons a list)
  thus ?thesis
    using assms
    by (cases x281; cases a) auto
qed (cases x281; auto)

```

```

lemma run-one-step-basic-relop-i-result:
  assumes run-one-step d i (s,vs,ves,$Relop-i x291 x292) = (s', vs', res)
  shows  $(\exists r. res = RSNormal\ r) \vee (\exists e. res = RSCrash\ e)$ 
  using assms
proof (cases ves)
  case (Cons a list)
  note outer-Cons = Cons
  thus ?thesis
    using assms
  proof (cases list)
    case (Cons a' list')
    thus ?thesis

```

```

    using assms outer-Cons
  proof (cases x291; cases a; cases a')
    fix x1 x2
    assume x291 = T-i32 a = ConstInt32 x1 and a' = ConstInt32 x2
    thus ?thesis
      using assms outer-Cons Cons
      by (cases app-relop-i x292 x2 x1) auto
  next
    fix x1 x2
    assume x291 = T-i64 a = ConstInt64 x1 and a' = ConstInt64 x2
    thus ?thesis
      using assms outer-Cons Cons
      by (cases app-relop-i x292 x2 x1) auto
  qed auto
qed (cases x291; cases a; auto)
qed (cases x291; auto)

lemma run-one-step-basic-relop-f-result:
  assumes run-one-step d i (s,vs,ves,$Relop-f x301 x302) = (s', vs', res)
  shows (∃ r. res = RSNormal r) ∨ (∃ e. res = RSCrash e)
  using assms
proof (cases ves)
  case (Cons a list)
  note outer-Cons = Cons
  thus ?thesis
    using assms
  proof (cases list)
    case (Cons a' list')
    thus ?thesis
      using assms outer-Cons
  proof (cases x301; cases a; cases a')
    fix x1 x2
    assume x301 = T-f32 a = ConstFloat32 x1 and a' = ConstFloat32 x2
    thus ?thesis
      using assms outer-Cons Cons
      by (cases app-relop-f x302 x2 x1) auto
  next
    fix x1 x2
    assume x301 = T-f64 a = ConstFloat64 x1 and a' = ConstFloat64 x2
    thus ?thesis
      using assms outer-Cons Cons
      by (cases app-relop-f x302 x2 x1) auto
  qed auto
qed (cases x301; cases a; auto)
qed (cases x301; auto)

lemma run-one-step-basic-cvtop-result:
  assumes run-one-step d i (s,vs,ves,$Cvtop t2 x312 t1 sx) = (s', vs', res)
  shows (∃ r. res = RSNormal r) ∨ (∃ e. res = RSCrash e)

```

```

using assms
proof (cases ves; cases x312)
  fix a ves'
  assume ves = a#ves' and x312 = Convert
  thus ?thesis
    using assms
    by (cases cvt t2 sx a; cases types-agree t1 a) auto
next
  fix a ves'
  assume ves = a#ves' and x312 = Reinterpret
  thus ?thesis
    using assms
    by (cases sx; cases types-agree t1 a) auto
qed auto

lemma run-one-step-trap-result:
  assumes run-one-step d i (s,vs,ves,Trap) = (s', vs', res)
  shows  $\exists e. res = RSCrash\ e$ 
  using assms
  by auto

lemma run-one-step-callcl-result:
  assumes run-one-step d i (s,vs,ves,Callcl cl) = (s', vs', res)
  shows  $(\exists r. res = RSNormal\ r) \vee (\exists e. res = RSCrash\ e)$ 
proof -
  obtain t1s t2s where cl-type-is:cl-type cl = (t1s -> t2s)
    using tf.exhaust
    by blast
  obtain ves' ves'' where split-n-is:split-n ves (length t1s) = (ves', ves'')
    by fastforce
  show ?thesis
proof (cases cl)
  case (Func-native x11 x12 x13 x14)
  thus ?thesis
    using assms cl-type-is split-n-is
    unfolding cl-type-def
    by (cases length t1s ≤ length ves) auto
next
  case (Func-host x21 x22)
  show ?thesis
proof (cases host-apply-impl s (t1s -> t2s) x22 (rev ves'))
  case None
  thus ?thesis
    using assms cl-type-is split-n-is Func-host
    unfolding cl-type-def
    by (cases length t1s ≤ length ves) auto
next
  case (Some a)
  thus ?thesis

```

```

proof (cases a)
  case (Pair s' vcs')
  thus ?thesis
    using assms cl-type-is split-n-is Func-host Some
    unfolding cl-type-def
    by (cases length t1s ≤ length ves; cases list-all2 types-agree t2s vcs') auto
qed
qed
qed
qed

```

lemma *run-one-step-label-result*:

```

assumes run-one-step d i (s,vs,ves,Label x41 x42 x43) = (s', vs', res)
shows (∃ r. res = RSNormal r) ∨ (∃ r rvs. res = RSBreak r rvs) ∨ (∃ rvs. res =
RSReturn rvs) ∨ (∃ e. res = RSCrash e)
using assms
by (cases res) auto

```

lemma *run-one-step-local-result*:

```

assumes run-one-step d i (s,vs,ves,Local x51 x52 x53 x54) = (s', vs', res)
shows (∃ r. res = RSNormal r) ∨ (∃ e. res = RSCrash e)
using assms

```

```

proof (cases x54 = [Trap])

```

```

  case False

```

```

    note outer-False = False

```

```

    thus ?thesis

```

```

  proof (cases const-list x54)

```

```

    case True

```

```

    thus ?thesis

```

```

      using assms outer-False

```

```

      by (cases length x54 = x51) auto

```

```

  next

```

```

    case False

```

```

    thus ?thesis

```

```

      using assms outer-False

```

```

  proof (cases d)

```

```

    case (Suc d')

```

```

    obtain s' vs' res where rs-def:run-step d' x52 (s, x53, x54) = (s', vs', res)

```

```

    by (metis surj-pair)

```

```

    thus ?thesis

```

```

      using assms outer-False False Suc

```

```

  proof (cases res)

```

```

    case (RSReturn x3)

```

```

    thus ?thesis

```

```

      using assms outer-False False rs-def Suc

```

```

      by (cases x51 ≤ length x3) auto

```

```

  qed auto

```

```

qed auto

```

```

qed

```

qed *auto*

lemma *run-one-step-break:*

assumes *run-one-step d i (s,vs,ves,e) = (s', vs', RSBreak n res)*

shows $(e = \$Br\ n) \vee (\exists n\ les\ es.\ e = Label\ n\ les\ es)$

proof (*cases e*)

case (*Basic x1*)

thus *?thesis*

proof (*cases x1*)

case *Unreachable*

thus *?thesis*

using *run-one-step-basic-unreachable-result assms Basic*

by *fastforce*

next

case *Nop*

thus *?thesis*

using *assms Basic*

by *fastforce*

next

case *Drop*

thus *?thesis*

using *run-one-step-basic-drop-result assms Basic*

by *fastforce*

next

case *Select*

thus *?thesis*

using *run-one-step-basic-select-result assms Basic*

by *fastforce*

next

case (*Block x51 x52*)

thus *?thesis*

using *run-one-step-basic-block-result assms Basic*

by *fastforce*

next

case (*Loop x61 x62*)

thus *?thesis*

using *run-one-step-basic-loop-result assms Basic*

by *fastforce*

next

case (*If x71 x72 x73*)

thus *?thesis*

using *run-one-step-basic-if-result assms Basic*

by *fastforce*

next

case (*Br x8*)

thus *?thesis*

using *run-one-step-basic-br-result assms Basic*

by *fastforce*

next

```

case (Br-if x9)
thus ?thesis
  using run-one-step-basic-br-if-result assms Basic
  by fastforce
next
case (Br-table x10)
thus ?thesis
  using run-one-step-basic-br-table-result assms Basic
  by fastforce
next
case Return
thus ?thesis
  using run-one-step-basic-return-result assms Basic
  by fastforce
next
case (Call x12)
thus ?thesis
  using run-one-step-basic-call-result assms Basic
  by fastforce
next
case (Call-indirect x13)
thus ?thesis
  using run-one-step-basic-call-indirect-result assms Basic
  by fastforce
next
case (Get-local x14)
thus ?thesis
  using run-one-step-basic-get-local-result assms Basic
  by fastforce
next
case (Set-local x15)
thus ?thesis
  using run-one-step-basic-set-local-result assms Basic
  by fastforce
next
case (Tee-local x16)
thus ?thesis
  using run-one-step-basic-tee-local-result assms Basic
  by fastforce
next
case (Get-global x17)
thus ?thesis
  using assms Basic
  by fastforce
next
case (Set-global x18)
thus ?thesis
  using run-one-step-basic-set-global-result assms Basic
  by fastforce

```

```

next
  case (Load x191 x192 x193 x194)
  thus ?thesis
    using run-one-step-basic-load-result assms Basic
    by fastforce
next
  case (Store x201 x202 x203 x204)
  thus ?thesis
    using run-one-step-basic-store-result assms Basic
    by fastforce
next
  case Current-memory
  thus ?thesis
    using run-one-step-basic-current-memory-result assms Basic
    by fastforce
next
  case Grow-memory
  thus ?thesis
    using run-one-step-basic-grow-memory-result assms Basic
    by fastforce
next
  case (EConst x23)
  thus ?thesis
    using assms Basic
    by fastforce
next
  case (Unop-i x241 x242)
  thus ?thesis
    using run-one-step-basic-unop-i-result assms Basic
    by fastforce
next
  case (Unop-f x251 x252)
  thus ?thesis
    using run-one-step-basic-unop-f-result assms Basic
    by fastforce
next
  case (Binop-i x261 x262)
  thus ?thesis
    using run-one-step-basic-binop-i-result assms Basic
    by fastforce
next
  case (Binop-f x271 x272)
  thus ?thesis
    using run-one-step-basic-binop-f-result assms Basic
    by fastforce
next
  case (Testop x281 x282)
  thus ?thesis
    using run-one-step-basic-testop-result assms Basic

```

```

    by fastforce
next
case (Relop-i x291 x292)
thus ?thesis
using run-one-step-basic-relop-i-result assms Basic
by fastforce
next
case (Relop-f x301 x302)
thus ?thesis
using run-one-step-basic-relop-f-result assms Basic
by fastforce
next
case (Cvtop x311 x312 x313 x314)
thus ?thesis
using run-one-step-basic-cvtop-result assms Basic
by fastforce
qed
next
case Trap
thus ?thesis
using assms
by auto
next
case (Callcl x3)
thus ?thesis
using assms run-one-step-callcl-result
by fastforce
next
case (Label x41 x42 x43)
thus ?thesis
by auto
next
case (Local x51 x52 x53 x54)
thus ?thesis
using assms run-one-step-local-result
by fastforce
qed

lemma run-one-step-return:
  assumes run-one-step d i (s,vs,ves,e) = (s', vs', RSReturn res)
  shows (e = $Return)  $\vee$  ( $\exists n$  les es. e = Label n les es)
proof (cases e)
case (Basic x1)
thus ?thesis
proof (cases x1)
case Unreachable
thus ?thesis
using run-one-step-basic-unreachable-result assms Basic
by fastforce

```

```

next
  case Nop
  thus ?thesis
    using assms Basic
    by fastforce
next
  case Drop
  thus ?thesis
    using run-one-step-basic-drop-result assms Basic
    by fastforce
next
  case Select
  thus ?thesis
    using run-one-step-basic-select-result assms Basic
    by fastforce
next
  case (Block x51 x52)
  thus ?thesis
    using run-one-step-basic-block-result assms Basic
    by fastforce
next
  case (Loop x61 x62)
  thus ?thesis
    using run-one-step-basic-loop-result assms Basic
    by fastforce
next
  case (If x71 x72 x73)
  thus ?thesis
    using run-one-step-basic-if-result assms Basic
    by fastforce
next
  case (Br x8)
  thus ?thesis
    using run-one-step-basic-br-result assms Basic
    by fastforce
next
  case (Br-if x9)
  thus ?thesis
    using run-one-step-basic-br-if-result assms Basic
    by fastforce
next
  case (Br-table x10)
  thus ?thesis
    using run-one-step-basic-br-table-result assms Basic
    by fastforce
next
  case Return
  thus ?thesis
    using run-one-step-basic-return-result assms Basic

```

```

    by fastforce
next
case (Call x12)
thus ?thesis
using run-one-step-basic-call-result assms Basic
by fastforce
next
case (Call-indirect x13)
thus ?thesis
using run-one-step-basic-call-indirect-result assms Basic
by fastforce
next
case (Get-local x14)
thus ?thesis
using run-one-step-basic-get-local-result assms Basic
by fastforce
next
case (Set-local x15)
thus ?thesis
using run-one-step-basic-set-local-result assms Basic
by fastforce
next
case (Tee-local x16)
thus ?thesis
using run-one-step-basic-tee-local-result assms Basic
by fastforce
next
case (Get-global x17)
thus ?thesis
using assms Basic
by fastforce
next
case (Set-global x18)
thus ?thesis
using run-one-step-basic-set-global-result assms Basic
by fastforce
next
case (Load x191 x192 x193 x194)
thus ?thesis
using run-one-step-basic-load-result assms Basic
by fastforce
next
case (Store x201 x202 x203 x204)
thus ?thesis
using run-one-step-basic-store-result assms Basic
by fastforce
next
case Current-memory
thus ?thesis

```

```

    using run-one-step-basic-current-memory-result assms Basic
  by fastforce
next
case Grow-memory
thus ?thesis
  using run-one-step-basic-grow-memory-result assms Basic
  by fastforce
next
case (EConst x23)
thus ?thesis
  using assms Basic
  by fastforce
next
case (Unop-i x241 x242)
thus ?thesis
  using run-one-step-basic-unop-i-result assms Basic
  by fastforce
next
case (Unop-f x251 x252)
thus ?thesis
  using run-one-step-basic-unop-f-result assms Basic
  by fastforce
next
case (Binop-i x261 x262)
thus ?thesis
  using run-one-step-basic-binop-i-result assms Basic
  by fastforce
next
case (Binop-f x271 x272)
thus ?thesis
  using run-one-step-basic-binop-f-result assms Basic
  by fastforce
next
case (Testop x281 x282)
thus ?thesis
  using run-one-step-basic-testop-result assms Basic
  by fastforce
next
case (Relop-i x291 x292)
thus ?thesis
  using run-one-step-basic-relop-i-result assms Basic
  by fastforce
next
case (Relop-f x301 x302)
thus ?thesis
  using run-one-step-basic-relop-f-result assms Basic
  by fastforce
next
case (Cvtop x311 x312 x313 x314)

```

```

    thus ?thesis
      using run-one-step-basic-cvtop-result assms Basic
      by fastforce
  qed
next
  case Trap
  thus ?thesis
    using assms
    by auto
next
  case (Callcl x3)
  thus ?thesis
    using assms run-one-step-callcl-result
    by fastforce
next
  case (Label x41 x42 x43)
  thus ?thesis
    by auto
next
  case (Local x51 x52 x53 x54)
  thus ?thesis
    using assms run-one-step-local-result
    by fastforce
qed

lemma run-step-break-imp-not-trap-const-list:
  assumes run-step d i (s, vs, es) = (s', vs', RSBreak n res)
  shows es ≠ [Trap] ¬const-list es
proof -
  {
    assume es = [Trap]
    hence False
      using assms
      by simp
  }
  thus es ≠ [Trap]
    by blast
  {
    assume const-list es
    then obtain vs where split-vals-e es = (vs, [])
      using split-vals-e-const-list e-type-const-conv-vs
      by fastforce
    hence False
      using assms
      by simp
  }
  thus ¬const-list es
    by blast
qed

```

lemma *run-step-return-imp-not-trap-const-list*:
assumes *run-step d i (s, vs, es) = (s', vs', RSReturn res)*
shows *es ≠ [Trap] ¬const-list es*
proof –
{
 assume *es = [Trap]*
 hence *False*
 using *assms*
 by *simp*
}
thus *es ≠ [Trap]*
 by *blast*
{
 assume *const-list es*
 then obtain *vs* **where** *split-vals-e es = (vs, [])*
 using *split-vals-e-const-list e-type-const-conv-vs*
 by *fastforce*
 hence *False*
 using *assms*
 by *simp*
}
thus *¬const-list es*
 by *blast*
qed

lemma *run-one-step-label-break-imp-break*:
assumes *run-one-step d i (s, vs, ves, Label ln les es) = (s', vs', RSBreak n res)*
shows *run-step d i (s, vs, es) = (s', vs', RSBreak (Suc n) res)*
using *assms*
proof (*cases es = [Trap]; cases const-list es*)
 assume *local-assms: es ≠ [Trap] ¬const-list es*
 obtain *s'' vs'' res''* **where** *rs-def:run-step d i (s, vs, es) = (s'', vs'', res'')*
 by (*metis surj-pair*)
 thus *?thesis*
 using *assms local-assms*
 proof (*cases res''*)
 case (*RSBreak x21 x22*)
 thus *?thesis*
 using *assms local-assms rs-def*
 by (*cases x21; cases ln ≤ length x22*) *auto*
 qed *auto*
qed *auto*

lemma *run-one-step-label-return-imp-return*:
assumes *run-one-step d i (s, vs, ves, Label n les es) = (s', vs', RSReturn res)*
shows *run-step d i (s, vs, es) = (s', vs', RSReturn res)*
using *assms*
proof (*cases es = [Trap]; cases const-list es*)

```

assume local-assms:es ≠ [Trap] ¬const-list es
obtain s'' vs'' res'' where rs-def:run-step d i (s, vs, es) = (s'', vs'', res'')
  by (metis surj-pair)
thus ?thesis
  using assms local-assms
proof (cases res'')
  case (RSBreak x21 x22)
  thus ?thesis
    using assms local-assms rs-def
    by (cases x21; cases n ≤ length x22) auto
qed auto
qed auto
thm run-step-run-one-step.induct

```

definition *run-step-break-imp-lfilled-prop* **where**

```

run-step-break-imp-lfilled-prop s' vs' n res =
  (λd i (s,vs,es). (run-step d i (s,vs,es) = (s', vs', RSBreak n res)) →
    s = s' ∧ vs = vs' ∧
    (∃ n' lfilled es-c. n' ≥ n ∧ Lfilled-exact (n'-n) lfilled ((vs-to-es res) @ [Br n'] @ es-c) es))

```

definition *run-one-step-break-imp-lfilled-prop* **where**

```

run-one-step-break-imp-lfilled-prop s' vs' n res =
  (λd i (s,vs,ves,e). run-one-step d i (s,vs,ves,e) = (s', vs', RSBreak n res) →
    s = s' ∧ vs = vs' ∧ ((res = ves ∧ e = Br n) ∨ (∃ n' lfilled es-c es les' ln. n' > n ∧ Lfilled-exact (n'-(n+1)) lfilled ((vs-to-es res) @ [Br n'] @ es-c) es ∧ e = Label ln les' es)))

```

lemma *run-step-break-imp-lfilled:*

```

assumes run-step d i (s,vs,es) = (s', vs', RSBreak n res)
shows s = s' ∧
  vs = vs' ∧
  (∃ n' lfilled es-c. n' ≥ n ∧
    Lfilled-exact (n'-n) lfilled ((vs-to-es res) @ [Br n'] @ es-c)
es)

```

proof –

```

fix ves e
have (run-step-break-imp-lfilled-prop s' vs' n res) d i (s,vs,es)
and (run-one-step-break-imp-lfilled-prop s' vs' n res) d i (s,vs,ves,e)
proof (induction d i (s,vs,es) and d i (s,vs,ves,e) arbitrary: n es and n ves e
rule: run-step-run-one-step.induct)
  case (1 d i es)
  {
    assume local-assms:run-step d i (s,vs,es) = (s', vs', RSBreak n res)
    obtain ves es' where split-vals-es:split-vals-e es = (ves, es')
    by (metis surj-pair)
    then obtain a as where es'-def:es' = a#as
    using local-assms

```

```

    by (cases es') auto
  hence a-def:a ≠ Trap
    using local-assms split-vals-es
    by (cases a = Trap; cases (as ≠ [] ∨ ves ≠ [])) simp-all
  obtain s'' vs'' res'' where run-one-step d i (s,vs,(rev ves),a) = (s'', vs'', res'')
    by (metis surj-pair)
  hence ros-def:run-one-step d i (s,vs,(rev ves),a) = (s', vs', RSBreak n res)
    using local-assms split-vals-es es'-def a-def
    by (cases res'') (auto simp del: run-one-step.simps)
  hence run-one-step-break-imp-lfilled-prop s' vs' n res d i (s, vs, rev ves, a)
    using 1 split-vals-es a-def es'-def
    by fastforce
  then obtain n' lfilled es-c les les' ln where
    s = s' vs = vs'
    ((res = (rev ves) ∧ a = $Br n) ∨
     n' > n ∧ (Lfilled-exact (n'-(n+1)) lfilled ((vs-to-es res) @ [$Br n'] @
    es-c) les ∧ a = Label ln les' les))
    using ros-def
    unfolding run-one-step-break-imp-lfilled-prop-def
    by fastforce
  then consider
    (1) s = s' vs = vs' res = (rev ves) a = $Br n
  | (2) s = s' vs = vs' n' > n Lfilled-exact (n'-(n+1)) lfilled ((vs-to-es res) @
  [$Br n'] @ es-c) les a = Label ln les' les
    by blast
  hence s = s' ∧ vs = vs' ∧
    (∃ n' lfilled es-c. n' ≥ n ∧ Lfilled-exact (n'-n) lfilled ((vs-to-es res) @
  [$Br n'] @ es-c) es)
  proof cases
  case 1
  thus ?thesis
    using es'-def split-vals-e-conv-app[OF split-vals-es] Lfilled-exact.intros(1)
    is-const-list[of - ves]
    by fastforce
  next
  case 2
  have test:const-list ($$* ves)
    using is-const-list
    by auto
  have (Suc (n' - Suc n)) = n' - n
    using 2(3)
    by simp
  thus ?thesis
    using 2(1,2,3,5) Lfilled-exact.intros(2)[OF test - 2(4), of - ln les' as] es'-def
    split-vals-e-conv-app[OF split-vals-es]
    by (metis Suc-eq-plus1 append-Cons append-Nil less-imp-le-nat)
  qed
}
thus ?case

```

```

unfolding run-step-break-imp-lfilled-prop-def
by fastforce
next
case (2 d i ves e)
{
assume local-assms:run-one-step d i (s,vs,ves,e) = (s', vs', RSBreak n res)
consider (a) e = $Br n | (b)  $(\exists n \text{ les } es. e = \text{Label } n \text{ les } es)$ 
using run-one-step-break[OF local-assms]
by blast
hence  $s = s' \wedge vs = vs' \wedge ((res = ves \wedge e = \$Br n) \vee (\exists n' \text{ lfilled } es\text{-c } es \text{ les}'$ 
ln. n' > n \wedge Lfilled-exact (n'-(n+1)) lfilled ((vs-to-es res) @ [$Br n'] @ es-c) es
 $\wedge e = \text{Label } ln \text{ les}' es))$ 
proof cases
case a
thus ?thesis
using local-assms
by simp
next
case b
then obtain ln les es where e-def:e = Label ln les es
by blast
hence run-one-step d i (s, vs, ves, Label ln les es) = (s', vs', RSBreak n res)
using local-assms by simp
hence rs-def:run-step d i (s, vs, es) = (s', vs', RSBreak (Suc n) res)
using run-one-step-label-break-imp-break
by fastforce
hence run-step-break-imp-lfilled-prop s' vs' (Suc n) res d i (s, vs, es)
using  $2(1)[OF \text{e-def} - \text{run-step-break-imp-not-trap-const-list}(2)]$ 
by fastforce
thus ?thesis
using e-def rs-def
unfolding run-step-break-imp-lfilled-prop-def
by fastforce
qed
}
thus ?case
unfolding run-one-step-break-imp-lfilled-prop-def
by fastforce
qed
thus ?thesis
using assms
unfolding run-step-break-imp-lfilled-prop-def
by fastforce
qed

lemma run-step-return-imp-lfilled:
assumes run-step d i (s,vs,es) = (s', vs', RSReturn res)
shows  $s = s' \wedge vs = vs' \wedge (\exists n \text{ lfilled } es\text{-c. Lfilled-exact } n \text{ lfilled } ((vs\text{-to-es } res)$ 
 $\text{@ } [\$Return] \text{@ } es\text{-c}) es)$ 

```

```

proof –
  fix ves e
  have (run-step d i (s,vs,es) = (s', vs', RSReturn res))  $\implies$ 
     $s = s' \wedge vs = vs' \wedge (\exists n \text{ lfilled } es\text{-c}. \text{Lfilled-exact } n \text{ lfilled } ((vs\text{-to-es } res) @$ 
     $[\$Return] @ es\text{-c}) es)$ 
  and (run-one-step d i (s,vs,ves,e) = (s', vs', RSReturn res))  $\implies$ 
     $s = s' \wedge vs = vs' \wedge$ 
     $((res = ves \wedge e = \$Return) \vee$ 
     $(\exists n \text{ lfilled } ves \text{ es-c } es \text{ n}' \text{ les}'. \text{Lfilled-exact } n \text{ lfilled } ((vs\text{-to-es } res) @$ 
     $[\$Return] @ es\text{-c}) es \wedge$ 
     $e = \text{Label } n' \text{ les}' es))$ 
  proof (induction d i (s,vs,es) and d i (s,vs,ves,e) arbitrary: s vs es s' vs' res
and s vs ves e s' vs' res rule: run-step-run-one-step.induct)
  case (1 d i s vs es)
  obtain ves es' where split-vals-es:split-vals-e es = (ves, es')
  by (metis surj-pair)
  then obtain a as where es'-def:es' = a#as
  using 1(2)
  by (cases es') auto
  hence a-def:¬ e-is-trap a
  using 1(2) split-vals-es
  by (cases a = Trap; cases (as ≠ [] ∨ ves ≠ []) simp-all)
  obtain s'' vs'' res'' where run-one-step d i (s,vs,(rev ves),a) = (s'', vs'', res'')
  by (metis surj-pair)
  hence ros-def:run-one-step d i (s,vs,(rev ves),a) = (s', vs', RSReturn res)
  using 1(2) split-vals-es es'-def a-def
  by (cases res'') (auto simp del: run-one-step.simps)
  obtain n lfilled les-c les n' les' where
     $s = s' \text{ vs} = \text{vs}'$ 
     $(res = \text{rev } ves \wedge a = \$Return) \vee (\text{Lfilled-exact } n \text{ lfilled } ((vs\text{-to-es } res) @$ 
     $[\$Return] @ les\text{-c}) les \wedge a = \text{Label } n' \text{ les}' les)$ 
  using 1(1)[OF split-vals-es[symmetric] - es'-def a-def ros-def]
  by fastforce
  then consider
    (1)  $s = s' \text{ vs} = \text{vs}' \text{ res} = \text{rev } ves \text{ a} = \$Return$ 
    | (2)  $s = s' \text{ vs} = \text{vs}' (\text{Lfilled-exact } n \text{ lfilled } ((vs\text{-to-es } res) @ [\$Return] @ les\text{-c})$ 
    les) (a = Label } n' \text{ les}' les)
  by blast
  thus ?case
  proof cases
  case 1
  thus ?thesis
  using es'-def split-vals-e-conv-app[OF split-vals-es] Lfilled-exact.intros(1)
is-const-list[of - ves]
  by fastforce
  next
  case 2
  have const-list ($$* ves)
  using is-const-list

```

```

    by fastforce
  thus ?thesis
    using 2 Lfilled-exact.intros(2) es'-def split-vals-e-conv-app[OF split-vals-es]
    by fastforce
qed
next
case (2 d i s vs ves e s' vs')
consider (a) e = $Return | (b) (∃ n les es. e = Label n les es)
  using run-one-step-return[OF 2(3)]
  by blast
thus ?case
proof cases
  case a
  thus ?thesis
    using 2(3)
    by simp
next
  case b
  then obtain n les es where e-def:e = Label n les es
    by blast
  hence run-one-step d i (s, vs, ves, Label n les es) = (s', vs', RSReturn res)
    using 2(3) by simp
  hence run-step d i (s, vs, es) = (s', vs', RSReturn res)
    using run-one-step-label-return-imp-return
    by fastforce
  thus ?thesis
    using 2(3) 2(1)[OF e-def - run-step-return-imp-not-trap-const-list(2)] e-def
    by fastforce
qed
qed
thus ?thesis
  using assms
  by blast
qed

lemma run-step-basic-unop-testop-sound:
  assumes (run-one-step d i (s,vs,ves,$b-e) = (s', vs', RSNormal es'))
    b-e = Unop-i t iop ∨ b-e = Unop-f t fop ∨ b-e = Testop t testop
  shows (|s;vs;(vs-to-es ves)@[ $b-e] |) ~~- i (|s';vs';es'|)
proof -
  consider (1) b-e = Unop-i t iop | (2) b-e = Unop-f t fop | (3) b-e = Testop t
  testop
    using assms(2)
    by blast
  note b-e-cases = this
  show ?thesis
    using assms(1)
  proof (cases ves)
    case (Cons a list)

```

```

thus ?thesis
  using assms(1)
proof (cases a; cases t)
  case (ConstInt32 x1)
  case T-i32
  thus ?thesis
    using assms(1) Cons ConstInt32
      is-const-list-vs-to-es-list[of rev list]
      progress-L0-left[OF reduce.intros(1)[OF reduce-simple.intros(1)]]
      progress-L0-left[OF reduce.intros(1)[OF reduce-simple.intros(13)]]
    by (cases rule: b-e-cases) auto
next
  case (ConstInt64 x2)
  case T-i64
  thus ?thesis
    using assms(1) Cons ConstInt64
      is-const-list-vs-to-es-list[of rev list]
      progress-L0-left[OF reduce.intros(1)[OF reduce-simple.intros(2)]]
      progress-L0-left[OF reduce.intros(1)[OF reduce-simple.intros(14)]]
    by (cases rule: b-e-cases) auto
next
  case (ConstFloat32 x3)
  case T-f32
  thus ?thesis
    using assms(1) Cons ConstFloat32
      is-const-list-vs-to-es-list[of rev list]
      progress-L0-left[OF reduce.intros(1)[OF reduce-simple.intros(3)]]
    by (cases rule: b-e-cases) auto
next
  case (ConstFloat64 x4)
  case T-f64
  thus ?thesis
    using assms(1) Cons ConstFloat64
      is-const-list-vs-to-es-list[of rev list]
      progress-L0-left[OF reduce.intros(1)[OF reduce-simple.intros(4)]]
    by (cases rule: b-e-cases) auto
  qed (cases rule: b-e-cases; auto)+
qed (cases rule: b-e-cases; cases t; auto)
qed

```

lemma *run-step-basic-binop-relop-sound*:

```

assumes (run-one-step d i (s,vs,ves,$b-e) = (s', vs', RSNormal es'))
   $b-e = \text{Binop-}i\ t\ iop \vee b-e = \text{Binop-}f\ t\ fop \vee b-e = \text{Relop-}i\ t\ irop \vee b-e = \text{Relop-}f\ t\ frop$ 
shows ( $\langle s;vs;(vs\text{-to-es}\ ves)\@[\$b-e] \rightsquigarrow - i \langle s';vs';es' \rangle$ )
proof –
consider
  (1)  $b-e = \text{Binop-}i\ t\ iop$ 
  | (2)  $b-e = \text{Binop-}f\ t\ fop$ 

```

```

| (3)  $b-e = \text{Relop-}i\ t\ irop$ 
| (4)  $b-e = \text{Relop-}f\ t\ frop$ 
  using assms(2)
  by blast
note  $b-e\text{-cases} = \text{this}$ 
show ?thesis
  using assms(1)
proof (cases ves)
  case outer-Cons:(Cons v1 list)
  thus ?thesis
    using assms(1)
  proof (cases list)
    case (Cons v2 list')
    thus ?thesis
      using assms(1) outer-Cons
  proof (cases v1; cases v2; cases t)
    fix c1 c2
    assume  $v1 = \text{ConstInt32}\ c1$  and  $v2 = \text{ConstInt32}\ c2$  and  $t = T\text{-}i32$ 
    thus ?thesis
      using assms(1) Cons outer-Cons
        is-const-list-vs-to-es-list[of rev list']
        progress-L0-left[OF reduce.intros(1)[OF reduce-simple.intros(5)]]
        progress-L0-left[OF reduce.intros(1)[OF reduce-simple.intros(6)]]
        progress-L0-left[OF reduce.intros(1)[OF reduce-simple.intros(15)]]
      by (cases rule: b-e-cases; cases app-binop-i iop c2 c1) auto
  next
  fix c1 c2
  assume  $v1 = \text{ConstInt64}\ c1$  and  $v2 = \text{ConstInt64}\ c2$  and  $t = T\text{-}i64$ 
  thus ?thesis
    using assms(1) Cons outer-Cons
      is-const-list-vs-to-es-list[of rev list']
      progress-L0-left[OF reduce.intros(1)[OF reduce-simple.intros(7)]]
      progress-L0-left[OF reduce.intros(1)[OF reduce-simple.intros(8)]]
      progress-L0-left[OF reduce.intros(1)[OF reduce-simple.intros(16)]]
    by (cases rule: b-e-cases; cases app-binop-i iop c2 c1) auto
  next
  fix c1 c2
  assume  $v1 = \text{ConstFloat32}\ c1$  and  $v2 = \text{ConstFloat32}\ c2$  and  $t = T\text{-}f32$ 
  thus ?thesis
    using assms(1) Cons outer-Cons
      is-const-list-vs-to-es-list[of rev list']
      progress-L0-left[OF reduce.intros(1)[OF reduce-simple.intros(9)]]
      progress-L0-left[OF reduce.intros(1)[OF reduce-simple.intros(10)]]
      progress-L0-left[OF reduce.intros(1)[OF reduce-simple.intros(17)]]
    by (cases rule: b-e-cases; cases app-binop-f fop c2 c1) auto
  next
  fix c1 c2
  assume  $v1 = \text{ConstFloat64}\ c1$  and  $v2 = \text{ConstFloat64}\ c2$  and  $t = T\text{-}f64$ 
  thus ?thesis

```

```

using assms(1) Cons outer-Cons
  is-const-list-vs-to-es-list[of rev list]
  progress-L0-left[OF reduce.intros(1)[OF reduce-simple.intros(11)]]
  progress-L0-left[OF reduce.intros(1)[OF reduce-simple.intros(12)]]
  progress-L0-left[OF reduce.intros(1)[OF reduce-simple.intros(18)]]
  by (cases rule: b-e-cases; cases app-binop-f fop c2 c1) auto
qed (cases rule: b-e-cases; auto)+
qed (cases rule: b-e-cases; cases t; cases v1; auto)
qed (cases rule: b-e-cases; cases t; auto)
qed

```

lemma *run-step-basic-sound*:

```

assumes (run-one-step d i (s,vs,ves,$b-e) = (s', vs', RSNormal es'))
shows (s;vs;(vs-to-es ves)@[ $b-e ])  $\rightsquigarrow$  i (s';vs';es')

```

proof –

show *?thesis*

proof (*cases b-e*)

case *Unreachable*

thus *?thesis*

using *is-const-list-vs-to-es-list*[of *rev ves*]

progress-L0-left[*OF reduce.intros*(1)[*OF reduce-simple.intros*(22)]]

assms

by *fastforce*

next

case *Nop*

thus *?thesis*

using *is-const-list-vs-to-es-list*[of *rev ves*]

progress-L0-left[*OF reduce.intros*(1)[*OF reduce-simple.intros*(23)]]

assms

by *fastforce*

next

case *Drop*

thus *?thesis*

using *assms*

proof (*cases ves*)

case (*Cons a list*)

hence *vs-to-es ves = vs-to-es list @ [\$C a]*

by *fastforce*

thus *?thesis*

using *is-const-list-vs-to-es-list*[of *rev list*]

progress-L0-left[*OF reduce.intros*(1)[*OF reduce-simple.intros*(24)]]

Drop assms Cons

by *auto*

qed *auto*

next

case *Select*

thus *?thesis*

using *assms*

proof (*cases ves*)

```

case outer-outer-cons:(Cons a list)
thus ?thesis
  using Select assms
proof (cases a; cases list)
  case (ConstInt32 x1a)
  case outer-cons:(Cons a' list')
  thus ?thesis
    using assms outer-outer-cons ConstInt32 Select
  proof (cases list')
    case (Cons a'' list'')
    hence vs-to-es ves = vs-to-es list'' @ [C a'', C a', C ConstInt32 x1a]
      using outer-outer-cons outer-cons ConstInt32
      by fastforce
    thus ?thesis
      using is-const-list-vs-to-es-list[of rev list'']
        progress-L0-left[OF reduce.intros(1)]
        reduce-simple.intros(25,26)
        assms outer-outer-cons outer-cons Cons ConstInt32 Select
      by (cases int-eq x1a 0) auto
    qed auto
  qed auto
qed auto
next
case (Block x51 x52)
thus ?thesis
proof (cases x51)
  case (Tf t1s t2s)
  thus ?thesis
    using Block assms
  proof (cases length t1s ≤ length ves; cases split-n ves (length t1s))
  case True
  case (Pair ves' ves'')
  hence vs-to-es ves = vs-to-es ves'' @ vs-to-es ves'
    using split-n-conv-app
    by fastforce
  moreover
  have s = s' vs = vs' es' = vs-to-es ves'' @ [Label (length t2s) [] (vs-to-es
ves' @ (* x52))]
    using Block assms Tf True Pair
    by auto
  moreover
  have (s;vs;(vs-to-es ves'')@(vs-to-es ves')@[Block x51 x52])  $\rightsquigarrow$ -i (s;vs;(vs-to-es
ves'')@[Label (length t2s) [] (vs-to-es ves' @ (* x52))])
    using Tf reduce-simple.intros(27) split-n-length[OF Pair True] progress-L0-left[OF
reduce.intros(1)]
      is-const-list-vs-to-es-list[of rev ves] is-const-list-vs-to-es-list[of rev ves'']
    by fastforce
  ultimately
  show ?thesis

```

```

    using Block
    by auto
  qed auto
qed
next
case (Loop x61 x62)
thus ?thesis
proof (cases x61)
case (Tf t1s t2s)
thus ?thesis
  using Loop assms
proof (cases length t1s ≤ length ves; cases split-n ves (length t1s))
case True
case (Pair ves' ves'')
hence vs-to-es ves = vs-to-es ves'' @ vs-to-es ves'
  using split-n-conv-app
  by fastforce
moreover
have s = s' vs = vs' es' = vs-to-es ves'' @ [Label (length t1s) [$Loop x61
x62] (vs-to-es ves' @ ($* x62))]
  using Loop assms Tf True Pair
  by auto
moreover
have (|s;vs;(vs-to-es ves'')@(vs-to-es ves')@[$Loop x61 x62]) ~-i (|s;vs;(vs-to-es
ves'')@[Label (length t1s) [$Loop x61 x62] (vs-to-es ves' @ ($* x62))])
  using Tf reduce-simple.intros(28) split-n-length[OF Pair True] progress-L0-left[OF
reduce.intros(1)]
  is-const-list-vs-to-es-list[of rev ves'] is-const-list-vs-to-es-list[of rev ves'']
  by fastforce
ultimately
show ?thesis
  using Loop
  by auto
qed auto
qed
next
case (If x71 x72 x73)
thus ?thesis
  using assms
proof (cases ves)
case (Cons a list)
thus ?thesis
  using assms If
proof (cases a)
case (ConstInt32 x1)
hence vs-to-es ves = vs-to-es list @ [$C ConstInt32 x1]
  unfolding Cons
  by simp
thus ?thesis

```

```

using progress-L0-left[OF reduce.intros(1)]
      is-const-list-vs-to-es-list[of rev list]
      reduce-simple.intros(29,30)
      assms Cons If ConstInt32
by (cases int-eq x1 0) auto
qed auto
qed auto
next
case (Br x8)
thus ?thesis
      using assms
      by auto
next
case (Br-if x9)
thus ?thesis
      using assms
proof (cases ves)
case (Cons a list)
thus ?thesis
      using assms Br-if
proof (cases a)
case (ConstInt32 x1)
hence vs-to-es ves = vs-to-es list @ [$C ConstInt32 x1]
      unfolding Cons
      by simp
thus ?thesis
      using progress-L0-left[OF reduce.intros(1)[OF reduce-simple.intros(34)]]
      progress-L0-left[OF reduce.intros(1)[OF reduce-simple.intros(35)]]
      is-const-list-vs-to-es-list[of rev list]
      assms Cons Br-if ConstInt32
      by (cases int-eq x1 0) auto
qed auto
qed auto
next
case (Br-table x10)
thus ?thesis
      using assms
proof (cases ves)
case (Cons a list)
thus ?thesis
      using assms Br-table
proof (cases a)
case (ConstInt32 x1)
thus ?thesis
      using progress-L0-left[OF reduce.intros(1)[OF reduce-simple.intros(36)]]
      progress-L0-left[OF reduce.intros(1)[OF reduce-simple.intros(37)]]
      is-const-list-vs-to-es-list[of rev list]
      assms Br-table Cons
      by (cases (nat-of-int x1) < length x10) auto

```

```

    qed auto
  qed auto
next
  case Return
  thus ?thesis
    using assms
    by simp
next
  case (Call x12)
  thus ?thesis
    using assms progress-L0-left[OF reduce.intros(2)]
      is-const-list-vs-to-es-list[of rev ves]
    by auto
next
  case (Call-indirect x13)
  thus ?thesis
    using assms
  proof (cases ves)
    case (Cons a list)
    thus ?thesis
      using assms Call-indirect
    proof (cases a)
      case (ConstInt32 c)
      thus ?thesis
      proof (cases stab s i (nat-of-int c))
        case None
        thus ?thesis
          using assms Call-indirect Cons ConstInt32
            progress-L0-left[OF reduce.intros(4)]
            is-const-list-vs-to-es-list[of rev list]
          by auto
      next
        case (Some cl)
        thus ?thesis
      proof (cases stypes s i x13 = cl-type cl)
        case True
        hence (|s;vs;(vs-to-es list) @ [$C ConstInt32 c, $Call-indirect x13]) ~~-
          i (|s;vs;(vs-to-es list) @ [Callcl cl])
          using progress-L0-left[OF reduce.intros(3)] True Some is-const-list-vs-to-es-list[of
            rev list]
          by fastforce
        thus ?thesis
          using assms Call-indirect Cons ConstInt32 Some True
          by auto
      next
        case False
        hence (|s;vs;(vs-to-es list)@[$C ConstInt32 c, $Call-indirect x13]) ~~- i
          (|s;vs;(vs-to-es list)@[Trap])
          using progress-L0-left[OF reduce.intros(4)] False Some is-const-list-vs-to-es-list[of

```

```

rev list]
  by fastforce
  thus ?thesis
  using assms Call-indirect Cons ConstInt32 Some False
  by auto
  qed
  qed
  qed auto
  qed auto
next
case (Get-local j)
thus ?thesis
  using assms
proof (cases j < length vs)
  case True
  then obtain vs1 v vs2 where vs = vs1@[v]@vs2 length vs1 = j
  using id-take-nth-drop
  by fastforce
  thus ?thesis
  using assms Get-local True
  progress-L0-left[OF reduce.intros(8)]
  is-const-list-vs-to-es-list[of rev ves]
  by auto
  qed auto
next
case (Set-local j)
thus ?thesis
  using assms
proof (cases ves)
  case (Cons a list)
  thus ?thesis
  using assms Set-local
  proof (cases j < length vs)
  case True
  obtain vs1 v vs2 where vs-def:vs = vs1@[v]@vs2 length vs1 = j
  using id-take-nth-drop True
  by fastforce
  thus ?thesis
  using assms Set-local True Cons
  progress-L0-left[OF reduce.intros(9)]
  is-const-list-vs-to-es-list[of rev list]
  by auto
  qed auto
  qed auto
next
case (Tee-local x16)
thus ?thesis
  using assms
proof (cases ves)

```

```

    case (Cons a list)
  thus ?thesis
    using assms Tee-local
      progress-L0-left[OF reduce.intros(1)[OF reduce-simple.intros(41)]]
      is-const-list-vs-to-es-list[of rev list]
    by (auto simp add: is-const-def)
qed auto
next
case (Get-global x17)
thus ?thesis
  using assms
    progress-L0-left[OF reduce.intros(10)]
    is-const-list-vs-to-es-list[of rev ves]
  by (auto simp add: is-const-def)
next
case (Set-global x18)
thus ?thesis
  using assms
proof (cases ves)
  case (Cons a list)
  thus ?thesis
    using assms Set-global
      progress-L0-left[OF reduce.intros(11)]
      is-const-list-vs-to-es-list[of rev list]
    by (auto simp add: is-const-def)
qed auto
next
case (Load x191 x192 x193 x194)
thus ?thesis
  using assms
proof (cases x192; cases ves)
  case None
  case (Cons a list)
  thus ?thesis
    using Load assms None
proof (cases a; cases smem-ind s i)
  case (ConstInt32 x1)
  case (Some a)
  thus ?thesis
    using Load assms None Cons ConstInt32
      progress-L0-left[OF reduce.intros(12)]
      progress-L0-left[OF reduce.intros(13)]
      is-const-list-vs-to-es-list[of rev list]
    by (cases load (s.mem s ! a) (nat-of-int x1) x194 (t-length x191) )
      (auto simp add: is-const-def)
qed auto
next
case outer-some:(Some tp-sx)
case (Cons a list)

```

```

thus ?thesis
  using Load assms outer-some
  proof (cases a; cases smem-ind s i; cases tp-sx)
    case (ConstInt32 x1)
    case (Pair tp sx)
    case (Some a)
  thus ?thesis
    using Load assms outer-some Cons ConstInt32 Pair
      progress-L0-left[OF reduce.intros(14)]
      progress-L0-left[OF reduce.intros(15)]
      is-const-list-vs-to-es-list[of rev list]
    by (cases load-packed sx (s.mem s ! a) (nat-of-int x1) x194 (tp-length tp)
      (t-length x191))
      (auto simp add: is-const-def)
  qed auto
qed auto
next
  case (Store t tp a off)
  thus ?thesis
    using assms
  proof (cases ves)
    case outer-Cons:(Cons a list)
  thus ?thesis
    using Store assms
  proof (cases list)
    case (Cons a' list')
  thus ?thesis
    using Store outer-Cons assms
  proof (cases a')
    case (ConstInt32 x1)
  thus ?thesis
    using Store outer-Cons Cons assms
  proof (cases (types-agree t a); cases smem-ind s i)
    case True
    case outer-Some:(Some j)
  show ?thesis
  proof (cases tp)
    case None
  thus ?thesis
    using Store outer-Cons Cons assms True outer-Some ConstInt32
      progress-L0-left[OF reduce.intros(16)]
      progress-L0-left[OF reduce.intros(17)]
      is-const-list-vs-to-es-list[of rev list']
    by (cases store (s.mem s ! j) (nat-of-int x1) off (bits a) (t-length t))
      auto
  next
  case (Some the-tp)
  thus ?thesis
    using Store outer-Cons Cons assms True outer-Some ConstInt32

```

```

        progress-L0-left[OF reduce.intros(18)]
        progress-L0-left[OF reduce.intros(19)]
        is-const-list-vs-to-es-list[of rev list]
    by (cases store-packed (s.mem s ! j) (nat-of-int x1) off (bits a)
(tp-length the-tp))
    auto
    qed
    qed (cases tp; auto)+
    qed (cases tp; auto)+
    qed (cases tp; auto)
    qed (cases tp; auto)
next
case Current-memory
thus ?thesis
using assms
proof (cases smem-ind s i)
case (Some a)
thus ?thesis
using assms Current-memory
progress-L0-left[OF reduce.intros(20)]
is-const-list-vs-to-es-list[of rev ves]
by (auto simp add: is-const-def)
qed auto
next
case Grow-memory
thus ?thesis
using assms
proof (cases ves)
case (Cons a list)
thus ?thesis
using assms Grow-memory
proof (cases a; cases smem-ind s i)
case (ConstInt32 x1)
case (Some j)
thus ?thesis
using assms Grow-memory Cons ConstInt32
progress-L0-left[OF reduce.intros(21)]
progress-L0-left[OF reduce.intros(22)]
is-const-list-vs-to-es-list[of rev list]
by (cases mem-grow-impl (s.mem s ! j) (nat-of-int x1)) (auto simp add:
mem-grow-impl-correct is-const-def)
qed auto
qed auto
next
case (EConst x23)
thus ?thesis
using assms
by auto
next

```

```

case (Unop-i x241 x242)
thus ?thesis
  using run-step-basic-unop-testop-sound[OF assms]
  by fastforce
next
case (Unop-f x251 x252)
thus ?thesis
  using run-step-basic-unop-testop-sound[OF assms]
  by fastforce
next
case (Binop-i x261 x262)
thus ?thesis
  using run-step-basic-binop-relop-sound[OF assms]
  by fastforce
next
case (Binop-f x271 x272)
thus ?thesis
  using run-step-basic-binop-relop-sound[OF assms]
  by fastforce
next
case (Testop x281 x282)
thus ?thesis
  using run-step-basic-unop-testop-sound[OF assms]
  by fastforce
next
case (Relop-i x291 x292)
thus ?thesis
  using run-step-basic-binop-relop-sound[OF assms]
  by fastforce
next
case (Relop-f x301 x302)
thus ?thesis
  using run-step-basic-binop-relop-sound[OF assms]
  by fastforce
next
case (Cvtop t2 cvtop t1 sx)
thus ?thesis
  using assms
proof (cases ves)
  case (Cons a list)
  thus ?thesis
    using assms Cvtop
  proof (cases cvtop; cases types-agree t1 a)
    case Convert
    case True
    thus ?thesis
      using Convert assms Cvtop Cons
      progress-L0-left[OF reduce.intros(1)[OF reduce-simple.intros(19)]]
      progress-L0-left[OF reduce.intros(1)[OF reduce-simple.intros(20)]]

```

```

      is-const-list-vs-to-es-list[of rev list]
    by (cases (cvt t2 sx a)) auto
  next
  case Reinterpret
  case True
  thus ?thesis
    using Reinterpret assms Cvtop Cons
      progress-L0-left[OF reduce.intros(1)[OF reduce-simple.intros(21)]]
      is-const-list-vs-to-es-list[of rev list]
    by (cases sx) auto
  qed auto
  qed (cases cvtop; auto)
  qed
  qed

theorem run-step-sound:
  assumes run-step d i (s,vs,es) = (s', vs', RSNormal es')
  shows (|s;vs;es|  $\rightsquigarrow$ - i (|s';vs';es'|))
  using assms
proof -
  fix ves e
  have (run-step d i (s,vs,es) = (s', vs', RSNormal es'))  $\implies$ 
    ( $\lambda i (s, vs, es). (|s;vs;es| \rightsquigarrow\text{-} i (|s';vs';es'|)) i (s, vs, es)$ )
  and (run-one-step d i (s,vs,ves,e) = (s', vs', RSNormal es'))  $\implies$ 
    ( $\lambda i (s, vs, ves, e). (|s;vs;(vs\text{-}to\text{-}es\ ves)@[e]| \rightsquigarrow\text{-} i (|s';vs';es'|)) i (s, vs, ves,$ 
e)
  proof (induction d i - and d i - arbitrary: s' vs' es' and s' vs' es' rule:
run-step-run-one-step.induct)
  case (1 d i s vs es)
  obtain ves ses where ves-def:split-vals-e es = (ves, ses)
  by (metis surj-pair)
  thus ?case
  proof (cases ses)
  case Nil
  thus ?thesis
  using 1(2) ves-def
  by simp
  next
  case (Cons a list)
  thus ?thesis
  proof (cases a = Trap)
  case True
  have c-ves:const-list ($$* ves)
  using is-const-list[of - ves]
  by simp
  have es' = [Trap]  $\wedge$  (list  $\neq$  []  $\vee$  ves  $\neq$  [])
  using Cons 1(2) ves-def True
  by (cases (list  $\neq$  []  $\vee$  ves  $\neq$  [])) auto
  thus ?thesis

```

```

    using Cons 1(2) ves-def split-vals-e-conv-app[OF ves-def] True progress-L0-trap[OF
c-ves]
      by auto
    next
      case False
      obtain os ovs oes where ros-def:run-one-step d i (s, vs, (rev ves), a) = (os,
ovs, oes)
        by (metis surj-pair)
      moreover
      then obtain roes where oes = RSNormal roes
        using 1(2) ves-def Cons False
        by (cases oes) auto
      moreover
      hence os = s' ovs = vs' and es'-def:es' = roes @ list
        using 1(2) ves-def Cons ros-def False
        by (cases roes = [Trap], auto simp del: run-one-step.simps)+
      ultimately
      have ros-red:(s;vs;($$* ves) @ [a])  $\rightsquigarrow$ - i (s';vs';roes)
        using 1(1)[OF ves-def[symmetric] - Cons] ros-def False
        by (simp del: run-one-step.simps)
      have (s;vs;($$* ves)@[a]@list)  $\rightsquigarrow$ - i (s';vs';roes@list)
        using progress-L0[OF ros-red, of [] list]
        unfolding const-list-def
        by simp
      thus ?thesis
        using es'-def Cons split-vals-e-conv-app[OF ves-def]
        by simp
    qed
  qed
next
  case (2 d i s vs ves e)
  show ?case
  proof (cases e)
    case (Basic x1)
    thus ?thesis
      using run-step-basic-sound 2(3)
      by simp
  next
    case Trap
    thus ?thesis
      using 2(3)
      by simp
  next
    case (Callcl cl)
    obtain t1s t2s where cl-type cl = (t1s -> t2s)
      using tf.exhaust[of - thesis]
      by fastforce
    moreover
    obtain n where length t1s = n

```

```

    by blast
  moreover
  obtain m where length t2s = m
    by blast
  moreover
  note local-defs = calculation
  show ?thesis
  proof (cases length ves ≥ n)
    case outer-True: True
      obtain ves' ves'' where true-defs: split-n ves n = (ves', ves'')
        by (metis surj-pair)
      have ves'-length: length (rev ves') = n
        using split-n-length[OF true-defs outer-True] inj-basic-econst length-rev
map-injective
      by blast
    show ?thesis
    proof (cases cl)
      case (Func-native i' tf fts fes)
        hence s' = s vs' = vs es' = (vs-to-es ves'' @ [Local (length t2s) i' (rev ves'
@ (n-zeros fts) [$Block ([] -> t2s) fes]])
          using 2(3) Callcl local-defs outer-True true-defs
          unfolding cl-type-def
          by auto
        moreover
        have (|s;vs;(vs-to-es ves')@[Callcl cl]|) ~~-i (|s;vs;([Local (length t2s) i' (rev
ves' @ (n-zeros fts) [$Block ([] -> t2s) fes]])|)
          using reduce.intros(5) local-defs(1,2) Func-native ves'-length
          unfolding cl-type-def
          by fastforce
        ultimately
        show ?thesis
          using Callcl progress-L0-left is-const-list[of - (rev ves'')]
          unfolding split-n-conv-app[OF true-defs(1)]
          by auto
    next
    case (Func-host x21 x22)
      thus ?thesis
      proof (cases host-apply-impl s (t1s -> t2s) x22 (rev ves'))
        case None
          hence s = s'
            vs = vs'
            es' = vs-to-es ves'' @ [Trap]
          using 2(3) Callcl local-defs outer-True true-defs Func-host
          unfolding cl-type-def
          by auto
        thus ?thesis
          using is-const-list[of - (rev ves'')]
          reduce.intros(7)[OF - - ves'-length local-defs(2)]
          split-n-conv-app[OF true-defs]

```

```

      progress-L0-left Callcl Func-host local-defs(1)
    unfolding cl-type-def
    by fastforce
  next
  case (Some a)
  show ?thesis
  proof (cases a)
  case (Pair rs rves)
  thus ?thesis
  using 2(3) Callcl local-defs outer-True true-defs Func-host Some
  unfolding cl-type-def
  proof (cases list-all2 types-agree t2s rves)
  case True
  hence rs = s'
  vs = vs'
  es' = vs-to-es ves'' @ ($$* rves)
  using 2(3) Callcl local-defs outer-True true-defs Func-host Pair
Some
  unfolding cl-type-def
  by auto
  thus ?thesis
  using progress-L0-left reduce.intros(6)[OF - - ves'-length local-defs(2)]
Pair
  Callcl Func-host local-defs(1) True is-const-list[of - (rev ves'')]
  split-n-conv-app[OF true-defs] host-apply-impl-correct[OF Some]
  unfolding cl-type-def
  by fastforce
  qed auto
  qed
  qed
  qed
  next
  case False
  thus ?thesis
  using 2(3) Callcl local-defs
  unfolding cl-type-def
  by (cases cl) auto
  qed
  next
  case (Label ln les es)
  thus ?thesis
  proof (cases es-is-trap es)
  case True
  thus ?thesis
  using 2(3) is-const-list-vs-to-es-list
  Label progress-L0[OF reduce.intros(1)[OF reduce-simple.intros(32)]]
  by fastforce
  next
  case False

```

```

note outer-outer-false = False
show ?thesis
proof (cases (const-list es))
  case True
  thus ?thesis
using 2(3) outer-outer-false Label reduce.intros(1)[OF reduce-simple.intros(31)]
  progress-L0[OF - is-const-list-vs-to-es-list[of rev ves], where ?es-c=[]]
  by fastforce
next
  case False
  obtain s'' vs'' es'' where run-step-is:run-step d i (s, vs, es) = (s'', vs'',
es'')
  by (metis surj-pair)
show ?thesis
proof (cases es'')
  case RSCrash
  thus ?thesis
  using outer-outer-false False run-step-is Label 2(3)
  by auto
next
  case (RSBreak bn bvs)
  thus ?thesis
  proof (cases bn)
  case 0
  have run-step-is-break0:run-step d i (s, vs, es) = (s'', vs'', RSBreak 0
bvs)
  using run-step-is RSBreak 0
  by simp
  hence es'-def:es' = ((vs-to-es ((take ln bvs)@ves))@les) ∧ s' = s'' ∧
vs' = vs'' ∧ ln ≤ length bvs
  using outer-outer-false False run-step-is Label 2(3) RSBreak
  by (cases ln ≤ length bvs) auto
  then obtain n lfilled es-c where local-egs:s=s' vs=vs' ln ≤ length bvs
Lfilled-exact n lfilled ((vs-to-es bvs) @ [$Br n] @ es-c) es
  using run-step-break-imp-lfilled[OF run-step-is-break0] RSBreak es'-def
  by fastforce
  then obtain lfilled' where lfilled-int:Lfilled n lfilled' ((vs-to-es bvs) @
[$Br n]) es
  using lfilled-collapse2[OF Lfilled-exact-imp-Lfilled]
  by fastforce
  obtain lfilled'' where Lfilled n lfilled'' ((drop (length bvs - ln) (vs-to-es
bvs)) @ [$Br n]) es
  using lfilled-collapse1[OF lfilled-int] is-const-list-vs-to-es-list[of rev
bvs] local-egs(3)
  by fastforce
  hence ([$Label ln les es]) ∼ ((drop (length bvs - ln) (vs-to-es bvs))@les)
  using reduce-simple.intros(33) local-egs(3) is-const-list-vs-to-es-list
  unfolding drop-map
  by fastforce

```

```

      hence 1:( $\downarrow s;vs;$ [Label ln les es])  $\rightsquigarrow$ -i ( $\downarrow s';vs';$ (drop (length bus - ln)
(vs-to-es bus))@les)
      using reduce.intros(1) local-eqs(1,2)
      by fastforce
      have ( $\downarrow s;vs;$ (vs-to-es ves)@[e])  $\rightsquigarrow$ -i ( $\downarrow s';vs';$ (vs-to-es ves)@(drop (length
bus - ln) (vs-to-es bus))@les)
      using progress-L0[OF 1 is-const-list-vs-to-es-list[of rev ves], of []]
Label
      by fastforce
      thus ?thesis
      using es'-def
      unfolding drop-map rev-take[symmetric]
      by auto
    next
      case (Suc nat)
      thus ?thesis
      using outer-outer-false False run-step-is Label 2(3) RSBreak
      by auto
    qed
  next
    case (RSReturn x3)
    thus ?thesis
    using outer-outer-false False run-step-is Label 2(3)
    by auto
  next
    case (RSNormal x4)
    hence es' = (vs-to-es ves)@[Label ln les x4] s' = s'' vs' = vs''
    using outer-outer-false False run-step-is Label 2(3) run-step-is
    by auto
    moreover
      have Lfilled 1 (LRec (vs-to-es ves) ln les (LBase [] [] [])) es ((vs-to-es
ves)@[Label ln les es])
      using Lfilled.intros(1)[of [] - [] es]
      Lfilled.intros(2)
      is-const-list-vs-to-es-list[of rev ves]
      unfolding const-list-def
      by fastforce
    moreover
      have Lfilled 1 (LRec (vs-to-es ves) ln les (LBase [] [] [])) x4 ((vs-to-es
ves)@[Label ln les x4])
      using Lfilled.intros(1)[of [] - [] x4]
      Lfilled.intros(2)
      is-const-list-vs-to-es-list[of rev ves]
      unfolding const-list-def
      by fastforce
    moreover
      have inner-reduce:( $\downarrow s;vs;es$ )  $\rightsquigarrow$ -i ( $\downarrow s'';vs'';x4$ )
      using 2(1)[OF Label outer-outer-false False] run-step-is RSNormal
      by auto

```

```

ultimately
show ?thesis
  using Label 2(3) outer-outer-false False run-step-is
    reduce.intros(23)[OF inner-reduce]
  by fastforce
qed
qed
qed
next
case (Local ln j vls es)
thus ?thesis
proof (cases es-is-trap es)
  case True
  thus ?thesis
  using 2(3) is-const-list-vs-to-es-list
    Local progress-L0[OF reduce.intros(1)[OF reduce-simple.intros(39)]]
  by fastforce
next
case False
note outer-outer-false = False
show ?thesis
proof (cases (const-list es))
  case True
  note outer-true = True
  thus ?thesis
  proof (cases length es = ln)
    case True
    thus ?thesis
    using 2(3) Local outer-true outer-outer-false is-const-list-vs-to-es-list[of
rev ves]
      reduce.intros(1)[OF reduce-simple.intros(38)[OF outer-true True]]
      progress-L0[where ?es-c= []]
    by fastforce
  next
  case False
  thus ?thesis
  using 2(3) Local outer-outer-false outer-true is-const-list-vs-to-es-list[of
rev ves]
    by auto
qed
next
case False
show ?thesis
proof (cases d)
  case 0
  thus ?thesis
  using 2(3) Local outer-outer-false False is-const-list-vs-to-es-list[of rev
ves]
    by auto

```

```

next
  case (Suc d')
  obtain s'' vls' les' where run-step-is:run-step d' j (s, vls, es) = (s'', vls',
les')
  by (metis surj-pair)
  show ?thesis
  proof (cases les')
    case RSCrash
    thus ?thesis
    using outer-outer-false False run-step-is Local 2(3) Suc
    by auto
  next
  case (RSBreak x21 x22)
  thus ?thesis
  using outer-outer-false False run-step-is Local 2(3) Suc
  by auto
  next
  case (RSReturn x3)
  hence es'-def:es' = (vs-to-es ((take ln x3)@ves)) ∧ s' = s'' ∧ vs = vs'
  ∧ ln ≤ length x3
  using outer-outer-false False run-step-is Local 2(3) Suc
  by (cases ln ≤ length x3) auto
  then obtain n lfilled es-c where local-eqs:s=s' vs=vs' ln ≤ length x3
  Lfilled-exact n lfilled ((vs-to-es x3) @ [Return] @ es-c) es
  using run-step-is run-step-return-imp-lfilled RSReturn
  by fastforce
  then obtain lfilled' where lfilled-int:Lfilled n lfilled' ((vs-to-es x3) @
[Return]) es
  using lfilled-collapse2[OF Lfilled-exact-imp-Lfilled]
  by fastforce
  obtain lfilled'' where Lfilled n lfilled'' ((drop (length x3 - ln) (vs-to-es
x3)) @ [Return]) es
  using lfilled-collapse1[OF lfilled-int] is-const-list-vs-to-es-list[of rev
x3] local-eqs(3)
  by fastforce
  hence (|[Local ln j vls es]|) ~ (|(drop (length x3 - ln) (vs-to-es x3))|)
  using reduce-simple.intros(40) local-eqs(3) is-const-list-vs-to-es-list
  unfolding drop-map
  by fastforce
  hence 1:(|s;vs;[Local ln j vls es]|) ~-i (|s';vs';(drop (length x3 - ln)
(vs-to-es x3))|)
  using reduce.intros(1) local-eqs(1,2)
  by fastforce
  have (|s;vs;(vs-to-es ves)@[e]|) ~-i (|s';vs';(vs-to-es ves)@(drop (length
x3 - ln) (vs-to-es x3))|)
  using progress-L0[OF 1 is-const-list-vs-to-es-list[of rev ves], of []
Local
  by fastforce
  thus ?thesis

```

```

    using es'-def
    unfolding drop-map rev-take[symmetric]
    by auto
  next
  case (RSNormal x4)
  hence inner-reduce:(s;vls;es)  $\rightsquigarrow$ -j (s'';vls';x4)
    using 2(2)[OF Local outer-outer-false False] run-step-is Suc
    by auto
  thus ?thesis
    using Local 2(3) Local outer-outer-false False run-step-is Suc
      reduce.intros(24)[OF inner-reduce] RSNormal
      progress-L0-left is-const-list-vs-to-es-list[of rev ves]
    by (auto simp del: run-step.simps)
  qed
qed
qed
qed
qed
qed
thus ?thesis
  using assms
  by blast
qed
end

```

References

- [1] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien. Bringing the web up to speed with webassembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 185–200, New York, NY, USA, 2017. ACM.
- [2] C. Watt. Mechanising and verifying the webassembly specification. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2018, pages 53–65, New York, NY, USA, 2018. ACM.