

# 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

<b>1</b>	<b>WebAssembly Core AST</b>	<b>2</b>
<b>2</b>	<b>Syntactic Typeclasses</b>	<b>5</b>
<b>3</b>	<b>WebAssembly Base Definitions</b>	<b>7</b>
<b>4</b>	<b>Host Properties</b>	<b>22</b>
<b>5</b>	<b>Auxiliary Type System Properties</b>	<b>23</b>
<b>6</b>	<b>Lemmas for Soundness Proof</b>	<b>35</b>
	6.1 Preservation . . . . .	35
	6.2 Progress . . . . .	42
<b>7</b>	<b>Soundness Theorems</b>	<b>46</b>
<b>8</b>	<b>Augmented Type Syntax for Concrete Checker</b>	<b>46</b>
<b>9</b>	<b>Executable Type Checker</b>	<b>55</b>
<b>10</b>	<b>Correctness of Type Checker</b>	<b>59</b>
	10.1 Soundness . . . . .	59
	10.2 Completeness . . . . .	60
<b>11</b>	<b>WebAssembly Interpreter</b>	<b>62</b>

## 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 <proof>
  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) <proof>
  lift-definition bytes-replicate :: nat  $\Rightarrow$  byte  $\Rightarrow$  bytes is ( $\lambda n b.$  replicate n (Abs-uint8
    b)) <proof>
  definition msbyte :: bytes  $\Rightarrow$  byte where
    msbyte bs = last (Rep-bytes bs)

  typedef mem = UNIV :: (byte list) set <proof>
  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)) <proof>
  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)) <proof>
  lift-definition mem-append :: mem  $\Rightarrow$  bytes  $\Rightarrow$  mem is append <proof>
  typedecl host
  typedecl host-state

```

**datatype** — value types  
 $t = T-i32 \mid T-i64 \mid T-f32 \mid T-f64$

**datatype** — packed types  
 $tp = Tp-i8 \mid Tp-i16 \mid Tp-i32$

**datatype** — mutability  
 $mut = T-immut \mid T-mut$

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

**datatype** — function types  
 $tf = Tf\ t\ list\ t\ list\ (\leftarrow \ ' \rightarrow \rightarrow 60)$

**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**  
 $sx = S \mid U$

**datatype**  
 $unop-i = Clz \mid Ctz \mid Popcnt$

**datatype**  
 $unop-f = Neg \mid Abs \mid Ceil \mid Floor \mid Trunc \mid Nearest \mid Sqrt$

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

**datatype**

*binop-f* = *Add* | *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-rotr :: '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 ⟨proof⟩ end
instantiation i64 :: wasm-int begin instance ⟨proof⟩ end
instantiation f32 :: wasm-float begin instance ⟨proof⟩ end
instantiation f64 :: wasm-float begin instance ⟨proof⟩ 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-rotate* (*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  $\Rightarrow$  t  $\Rightarrow$  v

*host-apply* :: s  $\Rightarrow$  tf  $\Rightarrow$  host  $\Rightarrow$  v list  $\Rightarrow$  host-state  $\Rightarrow$  (s  $\times$  v list) option

**definition** *typeof* :: v  $\Rightarrow$  t **where**

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

**definition** *option-projl* :: ('a  $\times$  'b) option  $\Rightarrow$  'a option **where**

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

**definition** *option-projr* :: ('a  $\times$  'b) option  $\Rightarrow$  'b option **where**

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

**definition** *t-length* :: t  $\Rightarrow$  nat **where**

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

**definition** *tp-length* :: tp  $\Rightarrow$  nat **where**

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

**definition** *is-int-t* :: t  $\Rightarrow$  bool **where**

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

**definition** *is-float-t* :: t  $\Rightarrow$  bool **where**

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

**definition** *is-mut* :: tg  $\Rightarrow$  bool **where**

*is-mut* tg = (tg-mut tg = *T-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**  $app\text{-}testop\text{-}i :: testop \Rightarrow 'i::wasm\text{-}int \Rightarrow bool$  **where**

$app-testop-i\ testop\ c = (case\ testop\ of\ Eqz \Rightarrow int-eqz\ c)$

**definition**  $app-relop-i :: relop-i \Rightarrow 'i::wasm-int \Rightarrow 'i::wasm-int \Rightarrow bool$  **where**

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

**definition**  $app-relop-f :: relop-f \Rightarrow 'f::wasm-float \Rightarrow 'f::wasm-float \Rightarrow bool$  **where**

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

**definition**  $types-agree :: t \Rightarrow v \Rightarrow bool$  **where**

$types-agree\ t\ v = (typeof\ v = t)$

**definition**  $cl-type :: cl \Rightarrow tf$  **where**

$cl-type\ cl = (case\ cl\ of\ Func-native\ -\ tf\ - \Rightarrow tf\ | Func-host\ tf\ - \Rightarrow tf)$

**definition**  $rglob-is-mut :: global \Rightarrow bool$  **where**

$rglob-is-mut\ g = (g-mut\ g = T-mut)$

**definition**  $stypes :: s \Rightarrow nat \Rightarrow nat \Rightarrow tf$  **where**

$stypes\ s\ i\ j = ((types\ ((inst\ s)!i))!j)$

**definition**  $sfunc-ind :: s \Rightarrow nat \Rightarrow nat \Rightarrow nat$  **where**

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

**definition**  $sfunc :: s \Rightarrow nat \Rightarrow nat \Rightarrow cl$  **where**

$sfunc\ s\ i\ j = (funcs\ s)! (sfunc-ind\ s\ i\ j)$

**definition**  $sglob-ind :: s \Rightarrow nat \Rightarrow nat \Rightarrow nat$  **where**

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

**definition**  $sglob :: s \Rightarrow nat \Rightarrow nat \Rightarrow global$  **where**

$sglob\ s\ i\ j = (globs\ s)! (sglob-ind\ s\ i\ j)$

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

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

**definition**  $smem\text{-}ind :: s \Rightarrow nat \Rightarrow nat\ option\ \mathbf{where}$   
 $smem\text{-}ind\ s\ i = (inst.mem\ ((inst\ s)!i))$

**definition**  $stab\text{-}s :: s \Rightarrow nat \Rightarrow nat \Rightarrow cl\ option\ \mathbf{where}$   
 $stab\text{-}s\ s\ i\ j = (let\ stabinst = ((tab\ s)!i)\ in\ (if\ (length\ (stabinst) > j)\ then\ (stabinst!j)\ else\ None))$

**definition**  $stab :: s \Rightarrow nat \Rightarrow nat \Rightarrow cl\ option\ \mathbf{where}$   
 $stab\ s\ i\ j = (case\ (inst.tab\ ((inst\ s)!i))\ of\ Some\ k => stab\text{-}s\ s\ k\ j\ | None => None)$

**definition**  $supdate\text{-}glob\text{-}s :: s \Rightarrow nat \Rightarrow v \Rightarrow s\ \mathbf{where}$   
 $supdate\text{-}glob\text{-}s\ s\ k\ v = s(globs := (globs\ s)[k := ((globs\ s)!k)(g\text{-}val := v)])$

**definition**  $supdate\text{-}glob :: s \Rightarrow nat \Rightarrow nat \Rightarrow v \Rightarrow s\ \mathbf{where}$   
 $supdate\text{-}glob\ s\ i\ j\ v = (let\ k = sglob\text{-}ind\ s\ i\ j\ in\ supdate\text{-}glob\text{-}s\ s\ k\ v)$

**definition**  $is\text{-}const :: e \Rightarrow bool\ \mathbf{where}$   
 $is\text{-}const\ e = (case\ e\ of\ Basic\ (C\ -) \Rightarrow True\ | - \Rightarrow False)$

**definition**  $const\text{-}list :: e\ list \Rightarrow bool\ \mathbf{where}$   
 $const\text{-}list\ xs = list\text{-}all\ is\text{-}const\ xs$

**inductive**  $store\text{-}extension :: s \Rightarrow s \Rightarrow bool\ \mathbf{where}$   
 $\llbracket insts = insts'; fs = fs'; tclss = tclss'; list\text{-}all2\ (\lambda bs\ bs'. mem\text{-}size\ bs \leq mem\text{-}size\ bs')\ bss\ bss'; gs = gs \rrbracket \Longrightarrow$   
 $store\text{-}extension\ (|s.inst = insts,\ s.funcs = fs,\ s.tab = tclss,\ s.mem = bss,\ s.globs = gs|)$   
 $(|s.inst = insts',\ s.funcs = fs',\ s.tab = tclss',\ s.mem = bss',\ s.globs = gs'|)$

**abbreviation**  $to\text{-}e\text{-}list :: b\text{-}e\ list \Rightarrow e\ list\ (\langle \$* \rightarrow 60) \mathbf{where}$   
 $to\text{-}e\text{-}list\ b\text{-}es \equiv map\ Basic\ b\text{-}es$

**abbreviation**  $v\text{-}to\text{-}e\text{-}list :: v\ list \Rightarrow e\ list\ (\langle \$\$* \rightarrow 60) \mathbf{where}$   
 $v\text{-}to\text{-}e\text{-}list\ ves \equiv map\ (\lambda v. \$C\ v)\ ves$

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

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

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

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

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

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

**definition** *load-store-t-bounds* ::  $a \Rightarrow tp \text{ option} \Rightarrow t \Rightarrow \text{bool}$  **where**

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

**definition** *cvt-i32* ::  $sx \text{ option} \Rightarrow v \Rightarrow i32 \text{ option}$  **where**

$\text{cvt-i32 } sx \ v = (\text{case } v \text{ of}$   
 $\quad \text{ConstInt32 } c \Rightarrow \text{None}$   
 $\quad | \ \text{ConstInt64 } c \Rightarrow \text{Some } (\text{wasm-wrap } c)$   
 $\quad | \ \text{ConstFloat32 } c \Rightarrow (\text{case } sx \text{ of}$   
 $\quad \quad \text{Some } U \Rightarrow \text{ui32-trunc-f32 } c$   
 $\quad \quad | \ \text{Some } S \Rightarrow \text{si32-trunc-f32 } c$   
 $\quad \quad | \ \text{None} \Rightarrow \text{None})$   
 $\quad | \ \text{ConstFloat64 } c \Rightarrow (\text{case } sx \text{ of}$   
 $\quad \quad \text{Some } U \Rightarrow \text{ui32-trunc-f64 } c$   
 $\quad \quad | \ \text{Some } S \Rightarrow \text{si32-trunc-f64 } c$   
 $\quad \quad | \ \text{None} \Rightarrow \text{None}))$

**definition** *cvt-i64* ::  $sx \text{ option} \Rightarrow v \Rightarrow i64 \text{ option}$  **where**

$\text{cvt-i64 } sx \ v = (\text{case } v \text{ of}$   
 $\quad \text{ConstInt32 } c \Rightarrow (\text{case } sx \text{ of}$   
 $\quad \quad \text{Some } U \Rightarrow \text{Some } (\text{wasm-extend-u } c)$   
 $\quad \quad | \ \text{Some } S \Rightarrow \text{Some } (\text{wasm-extend-s } c)$   
 $\quad \quad | \ \text{None} \Rightarrow \text{None})$   
 $\quad | \ \text{ConstInt64 } c \Rightarrow \text{None}$   
 $\quad | \ \text{ConstFloat32 } c \Rightarrow (\text{case } sx \text{ of}$   
 $\quad \quad \text{Some } U \Rightarrow \text{ui64-trunc-f32 } c$   
 $\quad \quad | \ \text{Some } S \Rightarrow \text{si64-trunc-f32 } c$   
 $\quad \quad | \ \text{None} \Rightarrow \text{None})$   
 $\quad | \ \text{ConstFloat64 } c \Rightarrow (\text{case } sx \text{ of}$   
 $\quad \quad \text{Some } U \Rightarrow \text{ui64-trunc-f64 } c$   
 $\quad \quad | \ \text{Some } S \Rightarrow \text{si64-trunc-f64 } c$   
 $\quad \quad | \ \text{None} \Rightarrow \text{None}))$

**definition** *cvt-f32* ::  $sx \text{ option} \Rightarrow v \Rightarrow f32 \text{ option}$  **where**

$\text{cvt-f32 } sx \ v = (\text{case } v \text{ of}$   
 $\quad \text{ConstInt32 } c \Rightarrow (\text{case } sx \text{ of}$   
 $\quad \quad \text{Some } U \Rightarrow \text{Some } (\text{f32-convert-ui32 } c)$   
 $\quad \quad | \ \text{Some } S \Rightarrow \text{Some } (\text{f32-convert-si32 } c)$   
 $\quad \quad | \ - \Rightarrow \text{None})$   
 $\quad | \ \text{ConstInt64 } c \Rightarrow (\text{case } sx \text{ of}$   
 $\quad \quad \text{Some } U \Rightarrow \text{Some } (\text{f32-convert-ui64 } c)$   
 $\quad \quad | \ - \Rightarrow \text{None}))$

$$\begin{array}{l}
| \text{Some } S \Rightarrow \text{Some } (f32\text{-convert-si64 } c) \\
| - \Rightarrow \text{None} \\
| \text{ConstFloat32 } c \Rightarrow \text{None} \\
| \text{ConstFloat64 } c \Rightarrow \text{Some } (\text{wasm-demote } c)
\end{array}$$

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

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

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

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

**definition** *bits* :: *v*  $\Rightarrow$  *bytes* **where**

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

**definition** *bitzero* :: *t*  $\Rightarrow$  *v* **where**

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

**definition** *n-zeros* :: *t list*  $\Rightarrow$  *v 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**  $t = T\text{-i32} \vee t = T\text{-i64}$

*<proof>*

**lemma** *is-float-t-exists:*

**assumes** *is-float-t t*

**shows**  $t = T-f32 \vee t = T-f64$

*<proof>*

**lemma** *int-float-disjoint: is-int-t t = -(is-float-t t)*

*<proof>*

**lemma** *stab-unfold:*

**assumes** *stab s i j = Some cl*

**shows**  $\exists k. \text{inst.tab } ((\text{inst } s)!i) = \text{Some } k \wedge \text{length } ((\text{tab } s)!k) > j \wedge ((\text{tab } s)!k)!j = \text{Some } cl$

*<proof>*

**lemma** *inj-basic: inj Basic*

*<proof>*

**lemma** *inj-basic-econst: inj ( $\lambda v. \$C v$ )*

*<proof>*

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

*<proof>*

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

*<proof>*

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

*<proof>*

**lemma** *v-exists-b-e: $\exists \text{ves. } (\$ \$ * \text{ves}) = (\$ * \text{ves})$*

*<proof>*

**lemma** *Lfilled-exact-imp-Lfilled:*

**assumes** *Lfilled-exact n lholed es LI*

**shows** *Lfilled n lholed es LI*

*<proof>*

**lemma** *Lfilled-exact-app-imp-exists-Lfilled:*

**assumes** *const-list ves*

*Lfilled-exact n lholed (ves@es) LI*

**shows**  $\exists \text{lholed}'. \text{Lfilled } n \text{ lholed}' \text{ es } LI$

*<proof>*

**lemma** *Lfilled-imp-exists-Lfilled-exact:*

**assumes** *Lfilled n lholed es LI*

**shows**  $\exists \text{lholed}' \text{ ves es-c. const-list ves} \wedge \text{Lfilled-exact } n \text{ lholed}' (\text{ves}@es@c) LI$

$\langle \text{proof} \rangle$

**lemma** *n-zeros-typeof*:

$n\text{-zeros } ts = vs \implies (ts = \text{map } \text{typeof } vs)$

$\langle \text{proof} \rangle$

**end**

**theory** *Wasm imports Wasm-Base-Defs begin*

**inductive** *b-e-typing* ::  $[t\text{-context}, b\text{-e list}, tf] \Rightarrow \text{bool}$  ( $\langle \text{-} \vdash \text{-} \rangle \rightarrow 60$ ) **where**

— *num ops*  
 $\text{const}:\mathcal{C} \vdash [C v] : ([\ ] \rightarrow [(\text{typeof } v)])$   
| *unop-i:is-int-t*  $t \implies \mathcal{C} \vdash [Unop\text{-}i\ t \text{-}] : ([t] \rightarrow [t])$   
| *unop-f:is-float-t*  $t \implies \mathcal{C} \vdash [Unop\text{-}f\ t \text{-}] : ([t] \rightarrow [t])$   
| *binop-i:is-int-t*  $t \implies \mathcal{C} \vdash [Binop\text{-}i\ t\ iop] : ([t, t] \rightarrow [t])$   
| *binop-f:is-float-t*  $t \implies \mathcal{C} \vdash [Binop\text{-}f\ t \text{-}] : ([t, t] \rightarrow [t])$   
| *testop:is-int-t*  $t \implies \mathcal{C} \vdash [Testop\ t \text{-}] : ([t] \rightarrow [T\text{-}i32])$   
| *relop-i:is-int-t*  $t \implies \mathcal{C} \vdash [Relop\text{-}i\ t \text{-}] : ([t, t] \rightarrow [T\text{-}i32])$   
| *relop-f:is-float-t*  $t \implies \mathcal{C} \vdash [Relop\text{-}f\ t \text{-}] : ([t, t] \rightarrow [T\text{-}i32])$   
— *convert*  
| *convert*: $[(t1 \neq t2); (sx = \text{None}) = ((\text{is-float-t } t1 \wedge \text{is-float-t } t2) \vee (\text{is-int-t } t1 \wedge \text{is-int-t } t2 \wedge (t\text{-length } t1 < t\text{-length } t2)))] \implies \mathcal{C} \vdash [Cvtop\ t1\ Convert\ t2\ sx] : ([t2] \rightarrow [t1])$   
— *reinterpret*  
| *reinterpret*: $[(t1 \neq t2); t\text{-length } t1 = t\text{-length } t2] \implies \mathcal{C} \vdash [Cvtop\ t1\ Reinterpret\ t2\ \text{None}] : ([t2] \rightarrow [t1])$   
— *unreachable, nop, drop, select*  
| *unreachable*: $\mathcal{C} \vdash [Unreachable] : (ts \rightarrow ts')$   
| *nop*: $\mathcal{C} \vdash [Nop] : ([\ ] \rightarrow [\ ])$   
| *drop*: $\mathcal{C} \vdash [Drop] : ([t] \rightarrow [\ ])$   
| *select*: $\mathcal{C} \vdash [Select] : ([t, t, T\text{-}i32] \rightarrow [t])$   
— *block*  
| *block*: $[(tf = (tn \rightarrow tm)); \mathcal{C}(\text{label} := ([tm] @ (\text{label } \mathcal{C})))] \vdash es : (tn \rightarrow tm)] \implies \mathcal{C} \vdash [Block\ tf\ es] : (tn \rightarrow tm)$   
— *loop*  
| *loop*: $[(tf = (tn \rightarrow tm)); \mathcal{C}(\text{label} := ([tn] @ (\text{label } \mathcal{C})))] \vdash es : (tn \rightarrow tm)] \implies \mathcal{C} \vdash [Loop\ tf\ es] : (tn \rightarrow tm)$   
— *if then else*  
| *if-wasm*: $[(tf = (tn \rightarrow tm)); \mathcal{C}(\text{label} := ([tm] @ (\text{label } \mathcal{C})))] \vdash es1 : (tn \rightarrow tm); \mathcal{C}(\text{label} := ([tm] @ (\text{label } \mathcal{C})))] \vdash es2 : (tn \rightarrow tm)] \implies \mathcal{C} \vdash [If\ tf\ es1\ es2] : (tn @ [T\text{-}i32] \rightarrow tm)$   
— *br*  
| *br*: $[i < \text{length}(\text{label } \mathcal{C}); (\text{label } \mathcal{C})!i = ts] \implies \mathcal{C} \vdash [Br\ i] : (t1s @ ts \rightarrow t2s)$   
— *br-if*  
| *br-if*: $[i < \text{length}(\text{label } \mathcal{C}); (\text{label } \mathcal{C})!i = ts] \implies \mathcal{C} \vdash [Br\text{-}if\ i] : (ts @ [T\text{-}i32] \rightarrow ts)$   
— *br-table*  
| *br-table*: $[(\lambda i. i < \text{length}(\text{label } \mathcal{C}) \wedge (\text{label } \mathcal{C})!i = ts) (is @ [i])] \implies \mathcal{C} \vdash [Br\text{-}table\ is\ i] : (t1s @ ts @ [T\text{-}i32] \rightarrow t2s)$

— *return*  
 $| \text{return} : \llbracket (\text{return } C) = \text{Some } ts \rrbracket \Longrightarrow C \vdash [\text{Return}] : (t1s @ ts \rightarrow t2s)$   
 — *call*  
 $| \text{call} : \llbracket i < \text{length}(\text{func-t } C); (\text{func-t } C)!i = tf \rrbracket \Longrightarrow C \vdash [\text{Call } i] : tf$   
 — *call-indirect*  
 $| \text{call-indirect} : \llbracket i < \text{length}(\text{types-t } C); (\text{types-t } C)!i = (t1s \rightarrow t2s); (\text{table } C) \neq \text{None} \rrbracket$   
 $\Longrightarrow C \vdash [\text{Call-indirect } i] : (t1s @ [T-i32] \rightarrow t2s)$   
 — *get-local*  
 $| \text{get-local} : \llbracket i < \text{length}(\text{local } C); (\text{local } C)!i = t \rrbracket \Longrightarrow C \vdash [\text{Get-local } i] : ([\ ] \rightarrow [t])$   
 — *set-local*  
 $| \text{set-local} : \llbracket i < \text{length}(\text{local } C); (\text{local } C)!i = t \rrbracket \Longrightarrow C \vdash [\text{Set-local } i] : ([t] \rightarrow [\ ])$   
 — *tee-local*  
 $| \text{tee-local} : \llbracket i < \text{length}(\text{local } C); (\text{local } C)!i = t \rrbracket \Longrightarrow C \vdash [\text{Tee-local } i] : ([t] \rightarrow [t])$   
 — *get-global*  
 $| \text{get-global} : \llbracket i < \text{length}(\text{global } C); \text{tg-t } ((\text{global } C)!i) = t \rrbracket \Longrightarrow C \vdash [\text{Get-global } i] : ([\ ] \rightarrow [t])$   
 — *set-global*  
 $| \text{set-global} : \llbracket i < \text{length}(\text{global } C); \text{tg-t } ((\text{global } C)!i) = t; \text{is-mut } ((\text{global } C)!i) \rrbracket \Longrightarrow$   
 $C \vdash [\text{Set-global } i] : ([t] \rightarrow [\ ])$   
 — *load*  
 $| \text{load} : \llbracket (\text{memory } C) = \text{Some } n; \text{load-store-t-bounds } a (\text{option-projl } tp\text{-sx } t) \rrbracket \Longrightarrow C$   
 $\vdash [\text{Load } t \text{ tp-sx } a \text{ off}] : ([T-i32] \rightarrow [t])$   
 — *store*  
 $| \text{store} : \llbracket (\text{memory } C) = \text{Some } n; \text{load-store-t-bounds } a \text{ tp } t \rrbracket \Longrightarrow C \vdash [\text{Store } t \text{ tp } a$   
 $\text{off}] : ([T-i32, t] \rightarrow [\ ])$   
 — *current-memory*  
 $| \text{current-memory} : (\text{memory } C) = \text{Some } n \Longrightarrow C \vdash [\text{Current-memory}] : ([\ ] \rightarrow [T-i32])$   
 — *Grow-memory*  
 $| \text{grow-memory} : (\text{memory } C) = \text{Some } n \Longrightarrow C \vdash [\text{Grow-memory}] : ([T-i32] \rightarrow [T-i32])$   
 — *empty program*  
 $| \text{empty} : C \vdash [\ ] : ([\ ] \rightarrow [\ ])$   
 — *composition*  
 $| \text{composition} : \llbracket C \vdash es : (t1s \rightarrow t2s); C \vdash [e] : (t2s \rightarrow t3s) \rrbracket \Longrightarrow C \vdash es @ [e] : (t1s \rightarrow t3s)$   
 — *weakening*  
 $| \text{weakening} : C \vdash es : (t1s \rightarrow t2s) \Longrightarrow C \vdash es : (ts @ t1s \rightarrow ts @ t2s)$

**inductive** *cl-typing* :: [s-context, cl, tf]  $\Rightarrow$  bool **where**

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

**inductive** *e-typing* :: [s-context, t-context, e list, tf]  $\Rightarrow$  bool ( $\langle \text{---} \vdash - : \rightarrow 60$ )

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

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

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

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

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

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

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

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

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

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

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

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

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

$\llbracket \text{list-all2 } (\text{funci-agree } (s\text{-funcs } \mathcal{S}))\ fs\ tfs; \text{list-all2 } (\text{globi-agree } (s\text{-globs } \mathcal{S}))\ gs\ tgs; (i = \text{Some } i' \wedge i' < \text{length } (s\text{-tab } \mathcal{S}) \wedge (s\text{-tab } \mathcal{S})!i' = (\text{the } n)) \vee (i = \text{None} \wedge n = \text{None}); \text{memi-agree } (s\text{-mem } \mathcal{S})\ j\ m \rrbracket \implies \text{inst-typing } \mathcal{S} (\text{types} = ts, \text{funcs} = fs, \text{tab} = i, \text{mem} = j, \text{globs} = gs) (\text{types-t} = ts, \text{func-t} = tfs, \text{global} = tgs, \text{table} = n, \text{memory} = m, \text{local} = [\ ], \text{label} = [\ ], \text{return} = \text{None})$

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

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

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

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

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

**inductive** *config-typing* :: [nat, s, v list, e list, t list] ⇒ bool (⟨⊢' - -; - : -⟩ 60)  
**where**

[[store-typing s S; S·None ⊢-i vs; es : ts]] ⇒ ⊢-i s; vs; es : ts

**inductive** *reduce-simple* :: [e list, e list] ⇒ bool (⟨⊢-⟩ ∼ ⟨⊢-⟩ 60) **where**

— *integer unary ops*

| unop-i32:([C (ConstInt32 c), (Unop-i T-i32 iop)]) ∼ ([C (ConstInt32 (app-unop-i iop c))])

| unop-i64:([C (ConstInt64 c), (Unop-i T-i64 iop)]) ∼ ([C (ConstInt64 (app-unop-i iop c))])

— *float unary ops*

| unop-f32:([C (ConstFloat32 c), (Unop-f T-f32 fop)]) ∼ ([C (ConstFloat32 (app-unop-f fop c))])

| unop-f64:([C (ConstFloat64 c), (Unop-f T-f64 fop)]) ∼ ([C (ConstFloat64 (app-unop-f fop c))])

— *int32 binary ops*

| binop-i32-Some:[app-binop-i iop c1 c2 = (Some c)] ⇒ ([C (ConstInt32 c1), C (ConstInt32 c2), (Binop-i T-i32 iop)]) ∼ ([C (ConstInt32 c)])

| binop-i32-None:[app-binop-i iop c1 c2 = None] ⇒ ([C (ConstInt32 c1), C (ConstInt32 c2), (Binop-i T-i32 iop)]) ∼ ([Trap])

— *int64 binary ops*

| binop-i64-Some:[app-binop-i iop c1 c2 = (Some c)] ⇒ ([C (ConstInt64 c1), C (ConstInt64 c2), (Binop-i T-i64 iop)]) ∼ ([C (ConstInt64 c)])

| binop-i64-None:[app-binop-i iop c1 c2 = None] ⇒ ([C (ConstInt64 c1), C (ConstInt64 c2), (Binop-i T-i64 iop)]) ∼ ([Trap])

— *float32 binary ops*

| binop-f32-Some:[app-binop-f fop c1 c2 = (Some c)] ⇒ ([C (ConstFloat32 c1), C (ConstFloat32 c2), (Binop-f T-f32 fop)]) ∼ ([C (ConstFloat32 c)])

| binop-f32-None:[app-binop-f fop c1 c2 = None] ⇒ ([C (ConstFloat32 c1), C (ConstFloat32 c2), (Binop-f T-f32 fop)]) ∼ ([Trap])

— *float64 binary ops*

| binop-f64-Some:[app-binop-f fop c1 c2 = (Some c)] ⇒ ([C (ConstFloat64 c1), C (ConstFloat64 c2), (Binop-f T-f64 fop)]) ∼ ([C (ConstFloat64 c)])

| binop-f64-None:[app-binop-f fop c1 c2 = None] ⇒ ([C (ConstFloat64 c1), C (ConstFloat64 c2), (Binop-f T-f64 fop)]) ∼ ([Trap])

— *testops*

| testop-i32:([C (ConstInt32 c), (Testop T-i32 testop)]) ∼ ([C ConstInt32 (wasm-bool (app-testop-i testop c))])

| testop-i64:([C (ConstInt64 c), (Testop T-i64 testop)]) ∼ ([C ConstInt32 (wasm-bool (app-testop-i testop c))])

— *int relops*

| relop-i32:([C (ConstInt32 c1), C (ConstInt32 c2), (Relop-i T-i32 iop)]) ∼ ([C (ConstInt32 (wasm-bool (app-relop-i iop c1 c2))])

| relop-i64:([C (ConstInt64 c1), C (ConstInt64 c2), (Relop-i T-i64 iop)]) ∼ ([C (ConstInt32 (wasm-bool (app-relop-i iop c1 c2))])

— *float relops*

| relop-f32:([C (ConstFloat32 c1), C (ConstFloat32 c2), (Relop-f T-f32 fop)])

$\rightsquigarrow$   $([\$C (ConstInt32 (wasm-bool (app-relop-f fop c1 c2)))])$   
 $|$  *relop-f64*: $([\$C (ConstFloat64 c1), \$C (ConstFloat64 c2), \$(Relop-f T-f64 fop)])$   
 $\rightsquigarrow$   $([\$C (ConstInt32 (wasm-bool (app-relop-f fop c1 c2)))])$   
— *convert*  
 $|$  *convert-Some*: $[\text{types-agree } t1 \ v; \text{cut } t2 \ sx \ v = (Some \ v')] \implies ([\$ (C \ v), \$(Cvtop \ t2 \ Convert \ t1 \ sx)]) \rightsquigarrow ([\$ (C \ v')])$   
 $|$  *convert-None*: $[\text{types-agree } t1 \ v; \text{cut } t2 \ sx \ v = None] \implies ([\$ (C \ v), \$(Cvtop \ t2 \ Convert \ t1 \ sx)]) \rightsquigarrow ([Trap])$   
— *reinterpret*  
 $|$  *reinterpret*: $\text{types-agree } t1 \ v \implies ([\$ (C \ v), \$(Cvtop \ t2 \ Reinterpret \ t1 \ None)]) \rightsquigarrow ([\$ (C (wasm-deserialise (bits \ v) \ t2))])$   
— *unreachable*  
 $|$  *unreachable*: $([\$ Unreachable]) \rightsquigarrow ([Trap])$   
— *nop*  
 $|$  *nop*: $([\$ Nop]) \rightsquigarrow ([\ ])$   
— *drop*  
 $|$  *drop*: $([\$ (C \ v), (\$ Drop)]) \rightsquigarrow ([\ ])$   
— *select*  
 $|$  *select-false*: $\text{int-eq } n \ 0 \implies ([\$ (C \ v1), \$ (C \ v2), \$C (ConstInt32 \ n), (\$ Select)]) \rightsquigarrow ([\$ (C \ v2)])$   
 $|$  *select-true*: $\text{int-ne } n \ 0 \implies ([\$ (C \ v1), \$ (C \ v2), \$C (ConstInt32 \ n), (\$ Select)]) \rightsquigarrow ([\$ (C \ v1)])$   
— *block*  
 $|$  *block*: $[\text{const-list } vs; \text{length } vs = n; \text{length } t1s = n; \text{length } t2s = m] \implies ([vs \ @ \ \$(Block (t1s \ -> \ t2s) \ es)]) \rightsquigarrow ([Label \ m \ [] \ (vs \ @ \ (\$* \ es))])$   
— *loop*  
 $|$  *loop*: $[\text{const-list } vs; \text{length } vs = n; \text{length } t1s = n; \text{length } t2s = m] \implies ([vs \ @ \ \$(Loop (t1s \ -> \ t2s) \ es)]) \rightsquigarrow ([Label \ n \ [\$(Loop (t1s \ -> \ t2s) \ es)] \ (vs \ @ \ (\$* \ es))])$   
— *if*  
 $|$  *if-false*: $\text{int-eq } n \ 0 \implies ([\$C (ConstInt32 \ n), \$(If \ tf \ e1s \ e2s)]) \rightsquigarrow ([\$ (Block \ tf \ e2s)])$   
 $|$  *if-true*: $\text{int-ne } n \ 0 \implies ([\$C (ConstInt32 \ n), \$(If \ tf \ e1s \ e2s)]) \rightsquigarrow ([\$ (Block \ tf \ e1s)])$   
— *label*  
 $|$  *label-const*: $\text{const-list } vs \implies ([Label \ n \ es \ vs]) \rightsquigarrow ([vs])$   
 $|$  *label-trap*: $([Label \ n \ es \ [Trap]]) \rightsquigarrow ([Trap])$   
— *br*  
 $|$  *br*: $[\text{const-list } vs; \text{length } vs = n; \text{Lfilled } i \ \text{hloled } (vs \ @ \ \$(Br \ i)) \ LI] \implies ([Label \ n \ es \ LI]) \rightsquigarrow ([vs \ @ \ es])$   
— *br-if*  
 $|$  *br-if-false*: $\text{int-eq } n \ 0 \implies ([\$C (ConstInt32 \ n), \$(Br-if \ i)]) \rightsquigarrow ([\ ])$   
 $|$  *br-if-true*: $\text{int-ne } n \ 0 \implies ([\$C (ConstInt32 \ n), \$(Br-if \ i)]) \rightsquigarrow ([\$ (Br \ i)])$   
— *br-table*  
 $|$  *br-table*: $[\text{length } is > (\text{nat-of-int } c)] \implies ([\$C (ConstInt32 \ c), \$(Br-table \ is \ i)]) \rightsquigarrow ([\$ (Br (is!(nat-of-int \ c)))])$   
 $|$  *br-table-length*: $[\text{length } is \leq (\text{nat-of-int } c)] \implies ([\$C (ConstInt32 \ c), \$(Br-table \ is \ i)]) \rightsquigarrow ([\$ (Br \ i)])$   
— *local*  
 $|$  *local-const*: $[\text{const-list } es; \text{length } es = n] \implies ([Local \ n \ i \ vs \ es]) \rightsquigarrow ([es])$   
 $|$  *local-trap*: $([Local \ n \ i \ vs \ [Trap]]) \rightsquigarrow ([Trap])$   
— *return*

| *return*: $\llbracket \text{const-list } vs; \text{length } vs = n; \text{Lfilled } j \text{ lholed } (vs \text{ @ } [\text{\$Return}]) \text{ es} \rrbracket \implies$   
 $\llbracket \text{Local } n \text{ i vls es} \rrbracket \rightsquigarrow \llbracket vs \rrbracket$   
— *tee-local*  
| *tee-local:is-const*  $v \implies \llbracket [v, \text{\$(Tee-local } i)] \rrbracket \rightsquigarrow \llbracket [v, v, \text{\$(Set-local } i)] \rrbracket$   
| *trap*: $\llbracket \text{es} \neq [\text{Trap}]; \text{Lfilled } 0 \text{ lholed } [\text{Trap}] \text{ es} \rrbracket \implies \llbracket \text{es} \rrbracket \rightsquigarrow \llbracket [\text{Trap}] \rrbracket$

**inductive** *reduce* ::  $[s, v \text{ list}, e \text{ list}, \text{nat}, s, v \text{ list}, e \text{ list}] \Rightarrow \text{bool}$  ( $\llbracket -; - \rrbracket \rightsquigarrow' - -$   
 $\llbracket -; - \rrbracket \gg 60$ ) **where**  
— *lifting basic reduction*  
*basic*: $\llbracket e \rrbracket \rightsquigarrow \llbracket e' \rrbracket \implies \llbracket s; vs; e \rrbracket \rightsquigarrow\text{-i } \llbracket s; vs; e' \rrbracket$   
— *call*  
| *call*: $\llbracket s; vs; [\text{\$(Call } j)] \rrbracket \rightsquigarrow\text{-i } \llbracket s; vs; [\text{Callcl } (\text{sfunc } s \text{ i } j)] \rrbracket$   
— *call-indirect*  
| *call-indirect-Some*: $\llbracket \text{stab } s \text{ i } (\text{nat-of-int } c) = \text{Some } cl; \text{stypes } s \text{ i } j = \text{tf}; \text{cl-type } cl = \text{tf} \rrbracket \implies \llbracket s; vs; [\text{\$(C } (\text{ConstInt32 } c), \text{\$(Call-indirect } j))] \rrbracket \rightsquigarrow\text{-i } \llbracket s; vs; [\text{Callcl } cl] \rrbracket$   
| *call-indirect-None*: $\llbracket (\text{stab } s \text{ i } (\text{nat-of-int } c) = \text{Some } cl \wedge \text{stypes } s \text{ i } j \neq \text{cl-type } cl) \vee \text{stab } s \text{ i } (\text{nat-of-int } c) = \text{None} \rrbracket \implies \llbracket s; vs; [\text{\$(C } (\text{ConstInt32 } c), \text{\$(Call-indirect } j))] \rrbracket \rightsquigarrow\text{-i } \llbracket s; vs; [\text{Trap}] \rrbracket$   
— *call*  
| *callcl-native*: $\llbracket cl = \text{Func-native } j \text{ (} t1s \text{ -> } t2s) \text{ ts es; ves} = (\text{\$(\$* } vcs); \text{length } vcs = n; \text{length } ts = k; \text{length } t1s = n; \text{length } t2s = m; (\text{n-zeros } ts = zs) \rrbracket \implies \llbracket s; vs; ves \text{ @ } [\text{Callcl } cl] \rrbracket \rightsquigarrow\text{-i } \llbracket s; vs; [\text{Local } m \text{ j } (vcs \text{ @ } zs) \text{ [\$(Block } (\text{[] -> } t2s) \text{ es} \rrbracket)] \rrbracket$   
| *callcl-host-Some*: $\llbracket cl = \text{Func-host } (t1s \text{ -> } t2s) \text{ f; ves} = (\text{\$(\$* } vcs); \text{length } vcs = n; \text{length } t1s = n; \text{length } t2s = m; \text{host-apply } s \text{ (} t1s \text{ -> } t2s) \text{ f } vcs \text{ hs} = \text{Some } (s', vcs') \rrbracket \implies \llbracket s; vs; ves \text{ @ } [\text{Callcl } cl] \rrbracket \rightsquigarrow\text{-i } \llbracket s'; vs; (\text{\$(\$* } vcs') \rrbracket$   
| *callcl-host-None*: $\llbracket cl = \text{Func-host } (t1s \text{ -> } t2s) \text{ f; ves} = (\text{\$(\$* } vcs); \text{length } vcs = n; \text{length } t1s = n; \text{length } t2s = m \rrbracket \implies \llbracket s; vs; ves \text{ @ } [\text{Callcl } cl] \rrbracket \rightsquigarrow\text{-i } \llbracket s; vs; [\text{Trap}] \rrbracket$   
— *get-local*  
| *get-local*: $\llbracket \text{length } vi = j \rrbracket \implies \llbracket s; (vi \text{ @ } [v] \text{ @ } vs); [\text{\$(Get-local } j)] \rrbracket \rightsquigarrow\text{-i } \llbracket s; (vi \text{ @ } [v] \text{ @ } vs); [\text{\$(C } v)] \rrbracket$   
— *set-local*  
| *set-local*: $\llbracket \text{length } vi = j \rrbracket \implies \llbracket s; (vi \text{ @ } [v] \text{ @ } vs); [\text{\$(C } v'), \text{\$(Set-local } j)] \rrbracket \rightsquigarrow\text{-i } \llbracket s; (vi \text{ @ } [v'] \text{ @ } vs); [] \rrbracket$   
— *get-global*  
| *get-global*: $\llbracket s; vs; [\text{\$(Get-global } j)] \rrbracket \rightsquigarrow\text{-i } \llbracket s; vs; [\text{\$(C } (\text{sglob-val } s \text{ i } j))] \rrbracket$   
— *set-global*  
| *set-global:supdate-glob*  $s \text{ i } j \text{ v} = s' \implies \llbracket s; vs; [\text{\$(C } v), \text{\$(Set-global } j)] \rrbracket \rightsquigarrow\text{-i } \llbracket s'; vs; [] \rrbracket$   
— *load*  
| *load-Some*: $\llbracket \text{smem-ind } s \text{ i} = \text{Some } j; ((\text{mem } s)!j) = m; \text{load } m \text{ (nat-of-int } k) \text{ off } (t\text{-length } t) = \text{Some } bs \rrbracket \implies \llbracket s; vs; [\text{\$(C } (\text{ConstInt32 } k), \text{\$(Load } t \text{ None } a \text{ off} \rrbracket)] \rightsquigarrow\text{-i } \llbracket s; vs; [\text{\$(C } (\text{wasm-deserialise } bs \text{ t} \rrbracket)] \rrbracket$   
| *load-None*: $\llbracket \text{smem-ind } s \text{ i} = \text{Some } j; ((\text{mem } s)!j) = m; \text{load } m \text{ (nat-of-int } k) \text{ off } (t\text{-length } t) = \text{None} \rrbracket \implies \llbracket s; vs; [\text{\$(C } (\text{ConstInt32 } k), \text{\$(Load } t \text{ None } a \text{ off} \rrbracket)] \rightsquigarrow\text{-i } \llbracket s; vs; [\text{Trap}] \rrbracket$   
— *load packed*  
| *load-packed-Some*: $\llbracket \text{smem-ind } s \text{ i} = \text{Some } j; ((\text{mem } s)!j) = m; \text{load-packed } sx \text{ m } (\text{nat-of-int } k) \text{ off } (tp\text{-length } tp) \text{ (} t\text{-length } t) = \text{Some } bs \rrbracket \implies \llbracket s; vs; [\text{\$(C } (\text{ConstInt32 } k), \text{\$(Load } t \text{ (Some } (tp, sx) \text{ a } \text{ off} \rrbracket)] \rightsquigarrow\text{-i } \llbracket s; vs; [\text{\$(C } (\text{wasm-deserialise } bs \text{ t} \rrbracket)] \rrbracket$

| *load-packed-None*: $\llbracket \text{smem-ind } s \ i = \text{Some } j; ((\text{mem } s)!j) = m; \text{load-packed } sx \ m$   
 $(\text{nat-of-int } k) \ \text{off } (tp\text{-length } tp) \ (t\text{-length } t) = \text{None} \rrbracket \Longrightarrow \langle s; vs; [\$C \ (\text{ConstInt32 } k),$   
 $\$(\text{Load } t \ (\text{Some } (tp, \ sx)) \ a \ \text{off})] \rangle \rightsquigarrow\text{-}i \langle s; vs; [\text{Trap}] \rangle$   
— *store*  
| *store-Some*: $\llbracket \text{types-agree } t \ v; \text{smem-ind } s \ i = \text{Some } j; ((\text{mem } s)!j) = m; \text{store } m$   
 $(\text{nat-of-int } k) \ \text{off } (\text{bits } v) \ (t\text{-length } t) = \text{Some } mem \rrbracket \Longrightarrow \langle s; vs; [\$C \ (\text{ConstInt32 } k),$   
 $\$C \ v, \ \$(\text{Store } t \ \text{None } a \ \text{off})] \rangle \rightsquigarrow\text{-}i \langle s \langle mem := ((\text{mem } s)[j] := mem) \rangle; vs; [] \rangle$   
| *store-None*: $\llbracket \text{types-agree } t \ v; \text{smem-ind } s \ i = \text{Some } j; ((\text{mem } s)!j) = m; \text{store } m$   
 $(\text{nat-of-int } k) \ \text{off } (\text{bits } v) \ (t\text{-length } t) = \text{None} \rrbracket \Longrightarrow \langle s; vs; [\$C \ (\text{ConstInt32 } k), \ \$C$   
 $v, \ \$(\text{Store } t \ \text{None } a \ \text{off})] \rangle \rightsquigarrow\text{-}i \langle s; vs; [\text{Trap}] \rangle$   
— *store packed*  
| *store-packed-Some*: $\llbracket \text{types-agree } t \ v; \text{smem-ind } s \ i = \text{Some } j; ((\text{mem } s)!j) = m;$   
 $\text{store-packed } m \ (\text{nat-of-int } k) \ \text{off } (\text{bits } v) \ (tp\text{-length } tp) = \text{Some } mem \rrbracket \Longrightarrow \langle s; vs; [\$C$   
 $(\text{ConstInt32 } k), \ \$C \ v, \ \$(\text{Store } t \ (\text{Some } tp) \ a \ \text{off})] \rangle \rightsquigarrow\text{-}i \langle s \langle mem := ((\text{mem } s)[j] :=$   
 $mem) \rangle \rangle; vs; [] \rangle$   
| *store-packed-None*: $\llbracket \text{types-agree } t \ v; \text{smem-ind } s \ i = \text{Some } j; ((\text{mem } s)!j) = m;$   
 $\text{store-packed } m \ (\text{nat-of-int } k) \ \text{off } (\text{bits } v) \ (tp\text{-length } tp) = \text{None} \rrbracket \Longrightarrow \langle s; vs; [\$C$   
 $(\text{ConstInt32 } k), \ \$C \ v, \ \$(\text{Store } t \ (\text{Some } tp) \ a \ \text{off})] \rangle \rightsquigarrow\text{-}i \langle s; vs; [\text{Trap}] \rangle$   
— *current-memory*  
| *current-memory*: $\llbracket \text{smem-ind } s \ i = \text{Some } j; ((\text{mem } s)!j) = m; \text{mem-size } m = n \rrbracket$   
 $\Longrightarrow \langle s; vs; [\$(\text{Current-memory})] \rangle \rightsquigarrow\text{-}i \langle s; vs; [\$C \ (\text{ConstInt32 } (\text{int-of-nat } n))] \rangle$   
— *grow-memory*  
| *grow-memory*: $\llbracket \text{smem-ind } s \ i = \text{Some } j; ((\text{mem } s)!j) = m; \text{mem-size } m = n;$   
 $\text{mem-grow } m \ (\text{nat-of-int } c) = mem \rrbracket \Longrightarrow \langle s; vs; [\$C \ (\text{ConstInt32 } c), \ \$(\text{Grow-memory})] \rangle$   
 $\rightsquigarrow\text{-}i \langle s \langle mem := ((\text{mem } s)[j] := mem) \rangle \rangle; vs; [\$C \ (\text{ConstInt32 } (\text{int-of-nat } n))] \rangle$   
— *grow-memory fail*  
| *grow-memory-fail*: $\llbracket \text{smem-ind } s \ i = \text{Some } j; ((\text{mem } s)!j) = m; \text{mem-size } m =$   
 $n \rrbracket \Longrightarrow \langle s; vs; [\$C \ (\text{ConstInt32 } c), \ \$(\text{Grow-memory})] \rangle \rightsquigarrow\text{-}i \langle s; vs; [\$C \ (\text{ConstInt32}$   
 $\text{int32-minus-one})] \rangle$   
  
— *inductive label reduction*  
| *label*: $\llbracket \langle s; vs; es \rangle \rightsquigarrow\text{-}i \langle s'; vs'; es' \rangle; L\text{filled } k \ \text{lholed } es \ \text{les}; L\text{filled } k \ \text{lholed } es' \ \text{les}' \rrbracket \Longrightarrow$   
 $\langle s; vs; les \rangle \rightsquigarrow\text{-}i \langle s'; vs'; les' \rangle$   
— *inductive local reduction*  
| *local*: $\llbracket \langle s; vs; es \rangle \rightsquigarrow\text{-}i \langle s'; vs'; es' \rangle \rrbracket \Longrightarrow \langle s; v0s; [\text{Local } n \ i \ vs \ es] \rangle \rightsquigarrow\text{-}j \langle s'; v0s; [\text{Local } n$   
 $i \ vs' \ es'] \rangle$

end

## 4 Host Properties

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

**lemma** *mem-grow-size*:

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

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

*(proof)*

**lemma** *load-size*:

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

**lemma** *load-packed-size*:

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

**lemma** *store-size1*:

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

**lemma** *store-size*:

**assumes**  $(\text{store } m \ n \ \text{off } v \ l = \text{Some } m')$   
**shows**  $\text{mem-size } m = \text{mem-size } m'$   
 $\langle \text{proof} \rangle$

**lemma** *store-packed-size1*:

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

**lemma** *store-packed-size*:

**assumes**  $(\text{store-packed } m \ n \ \text{off } v \ l = \text{Some } m')$   
**shows**  $\text{mem-size } m = \text{mem-size } m'$   
 $\langle \text{proof} \rangle$

**axiomatization** *where*

$\text{wasm-deserialise-type:typeof } (\text{wasm-deserialise } bs \ t) = t$

**axiomatization** *where*

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

## 5 Auxiliary Type System Properties

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

**lemma** *typeof-i32*:

**assumes**  $\text{typeof } v = T\text{-i32}$   
**shows**  $\exists c. v = \text{ConstInt32 } c$   
 $\langle \text{proof} \rangle$

**lemma** *typeof-i64*:

**assumes**  $\text{typeof } v = T\text{-i64}$   
**shows**  $\exists c. v = \text{ConstInt64 } c$   
 $\langle \text{proof} \rangle$

**lemma** *typeof-f32*:

**assumes** *typeof*  $v = T\text{-}f32$   
**shows**  $\exists c. v = \text{ConstFloat32 } c$   
*<proof>*

**lemma** *typeof-f64*:

**assumes** *typeof*  $v = T\text{-}f64$   
**shows**  $\exists c. v = \text{ConstFloat64 } c$   
*<proof>*

**lemma** *exists-v-typeof*:  $\exists v v. \text{typeof } v = t$

*<proof>*

**lemma** *lfilled-collapse1*:

**assumes** *Lfilled*  $n$  *lholed*  $(vs@es)$  *LI*  
*const-list*  $vs$   
 $\text{length } vs \geq l$   
**shows**  $\exists \text{lholed}' . \text{Lfilled } n \text{lholed}' ((\text{drop } (\text{length } vs - l) \text{ } vs)@es)$  *LI*  
*<proof>*

**lemma** *lfilled-collapse2*:

**assumes** *Lfilled*  $n$  *lholed*  $(es@es')$  *LI*  
**shows**  $\exists \text{lholed}' \text{ } vs' . \text{Lfilled } n \text{lholed}' \text{ } es$  *LI*  
*<proof>*

**lemma** *lfilled-collapse3*:

**assumes** *Lfilled*  $k$  *lholed*  $[Label \ n \ les \ es]$  *LI*  
**shows**  $\exists \text{lholed}' . \text{Lfilled } (\text{Suc } k) \text{lholed}' \text{ } es$  *LI*  
*<proof>*

**lemma** *unlift-b-e*: **assumes**  $\mathcal{S} \cdot \mathcal{C} \vdash \$*b\text{-}es : \text{tf}$  **shows**  $\mathcal{C} \vdash b\text{-}es : \text{tf}$

*<proof>*

**lemma** *store-typing-imp-inst-length-eq*:

**assumes** *store-typing*  $s \ \mathcal{S}$   
**shows**  $\text{length } (\text{inst } s) = \text{length } (s\text{-inst } \mathcal{S})$   
*<proof>*

**lemma** *store-typing-imp-func-length-eq*:

**assumes** *store-typing*  $s \ \mathcal{S}$   
**shows**  $\text{length } (\text{funcs } s) = \text{length } (s\text{-funcs } \mathcal{S})$   
*<proof>*

**lemma** *store-typing-imp-mem-length-eq*:

**assumes** *store-typing*  $s \ \mathcal{S}$   
**shows**  $\text{length } (s\text{-mem } s) = \text{length } (s\text{-mem } \mathcal{S})$   
*<proof>*

**lemma** *store-typing-imp-glob-length-eq*:

**assumes** *store-typing*  $s \mathcal{S}$   
**shows**  $\text{length } (\text{globs } s) = \text{length } (s\text{-globs } \mathcal{S})$   
*<proof>*

**lemma** *store-typing-imp-inst-typing*:

**assumes** *store-typing*  $s \mathcal{S}$   
 $i < \text{length } (\text{inst } s)$   
**shows** *inst-typing*  $\mathcal{S} ((\text{inst } s)!i) ((s\text{-inst } \mathcal{S})!i)$   
*<proof>*

**lemma** *stab-typed-some-imp-member*:

**assumes** *stab*  $s i c = \text{Some } cl$   
*store-typing*  $s \mathcal{S}$   
 $i < \text{length } (\text{inst } s)$   
**shows**  $\text{Some } cl \in \text{set } (\text{concat } (s.\text{tab } s))$   
*<proof>*

**lemma** *stab-typed-some-imp-cl-typed*:

**assumes** *stab*  $s i c = \text{Some } cl$   
*store-typing*  $s \mathcal{S}$   
 $i < \text{length } (\text{inst } s)$   
**shows**  $\exists tf. \text{cl-typing } \mathcal{S} cl tf$   
*<proof>*

**lemma** *b-e-type-empty1*[*dest*]: **assumes**  $\mathcal{C} \vdash [] : (ts \rightarrow ts')$  **shows**  $ts = ts'$   
*<proof>*

**lemma** *b-e-type-empty*:  $(\mathcal{C} \vdash [] : (ts \rightarrow ts')) = (ts = ts')$   
*<proof>*

**lemma** *b-e-type-value*:

**assumes**  $\mathcal{C} \vdash [e] : (ts \rightarrow ts')$   
 $e = \mathcal{C} v$   
**shows**  $ts' = ts @ [\text{typeof } v]$   
*<proof>*

**lemma** *b-e-type-load*:

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

**lemma** *b-e-type-store*:

**assumes**  $\mathcal{C} \vdash [e] : (ts \rightarrow ts')$   
 $e = \text{Store } t \text{ tp } a \text{ off}$   
**shows**  $ts = ts''@[T\text{-i32}, t]$

$\exists \text{sec } n. (\text{memory } \mathcal{C}) = \text{Some } n$   
 $\text{load-store-t-bounds a tp } t$   
 $\langle \text{proof} \rangle$

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

**assumes**  $\mathcal{C} \vdash [e] : (ts \rightarrow ts')$   
 $e = \text{Current-memory}$

**shows**  $\exists \text{sec } n. ts' = ts @ [T-i32] \wedge (\text{memory } \mathcal{C}) = \text{Some } n$   
 $\langle \text{proof} \rangle$

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

**assumes**  $\mathcal{C} \vdash [e] : (ts \rightarrow ts')$   
 $e = \text{Grow-memory}$

**shows**  $\exists ts''. ts = ts'' @ [T-i32] \wedge ts = ts' \wedge (\exists n. (\text{memory } \mathcal{C}) = \text{Some } n)$   
 $\langle \text{proof} \rangle$

**lemma** *b-e-type-nop*:

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

**shows**  $ts = ts'$   
 $\langle \text{proof} \rangle$

**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**  $\exists ts''. ts = ts'' @ [t, t] \wedge ts' = ts'' @ [\text{arity-2-result}(e)]$

$e = \text{Binop-f } t \text{ fop} \implies \text{is-float-t } t$

$e = \text{Relop-f } t \text{ frop} \implies \text{is-float-t } t$

$\langle \text{proof} \rangle$

**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**  $ts \neq []$

$\langle \text{proof} \rangle$

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

*arity-1-result op1* = (case op1 of  
 $\text{Unop-i } t \Rightarrow t$   
 $|\ \text{Unop-f } t \Rightarrow t$   
 $|\ \text{Testop } t \Rightarrow T-i32$

| *Cvtop t1 Convert - -*  $\Rightarrow$  *t1*  
 | *Cvtop t1 Reinterpret - -*  $\Rightarrow$  *t1*)

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

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

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

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

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

*<proof>*

**lemma** *b-e-type-cvtop*:

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

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

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

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

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

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

*<proof>*

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

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

$e = \text{Drop}$

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

*<proof>*

**lemma** *b-e-type-select*:

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

$e = \text{Select}$

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

*<proof>*

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

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

$e = \text{Call}\ i$

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

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

*<proof>*

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

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

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

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

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

$\rightarrow tf2)$

*<proof>*

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

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

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

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

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

**assumes**  $\mathcal{C} \vdash [e] : (ts \rightarrow ts')$   
 $e = \text{Set-local } i$

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

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

**assumes**  $\mathcal{C} \vdash [e] : (ts \rightarrow ts')$   
 $e = \text{Tee-local } i$

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

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

**assumes**  $\mathcal{C} \vdash [e] : (ts \rightarrow ts')$   
 $e = \text{Get-global } i$

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

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

**assumes**  $\mathcal{C} \vdash [e] : (ts \rightarrow ts')$   
 $e = \text{Set-global } i$

**shows**  $\exists t. ts = ts'@[t] \wedge (\text{global } \mathcal{C})!i = (\text{tg-mut} = T\text{-mut}, \text{tg-t} = t) \wedge i < \text{length}(\text{global } \mathcal{C})$   
 $\langle \text{proof} \rangle$

**lemma** *b-e-type-block*:

**assumes**  $\mathcal{C} \vdash [e] : (ts \rightarrow ts')$   
 $e = \text{Block } tf \ es$

**shows**  $\exists ts'' \text{tfn } \text{tfm}. tf = (\text{tfn} \rightarrow \text{tfm}) \wedge (ts = ts''@\text{tfn}) \wedge (ts' = ts''@\text{tfm}) \wedge$   
 $(\mathcal{C}(\text{label} := [\text{tfn}] @ \text{label } \mathcal{C}) \vdash es : tf)$   
 $\langle \text{proof} \rangle$

**lemma** *b-e-type-loop*:

**assumes**  $\mathcal{C} \vdash [e] : (ts \rightarrow ts')$   
 $e = \text{Loop } tf \ es$

**shows**  $\exists ts'' \text{tfn } \text{tfm}. tf = (\text{tfn} \rightarrow \text{tfm}) \wedge (ts = ts''@\text{tfn}) \wedge (ts' = ts''@\text{tfm}) \wedge$   
 $(\mathcal{C}(\text{label} := [\text{tfn}] @ \text{label } \mathcal{C}) \vdash es : tf)$   
 $\langle \text{proof} \rangle$

**lemma** *b-e-type-if*:

**assumes**  $\mathcal{C} \vdash [e] : (ts \rightarrow ts')$   
 $e = \text{If } tf \ es1 \ es2$

**shows**  $\exists ts'' \text{tfn } \text{tfm}. tf = (\text{tfn} \rightarrow \text{tfm}) \wedge (ts = ts''@\text{tfn} @ [T-i32]) \wedge (ts' = ts''@\text{tfm}) \wedge$

$(\mathcal{C}(\text{label} := [\text{tfn}] @ \text{label } \mathcal{C}) \vdash es1 : tf) \wedge$   
 $(\mathcal{C}(\text{label} := [\text{tfn}] @ \text{label } \mathcal{C}) \vdash es2 : tf)$

$\langle proof \rangle$

**lemma** *b-e-type-br*:

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

$e = Br\ i$

**shows**  $i < length(label\ \mathcal{C})$

$\exists ts-c\ ts''.\ ts = ts-c @ ts'' \wedge (label\ \mathcal{C})!i = ts''$

$\langle proof \rangle$

**lemma** *b-e-type-br-if*:

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

$e = Br-if\ i$

**shows**  $i < length(label\ \mathcal{C})$

$\exists ts-c\ ts''.\ ts = ts-c @ ts'' @ [T-i32] \wedge ts' = ts-c @ ts'' \wedge (label\ \mathcal{C})!i = ts''$

$\langle proof \rangle$

**lemma** *b-e-type-br-table*:

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

$e = Br-table\ is\ i$

**shows**  $\exists ts-c\ ts''.\ list-all\ (\lambda i.\ i < length(label\ \mathcal{C}) \wedge (label\ \mathcal{C})!i = ts'')\ (is@[i]) \wedge ts = ts-c @ ts''@[T-i32]$

$\langle proof \rangle$

**lemma** *b-e-type-return*:

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

$e = Return$

**shows**  $\exists ts-c\ ts''.\ ts = ts-c @ ts'' \wedge (return\ \mathcal{C}) = Some\ ts''$

$\langle proof \rangle$

**lemma** *b-e-type-comp*:

**assumes**  $\mathcal{C} \vdash es@[e] : (t1s \rightarrow t4s)$

**shows**  $\exists ts'.\ (\mathcal{C} \vdash es : (t1s \rightarrow ts')) \wedge (\mathcal{C} \vdash [e] : (ts' \rightarrow t4s))$

$\langle proof \rangle$

**lemma** *b-e-type-comp2-unlift*:

**assumes**  $\mathcal{S}\cdot\mathcal{C} \vdash [\$e1, \$e2] : (t1s \rightarrow t2s)$

**shows**  $\exists ts'.\ (\mathcal{C} \vdash [e1] : (t1s \rightarrow ts')) \wedge (\mathcal{C} \vdash [e2] : (ts' \rightarrow t2s))$

$\langle proof \rangle$

**lemma** *b-e-type-comp2-relift*:

**assumes**  $\mathcal{C} \vdash [e1] : (t1s \rightarrow ts')\ \mathcal{C} \vdash [e2] : (ts' \rightarrow t2s)$

**shows**  $\mathcal{S}\cdot\mathcal{C} \vdash [\$e1, \$e2] : (ts@t1s \rightarrow ts@t2s)$

$\langle proof \rangle$

**lemma** *b-e-type-value2*:

**assumes**  $\mathcal{C} \vdash [C\ v1, C\ v2] : (t1s \rightarrow t2s)$

**shows**  $t2s = t1s @ [typeof\ v1, typeof\ v2]$

$\langle proof \rangle$

**lemma** *e-type-comp*:

**assumes**  $\mathcal{S}\cdot\mathcal{C} \vdash es@[e] : (t1s \rightarrow t3s)$

**shows**  $\exists ts'. (\mathcal{S}\cdot\mathcal{C} \vdash es : (t1s \rightarrow ts')) \wedge (\mathcal{S}\cdot\mathcal{C} \vdash [e] : (ts' \rightarrow t3s))$

$\langle proof \rangle$

**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**  $\mathcal{S}\cdot\mathcal{C} \vdash es@es' : (t1s \rightarrow t3s)$

$\langle proof \rangle$

**lemma** *b-e-type-comp-conc*:

**assumes**  $\mathcal{C} \vdash es : (t1s \rightarrow t2s)$

$\mathcal{C} \vdash es' : (t2s \rightarrow t3s)$

**shows**  $\mathcal{C} \vdash es@es' : (t1s \rightarrow t3s)$

$\langle proof \rangle$

**lemma** *e-type-comp-conc1*:

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

**shows**  $\exists ts''. (\mathcal{S}\cdot\mathcal{C} \vdash es : (ts \rightarrow ts'')) \wedge (\mathcal{S}\cdot\mathcal{C} \vdash es' : (ts'' \rightarrow ts'))$

$\langle proof \rangle$

**lemma** *e-type-comp-conc2*:

**assumes**  $\mathcal{S}\cdot\mathcal{C} \vdash es@es'@es'' : (t1s \rightarrow t2s)$

**shows**  $\exists ts' ts''. (\mathcal{S}\cdot\mathcal{C} \vdash es : (t1s \rightarrow ts'))$

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

$\wedge (\mathcal{S}\cdot\mathcal{C} \vdash es'' : (ts'' \rightarrow t2s))$

$\langle proof \rangle$

**lemma** *b-e-type-value-list*:

**assumes**  $(\mathcal{C} \vdash es@[C v] : (ts \rightarrow ts'@[t]))$

**shows**  $(\mathcal{C} \vdash es : (ts \rightarrow ts'))$

$(\text{typeof } v = t)$

$\langle proof \rangle$

**lemma** *e-type-label*:

**assumes**  $\mathcal{S}\cdot\mathcal{C} \vdash [Label\ n\ es0\ es] : (ts \rightarrow ts')$

**shows**  $\exists t1s\ t2s. (ts' = (t1s@t2s))$

$\wedge \text{length } t1s = n$

$\wedge (\mathcal{S}\cdot\mathcal{C} \vdash es0 : (t1s \rightarrow t2s))$

$\wedge (\mathcal{S}\cdot\mathcal{C}(\text{label} := [t1s] @ (\text{label } \mathcal{C})) \vdash es : ([ ] \rightarrow t2s))$

$\langle proof \rangle$

**lemma** *e-type-callcl-native*:

**assumes**  $\mathcal{S}\cdot\mathcal{C} \vdash [Callcl\ cl] : (t1s' \rightarrow t2s')$

$cl = \text{Func-native } i \text{ } tf \text{ } ts \text{ } es$   
**shows**  $\exists t1s \ t2s \ ts\text{-}c. (t1s' = ts\text{-}c \ @ \ t1s)$   
 $\wedge (t2s' = ts\text{-}c \ @ \ t2s)$   
 $\wedge tf = (t1s \text{-} > \ t2s)$   
 $\wedge i < \text{length } (s\text{-inst } \mathcal{S})$   
 $\wedge ((s\text{-inst } \mathcal{S})!i) \backslash local := (local \ ((s\text{-inst } \mathcal{S})!i)) \ @ \ t1s \ @ \ ts, \text{label}$   
 $:= ([t2s] \ @ \ (\text{label } ((s\text{-inst } \mathcal{S})!i))), \text{return} := \text{Some } t2s \backslash \vdash \text{es} : ([ \text{-} > \ t2s])$   
 $\langle \text{proof} \rangle$

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

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

**lemma** *s-type-unfold*:  
**assumes**  $\mathcal{S} \cdot rs \Vdash\text{-}i \text{ } vs; \text{es} : ts$   
**shows**  $i < \text{length } (s\text{-inst } \mathcal{S})$   
 $(rs = \text{Some } ts) \vee rs = \text{None}$   
 $(\mathcal{S} \cdot ((s\text{-inst } \mathcal{S})!i) \backslash local := (local \ ((s\text{-inst } \mathcal{S})!i)) \ @ \ (\text{map } \text{typeof } vs), \text{return} :=$   
 $rs) \vdash \text{es} : ([ \text{-} > \ ts])$   
 $\langle \text{proof} \rangle$

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

**lemma** *e-type-local-shallow*:  
**assumes**  $\mathcal{S} \cdot \mathcal{C} \vdash [\text{Local } n \ i \ vs \ \text{es}] : (ts \text{-} > \ ts')$   
**shows**  $\exists t1s. \text{length } t1s = n \wedge ts' = ts \ @ \ t1s \wedge (\mathcal{S} \cdot (\text{Some } t1s) \Vdash\text{-}i \ \text{vs}; \text{es} : t1s)$   
 $\langle \text{proof} \rangle$

**lemma** *e-type-const-unwrap*:  
**assumes**  $is\text{-const } e$

**shows**  $\exists v. e = \$C v$   
*<proof>*

**lemma** *is-const-list1*:  
**assumes**  $ves = \text{map } (Basic \circ EConst) vs$   
**shows** *const-list ves*  
*<proof>*

**lemma** *is-const-list*:  
**assumes**  $ves = \$\$* vs$   
**shows** *const-list ves*  
*<proof>*

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

**lemma** *e-type-const1*:  
**assumes** *is-const e*  
**shows**  $\exists t. (\mathcal{S}\cdot\mathcal{C} \vdash [e] : (ts \rightarrow ts@[t]))$   
*<proof>*

**lemma** *e-type-const*:  
**assumes** *is-const e*  
 $\mathcal{S}\cdot\mathcal{C} \vdash [e] : (ts \rightarrow ts')$   
**shows**  $\exists t. (ts' = ts@[t]) \wedge (\mathcal{S}\cdot\mathcal{C}' \vdash [e] : ([\ ] \rightarrow [t]))$   
*<proof>*

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

**lemma** *e-type-const-list*:  
**assumes** *const-list vs*  
 $\mathcal{S}\cdot\mathcal{C} \vdash vs : (ts \rightarrow ts')$   
**shows**  $\exists tvs. ts' = ts @ tvs \wedge \text{length } vs = \text{length } tvs \wedge (\mathcal{S}\cdot\mathcal{C}' \vdash vs : ([\ ] \rightarrow tvs))$   
*<proof>*

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

*<proof>*

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

**assumes** *const-list vs*

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

**shows**  $\exists vs1\ vs2. (\mathcal{S}\cdot\mathcal{C} \vdash vs1 : (\square \rightarrow ts1))$

$\wedge (\mathcal{S}\cdot\mathcal{C} \vdash vs2 : (ts1 \rightarrow (ts1 @ ts2)))$

$\wedge vs = vs1 @ vs2$

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

$\wedge \text{const-list } vs2$

*<proof>*

**lemma** *e-type-const-conv-vs:*

**assumes** *const-list ves*

**shows**  $\exists vs. ves = \$\$* vs$

*<proof>*

**lemma** *types-exist-lfilled:*

**assumes** *Lfilled k lholed es lfilled*

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

**shows**  $\exists t1s\ t2s\ C' \text{ arb-label}. (\mathcal{S}\cdot\mathcal{C}(\text{label} := \text{arb-label}@(\text{label } C))) \vdash es : (t1s \rightarrow t2s)$

*<proof>*

**lemma** *types-exist-lfilled-weak:*

**assumes** *Lfilled k lholed es lfilled*

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

**shows**  $\exists t1s\ t2s\ C' \text{ arb-label } \text{arb-return}. (\mathcal{S}\cdot\mathcal{C}(\text{label} := \text{arb-label}, \text{return} := \text{arb-return})) \vdash es : (t1s \rightarrow t2s)$

*<proof>*

**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>*

**lemma** *store-typing-imp-glob-agree:*

**assumes** *store-typing s S*

$i < \text{length } (s\text{-inst } S)$

$j < \text{length } (\text{global } ((s\text{-inst } S)!i))$

**shows**  $(\text{sglob-ind } s\ i\ j) < \text{length } (s\text{-globs } S)$

$\text{glob-agree } (\text{sglob } s\ i\ j) ((s\text{-globs } S)!(\text{sglob-ind } s\ i\ j))$

$((s\text{-globs } S)!(\text{sglob-ind } s\ i\ j)) = (\text{global } ((s\text{-inst } S)!i))!j$

*<proof>*

**lemma** *store-typing-imp-mem-agree-Some*:

**assumes** *store-typing*  $s \mathcal{S}$

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

$\text{smem-ind } s \ i = \text{Some } j$

**shows**  $j < \text{length } (s\text{-mem } \mathcal{S})$

$\text{mem-agree } ((\text{mem } s)!j) ((s\text{-mem } \mathcal{S})!j)$

$\exists x. ((s\text{-mem } \mathcal{S})!j) = x \wedge (\text{memory } ((s\text{-inst } \mathcal{S})!i)) = \text{Some } x$

*<proof>*

**lemma** *store-typing-imp-mem-agree-None*:

**assumes** *store-typing*  $s \mathcal{S}$

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

$\text{smem-ind } s \ i = \text{None}$

**shows**  $(\text{memory } ((s\text{-inst } \mathcal{S})!i)) = \text{None}$

*<proof>*

**lemma** *store-mem-exists*:

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

*store-typing*  $s \mathcal{S}$

**shows**  $\text{Option.is-none } (\text{memory } ((s\text{-inst } \mathcal{S})!i)) = \text{Option.is-none } (\text{inst.mem } ((\text{inst } s)!i))$

*<proof>*

**lemma** *store-preserved-mem*:

**assumes** *store-typing*  $s \mathcal{S}$

$s' = s(\text{s.mem} := (\text{s.mem } s)[i := \text{mem}'])$

$\text{mem-size } \text{mem}' \geq \text{mem-size } \text{orig-mem}$

$((\text{s.mem } s)!i) = \text{orig-mem}$

**shows** *store-typing*  $s' \mathcal{S}$

*<proof>*

**lemma** *types-agree-imp-e-typing*:

**assumes** *types-agree*  $t \ v$

**shows**  $\mathcal{S} \cdot \mathcal{C} \vdash [\mathcal{C} \ v] : ([\ ] \rightarrow [t])$

*<proof>*

**lemma** *list-types-agree-imp-e-typing*:

**assumes** *list-all2 types-agree*  $ts \ vs$

**shows**  $\mathcal{S} \cdot \mathcal{C} \vdash \mathcal{S}\mathcal{S}^* \ vs : ([\ ] \rightarrow ts)$

*<proof>*

**lemma** *b-e-typing-imp-list-types-agree*:

**assumes**  $\mathcal{C} \vdash (\text{map } (\lambda v. \mathcal{C} \ v) \ vs) : (ts' \rightarrow ts'@ts)$

**shows** *list-all2 types-agree*  $ts \ vs$

*<proof>*

**lemma** *e-typing-imp-list-types-agree*:

**assumes**  $\mathcal{S} \cdot \mathcal{C} \vdash (\mathcal{S}\mathcal{S}^* \ vs) : (ts' \rightarrow ts'@ts)$

**shows** *list-all2 types-agree*  $ts \ vs$

*<proof>*

**lemma** *store-extension-imp-store-typing*:

**assumes** *store-extension*  $s$   $s'$

*store-typing*  $s$   $\mathcal{S}$

**shows** *store-typing*  $s'$   $\mathcal{S}$

*<proof>*

**lemma** *lfilled-deterministic*:

**assumes** *Lfilled*  $k$  *lfilled*  $es$   $les$

*Lfilled*  $k$  *lfilled*  $es$   $les'$

**shows**  $les = les'$

*<proof>*

**end**

## 6 Lemmas for Soundness Proof

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

### 6.1 Preservation

**lemma** *t-cut*: **assumes** *cut*  $t$   $sx$   $v = \text{Some } v'$  **shows**  $t = \text{typeof } v'$

*<proof>*

**lemma** *store-preserved1*:

**assumes**  $(\downarrow s; vs; es) \rightsquigarrow\text{-}i (\downarrow s'; vs'; es')$

*store-typing*  $s$   $\mathcal{S}$

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

$C = ((s\text{-inst } \mathcal{S})!i)(\downarrow local := local ((s\text{-inst } \mathcal{S})!i) @ (map\ \text{typeof } vs), label :=$

$arb\text{-}label, return := arb\text{-}return)$

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

**shows** *store-typing*  $s'$   $\mathcal{S}$

*<proof>*

**lemma** *store-preserved*:

**assumes**  $(\downarrow s; vs; es) \rightsquigarrow\text{-}i (\downarrow s'; vs'; es')$

*store-typing*  $s$   $\mathcal{S}$

$\mathcal{S}\cdot\text{None} \Vdash\text{-}i vs; es : ts$

**shows** *store-typing*  $s'$   $\mathcal{S}$

*<proof>*

**lemma** *typeof-unop-testop*:

**assumes**  $\mathcal{S}\cdot\mathcal{C} \vdash [\mathcal{S}C\ v, \mathcal{S}e] : (ts \rightarrow ts')$

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

**shows**  $(\text{typeof } v) = t$

$e = (\text{Unop-}f\ t\ \text{fop}) \implies is\text{-float-}t\ t$

*<proof>*

**lemma** *typeof-cutop*:

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

**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**  $\mathcal{S}\cdot\mathcal{C} \vdash [\$C v'] : (ts \rightarrow ts')$   
 $\langle \text{proof} \rangle$

**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**  $\text{typeof } v1 = t$   
 $\text{typeof } v2 = t$   
 $e = \text{Binop-}f \ t \ \text{fop} \implies \text{is-float-}t \ t$   
 $e = \text{Relop-}f \ t \ \text{frop} \implies \text{is-float-}t \ t$   
 $\langle \text{proof} \rangle$

**lemma** *types-preserved-binop-relop:*

**assumes**  $(\llbracket \$C v1, \$C v2, \$e \rrbracket) \rightsquigarrow (\llbracket \$C v' \rrbracket)$   
 $\mathcal{S}\cdot\mathcal{C} \vdash [\$C v1, \$C v2, \$e] : (ts \rightarrow ts')$   
 $e = \text{Binop-}i \ t \ \text{iop} \vee e = \text{Binop-}f \ t \ \text{fop} \vee e = \text{Relop-}i \ t \ \text{irop} \vee e = \text{Relop-}f$   
 $t \ \text{frop}$   
**shows**  $\mathcal{S}\cdot\mathcal{C} \vdash [\$C v'] : (ts \rightarrow ts')$   
 $\langle \text{proof} \rangle$

**lemma** *types-preserved-drop:*

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

**lemma** *types-preserved-select:*

**assumes**  $(\llbracket \$C v1, \$C v2, \$C vn, \$e \rrbracket) \rightsquigarrow (\llbracket \$C v3 \rrbracket)$   
 $\mathcal{S}\cdot\mathcal{C} \vdash [\$C v1, \$C v2, \$C vn, \$e] : (ts \rightarrow ts')$   
 $(e = \text{Select})$   
**shows**  $\mathcal{S}\cdot\mathcal{C} \vdash [\$C v3] : (ts \rightarrow ts')$   
 $\langle \text{proof} \rangle$

**lemma** *types-preserved-block:*

**assumes**  $(\langle vs @ [\text{\$Block } (tn \rightarrow tm) \text{ es}] \rangle \rightsquigarrow \langle [\text{\$Label } m \ ] (vs @ (\text{\$* es})) \rangle)$   
 $\mathcal{S}\cdot\mathcal{C} \vdash vs @ [\text{\$Block } (tn \rightarrow tm) \text{ es}] : (ts \rightarrow ts')$   
*const-list vs*  
*length vs = n*  
*length tn = n*  
*length tm = m*  
**shows**  $\mathcal{S}\cdot\mathcal{C} \vdash [\text{\$Label } m \ ] (vs @ (\text{\$* es})) : (ts \rightarrow ts')$   
 $\langle \text{proof} \rangle$

**lemma types-preserved-if:**  
**assumes**  $(\langle [\text{\$C ConstInt32 } n, \text{\$If } tf \text{ e1s } \text{ e2s}] \rangle \rightsquigarrow \langle [\text{\$Block } tf \text{ es}'] \rangle)$   
 $\mathcal{S}\cdot\mathcal{C} \vdash [\text{\$C ConstInt32 } n, \text{\$If } tf \text{ e1s } \text{ e2s}] : (ts \rightarrow ts')$   
**shows**  $\mathcal{S}\cdot\mathcal{C} \vdash [\text{\$Block } tf \text{ es}'] : (ts \rightarrow ts')$   
 $\langle \text{proof} \rangle$

**lemma types-preserved-tee-local:**  
**assumes**  $(\langle [v, \text{\$Tee-local } i] \rangle \rightsquigarrow \langle [v, v, \text{\$Set-local } i] \rangle)$   
 $\mathcal{S}\cdot\mathcal{C} \vdash [v, \text{\$Tee-local } i] : (ts \rightarrow ts')$   
*is-const v*  
**shows**  $\mathcal{S}\cdot\mathcal{C} \vdash [v, v, \text{\$Set-local } i] : (ts \rightarrow ts')$   
 $\langle \text{proof} \rangle$

**lemma types-preserved-loop:**  
**assumes**  $(\langle vs @ [\text{\$Loop } (t1s \rightarrow t2s) \text{ es}] \rangle \rightsquigarrow \langle [\text{\$Label } n [\text{\$Loop } (t1s \rightarrow t2s) \text{ es}] (vs @ (\text{\$* es})) \rangle)$   
 $\mathcal{S}\cdot\mathcal{C} \vdash vs @ [\text{\$Loop } (t1s \rightarrow t2s) \text{ es}] : (ts \rightarrow ts')$   
*const-list vs*  
*length vs = n*  
*length t1s = n*  
*length t2s = m*  
**shows**  $\mathcal{S}\cdot\mathcal{C} \vdash [\text{\$Label } n [\text{\$Loop } (t1s \rightarrow t2s) \text{ es}] (vs @ (\text{\$* es})) : (ts \rightarrow ts')$   
 $\langle \text{proof} \rangle$

**lemma types-preserved-label-value:**  
**assumes**  $(\langle [\text{\$Label } n \text{ es0 } vs] \rangle \rightsquigarrow \langle vs \rangle)$   
 $\mathcal{S}\cdot\mathcal{C} \vdash [\text{\$Label } n \text{ es0 } vs] : (ts \rightarrow ts')$   
*const-list vs*  
**shows**  $\mathcal{S}\cdot\mathcal{C} \vdash vs : (ts \rightarrow ts')$   
 $\langle \text{proof} \rangle$

**lemma types-preserved-br-if:**  
**assumes**  $(\langle [\text{\$C ConstInt32 } n, \text{\$Br-if } i] \rangle \rightsquigarrow \langle e \rangle)$   
 $\mathcal{S}\cdot\mathcal{C} \vdash [\text{\$C ConstInt32 } n, \text{\$Br-if } i] : (ts \rightarrow ts')$   
 $e = [\text{\$Br } i] \vee e = []$   
**shows**  $\mathcal{S}\cdot\mathcal{C} \vdash e : (ts \rightarrow ts')$   
 $\langle \text{proof} \rangle$

**lemma types-preserved-br-table:**  
**assumes**  $(\langle [\text{\$C ConstInt32 } c, \text{\$Br-table } is \text{ i}] \rangle \rightsquigarrow \langle [\text{\$Br } i'] \rangle)$

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

**lemma** *types-preserved-local-const*:  
**assumes**  $([Local \ n \ i \ vs \ es]) \rightsquigarrow (es)$   
 $\mathcal{S}\cdot\mathcal{C} \vdash [Local \ n \ i \ vs \ es] : (ts \rightarrow ts')$   
*const-list*  $es$   
**shows**  $\mathcal{S}\cdot\mathcal{C} \vdash es : (ts \rightarrow ts')$   
 $\langle \text{proof} \rangle$

**lemma** *typing-map-typeof*:  
**assumes**  $ves = \$\$* \ vs$   
 $\mathcal{S}\cdot\mathcal{C} \vdash ves : ([\ ] \rightarrow tvs)$   
**shows**  $tvs = \text{map } \text{typeof} \ vs$   
 $\langle \text{proof} \rangle$

**lemma** *types-preserved-call-indirect-Some*:  
**assumes**  $\mathcal{S}\cdot\mathcal{C} \vdash [\$C \text{ ConstInt32 } c, \$Call\text{-indirect } j] : (ts \rightarrow ts')$   
 $\text{stab } s \ i' \ (\text{nat-of-int } c) = \text{Some } cl$   
 $\text{stypes } s \ i' \ j = tf$   
 $cl\text{-type } cl = tf$   
 $\text{store-typing } s \ \mathcal{S}$   
 $i' < \text{length } (\text{inst } s)$   
 $\mathcal{C} = (s\text{-inst } \mathcal{S} \ ! \ i') \ (\text{local} := \text{local } (s\text{-inst } \mathcal{S} \ ! \ i') \ @ \ tvs, \ \text{label} := \text{arb-labs},$   
 $\text{return} := \text{arb-return})$   
**shows**  $\mathcal{S}\cdot\mathcal{C} \vdash [Callcl \ cl] : (ts \rightarrow ts')$   
 $\langle \text{proof} \rangle$

**lemma** *types-preserved-call-indirect-None*:  
**assumes**  $\mathcal{S}\cdot\mathcal{C} \vdash [\$C \text{ ConstInt32 } c, \$Call\text{-indirect } j] : (ts \rightarrow ts')$   
**shows**  $\mathcal{S}\cdot\mathcal{C} \vdash [Trap] : (ts \rightarrow ts')$   
 $\langle \text{proof} \rangle$

**lemma** *types-preserved-callcl-native*:  
**assumes**  $\mathcal{S}\cdot\mathcal{C} \vdash ves \ @ \ [Callcl \ cl] : (ts \rightarrow ts')$   
 $cl = \text{Func-native } i \ (t1s \rightarrow t2s) \ tfs \ es$   
 $ves = \$\$* \ vs$   
 $\text{length } vs = n$   
 $\text{length } tfs = k$   
 $\text{length } t1s = n$   
 $\text{length } t2s = m$   
 $n\text{-zeros } tfs = zs$   
 $\text{store-typing } s \ \mathcal{S}$   
**shows**  $\mathcal{S}\cdot\mathcal{C} \vdash [Local \ m \ i \ (vs \ @ \ zs) \ [\$Block \ ([\ ] \rightarrow t2s) \ es]] : (ts \rightarrow ts')$   
 $\langle \text{proof} \rangle$

**lemma** *types-preserved-callcl-host-some*:

**assumes**  $\mathcal{S}\cdot\mathcal{C} \vdash ves @ [Callcl\ cl] : (ts \rightarrow ts')$   
 $cl = Func-host\ (t1s \rightarrow t2s)\ f$   
 $ves = \mathbb{S}\mathbb{S}^* vcs$   
 $length\ vcs = n$   
 $length\ t1s = n$   
 $length\ t2s = m$   
 $host-apply\ s\ (t1s \rightarrow t2s)\ f\ vcs\ hs = Some\ (s',\ vcs')$   
 $store-typing\ s\ \mathcal{S}$   
**shows**  $\mathcal{S}\cdot\mathcal{C} \vdash \mathbb{S}\mathbb{S}^* vcs' : (ts \rightarrow ts')$   
 $\langle proof \rangle$

**lemma** *types-imp-concat*:  
**assumes**  $\mathcal{S}\cdot\mathcal{C} \vdash es @ [e] @ es' : (ts \rightarrow ts')$   
 $\bigwedge tes\ tes'. ((\mathcal{S}\cdot\mathcal{C} \vdash [e] : (tes \rightarrow tes')) \implies (\mathcal{S}\cdot\mathcal{C} \vdash [e'] : (tes \rightarrow tes')))$   
**shows**  $\mathcal{S}\cdot\mathcal{C} \vdash es @ [e'] @ es' : (ts \rightarrow ts')$   
 $\langle proof \rangle$

**lemma** *type-const-return*:  
**assumes**  $Lfilled\ i\ lholed\ (vs @ [Return])\ LI$   
 $(return\ \mathcal{C}) = Some\ tcs$   
 $length\ tcs = length\ vs$   
 $\mathcal{S}\cdot\mathcal{C} \vdash LI : (ts \rightarrow ts')$   
 $const-list\ vs$   
**shows**  $\mathcal{S}\cdot\mathcal{C}' \vdash vs : ([] \rightarrow tcs)$   
 $\langle proof \rangle$

**lemma** *types-preserved-return*:  
**assumes**  $([Local\ n\ i\ vls\ LI]) \rightsquigarrow (ves)$   
 $\mathcal{S}\cdot\mathcal{C} \vdash [Local\ n\ i\ vls\ LI] : (ts \rightarrow ts')$   
 $const-list\ ves$   
 $length\ ves = n$   
 $Lfilled\ j\ lholed\ (ves @ [Return])\ LI$   
**shows**  $\mathcal{S}\cdot\mathcal{C} \vdash ves : (ts \rightarrow ts')$   
 $\langle proof \rangle$

**lemma** *type-const-br*:  
**assumes**  $Lfilled\ i\ lholed\ (vs @ [Br\ (i+k)])\ LI$   
 $length\ (label\ \mathcal{C}) > k$   
 $(label\ \mathcal{C})!k = tcs$   
 $length\ tcs = length\ vs$   
 $\mathcal{S}\cdot\mathcal{C} \vdash LI : (ts \rightarrow ts')$   
 $const-list\ vs$   
**shows**  $\mathcal{S}\cdot\mathcal{C}' \vdash vs : ([] \rightarrow tcs)$   
 $\langle proof \rangle$

**lemma** *types-preserved-br*:  
**assumes**  $([Label\ n\ es0\ LI]) \rightsquigarrow (vs @ es0)$   
 $\mathcal{S}\cdot\mathcal{C} \vdash [Label\ n\ es0\ LI] : (ts \rightarrow ts')$   
 $const-list\ vs$

$length\ vs = n$   
 $Lfilled\ i\ lhole\ (vs\ @\ [\$Br\ i])\ LI$   
**shows**  $\mathcal{S}\cdot\mathcal{C} \vdash (vs\ @\ es0) : (ts \rightarrow ts')$   
 $\langle proof \rangle$

**lemma** *store-local-label-empty*:  
**assumes**  $i < length\ (s-inst\ \mathcal{S})$   
 $store-typing\ s\ \mathcal{S}$   
**shows**  $label\ ((s-inst\ \mathcal{S})!i) = []\ local\ ((s-inst\ \mathcal{S})!i) = []$   
 $\langle proof \rangle$

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

**lemma** *types-preserved-b-e*:  
**assumes**  $(es) \rightsquigarrow (es')$   
 $store-typing\ s\ \mathcal{S}$   
 $\mathcal{S}\cdot None \Vdash -i\ vs; es : ts$   
**shows**  $\mathcal{S}\cdot None \Vdash -i\ vs; es' : ts$   
 $\langle proof \rangle$

**lemma** *types-preserved-store*:  
**assumes**  $\mathcal{S}\cdot\mathcal{C} \vdash [\$C\ ConstInt32\ k,\ \$C\ v,\ \$Store\ t\ tp\ a\ off] : (ts \rightarrow ts')$   
**shows**  $\mathcal{S}\cdot\mathcal{C} \vdash [] : (ts \rightarrow ts')$   
 $types-agree\ t\ v$   
 $\langle proof \rangle$

**lemma** *types-preserved-current-memory*:  
**assumes**  $\mathcal{S}\cdot\mathcal{C} \vdash [\$Current-memory] : (ts \rightarrow ts')$   
**shows**  $\mathcal{S}\cdot\mathcal{C} \vdash [\$C\ ConstInt32\ c] : (ts \rightarrow ts')$   
 $\langle proof \rangle$

**lemma** *types-preserved-grow-memory*:  
**assumes**  $\mathcal{S}\cdot\mathcal{C} \vdash [\$C\ ConstInt32\ c,\ \$Grow-memory] : (ts \rightarrow ts')$   
**shows**  $\mathcal{S}\cdot\mathcal{C} \vdash [\$C\ ConstInt32\ c'] : (ts \rightarrow ts')$   
 $\langle proof \rangle$

**lemma** *types-preserved-set-global*:  
**assumes**  $\mathcal{S}\cdot\mathcal{C} \vdash [\$C\ v,\ \$Set-global\ j] : (ts \rightarrow ts')$   
**shows**  $\mathcal{S}\cdot\mathcal{C} \vdash [] : (ts \rightarrow ts')$   
 $tg-t\ (global\ \mathcal{C}\ !\ j) = typeof\ v$   
 $\langle proof \rangle$

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

$\text{typeof } v = t$   
**shows**  $\mathcal{S}\cdot\mathcal{C} \vdash [\$C v] : (ts \rightarrow ts')$   
 $\langle \text{proof} \rangle$

**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**  $\mathcal{S}\cdot\mathcal{C} \vdash [\$C v] : (ts \rightarrow ts')$   
 $\langle \text{proof} \rangle$

**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**  $(\mathcal{S}\cdot\mathcal{C} \vdash [] : (ts \rightarrow ts')) \wedge \text{map } \text{typeof } (vi @ [v] @ vs) = \text{map } \text{typeof } (vi @ [v'] @ vs)$   
 $\langle \text{proof} \rangle$

**lemma** *types-preserved-get-global*:  
**assumes**  $\text{typeof } (\text{sglob}\text{-val } s \ i \ j) = \text{tg}\text{-t } (\text{global } \mathcal{C} \ ! \ j)$   
 $\mathcal{S}\cdot\mathcal{C} \vdash [\$Get\text{-global } j] : (ts \rightarrow ts')$   
**shows**  $\mathcal{S}\cdot\mathcal{C} \vdash [\$C \text{sglob}\text{-val } s \ i \ j] : (ts \rightarrow ts')$   
 $\langle \text{proof} \rangle$

**lemma** *lholed-same-type*:  
**assumes**  $L\text{filled } k \ \text{lholed } es \ les$   
 $L\text{filled } k \ \text{lholed } es' \ les'$   
 $\mathcal{S}\cdot\mathcal{C} \vdash les : (ts \rightarrow ts')$   
 $\bigwedge \text{arb}\text{-labs } ts \ ts'$   
 $\mathcal{S}\cdot(\mathcal{C}(\text{label} := \text{arb}\text{-labs}@(\text{label } \mathcal{C}))) \vdash es : (ts \rightarrow ts')$   
 $\implies \mathcal{S}\cdot(\mathcal{C}(\text{label} := \text{arb}\text{-labs}@(\text{label } \mathcal{C}))) \vdash es' : (ts \rightarrow ts')$   
**shows**  $(\mathcal{S}\cdot\mathcal{C} \vdash les' : (ts \rightarrow ts'))$   
 $\langle \text{proof} \rangle$

**lemma** *types-preserved-e1*:  
**assumes**  $(\!|s; vs; es|) \rightsquigarrow\text{-}i \ (\!|s'; vs'; es'|)$   
 $\text{store}\text{-typing } s \ \mathcal{S}$   
 $tvs = \text{map } \text{typeof } vs$   
 $i < \text{length } (\text{inst } s)$   
 $\mathcal{C} = ((s\text{-inst } \mathcal{S})!i)(\text{local} := (\text{local } ((s\text{-inst } \mathcal{S})!i) @ tvs), \text{label} := \text{arb}\text{-labs},$   
 $\text{return} := \text{arb}\text{-return})$   
 $\mathcal{S}\cdot\mathcal{C} \vdash es : (ts \rightarrow ts')$   
**shows**  $(\mathcal{S}\cdot\mathcal{C} \vdash es' : (ts \rightarrow ts')) \wedge (\text{map } \text{typeof } vs = \text{map } \text{typeof } vs')$   
 $\langle \text{proof} \rangle$

**lemma** *types-preserved-e*:  
**assumes**  $(\!|s; vs; es|) \rightsquigarrow\text{-}i \ (\!|s'; vs'; es'|)$   
 $\text{store}\text{-typing } s \ \mathcal{S}$

$\mathcal{S}\cdot\text{None} \Vdash -i \text{ vs}; es : ts$   
**shows**  $\mathcal{S}\cdot\text{None} \Vdash -i \text{ vs}'; es' : ts$   
 $\langle \text{proof} \rangle$

## 6.2 Progress

**lemma** *const-list-no-progress*:  
**assumes** *const-list*  $es$   
**shows**  $\neg(\text{!}s; \text{vs}; es) \rightsquigarrow - i (\text{!}s'; \text{vs}'; es')$   
 $\langle \text{proof} \rangle$

**lemma** *empty-no-progress*:  
**assumes**  $es = []$   
**shows**  $\neg(\text{!}s; \text{vs}; es) \rightsquigarrow - i (\text{!}s'; \text{vs}'; es')$   
 $\langle \text{proof} \rangle$

**lemma** *trap-no-progress*:  
**assumes**  $es = [\text{Trap}]$   
**shows**  $\neg(\text{!}s; \text{vs}; es) \rightsquigarrow - i (\text{!}s'; \text{vs}'; es')$   
 $\langle \text{proof} \rangle$

**lemma** *terminal-no-progress*:  
**assumes** *const-list*  $es \vee es = [\text{Trap}]$   
**shows**  $\neg(\text{!}s; \text{vs}; es) \rightsquigarrow - i (\text{!}s'; \text{vs}'; es')$   
 $\langle \text{proof} \rangle$

**lemma** *progress-L0*:  
**assumes**  $(\text{!}s; \text{vs}; es) \rightsquigarrow - i (\text{!}s'; \text{vs}'; es')$   
*const-list*  $cs$   
**shows**  $(\text{!}s; \text{vs}; cs @ es @ es - c) \rightsquigarrow - i (\text{!}s'; \text{vs}'; cs @ es' @ es - c)$   
 $\langle \text{proof} \rangle$

**lemma** *progress-L0-left*:  
**assumes**  $(\text{!}s; \text{vs}; es) \rightsquigarrow - i (\text{!}s'; \text{vs}'; es')$   
*const-list*  $cs$   
**shows**  $(\text{!}s; \text{vs}; cs @ es) \rightsquigarrow - i (\text{!}s'; \text{vs}'; cs @ es')$   
 $\langle \text{proof} \rangle$

**lemma** *progress-L0-trap*:  
**assumes** *const-list*  $cs$   
 $cs \neq [] \vee es \neq []$   
**shows**  $\exists a. (\text{!}s; \text{vs}; cs @ [\text{Trap}] @ es) \rightsquigarrow - i (\text{!}s; \text{vs}; [\text{Trap}])$   
 $\langle \text{proof} \rangle$

**lemma** *progress-LN*:  
**assumes**  $(\text{Lfilled } j \text{ lholed } [\text{\$Br } (j+k)] \text{ } es)$   
 $\mathcal{S}\cdot\mathcal{C} \vdash es : ([\ ] \rightarrow ts)$   
 $(\text{label } \mathcal{C})!k = tvs$   
**shows**  $\exists \text{ lholed}' \text{ vs } \mathcal{C}'. (\text{Lfilled } j \text{ lholed}' (vs @ [\text{\$Br } (j+k)]) \text{ } es)$

$\wedge (\mathcal{S}\cdot\mathcal{C}' \vdash vs : (\square \rightarrow ts))$   
 $\wedge \text{const-list } vs$

*<proof>*

**lemma** *progress-LN-return*:  
**assumes**  $(Lfilled\ j\ lholed\ [\$Return]\ es)$   
 $\mathcal{S}\cdot\mathcal{C} \vdash es : (\square \rightarrow ts)$   
 $(return\ \mathcal{C}) = Some\ tvs$   
**shows**  $\exists\ lholed'\ vs\ \mathcal{C}'.$   $(Lfilled\ j\ lholed'\ (vs@[\$Return])\ es)$   
 $\wedge (\mathcal{S}\cdot\mathcal{C}' \vdash vs : (\square \rightarrow tvs))$   
 $\wedge \text{const-list } vs$

*<proof>*

**lemma** *progress-LN1*:  
**assumes**  $(Lfilled\ j\ lholed\ [\$Br\ (j+k)]\ es)$   
 $\mathcal{S}\cdot\mathcal{C} \vdash es : (ts \rightarrow ts')$   
**shows**  $length\ (label\ \mathcal{C}) > k$

*<proof>*

**lemma** *progress-LN2*:  
**assumes**  $(Lfilled\ j\ lholed\ e1s\ lfilled)$   
**shows**  $\exists\ lfilled'.$   $(Lfilled\ j\ lholed\ e2s\ lfilled')$

*<proof>*

**lemma** *const-of-const-list*:  
**assumes**  $length\ cs = 1$   
 $\text{const-list } cs$   
**shows**  $\exists\ v. cs = [\$C\ v]$

*<proof>*

**lemma** *const-of-i32*:  
**assumes**  $\text{const-list } cs$   
 $\mathcal{S}\cdot\mathcal{C} \vdash cs : (\square \rightarrow [(T-i32)])$   
**shows**  $\exists\ c. cs = [\$C\ ConstInt32\ c]$

*<proof>*

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

*<proof>*

**lemma** *const-of-f32*:  
**assumes**  $\text{const-list } cs$   
 $\mathcal{S}\cdot\mathcal{C} \vdash cs : (\square \rightarrow [T-f32])$   
**shows**  $\exists\ c. cs = [\$C\ ConstFloat32\ c]$

*<proof>*

**lemma** *const-of-f64*:

**assumes** *const-list cs*  
 $\mathcal{S}\cdot\mathcal{C} \vdash cs : (\square \rightarrow [T\text{-}f64])$   
**shows**  $\exists c. cs = [\$C \text{ ConstFloat64 } c]$   
 $\langle \text{proof} \rangle$

**lemma** *progress-unop-testop-i*:  
**assumes**  $\mathcal{S}\cdot\mathcal{C} \vdash cs : (\square \rightarrow [t])$   
*is-int-t t*  
*const-list cs*  
 $e = \text{Unop-}i \ t \ iop \vee e = \text{Testop } t \ \text{testop}$   
**shows**  $\exists a \ s' \ vs' \ es'. (\!s;vs;cs@([\$e])) \rightsquigarrow\text{-}i (\!s';vs';es')$   
 $\langle \text{proof} \rangle$

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

**lemma** *const-list-split-2*:  
**assumes** *const-list cs*  
 $\mathcal{S}\cdot\mathcal{C} \vdash cs : (\square \rightarrow [t1, t2])$   
**shows**  $\exists c1 \ c2. (\mathcal{S}\cdot\mathcal{C} \vdash [c1] : (\square \rightarrow [t1]))$   
 $\wedge (\mathcal{S}\cdot\mathcal{C} \vdash [c2] : (\square \rightarrow [t2]))$   
 $\wedge cs = [c1, c2]$   
 $\wedge \text{const-list } [c1]$   
 $\wedge \text{const-list } [c2]$   
 $\langle \text{proof} \rangle$

**lemma** *const-list-split-3*:  
**assumes** *const-list cs*  
 $\mathcal{S}\cdot\mathcal{C} \vdash cs : (\square \rightarrow [t1, t2, t3])$   
**shows**  $\exists c1 \ c2 \ c3. (\mathcal{S}\cdot\mathcal{C} \vdash [c1] : (\square \rightarrow [t1]))$   
 $\wedge (\mathcal{S}\cdot\mathcal{C} \vdash [c2] : (\square \rightarrow [t2]))$   
 $\wedge (\mathcal{S}\cdot\mathcal{C} \vdash [c3] : (\square \rightarrow [t3]))$   
 $\wedge cs = [c1, c2, c3]$   
 $\langle \text{proof} \rangle$

**lemma** *progress-binop-relop-i*:  
**assumes**  $\mathcal{S}\cdot\mathcal{C} \vdash cs : (\square \rightarrow [t, t])$   
*is-int-t t*  
*const-list cs*  
 $e = \text{Binop-}i \ t \ iop \vee e = \text{Relop-}i \ t \ irop$   
**shows**  $\exists a \ s' \ vs' \ es'. (\!s;vs;cs@([\$e])) \rightsquigarrow\text{-}i (\!s';vs';es')$   
 $\langle \text{proof} \rangle$

**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**  $\exists a \ s' \ vs' \ es'. (\!|s;vs;cs@[e]\!) \rightsquigarrow\!-\!i (\!|s';vs';es'\!)$   
*<proof>*

**lemma** *progress-b-e:*

**assumes**  $\mathcal{C} \vdash b\text{-es} : (ts \rightarrow ts')$   
 $\mathcal{S}\cdot\mathcal{C} \vdash cs : ([\ ] \rightarrow ts)$   
 $(\bigwedge \text{lholed}. \neg(\text{Lfilled } 0 \text{ lholed } [\text{\$Return}] (cs@[*\text{b-es}])))$   
 $(\bigwedge i \text{ lholed}. \neg(\text{Lfilled } 0 \text{ lholed } [\text{\$Br } (i)] (cs@[*\text{b-es}])))$   
*const-list cs*  
 $\neg \text{const-list } (* \text{ b-es})$   
 $i < \text{length } (s\text{-inst } \mathcal{S})$   
 $\text{length } (\text{local } \mathcal{C}) = \text{length } (vs)$   
 $\text{Option.is-none } (\text{memory } \mathcal{C}) = \text{Option.is-none } (\text{inst.mem } ((\text{inst } s)!i))$   
**shows**  $\exists a \ s' \ vs' \ es'. (\!|s;vs;cs@[*\text{b-es}]\!) \rightsquigarrow\!-\!i (\!|s';vs';es'\!)$   
*<proof>*

**lemma** *progress-e:*

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

**lemma** *progress-e1:*

**assumes**  $\mathcal{S}\cdot\text{None} \Vdash\!-\!i \ vs;es : ts$   
**shows**  $\neg(\text{Lfilled } k \text{ lholed } [\text{\$Return}] \text{es})$   
*<proof>*

**lemma** *progress-e2:*

**assumes**  $\mathcal{S}\cdot\text{None} \Vdash\!-\!i \ vs;es : ts$   
*store-typing s S*  
**shows**  $(\text{Lfilled } k \text{ lholed } [\text{\$Br } (j)] \text{es}) \implies j < k$   
*<proof>*

**lemma** *progress-e3:*

**assumes**  $\mathcal{S}\cdot\text{None} \Vdash\!-\!i \ vs;cs\text{-es} : ts'$   
 $cs\text{-es} \neq [\text{Trap}]$   
 $\neg \text{const-list } (cs\text{-es})$   
*store-typing s S*  
**shows**  $\exists a \ s' \ vs' \ es'. (\!|s;vs;cs\text{-es}\!) \rightsquigarrow\!-\!i (\!|s';vs';es'\!)$   
*<proof>*

end

## 7 Soundness Theorems

**theory** *Wasm-Soundness* **imports** *Main Wasm-Properties* **begin**

**theorem** *preservation*:

**assumes**  $\vdash\text{-}i\ s;vs;es : ts$

$(\langle s;vs;es \rangle \rightsquigarrow\text{-}i\ \langle s';vs';es' \rangle)$

**shows**  $\vdash\text{-}i\ s';vs';es' : ts$

*<proof>*

**theorem** *progress*:

**assumes**  $\vdash\text{-}i\ s;vs;es : ts$

**shows**  $\text{const-list } es \vee es = [\text{Trap}] \vee (\exists a\ s'\ vs'\ es'. \langle s;vs;es \rangle \rightsquigarrow\text{-}i\ \langle s';vs';es' \rangle)$

*<proof>*

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*

*<proof>*

**lemma** *ct-eq-flip*:  $ct\text{-eq}^{-1-1} = ct\text{-eq}$

*<proof>*

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

*<proof>*

**lemma** *ct-list-eq-commute*:

**assumes** *ct-list-eq xs ys*

**shows** *ct-list-eq ys xs*

*<proof>*

**lemma** *ct-list-eq-refl*: *ct-list-eq xs xs*

*<proof>*

**lemma** *ct-list-eq-length*:

**assumes** *ct-list-eq xs ys*

**shows**  $length \ xs = length \ ys$

*<proof>*

**lemma** *ct-list-eq-concat*:

**assumes** *ct-list-eq xs ys*

*ct-list-eq xs' ys'*

**shows** *ct-list-eq (xs@xs') (ys@ys')*

*<proof>*

**lemma** *ct-list-eq-ts-conv-eq*:

$ct\text{-list-eq } (to\text{-ct-list } ts) \ (to\text{-ct-list } ts') = (ts = ts')$

*<proof>*

**lemma** *ct-list-eq-exists*:  $\exists ys. ct\text{-list-eq } xs \ (to\text{-ct-list } ys)$

*<proof>*

**lemma** *ct-list-eq-common-tsome-list*:

$ct\text{-list-eq } xs \ ys = (\exists zs. ct\text{-list-eq } xs \ (to\text{-ct-list } zs) \wedge ct\text{-list-eq } (to\text{-ct-list } zs) \ ys)$

*<proof>*

**lemma** *ct-list-eq-cons-ct-list*:

**assumes** *ct-list-eq (to-ct-list as) (xs @ ys)*

**shows**  $\exists bs \ bs'. as = bs \ @ \ bs' \wedge ct\text{-list-eq } (to\text{-ct-list } bs) \ xs \wedge ct\text{-list-eq } (to\text{-ct-list } bs') \ ys$

*<proof>*

**lemma** *ct-list-eq-cons-ct-list1*:

**assumes** *ct-list-eq* (*to-ct-list as*) (*xs @ (to-ct-list ys)*)  
**shows**  $\exists bs. as = bs @ ys \wedge ct\text{-list-eq } (to\text{-ct-list } bs) \ xs$   
*<proof>*

**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*  
*<proof>*

**lemma** *ct-list-eq-take*:

**assumes** *ct-list-eq xs ys*  
**shows** *ct-list-eq (take n xs) (take n ys)*  
*<proof>*

**lemma** *ct-prefixI [intro?]*:

**assumes**  $ys = as @ zs$   
*ct-list-eq as xs*  
**shows** *ct-prefix xs ys*  
*<proof>*

**lemma** *ct-prefixE [elim?]*:

**assumes** *ct-prefix xs ys*  
**obtains** *as zs where*  $ys = as @ zs$  *ct-list-eq as xs*  
*<proof>*

**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>*

**lemma** *ct-prefix-nil:ct-prefix [] xs*  
 $\neg ct\text{-prefix } (x \# xs) \ []$   
*<proof>*

**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>*

**lemma** *ct-prefix-code [code]*:

$ct\text{-prefix } [] \ xs \longleftrightarrow True$   
 $ct\text{-prefix } (x \# xs) \ [] \longleftrightarrow False$   
 $ct\text{-prefix } (x \# xs) \ (y \# ys) \longleftrightarrow ct\text{-eq } x \ y \wedge ct\text{-prefix } xs \ ys$   
*<proof>*

**lemma** *ct-suffix-to-ct-prefix [code]*:  $ct\text{-suffix } xs \ ys = ct\text{-prefix } (rev \ xs) \ (rev \ ys)$   
*<proof>*

**lemma** *inj-TSome*: *inj TSome*

*<proof>*

**lemma** *to-ct-list-append*:

**assumes** *to-ct-list*  $ts = as@bs$

**shows**  $\exists as'. \text{to-ct-list } as' = as$

$\exists bs'. \text{to-ct-list } bs' = bs$

*<proof>*

**lemma** *ct-suffixI* [*intro?*]:

**assumes**  $ys = as @ zs$

*ct-list-eq*  $zs xs$

**shows** *ct-suffix*  $xs ys$

*<proof>*

**lemma** *ct-suffixE* [*elim?*]:

**assumes** *ct-suffix*  $xs ys$

**obtains**  $as zs$  **where**  $ys = as @ zs$  *ct-list-eq*  $zs xs$

*<proof>*

**lemma** *ct-suffix-nil*: *ct-suffix*  $[] ts$

*<proof>*

**lemma** *ct-suffix-refl*: *ct-suffix*  $ts ts$

*<proof>*

**lemma** *ct-suffix-length*:

**assumes** *ct-suffix*  $ts ts'$

**shows**  $\text{length } ts \leq \text{length } ts'$

*<proof>*

**lemma** *ct-suffix-take*:

**assumes** *ct-suffix*  $ts ts'$

**shows** *ct-suffix*  $((\text{take } (\text{length } ts - n) ts)) ((\text{take } (\text{length } ts' - n) ts'))$

*<proof>*

**lemma** *ct-suffix-ts-conv-suffix*:

*ct-suffix*  $(\text{to-ct-list } ts) (\text{to-ct-list } ts') = \text{suffix } ts ts'$

*<proof>*

**lemma** *ct-suffix-exists*:  $\exists ts-c. \text{ct-suffix } x1 (\text{to-ct-list } ts-c)$

*<proof>*

**lemma** *ct-suffix-ct-list-eq-exists*:

**assumes** *ct-suffix*  $x1 x2$

**shows**  $\exists ts-c. \text{ct-suffix } x1 (\text{to-ct-list } ts-c) \wedge \text{ct-list-eq } (\text{to-ct-list } ts-c) x2$

*<proof>*

**lemma** *ct-suffix-cons-ct-list*:

**assumes** *ct-suffix*  $(xs@ys) (\text{to-ct-list } zs)$

**shows**  $\exists as\ bs.\ zs = as@bs \wedge ct\text{-list}\text{-eq}\ ys\ (to\text{-}ct\text{-list}\ bs) \wedge ct\text{-suffix}\ xs\ (to\text{-}ct\text{-list}\ as)$   
 $\langle proof \rangle$

**lemma** *ct-suffix-cons-ct-list1*:  
**assumes**  $ct\text{-suffix}\ (xs@(to\text{-}ct\text{-list}\ ys))\ (to\text{-}ct\text{-list}\ zs)$   
**shows**  $\exists as.\ zs = as@ys \wedge ct\text{-suffix}\ xs\ (to\text{-}ct\text{-list}\ as)$   
 $\langle proof \rangle$

**lemma** *ct-suffix-cons2*:  
**assumes**  $ct\text{-suffix}\ (xs)\ (ys@zs)$   
 $length\ xs = length\ zs$   
**shows**  $ct\text{-list}\text{-eq}\ xs\ zs$   
 $\langle proof \rangle$

**lemma** *ct-suffix-imp-ct-list-eq*:  
**assumes**  $ct\text{-suffix}\ xs\ ys$   
**shows**  $ct\text{-list}\text{-eq}\ (drop\ (length\ ys - length\ xs)\ ys)\ xs$   
 $\langle proof \rangle$

**lemma** *ct-suffix-extend-ct-list-eq*:  
**assumes**  $ct\text{-suffix}\ xs\ ys$   
 $ct\text{-list}\text{-eq}\ xs'\ ys'$   
**shows**  $ct\text{-suffix}\ (xs@xs')\ (ys@ys')$   
 $\langle proof \rangle$

**lemma** *ct-suffix-extend-any1*:  
**assumes**  $ct\text{-suffix}\ xs\ ys$   
 $length\ xs < length\ ys$   
**shows**  $ct\text{-suffix}\ (TAny\#xs)\ ys$   
 $\langle proof \rangle$

**lemma** *ct-suffix-singleton-any*:  $ct\text{-suffix}\ [TAny]\ [t]$   
 $\langle proof \rangle$

**lemma** *ct-suffix-cons-it*:  $ct\text{-suffix}\ xs\ (xs'@xs)$   
 $\langle proof \rangle$

**lemma** *ct-suffix-singleton*:  
**assumes**  $length\ cts > 0$   
**shows**  $ct\text{-suffix}\ [TAny]\ cts$   
 $\langle proof \rangle$

**lemma** *ct-suffix-less*:  
**assumes**  $ct\text{-suffix}\ (xs@xs')\ ys$   
**shows**  $ct\text{-suffix}\ xs'\ ys$   
 $\langle proof \rangle$

**lemma** *ct-suffix-unfold-one*:  $ct\text{-suffix}\ (xs@[x])\ (ys@[y]) = ((ct\text{-eq}\ x\ y) \wedge ct\text{-suffix}\ xs\ ys)$

*xs ys*)  
 ⟨*proof*⟩

**lemma** *ct-suffix-shared*:  
**assumes** *ct-suffix cts (to-ct-list ts)*  
           *ct-suffix cts' (to-ct-list ts)*  
**shows** *ct-suffix cts cts' ∨ ct-suffix cts' cts*  
 ⟨*proof*⟩

**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⟩

```

**lemma** consume-top-geq:

```

assumes consume (TopType ts) ts' = c-t
  length ts ≥ length ts'
  c-t ≠ Bot
shows (∃ as bs. ts = as@bs ∧ ct-list-eq bs ts' ∧ c-t = TopType as)
⟨proof⟩

```

**lemma** consume-top-leq:

```

assumes consume (TopType ts) ts' = c-t
  length ts ≤ length ts'
  c-t ≠ Bot
shows c-t = TopType []
⟨proof⟩

```

**lemma** consume-type-type:

```

assumes consume xs cons = (Type t-int)
shows ∃ tn. xs = Type tn
⟨proof⟩

```

**lemma** produce-type-type:

```

assumes produce xs cons = (Type tm)
shows ∃ tn. xs = Type tn
⟨proof⟩

```

**lemma** consume-weaken-type:

```

assumes consume (Type tn) cons = (Type t-int)

```

**shows** *consume* (Type (ts@tn)) cons = (Type (ts@t-int))  
 ⟨proof⟩

**lemma** *produce-weaken-type*:  
**assumes** *produce* (Type tn) cons = (Type tm)  
**shows** *produce* (Type (ts@tn)) cons = (Type (ts@tm))  
 ⟨proof⟩

**lemma** *produce-nil*: *produce* ts (Type []) = ts  
 ⟨proof⟩

**lemma** *c-types-agree-id*: *c-types-agree* (Type ts) ts  
 ⟨proof⟩

**lemma** *c-types-agree-top1*: *c-types-agree* (TopType []) ts  
 ⟨proof⟩

**lemma** *c-types-agree-top2*:  
**assumes** *ct-list-eq* ts (to-ct-list ts')  
**shows** *c-types-agree* (TopType ts) (ts'@ts'')  
 ⟨proof⟩

**lemma** *c-types-agree-imp-ct-list-eq*:  
**assumes** *c-types-agree* (TopType cts) ts  
**shows**  $\exists ts' ts''.$  (ts = ts'@ts'')  $\wedge$  *ct-list-eq* cts (to-ct-list ts'')  
 ⟨proof⟩

**lemma** *c-types-agree-not-bot-exists*:  
**assumes** ts  $\neq$  Bot  
**shows**  $\exists ts-c.$  *c-types-agree* ts ts-c  
 ⟨proof⟩

**lemma** *consume-c-types-agree*:  
**assumes** *consume* (Type ts) cts = (Type ts')  
           *c-types-agree* ctn ts  
**shows**  $\exists c-t'.$  *consume* ctn cts = c-t'  $\wedge$  *c-types-agree* c-t' ts'  
 ⟨proof⟩

**lemma** *type-update-type*:  
**assumes** *type-update* (Type ts) (to-ct-list cons) prods = ts'  
           ts'  $\neq$  Bot  
**shows** (ts' = prods  $\wedge$  ( $\exists ts-c.$  prods = (TopType ts-c)))  
            $\vee$  ( $\exists ts-a ts-b.$  prods = Type ts-a  $\wedge$  ts = ts-b@cons  $\wedge$  ts' = Type  
 (ts-b@ts-a))  
 ⟨proof⟩

**lemma** *type-update-empty*: *type-update* ts cons (Type []) = *consume* ts cons  
 ⟨proof⟩

**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'*

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

*<proof>*

**lemma** *type-update-select-length0*:

**assumes** *type-update-select* (*TopType* *cts*) = *tm*

*length* *cts* = 0

*tm*  $\neq$  *Bot*

**shows** *tm* = *TopType* [*TAny*]

*<proof>*

**lemma** *type-update-select-length1*:

**assumes** *type-update-select* (*TopType* *cts*) = *tm*

*length* *cts* = 1

*tm*  $\neq$  *Bot*

**shows** *ct-list-eq* *cts* [*TSome* *T-i32*]

*tm* = *TopType* [*TAny*]

*<proof>*

**lemma** *type-update-select-length2*:

**assumes** *type-update-select* (*TopType* *cts*) = *tm*

*length* *cts* = 2

*tm*  $\neq$  *Bot*

**shows**  $\exists t1 t2. cts = [t1, t2] \wedge ct-eq t2 (TSome T-i32) \wedge tm = TopType [t1]$

*<proof>*

**lemma** *type-update-select-length3*:

**assumes** *type-update-select* (*TopType* *cts*) = (*TopType* *ctm*)

*length* *cts*  $\geq$  3

**shows**  $\exists cts' ct1 ct2 ct3. cts = cts'@[ct1, ct2, ct3] \wedge ct-eq ct3 (TSome T-i32)$

*<proof>*

**lemma** *type-update-select-type-length3*:

**assumes** *type-update-select* (*Type* *tn*) = (*Type* *tm*)

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

*<proof>*

**lemma** *select-return-top-exists*:

**assumes** *select-return-top* *cts* *c1* *c2* = *ctm*

*ctm*  $\neq$  *Bot*

**shows**  $\exists xs. ctm = TopType xs$

*<proof>*

**lemma** *type-update-select-top-exists*:

**assumes** *type-update-select* *xs* = (*TopType tm*)  
**shows**  $\exists tn. xs = \text{TopType } tn$   
 $\langle \text{proof} \rangle$

**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*!( $length\ cts - 3$ )))  
 $\langle \text{proof} \rangle$

**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**  $\exists c' cm'. cm = cm'@[c']$   
 $\wedge ct\text{-suffix}\ (take\ (length\ cts - 3)\ cts)\ (to\text{-ct-list}\ cm')$   
 $\wedge ct\text{-eq}\ c1\ (TSome\ c')$   
 $\wedge ct\text{-eq}\ c2\ (TSome\ c')$   
 $\langle \text{proof} \rangle$

**end**

## 9 Executable Type Checker

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

**fun** *convert-cond* ::  $t \Rightarrow t \Rightarrow sx\ option \Rightarrow bool$  **where**  
 $convert\text{-cond}\ t1\ t2\ sx = ((t1 \neq t2) \wedge (sx = None) = ((is\text{-float-t}\ t1 \wedge is\text{-float-t}\ t2)$   
 $\vee (is\text{-int-t}\ t1 \wedge is\text{-int-t}\ t2 \wedge (t\text{-length}\ t1 < t\text{-length}\ t2))))$

**fun** *same-lab-h* ::  $nat\ list \Rightarrow (t\ list)\ list \Rightarrow t\ list \Rightarrow (t\ list)\ option$  **where**  
 $same\text{-lab-h}\ []\ -\ ts = Some\ ts$   
 $| same\text{-lab-h}\ (i\#is)\ lab\text{-c}\ ts = (if\ i \geq length\ lab\text{-c}$   
 $then\ None$   
 $else\ (if\ lab\text{-c}!i = ts$   
 $then\ same\text{-lab-h}\ is\ lab\text{-c}\ (lab\text{-c}!i)$   
 $else\ None))$

**fun** *same-lab* ::  $nat\ list \Rightarrow (t\ list)\ list \Rightarrow (t\ list)\ option$  **where**  
 $same\text{-lab}\ []\ lab\text{-c} = None$   
 $| same\text{-lab}\ (i\#is)\ lab\text{-c} = (if\ i \geq length\ lab\text{-c}$   
 $then\ None$   
 $else\ same\text{-lab-h}\ is\ lab\text{-c}\ (lab\text{-c}!i))$

**lemma** *same-lab-h-conv-list-all*:  
**assumes** *same-lab-h ils ls ts' = Some ts*

**shows**  $list\text{-}all (\lambda i. i < length\ ls \wedge ls!i = ts) ils \wedge ts' = ts$   
 $\langle proof \rangle$

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

**assumes** *same-lab*  $ils\ ls = Some\ ts$   
**shows**  $list\text{-}all (\lambda i. i < length\ ls \wedge ls!i = ts) ils$   
 $\langle proof \rangle$

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

**assumes**  $list\text{-}all (\lambda i. i < length\ ls \wedge ls!i = ts) ils$   
**shows** *same-lab-h*  $ils\ ls\ ts = Some\ ts$   
 $\langle proof \rangle$

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

**assumes**  $list\text{-}all (\lambda i. i < length\ ls \wedge ls!i = ts) (is@[i])$   
**shows** *same-lab*  $(is@[i])\ ls = Some\ ts$   
 $\langle proof \rangle$

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

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

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

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

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

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

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

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

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

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

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

$| \text{check-single } \mathcal{C} (\text{Br-table } is \text{ } i) \text{ } ts = (\text{case } (\text{same-lab } (is \text{ @ } [i]) (\text{label } \mathcal{C})) \text{ } of$   
 $\quad \text{None } \Rightarrow \text{Bot}$

| *Some tls*  $\Rightarrow$  *type-update ts* (*to-ct-list* (*tls* @ [*T-i32*]))

(*TopType []*)

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

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

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

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

| *check-single C* (*Set-local i*) *ts* = (*if* *i* < *length* (*local C*)  
     *then* *type-update ts* [*TSome ((local C)!i)*] (*Type []*)  
     *else Bot*)

| *check-single C* (*Tee-local i*) *ts* = (*if* *i* < *length* (*local C*)  
     *then* *type-update ts* [*TSome ((local C)!i)*] (*Type [(local*  
*C)!i]*)  
     *else Bot*)

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

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

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

| *check-single C* (*Store t tp a off*) *ts* = (*if* (*memory C*)  $\neq$  *None*  $\wedge$  *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)
⟨proof⟩

```

**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⟩

```

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

```

assumes type-update ts (to-ct-list t-in) (TopType []) = ts'
          ts ≠ Bot
          ts' ≠ Bot
shows ∃ tn. c-types-agree ts tn ∧ suffix t-in tn
⟨proof⟩

```

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

```

assumes type-update ts (to-ct-list t-in) (Type t-out) = ts'
          ts ≠ Bot
          ts' ≠ Bot
          c-types-agree ts' tm
          C ⊢ [e] : (t-in -> t-out)
shows ∃ tn. c-types-agree ts tn ∧ C ⊢ [e] : (tn -> tm)
⟨proof⟩

```

**lemma** *b-e-check-single-sound-unop-testop-cvtop:*

**assumes** *check-single*  $\mathcal{C}$   $e$   $tn' = tm'$   
 $((e = (\text{Unop-}i\ t\ uu) \vee e = (\text{Testop}\ t\ ww)) \wedge \text{is-int-}t\ t)$   
 $\vee (e = (\text{Unop-}f\ t\ ww) \wedge \text{is-float-}t\ t)$   
 $\vee (e = (\text{Cvtop}\ t1\ \text{Convert}\ t\ sx) \wedge \text{convert-cond}\ t1\ t\ sx)$   
 $\vee (e = (\text{Cvtop}\ t1\ \text{Reinterpret}\ t\ sx) \wedge ((t1 \neq t) \wedge t\text{-length}\ t1 = t\text{-length}\ t$   
 $\wedge sx = \text{None}))$   
*c-types-agree*  $tm'\ tm$   
 $tn' \neq \text{Bot}$   
 $tm' \neq \text{Bot}$   
**shows**  $\exists tn. \text{c-types-agree}\ tn'\ tn \wedge \mathcal{C} \vdash [e] : (tn \rightarrow tm)$   
 $\langle \text{proof} \rangle$

**lemma** *b-e-check-single-sound-binop-relop:*

**assumes** *check-single*  $\mathcal{C}$   $e$   $tn' = tm'$   
 $((e = \text{Binop-}i\ t\ iop \wedge \text{is-int-}t\ t)$   
 $\vee (e = \text{Binop-}f\ t\ fop \wedge \text{is-float-}t\ t)$   
 $\vee (e = \text{Relop-}i\ t\ irop \wedge \text{is-int-}t\ t)$   
 $\vee (e = \text{Relop-}f\ t\ frop \wedge \text{is-float-}t\ t))$   
*c-types-agree*  $tm'\ tm$   
 $tn' \neq \text{Bot}$   
 $tm' \neq \text{Bot}$   
**shows**  $\exists tn. \text{c-types-agree}\ tn'\ tn \wedge \mathcal{C} \vdash [e] : (tn \rightarrow tm)$   
 $\langle \text{proof} \rangle$

**lemma** *b-e-type-checker-sound:*

**assumes** *b-e-type-checker*  $\mathcal{C}$   $es$   $(tn \rightarrow tm)$   
**shows**  $\mathcal{C} \vdash es : (tn \rightarrow tm)$   
 $\langle \text{proof} \rangle$

## 10.2 Completeness

**lemma** *check-single-imp:*

**assumes** *check-single*  $\mathcal{C}$   $e$   $ctn = ctm$   
 $ctm \neq \text{Bot}$   
**shows** *check-single*  $\mathcal{C}$   $e = id$   
 $\vee \text{check-single}\ \mathcal{C}\ e = (\lambda ctn. \text{type-update-select}\ ctn)$   
 $\vee (\exists \text{cons prods. } (\text{check-single}\ \mathcal{C}\ e = (\lambda ctn. \text{type-update}\ ctn\ \text{cons}\ \text{prods})))$   
 $\langle \text{proof} \rangle$

**lemma** *check-equiv-fold:*

*check*  $\mathcal{C}$   $es\ ts = \text{foldl}\ (\lambda\ ts\ e. (\text{case}\ ts\ \text{of}\ \text{Bot} \Rightarrow \text{Bot} \mid - \Rightarrow \text{check-single}\ \mathcal{C}\ e\ ts))$   
 $ts\ es$   
 $\langle \text{proof} \rangle$

**lemma** *check-neq-bot-snoc:*

**assumes** *check*  $\mathcal{C}$   $(es@[e])\ ts \neq \text{Bot}$   
**shows** *check*  $\mathcal{C}$   $es\ ts \neq \text{Bot}$

*<proof>*

**lemma** *check-unfold-snoc*:

**assumes** *check*  $\mathcal{C}$  *es*  $ts \neq \text{Bot}$

**shows** *check*  $\mathcal{C}$  (*es*@[*e*]) *ts* = *check-single*  $\mathcal{C}$  *e* (*check*  $\mathcal{C}$  *es* *ts*)

*<proof>*

**lemma** *check-single-imp-weakening*:

**assumes** *check-single*  $\mathcal{C}$  *e* (*Type* *t1s*) = *ctm*

*ctm*  $\neq \text{Bot}$

*c-types-agree* *ctn* *t1s*

*c-types-agree* *ctm* *t2s*

**shows**  $\exists$  *ctm'*. *check-single*  $\mathcal{C}$  *e* *ctn* = *ctm'*  $\wedge$  *c-types-agree* *ctm'* *t2s*

*<proof>*

**lemma** *b-e-type-checker-compose*:

**assumes** *b-e-type-checker*  $\mathcal{C}$  *es* (*t1s*  $\rightarrow$  *t2s*)

*b-e-type-checker*  $\mathcal{C}$  [*e*] (*t2s*  $\rightarrow$  *t3s*)

**shows** *b-e-type-checker*  $\mathcal{C}$  (*es* @ [*e*]) (*t1s*  $\rightarrow$  *t3s*)

*<proof>*

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

**assumes** *check-single*  $\mathcal{C}$  *e* *xs* = (*Type* *tm*)

**shows**  $\exists$  *tn*. *xs* = (*Type* *tn*)

*<proof>*

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

**assumes** *check-single*  $\mathcal{C}$  *e* (*Type* *tn*) = (*Type* *tm*)

**shows** *check-single*  $\mathcal{C}$  *e* (*Type* (*ts*@*tn*)) = *Type* (*ts*@*tm*)

*<proof>*

**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>*

**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*))

*<proof>*

**lemma** *check-bot*: *check*  $\mathcal{C}$  *es* *Bot* = *Bot*

*<proof>*

**lemma** *b-e-check-weaken-top*:

**assumes** *check*  $\mathcal{C}$  *es* (*Type* *tn*) = (*TopType* *tm*)

**shows** *check*  $\mathcal{C}$  *es* (*Type* (*ts*@*tn*)) = (*TopType* *tm*)

*<proof>*

**lemma** *b-e-type-checker-weaken*:  
**assumes** *b-e-type-checker*  $\mathcal{C}$  *es* ( $t1s \rightarrow t2s$ )  
**shows** *b-e-type-checker*  $\mathcal{C}$  *es* ( $ts@t1s \rightarrow ts@t2s$ )  
 $\langle$ *proof* $\rangle$

**lemma** *b-e-type-checker-complete*:  
**assumes**  $\mathcal{C} \vdash es : (tn \rightarrow tm)$   
**shows** *b-e-type-checker*  $\mathcal{C}$  *es* ( $tn \rightarrow tm$ )  
 $\langle$ *proof* $\rangle$

**theorem** *b-e-typing-equiv-b-e-type-checker*:  
**shows**  $(\mathcal{C} \vdash es : (tn \rightarrow tm)) = (b-e-type-checker \mathcal{C} es (tn \rightarrow tm))$   
 $\langle$ *proof* $\rangle$

**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 list*  $\times$  *e list*

**type-synonym** *config-one-tuple* = *s*  $\times$  *v list*  $\times$  *v list*  $\times$  *e*

**type-synonym** *res-tuple* = *s*  $\times$  *v list*  $\times$  *res-step*

**fun** *split-vals* :: *b-e list*  $\Rightarrow$  *v list*  $\times$  *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*  $\Rightarrow$  *v list*  $\times$  *e list* **where**  
*split-vals-e* ( $\$ C v$ )#*es*) = (let (*vs'*, *es'*) = *split-vals-e es* in (*v*#*vs'*, *es'*))  
| *split-vals-e es* = ( $\square$ , *es*)

**fun** *split-n* :: *v list*  $\Rightarrow$  *nat*  $\Rightarrow$  *v list*  $\times$  *v list* **where**  
*split-n*  $\square$  *n* = ( $\square$ ,  $\square$ )  
| *split-n es* 0 = ( $\square$ , *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*)  
 $\langle$ *proof* $\rangle$

**lemma** *split-n-length*:  
**assumes** *split-n es n* = (*es1*, *es2*) *length es*  $\geq$  *n*  
**shows** *length es1* = *n*  
 $\langle$ *proof* $\rangle$

**lemma** *split-n-conv-app*:  
**assumes** *split-n es n* = (*es1*, *es2*)  
**shows** *es* = *es1*@*es2*  
 $\langle$ *proof* $\rangle$

**lemma** *app-conv-split-n*:  
**assumes** *es* = *es1*@*es2*  
**shows** *split-n es (length es1)* = (*es1*, *es2*)  
 $\langle$ *proof* $\rangle$

**lemma** *split-vals-const-list*: *split-vals (map EConst vs)* = (*vs*,  $\square$ )  
 $\langle$ *proof* $\rangle$

**lemma** *split-vals-e-const-list*: *split-vals-e (\$\$\* vs)* = (*vs*,  $\square$ )  
 $\langle$ *proof* $\rangle$

**lemma** *split-vals-e-conv-app*:  
**assumes** *split-vals-e xs* = (*as*, *bs*)  
**shows** *xs* = (\$\$\* *as*)@*bs*  
 $\langle$ *proof* $\rangle$

**abbreviation** *expect* :: '*a option*  $\Rightarrow$  ('*a*  $\Rightarrow$  '*b*)  $\Rightarrow$  '*b*  $\Rightarrow$  '*b* **where**  
*expect a f b*  $\equiv$  (case *a* of  
  *Some a'*  $\Rightarrow$  *f a'*  
  | *None*  $\Rightarrow$  *b*)

**abbreviation** *vs-to-es* :: *v list*  $\Rightarrow$  *e list*  
**where** *vs-to-es v*  $\equiv$  \$\$\* (*rev v*)

**definition** *e-is-trap* :: *e*  $\Rightarrow$  *bool* **where**  
*e-is-trap e* = (case *e* of *Trap*  $\Rightarrow$  *True* | -  $\Rightarrow$  *False*)

**definition** *es-is-trap* :: *e list*  $\Rightarrow$  *bool* **where**  
*es-is-trap* *es* = (case *es* of [*e*]  $\Rightarrow$  *e-is-trap* *e* | -  $\Rightarrow$  *False*)

**lemma**<sup>[simp]</sup>: *e-is-trap* *e* = (*e* = *Trap*)  
 ⟨*proof*⟩

**lemma**<sup>[simp]</sup>: *es-is-trap* *es* = (*es* = [*Trap*])  
 ⟨*proof*⟩

**axiomatization**

*mem-grow-impl*:: *mem*  $\Rightarrow$  *nat*  $\Rightarrow$  *mem option* **where**  
*mem-grow-impl-correct*:(*mem-grow-impl* *m* *n* = *Some* *m'*)  $\Longrightarrow$  (*mem-grow* *m* *n* = *m'*)

**axiomatization**

*host-apply-impl*:: *s*  $\Rightarrow$  *tf*  $\Rightarrow$  *host*  $\Rightarrow$  *v list*  $\Rightarrow$  (*s*  $\times$  *v list*) *option* **where**  
*host-apply-impl-correct*:(*host-apply-impl* *s* *tf* *h* *vs* = *Some* *m'*)  $\Longrightarrow$  ( $\exists$  *hs*. *host-apply* *s* *tf* *h* *vs* *hs* = *Some* *m'*)

**function** (*sequential*)

*run-step* :: *depth*  $\Rightarrow$  *nat*  $\Rightarrow$  *config-tuple*  $\Rightarrow$  *res-tuple*  
**and** *run-one-step* :: *depth*  $\Rightarrow$  *nat*  $\Rightarrow$  *config-one-tuple*  $\Rightarrow$  *res-tuple* **where**  
*run-step* *d* *i* (*s*,*vs*,*es*) = (let (*ves*, *es'*) = *split-vals-e* *es* in  
 case *es'* of  
 []  $\Rightarrow$  (*s*,*vs*, *crash-error*)  
 | *e#es''*  $\Rightarrow$   
 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*  $\Rightarrow$  (*s'*, *vs'*, *RSNormal* (*res*@*es''*))  
 | -  $\Rightarrow$  (*s'*, *vs'*, *r*)))  
 | *run-one-step* *d* *i* (*s*, *vs*, *ves*, *e*) =  
 (case *e* of  
 — *B-E*  
 — *UNOPS*  
 \$(*Unop-i T-i32 iop*)  $\Rightarrow$   
 (case *ves* of  
 (*ConstInt32* *c*)#*ves'*  $\Rightarrow$   
 (*s*, *vs*, *RSNormal* (*vs-to-es* ((*ConstInt32* (*app-unop-i iop* *c*))#*ves'*)))  
 | -  $\Rightarrow$  (*s*, *vs*, *crash-error*))

$\mid \$(\text{Unop-i } T\text{-i64 } iop) \Rightarrow$   
 (case ves of  
    $(\text{ConstInt64 } c)\#ves' \Rightarrow$   
      $(s, vs, \text{RSNormal } (vs\text{-to-es } ((\text{ConstInt64 } (\text{app-unop-i } iop \ c))\#ves')))$   
    $\mid - \Rightarrow (s, vs, \text{crash-error})$ )  
 $\mid \$(\text{Unop-i } - \ iop) \Rightarrow (s, vs, \text{crash-error})$   
 $\mid \$(\text{Unop-f } T\text{-f32 } fop) \Rightarrow$   
 (case ves of  
    $(\text{ConstFloat32 } c)\#ves' \Rightarrow$   
      $(s, vs, \text{RSNormal } (vs\text{-to-es } ((\text{ConstFloat32 } (\text{app-unop-f } fop \ c))\#ves')))$   
    $\mid - \Rightarrow (s, vs, \text{crash-error})$ )  
 $\mid \$(\text{Unop-f } T\text{-f64 } fop) \Rightarrow$   
 (case ves of  
    $(\text{ConstFloat64 } c)\#ves' \Rightarrow$   
      $(s, vs, \text{RSNormal } (vs\text{-to-es } ((\text{ConstFloat64 } (\text{app-unop-f } fop \ c))\#ves')))$   
    $\mid - \Rightarrow (s, vs, \text{crash-error})$ )  
 $\mid \$(\text{Unop-f } - \ fop) \Rightarrow (s, vs, \text{crash-error})$   
 — BINOPS  
 $\mid \$(\text{Binop-i } T\text{-i32 } iop) \Rightarrow$   
 (case ves of  
    $(\text{ConstInt32 } c2)\#(\text{ConstInt32 } c1)\#ves' \Rightarrow$   
      $\text{expect } (\text{app-binop-i } iop \ c1 \ c2) \ (\lambda c. (s, vs, \text{RSNormal } (vs\text{-to-es } ((\text{ConstInt32 } c)\#ves')))) \ (s, vs, \text{RSNormal } ((vs\text{-to-es } ves')@[Trap]))$   
    $\mid - \Rightarrow (s, vs, \text{crash-error})$ )  
 $\mid \$(\text{Binop-i } T\text{-i64 } iop) \Rightarrow$   
 (case ves of  
    $(\text{ConstInt64 } c2)\#(\text{ConstInt64 } c1)\#ves' \Rightarrow$   
      $\text{expect } (\text{app-binop-i } iop \ c1 \ c2) \ (\lambda c. (s, vs, \text{RSNormal } (vs\text{-to-es } ((\text{ConstInt64 } c)\#ves')))) \ (s, vs, \text{RSNormal } ((vs\text{-to-es } ves')@[Trap]))$   
    $\mid - \Rightarrow (s, vs, \text{crash-error})$ )  
 $\mid \$(\text{Binop-i } - \ iop) \Rightarrow (s, vs, \text{crash-error})$   
 $\mid \$(\text{Binop-f } T\text{-f32 } fop) \Rightarrow$   
 (case ves of  
    $(\text{ConstFloat32 } c2)\#(\text{ConstFloat32 } c1)\#ves' \Rightarrow$   
      $\text{expect } (\text{app-binop-f } fop \ c1 \ c2) \ (\lambda c. (s, vs, \text{RSNormal } (vs\text{-to-es } ((\text{ConstFloat32 } c)\#ves')))) \ (s, vs, \text{RSNormal } ((vs\text{-to-es } ves')@[Trap]))$   
    $\mid - \Rightarrow (s, vs, \text{crash-error})$ )  
 $\mid \$(\text{Binop-f } T\text{-f64 } fop) \Rightarrow$   
 (case ves of  
    $(\text{ConstFloat64 } c2)\#(\text{ConstFloat64 } c1)\#ves' \Rightarrow$   
      $\text{expect } (\text{app-binop-f } fop \ c1 \ c2) \ (\lambda c. (s, vs, \text{RSNormal } (vs\text{-to-es } ((\text{ConstFloat64 } c)\#ves')))) \ (s, vs, \text{RSNormal } ((vs\text{-to-es } ves')@[Trap]))$   
    $\mid - \Rightarrow (s, vs, \text{crash-error})$ )  
 $\mid \$(\text{Binop-f } - \ fop) \Rightarrow (s, vs, \text{crash-error})$   
 — TESTOPS  
 $\mid \$(\text{Testop } T\text{-i32 } testop) \Rightarrow$   
 (case ves of  
    $(\text{ConstInt32 } c)\#ves' \Rightarrow$   
      $(s, vs, \text{RSNormal } (vs\text{-to-es } ((\text{ConstInt32 } (\text{wasm-bool } (\text{app-testop-i } testop$

```

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 (cvt 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))
— 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
  | $C v ⇒ (s, vs, crash-error)
— E
— CALLCL
  | Calcl cl ⇒
    (case cl of
      Func-native i' (t1s -> t2s) ts es ⇒
        let n = length t1s in
        let m = length t2s in
        if length ves ≥ n
        then
          let (ves', ves'') = split-n ves n in
          let zs = n-zeros ts in
          (s, vs, RSNormal ((vs-to-es ves'') @ ([Local m i' ((rev ves')@zs)
[$(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))

```

*<proof>*

### termination

*<proof>*

```

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
          RSNormal es' ⇒ run-v n d i (s',vs',es')
          | RSCrash error ⇒ (s, RCrash error)

```

|  $run\text{-}v\ 0\ d\ i\ (s,vs,es) = (s, RCrash\ CExhaustion)$  |  $- \Rightarrow (s, RCrash\ CError)))$

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* ( $\$ \$ * vs$ )  
 $\langle proof \rangle$

**lemma** *not-const-vs-to-es-list:*  
**assumes**  $\sim (is\text{-}const\ e)$   
**shows**  $vs1\ @\ [e]\ @\ vs2 \neq \$ \$ * vs$   
 $\langle proof \rangle$

**lemma** *neq-label-nested:* $[Label\ n\ les\ es] \neq es$   
 $\langle proof \rangle$

**lemma** *neq-local-nested:* $[Local\ n\ i\ lvs\ es] \neq es$   
 $\langle proof \rangle$

**lemma** *trap-not-value:* $[Trap] \neq \$ \$ * es$   
 $\langle proof \rangle$

**thm** *Lfilled.simps* $[of\ -\ -\ [e],\ simplified]$

**lemma** *lfilled-single:*  
**assumes**  $Lfilled\ k\ lholed\ es\ [e]$   
 $\bigwedge a\ b\ c.\ e \neq Label\ a\ b\ c$   
**shows**  $(es = [e] \wedge lholed = LBase\ []\ []) \vee es = []$   
 $\langle proof \rangle$

**lemma** *lfilled-eq:*  
**assumes**  $Lfilled\ j\ lholed\ es\ LI$   
 $Lfilled\ j\ lholed\ es'\ LI$   
**shows**  $es = es'$   
 $\langle proof \rangle$

**lemma** *lfilled-size:*  
**assumes**  $Lfilled\ j\ lholed\ es\ LI$   
**shows**  $size\text{-}list\ size\ LI \geq size\text{-}list\ size\ es$   
 $\langle proof \rangle$

**thm** *Lfilled.simps* $[of\ -\ -\ es\ es,\ simplified]$

**lemma** *reduce-simple-not-eq:*

**assumes**  $(\text{es}) \rightsquigarrow (\text{es}')$   
**shows**  $\text{es} \neq \text{es}'$   
 $\langle \text{proof} \rangle$

**lemma** *reduce-not-eq*:  
**assumes**  $(s; \text{vs}; \text{es}) \rightsquigarrow\text{-}i (s'; \text{vs}'; \text{es}')$   
**shows**  $\text{es} \neq \text{es}'$   
 $\langle \text{proof} \rangle$

**lemma** *reduce-simple-not-value*:  
**assumes**  $(\text{es}) \rightsquigarrow (\text{es}')$   
**shows**  $\text{es} \neq \text{\$}\$* \text{vs}$   
 $\langle \text{proof} \rangle$

**lemma** *reduce-not-value*:  
**assumes**  $(s; \text{vs}; \text{es}) \rightsquigarrow\text{-}i (s'; \text{vs}'; \text{es}')$   
**shows**  $\text{es} \neq \text{\$}\$* \text{ves}$   
 $\langle \text{proof} \rangle$

**lemma** *reduce-simple-not-nil*:  
**assumes**  $(\text{es}) \rightsquigarrow (\text{es}')$   
**shows**  $\text{es} \neq []$   
 $\langle \text{proof} \rangle$

**lemma** *reduce-not-nil*:  
**assumes**  $(s; \text{vs}; \text{es}) \rightsquigarrow\text{-}i (s'; \text{vs}'; \text{es}')$   
**shows**  $\text{es} \neq []$   
 $\langle \text{proof} \rangle$

**lemma** *reduce-simple-not-trap*:  
**assumes**  $(\text{es}) \rightsquigarrow (\text{es}')$   
**shows**  $\text{es} \neq [\text{Trap}]$   
 $\langle \text{proof} \rangle$

**lemma** *reduce-not-trap*:  
**assumes**  $(s; \text{vs}; \text{es}) \rightsquigarrow\text{-}i (s'; \text{vs}'; \text{es}')$   
**shows**  $\text{es} \neq [\text{Trap}]$   
 $\langle \text{proof} \rangle$

**lemma** *reduce-simple-call*:  $\neg([\text{\$ Call } j]) \rightsquigarrow (\text{es}')$   
 $\langle \text{proof} \rangle$

**lemma** *reduce-call*:  
**assumes**  $(s; \text{vs}; [\text{\$ Call } j]) \rightsquigarrow\text{-}i (s'; \text{vs}'; \text{es}')$   
**shows**  $s = s'$   
 $\text{vs} = \text{vs}'$   
 $\text{es}' = [\text{Callcl } (\text{sfunc } s \ i \ j)]$   
 $\langle \text{proof} \rangle$

**lemma** *run-one-step-basic-unreachable-result*:  
**assumes** *run-one-step d i (s,vs,ves,\$Unreachable) = (s', vs', res)*  
**shows**  $\exists r. res = RSNormal\ r$   
 $\langle proof \rangle$

**lemma** *run-one-step-basic-nop-result*:  
**assumes** *run-one-step d i (s,vs,ves,\$Nop) = (s', vs', res)*  
**shows**  $\exists r. res = RSNormal\ r$   
 $\langle proof \rangle$

**lemma** *run-one-step-basic-drop-result*:  
**assumes** *run-one-step d i (s,vs,ves,\$Drop) = (s', vs', res)*  
**shows**  $(\exists r. res = RSNormal\ r) \vee (\exists e. res = RSCrash\ e)$   
 $\langle proof \rangle$

**lemma** *run-one-step-basic-select-result*:  
**assumes** *run-one-step d i (s,vs,ves,\$Select) = (s', vs', res)*  
**shows**  $(\exists r. res = RSNormal\ r) \vee (\exists e. res = RSCrash\ e)$   
 $\langle proof \rangle$

**lemma** *run-one-step-basic-block-result*:  
**assumes** *run-one-step d i (s,vs,ves,\$(Block x51 x52)) = (s', vs', res)*  
**shows**  $(\exists r. res = RSNormal\ r) \vee (\exists e. res = RSCrash\ e)$   
 $\langle proof \rangle$

**lemma** *run-one-step-basic-loop-result*:  
**assumes** *run-one-step d i (s,vs,ves,\$(Loop x61 x62)) = (s', vs', res)*  
**shows**  $(\exists r. res = RSNormal\ r) \vee (\exists e. res = RSCrash\ e)$   
 $\langle proof \rangle$

**lemma** *run-one-step-basic-if-result*:  
**assumes** *run-one-step d i (s,vs,ves,\$(If x71 x72 x73)) = (s', vs', res)*  
**shows**  $(\exists r. res = RSNormal\ r) \vee (\exists e. res = RSCrash\ e)$   
 $\langle proof \rangle$

**lemma** *run-one-step-basic-br-result*:  
**assumes** *run-one-step d i (s,vs,ves,\$Br x8) = (s', vs', res)*  
**shows**  $\exists r\ vrs. res = RSBreak\ r\ vrs$   
 $\langle proof \rangle$

**lemma** *run-one-step-basic-br-if-result*:  
**assumes** *run-one-step d i (s,vs,ves,\$Br-if x9) = (s', vs', res)*  
**shows**  $(\exists r. res = RSNormal\ r) \vee (\exists e. res = RSCrash\ e)$   
 $\langle proof \rangle$

**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**  $(\exists r. res = RSNormal\ r) \vee (\exists e. res = RSCrash\ e)$   
 $\langle proof \rangle$

**lemma** *run-one-step-basic-return-result:*

**assumes** *run-one-step d i (s,vs,ves,\$Return) = (s', vs', res)*  
**shows**  $\exists vrs. res = RSReturn vrs$   
*<proof>*

**lemma** *run-one-step-basic-call-result:*

**assumes** *run-one-step d i (s,vs,ves,\$Call x12) = (s', vs', res)*  
**shows**  $\exists r. res = RSNormal r$   
*<proof>*

**lemma** *run-one-step-basic-call-indirect-result:*

**assumes** *run-one-step d i (s,vs,ves,\$Call-indirect x13) = (s', vs', res)*  
**shows**  $(\exists r. res = RSNormal r) \vee (\exists e. res = RSCrash e)$   
*<proof>*

**lemma** *run-one-step-basic-get-local-result:*

**assumes** *run-one-step d i (s,vs,ves,\$Get-local x14) = (s', vs', res)*  
**shows**  $(\exists r. res = RSNormal r) \vee (\exists e. res = RSCrash e)$   
*<proof>*

**lemma** *run-one-step-basic-set-local-result:*

**assumes** *run-one-step d i (s,vs,ves,\$Set-local x15) = (s', vs', res)*  
**shows**  $(\exists r. res = RSNormal r) \vee (\exists e. res = RSCrash e)$   
*<proof>*

**lemma** *run-one-step-basic-tee-local-result:*

**assumes** *run-one-step d i (s,vs,ves,\$Tee-local x16) = (s', vs', res)*  
**shows**  $(\exists r. res = RSNormal r) \vee (\exists e. res = RSCrash e)$   
*<proof>*

**lemma** *run-one-step-basic-get-global-result:*

**assumes** *run-one-step d i (s,vs,ves,\$Get-global x17) = (s', vs', res)*  
**shows**  $(\exists r. res = RSNormal r) \vee (\exists e. res = RSCrash e)$   
*<proof>*

**lemma** *run-one-step-basic-set-global-result:*

**assumes** *run-one-step d i (s,vs,ves,\$Set-global x18) = (s', vs', res)*  
**shows**  $(\exists r. res = RSNormal r) \vee (\exists e. res = RSCrash e)$   
*<proof>*

**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>*

**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>*

**lemma** *run-one-step-basic-current-memory-result:*

**assumes** *run-one-step d i (s,vs,ves,\$Current-memory) = (s', vs', res)*

**shows**  $(\exists r. \text{res} = \text{RSNormal } r) \vee (\exists e. \text{res} = \text{RSCrash } e)$

*<proof>*

**lemma** *run-one-step-basic-grow-memory-result:*

**assumes** *run-one-step d i (s,vs,ves,\$Grow-memory) = (s', vs', res)*

**shows**  $(\exists r. \text{res} = \text{RSNormal } r) \vee (\exists e. \text{res} = \text{RSCrash } e)$

*<proof>*

**lemma** *run-one-step-basic-const-result:*

**assumes** *run-one-step d i (s,vs,ves,\$EConst x23) = (s', vs', res)*

**shows**  $\exists e. \text{res} = \text{RSCrash } e$

*<proof>*

**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**  $(\exists r. \text{res} = \text{RSNormal } r) \vee (\exists e. \text{res} = \text{RSCrash } e)$

*<proof>*

**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**  $(\exists r. \text{res} = \text{RSNormal } r) \vee (\exists e. \text{res} = \text{RSCrash } e)$

*<proof>*

**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**  $(\exists r. \text{res} = \text{RSNormal } r) \vee (\exists e. \text{res} = \text{RSCrash } e)$

*<proof>*

**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**  $(\exists r. \text{res} = \text{RSNormal } r) \vee (\exists e. \text{res} = \text{RSCrash } e)$

*<proof>*

**lemma** *run-one-step-basic-testop-result:*

**assumes** *run-one-step d i (s,vs,ves,\$Testop x281 x282) = (s', vs', res)*

**shows**  $(\exists r. \text{res} = \text{RSNormal } r) \vee (\exists e. \text{res} = \text{RSCrash } e)$

*<proof>*

**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**  $(\exists r. \text{res} = \text{RSNormal } r) \vee (\exists e. \text{res} = \text{RSCrash } e)$

*<proof>*

**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**  $(\exists r. \text{res} = \text{RSNormal } r) \vee (\exists e. \text{res} = \text{RSCrash } e)$   
<proof>

**lemma** *run-one-step-basic-cvtop-result:*

**assumes**  $\text{run-one-step } d \ i \ (s, vs, ves, \$Cvtop \ t2 \ x312 \ t1 \ sx) = (s', vs', \text{res})$   
**shows**  $(\exists r. \text{res} = \text{RSNormal } r) \vee (\exists e. \text{res} = \text{RSCrash } e)$   
<proof>

**lemma** *run-one-step-trap-result:*

**assumes**  $\text{run-one-step } d \ i \ (s, vs, ves, \text{Trap}) = (s', vs', \text{res})$   
**shows**  $\exists e. \text{res} = \text{RSCrash } e$   
<proof>

**lemma** *run-one-step-callcl-result:*

**assumes**  $\text{run-one-step } d \ i \ (s, vs, ves, \text{Callcl } cl) = (s', vs', \text{res})$   
**shows**  $(\exists r. \text{res} = \text{RSNormal } r) \vee (\exists e. \text{res} = \text{RSCrash } e)$   
<proof>

**lemma** *run-one-step-label-result:*

**assumes**  $\text{run-one-step } d \ i \ (s, vs, ves, \text{Label } x41 \ x42 \ x43) = (s', vs', \text{res})$   
**shows**  $(\exists r. \text{res} = \text{RSNormal } r) \vee (\exists r \ rvs. \text{res} = \text{RSBreak } r \ rvs) \vee (\exists rvs. \text{res} = \text{RSReturn } rvs) \vee (\exists e. \text{res} = \text{RSCrash } e)$   
<proof>

**lemma** *run-one-step-local-result:*

**assumes**  $\text{run-one-step } d \ i \ (s, vs, ves, \text{Local } x51 \ x52 \ x53 \ x54) = (s', vs', \text{res})$   
**shows**  $(\exists r. \text{res} = \text{RSNormal } r) \vee (\exists e. \text{res} = \text{RSCrash } e)$   
<proof>

**lemma** *run-one-step-break:*

**assumes**  $\text{run-one-step } d \ i \ (s, vs, ves, e) = (s', vs', \text{RSBreak } n \ \text{res})$   
**shows**  $(e = \$Br \ n) \vee (\exists n \ \text{les } es. e = \text{Label } n \ \text{les } es)$   
<proof>

**lemma** *run-one-step-return:*

**assumes**  $\text{run-one-step } d \ i \ (s, vs, ves, e) = (s', vs', \text{RSReturn } \text{res})$   
**shows**  $(e = \$Return) \vee (\exists n \ \text{les } es. e = \text{Label } n \ \text{les } es)$   
<proof>

**lemma** *run-step-break-imp-not-trap-const-list:*

**assumes**  $\text{run-step } d \ i \ (s, vs, es) = (s', vs', \text{RSBreak } n \ \text{res})$   
**shows**  $es \neq [\text{Trap}] \neg\text{const-list } es$   
<proof>

**lemma** *run-step-return-imp-not-trap-const-list:*

**assumes**  $\text{run-step } d \ i \ (s, vs, es) = (s', vs', \text{RSReturn } \text{res})$   
**shows**  $es \neq [\text{Trap}] \neg\text{const-list } es$   
<proof>

**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**  $run-step\ d\ i\ (s, vs, es) = (s', vs', RSBreak\ (Suc\ n)\ res)$

*<proof>*

**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**  $run-step\ d\ i\ (s, vs, es) = (s', vs', RSReturn\ res)$

*<proof>*

**thm** *run-step-run-one-step.induct*

**definition** *run-step-break-imp-lfilled-prop where*

*run-step-break-imp-lfilled-prop*  $s'\ vs'\ n\ res =$

$(\lambda d\ i\ (s,vs,es). (run-step\ d\ i\ (s,vs,es) = (s', vs', RSBreak\ n\ res)) \longrightarrow$

$s = s' \wedge vs = vs' \wedge$

$(\exists n'\ lfilled\ es-c. n' \geq n \wedge Lfilled-exact\ (n'-n)\ lfilled\ ((vs-to-es\ res) @ [\$Br\ n'] @ es-c)\ es))$

**definition** *run-one-step-break-imp-lfilled-prop where*

*run-one-step-break-imp-lfilled-prop*  $s'\ vs'\ n\ res =$

$(\lambda d\ i\ (s,vs,ves,e). run-one-step\ d\ i\ (s,vs,ves,e) = (s', vs', RSBreak\ n\ res) \longrightarrow$

$s = s' \wedge vs = vs' \wedge ((res = ves \wedge e = \$Br\ n) \vee (\exists n'\ lfilled\ es-c\ es\ les'\ ln.$

$n' > n \wedge Lfilled-exact\ (n'-(n+1))\ lfilled\ ((vs-to-es\ res) @ [\$Br\ n'] @ es-c)\ es \wedge e = Label\ ln\ les'\ es)))$

**lemma** *run-step-break-imp-lfilled:*

**assumes**  $run-step\ d\ i\ (s,vs,es) = (s', vs', RSBreak\ n\ res)$

**shows**  $s = s' \wedge$

$vs = vs' \wedge$

$(\exists n'\ lfilled\ es-c. n' \geq n \wedge$

$Lfilled-exact\ (n'-n)\ lfilled\ ((vs-to-es\ res) @ [\$Br\ n'] @ es-c)$

$es)$

*<proof>*

**lemma** *run-step-return-imp-lfilled:*

**assumes**  $run-step\ d\ i\ (s,vs,es) = (s', vs', RSReturn\ res)$

**shows**  $s = s' \wedge vs = vs' \wedge (\exists n\ lfilled\ es-c. Lfilled-exact\ n\ lfilled\ ((vs-to-es\ res)$

$@ [\$Return] @ es-c)\ es)$

*<proof>*

**lemma** *run-step-basic-unop-testop-sound:*

**assumes**  $(run-one-step\ d\ i\ (s,vs,ves,\$b-e) = (s', vs', RSNormal\ es'))$

$b-e = Unop-i\ t\ iop \vee b-e = Unop-f\ t\ fop \vee b-e = Testop\ t\ testop$

**shows**  $(\downarrow s;vs;(vs-to-es\ ves)@[\$b-e]) \rightsquigarrow - i\ (\downarrow s';vs';es')$

*<proof>*

**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**  $(s;vs;(vs\text{-to-}es\ ves)@[b-e]) \rightsquigarrow\text{-} i (s';vs';es')$   
 $\langle\text{proof}\rangle$

**lemma** *run-step-basic-sound*:

**assumes**  $(\text{run-one-step } d\ i (s,vs,ves,[b-e]) = (s', vs', \text{RSNormal } es'))$

**shows**  $(s;vs;(vs\text{-to-}es\ ves)@[b-e]) \rightsquigarrow\text{-} i (s';vs';es')$   
 $\langle\text{proof}\rangle$

**theorem** *run-step-sound*:

**assumes**  $\text{run-step } d\ i (s,vs,es) = (s', vs', \text{RSNormal } es')$

**shows**  $(s;vs;es) \rightsquigarrow\text{-} i (s';vs';es')$

$\langle\text{proof}\rangle$

**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.