WebAssembly

Conrad Watt

June 16, 2019

Abstract

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

Contents

1 WebAssembly Core AST

theory *Wasm-Ast* **imports** *Main Native-Word.Uint8* **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* ⇒ *nat* ⇒ *bytes* ⇒ *bytes* **is** (λ*a n as*. *takefill* (*Abs-uint8 a*) *n as*) ⟨*proof*⟩
lift-definition *bytes-replicate* :: *nat* ⇒ *byte* ⇒ *bytes* **is** (λ*n b*. *replicate n* (*Abs-uint8 b*)) ⟨*proof*⟩
definition *msbyte* :: *bytes* ⇒ *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* ⇒ *nat* ⇒ *nat* ⇒ *bytes* **is** (λ*m n l*. *take l* (*drop n m*)) ⟨*proof*⟩
lift-definition *write-bytes* :: *mem* ⇒ *nat* ⇒ *bytes* ⇒ *mem* **is** (λ*m n bs*. (*take n m*) @ *bs* @ (*drop* (*n* + *length bs*) *m*)) ⟨*proof*⟩
lift-definition *mem-append* :: *mem* ⇒ *bytes* ⇒ *mem* **is** *append* ⟨*proof*⟩
typedecl *host*
typedecl *host-state*

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

datatype — packed types

tp = *Tp-i8* | *Tp-i16* | *Tp-i32*

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

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

datatype — function types
tf = *Tf t list t list* (- '-> - 60)

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

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

datatype
sx = *S* | *U*

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

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

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

datatype
binop-f = *Addf* | *Subf* | *Mulf* | *Divf* | *Min* | *Max* | *Copysign*

datatype
testop = *Eqz*

datatype
relop-i = *Eq* | *Ne* | *Lt sx* | *Gt sx* | *Le sx* | *Ge sx*

datatype
relop-f = *Eqf* | *Nef* | *Ltf* | *Gtf* | *Lef* | *Gef*

datatype
cvtop = *Convert* | *Reinterpret*

datatype — values
v =
ConstInt32 i32
| *ConstInt64 i64*
| *ConstFloat32 f32*
| *ConstFloat64 f64*

datatype — basic instructions
b-e =
Unreachable
| *Nop*
| *Drop*
| *Select*
| *Block tf b-e list*
| *Loop tf b-e list*
| *If tf b-e list b-e list*
| *Br i*
| *Br-if i*
| *Br-table i list i*
| *Return*
| *Call i*
| *Call-indirect i*
| *Get-local i*
| *Set-local i*
| *Tee-local i*
| *Get-global i*
| *Set-global i*
| *Load t* (*tp* × *sx*) *option a off*
| *Store t tp option a off*
| *Current-memory*
| *Grow-memory*
| *EConst v* (*C - 60*)
| *Unop-i t unop-i*
| *Unop-f t unop-f*
| *Binop-i t binop-i*
| *Binop-f t binop-f*
| *Testop t testop*
| *Relop-i t relop-i*
| *Relop-f t relop-f*
| *Cvtop t cvtop t sx option*

datatype *cl* = — function closures
  Func-native i tf t list b-e list
| *Func-host tf host*

record *inst* = — instances
  types :: *tf list*
  funcs :: *i list*
  tab :: *i option*
  mem :: *i option*
  globs :: *i list*

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

record *global* =
  g-mut :: *mut*
  g-val :: *v*

record *s* = — store
  inst :: *inst list*
  funcs :: *cl list*
  tab :: *tabinst list*
  mem :: *mem list*
  globs :: *global list*

datatype *e* = — administrative instruction
  Basic b-e ($- *60*)
  | *Trap*
  | *Callcl cl*
  | *Label nat e list e list*
  | *Local nat i v list e list*

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

2 Syntactic Typeclasses

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

class *wasm-base* = *zero*

class *wasm-int* = *wasm-base* +

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

```
  fixes int-popcnt :: 'a ⇒ 'a

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

  fixes int-eqz :: 'a ⇒ bool

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

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

class wasm-float = wasm-base +

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

  fixes float-add :: 'a ⇒ 'a ⇒ 'a
```

fixes *float-sub* :: $'a \Rightarrow 'a \Rightarrow 'a$
fixes *float-mul* :: $'a \Rightarrow 'a \Rightarrow 'a$
fixes *float-div* :: $'a \Rightarrow 'a \Rightarrow 'a$
fixes *float-min* :: $'a \Rightarrow 'a \Rightarrow 'a$
fixes *float-max* :: $'a \Rightarrow 'a \Rightarrow 'a$
fixes *float-copysign* :: $'a \Rightarrow 'a \Rightarrow 'a$

fixes *float-eq* :: $'a \Rightarrow 'a \Rightarrow bool$
fixes *float-lt* :: $'a \Rightarrow 'a \Rightarrow bool$
fixes *float-gt* :: $'a \Rightarrow 'a \Rightarrow bool$
fixes *float-le* :: $'a \Rightarrow 'a \Rightarrow bool$
fixes *float-ge* :: $'a \Rightarrow 'a \Rightarrow bool$
begin
abbreviation (*input*)
float-ne **where**
float-ne x y $\equiv \neg$ (*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-replicate* (*n* ∗ *64000*) *0*)

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

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

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

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

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

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

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

definition *typeof* :: $v \Rightarrow t$ **where**
typeof v = (*case v of*
ConstInt32 - ⇒ T-i32
| *ConstInt64 - ⇒ T-i64*
| *ConstFloat32 - ⇒ T-f32*
| *ConstFloat64 - ⇒ 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 ⇒ 4
| *T-i64 ⇒ 8*
| *T-f32 ⇒ 4*
| *T-f64 ⇒ 8*)

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

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

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

definition *is-mut* :: *tg* $\Rightarrow$ *bool* **where**
is-mut tg = (*tg-mut tg* = *T-mut*)

definition *app-unop-i* :: *unop-i* $\Rightarrow$ $'i$::*wasm-int* $\Rightarrow$ $'i$::*wasm-int* **where**
app-unop-i iop c =
(*case iop of*
Ctz ⇒ int-ctz c
| *Clz ⇒ int-clz c*

```
  | Popcnt ⇒ int-popcnt c)

definition app-unop-f :: unop-f ⇒ 'f::wasm-float ⇒ 'f::wasm-float where
  app-unop-f fop c =
              (case fop of
                 Neg ⇒ float-neg c
               | Abs ⇒ float-abs c
               | Ceil ⇒ float-ceil c
               | Floor ⇒ float-floor c
               | Trunc ⇒ float-trunc c
               | Nearest ⇒ float-nearest c
               | Sqrt ⇒ float-sqrt c)

definition app-binop-i :: binop-i ⇒ 'i::wasm-int ⇒ 'i::wasm-int ⇒ ('i::wasm-int)
option where
  app-binop-i iop c1 c2 = (case iop of
                            Add ⇒ Some (int-add c1 c2)
                          | Sub ⇒ Some (int-sub c1 c2)
                          | Mul ⇒ Some (int-mul c1 c2)
                          | Div U ⇒ int-div-u c1 c2
                          | Div S ⇒ int-div-s c1 c2
                          | Rem U ⇒ int-rem-u c1 c2
                          | Rem S ⇒ int-rem-s c1 c2
                          | And ⇒ Some (int-and c1 c2)
                          | Or ⇒ Some (int-or c1 c2)
                          | Xor ⇒ Some (int-xor c1 c2)
                          | Shl ⇒ Some (int-shl c1 c2)
                          | Shr U ⇒ Some (int-shr-u c1 c2)
                          | Shr S ⇒ Some (int-shr-s c1 c2)
                          | Rotl ⇒ Some (int-rotl c1 c2)
                          | Rotr ⇒ Some (int-rotr c1 c2))

definition app-binop-f :: binop-f ⇒ 'f::wasm-float ⇒ 'f::wasm-float ⇒ ('f::wasm-float)
option where
  app-binop-f fop c1 c2 = (case fop of
                            Addf ⇒ Some (float-add c1 c2)
                          | Subf ⇒ Some (float-sub c1 c2)
                          | Mulf ⇒ Some (float-mul c1 c2)
                          | Divf ⇒ Some (float-div c1 c2)
                          | Min ⇒ Some (float-min c1 c2)
                          | Max ⇒ Some (float-max c1 c2)
                          | Copysign ⇒ Some (float-copysign c1 c2))

definition app-testop-i :: testop ⇒ 'i::wasm-int ⇒ bool where
  app-testop-i testop c = (case testop of Eqz ⇒ int-eqz c)

definition app-relop-i :: relop-i ⇒ 'i::wasm-int ⇒ 'i::wasm-int ⇒ bool where
  app-relop-i rop c1 c2 = (case rop of
                            Eq ⇒ int-eq c1 c2
```

```
| Ne ⇒ int-ne c1 c2
| Lt U ⇒ int-lt-u c1 c2
| Lt S ⇒ int-lt-s c1 c2
| Gt U ⇒ int-gt-u c1 c2
| Gt S ⇒ int-gt-s c1 c2
| Le U ⇒ int-le-u c1 c2
| Le S ⇒ int-le-s c1 c2
| Ge U ⇒ int-ge-u c1 c2
| Ge S ⇒ int-ge-s c1 c2)
```

definition *app-relop-f* :: *relop-f* ⇒ *'f*::*wasm-float* ⇒ *'f*::*wasm-float* ⇒ *bool* **where**
app-relop-f rop c1 c2 = (*case rop of*

```
  Eqf ⇒ float-eq c1 c2
| Nef ⇒ float-ne c1 c2
| Ltf ⇒ float-lt c1 c2
| Gtf ⇒ float-gt c1 c2
| Lef ⇒ float-le c1 c2
| Gef ⇒ float-ge c1 c2)
```

definition *types-agree* :: *t* ⇒ *v* ⇒ *bool* **where**
types-agree t v = (*typeof v* = *t*)

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

definition *rglob-is-mut* :: *global* ⇒ *bool* **where**
rglob-is-mut g = (*g-mut g* = *T-mut*)

definition *stypes* :: *s* ⇒ *nat* ⇒ *nat* ⇒ *tf* **where**
stypes s i j = ((*types* ((*inst s*)!*i*))!*j*)

definition *sfunc-ind* :: *s* ⇒ *nat* ⇒ *nat* ⇒ *nat* **where**
sfunc-ind s i j = ((*inst.funcs* ((*inst s*)!*i*))!*j*)

definition *sfunc* :: *s* ⇒ *nat* ⇒ *nat* ⇒ *cl* **where**
sfunc s i j = (*funcs s*)!(*sfunc-ind s i j*)

definition *sglob-ind* :: *s* ⇒ *nat* ⇒ *nat* ⇒ *nat* **where**
sglob-ind s i j = ((*inst.globs* ((*inst s*)!*i*))!*j*)

definition *sglob* :: *s* ⇒ *nat* ⇒ *nat* ⇒ *global* **where**
sglob s i j = (*globs s*)!(*sglob-ind s i j*)

definition *sglob-val* :: *s* ⇒ *nat* ⇒ *nat* ⇒ *v* **where**
sglob-val s i j = *g-val* (*sglob s i j*)

definition *smem-ind* :: *s* ⇒ *nat* ⇒ *nat option* **where**
smem-ind s i = (*inst.mem* ((*inst s*)!*i*))

definition *stab-s* :: *s* ⇒ *nat* ⇒ *nat* ⇒ *cl option* **where**
stab-s s i j = (*let stabinst* = ((*tab s*)!*i*) *in* (*if* (*length* (*stabinst*) > *j*) *then* (*stabinst*!*j*) *else None*))

definition *stab* :: *s* ⇒ *nat* ⇒ *nat* ⇒ *cl option* **where**
stab s i j = (*case* (*inst.tab* ((*inst s*)!*i*)) *of Some k* => *stab-s s k j* | *None* => *None*)

definition *supdate-glob-s* :: *s* ⇒ *nat* ⇒ *v* ⇒ *s* **where**
supdate-glob-s s k v = *s*(|*globs* := (*globs s*)[*k*:=((*globs s*)!*k*)(|*g-val* := *v*|)]|)

definition *supdate-glob* :: *s* ⇒ *nat* ⇒ *nat* ⇒ *v* ⇒ *s* **where**
supdate-glob s i j v = (*let k* = *sglob-ind s i j in supdate-glob-s s k v*)

definition *is-const* :: *e* ⇒ *bool* **where**
is-const e = (*case e of Basic* (*C* -) ⇒ *True* | - ⇒ *False*)

definition *const-list* :: *e list* ⇒ *bool* **where**
const-list xs = *list-all is-const xs*

inductive *store-extension* :: *s* ⇒ *s* ⇒ *bool* **where**
⟦*insts* = *insts'*; *fs* = *fs'*; *tclss* = *tclss'*; *list-all2* (λ*bs bs'*. *mem-size bs* ≤ *mem-size bs'*) *bss bss'*; *gs* = *gs'*⟧ ⟹
store-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-e-list* :: *b-e list* ⇒ *e list* (*$∗ - 60*) **where**
to-e-list b-es ≡ *map Basic b-es*

abbreviation *v-to-e-list* :: *v list* ⇒ *e list* (*$$∗ - 60*) **where**
v-to-e-list ves ≡ *map* (λ*v*. *$C v*) *ves*

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

L0:⟦*const-list vs*; *lholed* = (*LBase vs es'*)⟧ ⟹ *Lfilled 0 lholed es* (*vs* @ *es* @ *es'*)

| *LN*:⟦*const-list vs*; *lholed* = (*LRec vs n es' l es''*); *Lfilled k l es lfilledk*⟧ ⟹ *Lfilled* (*k*+1) *lholed es* (*vs* @ [*Label n es' lfilledk*] @ *es''*)

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

L0:⟦*lholed* = (*LBase* [] [])⟧ ⟹ *Lfilled-exact 0 lholed es es*

| *LN*:⟦*const-list vs*; *lholed* = (*LRec vs n es' l es''*); *Lfilled-exact k l es lfilledk*⟧ ⟹ *Lfilled-exact* (*k*+1) *lholed es* (*vs* @ [*Label n es' lfilledk*] @ *es''*)

definition *load-store-t-bounds* :: *a* ⇒ *tp option* ⇒ *t* ⇒ *bool* **where**
load-store-t-bounds a tp t = (*case tp of*
None ⇒ *2^a* ≤ *t-length t*
| *Some tp* ⇒ *2^a* ≤ *tp-length tp* ∧ *tp-length tp* < *t-length t* ∧ *is-int-t t*)

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

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

definition *cvt-f32* :: *sx option* ⇒ *v* ⇒ *f32 option* **where**
cvt-f32 sx v = (*case v of*
ConstInt32 c ⇒ (*case sx of*
Some U ⇒ *Some* (*f32-convert-ui32 c*)
| *Some S* ⇒ *Some* (*f32-convert-si32 c*)
| - ⇒ *None*)
| *ConstInt64 c* ⇒ (*case sx of*
Some U ⇒ *Some* (*f32-convert-ui64 c*)
| *Some S* ⇒ *Some* (*f32-convert-si64 c*)
| - ⇒ *None*)
| *ConstFloat32 c* ⇒ *None*
| *ConstFloat64 c* ⇒ *Some* (*wasm-demote c*))

definition *cvt-f64* :: *sx option* ⇒ *v* ⇒ *f64 option* **where**
cvt-f64 sx v = (*case v of*
ConstInt32 c ⇒ (*case sx of*
Some U ⇒ *Some* (*f64-convert-ui32 c*)
| *Some S* ⇒ *Some* (*f64-convert-si32 c*)
| - ⇒ *None*)
| *ConstInt64 c* ⇒ (*case sx of*
Some U ⇒ *Some* (*f64-convert-ui64 c*)
| *Some S* ⇒ *Some* (*f64-convert-si64 c*)
| - ⇒ *None*)
| *ConstFloat32 c* ⇒ *Some* (*wasm-promote c*)
| *ConstFloat64 c* ⇒ *None*)

definition *cvt* :: *t* ⇒ *sx option* ⇒ *v* ⇒ *v option* **where**
cvt t sx v = (*case t of*
T-i32 ⇒ (*case* (*cvt-i32 sx v*) *of Some c* ⇒ *Some* (*ConstInt32 c*) |
None ⇒ *None*)
| *T-i64* ⇒ (*case* (*cvt-i64 sx v*) *of Some c* ⇒ *Some* (*ConstInt64 c*) |
None ⇒ *None*)
| *T-f32* ⇒ (*case* (*cvt-f32 sx v*) *of Some c* ⇒ *Some* (*ConstFloat32 c*) |
None ⇒ *None*)
| *T-f64* ⇒ (*case* (*cvt-f64 sx v*) *of Some c* ⇒ *Some* (*ConstFloat64 c*) |
None ⇒ *None*))

definition *bits* :: *v* ⇒ *bytes* **where**
bits v = (*case v of*
ConstInt32 c ⇒ (*serialise-i32 c*)
| *ConstInt64 c* ⇒ (*serialise-i64 c*)
| *ConstFloat32 c* ⇒ (*serialise-f32 c*)
| *ConstFloat64 c* ⇒ (*serialise-f64 c*))

definition *bitzero* :: *t* ⇒ *v* **where**
bitzero t = (*case t of*
T-i32 ⇒ *ConstInt32 0*
| *T-i64* ⇒ *ConstInt64 0*
| *T-f32* ⇒ *ConstFloat32 0*
| *T-f64* ⇒ *ConstFloat64 0*)

definition *n-zeros* :: *t list* ⇒ *v list* **where**
n-zeros ts = (*map* (λ*t*. *bitzero t*) *ts*)

lemma *is-int-t-exists*:
assumes *is-int-t t*
shows *t* = *T-i32* ∨ *t* = *T-i64*
⟨*proof*⟩

lemma *is-float-t-exists*:
assumes *is-float-t t*
shows *t* = *T-f32* ∨ *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 ∃ *k*. *inst.tab* ((*inst s*)!*i*) = *Some k* ∧ *length* ((*tab s*)!*k*) > *j* ∧((*tab s*)!*k*)!*j* = *Some cl*
⟨*proof*⟩

lemma *inj-basic*: *inj Basic*
⟨*proof*⟩

lemma *inj-basic-econst*: *inj* (λ*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*:∃ *ves*. ($$∗*vs*) = ($∗*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 ∃ *lholed′*. *Lfilled n lholed′ es LI*
⟨*proof*⟩

lemma *Lfilled-imp-exists-Lfilled-exact*:
assumes *Lfilled n lholed es LI*
shows ∃ *lholed′ ves es-c*. *const-list ves* ∧ *Lfilled-exact n lholed′* (*ves*@*es*@*es-c*) *LI*
⟨*proof*⟩

lemma *n-zeros-typeof*:
n-zeros ts = *vs* ⟹ (*ts* = *map typeof vs*)
⟨*proof*⟩

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

inductive *b-e-typing* :: [*t-context*, *b-e list*, *tf*] ⇒ *bool* (- ⊢ - : - *60*) **where**
— *num ops*
const:$\mathcal{C} \vdash [C\ v]$: ([] -> [(*typeof v*)])
| *unop-i*:*is-int-t t* ⟹ $\mathcal{C} \vdash$ [*Unop-i t* -] : ([*t*] -> [*t*])
| *unop-f*:*is-float-t t* ⟹ $\mathcal{C} \vdash$ [*Unop-f t* -] : ([*t*] -> [*t*])
| *binop-i*:*is-int-t t* ⟹ $\mathcal{C} \vdash$ [*Binop-i t iop*] : ([*t*,*t*] -> [*t*])
| *binop-f*:*is-float-t t* ⟹ $\mathcal{C} \vdash$ [*Binop-f t* -] : ([*t*,*t*] -> [*t*])
| *testop*:*is-int-t t* ⟹ $\mathcal{C} \vdash$ [*Testop t* -] : ([*t*] -> [*T-i32*])
| *relop-i*:*is-int-t t* ⟹ $\mathcal{C} \vdash$ [*Relop-i t* -] : ([*t*,*t*] -> [*T-i32*])
| *relop-f*:*is-float-t t* ⟹ $\mathcal{C} \vdash$ [*Relop-f t* -] : ([*t*,*t*] -> [*T-i32*])
— *convert*
| *convert*:⟦(*t1* ≠ *t2*); (*sx* = *None*) = ((*is-float-t t1* ∧ *is-float-t t2*) ∨ (*is-int-t t1* ∧ *is-int-t t2* ∧ (*t-length t1* < *t-length t2*)))⟧ ⟹ $\mathcal{C} \vdash$ [*Cvtop t1 Convert t2 sx*] : ([*t2*] -> [*t1*])
— *reinterpret*
| *reinterpret*:⟦(*t1* ≠ *t2*); *t-length t1* = *t-length t2*⟧ ⟹ $\mathcal{C} \vdash$ [*Cvtop t1 Reinterpret t2 None*] : ([*t2*] -> [*t1*])
— *unreachable*, *nop*, *drop*, *select*
| *unreachable*:$\mathcal{C} \vdash$ [*Unreachable*] : (*ts* -> *ts'*)
| *nop*:$\mathcal{C} \vdash$ [*Nop*] : ([] -> [])
| *drop*:$\mathcal{C} \vdash$ [*Drop*] : ([*t*] -> [])
| *select*:$\mathcal{C} \vdash$ [*Select*] : ([*t*,*t*,*T-i32*] -> [*t*])
— *block*
| *block*:⟦*tf* = (*tn* -> *tm*); $\mathcal{C}$(|*label* := ([*tm*] @ (*label* $\mathcal{C}$))|) ⊢ *es* : (*tn* -> *tm*)⟧ ⟹ $\mathcal{C}$ ⊢ [*Block tf es*] : (*tn* -> *tm*)
— *loop*
| *loop*:⟦*tf* = (*tn* -> *tm*); $\mathcal{C}$(|*label* := ([*tn*] @ (*label* $\mathcal{C}$))|) ⊢ *es* : (*tn* -> *tm*)⟧ ⟹ $\mathcal{C}$ ⊢ [*Loop tf es*] : (*tn* -> *tm*)
— *if then else*
| *if-wasm*:⟦*tf* = (*tn* -> *tm*); $\mathcal{C}$(|*label* := ([*tm*] @ (*label* $\mathcal{C}$))|) ⊢ *es1* : (*tn* -> *tm*); $\mathcal{C}$(|*label* := ([*tm*] @ (*label* $\mathcal{C}$))|) ⊢ *es2* : (*tn* -> *tm*)⟧ ⟹ $\mathcal{C} \vdash$ [*If tf es1 es2*] : (*tn* @ [*T-i32*] -> *tm*)
— *br*
| *br*:⟦*i* < *length*(*label* $\mathcal{C}$); (*label* $\mathcal{C}$)!*i* = *ts*⟧ ⟹ $\mathcal{C} \vdash$ [*Br i*] : (*t1s* @ *ts* -> *t2s*)
— *br-if*
| *br-if*:⟦*i* < *length*(*label* $\mathcal{C}$); (*label* $\mathcal{C}$)!*i* = *ts*⟧ ⟹ $\mathcal{C} \vdash$ [*Br-if i*] : (*ts* @ [*T-i32*] -> *ts*)
— *br-table*
| *br-table*:⟦*list-all* (λ*i*. *i* < *length*(*label* $\mathcal{C}$) ∧ (*label* $\mathcal{C}$)!*i* = *ts*) (*is*@[*i*])⟧ ⟹ $\mathcal{C} \vdash$ [*Br-table is i*] : (*t1s* @ *ts* @ [*T-i32*] -> *t2s*)
— *return*
| *return*:⟦(*return* $\mathcal{C}$) = *Some ts*⟧ ⟹ $\mathcal{C} \vdash$ [*Return*] : (*t1s* @ *ts* -> *t2s*)
— *call*
| *call*:⟦*i* < *length*(*func-t* $\mathcal{C}$); (*func-t* $\mathcal{C}$)!*i* = *tf*⟧ ⟹ $\mathcal{C} \vdash$ [*Call i*] : *tf*

— *call-indirect*
| *call-indirect*:⟦i < *length*(*types-t* $\mathcal{C}$); (*types-t* $\mathcal{C}$)!i = (*t1s* -> *t2s*); (*table* $\mathcal{C}$) ≠ *None*⟧ ⟹ $\mathcal{C}$ ⊢ [*Call-indirect* i] : (*t1s* @ [*T-i32*] -> *t2s*)
— *get-local*
| *get-local*:⟦i < *length*(*local* $\mathcal{C}$); (*local* $\mathcal{C}$)!i = t⟧ ⟹ $\mathcal{C}$ ⊢ [*Get-local* i] : ([] -> [t])
— *set-local*
| *set-local*:⟦i < *length*(*local* $\mathcal{C}$); (*local* $\mathcal{C}$)!i = t⟧ ⟹ $\mathcal{C}$ ⊢ [*Set-local* i] : ([t] -> [])
— *tee-local*
| *tee-local*:⟦i < *length*(*local* $\mathcal{C}$); (*local* $\mathcal{C}$)!i = t⟧ ⟹ $\mathcal{C}$ ⊢ [*Tee-local* i] : ([t] -> [t])
— *get-global*
| *get-global*:⟦i < *length*(*global* $\mathcal{C}$); *tg-t* ((*global* $\mathcal{C}$)!i) = t⟧ ⟹ $\mathcal{C}$ ⊢ [*Get-global* i] : ([] -> [t])
— *set-global*
| *set-global*:⟦i < *length*(*global* $\mathcal{C}$); *tg-t* ((*global* $\mathcal{C}$)!i) = t; *is-mut* ((*global* $\mathcal{C}$)!i)⟧ ⟹ $\mathcal{C}$ ⊢ [*Set-global* i] : ([t] -> [])
— *load*
| *load*:⟦(*memory* $\mathcal{C}$) = *Some* n; *load-store-t-bounds* a (*option-projl* *tp-sx*) t⟧ ⟹ $\mathcal{C}$ ⊢ [*Load* t *tp-sx* a *off*] : ([*T-i32*] -> [t])
— *store*
| *store*:⟦(*memory* $\mathcal{C}$) = *Some* n; *load-store-t-bounds* a *tp* t⟧ ⟹ $\mathcal{C}$ ⊢ [*Store* t *tp* a *off*] : ([*T-i32*,t] -> [])
— *current-memory*
| *current-memory*:(*memory* $\mathcal{C}$) = *Some* n ⟹ $\mathcal{C}$ ⊢ [*Current-memory*] : ([] -> [*T-i32*])
— *Grow-memory*
| *grow-memory*:(*memory* $\mathcal{C}$) = *Some* n ⟹ $\mathcal{C}$ ⊢ [*Grow-memory*] : ([*T-i32*] -> [*T-i32*])
— *empty program*
| *empty*:$\mathcal{C}$ ⊢ [] : ([] -> [])
— *composition*
| *composition*:⟦$\mathcal{C}$ ⊢ *es* : (*t1s* -> *t2s*); $\mathcal{C}$ ⊢ [e] : (*t2s* -> *t3s*)⟧ ⟹ $\mathcal{C}$ ⊢ *es* @ [e] : (*t1s* -> *t3s*)
— *weakening*
| *weakening*:$\mathcal{C}$ ⊢ *es* : (*t1s* -> *t2s*) ⟹ $\mathcal{C}$ ⊢ *es* : (*ts* @ *t1s* -> *ts* @ *t2s*)

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

inductive *e-typing* :: [*s-context*, *t-context*, *e list*, *tf*] ⇒ *bool* (-•- ⊢ - : - *60*)
and *s-typing* :: [*s-context*, (*t list*) *option*, *nat*, *v list*, *e list*, *t list*] ⇒ *bool* (-•- ⊩'- - -;- : - *60*) **where**

$\mathcal{C}$ ⊢ *b-es* : *tf* ⟹ $\mathcal{S}$•$\mathcal{C}$ ⊢ $*b-es : tf$

| $[\![\mathcal{S}{\cdot}\mathcal{C} \vdash es : (t1s \text{ -> } t2s);\ \mathcal{S}{\cdot}\mathcal{C} \vdash [e] : (t2s \text{ -> } t3s)]\!] \Longrightarrow \mathcal{S}{\cdot}\mathcal{C} \vdash es \ @\ [e] : (t1s \text{ -> } t3s)$

| $\mathcal{S}{\cdot}\mathcal{C} \vdash es : (t1s \text{ -> } t2s) \Longrightarrow \mathcal{S}{\cdot}\mathcal{C} \vdash es : (ts \ @\ t1s \text{ -> } ts \ @\ t2s)$

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

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

| $[\![cl\text{-}typing\ \mathcal{S}\ cl\ tf]\!] \Longrightarrow \mathcal{S}{\cdot}\mathcal{C}\ \vdash [Callcl\ cl] : tf$

| $[\![\mathcal{S}{\cdot}\mathcal{C} \vdash e0s : (ts \text{ -> } t2s);\ \mathcal{S}{\cdot}\mathcal{C}(\!|label := ([ts]\ @\ (label\ \mathcal{C}))|\!) \vdash es : ([] \text{ -> } t2s);\ length\ ts = n]\!] \Longrightarrow \mathcal{S}{\cdot}\mathcal{C} \vdash [Label\ n\ e0s\ es] : ([] \text{ -> } t2s)$

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

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

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

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

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

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

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

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

inductive *store-typing* :: [*s*, *s-context*] ⇒ *bool* **where**
$[\![\mathcal{S} = (\!|s\text{-}inst = \mathcal{C}s,\ s\text{-}funcs = tfs,\ s\text{-}tab = ns,\ s\text{-}mem = ms,\ s\text{-}globs = tgs|\!);\ list\text{-}all2\ (inst\text{-}typing\ \mathcal{S})\ insts\ \mathcal{C}s;\ list\text{-}all2\ (cl\text{-}typing\ \mathcal{S})\ fs\ tfs;\ list\text{-}all\ (tab\text{-}agree\ \mathcal{S})\ (concat\ tclss);\ list\text{-}all2\ (\lambda\ tcls\ n.\ n \leq length\ tcls)\ tclss\ ns;\ list\text{-}all2\ mem\text{-}agree\ bss\ ms;\ list\text{-}all2\ glob\text{-}agree\ gs\ tgs]\!] \Longrightarrow store\text{-}typing\ (\!|s.inst = insts, s.funcs = fs, s.tab = tclss, s.mem = bss, s.globs = gs|\!)\ \mathcal{S}$

inductive *config-typing* :: [*nat*, *s*, *v list*, *e list*, *t list*] ⇒ *bool* (⊢′- - -;-;- : - *60*) **where**
$[\![store\text{-}typing\ s\ \mathcal{S};\ \mathcal{S}{\cdot}None \Vdash i\ vs;es : ts]\!] \Longrightarrow \vdash i\ s;vs;es : ts$

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*), $(*Relop-f T-f32 fop*)]⦈) ⇝ (⦇[$*C* (*ConstInt32* (*wasm-bool* (*app-relop-f fop c1 c2*)))]⦈)
| *relop-f64*:(⦇[$*C* (*ConstFloat64 c1*), $*C* (*ConstFloat64 c2*), $(*Relop-f T-f64 fop*)]⦈) ⇝ (⦇[$*C* (*ConstInt32* (*wasm-bool* (*app-relop-f fop c1 c2*)))]⦈)

— *convert*
| *convert-Some*:⟦*types-agree t1 v*; *cvt t2 sx v* = (*Some v'*)⟧ ⟹ (|[$(*C v*), $(*Cvtop t2 Convert t1 sx*)]|) ⇝ (|[$(*C v'*)]|)
| *convert-None*:⟦*types-agree t1 v*; *cvt t2 sx v* = *None*⟧ ⟹ (|[$(*C v*), $(*Cvtop t2 Convert t1 sx*)]|) ⇝ (|[*Trap*]|)
— *reinterpret*
| *reinterpret*:*types-agree t1 v* ⟹ (|[$(*C v*), $(*Cvtop t2 Reinterpret t1 None*)]|) ⇝ (|[$(*C* (*wasm-deserialise* (*bits v*) *t2*))]|)
— *unreachable*
| *unreachable*:(|[$ *Unreachable*]|) ⇝ (|[*Trap*]|)
— *nop*
| *nop*:(|[$ *Nop*]|) ⇝ (|[]|)
— *drop*
| *drop*:(|[$(*C v*), ($ *Drop*)]|) ⇝ (|[]|)
— *select*
| *select-false*:*int-eq n 0* ⟹ (|[$(*C v1*), $(*C v2*), $*C* (*ConstInt32 n*), ($ *Select*)]|) ⇝ (|[$(*C v2*)]|)
| *select-true*:*int-ne n 0* ⟹ (|[$(*C v1*), $(*C v2*), $*C* (*ConstInt32 n*), ($ *Select*)]|) ⇝ (|[$(*C v1*)]|)
— *block*
| *block*:⟦*const-list vs*; *length vs* = *n*; *length t1s* = *n*; *length t2s* = *m*⟧ ⟹ (|*vs* @ [$(*Block* (*t1s* -> *t2s*) *es*)]|) ⇝ (|[*Label m* [] (*vs* @ ($∗ *es*))]|)
— *loop*
| *loop*:⟦*const-list vs*; *length vs* = *n*; *length t1s* = *n*; *length t2s* = *m*⟧ ⟹ (|*vs* @ [$(*Loop* (*t1s* -> *t2s*) *es*)]|) ⇝ (|[*Label n* [$(*Loop* (*t1s* -> *t2s*) *es*)] (*vs* @ ($∗ *es*))]|)
— *if*
| *if-false*:*int-eq n 0* ⟹ (|[$*C* (*ConstInt32 n*), $(*If tf e1s e2s*)]|) ⇝ (|[$(*Block tf e2s*)]|)
| *if-true*:*int-ne n 0* ⟹ (|[$*C* (*ConstInt32 n*), $(*If tf e1s e2s*)]|) ⇝ (|[$(*Block tf e1s*)]|)
— *label*
| *label-const*:*const-list vs* ⟹ (|[*Label n es vs*]|) ⇝ (|*vs*|)
| *label-trap*:(|[*Label n es* [*Trap*]]|) ⇝ (|[*Trap*]|)
— *br*
| *br*:⟦*const-list vs*; *length vs* = *n*; *Lfilled i lholed* (*vs* @ [$(*Br i*)]) *LI*⟧ ⟹ (|[*Label n es LI*]|) ⇝ (|*vs* @ *es*|)
— *br-if*
| *br-if-false*:*int-eq n 0* ⟹ (|[$*C* (*ConstInt32 n*), $(*Br-if i*)]|) ⇝ (|[]|)
| *br-if-true*:*int-ne n 0* ⟹ (|[$*C* (*ConstInt32 n*), $(*Br-if i*)]|) ⇝ (|[$(*Br i*)]|)
— *br-table*
| *br-table*:⟦*length is* > (*nat-of-int c*)⟧ ⟹ (|[$*C* (*ConstInt32 c*), $(*Br-table is i*)]|) ⇝ (|[$(*Br* (*is*!(*nat-of-int c*)))]|)
| *br-table-length*:⟦*length is* ≤ (*nat-of-int c*)⟧ ⟹ (|[$*C* (*ConstInt32 c*), $(*Br-table is i*)]|) ⇝ (|[$(*Br i*)]|)
— *local*
| *local-const*:⟦*const-list es*; *length es* = *n*⟧ ⟹ (|[*Local n i vs es*]|) ⇝ (|*es*|)
| *local-trap*:(|[*Local n i vs* [*Trap*]]|) ⇝ (|[*Trap*]|)
— *return*
| *return*:⟦*const-list vs*; *length vs* = *n*; *Lfilled j lholed* (*vs* @ [$*Return*]) *es*⟧ ⟹

$(\!|[Local\ n\ i\ vls\ es]|\!) \rightsquigarrow (\!|vs|\!)$
— *tee-local*
| *tee-local*:$is\text{-}const\ v \Longrightarrow (\!|[v,\ \$(Tee\text{-}local\ i)]|\!) \rightsquigarrow (\!|[v,\ v,\ \$(Set\text{-}local\ i)]|\!)$
| *trap*:$[\![es \neq [Trap];\ Lfilled\ 0\ lholed\ [Trap]\ es]\!] \Longrightarrow (\!|es|\!) \rightsquigarrow (\!|[Trap]|\!)$

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

(*nat-of-int k*) *off* (*tp-length tp*) (*t-length t*) = *None*⟧ ⟹ (|*s*;*vs*;[$*C* (*ConstInt32 k*), $(*Load t* (*Some* (*tp*, *sx*)) *a off*)]|) ⇝-*i* (|*s*;*vs*;[*Trap*]|)
— *store*
| *store-Some*:⟦*types-agree t v*; *smem-ind s i* = *Some j*; ((*mem s*)!*j*) = *m*; *store m* (*nat-of-int k*) *off* (*bits v*) (*t-length t*) = *Some mem'*⟧ ⟹ (|*s*;*vs*;[$*C* (*ConstInt32 k*), $*C v*, $(*Store t None a off*)]|) ⇝-*i* (|*s*(|*mem*:= ((*mem s*)[*j* := *mem'*])|);*vs*;[]|)
| *store-None*:⟦*types-agree t v*; *smem-ind s i* = *Some j*; ((*mem s*)!*j*) = *m*; *store m* (*nat-of-int k*) *off* (*bits v*) (*t-length t*) = *None*⟧ ⟹ (|*s*;*vs*;[$*C* (*ConstInt32 k*), $*C v*, $(*Store t None a off*)]|) ⇝-*i* (|*s*;*vs*;[*Trap*]|)
— *store packed*
| *store-packed-Some*:⟦*types-agree t v*; *smem-ind s i* = *Some j*; ((*mem s*)!*j*) = *m*; *store-packed m* (*nat-of-int k*) *off* (*bits v*) (*tp-length tp*) = *Some mem'*⟧ ⟹ (|*s*;*vs*;[$*C* (*ConstInt32 k*), $*C v*, $(*Store t* (*Some tp*) *a off*)]|) ⇝-*i* (|*s*(|*mem*:= ((*mem s*)[*j* := *mem'*])|);*vs*;[]|)
| *store-packed-None*:⟦*types-agree t v*; *smem-ind s i* = *Some j*; ((*mem s*)!*j*) = *m*; *store-packed m* (*nat-of-int k*) *off* (*bits v*) (*tp-length tp*) = *None*⟧ ⟹ (|*s*;*vs*;[$*C* (*ConstInt32 k*), $*C v*, $(*Store t* (*Some tp*) *a off*)]|) ⇝-*i* (|*s*;*vs*;[*Trap*]|)
— *current-memory*
| *current-memory*:⟦*smem-ind s i* = *Some j*; ((*mem s*)!*j*) = *m*; *mem-size m* = *n*⟧ ⟹ (|*s*;*vs*;[$(*Current-memory*)]|) ⇝-*i* (|*s*;*vs*;[$*C* (*ConstInt32* (*int-of-nat n*))]|)
— *grow-memory*
| *grow-memory*:⟦*smem-ind s i* = *Some j*; ((*mem s*)!*j*) = *m*; *mem-size m* = *n*; *mem-grow m* (*nat-of-int c*) = *mem'*⟧ ⟹ (|*s*;*vs*;[$*C* (*ConstInt32 c*), $(*Grow-memory*)]|) ⇝-*i* (|*s*(|*mem*:= ((*mem s*)[*j* := *mem'*])|);*vs*;[$*C* (*ConstInt32* (*int-of-nat n*))]|)
— *grow-memory fail*
| *grow-memory-fail*:⟦*smem-ind s i* = *Some j*; ((*mem s*)!*j*) = *m*; *mem-size m* = *n*⟧ ⟹ (|*s*;*vs*;[$*C* (*ConstInt32 c*),$(*Grow-memory*)]|) ⇝-*i* (|*s*;*vs*;[$*C* (*ConstInt32 int32-minus-one*)]|)

— *inductive label reduction*
| *label*:⟦(|*s*;*vs*;*es*|) ⇝-*i* (|*s'*;*vs'*;*es'*|); *Lfilled k lholed es les*; *Lfilled k lholed es' les'*⟧ ⟹ (|*s*;*vs*;*les*|) ⇝-*i* (|*s'*;*vs'*;*les'*|)
— *inductive local reduction*
| *local*:⟦(|*s*;*vs*;*es*|) ⇝-*i* (|*s'*;*vs'*;*es'*|)⟧ ⟹ (|*s*;*v0s*;[*Local n i vs es*]|) ⇝-*j* (|*s'*;*v0s*;[*Local n i vs' es'*]|)

end

4 Host Properties

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

lemma *mem-grow-size*:
assumes *mem-grow m n* = *m'*
shows (*mem-size m* + (*64000* ∗ *n*)) = *mem-size m'*
⟨*proof*⟩

lemma *load-size*:

(*load m n off l = None*) = (*mem-size m* < (*off* + *n* + *l*))
⟨*proof*⟩

lemma *load-packed-size*:
(*load-packed sx m n off lp l = None*) = (*mem-size m* < (*off* + *n* + *lp*))
⟨*proof*⟩

lemma *store-size1*:
(*store m n off v l = None*) = (*mem-size m* < (*off* + *n* + *l*))
⟨*proof*⟩

lemma *store-size*:
assumes (*store m n off v l = Some m'*)
shows *mem-size m = mem-size m'*
⟨*proof*⟩

lemma *store-packed-size1*:
(*store-packed m n off v l = None*) = (*mem-size m* < (*off* + *n* + *l*))
⟨*proof*⟩

lemma *store-packed-size*:
assumes (*store-packed m n off v l = Some m'*)
shows *mem-size m = mem-size m'*
⟨*proof*⟩

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

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

5 Auxiliary Type System Properties

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

lemma *typeof-i32*:
assumes *typeof v = T-i32*
shows ∃ *c*. *v = ConstInt32 c*
⟨*proof*⟩

lemma *typeof-i64*:
assumes *typeof v = T-i64*
shows ∃ *c*. *v = ConstInt64 c*
⟨*proof*⟩

lemma *typeof-f32*:
assumes *typeof v* = *T-f32*
shows ∃ *c*. *v* = *ConstFloat32 c*
⟨*proof*⟩

lemma *typeof-f64*:
assumes *typeof v* = *T-f64*
shows ∃ *c*. *v* = *ConstFloat64 c*
⟨*proof*⟩

lemma *exists-v-typeof*: ∃ *v v*. *typeof v* = *t*
⟨*proof*⟩

lemma *lfilled-collapse1*:
assumes *Lfilled n lholed* (*vs*@*es*) *LI*
const-list vs
length vs ≥ *l*
shows ∃ *lholed'*. *Lfilled n lholed'* ((*drop* (*length vs* − *l*) *vs*)@*es*) *LI*
⟨*proof*⟩

lemma *lfilled-collapse2*:
assumes *Lfilled n lholed* (*es*@*es'*) *LI*
shows ∃ *lholed' vs'*. *Lfilled n lholed' es LI*
⟨*proof*⟩

lemma *lfilled-collapse3*:
assumes *Lfilled k lholed* [*Label n les es*] *LI*
shows ∃ *lholed'*. *Lfilled* (*Suc k*) *lholed' es LI*
⟨*proof*⟩

lemma *unlift-b-e*: **assumes** $\mathcal{S}$·$\mathcal{C}$ ⊢ $∗*b-es* : *tf* **shows** $\mathcal{C}$ ⊢ *b-es* : *tf*
⟨*proof*⟩

lemma *store-typing-imp-inst-length-eq*:
assumes *store-typing s* $\mathcal{S}$
shows *length* (*inst s*) = *length* (*s-inst* $\mathcal{S}$)
⟨*proof*⟩

lemma *store-typing-imp-func-length-eq*:
assumes *store-typing s* $\mathcal{S}$
shows *length* (*funcs s*) = *length* (*s-funcs* $\mathcal{S}$)
⟨*proof*⟩

lemma *store-typing-imp-mem-length-eq*:
assumes *store-typing s* $\mathcal{S}$
shows *length* (*s.mem s*) = *length* (*s-mem* $\mathcal{S}$)
⟨*proof*⟩

lemma *store-typing-imp-glob-length-eq*:
assumes *store-typing s* $\mathcal{S}$
shows *length* (*globs s*) = *length* (*s-globs* $\mathcal{S}$)
⟨*proof*⟩

lemma *store-typing-imp-inst-typing*:
assumes *store-typing s* $\mathcal{S}$
i < *length* (*inst s*)
shows *inst-typing* $\mathcal{S}$ ((*inst s*)!*i*) ((*s-inst* $\mathcal{S}$)!*i*)
⟨*proof*⟩

lemma *stab-typed-some-imp-member*:
assumes *stab s i c* = *Some cl*
store-typing s $\mathcal{S}$
i < *length* (*inst s*)
shows *Some cl* ∈ *set* (*concat* (*s.tab s*))
⟨*proof*⟩

lemma *stab-typed-some-imp-cl-typed*:
assumes *stab s i c* = *Some cl*
store-typing s $\mathcal{S}$
i < *length* (*inst s*)
shows ∃ *tf*. *cl-typing* $\mathcal{S}$ *cl tf*
⟨*proof*⟩

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

lemma *b-e-type-empty*: ($\mathcal{C}$ ⊢ [] : (*ts* -> *ts'*)) = (*ts* = *ts'*)
⟨*proof*⟩

lemma *b-e-type-value*:
assumes $\mathcal{C}$ ⊢ [*e*] : (*ts* -> *ts'*)
e = *C v*
shows *ts'* = *ts* @ [*typeof v*]
⟨*proof*⟩

lemma *b-e-type-load*:
assumes $\mathcal{C}$ ⊢ [*e*] : (*ts* -> *ts'*)
e = *Load t tp-sx a off*
shows ∃ *ts''* *sec n*. *ts* = *ts''*@[*T-i32*] ∧ *ts'* = *ts''*@[*t*] ∧ (*memory* $\mathcal{C}$) = *Some n*
load-store-t-bounds a (*option-projl tp-sx*) *t*
⟨*proof*⟩

lemma *b-e-type-store*:
assumes $\mathcal{C}$ ⊢ [*e*] : (*ts* -> *ts'*)
e = *Store t tp a off*
shows *ts* = *ts'*@[*T-i32*, *t*]
∃ *sec n*. (*memory* $\mathcal{C}$) = *Some n*

load-store-t-bounds a tp t
⟨*proof*⟩

lemma *b-e-type-current-memory*:
assumes $\mathcal{C} \vdash [e] : (ts \text{ -> } ts')$
e = Current-memory
shows $\exists sec\ n.\ ts' = ts\ @\ [T\text{-}i32] \wedge (memory\ \mathcal{C}) = Some\ n$
⟨*proof*⟩

lemma *b-e-type-grow-memory*:
assumes $\mathcal{C} \vdash [e] : (ts \text{ -> } ts')$
e = Grow-memory
shows $\exists ts''.\ ts = ts''@[T\text{-}i32] \wedge ts = ts' \wedge (\exists n.\ (memory\ \mathcal{C}) = Some\ n)$
⟨*proof*⟩

lemma *b-e-type-nop*:
assumes $\mathcal{C} \vdash [e] : (ts \text{ -> } ts')$
e = Nop
shows $ts = ts'$
⟨*proof*⟩

definition *arity-2-result* :: *b-e* ⇒ *t* **where**
arity-2-result op2 = (*case op2 of*
Binop-i t - ⇒ *t*
| *Binop-f t* - ⇒ *t*
| *Relop-i t* - ⇒ *T-i32*
| *Relop-f t* - ⇒ *T-i32*)

lemma *b-e-type-binop-relop*:
assumes $\mathcal{C} \vdash [e] : (ts \text{ -> } ts')$
e = Binop-i t iop ∨ *e = Binop-f t fop* ∨ *e = Relop-i t irop* ∨ *e = Relop-f t frop*
shows $\exists ts''.\ ts = ts''@[t,t] \wedge ts' = ts''@[arity\text{-}2\text{-}result(e)]$
e = Binop-f t fop ⟹ *is-float-t t*
e = Relop-f t frop ⟹ *is-float-t t*
⟨*proof*⟩

lemma *b-e-type-testop-drop-cvt0*:
assumes $\mathcal{C} \vdash [e] : (ts \text{ -> } ts')$
e = Testop t testop ∨ *e = Drop* ∨ *e = Cvtop t1 cvtop t2 sx*
shows $ts \neq []$
⟨*proof*⟩

definition *arity-1-result* :: *b-e* ⇒ *t* **where**
arity-1-result op1 = (*case op1 of*
Unop-i t - ⇒ *t*
| *Unop-f t* - ⇒ *t*
| *Testop t* - ⇒ *T-i32*
| *Cvtop t1 Convert* - - ⇒ *t1*

| *Cvtop t1 Reinterpret - - ⇒ t1*)

lemma *b-e-type-unop-testop*:
assumes $\mathcal{C} \vdash [e] : (ts \text{ -> } ts')$
$e = Unop\text{-}i\ t\ iop \vee e = Unop\text{-}f\ t\ fop \vee e = Testop\ t\ testop$
shows $\exists ts''.\ ts = ts''@[t] \wedge ts' = ts''@[arity\text{-}1\text{-}result\ e]$
$e = Unop\text{-}f\ t\ fop \Longrightarrow is\text{-}float\text{-}t\ t$
⟨*proof*⟩

lemma *b-e-type-cvtop*:
assumes $\mathcal{C} \vdash [e] : (ts \text{ -> } ts')$
$e = Cvtop\ t1\ cvtop\ t\ sx$
shows $\exists ts''.\ ts = ts''@[t] \wedge ts' = ts''@[arity\text{-}1\text{-}result\ e]$
$cvtop = Convert \Longrightarrow (t1 \neq t) \wedge (sx = None) = ((is\text{-}float\text{-}t\ t1 \wedge is\text{-}float\text{-}t\ t) \vee (is\text{-}int\text{-}t\ t1 \wedge is\text{-}int\text{-}t\ t \wedge (t\text{-}length\ t1 < t\text{-}length\ t)))$
$cvtop = Reinterpret \Longrightarrow (t1 \neq t) \wedge t\text{-}length\ t1 = t\text{-}length\ t$
⟨*proof*⟩

lemma *b-e-type-drop*:
assumes $\mathcal{C} \vdash [e] : (ts \text{ -> } ts')$
$e = Drop$
shows $\exists t.\ ts = ts'@[t]$
⟨*proof*⟩

lemma *b-e-type-select*:
assumes $\mathcal{C} \vdash [e] : (ts \text{ -> } ts')$
$e = Select$
shows $\exists ts''\ t.\ ts = ts''@[t,t,T\text{-}i32] \wedge ts' = ts''@[t]$
⟨*proof*⟩

lemma *b-e-type-call*:
assumes $\mathcal{C} \vdash [e] : (ts \text{ -> } ts')$
$e = Call\ i$
shows $i < length\ (func\text{-}t\ \mathcal{C})$
$\exists ts''\ tf1\ tf2.\ ts = ts''@tf1 \wedge ts' = ts''@tf2 \wedge (func\text{-}t\ \mathcal{C})!i = (tf1 \text{ -> } tf2)$
⟨*proof*⟩

lemma *b-e-type-call-indirect*:
assumes $\mathcal{C} \vdash [e] : (ts \text{ -> } ts')$
$e = Call\text{-}indirect\ i$
shows $i < length\ (types\text{-}t\ \mathcal{C})$
$\exists ts''\ tf1\ tf2.\ ts = ts''@tf1@[T\text{-}i32] \wedge ts' = ts''@tf2 \wedge (types\text{-}t\ \mathcal{C})!i = (tf1 \text{ -> } tf2)$
⟨*proof*⟩

lemma *b-e-type-get-local*:
assumes $\mathcal{C} \vdash [e] : (ts \text{ -> } ts')$
$e = Get\text{-}local\ i$
shows $\exists t.\ ts' = ts@[t] \wedge (local\ \mathcal{C})!i = t\ i < length(local\ \mathcal{C})$

⟨*proof*⟩

lemma *b-e-type-set-local*:
assumes $\mathcal{C} \vdash [e] : (ts \text{ -> } ts')$
$e = Set\text{-}local\ i$
shows $\exists t.\ ts = ts'@[t] \wedge (local\ \mathcal{C})!i = t\ i < length(local\ \mathcal{C})$
⟨*proof*⟩

lemma *b-e-type-tee-local*:
assumes $\mathcal{C} \vdash [e] : (ts \text{ -> } ts')$
$e = Tee\text{-}local\ i$
shows $\exists ts''\ t.\ ts = ts''@[t] \wedge ts' = ts''@[t] \wedge (local\ \mathcal{C})!i = t\ i < length(local\ \mathcal{C})$
⟨*proof*⟩

lemma *b-e-type-get-global*:
assumes $\mathcal{C} \vdash [e] : (ts \text{ -> } ts')$
$e = Get\text{-}global\ i$
shows $\exists t.\ ts' = ts@[t] \wedge tg\text{-}t((global\ \mathcal{C})!i) = t\ i < length(global\ \mathcal{C})$
⟨*proof*⟩

lemma *b-e-type-set-global*:
assumes $\mathcal{C} \vdash [e] : (ts \text{ -> } ts')$
$e = Set\text{-}global\ i$
shows $\exists t.\ ts = ts'@[t] \wedge (global\ \mathcal{C})!i = (\!|tg\text{-}mut = T\text{-}mut,\ tg\text{-}t = t|\!) \wedge i < length(global\ \mathcal{C})$
⟨*proof*⟩

lemma *b-e-type-block*:
assumes $\mathcal{C} \vdash [e] : (ts \text{ -> } ts')$
$e = Block\ tf\ es$
shows $\exists ts''\ tfn\ tfm.\ tf = (tfn \text{ -> } tfm) \wedge (ts = ts''@tfn) \wedge (ts' = ts''@tfm) \wedge$
$(\mathcal{C}(\!|label := [tfm] @ label\ \mathcal{C}|\!) \vdash es : tf)$
⟨*proof*⟩

lemma *b-e-type-loop*:
assumes $\mathcal{C} \vdash [e] : (ts \text{ -> } ts')$
$e = Loop\ tf\ es$
shows $\exists ts''\ tfn\ tfm.\ tf = (tfn \text{ -> } tfm) \wedge (ts = ts''@tfn) \wedge (ts' = ts''@tfm) \wedge$
$(\mathcal{C}(\!|label := [tfn] @ label\ \mathcal{C}|\!) \vdash es : tf)$
⟨*proof*⟩

lemma *b-e-type-if*:
assumes $\mathcal{C} \vdash [e] : (ts \text{ -> } ts')$
$e = If\ tf\ es1\ es2$
shows $\exists ts''\ tfn\ tfm.\ tf = (tfn \text{ -> } tfm) \wedge (ts = ts''@tfn @ [T\text{-}i32]) \wedge (ts' = ts''@tfm) \wedge$
$(\mathcal{C}(\!|label := [tfm] @ label\ \mathcal{C}|\!) \vdash es1 : tf) \wedge$
$(\mathcal{C}(\!|label := [tfm] @ label\ \mathcal{C}|\!) \vdash es2 : tf)$
⟨*proof*⟩

lemma *b-e-type-br*:
assumes $\mathcal{C} \vdash [e] : (ts \text{ -> } ts')$
$e = Br\ i$
shows $i < length(label\ \mathcal{C})$
$\exists ts\text{-}c\ ts''.\ ts = ts\text{-}c\ @\ ts'' \wedge (label\ \mathcal{C})!i = ts''$
⟨*proof*⟩

lemma *b-e-type-br-if*:
assumes $\mathcal{C} \vdash [e] : (ts \text{ -> } ts')$
$e = Br\text{-}if\ i$
shows $i < length(label\ \mathcal{C})$
$\exists ts\text{-}c\ ts''.\ ts = ts\text{-}c\ @\ ts''\ @\ [T\text{-}i32] \wedge ts' = ts\text{-}c\ @\ ts'' \wedge (label\ \mathcal{C})!i = ts''$
⟨*proof*⟩

lemma *b-e-type-br-table*:
assumes $\mathcal{C} \vdash [e] : (ts \text{ -> } ts')$
$e = Br\text{-}table\ is\ i$
shows $\exists ts\text{-}c\ ts''.\ list\text{-}all\ (\lambda i.\ i < length(label\ \mathcal{C}) \wedge (label\ \mathcal{C})!i = ts'')\ (is@[i]) \wedge ts = ts\text{-}c\ @\ ts''@[T\text{-}i32]$
⟨*proof*⟩

lemma *b-e-type-return*:
assumes $\mathcal{C} \vdash [e] : (ts \text{ -> } ts')$
$e = Return$
shows $\exists ts\text{-}c\ ts''.\ ts = ts\text{-}c\ @\ ts'' \wedge (return\ \mathcal{C}) = Some\ ts''$
⟨*proof*⟩

lemma *b-e-type-comp*:
assumes $\mathcal{C} \vdash es@[e] : (t1s \text{ -> } t4s)$
shows $\exists ts'.\ (\mathcal{C} \vdash es : (t1s \text{ -> } ts')) \wedge (\mathcal{C} \vdash [e] : (ts' \text{ -> } t4s))$
⟨*proof*⟩

lemma *b-e-type-comp2-unlift*:
assumes $\mathcal{S}{\cdot}\mathcal{C} \vdash [\$e1, \$e2] : (t1s \text{ -> } t2s)$
shows $\exists ts'.\ (\mathcal{C} \vdash [e1] : (t1s \text{ -> } ts')) \wedge (\mathcal{C} \vdash [e2] : (ts' \text{ -> } t2s))$
⟨*proof*⟩

lemma *b-e-type-comp2-relift*:
assumes $\mathcal{C} \vdash [e1] : (t1s \text{ -> } ts')\ \mathcal{C} \vdash [e2] : (ts' \text{ -> } t2s)$
shows $\mathcal{S}{\cdot}\mathcal{C} \vdash [\$e1, \$e2] : (ts@t1s \text{ -> } ts@t2s)$
⟨*proof*⟩

lemma *b-e-type-value2*:
assumes $\mathcal{C} \vdash [C\ v1,\ C\ v2] : (t1s \text{ -> } t2s)$
shows $t2s = t1s\ @\ [typeof\ v1,\ typeof\ v2]$
⟨*proof*⟩

lemma *e-type-comp*:
assumes $\mathcal{S}{\cdot}\mathcal{C} \vdash es@[e] : (t1s \text{ -> } t3s)$
shows $\exists\, ts'.\ (\mathcal{S}{\cdot}\mathcal{C} \vdash es : (t1s \text{ -> } ts')) \wedge (\mathcal{S}{\cdot}\mathcal{C} \vdash [e] : (ts' \text{ -> } t3s))$
⟨*proof*⟩

lemma *e-type-comp-conc*:
assumes $\mathcal{S}{\cdot}\mathcal{C} \vdash es : (t1s \text{ -> } t2s)$
$\mathcal{S}{\cdot}\mathcal{C} \vdash es' : (t2s \text{ -> } t3s)$
shows $\mathcal{S}{\cdot}\mathcal{C} \vdash es@es' : (t1s \text{ -> } t3s)$
⟨*proof*⟩

lemma *b-e-type-comp-conc*:
assumes $\mathcal{C} \vdash es : (t1s \text{ -> } t2s)$
$\mathcal{C} \vdash es' : (t2s \text{ -> } t3s)$
shows $\mathcal{C} \vdash es@es' : (t1s \text{ -> } t3s)$
⟨*proof*⟩

lemma *e-type-comp-conc1*:
assumes $\mathcal{S}{\cdot}\mathcal{C} \vdash es@es' : (ts \text{ -> } ts')$
shows $\exists\, ts''.\ (\mathcal{S}{\cdot}\mathcal{C} \vdash es : (ts \text{ -> } ts'')) \wedge (\mathcal{S}{\cdot}\mathcal{C} \vdash es' : (ts'' \text{ -> } ts'))$
⟨*proof*⟩

lemma *e-type-comp-conc2*:
assumes $\mathcal{S}{\cdot}\mathcal{C} \vdash es@es'@es'' : (t1s \text{ -> } t2s)$
shows $\exists\, ts'\ ts''.\ (\mathcal{S}{\cdot}\mathcal{C} \vdash es : (t1s \text{ -> } ts'))$
$\wedge\ (\mathcal{S}{\cdot}\mathcal{C} \vdash es' : (ts' \text{ -> } ts''))$
$\wedge\ (\mathcal{S}{\cdot}\mathcal{C} \vdash es'' : (ts'' \text{ -> } t2s))$
⟨*proof*⟩

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

lemma *e-type-label*:
assumes $\mathcal{S}{\cdot}\mathcal{C} \vdash [Label\ n\ es0\ es] : (ts \text{ -> } ts')$
shows $\exists\, tls\ t2s.\ (ts' = (ts@t2s))$
$\wedge\ length\ tls = n$
$\wedge\ (\mathcal{S}{\cdot}\mathcal{C} \vdash es0 : (tls \text{ -> } t2s))$
$\wedge\ (\mathcal{S}{\cdot}\mathcal{C}(\!|label := [tls] \ @\ (label\ \mathcal{C})|\!) \vdash es : ([] \text{ -> } t2s))$
⟨*proof*⟩

lemma *e-type-callcl-native*:
assumes $\mathcal{S}{\cdot}\mathcal{C} \vdash [Callcl\ cl] : (t1s' \text{ -> } t2s')$
$cl = Func\text{-}native\ i\ tf\ ts\ es$

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

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

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

lemma *s-type-unfold*:
assumes $\mathcal{S}$·*rs* ⊩-*i vs*;*es* : *ts*
shows *i* < *length* (*s-inst* $\mathcal{S}$)
(*rs* = *Some ts*) $\vee$ *rs* = *None*
($\mathcal{S}$·((*s-inst* $\mathcal{S}$)!*i*)(|*local* := (*local* ((*s-inst* $\mathcal{S}$)!*i*)) @ (*map typeof vs*), *return* := *rs*|) $\vdash$ *es* : ([] -> *ts*))
⟨*proof*⟩

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

lemma *e-type-local-shallow*:
assumes $\mathcal{S}$·$\mathcal{C}$ $\vdash$ [*Local n i vs es*] : (*ts* -> *ts'*)
shows $\exists$ *tls*. *length tls* = *n* $\wedge$ *ts'* = *ts*@*tls* $\wedge$ ($\mathcal{S}$·(*Some tls*) ⊩-*i vs*;*es* : *tls*)
⟨*proof*⟩

lemma *e-type-const-unwrap*:
assumes *is-const e*
shows $\exists$ *v*. *e* = $*C v*

⟨*proof*⟩

lemma *is-const-list1*:
assumes *ves* = *map* (*Basic* ∘ *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 ∃ *t*. (*S*·*C* ⊢ [*e*] : (*ts* -> *ts*@[*t*]))
⟨*proof*⟩

lemma *e-type-const*:
assumes *is-const e*
S·*C* ⊢ [*e*] : (*ts* -> *ts*′)
shows ∃ *t*. (*ts*′ = *ts*@[*t*]) ∧ (*S*·*C*′ ⊢ [*e*] : ([] -> [*t*]))
⟨*proof*⟩

lemma *const-typeof*:
assumes *S*·*C* ⊢ [$*C v*] : ([] -> [*t*])
shows *typeof v* = *t*
⟨*proof*⟩

lemma *e-type-const-list*:
assumes *const-list vs*
S·*C* ⊢ *vs* : (*ts* -> *ts*′)
shows ∃ *tvs*. *ts*′ = *ts* @ *tvs* ∧ *length vs* = *length tvs* ∧ (*S*·*C*′ ⊢ *vs* : ([] -> *tvs*))
⟨*proof*⟩

lemma *e-type-const-list-snoc*:
assumes *const-list vs*
S·*C* ⊢ *vs* : ([] -> *ts*@[*t*])
shows ∃ *vs1 v2*. (*S*·*C* ⊢ *vs1* : ([] -> *ts*))
∧ (*S*·*C* ⊢ [*v2*] : (*ts* -> *ts*@[*t*]))
∧ (*vs* = *vs1*@[*v2*])
∧ *const-list vs1*
∧ *is-const v2*
⟨*proof*⟩

lemma *e-type-const-list-cons*:
assumes *const-list vs*
$\mathcal{S}{\cdot}\mathcal{C} \vdash vs : ([] \text{ -> } (ts1@ts2))$
shows $\exists\, vs1\ vs2.\ (\mathcal{S}{\cdot}\mathcal{C} \vdash vs1 : ([] \text{ -> } ts1))$
$\wedge\ (\mathcal{S}{\cdot}\mathcal{C} \vdash vs2 : (ts1 \text{ -> } (ts1@ts2)))$
$\wedge\ vs = vs1@vs2$
$\wedge$ *const-list vs1*
$\wedge$ *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*
$\mathcal{S}{\cdot}\mathcal{C} \vdash lfilled : (ts \text{ -> } ts')$
shows $\exists\, t1s\ t2s\ \mathcal{C}'\ arb\text{-}label.\ (\mathcal{S}{\cdot}\mathcal{C}(\!|label := arb\text{-}label@(label\ \mathcal{C})|\!) \vdash es : (t1s \text{ -> } t2s))$
⟨*proof*⟩

lemma *types-exist-lfilled-weak*:
assumes *Lfilled k lholed es lfilled*
$\mathcal{S}{\cdot}\mathcal{C} \vdash lfilled : (ts \text{ -> } ts')$
shows $\exists\, t1s\ t2s\ \mathcal{C}'\ arb\text{-}label\ arb\text{-}return.\ (\mathcal{S}{\cdot}\mathcal{C}(\!|label := arb\text{-}label, return := arb\text{-}return|\!) \vdash es : (t1s \text{ -> } t2s))$
⟨*proof*⟩

lemma *store-typing-imp-func-agree*:
assumes *store-typing s* $\mathcal{S}$
$i < length\ (s\text{-}inst\ \mathcal{S})$
$j < length\ (func\text{-}t\ ((s\text{-}inst\ \mathcal{S})!i))$
shows $(sfunc\text{-}ind\ s\ i\ j) < length\ (s\text{-}funcs\ \mathcal{S})$
$cl\text{-}typing\ \mathcal{S}\ (sfunc\ s\ i\ j)\ ((s\text{-}funcs\ \mathcal{S})!(sfunc\text{-}ind\ s\ i\ j))$
$((s\text{-}funcs\ \mathcal{S})!(sfunc\text{-}ind\ s\ i\ j)) = (func\text{-}t\ ((s\text{-}inst\ \mathcal{S})!i))!j$
⟨*proof*⟩

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

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

assumes *store-typing s* $\mathcal{S}$
i < *length* (*s-inst* $\mathcal{S}$)
smem-ind s i = *Some j*
shows *j* < *length* (*s-mem* $\mathcal{S}$)
mem-agree ((*mem s*)!*j*) ((*s-mem* $\mathcal{S}$)!*j*)
$\exists x.$ ((*s-mem* $\mathcal{S}$)!*j*) = *x* $\wedge$ (*memory* ((*s-inst* $\mathcal{S}$)!*i*)) = *Some x*
⟨*proof*⟩

lemma *store-typing-imp-mem-agree-None*:
assumes *store-typing s* $\mathcal{S}$
i < *length* (*s-inst* $\mathcal{S}$)
smem-ind s i = *None*
shows (*memory* ((*s-inst* $\mathcal{S}$)!*i*)) = *None*
⟨*proof*⟩

lemma *store-mem-exists*:
assumes *i* < *length* (*s-inst* $\mathcal{S}$)
store-typing s $\mathcal{S}$
shows *Option.is-none* (*memory* ((*s-inst* $\mathcal{S}$)!*i*)) = *Option.is-none* (*inst.mem* ((*inst s*)!*i*))
⟨*proof*⟩

lemma *store-preserved-mem*:
assumes *store-typing s* $\mathcal{S}$
s' = *s*(|*s.mem* := (*s.mem s*)[*i* := *mem'*]|)
mem-size mem' ≥ *mem-size orig-mem*
((*s.mem s*)!*i*) = *orig-mem*
shows *store-typing s'* $\mathcal{S}$
⟨*proof*⟩

lemma *types-agree-imp-e-typing*:
assumes *types-agree t v*
shows $\mathcal{S}$·$\mathcal{C}$ ⊢ [$*C v*] : ([] -> [*t*])
⟨*proof*⟩

lemma *list-types-agree-imp-e-typing*:
assumes *list-all2 types-agree ts vs*
shows $\mathcal{S}$·$\mathcal{C}$ ⊢ $$* *vs* : ([] -> *ts*)
⟨*proof*⟩

lemma *b-e-typing-imp-list-types-agree*:
assumes $\mathcal{C}$ ⊢ (*map* (λ*v*. *C v*) *vs*) : (*ts'* -> *ts'*@*ts*)
shows *list-all2 types-agree ts vs*
⟨*proof*⟩

lemma *e-typing-imp-list-types-agree*:
assumes $\mathcal{S}$·$\mathcal{C}$ ⊢ ($$* *vs*) : (*ts'* -> *ts'*@*ts*)
shows *list-all2 types-agree ts vs*
⟨*proof*⟩

lemma *store-extension-imp-store-typing*:
assumes *store-extension s s′*
store-typing s 𝒮
shows *store-typing s′ 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-cvt*: **assumes** *cvt t sx v = Some v′* **shows** *t = typeof v′*
⟨*proof*⟩

lemma *store-preserved1*:
assumes ⦇*s*;*vs*;*es*⦈ ⇝-*i* ⦇*s′*;*vs′*;*es′*⦈
store-typing s 𝒮
𝒮·*𝒞* ⊢ *es* : (*ts* -> *ts′*)
𝒞 = ((*s-inst 𝒮*)!*i*)⦇*local* := *local* ((*s-inst 𝒮*)!*i*) @ (*map typeof vs*), *label* := *arb-label*, *return* := *arb-return*⦈
i < *length* (*s-inst 𝒮*)
shows *store-typing s′ 𝒮*
⟨*proof*⟩

lemma *store-preserved*:
assumes ⦇*s*;*vs*;*es*⦈ ⇝-*i* ⦇*s′*;*vs′*;*es′*⦈
store-typing s 𝒮
𝒮·*None* ⊨-*i vs*;*es* : *ts*
shows *store-typing s′ 𝒮*
⟨*proof*⟩

lemma *typeof-unop-testop*:
assumes *𝒮*·*𝒞* ⊢ [$*C v*, $*e*] : (*ts* -> *ts′*)
(*e* = (*Unop-i t iop*)) ∨ (*e* = (*Unop-f t fop*)) ∨ (*e* = (*Testop t testop*))
shows (*typeof v*) = *t*
e = (*Unop-f t fop*) ⟹ *is-float-t t*
⟨*proof*⟩

lemma *typeof-cvtop*:
assumes *𝒮*·*𝒞* ⊢ [$*C v*, $*e*] : (*ts* -> *ts′*)

$e = Cvtop\ t1\ cvtop\ t\ sx$
shows $(typeof\ v) = t$
$cvtop = Convert \Longrightarrow (t1 \neq t) \wedge ((sx = None) = ((is\text{-}float\text{-}t\ t1 \wedge is\text{-}float\text{-}t\ t) \vee (is\text{-}int\text{-}t\ t1 \wedge is\text{-}int\text{-}t\ t \wedge (t\text{-}length\ t1 < t\text{-}length\ t))))$
$cvtop = Reinterpret \Longrightarrow (t1 \neq t) \wedge t\text{-}length\ t1 = t\text{-}length\ t$
⟨*proof*⟩

lemma *types-preserved-unop-testop-cvtop*:
assumes $(|[\$C\ v, \$e]|) \rightsquigarrow (|[\$C\ v']|)$
$\mathcal{S}{\cdot}\mathcal{C} \vdash [\$C\ v, \$e] : (ts\ ->\ ts')$
$(e = (Unop\text{-}i\ t\ iop)) \vee (e = (Unop\text{-}f\ t\ fop)) \vee (e = (Testop\ t\ testop)) \vee (e = (Cvtop\ t2\ cvtop\ t\ sx))$
shows $\mathcal{S}{\cdot}\mathcal{C} \vdash [\$C\ v'] : (ts\ ->\ ts')$
⟨*proof*⟩

lemma *typeof-binop-relop*:
assumes $\mathcal{S}{\cdot}\mathcal{C} \vdash [\$C\ v1, \$C\ v2, \$e] : (ts\ ->\ ts')$
$e = Binop\text{-}i\ t\ iop \vee e = Binop\text{-}f\ t\ fop \vee e = Relop\text{-}i\ t\ irop \vee e = Relop\text{-}f\ t\ frop$
shows $typeof\ v1 = t$
$typeof\ v2 = t$
$e = Binop\text{-}f\ t\ fop \Longrightarrow is\text{-}float\text{-}t\ t$
$e = Relop\text{-}f\ t\ frop \Longrightarrow is\text{-}float\text{-}t\ t$
⟨*proof*⟩

lemma *types-preserved-binop-relop*:
assumes $(|[\$C\ v1, \$C\ v2, \$e]|) \rightsquigarrow (|[\$C\ v']|)$
$\mathcal{S}{\cdot}\mathcal{C} \vdash [\$C\ v1, \$C\ v2, \$e] : (ts\ ->\ ts')$
$e = Binop\text{-}i\ t\ iop \vee e = Binop\text{-}f\ t\ fop \vee e = Relop\text{-}i\ t\ irop \vee e = Relop\text{-}f\ t\ frop$
shows $\mathcal{S}{\cdot}\mathcal{C} \vdash [\$C\ v'] : (ts\ ->\ ts')$
⟨*proof*⟩

lemma *types-preserved-drop*:
assumes $(|[\$C\ v, \$e]|) \rightsquigarrow (|[]|)$
$\mathcal{S}{\cdot}\mathcal{C} \vdash [\$C\ v, \$e] : (ts\ ->\ ts')$
$(e = (Drop))$
shows $\mathcal{S}{\cdot}\mathcal{C} \vdash [] : (ts\ ->\ ts')$
⟨*proof*⟩

lemma *types-preserved-select*:
assumes $(|[\$C\ v1, \$C\ v2, \$C\ vn, \$e]|) \rightsquigarrow (|[\$C\ v3]|)$
$\mathcal{S}{\cdot}\mathcal{C} \vdash [\$C\ v1, \$C\ v2, \$C\ vn, \$e] : (ts\ ->\ ts')$
$(e = Select)$
shows $\mathcal{S}{\cdot}\mathcal{C} \vdash [\$C\ v3] : (ts\ ->\ ts')$
⟨*proof*⟩

lemma *types-preserved-block*:
assumes $(|vs\ @\ [\$Block\ (tn\ ->\ tm)\ es]|) \rightsquigarrow (|[Label\ m\ []\ (vs\ @\ (\$*\ es))]|)$

$\mathcal{S}{\cdot}\mathcal{C} \vdash vs$ @ [\$*Block* (*tn* -> *tm*) *es*] : (*ts* -> *ts'*)
const-list vs
length vs = *n*
length tn = *n*
length tm = *m*
shows $\mathcal{S}{\cdot}\mathcal{C} \vdash$ [*Label m* [] (*vs* @ (\$∗ *es*))] : (*ts* -> *ts'*)
⟨*proof*⟩

lemma *types-preserved-if*:
assumes (|[\$*C ConstInt32 n*, \$*If tf e1s e2s*]|) ↝ (|[\$*Block tf es'*]|)
$\mathcal{S}{\cdot}\mathcal{C} \vdash$ [\$*C ConstInt32 n*, \$*If tf e1s e2s*] : (*ts* -> *ts'*)
shows $\mathcal{S}{\cdot}\mathcal{C} \vdash$ [\$*Block tf es'*] : (*ts* -> *ts'*)
⟨*proof*⟩

lemma *types-preserved-tee-local*:
assumes (|[*v*, \$*Tee-local i*]|) ↝ (|[*v*, *v*, \$*Set-local i*]|)
$\mathcal{S}{\cdot}\mathcal{C} \vdash$ [*v*, \$*Tee-local i*] : (*ts* -> *ts'*)
is-const v
shows $\mathcal{S}{\cdot}\mathcal{C} \vdash$ [*v*, *v*, \$*Set-local i*] : (*ts* -> *ts'*)
⟨*proof*⟩

lemma *types-preserved-loop*:
assumes (|*vs* @ [\$*Loop* (*t1s* -> *t2s*) *es*]|) ↝ (|[*Label n* [\$*Loop* (*t1s* -> *t2s*) *es*] (*vs* @ (\$∗ *es*))]|)
$\mathcal{S}{\cdot}\mathcal{C} \vdash vs$ @ [\$*Loop* (*t1s* -> *t2s*) *es*] : (*ts* -> *ts'*)
const-list vs
length vs = *n*
length t1s = *n*
length t2s = *m*
shows $\mathcal{S}{\cdot}\mathcal{C} \vdash$ [*Label n* [\$*Loop* (*t1s* -> *t2s*) *es*] (*vs* @ (\$∗ *es*))] : (*ts* -> *ts'*)
⟨*proof*⟩

lemma *types-preserved-label-value*:
assumes (|[*Label n es0 vs*]|) ↝ (|*vs*|)
$\mathcal{S}{\cdot}\mathcal{C} \vdash$ [*Label n es0 vs*] : (*ts* -> *ts'*)
const-list vs
shows $\mathcal{S}{\cdot}\mathcal{C} \vdash vs$: (*ts* -> *ts'*)
⟨*proof*⟩

lemma *types-preserved-br-if*:
assumes (|[\$*C ConstInt32 n*, \$*Br-if i*]|) ↝ (|*e*|)
$\mathcal{S}{\cdot}\mathcal{C} \vdash$ [\$*C ConstInt32 n*, \$*Br-if i*] : (*ts* -> *ts'*)
e = [\$*Br i*] ∨ *e* = []
shows $\mathcal{S}{\cdot}\mathcal{C} \vdash e$: (*ts* -> *ts'*)
⟨*proof*⟩

lemma *types-preserved-br-table*:
assumes (|[\$*C ConstInt32 c*, \$*Br-table is i*]|) ↝ (|[\$*Br i'*]|)
$\mathcal{S}{\cdot}\mathcal{C} \vdash$ [\$*C ConstInt32 c*, \$*Br-table is i*] : (*ts* -> *ts'*)

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

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

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

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

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

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

lemma *types-preserved-callcl-host-some*:
assumes $\mathcal{S}{\cdot}\mathcal{C} \vdash ves\ @\ [Callcl\ cl] : (ts \text{ -> } ts')$

$cl = Func\text{-}host\ (t1s\ ->\ t2s)\ f$
$ves = \$\$*\ vcs$
$length\ vcs = n$
$length\ t1s = n$
$length\ t2s = m$
$host\text{-}apply\ s\ (t1s\ ->\ t2s)\ f\ vcs\ hs = Some\ (s', vcs')$
$store\text{-}typing\ s\ \mathcal{S}$
shows $\mathcal{S}\cdot\mathcal{C} \vdash \$\$*\ vcs' : (ts\ ->\ ts')$
⟨*proof*⟩

lemma *types-imp-concat*:
assumes $\mathcal{S}\cdot\mathcal{C} \vdash es\ @\ [e]\ @\ es' : (ts\ ->\ ts')$
$\bigwedge tes\ tes'.\ ((\mathcal{S}\cdot\mathcal{C} \vdash [e] : (tes\ ->\ tes')) \Longrightarrow (\mathcal{S}\cdot\mathcal{C} \vdash [e'] : (tes\ ->\ tes')))$
shows $\mathcal{S}\cdot\mathcal{C} \vdash es\ @\ [e']\ @\ es' : (ts\ ->\ ts')$
⟨*proof*⟩

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

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

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

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

Lfilled i lholed (vs @ [$Br i]) LI
shows $\mathcal{S}\cdot\mathcal{C} \vdash (vs\ @\ es0) : (ts\ ->\ ts')$
⟨*proof*⟩

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

lemma *types-preserved-b-e1*:
assumes $(\!|es|\!) \rightsquigarrow (\!|es'|\!)$
store-typing s $\mathcal{S}$
$\mathcal{S}\cdot\mathcal{C} \vdash es : (ts\ ->\ ts')$
shows $\mathcal{S}\cdot\mathcal{C} \vdash es' : (ts\ ->\ ts')$
⟨*proof*⟩

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

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

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

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

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

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

shows $\mathcal{S}\cdot\mathcal{C} \vdash [\$C\ v] : (ts\ \text{->}\ ts')$
⟨*proof*⟩

lemma *types-preserved-get-local*:
assumes $\mathcal{S}\cdot\mathcal{C} \vdash [\$Get\text{-}local\ i] : (ts\ \text{->}\ ts')$
$length\ vi = i$
$(local\ \mathcal{C}) = map\ typeof\ (vi\ @\ [v]\ @\ vs)$
shows $\mathcal{S}\cdot\mathcal{C} \vdash [\$C\ v] : (ts\ \text{->}\ ts')$
⟨*proof*⟩

lemma *types-preserved-set-local*:
assumes $\mathcal{S}\cdot\mathcal{C} \vdash [\$C\ v', \$Set\text{-}local\ i] : (ts\ \text{->}\ ts')$
$length\ vi = i$
$(local\ \mathcal{C}) = map\ typeof\ (vi\ @\ [v]\ @\ vs)$
shows $(\mathcal{S}\cdot\mathcal{C} \vdash [] : (ts\ \text{->}\ ts')) \wedge map\ typeof\ (vi\ @\ [v]\ @\ vs) = map\ typeof\ (vi\ @\ [v']\ @\ vs)$
⟨*proof*⟩

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

lemma *lholed-same-type*:
assumes *Lfilled k lholed es les*
Lfilled k lholed es' les'
$\mathcal{S}\cdot\mathcal{C} \vdash les : (ts\ \text{->}\ ts')$
$\bigwedge arb\text{-}labs\ ts\ ts'.$
$\mathcal{S}\cdot(\mathcal{C}(\!|label := arb\text{-}labs@(label\ \mathcal{C})|\!)) \vdash es : (ts\ \text{->}\ ts')$
$\Longrightarrow \mathcal{S}\cdot(\mathcal{C}(\!|label := arb\text{-}labs@(label\ \mathcal{C})|\!)) \vdash es' : (ts\ \text{->}\ ts')$
shows $(\mathcal{S}\cdot\mathcal{C} \vdash les' : (ts\ \text{->}\ ts'))$
⟨*proof*⟩

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

lemma *types-preserved-e*:
assumes $(\!|s;vs;es|\!) \rightsquigarrow\text{-}i\ (\!|s';vs';es'|\!)$
store-typing s $\mathcal{S}$
$\mathcal{S}\cdot None \Vdash\text{-}i\ vs;es : ts$

shows $\mathcal{S}{\cdot}None \Turnstile\text{-}i\ vs';es' : ts$
⟨*proof*⟩

6.2 Progress

lemma *const-list-no-progress*:
assumes *const-list es*
shows $\neg(\lfloor s;vs;es \rfloor) \rightsquigarrow\text{-}\ i\ (\lfloor s';vs';es' \rfloor)$
⟨*proof*⟩

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

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

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

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

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

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

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

$\wedge$ *const-list vs*
⟨*proof*⟩

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

lemma *progress-LN1*:
assumes (*Lfilled j lholed* [$*Br* (*j*+*k*)] *es*)
$\mathcal{S}{\cdot}\mathcal{C} \vdash es : (ts \text{ -> } ts')$
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*
const-list cs
shows $\exists v.\ cs = [\$C\ v]$
⟨*proof*⟩

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

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

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

lemma *const-of-f64*:
assumes *const-list cs*

$\mathcal{S}{\cdot}\mathcal{C} \vdash cs : ([] \text{ -> } [T\text{-}f64])$
shows $\exists c.\ cs = [\$C\ ConstFloat64\ c]$
⟨*proof*⟩

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

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

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

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

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

lemma *progress-binop-relop-f*:
assumes $\mathcal{S}{\cdot}\mathcal{C} \vdash cs : ([] \text{ -> } [t,\ t])$

is-float-t t
const-list cs
$e = Binop\text{-}f\ t\ fop \vee e = Relop\text{-}f\ t\ frop$
shows $\exists a\ s'\ vs'\ es'.\ (\!|s;vs;cs@([\$e])|\!) \rightsquigarrow\text{-}i\ (\!|s';vs';es'|\!)$
⟨*proof*⟩

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

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

lemma *progress-e1*:
assumes $\mathcal{S}{\cdot}None \vdash\text{-}i\ vs;es : ts$
shows $\neg(Lfilled\ k\ lholed\ [\$Return]\ es)$
⟨*proof*⟩

lemma *progress-e2*:
assumes $\mathcal{S}{\cdot}None \vdash\text{-}i\ vs;es : ts$
store-typing s $\mathcal{S}$
shows $(Lfilled\ k\ lholed\ [\$Br\ (j)]\ es) \Longrightarrow j < k$
⟨*proof*⟩

lemma *progress-e3*:
assumes $\mathcal{S}{\cdot}None \vdash\text{-}i\ vs;cs\text{-}es : ts'$
$cs\text{-}es \neq [Trap]$
$\neg\ const\text{-}list\ (cs\text{-}es)$
store-typing s $\mathcal{S}$
shows $\exists a\ s'\ vs'\ es'.\ (\!|s;vs;cs\text{-}es|\!) \rightsquigarrow\text{-}i\ (\!|s';vs';es'|\!)$
⟨*proof*⟩

end

7 Soundness Theorems

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

theorem *preservation*:
  assumes $\vdash$-*i* *s*;*vs*;*es* : *ts*
    ($\!|$*s*;*vs*;*es*$|\!$) $\rightsquigarrow$-*i* ($\!|$*s'*;*vs'*;*es'*$|\!$)
  shows $\vdash$-*i* *s'*;*vs'*;*es'* : *ts*
⟨*proof*⟩

theorem *progress*:
  assumes $\vdash$-*i* *s*;*vs*;*es* : *ts*
  shows *const-list es* ∨ *es* = [*Trap*] ∨ (∃ *a s' vs' es'*. ($\!|$*s*;*vs*;*es*$|\!$) $\rightsquigarrow$-*i* ($\!|$*s'*;*vs'*;*es'*$|\!$))
⟨*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* ⇒ *ct list* **where**
  to-ct-list ts = *map TSome ts*

fun *ct-eq* :: *ct* ⇒ *ct* ⇒ *bool* **where**
  ct-eq (*TSome t*) (*TSome t'*) = (*t* = *t'*)
| *ct-eq TAny* - = *True*
| *ct-eq* - *TAny* = *True*

definition *ct-list-eq* :: *ct list* ⇒ *ct list* ⇒ *bool* **where**
  ct-list-eq ct1s ct2s = *list-all2 ct-eq ct1s ct2s*

definition *ct-prefix* :: *ct list* ⇒ *ct list* ⇒ *bool* **where**
  ct-prefix xs ys = (∃ *as bs*. *ys* = *as*@*bs* ∧ *ct-list-eq as xs*)

definition *ct-suffix* :: *ct list* ⇒ *ct list* ⇒ *bool* **where**
  ct-suffix xs ys = (∃ *as bs*. *ys* = *as*@*bs* ∧ *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-eq x y* = (∃ *t*. *ct-eq x* (*TSome t*) ∧ *ct-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-list-eq (*to-ct-list ts*) (*to-ct-list ts′*) = (*ts* = *ts′*)
⟨*proof*⟩

lemma *ct-list-eq-exists*: ∃ *ys*. *ct-list-eq xs* (*to-ct-list ys*)
⟨*proof*⟩

lemma *ct-list-eq-common-tsome-list*:
ct-list-eq xs ys = (∃ *zs*. *ct-list-eq xs* (*to-ct-list zs*) ∧ *ct-list-eq* (*to-ct-list zs*) *ys*)
⟨*proof*⟩

lemma *ct-list-eq-cons-ct-list*:
assumes *ct-list-eq* (*to-ct-list as*) (*xs* @ *ys*)
shows ∃ *bs bs′*. *as* = *bs* @ *bs′* ∧ *ct-list-eq* (*to-ct-list bs*) *xs* ∧ *ct-list-eq* (*to-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 ∃ *bs*. *as* = *bs* @ *ys* ∧ *ct-list-eq* (*to-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-prefix xs* (*ys* @ [*y*]) = (*ct-list-eq xs* (*ys*@[*y*]) ∨ *ct-prefix xs ys*)
⟨*proof*⟩

lemma *ct-prefix-nil*:*ct-prefix* [] *xs*
¬*ct-prefix* (*x* # *xs*) []
⟨*proof*⟩

lemma *Cons-ct-prefix-Cons*[*simp*]: *ct-prefix* (*x* # *xs*) (*y* # *ys*) = ((*ct-eq x y*) ∧ *ct-prefix xs ys*)
⟨*proof*⟩

lemma *ct-prefix-code* [*code*]:
ct-prefix [] *xs* = *True*
ct-prefix (*x* # *xs*) [] = *False*
ct-prefix (*x* # *xs*) (*y* # *ys*) = ((*ct-eq x y*) ∧ *ct-prefix xs ys*)
⟨*proof*⟩

lemma *ct-suffix-to-ct-prefix* [*code*]: *ct-suffix xs ys* = *ct-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 ∃ *as′*. *to-ct-list as′* = *as*
∃ *bs′*. *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 *length ts* ≤ *length ts′*
⟨*proof*⟩

lemma *ct-suffix-take*:
assumes *ct-suffix ts ts′*
shows *ct-suffix* ((*take* (*length ts* − *n*) *ts*)) ((*take* (*length ts′* − *n*) *ts′*))
⟨*proof*⟩

lemma *ct-suffix-ts-conv-suffix*:
ct-suffix (*to-ct-list ts*) (*to-ct-list ts′*) = *suffix ts ts′*
⟨*proof*⟩

lemma *ct-suffix-exists*: ∃ *ts-c*. *ct-suffix x1* (*to-ct-list ts-c*)
⟨*proof*⟩

lemma *ct-suffix-ct-list-eq-exists*:
assumes *ct-suffix x1 x2*
shows ∃ *ts-c*. *ct-suffix x1* (*to-ct-list ts-c*) ∧ *ct-list-eq* (*to-ct-list ts-c*) *x2*
⟨*proof*⟩

lemma *ct-suffix-cons-ct-list*:
assumes *ct-suffix* (*xs*@*ys*) (*to-ct-list zs*)
shows ∃ *as bs*. *zs* = *as*@*bs* ∧ *ct-list-eq ys* (*to-ct-list bs*) ∧ *ct-suffix xs* (*to-ct-list as*)
⟨*proof*⟩

lemma *ct-suffix-cons-ct-list1*:
assumes *ct-suffix* (*xs*@(*to-ct-list ys*)) (*to-ct-list zs*)
shows ∃ *as*. *zs* = *as*@*ys* ∧ *ct-suffix xs* (*to-ct-list as*)
⟨*proof*⟩

lemma *ct-suffix-cons2*:
assumes *ct-suffix* (*xs*) (*ys*@*zs*)
length xs = *length zs*
shows *ct-list-eq xs zs*
⟨*proof*⟩

lemma *ct-suffix-imp-ct-list-eq*:
assumes *ct-suffix xs ys*
shows *ct-list-eq* (*drop* (*length ys* − *length xs*) *ys*) *xs*
⟨*proof*⟩

lemma *ct-suffix-extend-ct-list-eq*:
assumes *ct-suffix xs ys*
ct-list-eq xs' ys'
shows *ct-suffix* (*xs*@*xs'*) (*ys*@*ys'*)
⟨*proof*⟩

lemma *ct-suffix-extend-any1*:
assumes *ct-suffix xs ys*
length xs < *length ys*
shows *ct-suffix* (*TAny*#*xs*) *ys*
⟨*proof*⟩

lemma *ct-suffix-singleton-any*: *ct-suffix* [*TAny*] [*t*]
⟨*proof*⟩

lemma *ct-suffix-cons-it*: *ct-suffix xs* (*xs'*@*xs*)
⟨*proof*⟩

lemma *ct-suffix-singleton*:
assumes *length cts* > *0*
shows *ct-suffix* [*TAny*] *cts*
⟨*proof*⟩

lemma *ct-suffix-less*:
assumes *ct-suffix* (*xs*@*xs'*) *ys*
shows *ct-suffix xs' ys*
⟨*proof*⟩

lemma *ct-suffix-unfold-one*: *ct-suffix* (*xs*@[*x*]) (*ys*@[*y*]) = ((*ct-eq x y*) ∧ *ct-suffix 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 ∃ *ts' ts''*. (*ts* = *ts'*@*ts''*) ∧ *ct-list-eq cts* (*to-ct-list ts''*)
⟨*proof*⟩

lemma *c-types-agree-not-bot-exists*:
assumes *ts* ≠ *Bot*
shows ∃ *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 ∃ *c-t'*. *consume ctn cts* = *c-t'* ∧ *c-types-agree c-t' ts'*
⟨*proof*⟩

lemma *type-update-type*:
assumes *type-update* (*Type ts*) (*to-ct-list cons*) *prods* = *ts'*
ts' ≠ *Bot*
shows (*ts'* = *prods* ∧ (∃ *ts-c*. *prods* = (*TopType ts-c*)))
∨ (∃ *ts-a ts-b*. *prods* = *Type ts-a* ∧ *ts* = *ts-b*@*cons* ∧ *ts'* = *Type* (*ts-b*@*ts-a*))
⟨*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'*
∃ *t-ag'*. *t-ag* = *t-ag'*@*prods* ∧ *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 ≠ *Bot*
shows *tm* = *TopType* [*TAny*]
⟨*proof*⟩

lemma *type-update-select-length1*:
assumes *type-update-select* (*TopType cts*) = *tm*
length cts = *1*
tm ≠ *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 ≠ *Bot*
shows ∃ *t1 t2*. *cts* = [*t1*, *t2*] ∧ *ct-eq t2* (*TSome T-i32*) ∧ *tm* = *TopType* [*t1*]
⟨*proof*⟩

lemma *type-update-select-length3*:
assumes *type-update-select* (*TopType cts*) = (*TopType ctm*)
length cts ≥ *3*
shows ∃ *cts' ct1 ct2 ct3*. *cts* = *cts'*@[*ct1*, *ct2*, *ct3*] ∧ *ct-eq ct3* (*TSome T-i32*)
⟨*proof*⟩

lemma *type-update-select-type-length3*:
assumes *type-update-select* (*Type tn*) = (*Type tm*)
shows ∃ *t ts'*. *tn* = *ts'*@[*t*, *t*, *T-i32*]
⟨*proof*⟩

lemma *select-return-top-exists*:
assumes *select-return-top cts c1 c2* = *ctm*
ctm ≠ *Bot*
shows ∃ *xs*. *ctm* = *TopType xs*
⟨*proof*⟩

lemma *type-update-select-top-exists*:
assumes *type-update-select xs* = (*TopType tm*)
shows ∃ *tn*. *xs* = *TopType tn*
⟨*proof*⟩

lemma *type-update-select-conv-select-return-top*:
assumes *ct-suffix* [*TSome T-i32*] *cts*
length cts ≥ *3*
shows *type-update-select* (*TopType cts*) = (*select-return-top cts* (*cts*!(*length cts*−*2*))
(*cts*!(*length cts*−*3*)))
⟨*proof*⟩

lemma *select-return-top-ct-eq*:
assumes *select-return-top cts c1 c2* = *TopType ctm*
length cts ≥ *3*
c-types-agree (*TopType ctm*) *cm*
shows ∃ *c′ cm′*. *cm* = *cm′*@[*c′*]
∧ *ct-suffix* (*take* (*length cts* − *3*) *cts*) (*to-ct-list cm′*)
∧ *ct-eq c1* (*TSome c′*)
∧ *ct-eq c2* (*TSome c′*)
⟨*proof*⟩

end

9 Executable Type Checker

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

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

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

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

lemma *same-lab-h-conv-list-all*:
assumes *same-lab-h ils ls ts′* = *Some ts*
shows *list-all* (λ*i*. *i* < *length ls* ∧ *ls*!*i* = *ts*) *ils* ∧ *ts′* = *ts*
⟨*proof*⟩

lemma *same-lab-conv-list-all*:
assumes *same-lab ils ls = Some ts*
shows *list-all* (λ*i*. *i* < *length ls* ∧ *ls*!*i* = *ts*) *ils*
⟨*proof*⟩

lemma *list-all-conv-same-lab-h*:
assumes *list-all* (λ*i*. *i* < *length ls* ∧ *ls*!*i* = *ts*) *ils*
shows *same-lab-h ils ls ts = Some ts*
⟨*proof*⟩

lemma *list-all-conv-same-lab*:
assumes *list-all* (λ*i*. *i* < *length ls* ∧*ls*!*i* = *ts*) (*is*@[*i*])
shows *same-lab* (*is*@[*i*]) *ls = Some ts*
⟨*proof*⟩

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

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

```
                                                    else Bot)

| check-single C (Cvtop t1 Convert t2 sx) ts = (if (convert-cond t1 t2 sx)
                                                  then type-update ts [TSome t2] (Type [t1])
                                                    else Bot)

| check-single C (Cvtop t1 Reinterpret t2 sx) ts = (if ((t1 ≠ t2) ∧ t-length t1 =
t-length t2 ∧ sx = None)
                                                       then type-update ts [TSome t2] (Type
[t1])
                                                         else Bot)

| check-single C (Unreachable) ts = type-update ts [] (TopType [])
| check-single C (Nop) ts = ts
| check-single C (Drop) ts = type-update ts [TAny] (Type [])
| check-single C (Select) ts = type-update-select ts

| check-single C (Block (tn -> tm) es) ts = (if (b-e-type-checker (C⦇label := ([tm]
@ (label C))⦈) es (tn -> tm))
                                                 then type-update ts (to-ct-list tn) (Type tm)
                                                  else Bot)

| check-single C (Loop (tn -> tm) es) ts = (if (b-e-type-checker (C⦇label := ([tn]
@ (label C))⦈) es (tn -> tm))
                                                 then type-update ts (to-ct-list tn) (Type tm)
                                                  else Bot)

| check-single C (If (tn -> tm) es1 es2) ts = (if (b-e-type-checker (C⦇label := ([tm]
@ (label C))⦈) es1 (tn -> tm)
                                                       ∧ b-e-type-checker (C⦇label := ([tm] @
(label C))⦈) es2 (tn -> tm))
                                                 then type-update ts (to-ct-list (tn@[T-i32]))
(Type tm)
                                                  else Bot)

| check-single C (Br i) ts = (if i < length (label C)
                               then type-update ts (to-ct-list ((label C)!i)) (TopType [])
                                else Bot)

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

| check-single C (Br-table is i) ts = (case (same-lab (is@[i]) (label C)) of
                                         None ⇒ Bot
                                       | Some tls ⇒ type-update ts (to-ct-list (tls @ [T-i32]))
(TopType []))
```

```
| check-single C (Return) ts = (case (return C) of
                                  None ⇒ Bot
                                | Some tls ⇒ 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) ⇒ type-update ts (to-ct-list tn) (Type
tm))
                                  else Bot)

| check-single C (Call-indirect i) ts = (if (table C) ≠ None ∧ i < length (types-t
C)
                                      then (case ((types-t C)!i) of
                                          (tn -> tm) ⇒ 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) ∧ 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) ≠ None ∧ 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) ≠ None ∧ load-store-t-bounds
a tp t
                                       then type-update ts [TSome T-i32, TSome t]
(Type [])
                                      else Bot)
```

| *check-single* $\mathcal{C}$ *Current-memory ts* = (*if* (*memory* $\mathcal{C}$) $\neq$ *None*
then type-update ts [] (*Type* [*T-i32*])
else Bot)

| *check-single* $\mathcal{C}$ *Grow-memory ts* = (*if* (*memory* $\mathcal{C}$) $\neq$ *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*
$\mathcal{C} \vdash$ [*e*] : (*t-in* -> *t-out*)
shows $\exists$ *tn*. *c-types-agree* (*Type x1*) *tn* $\wedge$ $\mathcal{C} \vdash$ [*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*
$\mathcal{C} \vdash$ [*e*] : (*t-in* -> *t-out*)
shows $\exists$ *tn*. *c-types-agree* (*TopType x1*) *tn* $\wedge$ $\mathcal{C} \vdash$ [*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 $\neq$ *Bot*
ts' $\neq$ *Bot*
shows $\exists$ *tn*. *c-types-agree ts tn* $\wedge$ *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 $\neq$ *Bot*
ts' $\neq$ *Bot*
c-types-agree ts' tm
$\mathcal{C} \vdash$ [*e*] : (*t-in* -> *t-out*)
shows $\exists$ *tn*. *c-types-agree ts tn* $\wedge$ $\mathcal{C} \vdash$ [*e*] : (*tn* -> *tm*)
⟨*proof*⟩

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

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

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

lemma *b-e-type-checker-sound*:
assumes *b-e-type-checker* $\mathcal{C}$ *es* $(tn \text{ -> } tm)$
shows $\mathcal{C} \vdash es : (tn \text{ -> } tm)$
⟨*proof*⟩

10.2 Completeness

lemma *check-single-imp*:
assumes *check-single* $\mathcal{C}$ *e ctn* = *ctm*
$ctm \neq Bot$
shows *check-single* $\mathcal{C}$ *e* = *id*
$\vee$ *check-single* $\mathcal{C}$ *e* = $(\lambda ctn.$ *type-update-select ctn*$)$
$\vee\ (\exists cons\ prods.$ (*check-single* $\mathcal{C}$ *e* = $(\lambda ctn.$ *type-update ctn cons prods*$)))$
⟨*proof*⟩

lemma *check-equiv-fold*:
check $\mathcal{C}$ *es ts* = *foldl* $(\lambda$ *ts e.* (*case ts of Bot* $\Rightarrow$ *Bot* | - $\Rightarrow$ *check-single* $\mathcal{C}$ *e ts*$)$
ts es
⟨*proof*⟩

lemma *check-neq-bot-snoc*:
assumes *check* $\mathcal{C}$ (*es*@[*e*]) *ts* $\neq$ *Bot*
shows *check* $\mathcal{C}$ *es ts* $\neq$ *Bot*
⟨*proof*⟩

lemma *check-unfold-snoc*:
assumes *check* $\mathcal{C}$ *es ts* $\neq$ *Bot*
shows *check* $\mathcal{C}$ (*es*@[*e*]) *ts* = *check-single* $\mathcal{C}$ *e* (*check* $\mathcal{C}$ *es ts*)
⟨*proof*⟩

lemma *check-single-imp-weakening*:
assumes *check-single* $\mathcal{C}$ *e* (*Type t1s*) = *ctm*
ctm $\neq$ *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* -> *t2s*)
b-e-type-checker $\mathcal{C}$ [*e*] (*t2s* -> *t3s*)
shows *b-e-type-checker* $\mathcal{C}$ (*es* @ [*e*]) (*t1s* -> *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* -> *t2s*)

shows *b-e-type-checker* $\mathcal{C}$ *es* (*ts*@*t1s* -> *ts*@*t2s*)
⟨*proof*⟩

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

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

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* ≡ *RSCrash CError*

type-synonym *depth* = *nat*
type-synonym *fuel* = *nat*

type-synonym *config-tuple* = *s* × *v list* × *e list*

type-synonym *config-one-tuple* = *s* × *v list* × *v list* × *e*

type-synonym *res-tuple* = *s* × *v list* × *res-step*

fun *split-vals* :: *b-e list* ⇒ *v list* × *b-e list* **where**
split-vals ((*C v*)#*es*) = (*let* (*vs'*, *es'*) = *split-vals es in* (*v*#*vs'*, *es'*))
| *split-vals es* = ([], *es*)

fun *split-vals-e* :: *e list* ⇒ *v list* × *e list* **where**

```
  split-vals-e (($ C v)#es) = (let (vs', es') = split-vals-e es in (v#vs', es'))
| split-vals-e es = ([], es)

fun split-n :: v list ⇒ nat ⇒ v list × v list where
  split-n [] n = ([], [])
| split-n es 0 = ([], es)
| split-n (e#es) (Suc n) = (let (es', es'') = split-n es n in (e#es', es''))

lemma split-n-conv-take-drop: split-n es n = (take n es, drop n es)
  ⟨proof⟩

lemma split-n-length:
  assumes split-n es n = (es1, es2) length es ≥ n
  shows length es1 = n
  ⟨proof⟩

lemma split-n-conv-app:
  assumes split-n es n = (es1, es2)
  shows es = es1@es2
  ⟨proof⟩

lemma app-conv-split-n:
  assumes es = es1@es2
  shows split-n es (length es1) = (es1, es2)
  ⟨proof⟩

lemma split-vals-const-list: split-vals (map EConst vs) = (vs, [])
  ⟨proof⟩

lemma split-vals-e-const-list: split-vals-e ($$* vs) = (vs, [])
  ⟨proof⟩

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

abbreviation expect :: 'a option ⇒ ('a ⇒ 'b) ⇒ 'b ⇒ 'b where
  expect a f b ≡ (case a of
                    Some a' ⇒ f a'
                  | None ⇒ b)

abbreviation vs-to-es ::  v list ⇒ e list
  where vs-to-es v ≡ $$* (rev v)

definition e-is-trap :: e ⇒ bool where
  e-is-trap e = (case e of Trap ⇒ True | - ⇒ False)

definition es-is-trap :: e list ⇒ bool where
```

```
  es-is-trap es = (case es of [e] ⇒ e-is-trap e | - ⇒ False)

lemma[simp]: e-is-trap e = (e = Trap)
  ⟨proof⟩

lemma[simp]: es-is-trap es = (es = [Trap])
⟨proof⟩

axiomatization
  mem-grow-impl:: mem ⇒ nat ⇒ mem option where
  mem-grow-impl-correct:(mem-grow-impl m n = Some m') ⟹ (mem-grow m n
= m')


axiomatization
  host-apply-impl:: s ⇒ tf ⇒ host ⇒ v list ⇒ (s × v list) option where
 host-apply-impl-correct:(host-apply-impl s tf h vs = Some m') ⟹ (∃ hs. host-apply
s tf h vs hs = Some m')

function (sequential)
    run-step :: depth ⇒ nat ⇒ config-tuple ⇒ res-tuple
and run-one-step :: depth ⇒ nat ⇒ config-one-tuple ⇒ res-tuple where
  run-step d i (s,vs,es) = (let (ves, es') = split-vals-e es in
                           case es' of
                             [] ⇒ (s,vs, crash-error)
                           | e#es'' ⇒
                             if e-is-trap e
                               then
                                 if (es'' ≠ [] ∨ ves ≠ [])
                                   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 ⇒ (s', vs', RSNormal (res@es''))
                                  | - ⇒ (s', vs', r)))
| run-one-step d i (s, vs, ves, e) =
    (case e of
   — B-E
     — UNOPS
       $(Unop-i T-i32 iop) ⇒
        (case ves of
           (ConstInt32 c)#ves' ⇒
             (s, vs, RSNormal (vs-to-es ((ConstInt32 (app-unop-i iop c))#ves')))
         | - ⇒ (s, vs, crash-error))
     | $(Unop-i T-i64 iop) ⇒
         (case ves of
```

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

```
    | $(Testop T-i64 testop) ⇒
        (case ves of
          (ConstInt64 c)#ves' ⇒
              (s, vs, RSNormal (vs-to-es ((ConstInt32 (wasm-bool (app-testop-i
testop c)))#ves')))
          | - ⇒ (s, vs, crash-error))
    | $(Testop - testop) ⇒ (s, vs, crash-error)
    — RELOPS
    | $(Relop-i T-i32 iop) ⇒
        (case ves of
          (ConstInt32 c2)#(ConstInt32 c1)#ves' ⇒
            (s, vs, RSNormal (vs-to-es ((ConstInt32 (wasm-bool (app-relop-i iop
c1 c2)))#ves')))
          | - ⇒ (s, vs, crash-error))
    | $(Relop-i T-i64 iop) ⇒
        (case ves of
          (ConstInt64 c2)#(ConstInt64 c1)#ves' ⇒
            (s, vs, RSNormal (vs-to-es ((ConstInt32 (wasm-bool (app-relop-i iop
c1 c2)))#ves')))
          | - ⇒ (s, vs, crash-error))
    | $(Relop-i - iop) ⇒ (s, vs, crash-error)
    | $(Relop-f T-f32 fop) ⇒
        (case ves of
          (ConstFloat32 c2)#(ConstFloat32 c1)#ves' ⇒
            (s, vs, RSNormal (vs-to-es ((ConstInt32 (wasm-bool (app-relop-f fop
c1 c2)))#ves')))
          | - ⇒ (s, vs, crash-error))
    | $(Relop-f T-f64 fop) ⇒
        (case ves of
          (ConstFloat64 c2)#(ConstFloat64 c1)#ves' ⇒
            (s, vs, RSNormal (vs-to-es ((ConstInt32 (wasm-bool (app-relop-f fop
c1 c2)))#ves')))
          | - ⇒ (s, vs, crash-error))
    | $(Relop-f - fop) ⇒ (s, vs, crash-error)
    — CONVERT
    | $(Cvtop t2 Convert t1 sx) ⇒
        (case ves of
          v#ves' ⇒
            (if (types-agree t1 v)
              then
                expect (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))
          | - ⇒ (s, vs, crash-error))
      — UNREACHABLE
      | $Unreachable ⇒
          (s, vs, RSNormal ((vs-to-es ves)@[Trap]))
      — NOP
      | $Nop ⇒
          (s, vs, RSNormal (vs-to-es ves))
      — DROP
      | $Drop ⇒
          (case ves of
            v#ves' ⇒
              (s, vs, RSNormal (vs-to-es ves'))
          | - ⇒ (s, vs, crash-error))
      — SELECT
      | $Select ⇒
          (case ves of
            (ConstInt32 c)#v2#v1#ves' ⇒
              (if int-eq c 0 then (s, vs, RSNormal (vs-to-es (v2#ves'))) else (s, vs,
RSNormal (vs-to-es (v1#ves'))))
          | - ⇒ (s, vs, crash-error))
      — BLOCK
      | $(Block (t1s -> t2s) es) ⇒
          (if length ves ≥ length t1s
            then
              let (ves', ves'') = split-n ves (length t1s) in
                (s, vs, RSNormal ((vs-to-es ves'') @ [Label (length t2s) [] ((vs-to-es
ves')@($∗ es))]))
            else
              (s, vs, crash-error))
      — LOOP
      | $(Loop (t1s -> t2s) es) ⇒
          (if length ves ≥ length t1s
            then
              let (ves', ves'') = split-n ves (length t1s) in
                (s, vs, RSNormal ((vs-to-es ves'') @ [Label (length t1s) [$(Loop (t1s
-> t2s) es)] ((vs-to-es ves')@($∗ es))]))
            else
              (s, vs, crash-error))
      — IF
      | $(If tf es1 es2) ⇒
          (case ves of
            (ConstInt32 c)#ves' ⇒
              if int-eq c 0
                then
                  (s, vs, RSNormal ((vs-to-es ves')@[$(Block tf es2)]))
```

```
          else
            (s, vs, RSNormal ((vs-to-es ves')@[$(Block tf es1)]))
    | - ⇒ (s, vs, crash-error))
— BR
| $Br j ⇒
    (s, vs, RSBreak j ves)
— BR-IF
| $Br-if j ⇒
    (case ves of
      (ConstInt32 c)#ves' ⇒
        if int-eq c 0
          then
            (s, vs, RSNormal (vs-to-es ves'))
          else
            (s, vs, RSNormal ((vs-to-es ves') @ [$Br j]))
    | - ⇒ (s, vs, crash-error))
— BR-TABLE
| $Br-table js j ⇒
    (case ves of
      (ConstInt32 c)#ves' ⇒
      let k = nat-of-int c in
        if k < length js
          then
            (s, vs, RSNormal ((vs-to-es ves') @ [$Br (js!k)]))
          else
            (s, vs, RSNormal ((vs-to-es ves') @ [$Br j]))
    | - ⇒ (s, vs, crash-error))
— CALL
| $Call j ⇒
    (s, vs, RSNormal ((vs-to-es ves) @ [Callcl (sfunc s i j)]))
— CALL-INDIRECT
| $Call-indirect j ⇒
    (case ves of
      (ConstInt32 c)#ves' ⇒
        (case (stab s i (nat-of-int c)) of
          Some cl ⇒
            if (stypes s i j = cl-type cl)
              then
                (s, vs, RSNormal ((vs-to-es ves') @ [Callcl cl]))
              else
                (s, vs, RSNormal ((vs-to-es ves')@[Trap]))
        | - ⇒ (s, vs, RSNormal ((vs-to-es ves')@[Trap])))
    | - ⇒ (s, vs, crash-error))
— RETURN
| $Return ⇒
    (s, vs, RSReturn ves)
— GET-LOCAL
| $Get-local j ⇒
    (if j < length vs
```

```
        then (s, vs, RSNormal (vs-to-es ((vs!j)#ves)))
        else (s, vs, crash-error))
    — SET-LOCAL
    | $Set-local j ⇒
      (case ves of
        v#ves' ⇒
          if j < length vs
            then (s, vs[j := v], RSNormal (vs-to-es ves'))
            else (s, vs, crash-error)
      | - ⇒ (s, vs, crash-error))
    — TEE-LOCAL
    | $Tee-local j ⇒
      (case ves of
        v#ves' ⇒
          (s, vs, RSNormal ((vs-to-es (v#ves)) @ [$(Set-local j)]))
      | - ⇒ (s, vs, crash-error))
    — GET-GLOBAL
    | $Get-global j ⇒
      (s, vs, RSNormal (vs-to-es ((sglob-val s i j)#ves)))
    — SET-GLOBAL
    | $Set-global j ⇒
      (case ves of
        v#ves' ⇒ ((supdate-glob s i j v), vs, RSNormal (vs-to-es ves'))
      | - ⇒ (s, vs, crash-error))
    — LOAD
    | $(Load t None a off) ⇒
      (case ves of
        (ConstInt32 k)#ves' ⇒
          expect (smem-ind s i)
            (λj.
              expect (load ((mem s)!j) (nat-of-int k) off (t-length t))
              (λbs. (s, vs, RSNormal (vs-to-es ((wasm-deserialise bs t)#ves'))))
                (s, vs, RSNormal ((vs-to-es ves')@[Trap])))
            (s, vs, crash-error)
      | - ⇒ (s, vs, crash-error))
    — LOAD PACKED
    | $(Load t (Some (tp, sx)) a off) ⇒
      (case ves of
        (ConstInt32 k)#ves' ⇒
          expect (smem-ind s i)
            (λj.
              expect (load-packed sx ((mem s)!j) (nat-of-int k) off (tp-length tp)
(t-length t))
              (λbs. (s, vs, RSNormal (vs-to-es ((wasm-deserialise bs t)#ves'))))
                (s, vs, RSNormal ((vs-to-es ves')@[Trap])))
            (s, vs, crash-error)
      | - ⇒ (s, vs, crash-error))
    — STORE
    | $(Store t None a off) ⇒
```

```
        (case ves of
          v#(ConstInt32 k)#ves' ⇒
            (if (types-agree t v)
              then
                expect (smem-ind s i)
                  (λj.
                   expect (store ((mem s)!j) (nat-of-int k) off (bits v) (t-length t))
                       (λmem'. (s(|mem:= ((mem s)[j := mem']) |), vs, RSNormal
(vs-to-es ves')))
                       (s, vs, RSNormal ((vs-to-es ves')@[Trap])))
                  (s, vs, crash-error)
              else
                (s, vs, crash-error))
          | - ⇒ (s, vs, crash-error))
      — STORE-PACKED
      | $(Store t (Some tp) a off) ⇒
          (case ves of
                v#(ConstInt32 k)#ves' ⇒
                  (if (types-agree t v)
                    then
                      expect (smem-ind s i)
                        (λj.
                          expect (store-packed ((mem s)!j) (nat-of-int k) off (bits v)
(tp-length tp))
                          (λmem'. (s(|mem:= ((mem s)[j := mem']) |), vs, RSNormal
(vs-to-es ves')))
                              (s, vs, RSNormal ((vs-to-es ves')@[Trap])))
                        (s, vs, crash-error)
                    else
                      (s, vs, crash-error))
              | - ⇒ (s, vs, crash-error))
      — CURRENT-MEMORY
      | $Current-memory ⇒
          expect (smem-ind s i)
              (λj. (s, vs, RSNormal (vs-to-es ((ConstInt32 (int-of-nat (mem-size
((s.mem s)!j))))#ves))))
            (s, vs, crash-error)
      — GROW-MEMORY
      | $Grow-memory ⇒
          (case ves of
            (ConstInt32 c)#ves' ⇒
              expect (smem-ind s i)
                (λj.
                  let l = (mem-size ((s.mem s)!j)) in
                  (expect (mem-grow-impl ((mem s)!j) (nat-of-int c))
                       (λmem'. (s(|mem:= ((mem s)[j := mem']) |), vs, RSNormal
(vs-to-es ((ConstInt32 (int-of-nat l))#ves'))))
                (s, vs, RSNormal (vs-to-es ((ConstInt32 int32-minus-one)#ves')))))
                (s, vs, crash-error)
```

```
        | - ⇒ (s, vs, crash-error))
    — VAL - should not be executed
    | $C v ⇒ (s, vs, crash-error)
  — E
    — CALLCL
    | Callcl cl ⇒
      (case cl of
        Func-native i' (t1s -> t2s) ts es ⇒
          let n = length t1s in
          let m = length t2s in
          if length ves ≥ n
            then
              let (ves', ves'') = split-n ves n in
              let zs = n-zeros ts in
                (s, vs, RSNormal ((vs-to-es ves'') @ ([Local m i' ((rev ves')@zs)
[$(Block ([] -> t2s) es)])))
            else
              (s, vs, crash-error)
        | Func-host (t1s -> t2s) f ⇒
          let n = length t1s in
          let m = length t2s in
          if length ves ≥ n
            then
              let (ves', ves'') = split-n ves n in
              case host-apply-impl s (t1s -> t2s) f (rev ves') of
                Some (s',rves) ⇒
                  if list-all2 types-agree t2s rves
                    then
                      (s', vs, RSNormal ((vs-to-es ves'') @ ($$* rves)))
                    else
                      (s', vs, crash-error)
              | None ⇒ (s, vs, RSNormal ((vs-to-es ves'')@[Trap]))
            else
              (s, vs, crash-error))
    — LABEL
    | Label ln les es ⇒
      if es-is-trap es
        then
          (s, vs, RSNormal ((vs-to-es ves)@[Trap]))
        else
          (if (const-list es)
            then
              (s, vs, RSNormal ((vs-to-es ves)@es))
            else
              let (s', vs', res) = run-step d i (s, vs, es) in
              (case res of
                RSBreak 0 bvs ⇒
                  if (length bvs ≥ ln)
                  then (s', vs', RSNormal ((vs-to-es ((take ln bvs)@ves))@les))
```

```
                    else (s', vs', crash-error)
                | RSBreak (Suc n) bvs ⇒
                    (s', vs', RSBreak n bvs)
                | RSReturn rvs ⇒
                    (s', vs', RSReturn rvs)
                | RSNormal es' ⇒
                    (s', vs', RSNormal ((vs-to-es ves)@[Label ln les es']))
                | - ⇒ (s', vs', crash-error)))
    — LOCAL
    | Local ln j vls es ⇒
        if es-is-trap es
          then
            (s, vs, RSNormal ((vs-to-es ves)@[Trap]))
          else
            (if (const-list es)
              then
                if (length es = ln)
                  then (s, vs, RSNormal ((vs-to-es ves)@es))
                  else (s, vs, crash-error)
              else
                case d of
                  0 ⇒ (s, vs, crash-error)
                | Suc d' ⇒
                    let (s', vls', res) = run-step d' j (s, vls, es) in
                    (case res of
                       RSReturn rvs ⇒
                         if (length rvs ≥ ln)
                           then (s', vs, RSNormal (vs-to-es ((take ln rvs)@ves)))
                           else (s', vs, crash-error)
                     | RSNormal es' ⇒
                         (s', vs, RSNormal ((vs-to-es ves)@[Local ln j vls' es']))
                     | - ⇒ (s', vs, RSCrash CExhaustion)))
    — TRAP - should not be executed
    | Trap ⇒ (s, vs, crash-error))
  ⟨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)
                                              | - ⇒ (s, RCrash CError)))
| run-v 0 d i (s,vs,es) = (s, RCrash CExhaustion)
```

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*)
⟨*proof*⟩

lemma *not-const-vs-to-es-list*:
assumes ~(*is-const e*)
shows *vs1* @ [*e*] @ *vs2* ≠ $$∗ *vs*
⟨*proof*⟩

lemma *neq-label-nested*:[*Label n les es*] ≠ *es*
⟨*proof*⟩

lemma *neq-local-nested*:[*Local n i lvs es*] ≠ *es*
⟨*proof*⟩

lemma *trap-not-value*:[*Trap*] ≠ $$∗*es*
⟨*proof*⟩

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

lemma *lfilled-single*:
assumes *Lfilled k lholed es* [*e*]
⋀ *a b c*. *e* ≠ *Label a b c*
shows (*es* = [*e*] ∧ *lholed* = *LBase* [] []) ∨ *es* = []
⟨*proof*⟩

lemma *lfilled-eq*:
assumes *Lfilled j lholed es LI*
Lfilled j lholed es′ LI
shows *es* = *es′*
⟨*proof*⟩

lemma *lfilled-size*:
assumes *Lfilled j lholed es LI*
shows *size-list size LI* ≥ *size-list size es*
⟨*proof*⟩

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

lemma *reduce-simple-not-eq*:
assumes (|*es*|) ⇝ (|*es′*|)
shows *es* ≠ *es′*

⟨*proof*⟩

lemma *reduce-not-eq*:
assumes (|s;vs;es|) ⇝-i (|s';vs';es'|)
shows $es \neq es'$
⟨*proof*⟩

lemma *reduce-simple-not-value*:
assumes (|es|) ⇝ (|es'|)
shows $es \neq \$\$* \; vs$
⟨*proof*⟩

lemma *reduce-not-value*:
assumes (|s;vs;es|) ⇝-i (|s';vs';es'|)
shows $es \neq \$\$* \; ves$
⟨*proof*⟩

lemma *reduce-simple-not-nil*:
assumes (|es|) ⇝ (|es'|)
shows $es \neq []$
⟨*proof*⟩

lemma *reduce-not-nil*:
assumes (|s;vs;es|) ⇝-i (|s';vs';es'|)
shows $es \neq []$
⟨*proof*⟩

lemma *reduce-simple-not-trap*:
assumes (|es|) ⇝ (|es'|)
shows $es \neq [Trap]$
⟨*proof*⟩

lemma *reduce-not-trap*:
assumes (|s;vs;es|) ⇝-i (|s';vs';es'|)
shows $es \neq [Trap]$
⟨*proof*⟩

lemma *reduce-simple-call*: ¬(|[$\$Call \; j$]|) ⇝ (|$es'$|)
⟨*proof*⟩

lemma *reduce-call*:
assumes (|s;vs;[$\$Call \; j$]|) ⇝- i (|s';vs';es'|)
shows $s = s'$
$vs = vs'$
$es' = [Callcl \; (sfunc \; s \; i \; j)]$
⟨*proof*⟩

lemma *run-one-step-basic-unreachable-result*:
assumes *run-one-step* $d \; i \; (s,vs,ves,\$Unreachable) = (s', \; vs', \; res)$

shows $\exists\, r.\ res = RSNormal\ r$
⟨*proof*⟩

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

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

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

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

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

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

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

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

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 $(\exists\, r.\ res = RSNormal\ r) \vee (\exists\, e.\ res = RSCrash\ e)$
⟨*proof*⟩

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

assumes *run-one-step d i* (*s*,*vs*,*ves*,$*Return*) = (*s'*, *vs'*, *res*)
shows ∃ *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 ∃ *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 (∃ *r*. *res* = *RSNormal r*) ∨ (∃ *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 (∃ *r*. *res* = *RSNormal r*) ∨ (∃ *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 (∃ *r*. *res* = *RSNormal r*) ∨ (∃ *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 (∃ *r*. *res* = *RSNormal r*) ∨ (∃ *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 (∃ *r*. *res* = *RSNormal r*) ∨ (∃ *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 (∃ *r*. *res* = *RSNormal r*) ∨ (∃ *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 (∃ *r*. *res* = *RSNormal r*) ∨ (∃ *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 (∃ *r*. *res* = *RSNormal r*) ∨ (∃ *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 (∃ *r*. *res* = *RSNormal r*) ∨ (∃ *e*. *res* = *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 (∃ *r*. *res* = *RSNormal r*) ∨ (∃ *e*. *res* = *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 ∃ *e*. *res* = *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 (∃ *r*. *res* = *RSNormal r*) ∨ (∃ *e*. *res* = *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 (∃ *r*. *res* = *RSNormal r*) ∨ (∃ *e*. *res* = *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 (∃ *r*. *res* = *RSNormal r*) ∨ (∃ *e*. *res* = *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 (∃ *r*. *res* = *RSNormal r*) ∨ (∃ *e*. *res* = *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 (∃ *r*. *res* = *RSNormal r*) ∨ (∃ *e*. *res* = *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 (∃ *r*. *res* = *RSNormal r*) ∨ (∃ *e*. *res* = *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 (∃ *r*. *res* = *RSNormal r*) ∨ (∃ *e*. *res* = *RSCrash e*)
⟨*proof*⟩

lemma *run-one-step-basic-cvtop-result*:
assumes *run-one-step d i* (*s*,*vs*,*ves*,*$Cvtop t2 x312 t1 sx*) = (*s'*, *vs'*, *res*)
shows (∃ *r*. *res* = *RSNormal r*) ∨ (∃ *e*. *res* = *RSCrash e*)
⟨*proof*⟩

lemma *run-one-step-trap-result*:
assumes *run-one-step d i* (*s*,*vs*,*ves*,*Trap*) = (*s'*, *vs'*, *res*)
shows ∃ *e*. *res* = *RSCrash e*
⟨*proof*⟩

lemma *run-one-step-callcl-result*:
assumes *run-one-step d i* (*s*,*vs*,*ves*,*Callcl cl*) = (*s'*, *vs'*, *res*)
shows (∃ *r*. *res* = *RSNormal r*) ∨ (∃ *e*. *res* = *RSCrash e*)
⟨*proof*⟩

lemma *run-one-step-label-result*:
assumes *run-one-step d i* (*s*,*vs*,*ves*,*Label x41 x42 x43*) = (*s'*, *vs'*, *res*)
shows (∃ *r*. *res* = *RSNormal r*) ∨ (∃ *r rvs*. *res* = *RSBreak r rvs*) ∨ (∃ *rvs*. *res* = *RSReturn rvs*) ∨ (∃ *e*. *res* = *RSCrash e*)
⟨*proof*⟩

lemma *run-one-step-local-result*:
assumes *run-one-step d i* (*s*,*vs*,*ves*,*Local x51 x52 x53 x54*) = (*s'*, *vs'*, *res*)
shows (∃ *r*. *res* = *RSNormal r*) ∨ (∃ *e*. *res* = *RSCrash e*)
⟨*proof*⟩

lemma *run-one-step-break*:
assumes *run-one-step d i* (*s*,*vs*,*ves*,*e*) = (*s'*, *vs'*, *RSBreak n res*)
shows (*e* = *$Br n*) ∨ (∃ *n les es*. *e* = *Label n les es*)
⟨*proof*⟩

lemma *run-one-step-return*:
assumes *run-one-step d i* (*s*,*vs*,*ves*,*e*) = (*s'*, *vs'*, *RSReturn res*)
shows (*e* = *$Return*) ∨ (∃ *n les es*. *e* = *Label n les es*)
⟨*proof*⟩

lemma *run-step-break-imp-not-trap-const-list*:
assumes *run-step d i* (*s*, *vs*, *es*) = (*s'*, *vs'*, *RSBreak n res*)
shows *es* ≠ [*Trap*] ¬*const-list es*
⟨*proof*⟩

lemma *run-step-return-imp-not-trap-const-list*:
assumes *run-step d i* (*s*, *vs*, *es*) = (*s'*, *vs'*, *RSReturn res*)
shows *es* ≠ [*Trap*] ¬*const-list es*
⟨*proof*⟩

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 =
(λ*d i* (*s*,*vs*,*es*). (*run-step d i* (*s*,*vs*,*es*) = (*s′*, *vs′*, *RSBreak n res*)) ⟶
s = *s′* ∧ *vs* = *vs′* ∧
(∃ *n′ lfilled es-c*. *n′* ≥ *n* ∧ *Lfilled-exact* (*n′*−*n*) *lfilled* ((*vs-to-es res*) @ [$*Br n′*] @ *es-c*) *es*))

definition *run-one-step-break-imp-lfilled-prop* **where**
run-one-step-break-imp-lfilled-prop s′ vs′ n res =
(λ*d i* (*s*,*vs*,*ves*,*e*). *run-one-step d i* (*s*,*vs*,*ves*,*e*) = (*s′*, *vs′*, *RSBreak n res*) ⟶
s = *s′* ∧ *vs* = *vs′* ∧ ((*res* = *ves* ∧ *e* = $*Br n*) ∨ (∃ *n′ lfilled es-c es les′ ln*. *n′* > *n* ∧ *Lfilled-exact* (*n′*−(*n*+*1*)) *lfilled* ((*vs-to-es res*) @ [$*Br n′*] @ *es-c*) *es* ∧ *e* = *Label ln les′ es*)))

lemma *run-step-break-imp-lfilled*:
assumes *run-step d i* (*s*,*vs*,*es*) = (*s′*, *vs′*, *RSBreak n res*)
shows *s* = *s′* ∧
vs = *vs′* ∧
(∃ *n′ lfilled es-c*. *n′* ≥ *n* ∧
Lfilled-exact (*n′*−*n*) *lfilled* ((*vs-to-es res*) @ [$*Br n′*] @ *es-c*) *es*)
⟨*proof*⟩

lemma *run-step-return-imp-lfilled*:
assumes *run-step d i* (*s*,*vs*,*es*) = (*s′*, *vs′*, *RSReturn res*)
shows *s* = *s′* ∧ *vs* = *vs′* ∧ (∃ *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*,$*b-e*) = (*s′*, *vs′*, *RSNormal es′*))
b-e = *Unop-i t iop* ∨ *b-e* = *Unop-f t fop* ∨ *b-e* = *Testop t testop*
shows (|*s*;*vs*;(*vs-to-es ves*)@[$*b-e*]|) ⇝- *i* (|*s′*;*vs′*;*es′*|)
⟨*proof*⟩

lemma *run-step-basic-binop-relop-sound*:
assumes (*run-one-step d i* (*s*,*vs*,*ves*,$*b-e*) = (*s′*, *vs′*, *RSNormal es′*))
b-e = *Binop-i t iop* ∨ *b-e* = *Binop-f t fop* ∨ *b-e* = *Relop-i t irop* ∨ *b-e* = *Relop-f t frop*

shows $(\!|s;vs;(\textit{vs-to-es ves})@[\$b\text{-}e]|\!) \rightsquigarrow^{*} i\ (\!|s';vs';es'|\!)$
⟨*proof*⟩

lemma *run-step-basic-sound*:
assumes (*run-one-step d i* (*s*,*vs*,*ves*,$*b*-*e*) = (*s'*, *vs'*, *RSNormal es'*))
shows $(\!|s;vs;(\textit{vs-to-es ves})@[\$b\text{-}e]|\!) \rightsquigarrow^{*} i\ (\!|s';vs';es'|\!)$
⟨*proof*⟩

theorem *run-step-sound*:
assumes *run-step d i* (*s*,*vs*,*es*) = (*s'*, *vs'*, *RSNormal es'*)
shows $(\!|s;vs;es|\!) \rightsquigarrow^{*} i\ (\!|s';vs';es'|\!)$
⟨*proof*⟩

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.