CakeML

Lars Hupel, Yu Zhang

March 17, 2025

Abstract

CakeML is a functional programming language with a proven-correct compiler and runtime system. This entry contains an unofficial version of the CakeML semantics that has been exported from the Lem specifications to Isabelle. Additionally, there are some hand-written theory files that adapt the exported code to Isabelle and port proofs from the HOL4 formalization, e.g. termination and equivalence proofs.

Contents

Contributors

The export script has been written by Lars Hupel. Hand-written theory files, including definitions and proofs, have been developed by Lars Hupel and Yu Zhang.

Lem is a project by Peter Sewell et.al. Contributors can be found on its project page[1] and on GitHub.[2]

CakeML is a project with many developers and contributors that can be found on its project page[3] and on GitHub.[4]

In particular, Fabian Immler and Johannes Åman Pohjola have contributed Isabelle mappings for constants in the Lem specification of the CakeML semantics.

[1] https://www.cl.cam.ac.uk/~pes20/lem/
[2] https://github.com/rems-project/lem/graphs/contributors
[3] https://cakeml.org/
[4] https://github.com/CakeML/cakeml/graphs/contributors

Part I

Generated Isabelle code

Chapter 1

Generated by Lem from *misc/lem-lib-stub/lib.lem*.

theory *Lib*

imports
Main
HOL−Library.Datatype-Records
LEM.Lem-pervasives
LEM.Lem-list-extra
LEM.Lem-string
Coinductive.Coinductive-List

begin

— *Extensions to Lem′s built−in library to target things we need in HOL.*
— *open import Pervasives*
— *import List-extra*
— *import String*

— *TODO: look for these in the built−in library*

— *val rtc : forall ′a. (′a −> ′a −> bool) −> (′a −> ′a −> bool)*

— *val disjoint : forall ′a. set ′a −> set ′a −> bool*

— *val all2 : forall ′a ′b. (′a −> ′b −> bool) −> list ′a −> list ′b −> bool*

fun *the* :: *′a ⇒ ′a option ⇒ ′a* **where**
the - (Some x) = (x) | the x None = (x)

— *val fapply : forall ′a ′b. MapKeyType ′b => ′a −> ′b −> Map.map ′b ′a −> ′a*
definition *fapply* :: *′a ⇒ ′b ⇒(′b,′a)Map.map ⇒ ′a* **where**
fapply d x f = ((case f x of Some d => d | None => d))

function (*sequential*,*domintros*)
lunion :: *'a list* ⇒ *'a list* ⇒ *'a list* **where**

lunion [] *s* = (*s*)
|
lunion (*x* # *xs*) *s* = (
if Set.member x (*set s*)
then lunion xs s
else x #(*lunion xs s*))
by *pat-completeness auto*

— *TODO*: *proper support for nat sets as sptrees...*
— *open import* {*hol*} *'sptreeTheory'*
— *type nat-set*
— *val nat-set-empty* : *nat-set*
— *val nat-set-is-empty* : *nat-set* −> *bool*
— *val nat-set-insert* : *nat-set* −> *nat* −> *nat-set*
— *val nat-set-delete* : *nat-set* −> *nat* −> *nat-set*
— *val nat-set-to-set* : *nat-set* −> *set nat*
— *val nat-set-elem* : *nat-set* −> *nat* −> *bool*
— *val nat-set-from-list* : *list nat* −> *nat-set*
— *val nat-set-upto* : *nat* −> *nat-set*

— *type nat-map 'a*
— *val nat-map-empty* : *forall 'a. nat-map 'a*
— *val nat-map-domain* : *forall 'a. nat-map 'a* −> *set nat*
— *val nat-map-insert* : *forall 'a. nat-map 'a* −> *nat* −> *'a* −> *nat-map 'a*
— *val nat-map-lookup* : *forall 'a. nat-map 'a* −> *nat* −> *maybe 'a*
— *val nat-map-to-list* : *forall 'a. nat-map 'a* −> *list 'a*
— *val nat-map-map* : *forall 'a 'b.* (*'a* −> *'b*) −> *nat-map 'a* −> *nat-map 'b*

— *TODO*: *proper support for lazy lists*

— *open import* {*hol*} *'llistTheory'*
— *open import* {*isabelle*} *'Coinductive.Coinductive-List'*
— *type llist 'a*
— *val lhd* : *forall 'a. llist 'a* −> *maybe 'a*
— *val ltl* : *forall 'a. llist 'a* −> *maybe* (*llist 'a*)
— *val lnil* : *forall 'a. llist 'a*
— *val lcons* : *forall 'a. 'a* −> *llist 'a* −> *llist 'a*

— *TODO*: *proper support for words...*
— *open import* {*hol*} *'wordsTheory' 'integer-wordTheory'*
— *type word8*
— *val natFromWord8* : *word8* −> *nat*

— *val word-to-hex-string* : *word8* −> *string*
— *val word8FromNat* : *nat* −> *word8*
— *val word8FromInteger* : *integer* −> *word8*
— *val integerFromWord8* : *word8* −> *integer*
— *type word64*
— *val natFromWord64* : *word64* −> *nat*
— *val word64FromNat* : *nat* −> *word64*
— *val word64FromInteger* : *integer* −> *word64*
— *val integerFromWord64* : *word64* −> *integer*
— *val word8FromWord64* : *word64* −> *word8*
— *val word64FromWord8* : *word8* −> *word64*
— *val W8and* : *word8* −> *word8* −> *word8*
— *val W8or* : *word8* −> *word8* −> *word8*
— *val W8xor* : *word8* −> *word8* −> *word8*
— *val W8add* : *word8* −> *word8* −> *word8*
— *val W8sub* : *word8* −> *word8* −> *word8*
— *val W64and* : *word64* −> *word64* −> *word64*
— *val W64or* : *word64* −> *word64* −> *word64*
— *val W64xor* : *word64* −> *word64* −> *word64*
— *val W64add* : *word64* −> *word64* −> *word64*
— *val W64sub* : *word64* −> *word64* −> *word64*

— *val W8lsl* : *word8* −> *nat* −> *word8*
— *val W8lsr* : *word8* −> *nat* −> *word8*
— *val W8asr* : *word8* −> *nat* −> *word8*
— *val W8ror* : *word8* −> *nat* −> *word8*
— *val W64lsl* : *word64* −> *nat* −> *word64*
— *val W64lsr* : *word64* −> *nat* −> *word64*
— *val W64asr* : *word64* −> *nat* −> *word64*
— *val W64ror* : *word64* −> *nat* −> *word64*

— *open import* {*hol*} ‘*alistTheory*‘
type-synonym(′*a*, ′*b*) *alist* = (′*a* ∗ ′*b*) *list*
— *val alistToFmap* : *forall* ′*k* ′*v*. *alist* ′*k* ′*v* −> *Map.map* ′*k* ′*v*

— *val opt-bind* : *forall* ′*a* ′*b*. *maybe* ′*a* −> ′*b* −> *alist* ′*a* ′*b* −> *alist* ′*a* ′*b*
fun *opt-bind* :: ′*a option* ⇒ ′*b* ⇒(′*a*∗′*b*)*list* ⇒(′*a*∗′*b*)*list* **where**
  opt-bind None v2 e = (*e*)
| *opt-bind* (*Some n*′) *v2 e* = ((*n*′,*v2*)# *e*)

— *Lists of indices*

fun
lshift :: *nat* ⇒(*nat*)*list* ⇒(*nat*)*list* **where**

lshift (*n* :: *nat*) *ls* = (
  List.map (λ *v2* . *v2* − *n*) (*List.filter* (λ *v2* . *n* ≤ *v2*) *ls*))

— *open import* {*hol*} *'locationTheory'*
datatype-record *locn* =
row :: *nat*
col :: *nat*
offset :: *nat*

type-synonym *locs* = (*locn* ∗ *locn*)
— *val unknown-loc* : *locs*
end

Chapter 2

Generated by Lem from *semantics/namespace.lem*.

```
theory Namespace

imports
  Main
  HOL−Library.Datatype-Records
  LEM.Lem-pervasives
  LEM.Lem-set-extra

begin

— open import Pervasives
— open import Set-extra

type-synonym( 'k, 'v) alist0 = ('k ∗ 'v) list

— Identifiers
datatype( 'm, 'n) id0 =
    Short 'n
  | Long 'm   ('m, 'n) id0

— val mk-id : forall 'n 'm. list 'm −> 'n −> id 'm 'n
fun mk-id :: 'm list ⇒ 'n ⇒('m,'n)id0  where
      mk-id [] n = ( Short n )
   | mk-id (mn # mns) n = ( Long mn (mk-id mns n))


— val id-to-n : forall 'n 'm. id 'm 'n −> 'n
fun id-to-n :: ('m,'n)id0 ⇒ 'n  where
      id-to-n (Short n) = ( n )
   | id-to-n (Long - id1) = ( id-to-n id1 )
```

```
— val id-to-mods : forall 'n 'm. id 'm 'n −> list 'm
fun id-to-mods :: ('m,'n)id0 ⇒ 'm list where
    id-to-mods (Short -) = ( [])
  | id-to-mods (Long mn id1) = ( mn # id-to-mods id1 )


datatype( 'm, 'n, 'v) namespace =
  Bind ('n, 'v) alist0 ('m, ( ('m, 'n, 'v)namespace)) alist0

— val nsLookup : forall 'v 'm 'n. Eq 'n, Eq 'm => namespace 'm 'n 'v −> id 'm
'n −> maybe 'v
fun nsLookup :: ('m,'n,'v)namespace ⇒('m,'n)id0 ⇒ 'v option where
    nsLookup (Bind v2 m) (Short n) = ( Map.map-of v2 n )
  | nsLookup (Bind v2 m) (Long mn id1) = (
    (case Map.map-of m mn of
      None => None
    | Some env => nsLookup env id1
    ))


— val nsLookupMod : forall 'm 'n 'v. Eq 'n, Eq 'm => namespace 'm 'n 'v −>
list 'm −> maybe (namespace 'm 'n 'v)
fun nsLookupMod :: ('m,'n,'v)namespace ⇒ 'm list ⇒(('m,'n,'v)namespace)option
where
    nsLookupMod e [] = ( Some e )
  | nsLookupMod (Bind v2 m) (mn # path) = (
    (case Map.map-of m mn of
      None => None
    | Some env => nsLookupMod env path
    ))


— val nsEmpty : forall 'v 'm 'n. namespace 'm 'n 'v
definition nsEmpty :: ('m,'n,'v)namespace where
    nsEmpty = ( Bind [] [])


— val nsAppend : forall 'v 'm 'n. namespace 'm 'n 'v −> namespace 'm 'n 'v −>
namespace 'm 'n 'v
fun nsAppend :: ('m,'n,'v)namespace ⇒('m,'n,'v)namespace ⇒('m,'n,'v)namespace
where
    nsAppend (Bind v1 m1) (Bind v2 m2) = ( Bind (v1 @ v2) (m1 @ m2))


— val nsLift : forall 'v 'm 'n. 'm −> namespace 'm 'n 'v −> namespace 'm 'n 'v
definition nsLift :: 'm ⇒('m,'n,'v)namespace ⇒('m,'n,'v)namespace where
    nsLift mn env = ( Bind [] [(mn, env)])
```

— *val alist-to-ns : forall 'v 'm 'n. alist 'n 'v –> namespace 'm 'n 'v*
definition *alist-to-ns* :: *('n∗'v)list* ⇒*('m,'n,'v)namespace* **where**
alist-to-ns a = (*Bind a* [])

— *val nsBind : forall 'v 'm 'n. 'n –> 'v –> namespace 'm 'n 'v –> namespace 'm 'n 'v*
fun *nsBind* :: *'n* ⇒ *'v* ⇒*('m,'n,'v)namespace* ⇒*('m,'n,'v)namespace* **where**
nsBind k x (*Bind v2 m*) = (*Bind* ((*k*,*x*)# *v2*) *m*)

— *val nsBindList : forall 'v 'm 'n. list ('n ∗ 'v) –> namespace 'm 'n 'v –> namespace 'm 'n 'v*
definition *nsBindList* :: *('n∗'v)list* ⇒*('m,'n,'v)namespace* ⇒*('m,'n,'v)namespace* **where**
nsBindList l e = (*List.foldr* (*λx* .
(*case x of* (*x*,*v2*) => *λ e* . *nsBind x v2 e*)) *l e*)

— *val nsOptBind : forall 'v 'm 'n. maybe 'n –> 'v –> namespace 'm 'n 'v –> namespace 'm 'n 'v*
fun *nsOptBind* :: *'n option* ⇒ *'v* ⇒*('m,'n,'v)namespace* ⇒*('m,'n,'v)namespace* **where**
nsOptBind None x env = (*env*)
| *nsOptBind* (*Some n'*) *x env* = (*nsBind n' x env*)

— *val nsSing : forall 'v 'm 'n. 'n –> 'v –> namespace 'm 'n 'v*
definition *nsSing* :: *'n* ⇒ *'v* ⇒*('m,'n,'v)namespace* **where**
nsSing n x = (*Bind* ([(*n*,*x*)]) [])

— *val nsSub : forall 'v1 'v2 'm 'n. Eq 'm, Eq 'n, Eq 'v1, Eq 'v2 => (id 'm 'n –> 'v1 –> 'v2 –> bool) –> namespace 'm 'n 'v1 –> namespace 'm 'n 'v2 –> bool*
definition *nsSub* :: ((*'m,'n)id0* ⇒ *'v1* ⇒ *'v2* ⇒ *bool*)⇒(*'m,'n,'v1)namespace* ⇒(*'m,'n,'v2)namespace* ⇒ *bool* **where**
nsSub r env1 env2 = (
((∀ *id0*. ∀ *v1*.
(*nsLookup env1 id0* = *Some v1*)
⟶
((∃ *v2*. (*nsLookup env2 id0* = *Some v2*) ∧ *r id0 v1 v2*))))
∧
((∀ *path*.
(*nsLookupMod env2 path* = *None*) ⟶ (*nsLookupMod env1 path* = *None*))))

— *val nsAll : forall 'v 'm 'n. Eq 'm, Eq 'n, Eq 'v => (id 'm 'n –> 'v –> bool) –> namespace 'm 'n 'v –> bool*

```
fun nsAll :: (('m,'n)id0 ⇒ 'v ⇒ bool)⇒('m,'n,'v)namespace ⇒ bool   where
      nsAll f env = (
  ((∀ id0. ∀ v1.
     (nsLookup env id0 = Some v1)
     ⟶
     f id0 v1)))
```

— *val eAll2 : forall 'v1 'v2 'm 'n. Eq 'm, Eq 'n, Eq 'v1, Eq 'v2 => (id 'm 'n -> 'v1 -> 'v2 -> bool) -> namespace 'm 'n 'v1 -> namespace 'm 'n 'v2 -> bool*

```
definition nsAll2 :: (('d,'c)id0 ⇒ 'b ⇒ 'a ⇒ bool)⇒('d,'c,'b)namespace ⇒('d,'c,'a)namespace
⇒ bool   where
      nsAll2 r env1 env2 = (
  nsSub r env1 env2 ∧
  nsSub (λ x y z .  r x z y) env2 env1 )
```

— *val nsDom : forall 'v 'm 'n. Eq 'm, Eq 'n, Eq 'v, SetType 'v => namespace 'm 'n 'v -> set (id 'm 'n)*

```
definition nsDom :: ('m,'n,'v)namespace ⇒(('m,'n)id0)set   where
      nsDom env = ( (let x2 =
  ({}) in  Finite-Set.fold
   (λv2 x2 .  Finite-Set.fold
                    (λn x2 .
                     if nsLookup env n = Some v2 then Set.insert n x2
                     else x2) x2 UNIV) x2 UNIV))
```

— *val nsDomMod : forall 'v 'm 'n. SetType 'm, Eq 'm, Eq 'n, Eq 'v => namespace 'm 'n 'v -> set (list 'm)*

```
definition nsDomMod :: ('m,'n,'v)namespace ⇒('m list)set   where
      nsDomMod env = ( (let x2 =
  ({}) in  Finite-Set.fold
   (λv2 x2 .  Finite-Set.fold
                    (λn x2 .
                     if nsLookupMod env n = Some v2 then Set.insert n x2
                     else x2) x2 UNIV) x2 UNIV))
```

— *val nsMap : forall 'v 'w 'm 'n. ('v -> 'w) -> namespace 'm 'n 'v -> namespace 'm 'n 'w*

```
function (sequential,domintros) nsMap :: ('v ⇒ 'w)⇒('m,'n,'v)namespace ⇒('m,'n,'w)namespace
where
      nsMap f (Bind v2 m) = (
  Bind (List.map ( λx .
  (case  x of (n,x) => (n, f x) )) v2)
      (List.map ( λx .
  (case  x of (mn,e) => (mn, nsMap f e) )) m))
by pat-completeness auto
```

end

Chapter 3

Generated by Lem from *semantics/fpSem.lem*.

theory *FpSem*

imports
Main
HOL−Library.Datatype-Records
LEM.Lem-pervasives
Lib
IEEE-Floating-Point.FP64

begin

— *open import Pervasives*
— *open import Lib*

— *open import {hol} ‘machine-ieeeTheory‘*
— *open import {isabelle} ‘IEEE-Floating-Point.FP64‘*

— *type rounding*

datatype *fp-cmp-op* = *FP-Less* | *FP-LessEqual* | *FP-Greater* | *FP-GreaterEqual*
| *FP-Equal*
datatype *fp-uop-op* = *FP-Abs* | *FP-Neg* | *FP-Sqrt*
datatype *fp-bop-op* = *FP-Add* | *FP-Sub* | *FP-Mul* | *FP-Div*

— *val fp64-lessThan : word64 −> word64 −> bool*
— *val fp64-lessEqual : word64 −> word64 −> bool*
— *val fp64-greaterThan : word64 −> word64 −> bool*
— *val fp64-greaterEqual : word64 −> word64 −> bool*
— *val fp64-equal : word64 −> word64 −> bool*

— *val fp64-abs : word64 −> word64*
— *val fp64-negate : word64 −> word64*

— *val fp64-sqrt : rounding −> word64 −> word64*

— *val fp64-add : rounding −> word64 −> word64 −> word64*
— *val fp64-sub : rounding −> word64 −> word64 −> word64*
— *val fp64-mul : rounding −> word64 −> word64 −> word64*
— *val fp64-div : rounding −> word64 −> word64 −> word64*

— *val roundTiesToEven : rounding*

— *val fp-cmp : fp-cmp −> word64 −> word64 −> bool*
fun *fp-cmp* :: *fp-cmp-op* ⇒ *64 word* ⇒ *64 word* ⇒ *bool* **where**
    fp-cmp FP-Less = (*fp64-lessThan*)
| *fp-cmp FP-LessEqual* = (*fp64-lessEqual*)
| *fp-cmp FP-Greater* = (*fp64-greaterThan*)
| *fp-cmp FP-GreaterEqual* = (*fp64-greaterEqual*)
| *fp-cmp FP-Equal* = (*fp64-equal*)

— *val fp-uop : fp-uop −> word64 −> word64*
fun *fp-uop* :: *fp-uop-op* ⇒ *64 word* ⇒ *64 word* **where**
    fp-uop FP-Abs = (*fp64-abs*)
| *fp-uop FP-Neg* = (*fp64-negate*)
| *fp-uop FP-Sqrt* = (*fp64-sqrt RNE*)

— *val fp-bop : fp-bop −> word64 −> word64 −> word64*
fun *fp-bop* :: *fp-bop-op* ⇒ *64 word* ⇒ *64 word* ⇒ *64 word* **where**
    fp-bop FP-Add = (*fp64-add RNE*)
| *fp-bop FP-Sub* = (*fp64-sub RNE*)
| *fp-bop FP-Mul* = (*fp64-mul RNE*)
| *fp-bop FP-Div* = (*fp64-div RNE*)

end

Chapter 4

Generated by Lem from *semantics/ast.lem*.

theory *Ast*

imports
Main
HOL−Library.Datatype-Records
LEM.Lem-pervasives
Lib
Namespace
FpSem

begin

— *open import Pervasives*
— *open import Lib*
— *open import Namespace*
— *open import FpSem*

— *Literal constants*
datatype *lit* =
IntLit int
| *Char char*
| *StrLit string*
| *Word8 8 word*
| *Word64 64 word*

— *Built−in binary operations*
datatype *opn* = *Plus* | *Minus* | *Times* | *Divide* | *Modulo*
datatype *opb* = *Lt* | *Gt* | *Leq* | *Geq*
datatype *opw* = *Andw* | *Orw* | *Xor* | *Add* | *Sub*
datatype *shift* = *Lsl* | *Lsr* | *Asr* | *Ror*

— *Module names*

type-synonym *modN* = *string*

— *Variable names*
type-synonym *varN* = *string*

— *Constructor names* (*from datatype definitions*)
type-synonym *conN* = *string*

— *Type names*
type-synonym *typeN* = *string*

— *Type variable names*
type-synonym *tvarN* = *string*

datatype *word-size* = *W8* | *W64*

datatype *op0* =
 — *Operations on integers*
  Opn opn
 | *Opb opb*
 — *Operations on words*
 | *Opw word-size opw*
 | *Shift word-size shift nat*
 | *Equality*
 — *FP operations*
 | *FP-cmp fp-cmp-op*
 | *FP-uop fp-uop-op*
 | *FP-bop fp-bop-op*
 — *Function application*
 | *Opapp*
 — *Reference operations*
 | *Opassign*
 | *Opref*
 | *Opderef*
 — *Word8Array operations*
 | *Aw8alloc*
 | *Aw8sub*
 | *Aw8length*
 | *Aw8update*
 — *Word/integer conversions*
 | *WordFromInt word-size*
 | *WordToInt word-size*
 — *string/bytearray conversions*
 | *CopyStrStr*
 | *CopyStrAw8*
 | *CopyAw8Str*
 | *CopyAw8Aw8*
 — *Char operations*
 | *Ord*

| *Chr*
| *Chopb opb*
— *String operations*
| *Implode*
| *Strsub*
| *Strlen*
| *Strcat*
— *Vector operations*
| *VfromList*
| *Vsub*
| *Vlength*
— *Array operations*
| *Aalloc*
| *AallocEmpty*
| *Asub*
| *Alength*
| *Aupdate*
— *Configure the GC*
| *ConfigGC*
— *Call a given foreign function*
| *FFI string*

— *Logical operations*
datatype *lop* =
And
| *Or*

— *Type constructors. ∗ 0−ary type applications represent unparameterised types (e.g., num or string)*
datatype *tctor* =
— *User defined types*
TC-name (modN, typeN) id0
— *Built−in types*
| *TC-int*
| *TC-char*
| *TC-string*
| *TC-ref*
| *TC-word8*
| *TC-word64*
| *TC-word8array*
| *TC-fn*
| *TC-tup*
| *TC-exn*
| *TC-vector*
| *TC-array*

— *Types*
datatype *t* =
— *Type variables that the user writes down ('a, 'b, etc.)*

Tvar tvarN
— *deBruijn indexed type variables. The type system uses these internally.*
| *Tvar-db nat*
| *Tapp t list tctor*

— *Some abbreviations*
definition *Tint* :: *t* **where**
Tint = (*Tapp* [] *TC-int*)

definition *Tchar* :: *t* **where**
Tchar = (*Tapp* [] *TC-char*)

definition *Tstring* :: *t* **where**
Tstring = (*Tapp* [] *TC-string*)

definition *Tref* :: *t* ⇒ *t* **where**
Tref t1 = (*Tapp* [*t1*] *TC-ref*)

fun *TC-word* :: *word-size* ⇒ *tctor* **where**
TC-word W8 = (*TC-word8*)
| *TC-word W64* = (*TC-word64*)

definition *Tword* :: *word-size* ⇒ *t* **where**
Tword wz = (*Tapp* [] (*TC-word wz*))

definition *Tword8* :: *t* **where**
Tword8 = (*Tword W8*)

definition *Tword64* :: *t* **where**
Tword64 = (*Tword W64*)

definition *Tword8array* :: *t* **where**
Tword8array = (*Tapp* [] *TC-word8array*)

definition *Tfn* :: *t* ⇒ *t* ⇒ *t* **where**
Tfn t1 t2 = (*Tapp* [*t1*,*t2*] *TC-fn*)

definition *Texn* :: *t* **where**
Texn = (*Tapp* [] *TC-exn*)

— *Patterns*
datatype *pat* =
Pany
| *Pvar varN*
| *Plit lit*
— *Constructor applications. A Nothing constructor indicates a tuple pattern.*
| *Pcon* ((*modN*, *conN*)*id0*)*option pat list*
| *Pref pat*

| *Ptannot pat t*

— *Expressions*
datatype *exp0* =
Raise exp0
| *Handle exp0 (pat ∗ exp0) list*
| *Lit lit*
— *Constructor application. A Nothing constructor indicates a tuple pattern.*
| *Con ((modN, conN)id0)option exp0 list*
| *Var (modN, varN) id0*
| *Fun varN exp0*
— *Application of a primitive operator to arguments. Includes function application.*
| *App op0 exp0 list*
— *Logical operations (and, or)*
| *Log lop exp0 exp0*
| *If exp0 exp0 exp0*
— *Pattern matching*
| *Mat exp0 (pat ∗ exp0) list*
— *A let expression A Nothing value for the binding indicates that this is a sequencing expression, that is: (e1; e2).*
| *Let varN option exp0 exp0*
— *Local definition of (potentially) mutually recursive functions. The first varN is the function's name, and the second varN is its parameter.*
| *Letrec (varN ∗ varN ∗ exp0) list exp0*
| *Tannot exp0 t*
— *Location annotated expressions, not expected in source programs*
| *Lannot exp0 locs*

type-synonym *type-def* = (*tvarN list ∗ typeN ∗ (conN ∗ t list) list) list*

— *Declarations*
datatype *dec* =
— *Top−level bindings ∗ The pattern allows several names to be bound at once*
Dlet locs pat exp0
— *Mutually recursive function definition*
| *Dletrec locs (varN ∗ varN ∗ exp0) list*
— *Type definition Defines several data types, each of which has several named variants, which can in turn have several arguments.*
| *Dtype locs type-def*
— *Type abbreviations*
| *Dtabbrev locs tvarN list typeN t*
— *New exceptions*
| *Dexn locs conN t list*

type-synonym *decs* = *dec list*

— *Specifications For giving the signature of a module*
datatype *spec* =
Sval varN t

| *Stype type-def*
| *Stabbrev tvarN list typeN t*
| *Stype-opq tvarN list typeN*
| *Sexn conN t list*

type-synonym *specs* = *spec list*

datatype *top0* =
Tmod modN specs option decs
| *Tdec dec*

type-synonym *prog* = *top0 list*

— *Accumulates the bindings of a pattern*
— *val pat-bindings : pat −> list varN −> list varN*
function (*sequential*,*domintros*)
pats-bindings :: (*pat*)*list* ⇒(*string*)*list* ⇒(*string*)*list*
and
pat-bindings :: *pat* ⇒(*string*)*list* ⇒(*string*)*list* **where**

pat-bindings Pany already-bound = (
already-bound)
|
pat-bindings (*Pvar n*) *already-bound* = (
n # *already-bound*)
|
pat-bindings (*Plit l*) *already-bound* = (
already-bound)
|
pat-bindings (*Pcon - ps*) *already-bound* = (
pats-bindings ps already-bound)
|
pat-bindings (*Pref p*) *already-bound* = (
pat-bindings p already-bound)
|
pat-bindings (*Ptannot p -*) *already-bound* = (
pat-bindings p already-bound)
|
pats-bindings [] *already-bound* = (
already-bound)
|
pats-bindings (*p* # *ps*) *already-bound* = (
pats-bindings ps (*pat-bindings p already-bound*))
by *pat-completeness auto*

end

Chapter 5

Generated by Lem from *semantics/ast.lem*.

theory *AstAuxiliary*

imports
Main
HOL−Library.Datatype-Records
LEM.Lem-pervasives
Lib
Namespace
FpSem
Ast

begin

— **
—
— *Termination Proofs*
—
— **

termination *pat-bindings* **by** *lexicographic-order*

end

Chapter 6

Generated by Lem from *semantics/ffi/ffi.lem*.

theory *Ffi*

imports
Main
HOL−Library.Datatype-Records
LEM.Lem-pervasives
LEM.Lem-pervasives-extra
Lib

begin

— *open import Pervasives*
— *open import Pervasives-extra*
— *open import Lib*

— *An oracle says how to perform an ffi call based on its internal state,* ∗ *represented by the type variable 'ffi.*

datatype *'ffi oracle-result* = *Oracle-return 'ffi 8 word list* | *Oracle-diverge* | *Oracle-fail*
type-synonym *'ffi oracle-function* = *'ffi* ⇒ *8 word list* ⇒ *8 word list* ⇒ *'ffi oracle-result*
type-synonym *'ffi oracle0* = *string* ⇒ *'ffi oracle-function*

— *An I/O event, IO-event s bytes bytes2, represents the call of FFI function s with* ∗ *immutable input bytes and mutable input map fst bytes2,* ∗ *returning map snd bytes2 in the mutable array.*

datatype *io-event* = *IO-event string 8 word list* ((*8 word* ∗ *8 word*)*list*)

datatype *ffi-outcome* = *FFI-diverged* | *FFI-failed*
datatype *final-event* = *Final-event string 8 word list 8 word list ffi-outcome*

datatype-record *'ffi ffi-state* =

oracle0 :: *'ffi oracle0*

ffi-state :: *'ffi*

final-event :: *final-event option*

io-events :: *io-event list*

— *val initial-ffi-state : forall 'ffi. oracle 'ffi −> 'ffi −> ffi-state 'ffi*
definition *initial-ffi-state* :: (*string* ⇒ *'ffi oracle-function*)⇒ *'ffi* ⇒ *'ffi ffi-state*
where

```
    initial-ffi-state oc ffi1 = (
(| oracle0     = oc
 , ffi-state   = ffi1
 , final-event = None
 , io-events   = ([])
 |) )
```

— *val call-FFI : forall 'ffi. ffi-state 'ffi −> string −> list word8 −> list word8 −> ffi-state 'ffi ∗ list word8*
definition *call-FFI* :: *'ffi ffi-state* ⇒ *string* ⇒(*8 word*)*list* ⇒(*8 word*)*list* ⇒ *'ffi ffi-state*∗(*8 word*)*list* **where**

```
    call-FFI st s conf bytes = (
  if ((final-event  st) = None) ∧ ¬ (s = ('''')) then
    (case (oracle0  st) s(ffi-state  st) conf bytes of
      Oracle-return ffi' bytes' =>
        if List.length bytes' = List.length bytes then
          (( st (| ffi-state := ffi'
                   , io-events :=
                       ((io-events  st) @
                         [IO-event s conf (zipSameLength bytes bytes')])
            |)), bytes')
        else (( st (| final-event := (Some (Final-event s conf bytes FFI-failed)) |)),
bytes)
    | Oracle-diverge =>
          (( st (| final-event := (Some (Final-event s conf bytes FFI-diverged)) |)),
bytes)
    | Oracle-fail =>
        (( st (| final-event := (Some (Final-event s conf bytes FFI-failed)) |)), bytes)
    )
  else (st, bytes))
```

datatype *outcome* = *Success* | *Resource-limit-hit* | *FFI-outcome final-event*

— *A program can Diverge, Terminate, or Fail. We prove that Fail is avoided. For Diverge and Terminate, we keep track of what I/O events are valid I/O events for this behaviour.*
datatype *behaviour* =
— *There cannot be any non−returning FFI calls in a diverging exeuction. The list of I/O events can be finite or infinite, hence the llist (lazy list) type.*
Diverge io-event llist
— *Terminating executions can only perform a finite number of FFI calls. The execution can be terminated by a non−returning FFI call.*
| *Terminate outcome io-event list*
— *Failure is a behaviour which we prove cannot occur for any well−typed program.*
| *Fail*

— *trace−based semantics can be recovered as an instance of oracle−based ∗ semantics as follows.*

— *val trace-oracle* : *oracle* (*llist io-event*)
definition *trace-oracle* :: *string* ⇒(*io-event*)*llist* ⇒(*8 word*)*list* ⇒(*8 word*)*list* ⇒((*io-event*)*llist*)*oracle-result* **where**

```
  trace-oracle s io-trace conf input1 = (
 (case lhd' io-trace of
   Some (IO-event s' conf' bytes2) =>
     if (s = s') ∧ ((List.map fst bytes2 = input1) ∧ (conf = conf')) then
       Oracle-return (Option.the (ltl' io-trace)) (List.map snd bytes2)
     else Oracle-fail
 | - => Oracle-fail
 ))
```

end

Chapter 7

Generated by Lem from *semantics/semanticPrimitives.lem*.

theory *SemanticPrimitives*

imports
Main
HOL−Library.Datatype-Records
LEM.Lem-pervasives
LEM.Lem-list-extra
LEM.Lem-string
Lib
Namespace
Ast
Ffi
FpSem
LEM.Lem-string-extra

begin

— *open import Pervasives*
— *open import Lib*
— *import List-extra*
— *import String*
— *import String-extra*
— *open import Ast*
— *open import Namespace*
— *open import Ffi*
— *open import FpSem*

— *The type that a constructor builds is either a named datatype or an exception. * For exceptions, we also keep the module that the exception was declared in.*
datatype *tid-or-exn* =
TypeId (*modN*, *typeN*) *id0*
| *TypeExn* (*modN*, *conN*) *id0*

— *val type-defs-to-new-tdecs* : *list modN −> type-def −> set tid-or-exn*
definition *type-defs-to-new-tdecs* :: (*string*)*list* ⇒((*tvarN*)*list*∗*string*∗(*conN*∗(*t*)*list*)*list*)*list* ⇒(*tid-or-exn*)*set* **where**
type-defs-to-new-tdecs mn tdefs = (
List.set (*List.map* (*λx* .
(*case x of* (*tvs*,*tn*,*ctors*) => *TypeId* (*mk-id mn tn*))) *tdefs*))

datatype-record ′*v sem-env* =

v :: (*modN*, *varN*, ′*v*) *namespace*

c :: (*modN*, *conN*, (*nat* ∗ *tid-or-exn*)) *namespace*

— *Value forms*
datatype *v* =
Litv lit
— *Constructor application.*
| *Conv* (*conN* ∗ *tid-or-exn*)*option* *v list*
— *Function closures The environment is used for the free variables in the function*
| *Closure v sem-env varN exp0*
— *Function closure for recursive functions* ∗ *See Closure and Letrec above* ∗ *The last variable name indicates which function from the mutually* ∗ *recursive bundle this closure value represents*
| *Recclosure v sem-env* (*varN* ∗ *varN* ∗ *exp0*) *list* *varN*
| *Loc nat*
| *Vectorv v list*

type-synonym *env-ctor* = (*modN*, *conN*, (*nat* ∗ *tid-or-exn*)) *namespace*
type-synonym *env-val* = (*modN*, *varN*, *v*) *namespace*

definition *Bindv* :: *v* **where**
Bindv = (*Conv* (*Some*((′′*Bind*′′),*TypeExn*(*Short*(′′*Bind*′′)))) [])

— *The result of evaluation*
datatype *abort* =
Rtype-error
| *Rtimeout-error*

datatype ′*a error-result* =
Rraise ′*a* — *Should only be a value of type exn*
| *Rabort abort*

datatype(′*a*, ′*b*) *result* =
Rval ′*a*

```
  | Rerr  'b error-result

— Stores
datatype 'a store-v =
  — A ref cell
    Refv  'a
  — A byte array
  | W8array  8 word list
  — An array of values
  | Varray  'a list

— val store-v-same-type : forall 'a. store-v 'a −> store-v 'a −> bool
definition store-v-same-type  ::  'a store-v ⇒ 'a store-v ⇒ bool   where
      store-v-same-type v1 v2 = (
  (case  (v1,v2)  of
    (Refv -, Refv -) => True
  | (W8array -,W8array -) => True
  | (Varray -,Varray -) => True
  | - => False
  ))


— The nth item in the list is the value at location n
type-synonym 'a store = ( 'a store-v) list

— val empty-store : forall 'a. store 'a
definition empty-store  :: ('a store-v)list   where
      empty-store = ( [])


— val store-lookup : forall 'a. nat −> store 'a −> maybe (store-v 'a)
definition store-lookup  ::  nat ⇒('a store-v)list ⇒('a store-v)option   where
      store-lookup l st = (
  if l < List.length st then
    Some (List.nth st l)
  else
    None )


— val store-alloc : forall 'a. store-v 'a −> store 'a −> store 'a ∗ nat
definition store-alloc  ::  'a store-v ⇒('a store-v)list ⇒('a store-v)list∗nat   where

      store-alloc v2 st = (
  ((st @ [v2]), List.length st))


— val store-assign : forall 'a. nat −> store-v 'a −> store 'a −> maybe (store 'a)
definition store-assign  ::  nat ⇒ 'a store-v ⇒('a store-v)list ⇒(('a store-v)list)option
where
```

```
    store-assign n v2 st = (
  if (n < List.length st) ∧
     store-v-same-type (List.nth st n) v2
  then
    Some (List.list-update st n v2)
  else
    None )
```

datatype-record *'ffi state =*

clock :: nat

refs :: v store

ffi :: 'ffi ffi-state

defined-types :: tid-or-exn set

defined-mods :: (modN list) set

```
— Other primitives
— Check that a constructor is properly applied
— val do-con-check : env-ctor −> maybe (id modN conN) −> nat −> bool
fun do-con-check :: ((string),(string),(nat∗tid-or-exn))namespace ⇒(((string),(string))id0)option
⇒ nat ⇒ bool   where
    do-con-check cenv None l = ( True )
| do-con-check cenv (Some n) l = (
      (case nsLookup cenv n of
          None => False
        | Some (l',ns) => l = l'
      ))
```

```
— val build-conv : env-ctor −> maybe (id modN conN) −> list v −> maybe v
fun build-conv :: ((string),(string),(nat∗tid-or-exn))namespace ⇒(((string),(string))id0)option
⇒(v)list ⇒(v)option   where
    build-conv envC None vs = (
      Some (Conv None vs))
| build-conv envC (Some id1) vs = (
      (case nsLookup envC id1 of
          None => None
        | Some (len,t1) => Some (Conv (Some (id-to-n id1, t1)) vs)
      ))
```

— *val lit-same-type : lit −> lit −> bool*

definition *lit-same-type* :: *lit* ⇒ *lit* ⇒ *bool* **where**
lit-same-type l1 l2 = (
(*case* (*l1*,*l2*) *of*
(*IntLit* -, *IntLit* -) => *True*
| (*Char* -, *Char* -) => *True*
| (*StrLit* -, *StrLit* -) => *True*
| (*Word8* -, *Word8* -) => *True*
| (*Word64* -, *Word64* -) => *True*
| - => *False*
))

datatype *'a match-result* =
No-match
| *Match-type-error*
| *Match 'a*

— *val same-tid* : *tid-or-exn* −> *tid-or-exn* −> *bool*
fun *same-tid* :: *tid-or-exn* ⇒ *tid-or-exn* ⇒ *bool* **where**
same-tid (*TypeId tn1*) (*TypeId tn2*) = (*tn1* = *tn2*)
| *same-tid* (*TypeExn* -) (*TypeExn* -) = (*True*)
| *same-tid* - - = (*False*)

— *val same-ctor* : *conN* ∗ *tid-or-exn* −> *conN* ∗ *tid-or-exn* −> *bool*
fun *same-ctor* :: *string*∗*tid-or-exn* ⇒ *string*∗*tid-or-exn* ⇒ *bool* **where**
same-ctor (*cn1*, *TypeExn mn1*) (*cn2*, *TypeExn mn2*) = ((*cn1* = *cn2*) ∧ (*mn1* = *mn2*))
| *same-ctor* (*cn1*, -) (*cn2*, -) = (*cn1* = *cn2*)

— *val ctor-same-type* : *maybe* (*conN* ∗ *tid-or-exn*) −> *maybe* (*conN* ∗ *tid-or-exn*) −> *bool*
definition *ctor-same-type* :: (*string*∗*tid-or-exn*)*option* ⇒(*string*∗*tid-or-exn*)*option* ⇒ *bool* **where**
ctor-same-type c1 c2 = (
(*case* (*c1*,*c2*) *of*
(*None*, *None*) => *True*
| (*Some* (-,*t1*), *Some* (-,*t2*)) => *same-tid t1 t2*
| - => *False*
))

— *A big−step pattern matcher. If the value matches the pattern, return an* ∗ *environment with the pattern variables bound to the corresponding sub−terms* ∗ *of the value; this environment extends the environment given as an argument.* ∗ *No-match is returned when there is no match, but any constructors* ∗ *encountered in determining the match failure are applied to the correct* ∗ *number of arguments, and constructors in corresponding positions in the* ∗ *pattern and value come from*

the same type. Match-type-error is returned ∗ when one of these conditions is violated
— *val pmatch : env-ctor −> store v −> pat −> v −> alist varN v −> match-result* (*alist varN v*)
function (*sequential*,*domintros*)
pmatch-list :: ((*string*),(*string*),(*nat*∗*tid-or-exn*))*namespace* ⇒((*v*)*store-v*)*list* ⇒(*pat*)*list* ⇒(*v*)*list* ⇒(*string*∗*v*)*list* ⇒((*string*∗*v*)*list*)*match-result*
and
pmatch :: ((*string*),(*string*),(*nat*∗*tid-or-exn*))*namespace* ⇒((*v*)*store-v*)*list* ⇒ *pat* ⇒ *v* ⇒(*string*∗*v*)*list* ⇒((*string*∗*v*)*list*)*match-result* **where**

```
pmatch envC s Pany v' env = ( Match env )
|
pmatch envC s (Pvar x) v' env = ( Match ((x,v')# env))
|
pmatch envC s (Plit l) (Litv l') env = (
  if l = l' then
    Match env
  else if lit-same-type l l' then
    No-match
  else
    Match-type-error )
|
pmatch envC s (Pcon (Some n) ps) (Conv (Some (n', t')) vs) env = (
  (case  nsLookup envC n of
      Some (l, t1) =>
        if same-tid t1 t' ∧ (List.length ps = l) then
          if same-ctor (id-to-n n, t1) (n',t') then
            if List.length vs = l then
              pmatch-list envC s ps vs env
            else
              Match-type-error
          else
            No-match
        else
          Match-type-error
    | - => Match-type-error
  ))
|
pmatch envC s (Pcon None ps) (Conv None vs) env = (
  if List.length ps = List.length vs then
    pmatch-list envC s ps vs env
  else
    Match-type-error )
|
pmatch envC s (Pref p) (Loc lnum) env = (
  (case  store-lookup lnum s of
      Some (Refv v2) => pmatch envC s p v2 env
    | Some - => Match-type-error
```

```
    | None => Match-type-error
  ))
|
pmatch envC s (Ptannot p t1) v2 env = (
  pmatch envC s p v2 env )
|
pmatch envC - - - env = ( Match-type-error )
|
pmatch-list envC s [] [] env = ( Match env )
|
pmatch-list envC s (p # ps) (v2 # vs) env = (
  (case  pmatch envC s p v2 env of
      No-match => No-match
    | Match-type-error => Match-type-error
    | Match env' => pmatch-list envC s ps vs env'
  ))
|
pmatch-list envC s - - env = ( Match-type-error )
by pat-completeness auto


— Bind each function of a mutually recursive set of functions to its closure
— val build-rec-env : list (varN * varN * exp) -> sem-env v -> env-val ->
env-val
definition build-rec-env :: (varN*varN*exp0)list ⇒(v)sem-env ⇒((string),(string),(v))namespace
⇒((string),(string),(v))namespace   where
      build-rec-env funs cl-env add-to-env = (
  List.foldr ( λx .
  (case  x of
      (f,x,e) => λ env' .  nsBind f (Recclosure cl-env funs f) env'
  )) funs add-to-env )


— Lookup in the list of mutually recursive functions
— val find-recfun : forall 'a 'b. varN -> list (varN * 'a * 'b) -> maybe ('a * 'b)
fun  find-recfun  ::  string ⇒(string*'a*'b)list ⇒('a*'b)option   where
      find-recfun n ([]) = ( None )
| find-recfun n ((f,x,e) # funs) = (
        if f = n then
          Some (x,e)
        else
          find-recfun n funs )


datatype eq-result =
    Eq-val  bool
  | Eq-type-error

— val do-eq : v -> v -> eq-result
```

```
function (sequential,domintros)
do-eq-list :: (v)list ⇒(v)list ⇒ eq-result
                    and
do-eq :: v ⇒ v ⇒ eq-result   where

do-eq (Litv l1) (Litv l2) = (
  if lit-same-type l1 l2 then Eq-val (l1 = l2)
  else Eq-type-error )
|
do-eq (Loc l1) (Loc l2) = ( Eq-val (l1 = l2))
|
do-eq (Conv cn1 vs1) (Conv cn2 vs2) = (
  if (cn1 = cn2) ∧ (List.length vs1 = List.length vs2) then
    do-eq-list vs1 vs2
  else if ctor-same-type cn1 cn2 then
    Eq-val False
  else
    Eq-type-error )
|
do-eq (Vectorv vs1) (Vectorv vs2) = (
  if List.length vs1 = List.length vs2 then
    do-eq-list vs1 vs2
  else
    Eq-val False )
|
do-eq (Closure - - -) (Closure - - -) = ( Eq-val True )
|
do-eq (Closure - - -) (Recclosure - - -) = ( Eq-val True )
|
do-eq (Recclosure - - -) (Closure - - -) = ( Eq-val True )
|
do-eq (Recclosure - - -) (Recclosure - - -) = ( Eq-val True )
|
do-eq - - = ( Eq-type-error )
|
do-eq-list [] [] = ( Eq-val True )
|
do-eq-list (v1 # vs1) (v2 # vs2) = (
  (case  do-eq v1 v2 of
      Eq-type-error => Eq-type-error
    | Eq-val r =>
        if ¬ r then
          Eq-val False
        else
          do-eq-list vs1 vs2
  ))
|
do-eq-list - - = ( Eq-val False )
by pat-completeness auto
```

— *val prim-exn : conN −> v*
definition *prim-exn* :: *string* ⇒ *v* **where**
prim-exn cn = (*Conv* (*Some* (*cn*, *TypeExn* (*Short cn*))) [])

— *Do an application*
— *val do-opapp : list v −> maybe* (*sem-env v* ∗ *exp*)
fun *do-opapp* :: (*v*)*list* ⇒((*v*)*sem-env*∗*exp0*)*option* **where**
do-opapp ([*Closure env n e*, *v2*]) = (
Some ((*env* (| *v* := (*nsBind n v2*(*v env*)) |)), *e*))
| *do-opapp* ([*Recclosure env funs n*, *v2*]) = (
if allDistinct (*List.map* (*λx* .
(*case x of* (*f*,*x*,*e*) => *f*)) *funs*) *then*
(*case find-recfun n funs of*
Some (*n*,*e*) => *Some* ((*env* (| *v* := (*nsBind n v2* (*build-rec-env funs env*(*v env*))) |)), *e*)
| *None* => *None*
)
else
None)
| *do-opapp* - = (*None*)

— *If a value represents a list, get that list. Otherwise return Nothing*
— *val v-to-list : v −> maybe* (*list v*)
function (*sequential*,*domintros*) *v-to-list* :: *v* ⇒((*v*)*list*)*option* **where**
v-to-list (*Conv* (*Some* (*cn*, *TypeId* (*Short tn*))) []) = (
if (*cn* = (''*nil*'')) ∧ (*tn* = (''*list*'')) *then*
Some []
else
None)
| *v-to-list* (*Conv* (*Some* (*cn*,*TypeId* (*Short tn*))) [*v1*,*v2*]) = (
if (*cn* = ('':: '')) ∧ (*tn* = (''*list*'')) *then*
(*case v-to-list v2 of*
Some vs => *Some* (*v1* # *vs*)
| *None* => *None*
)
else
None)
| *v-to-list* - = (*None*)
by *pat-completeness auto*

— *val v-to-char-list : v −> maybe* (*list char*)
function (*sequential*,*domintros*) *v-to-char-list* :: *v* ⇒((*char*)*list*)*option* **where**

v-to-char-list (*Conv* (*Some* (*cn*, *TypeId* (*Short tn*))) []) = (

```
  if (cn = (''nil'')) ∧ (tn = (''list'')) then
    Some []
  else
    None )
| v-to-char-list (Conv (Some (cn,TypeId (Short tn))) [Litv (Char c2),v2]) = (
  if (cn = ('':::'')) ∧ (tn = (''list'')) then
    (case v-to-char-list v2 of
        Some cs => Some (c2 # cs)
      | None => None
    )
  else
    None )
| v-to-char-list - = ( None )
by pat-completeness auto
```

```
— val vs-to-string : list v −> maybe string
function (sequential,domintros) vs-to-string :: (v)list ⇒(string)option where
      vs-to-string [] = ( Some (''''))
| vs-to-string (Litv(StrLit s1)# vs) = (
  (case vs-to-string vs of
    Some s2 => Some (s1 @ s2)
  | - => None
  ))
| vs-to-string - = ( None )
by pat-completeness auto
```

```
— val copy-array : forall 'a. list 'a ∗ integer −> integer −> maybe (list 'a ∗
integer) −> maybe (list 'a)
fun copy-array :: 'a list∗int ⇒ int ⇒('a list∗int)option ⇒('a list)option where

      copy-array (src,srcoff) len d = (
  if (srcoff <( 0 :: int)) ∨ ((len <( 0 :: int)) ∨ (List.length src < nat (abs ( (srcoff
+ len))))) then None else
    (let copied = (List.take (nat (abs ( len))) (List.drop (nat (abs ( srcoff))) src))
in
    (case d of
      Some (dst,dstoff) =>
        if (dstoff <( 0 :: int)) ∨ (List.length dst < nat (abs ( (dstoff + len)))) then
None else
          Some ((List.take (nat (abs ( dstoff))) dst @
                copied) @
                List.drop (nat (abs ( (dstoff + len)))) dst)
    | None => Some copied
    )))
```

```
— val ws-to-chars : list word8 −> list char
```

definition *ws-to-chars* :: (*8 word*)*list* ⇒(*char*)*list* **where**
ws-to-chars ws = (*List.map* (λ *w* . (%*n. char-of* (*n*::*nat*))(*unat w*)) *ws*)

— *val chars-to-ws* : *list char* −> *list word8*
definition *chars-to-ws* :: (*char*)*list* ⇒(*8 word*)*list* **where**
chars-to-ws cs = (*List.map* (λ *c2* . *word-of-int*(*int*(*of-char c2*))) *cs*)

— *val opn-lookup* : *opn* −> *integer* −> *integer* −> *integer*
fun *opn-lookup* :: *opn* ⇒ *int* ⇒ *int* ⇒ *int* **where**
opn-lookup Plus = ((+))
| *opn-lookup Minus* = ((−))
| *opn-lookup Times* = ((∗))
| *opn-lookup Divide* = ((*div*))
| *opn-lookup Modulo* = ((*mod*))

— *val opb-lookup* : *opb* −> *integer* −> *integer* −> *bool*
fun *opb-lookup* :: *opb* ⇒ *int* ⇒ *int* ⇒ *bool* **where**
opb-lookup Lt = ((<))
| *opb-lookup Gt* = ((>))
| *opb-lookup Leq* = ((≤))
| *opb-lookup Geq* = ((≥))

— *val opw8-lookup* : *opw* −> *word8* −> *word8* −> *word8*
fun *opw8-lookup* :: *opw* ⇒ *8 word* ⇒ *8 word* ⇒ *8 word* **where**
opw8-lookup Andw = (*Bit-Operations.and*)
| *opw8-lookup Orw* = (*Bit-Operations.or*)
| *opw8-lookup Xor* = (*Bit-Operations.xor*)
| *opw8-lookup Add* = (*Groups.plus*)
| *opw8-lookup Sub* = (*Groups.minus*)

— *val opw64-lookup* : *opw* −> *word64* −> *word64* −> *word64*
fun *opw64-lookup* :: *opw* ⇒ *64 word* ⇒ *64 word* ⇒ *64 word* **where**
opw64-lookup Andw = (*Bit-Operations.and*)
| *opw64-lookup Orw* = (*Bit-Operations.or*)
| *opw64-lookup Xor* = (*Bit-Operations.xor*)
| *opw64-lookup Add* = (*Groups.plus*)
| *opw64-lookup Sub* = (*Groups.minus*)

— *val shift8-lookup* : *shift* −> *word8* −> *nat* −> *word8*
fun *shift8-lookup* :: *shift* ⇒ *8 word* ⇒ *nat* ⇒ *8 word* **where**
shift8-lookup Lsl = (*shiftl*)
| *shift8-lookup Lsr* = (*shiftr*)
| *shift8-lookup Asr* = (*sshiftr*)

| *shift8-lookup Ror* = ((% *a b. word-rotr b a*))

— *val shift64-lookup : shift −> word64 −> nat −> word64*
fun *shift64-lookup* :: *shift* ⇒ *64 word* ⇒ *nat* ⇒ *64 word* **where**
shift64-lookup Lsl = (*shiftl*)
| *shift64-lookup Lsr* = (*shiftr*)
| *shift64-lookup Asr* = (*sshiftr*)
| *shift64-lookup Ror* = ((% *a b. word-rotr b a*))

— *val Boolv : bool −> v*
definition *Boolv* :: *bool* ⇒ *v* **where**
Boolv b = (*if b*
then Conv (*Some* ((''*true*''), *TypeId* (*Short* (''*bool*'')))) []
else Conv (*Some* ((''*false*''), *TypeId* (*Short* (''*bool*'')))) [])

datatype *exp-or-val* =
Exp exp0
| *Val v*

type-synonym(*'ffi*, *'v*) *store-ffi* = *'v store* ∗ *'ffi ffi-state*

— *val do-app : forall 'ffi. store-ffi 'ffi v −> op −> list v −> maybe (store-ffi 'ffi v ∗ result v v)*
fun *do-app* :: ((*v*)*store-v*)*list*∗*'ffi ffi-state* ⇒ *op0* ⇒(*v*)*list* ⇒((((*v*)*store-v*)*list*∗*'ffi ffi-state*)∗((*v*),(*v*))*result*)*option* **where**
do-app ((*s*:: *v store*),(*t1*:: *'ffi ffi-state*)) *op1 vs* = (
(*case* (*op1*, *vs*) *of*
(*Opn op1*, [*Litv* (*IntLit n1*), *Litv* (*IntLit n2*)]) =>
if ((*op1* = *Divide*) ∨ (*op1* = *Modulo*)) ∧ (*n2* =(*0* :: *int*)) *then*
Some ((*s*,*t1*), *Rerr* (*Rraise* (*prim-exn* (''*Div*''))))
else
Some ((*s*,*t1*), *Rval* (*Litv* (*IntLit* (*opn-lookup op1 n1 n2*))))
| (*Opb op1*, [*Litv* (*IntLit n1*), *Litv* (*IntLit n2*)]) =>
Some ((*s*,*t1*), *Rval* (*Boolv* (*opb-lookup op1 n1 n2*)))
| (*Opw W8 op1*, [*Litv* (*Word8 w1*), *Litv* (*Word8 w2*)]) =>
Some ((*s*,*t1*), *Rval* (*Litv* (*Word8* (*opw8-lookup op1 w1 w2*))))
| (*Opw W64 op1*, [*Litv* (*Word64 w1*), *Litv* (*Word64 w2*)]) =>
Some ((*s*,*t1*), *Rval* (*Litv* (*Word64* (*opw64-lookup op1 w1 w2*))))
| (*FP-bop bop*, [*Litv* (*Word64 w1*), *Litv* (*Word64 w2*)]) =>
Some ((*s*,*t1*),*Rval* (*Litv* (*Word64* (*fp-bop bop w1 w2*))))
| (*FP-uop uop*, [*Litv* (*Word64 w*)]) =>
Some ((*s*,*t1*),*Rval* (*Litv* (*Word64* (*fp-uop uop w*))))
| (*FP-cmp cmp*, [*Litv* (*Word64 w1*), *Litv* (*Word64 w2*)]) =>
Some ((*s*,*t1*),*Rval* (*Boolv* (*fp-cmp cmp w1 w2*)))
| (*Shift W8 op1 n*, [*Litv* (*Word8 w*)]) =>
Some ((*s*,*t1*), *Rval* (*Litv* (*Word8* (*shift8-lookup op1 w n*))))

```
| (Shift W64 op1 n, [Litv (Word64 w)]) =>
    Some ((s,t1), Rval (Litv (Word64 (shift64-lookup op1 w n))))
| (Equality, [v1, v2]) =>
    (case do-eq v1 v2 of
        Eq-type-error => None
      | Eq-val b => Some ((s,t1), Rval (Boolv b))
    )
| (Opassign, [Loc lnum, v2]) =>
    (case store-assign lnum (Refv v2) s of
        Some s' => Some ((s',t1), Rval (Conv None []))
      | None => None
    )
| (Opref, [v2]) =>
    (let (s',n) = (store-alloc (Refv v2) s) in
      Some ((s',t1), Rval (Loc n)))
| (Opderef, [Loc n]) =>
    (case store-lookup n s of
        Some (Refv v2) => Some ((s,t1),Rval v2)
      | - => None
    )
| (Aw8alloc, [Litv (IntLit n), Litv (Word8 w)]) =>
    if n <( 0 :: int) then
      Some ((s,t1), Rerr (Rraise (prim-exn (''Subscript''))))
    else
      (let (s',lnum) =
        (store-alloc (W8array (List.replicate (nat (abs ( n))) w)) s)
      in
        Some ((s',t1), Rval (Loc lnum)))
| (Aw8sub, [Loc lnum, Litv (IntLit i)]) =>
    (case store-lookup lnum s of
        Some (W8array ws) =>
          if i <( 0 :: int) then
            Some ((s,t1), Rerr (Rraise (prim-exn (''Subscript''))))
          else
            (let n = (nat (abs ( i))) in
              if n ≥ List.length ws then
                Some ((s,t1), Rerr (Rraise (prim-exn (''Subscript''))))
              else
                Some ((s,t1), Rval (Litv (Word8 (List.nth ws n)))))
      | - => None
    )
| (Aw8length, [Loc n]) =>
    (case store-lookup n s of
        Some (W8array ws) =>
          Some ((s,t1),Rval (Litv(IntLit(int(List.length ws)))))
      | - => None
     )
| (Aw8update, [Loc lnum, Litv(IntLit i), Litv(Word8 w)]) =>
    (case store-lookup lnum s of
```

```
      Some (W8array ws) =>
        if i <( 0 :: int) then
          Some ((s,t1), Rerr (Rraise (prim-exn (''Subscript''))))
        else
          (let n = (nat (abs ( i))) in
            if n ≥ List.length ws then
              Some ((s,t1), Rerr (Rraise (prim-exn (''Subscript''))))
            else
              (case  store-assign lnum (W8array (List.list-update ws n w)) s of
                  None => None
                | Some s' => Some ((s',t1), Rval (Conv None []))
              ))
    | - => None
  )
| (WordFromInt W8, [Litv(IntLit i)]) =>
    Some ((s,t1), Rval (Litv (Word8 (word-of-int i))))
| (WordFromInt W64, [Litv(IntLit i)]) =>
    Some ((s,t1), Rval (Litv (Word64 (word-of-int i))))
| (WordToInt W8, [Litv (Word8 w)]) =>
    Some ((s,t1), Rval (Litv (IntLit (int(unat w)))))
| (WordToInt W64, [Litv (Word64 w)]) =>
    Some ((s,t1), Rval (Litv (IntLit (int(unat w)))))
| (CopyStrStr, [Litv(StrLit str),Litv(IntLit off),Litv(IntLit len)]) =>
    Some ((s,t1),
    (case  copy-array ( str,off) len None of
      None => Rerr (Rraise (prim-exn (''Subscript'')))
    | Some cs => Rval (Litv(StrLit((cs))))
    ))
| (CopyStrAw8, [Litv(StrLit str),Litv(IntLit off),Litv(IntLit len),
              Loc dst,Litv(IntLit dstoff)]) =>
    (case  store-lookup dst s of
      Some (W8array ws) =>
        (case  copy-array ( str,off) len (Some(ws-to-chars ws,dstoff)) of
          None => Some ((s,t1), Rerr (Rraise (prim-exn (''Subscript''))))
        | Some cs =>
          (case  store-assign dst (W8array (chars-to-ws cs)) s of
            Some s' =>  Some ((s',t1), Rval (Conv None []))
          | - => None
          )
        )
    | - => None
    )
| (CopyAw8Str, [Loc src,Litv(IntLit off),Litv(IntLit len)]) =>
  (case  store-lookup src s of
    Some (W8array ws) =>
    Some ((s,t1),
      (case  copy-array (ws,off) len None of
        None => Rerr (Rraise (prim-exn (''Subscript'')))
      | Some ws => Rval (Litv(StrLit((ws-to-chars ws))))
```

```
      ))
    | - => None
    )
  | (CopyAw8Aw8, [Loc src,Litv(IntLit off),Litv(IntLit len),
                 Loc dst,Litv(IntLit dstoff)]) =>
    (case (store-lookup src s, store-lookup dst s) of
      (Some (W8array ws), Some (W8array ds)) =>
        (case copy-array (ws,off) len (Some(ds,dstoff)) of
          None => Some ((s,t1), Rerr (Rraise (prim-exn (''Subscript''))))
        | Some ws =>
            (case store-assign dst (W8array ws) s of
              Some s' => Some ((s',t1), Rval (Conv None []))
            | - => None
            )
        )
    | - => None
    )
  | (Ord, [Litv (Char c2)]) =>
        Some ((s,t1), Rval (Litv(IntLit(int(of-char c2)))))
  | (Chr, [Litv (IntLit i)]) =>
      Some ((s,t1),
        (if (i <( 0 :: int)) ∨ (i >( 255 :: int)) then
          Rerr (Rraise (prim-exn (''Chr'')))
        else
          Rval (Litv(Char((%n. char-of (n::nat))(nat (abs ( i))))))))
  | (Chopb op1, [Litv (Char c1), Litv (Char c2)]) =>
        Some ((s,t1), Rval (Boolv (opb-lookup op1 (int(of-char c1)) (int(of-char
c2)))))
  | (Implode, [v2]) =>
        (case v-to-char-list v2 of
          Some ls =>
            Some ((s,t1), Rval (Litv (StrLit ( ls))))
        | None => None
        )
  | (Strsub, [Litv (StrLit str), Litv (IntLit i)]) =>
      if i <( 0 :: int) then
        Some ((s,t1), Rerr (Rraise (prim-exn (''Subscript''))))
      else
        (let n = (nat (abs ( i))) in
          if n ≥ List.length str then
            Some ((s,t1), Rerr (Rraise (prim-exn (''Subscript''))))
          else
            Some ((s,t1), Rval (Litv (Char (List.nth ( str) n)))))
  | (Strlen, [Litv (StrLit str)]) =>
      Some ((s,t1), Rval (Litv(IntLit(int(List.length str)))))
  | (Strcat, [v2]) =>
      (case v-to-list v2 of
        Some vs =>
          (case vs-to-string vs of
```

```
          Some str =>
            Some ((s,t1), Rval (Litv(StrLit str)))
        | - => None
        )
    | - => None
    )
| (VfromList, [v2]) =>
      (case  v-to-list v2 of
          Some vs =>
            Some ((s,t1), Rval (Vectorv vs))
        | None => None
      )
| (Vsub, [Vectorv vs, Litv (IntLit i)]) =>
    if i <( 0 :: int) then
      Some ((s,t1), Rerr (Rraise (prim-exn (''Subscript''))))
    else
      (let n = (nat (abs ( i))) in
        if n ≥ List.length vs then
          Some ((s,t1), Rerr (Rraise (prim-exn (''Subscript''))))
        else
          Some ((s,t1), Rval (List.nth vs n)))
| (Vlength, [Vectorv vs]) =>
    Some ((s,t1), Rval (Litv (IntLit (int (List.length vs)))))
| (Aalloc, [Litv (IntLit n), v2]) =>
    if n <( 0 :: int) then
      Some ((s,t1), Rerr (Rraise (prim-exn (''Subscript''))))
    else
      (let (s',lnum) =
        (store-alloc (Varray (List.replicate (nat (abs ( n))) v2)) s)
      in
        Some ((s',t1), Rval (Loc lnum)))
| (AallocEmpty, [Conv None []]) =>
    (let (s',lnum) = (store-alloc (Varray []) s) in
      Some ((s',t1), Rval (Loc lnum)))
| (Asub, [Loc lnum, Litv (IntLit i)]) =>
    (case  store-lookup lnum s of
        Some (Varray vs) =>
          if i <( 0 :: int) then
            Some ((s,t1), Rerr (Rraise (prim-exn (''Subscript''))))
          else
            (let n = (nat (abs ( i))) in
              if n ≥ List.length vs then
                Some ((s,t1), Rerr (Rraise (prim-exn (''Subscript''))))
              else
                Some ((s,t1), Rval (List.nth vs n)))
      | - => None
    )
| (Alength, [Loc n]) =>
    (case  store-lookup n s of
```

```
          Some (Varray ws) =>
            Some ((s,t1),Rval (Litv(IntLit(int(List.length ws)))))
        | - => None
        )
  | (Aupdate, [Loc lnum, Litv (IntLit i), v2]) =>
      (case  store-lookup lnum s of
        Some (Varray vs) =>
          if i <( 0 :: int) then
            Some ((s,t1), Rerr (Rraise (prim-exn (''Subscript''))))
          else
            (let n = (nat (abs ( i))) in
              if n ≥ List.length vs then
                Some ((s,t1), Rerr (Rraise (prim-exn (''Subscript''))))
              else
                (case  store-assign lnum (Varray (List.list-update vs n v2)) s of
                    None => None
                  | Some s' => Some ((s',t1), Rval (Conv None []))
                ))
      | - => None
    )
  | (ConfigGC, [Litv (IntLit i), Litv (IntLit j)]) =>
      Some ((s,t1), Rval (Conv None []))
  | (FFI n, [Litv(StrLit conf), Loc lnum]) =>
      (case  store-lookup lnum s of
        Some (W8array ws) =>
         (case  call-FFI t1 n (List.map (λ c2 .  of-nat(of-char c2)) ( conf)) ws of
            (t', ws') =>
             (case  store-assign lnum (W8array ws') s of
               Some s' => Some ((s', t'), Rval (Conv None []))
             | None => None
             )
          )
      | - => None
      )
  | - => None
 ))


— Do a logical operation
— val do-log : lop -> v -> exp -> maybe exp-or-val
fun do-log  ::  lop ⇒ v ⇒ exp0 ⇒(exp-or-val)option   where
    do-log And v2 e = (
 (case  v2 of
    Litv - => None
  | Conv m l2 => (case  m of
                   None => None
                 | Some p => (case  p of
                                (s1,t1) =>
                             if(s1 = (''true'')) then
```

```
((case  t1 of
      TypeId i => (case  i of
                        Short s2 =>
                  if(s2 = (''bool'')) then
                    ((case  l2 of
                          [] => Some (Exp e)
                        | - => None
                      )) else None
                    | Long - - => None
                  )
   | TypeExn - => None
  )) else
(
if(s1 = (''false'')) then
  ((case  t1 of
        TypeId i2 => (case  i2 of
                          Short s4 =>
                    if(s4 = (''bool'')) then
                      ((case  l2 of
                            [] => Some
                                  (Val v2)
                          | - => None
                        )) else None
                      | Long - - =>
                    None
                    )
     | TypeExn - => None
    )) else None)
)
             )
    | Closure - - - => None
    | Recclosure - - - => None
    | Loc - => None
    | Vectorv - => None
  ) )
| do-log Or v2 e = (
  (case  v2 of
      Litv - => None
    | Conv m0 l6 => (case  m0 of
                        None => None
                      | Some p0 => (case  p0 of
                                        (s8,t0) =>
                                  if(s8 = (''false'')) then
                                    ((case  t0 of
                                          TypeId i5 => (case  i5 of
                                                            Short s9 =>
                                                      if(s9 = (''bool'')) then
                                                        ((case  l6 of
                                                              [] => Some
```

```
                                        (Exp e)
                                      | _ => None
                                    )) else None
                                   | Long _ _ =>
                                  None
                                  )
                       | TypeExn _ => None
                      )) else
                      (
                      if(s8 = (''true'')) then
                        ((case  t0 of
                            TypeId i8 => (case  i8 of
                                           Short s11 =>
                                          if(s11 = (''bool'')) then
                                            ((case  l6 of
                                                [] =>
                                             Some (Val v2)
                                               | _ =>
                                             None
                                             )) else None
                                            | Long _ _ =>
                                          None
                                          )
                          | TypeExn _ => None
                         )) else None)
                      )
              )
   | Closure _ _ _ => None
   | Recclosure _ _ _ => None
   | Loc _ => None
   | Vectorv _ => None
  ) )
```

— Do an if−then−else
— val do-if : v −> exp −> exp −> maybe exp

```
definition do-if ::  v ⇒ exp0 ⇒ exp0 ⇒(exp0)option   where
      do-if v2 e1 e2 = (
  if v2 = (Boolv True) then
    Some e1
  else if v2 = (Boolv False) then
    Some e2
  else
    None )
```

— Semantic helpers for definitions

— Build a constructor environment for the type definition tds

— *val build-tdefs : list modN −> list (list tvarN ∗ typeN ∗ list (conN ∗ list t)) −> env-ctor*
definition *build-tdefs* :: *(string)list* ⇒*((tvarN)list*∗*string*∗*(string*∗*(t)list)list)list* ⇒*((string),(string),(nat*∗*tid-or-exn))namespace* **where**

```
    build-tdefs mn tds = (
  alist-to-ns
    (List.rev
      (List.concat
        (List.map
          ( λx .
  (case  x of
      (tvs, tn, condefs) =>
  List.map
    ( λx .  (case  x of
                     (conN, ts) =>
                 (conN, (List.length ts, TypeId (mk-id mn tn)))
                 )) condefs
  ))
         tds))))
```

— *Checks that no constructor is defined twice in a type*
— *val check-dup-ctors : list (list tvarN ∗ typeN ∗ list (conN ∗ list t)) −> bool*
definition *check-dup-ctors* :: *((tvarN)list*∗*string*∗*(string*∗*(t)list)list)list* ⇒ *bool*
where

```
    check-dup-ctors tds = (
  Lem-list.allDistinct ((let x2 =
  ([]) in  List.foldr
   (λx .  (case  x of
                   (tvs, tn, condefs) => λ x2 .  List.foldr
                                                   (λx .
                                                    (case  x of
                                                        (n, ts) =>
                                                    λ x2 .
                                                      if True then
                                                        n # x2
                                                      else
                                                      x2
                                                    )) condefs
                                                 x2
                 )) tds x2)))
```

— *val combine-dec-result : forall 'a. sem-env v −> result (sem-env v) 'a −> result (sem-env v) 'a*
fun *combine-dec-result* :: *(v)sem-env* ⇒*(((v)sem-env),'a)result* ⇒*(((v)sem-env),'a)result*
where

```
    combine-dec-result env (Rerr e) = ( Rerr e )
| combine-dec-result env (Rval env') = ( Rval (| v = (nsAppend(v   env')(v   env)),
```

$c = (nsAppend(c \quad env')(c \quad env))$ |))

— *val extend-dec-env* : *sem-env v* −> *sem-env v* −> *sem-env v*
definition *extend-dec-env* :: (*v*)*sem-env* ⇒(*v*)*sem-env* ⇒(*v*)*sem-env* **where**
extend-dec-env new-env env = (
(| $v = (nsAppend(v \quad new\text{-}env)(v \quad env))$, $c = (nsAppend(c \quad new\text{-}env)(c \quad env))$
|))

— *val decs-to-types* : *list dec* −> *list typeN*
definition *decs-to-types* :: (*dec*)*list* ⇒(*string*)*list* **where**
decs-to-types ds = (
List.concat (*List.map* (λ *d* .
(*case d of*
Dtype locs tds => *List.map* (λ*x* .
(*case x of* (*tvs*,*tn*,*ctors*) => *tn*)) *tds*
| - => []))
ds))

— *val no-dup-types* : *list dec* −> *bool*
definition *no-dup-types* :: (*dec*)*list* ⇒ *bool* **where**
no-dup-types ds = (
Lem-list.allDistinct (*decs-to-types ds*))

— *val prog-to-mods* : *list top* −> *list* (*list modN*)
definition *prog-to-mods* :: (*top0*)*list* ⇒((*string*)*list*)*list* **where**
prog-to-mods tops = (
List.concat (*List.map* (λ *top1* .
(*case top1 of*
Tmod mn - - => [[*mn*]]
| - => []))
tops))

— *val no-dup-mods* : *list top* −> *set* (*list modN*) −> *bool*
definition *no-dup-mods* :: (*top0*)*list* ⇒((*modN*)*list*)*set* ⇒ *bool* **where**
no-dup-mods tops defined-mods2 = (
Lem-list.allDistinct (*prog-to-mods tops*) ∧
disjnt (*List.set* (*prog-to-mods tops*)) *defined-mods2*)

— *val prog-to-top-types* : *list top* −> *list typeN*
definition *prog-to-top-types* :: (*top0*)*list* ⇒(*string*)*list* **where**
prog-to-top-types tops = (
List.concat (*List.map* (λ *top1* .
(*case top1 of*

```
          Tdec d => decs-to-types [d]
        | - => [] ))
    tops))
```

— *val no-dup-top-types : list top −> set tid-or-exn −> bool*
definition *no-dup-top-types* :: (*top0*)*list* ⇒(*tid-or-exn*)*set* ⇒ *bool* **where**

```
     no-dup-top-types tops defined-types2 = (
  Lem-list.allDistinct (prog-to-top-types tops) ∧
  disjnt (List.set (List.map (λ tn .  TypeId (Short tn)) (prog-to-top-types tops)))
defined-types2 )
```

end

Chapter 8

Generated by Lem from *semantics/alt-semantics/smallStep.lem*.

theory *SmallStep*

imports
  Main
  HOL−Library.Datatype-Records
  LEM.Lem-pervasives-extra
  Lib
  Namespace
  Ast
  SemanticPrimitives
  Ffi

begin

— *open import Pervasives-extra*
— *open import Lib*
— *open import Ast*
— *open import Namespace*
— *open import SemanticPrimitives*
— *open import Ffi*

— *Small−step semantics for expression only. Modules and definitions have ∗ big−step semantics only*

— *Evaluation contexts ∗ The hole is denoted by the unit type ∗ The env argument contains bindings for the free variables of expressions in the context*
datatype *ctxt-frame* =
    Craise unit
  | *Chandle unit (pat ∗ exp0) list*

| *Capp op0 v list unit exp0 list*
| *Clog lop unit exp0*
| *Cif unit exp0 exp0*
— *The value is raised if none of the patterns match*
| *Cmat unit (pat ∗ exp0) list v*
| *Clet varN option unit exp0*
— *Evaluating a constructor's arguments ∗ The v list should be in reverse order.*
| *Ccon ((modN, conN)id0)option v list unit exp0 list*
| *Ctannot unit t*
| *Clannot unit locs*

type-synonym *ctxt = ctxt-frame ∗ v sem-env*

— *State for CEK−style expression evaluation ∗ − constructor data ∗ − the store ∗ − the environment for the free variables of the current expression ∗ − the current expression to evaluate, or a value if finished ∗ − the context stack (continuation) of what to do once the current expression ∗ is finished. Each entry has an environment for it's free variables*

type-synonym *'ffi small-state = v sem-env ∗ ('ffi, v) store-ffi ∗ exp-or-val ∗ ctxt list*

datatype *'ffi e-step-result =*
Estep 'ffi small-state
| *Eabort abort*
| *Estuck*

— *The semantics are deterministic, and presented functionally instead of ∗ relationally for proof rather that readability; the steps are very small: we ∗ push individual frames onto the context stack instead of finding a redex in a ∗ single step*

— *val push : forall 'ffi. sem-env v −> store-ffi 'ffi v −> exp −> ctxt-frame −> list ctxt −> e-step-result 'ffi*
definition *push* :: *(v)sem-env ⇒(v)store∗'ffi ffi-state ⇒ exp0 ⇒ ctxt-frame ⇒(ctxt-frame∗(v)sem-env)list ⇒ 'ffi e-step-result* **where**
push env s e c' cs = (Estep (env, s, Exp e, ((c',env)# cs)))

— *val return : forall 'ffi. sem-env v −> store-ffi 'ffi v −> v −> list ctxt −> e-step-result 'ffi*
definition *return* :: *(v)sem-env ⇒(v)store∗'ffi ffi-state ⇒ v ⇒(ctxt)list ⇒ 'ffi e-step-result* **where**
return env s v2 c2 = (Estep (env, s, Val v2, c2))

— *val application : forall 'ffi. op −> sem-env v −> store-ffi 'ffi v −> list v −> list ctxt −> e-step-result 'ffi*
definition *application* :: *op0 ⇒(v)sem-env ⇒(v)store∗'ffi ffi-state ⇒(v)list ⇒(ctxt)list ⇒ 'ffi e-step-result* **where**
application op1 env s vs c2 = (

```
  (case  op1 of
      Opapp =>
      (case  do-opapp vs of
          Some (env,e) => Estep (env, s, Exp e, c2)
        | None => Eabort Rtype-error
      )
    | - =>
      (case  do-app s op1 vs of
          Some (s',r) =>
          (case  r of
              Rerr (Rraise v2) => Estep (env,s',Val v2,((Craise () ,env)# c2))
            | Rerr (Rabort a) => Eabort a
            | Rval v2 => return env s' v2 c2
          )
        | None => Eabort Rtype-error
      )
    ))


— apply a context to a value
— val continue : forall 'ffi. store-ffi 'ffi v -> v -> list ctxt -> e-step-result 'ffi
fun continue  :: (v)store∗'ffi ffi-state ⇒ v ⇒(ctxt-frame∗(v)sem-env)list ⇒ 'ffi
e-step-result  where
      continue s v2 ([]) = ( Estuck )
| continue s v2 ((Craise -, env) # c2) = (
        (case  c2 of
            [] => Estuck
          | ((Chandle - pes,env') # c2) =>
              Estep (env,s,Val v2,((Cmat ()  pes v2, env')# c2))
          | - # c2 => Estep (env,s,Val v2,((Craise () ,env)# c2))
        ))
| continue s v2 ((Chandle - pes, env) # c2) = (
        return env s v2 c2 )
| continue s v2 ((Capp op1 vs - [], env) # c2) = (
        application op1 env s (v2 # vs) c2 )
| continue s v2 ((Capp op1 vs - (e # es), env) # c2) = (
        push env s e (Capp op1 (v2 # vs) ()  es) c2 )
| continue s v2 ((Clog l - e, env) # c2) = (
        (case  do-log l v2 e of
            Some (Exp e) => Estep (env, s, Exp e, c2)
          | Some (Val v2) => return env s v2 c2
          | None => Eabort Rtype-error
        ))
| continue s v2 ((Cif - e1 e2, env) # c2) = (
        (case  do-if v2 e1 e2 of
            Some e => Estep (env, s, Exp e, c2)
          | None => Eabort Rtype-error
        ))
| continue s v2 ((Cmat - [] err-v, env) # c2) = (
```

```
        Estep (env, s, Val err-v, ((Craise () , env) # c2)))
| continue s v2 ((Cmat - ((p,e)# pes) err-v, env) # c2) = (
        if Lem-list.allDistinct (pat-bindings p []) then
          (case  pmatch(c   env) (fst s) p v2 [] of
              Match-type-error => Eabort Rtype-error
            | No-match => Estep (env, s, Val v2, ((Cmat ()  pes err-v,env)# c2))
              | Match env' => Estep (( env (| v := (nsAppend (alist-to-ns env')(v
env)) |)), s, Exp e, c2)
          )
        else
          Eabort Rtype-error )
| continue s v2 ((Clet n - e, env) # c2) = (
        Estep (( env (| v := (nsOptBind n v2(v   env)) |)), s, Exp e, c2))
| continue s v2 ((Ccon n vs - [], env) # c2) = (
        if do-con-check(c   env) n (List.length vs +( 1 :: nat)) then
           (case  build-conv(c   env) n (v2 # vs) of
               None => Eabort Rtype-error
             | Some v2 => return env s v2 c2
           )
        else
          Eabort Rtype-error )
| continue s v2 ((Ccon n vs - (e # es), env) # c2) = (
         if do-con-check(c   env) n (((List.length vs +( 1 :: nat)) +( 1 :: nat)) +
List.length es) then
          push env s e (Ccon n (v2 # vs) ()  es) c2
        else
          Eabort Rtype-error )
| continue s v2 ((Ctannot - t1, env) # c2) = (
        return env s v2 c2 )
| continue s v2 ((Clannot - l, env) # c2) = (
        return env s v2 c2 )
```

— *The single step expression evaluator. Returns None if there is nothing to ∗ do, but no type error. Returns Type-error on encountering free variables, ∗ mis−applied (or non−existent) constructors, and when the wrong kind of value ∗ if given to a primitive. Returns Bind-error when no pattern in a match ∗ matches the value. Otherwise it returns the next state*

— *val e-step : forall 'ffi. small-state 'ffi −> e-step-result 'ffi*

```
fun e-step :: (v)sem-env∗((v)store∗'ffi ffi-state)∗exp-or-val∗(ctxt)list ⇒ 'ffi e-step-result
where
      e-step (env, s,(Val v2), c2) = (
        continue s v2 c2 )
| e-step (env, s,(Exp e), c2) = (
        (case  e of
            Lit l => return env s (Litv l) c2
          | Raise e =>
              push env s e (Craise () ) c2
```

```
        | Handle e pes =>
            push env s e (Chandle ()  pes) c2
        | Con n es =>
            if do-con-check(c   env) n (List.length es) then
              (case  List.rev es of
                  [] =>
                    (case  build-conv(c   env) n [] of
                        None => Eabort Rtype-error
                      | Some v2 => return env s v2 c2
                    )
                | e # es =>
                    push env s e (Ccon n [] ()  es) c2
              )
            else
              Eabort Rtype-error
        | Var n =>
            (case  nsLookup(v   env) n of
                None => Eabort Rtype-error
              | Some v2 =>
                  return env s v2 c2
            )
        | Fun n e => return env s (Closure env n e) c2
        | App op1 es =>
            (case  List.rev es of
                [] => application op1 env s [] c2
              | (e # es) => push env s e (Capp op1 [] ()  es) c2
            )
        | Log l e1 e2 => push env s e1 (Clog l ()  e2) c2
        | If e1 e2 e3 => push env s e1 (Cif ()  e2 e3) c2
         | Mat e pes => push env s e (Cmat ()  pes (Conv (Some ((''Bind''),
TypeExn (Short (''Bind'')))) [])) c2
        | Let n e1 e2 => push env s e1 (Clet n ()  e2) c2
        | Letrec funs e =>
            if ¬ (allDistinct (List.map ( λx .
  (case  x of (x,y,z) => x )) funs)) then
              Eabort Rtype-error
            else
              Estep (( env (| v := (build-rec-env funs env(v   env)) |)),
                     s, Exp e, c2)
        | Tannot e t1 => push env s e (Ctannot ()  t1) c2
        | Lannot e l => push env s e (Clannot ()  l) c2
      ))
```

— *Define a semantic function using the steps*

— *val e-step-reln : forall 'ffi. small-state 'ffi −> small-state 'ffi −> bool*
— *val small-eval : forall 'ffi. sem-env v −> store-ffi 'ffi v −> exp −> list ctxt −> store-ffi 'ffi v ∗ result v v −> bool*

definition *e-step-reln* :: (v)*sem-env*$*$('*ffi*,(v))*store-ffi*$*$*exp-or-val*$*$(*ctxt*)*list* ⇒(v)*sem-env*$*$('*ffi*,(v))*store-ffi*$*$*exp*
⇒ *bool* **where**
e-step-reln st1 st2 = (
(*e-step st1* = *Estep st2*))

fun
small-eval :: (v)*sem-env* ⇒(v)*store*$*$'*ffi ffi-state* ⇒ *exp0* ⇒(*ctxt*)*list* ⇒((v)*store*$*$'*ffi ffi-state*)$*$((v),(v))*result* ⇒ *bool* **where**

small-eval env s e c2 (*s*′, *Rval v2*) = ((
∃ *env*′. (*rtranclp* (*e-step-reln*)) (*env*,*s*,*Exp e*,*c2*) (*env*′,*s*′,*Val v2*,[])))
|
small-eval env s e c2 (*s*′, *Rerr* (*Rraise v2*)) = ((
∃ *env*′.
∃ *env*′′. (*rtranclp* (*e-step-reln*)) (*env*,*s*,*Exp e*,*c2*) (*env*′,*s*′,*Val v2*,[(*Craise* () , *env*′′)])))
|
small-eval env s e c2 (*s*′, *Rerr* (*Rabort a*)) = ((
∃ *env*′.
∃ *e*′.
∃ *c*′.
(*rtranclp* (*e-step-reln*)) (*env*,*s*,*Exp e*,*c2*) (*env*′,*s*′,*e*′,*c*′) ∧
(*e-step* (*env*′,*s*′,*e*′,*c*′) = *Eabort a*)))

— *val e-diverges* : *forall* '*ffi*. *sem-env v* −> *store-ffi* '*ffi v* −> *exp* −> *bool*
definition *e-diverges* :: (v)*sem-env* ⇒(v)*store*$*$'*ffi ffi-state* ⇒ *exp0* ⇒ *bool* **where**

e-diverges env s e = ((
∀ *env*′.
∀ *s*′.
∀ *e*′.
∀ *c*′.
(*rtranclp* (*e-step-reln*)) (*env*,*s*,*Exp e*,[]) (*env*′,*s*′,*e*′,*c*′)
⟶
((∃ *env*′′. ∃ *s*′′. ∃ *e*′′. ∃ *c*′′.
e-step-reln (*env*′,*s*′,*e*′,*c*′) (*env*′′,*s*′′,*e*′′,*c*′′)))))

end

Chapter 9

Generated by Lem from *semantics/alt-semantics/bigStep.lem*.

theory *BigStep*

imports
Main
HOL−Library.Datatype-Records
LEM.Lem-pervasives-extra
Lib
Namespace
Ast
SemanticPrimitives
Ffi
SmallStep

begin

— *open import Pervasives-extra*
— *open import Lib*
— *open import Namespace*
— *open import Ast*
— *open import SemanticPrimitives*
— *open import Ffi*

— *To get the definition of expression divergence to use in defining definition ∗ divergence*
— *open import SmallStep*

— −−−−−−−−−−−−−−−−−−−−−−−−−− *Big step semantics* −−−−−−−−−−−−−−−−−−−−−−−−−−−−−

— *If the first argument is true, the big step semantics counts down how many* *functions applications have happened, and raises an exception when the counter runs out.*

inductive
evaluate-match :: *bool* ⇒(*v*)*sem-env* ⇒ *'ffi state* ⇒ *v* ⇒(*pat*∗*exp0*)*list* ⇒ *v* ⇒ *'ffi state*∗((*v*),(*v*))*result* ⇒ *bool*
and
evaluate-list :: *bool* ⇒(*v*)*sem-env* ⇒ *'ffi state* ⇒(*exp0*)*list* ⇒ *'ffi state*∗(((*v*)*list*),(*v*))*result* ⇒ *bool*
and
evaluate :: *bool* ⇒(*v*)*sem-env* ⇒ *'ffi state* ⇒ *exp0* ⇒ *'ffi state*∗((*v*),(*v*))*result* ⇒ *bool* **where**

lit : ⋀ *ck env l s.*

evaluate ck env s (*Lit l*) (*s*, *Rval* (*Litv l*))

|

raise1 : ⋀ *ck env e s1 s2 v1*.
evaluate ck s1 env e (*s2*, *Rval v1*)
==>
evaluate ck s1 env (*Raise e*) (*s2*, *Rerr* (*Rraise v1*))

|

raise2 : ⋀ *ck env e s1 s2 err*.
evaluate ck s1 env e (*s2*, *Rerr err*)
==>
evaluate ck s1 env (*Raise e*) (*s2*, *Rerr err*)

|

handle1 : ⋀ *ck s1 s2 env e v1 pes.*
evaluate ck s1 env e (*s2*, *Rval v1*)
==>
evaluate ck s1 env (*Handle e pes*) (*s2*, *Rval v1*)

|

handle2 : ⋀ *ck s1 s2 env e pes v1 bv.*
evaluate ck env s1 e (*s2*, *Rerr* (*Rraise v1*)) ∧
evaluate-match ck env s2 v1 pes v1 bv
==>
evaluate ck env s1 (*Handle e pes*) *bv*

|

handle3 : ⋀ *ck s1 s2 env e pes a.*

$\textit{evaluate ck env s1 e}\ (\textit{s2},\ \textit{Rerr}\ (\textit{Rabort a}))$
==>
$\textit{evaluate ck env s1}\ (\textit{Handle e pes})\ (\textit{s2},\ \textit{Rerr}\ (\textit{Rabort a}))$

|

$\textit{con1} : \bigwedge \textit{ck env cn es vs s}\ s'\ \textit{v1}.$
$\textit{do-con-check}(c \quad \textit{env})\ \textit{cn}\ (\textit{List.length es}) \wedge$
$((\textit{build-conv}(c \quad \textit{env})\ \textit{cn}\ (\textit{List.rev vs}) = \textit{Some v1}) \wedge$
$\textit{evaluate-list ck env s}\ (\textit{List.rev es})\ (s',\ \textit{Rval vs}))$
==>
$\textit{evaluate ck env s}\ (\textit{Con cn es})\ (s',\ \textit{Rval v1})$

|

$\textit{con2} : \bigwedge \textit{ck env cn es s}.$
$\neg\ (\textit{do-con-check}(c \quad \textit{env})\ \textit{cn}\ (\textit{List.length es}))$
==>
$\textit{evaluate ck env s}\ (\textit{Con cn es})\ (s,\ \textit{Rerr}\ (\textit{Rabort Rtype-error}))$

|

$\textit{con3} : \bigwedge \textit{ck env cn es err s}\ s'.$
$\textit{do-con-check}(c \quad \textit{env})\ \textit{cn}\ (\textit{List.length es}) \wedge$
$\textit{evaluate-list ck env s}\ (\textit{List.rev es})\ (s',\ \textit{Rerr err})$
==>
$\textit{evaluate ck env s}\ (\textit{Con cn es})\ (s',\ \textit{Rerr err})$

|

$\textit{var1} : \bigwedge \textit{ck env n v1 s}.$
$\textit{nsLookup}(v \quad \textit{env})\ n = \textit{Some v1}$
==>
$\textit{evaluate ck env s}\ (\textit{Var n})\ (s,\ \textit{Rval v1})$

|

$\textit{var2} : \bigwedge \textit{ck env n s}.$
$\textit{nsLookup}(v \quad \textit{env})\ n = \textit{None}$
==>
$\textit{evaluate ck env s}\ (\textit{Var n})\ (s,\ \textit{Rerr}\ (\textit{Rabort Rtype-error}))$

|

$\textit{fn} : \bigwedge \textit{ck env n e s}.$

$\textit{evaluate ck env s}\ (\textit{Fun n e})\ (s,\ \textit{Rval}\ (\textit{Closure env n e}))$

|

app1 : $\bigwedge$ *ck env es vs env′ e bv s1 s2*.
evaluate-list ck env s1 (*List.rev es*) (*s2*, *Rval vs*) ∧
((*do-opapp* (*List.rev vs*) = *Some* (*env′*, *e*)) ∧
((*ck* ⟶ ¬ ((*clock s2*) =((*0* :: *nat*)))) ∧
evaluate ck env′ (*if ck then* (*s2* (| *clock* := ((*clock s2*) −(*1* :: *nat*)) |)) *else s2*)
e bv))
==>
evaluate ck env s1 (*App Opapp es*) *bv*

|

app2 : $\bigwedge$ *ck env es vs env′ e s1 s2*.
evaluate-list ck env s1 (*List.rev es*) (*s2*, *Rval vs*) ∧
((*do-opapp* (*List.rev vs*) = *Some* (*env′*, *e*)) ∧
(((*clock s2*) =(*0* :: *nat*)) ∧
ck))
==>
evaluate ck env s1 (*App Opapp es*) (*s2*, *Rerr* (*Rabort Rtimeout-error*))

|

app3 : $\bigwedge$ *ck env es vs s1 s2*.
evaluate-list ck env s1 (*List.rev es*) (*s2*, *Rval vs*) ∧
(*do-opapp* (*List.rev vs*) = *None*)
==>
evaluate ck env s1 (*App Opapp es*) (*s2*, *Rerr* (*Rabort Rtype-error*))

|

app4 : $\bigwedge$ *ck env op0 es vs res s1 s2 refs′ ffi′*.
evaluate-list ck env s1 (*List.rev es*) (*s2*, *Rval vs*) ∧
((*do-app* ((*refs s2*),(*ffi s2*)) *op0* (*List.rev vs*) = *Some* ((*refs′*,*ffi′*), *res*)) ∧
(*op0* ≠ *Opapp*))
==>
evaluate ck env s1 (*App op0 es*) ((*s2* (| *refs* := *refs′*, *ffi* :=*ffi′* |)), *res*)

|

app5 : $\bigwedge$ *ck env op0 es vs s1 s2*.
evaluate-list ck env s1 (*List.rev es*) (*s2*, *Rval vs*) ∧
((*do-app* ((*refs s2*),(*ffi s2*)) *op0* (*List.rev vs*) = *None*) ∧
(*op0* ≠ *Opapp*))
==>
evaluate ck env s1 (*App op0 es*) (*s2*, *Rerr* (*Rabort Rtype-error*))

|

app6 : $\bigwedge$ *ck env op0 es err s1 s2*.

evaluate-list ck env s1 (*List.rev es*) (*s2*, *Rerr err*)
==>
evaluate ck env s1 (*App op0 es*) (*s2*, *Rerr err*)

|

log1 : ⋀ *ck env op0 e1 e2 v1 e′ bv s1 s2*.
evaluate ck env s1 e1 (*s2*, *Rval v1*) ∧
((*do-log op0 v1 e2* = *Some* (*Exp e′*)) ∧
evaluate ck env s2 e′ bv)
==>
evaluate ck env s1 (*Log op0 e1 e2*) *bv*

|

log2 : ⋀ *ck env op0 e1 e2 v1 bv s1 s2*.
evaluate ck env s1 e1 (*s2*, *Rval v1*) ∧
(*do-log op0 v1 e2* = *Some* (*Val bv*))
==>
evaluate ck env s1 (*Log op0 e1 e2*) (*s2*, *Rval bv*)

|

log3 : ⋀ *ck env op0 e1 e2 v1 s1 s2*.
evaluate ck env s1 e1 (*s2*, *Rval v1*) ∧
(*do-log op0 v1 e2* = *None*)
==>
evaluate ck env s1 (*Log op0 e1 e2*) (*s2*, *Rerr* (*Rabort Rtype-error*))

|

log4 : ⋀ *ck env op0 e1 e2 err s s′*.
evaluate ck env s e1 (*s′*, *Rerr err*)
==>
evaluate ck env s (*Log op0 e1 e2*) (*s′*, *Rerr err*)

|

if1 : ⋀ *ck env e1 e2 e3 v1 e′ bv s1 s2*.
evaluate ck env s1 e1 (*s2*, *Rval v1*) ∧
((*do-if v1 e2 e3* = *Some e′*) ∧
evaluate ck env s2 e′ bv)
==>
evaluate ck env s1 (*If e1 e2 e3*) *bv*

|

if2 : ⋀ *ck env e1 e2 e3 v1 s1 s2*.
evaluate ck env s1 e1 (*s2*, *Rval v1*) ∧

(*do-if v1 e2 e3* = *None*)
==>
evaluate ck env s1 (*If e1 e2 e3*) (*s2*, *Rerr* (*Rabort Rtype-error*))

|

if3 : $\bigwedge$ *ck env e1 e2 e3 err s s′*.
evaluate ck env s e1 (*s′*, *Rerr err*)
==>
evaluate ck env s (*If e1 e2 e3*) (*s′*, *Rerr err*)

|

mat1 : $\bigwedge$ *ck env e pes v1 bv s1 s2*.
evaluate ck env s1 e (*s2*, *Rval v1*) ∧
evaluate-match ck env s2 v1 pes (*Conv* (*Some* ((′′*Bind*′′), *TypeExn* (*Short* (′′*Bind*′′))))
[]) *bv*
==>
evaluate ck env s1 (*Mat e pes*) *bv*

|

mat2 : $\bigwedge$ *ck env e pes err s s′*.
evaluate ck env s e (*s′*, *Rerr err*)
==>
evaluate ck env s (*Mat e pes*) (*s′*, *Rerr err*)

|

let1 : $\bigwedge$ *ck env n e1 e2 v1 bv s1 s2*.
evaluate ck env s1 e1 (*s2*, *Rval v1*) ∧
evaluate ck (*env* (| *v* := (*nsOptBind n v1*(*v* *env*)) |)) *s2 e2 bv*
==>
evaluate ck env s1 (*Let n e1 e2*) *bv*

|

let2 : $\bigwedge$ *ck env n e1 e2 err s s′*.
evaluate ck env s e1 (*s′*, *Rerr err*)
==>
evaluate ck env s (*Let n e1 e2*) (*s′*, *Rerr err*)

|

letrec1 : $\bigwedge$ *ck env funs e bv s*.
Lem-list.allDistinct (*List.map* (λ*x* .
 (*case x of* (*x*,*y*,*z*) => *x*)) *funs*) ∧
evaluate ck (*env* (| *v* := (*build-rec-env funs env*(*v* *env*)) |)) *s e bv*
==>

evaluate ck env s (*Letrec funs e*) *bv*

|

letrec2 : $\bigwedge$ *ck env funs e s*.
¬ (*Lem-list.allDistinct* (*List.map* (λ*x* .
(*case x of* (*x*,*y*,*z*) => *x*)) *funs*))
==>
evaluate ck env s (*Letrec funs e*) (*s*, *Rerr* (*Rabort Rtype-error*))

|

tannot : $\bigwedge$ *ck env e t0 s bv*.
evaluate ck env s e bv
==>
evaluate ck env s (*Tannot e t0*) *bv*

|

locannot : $\bigwedge$ *ck env e l s bv*.
evaluate ck env s e bv
==>
evaluate ck env s (*Lannot e l*) *bv*

|

empty : $\bigwedge$ *ck env s*.

evaluate-list ck env s [] (*s*, *Rval* [])

|

cons1 : $\bigwedge$ *ck env e es v1 vs s1 s2 s3*.
evaluate ck env s1 e (*s2*, *Rval v1*) ∧
evaluate-list ck env s2 es (*s3*, *Rval vs*)
==>
evaluate-list ck env s1 (*e* # *es*) (*s3*, *Rval* (*v1* # *vs*))

|

cons2 : $\bigwedge$ *ck env e es err s s'*.
evaluate ck env s e (*s'*, *Rerr err*)
==>
evaluate-list ck env s (*e* # *es*) (*s'*, *Rerr err*)

|

cons3 : $\bigwedge$ *ck env e es v1 err s1 s2 s3*.
evaluate ck env s1 e (*s2*, *Rval v1*) ∧

evaluate-list ck env s2 es (*s3*, *Rerr err*)
==>
evaluate-list ck env s1 (*e* # *es*) (*s3*, *Rerr err*)

|

mat-empty : $\bigwedge$ *ck env v1 err-v s.*

evaluate-match ck env s v1 [] *err-v* (*s*, *Rerr* (*Rraise err-v*))

|

mat-cons1 : $\bigwedge$ *ck env env′ v1 p pes e bv err-v s.*
Lem-list.allDistinct (*pat-bindings p* []) ∧
((*pmatch*(*c* *env*)(*refs* *s*) *p v1* [] = *Match env′*) ∧
evaluate ck (*env* (| *v* := (*nsAppend* (*alist-to-ns env′*)(*v* *env*)) |)) *s e bv*)
==>
evaluate-match ck env s v1 ((*p*,*e*)# *pes*) *err-v bv*

|

mat-cons2 : $\bigwedge$ *ck env v1 p e pes bv s err-v.*
Lem-list.allDistinct (*pat-bindings p* []) ∧
((*pmatch*(*c* *env*)(*refs* *s*) *p v1* [] = *No-match*) ∧
evaluate-match ck env s v1 pes err-v bv)
==>
evaluate-match ck env s v1 ((*p*,*e*)# *pes*) *err-v bv*

|

mat-cons3 : $\bigwedge$ *ck env v1 p e pes s err-v.*
pmatch(*c* *env*)(*refs* *s*) *p v1* [] = *Match-type-error*
==>
evaluate-match ck env s v1 ((*p*,*e*)# *pes*) *err-v* (*s*, *Rerr* (*Rabort Rtype-error*))

|

mat-cons4 : $\bigwedge$ *ck env v1 p e pes s err-v.*
¬ (*Lem-list.allDistinct* (*pat-bindings p* []))
==>
evaluate-match ck env s v1 ((*p*,*e*)# *pes*) *err-v* (*s*, *Rerr* (*Rabort Rtype-error*))

— *The set tid-or-exn part of the state tracks all of the types and exceptions* ∗ *that have been declared*
inductive
evaluate-dec :: *bool* ⇒(*modN*)*list* ⇒(*v*)*sem-env* ⇒ ′*ffi state* ⇒ *dec* ⇒ ′*ffi state*∗(((*v*)*sem-env*),(*v*))*result* ⇒ *bool* **where**

dlet1 : $\bigwedge$ *ck mn env p e v1 env′ s1 s2 locs.*

evaluate ck env s1 e (s2, Rval v1) ∧
(Lem-list.allDistinct (pat-bindings p []) ∧
(pmatch(c env)(refs s2) p v1 [] = Match env′))
==>
evaluate-dec ck mn env s1 (Dlet locs p e) (s2, Rval (| v = (alist-to-ns env′), c =
nsEmpty |))

|

dlet2 : ⋀ ck mn env p e v1 s1 s2 locs.
evaluate ck env s1 e (s2, Rval v1) ∧
(Lem-list.allDistinct (pat-bindings p []) ∧
(pmatch(c env)(refs s2) p v1 [] = No-match))
==>
evaluate-dec ck mn env s1 (Dlet locs p e) (s2, Rerr (Rraise Bindv))

|

dlet3 : ⋀ ck mn env p e v1 s1 s2 locs.
evaluate ck env s1 e (s2, Rval v1) ∧
(Lem-list.allDistinct (pat-bindings p []) ∧
(pmatch(c env)(refs s2) p v1 [] = Match-type-error))
==>
evaluate-dec ck mn env s1 (Dlet locs p e) (s2, Rerr (Rabort Rtype-error))

|

dlet4 : ⋀ ck mn env p e s locs.
¬ (Lem-list.allDistinct (pat-bindings p []))
==>
evaluate-dec ck mn env s (Dlet locs p e) (s, Rerr (Rabort Rtype-error))

|

dlet5 : ⋀ ck mn env p e err s s′ locs.
evaluate ck env s e (s′, Rerr err) ∧
Lem-list.allDistinct (pat-bindings p [])
==>
evaluate-dec ck mn env s (Dlet locs p e) (s′, Rerr err)

|

dletrec1 : ⋀ ck mn env funs s locs.
Lem-list.allDistinct (List.map (λx .
(case x of (x,y,z) => x)) funs)
==>
evaluate-dec ck mn env s (Dletrec locs funs) (s, Rval (| v = (build-rec-env funs env
nsEmpty), c = nsEmpty |))

|

dletrec2 : $\bigwedge$ *ck mn env funs s locs.*
¬ (*Lem-list.allDistinct* (*List.map* (λx .
(*case* *x of* (*x*,*y*,*z*) => *x*)) *funs*))
==>
evaluate-dec ck mn env s (*Dletrec locs funs*) (*s*, *Rerr* (*Rabort Rtype-error*))

|

dtype1 : $\bigwedge$ *ck mn env tds s new-tdecs locs.*
check-dup-ctors tds ∧
((*new-tdecs* = *type-defs-to-new-tdecs mn tds*) ∧
(*disjnt new-tdecs*(*defined-types* *s*) ∧
Lem-list.allDistinct (*List.map* (λx .
(*case* *x of* (*tvs*,*tn*,*ctors*) => *tn*)) *tds*)))
==>
evaluate-dec ck mn env s (*Dtype locs tds*) ((*s* (| *defined-types* := (*new-tdecs* ∪(*defined-types* *s*)) |)), *Rval* (| *v* = *nsEmpty*, *c* = (*build-tdefs mn tds*) |))

|

dtype2 : $\bigwedge$ *ck mn env tds s locs.*
¬ (*check-dup-ctors tds*) ∨
(¬ (*disjnt* (*type-defs-to-new-tdecs mn tds*)(*defined-types* *s*)) ∨
¬ (*Lem-list.allDistinct* (*List.map* (λx .
(*case* *x of* (*tvs*,*tn*,*ctors*) => *tn*)) *tds*)))
==>
evaluate-dec ck mn env s (*Dtype locs tds*) (*s*, *Rerr* (*Rabort Rtype-error*))

|

dtabbrev : $\bigwedge$ *ck mn env tvs tn t0 s locs.*

evaluate-dec ck mn env s (*Dtabbrev locs tvs tn t0*) (*s*, *Rval* (| *v* = *nsEmpty*, *c* = *nsEmpty* |))

|

dexn1 : $\bigwedge$ *ck mn env cn ts s locs.*
¬ (*TypeExn* (*mk-id mn cn*) ∈(*defined-types* *s*))
==>
evaluate-dec ck mn env s (*Dexn locs cn ts*) ((*s* (| *defined-types* := ({*TypeExn* (*mk-id mn cn*)} ∪(*defined-types* *s*)) |)), *Rval* (| *v* = *nsEmpty*, *c* = (*nsSing cn* (*List.length ts*, *TypeExn* (*mk-id mn cn*))) |))

|

dexn2 : $\bigwedge$ *ck mn env cn ts s locs.*

TypeExn (*mk-id mn cn*) ∈(*defined-types* *s*)
==>
evaluate-dec ck mn env s (*Dexn locs cn ts*) (*s*, *Rerr* (*Rabort Rtype-error*))

inductive
evaluate-decs :: *bool* ⇒(*modN*)*list* ⇒(*v*)*sem-env* ⇒ ′*ffi state* ⇒(*dec*)*list* ⇒ ′*ffi state*∗(((*v*)*sem-env*),(*v*))*result* ⇒ *bool* **where**

empty : ⋀ *ck mn env s*.

evaluate-decs ck mn env s [] (*s*, *Rval* (| *v* = *nsEmpty*, *c* = *nsEmpty* |))

|

cons1 : ⋀ *ck mn s1 s2 env d ds e*.
evaluate-dec ck mn env s1 d (*s2*, *Rerr e*)
==>
evaluate-decs ck mn env s1 (*d* # *ds*) (*s2*, *Rerr e*)

|

cons2 : ⋀ *ck mn s1 s2 s3 env d ds new-env r*.
evaluate-dec ck mn env s1 d (*s2*, *Rval new-env*) ∧
evaluate-decs ck mn (*extend-dec-env new-env env*) *s2 ds* (*s3*, *r*)
==>
evaluate-decs ck mn env s1 (*d* # *ds*) (*s3*, *combine-dec-result new-env r*)

inductive
evaluate-top :: *bool* ⇒(*v*)*sem-env* ⇒ ′*ffi state* ⇒ *top0* ⇒ ′*ffi state*∗(((*v*)*sem-env*),(*v*))*result* ⇒ *bool* **where**

tdec1 : ⋀ *ck s1 s2 env d new-env*.
evaluate-dec ck [] *env s1 d* (*s2*, *Rval new-env*)
==>
evaluate-top ck env s1 (*Tdec d*) (*s2*, *Rval new-env*)
|

tdec2 : ⋀ *ck s1 s2 env d err*.
evaluate-dec ck [] *env s1 d* (*s2*, *Rerr err*)
==>
evaluate-top ck env s1 (*Tdec d*) (*s2*, *Rerr err*)

|

tmod1 : ⋀ *ck s1 s2 env ds mn specs new-env*.
¬ ([*mn*] ∈(*defined-mods* *s1*)) ∧
(*no-dup-types ds* ∧
evaluate-decs ck [*mn*] *env s1 ds* (*s2*, *Rval new-env*))
==>

evaluate-top ck env s1 (Tmod mn specs ds) ((s2 (| defined-mods := ({[mn]} ∪(defined-mods s2)) |)), Rval (| v = (nsLift mn(v new-env)), c = (nsLift mn(c new-env)) |))

|

tmod2 : $\bigwedge$ *ck s1 s2 env ds mn specs err.*
¬ (*[mn]* ∈(*defined-mods s1*)) ∧
(*no-dup-types ds* ∧
evaluate-decs ck [mn] env s1 ds (s2, Rerr err))
==>
evaluate-top ck env s1 (Tmod mn specs ds) ((s2 (| defined-mods := ({[mn]} ∪(defined-mods s2)) |)), Rerr err)

|

tmod3 : $\bigwedge$ *ck s1 env ds mn specs.*
¬ (*no-dup-types ds*)
==>
evaluate-top ck env s1 (Tmod mn specs ds) (s1, Rerr (Rabort Rtype-error))

|

tmod4 : $\bigwedge$ *ck env s mn specs ds.*
[mn] ∈(*defined-mods s*)
==>
evaluate-top ck env s (Tmod mn specs ds) (s, Rerr (Rabort Rtype-error))

inductive
evaluate-prog :: *bool* ⇒(*v*)*sem-env* ⇒ *'ffi state* ⇒ *prog* ⇒ *'ffi state*∗(((*v*)*sem-env*),(*v*))*result* ⇒ *bool* **where**

empty : $\bigwedge$ *ck env s.*

evaluate-prog ck env s [] (s, Rval (| v = nsEmpty, c = nsEmpty |))

|

cons1 : $\bigwedge$ *ck s1 s2 s3 env top0 tops new-env r.*
evaluate-top ck env s1 top0 (s2, Rval new-env) ∧
evaluate-prog ck (extend-dec-env new-env env) s2 tops (s3,r)
==>
evaluate-prog ck env s1 (top0 # tops) (s3, combine-dec-result new-env r)

|

cons2 : $\bigwedge$ *ck s1 s2 env top0 tops err.*
evaluate-top ck env s1 top0 (s2, Rerr err)
==>

evaluate-prog ck env s1 (*top0* # *tops*) (*s2*, *Rerr err*)

— *val evaluate-whole-prog* : *forall 'ffi. Eq 'ffi => bool −> sem-env v −> state 'ffi −> prog −> state 'ffi ∗ result* (*sem-env v*) *v −> bool*
fun *evaluate-whole-prog* :: *bool* ⇒(*v*)*sem-env* ⇒ *'ffi state* ⇒(*top0*)*list* ⇒ *'ffi state*∗(((*v*)*sem-env*),(*v*))*result* ⇒ *bool* **where**
evaluate-whole-prog ck env s1 tops (*s2*, *res*) = (
if no-dup-mods tops(*defined-mods s1*) ∧ *no-dup-top-types tops*(*defined-types s1*) *then*
evaluate-prog ck env s1 tops (*s2*, *res*)
else
(*s1* = *s2*) ∧ (*res* = *Rerr* (*Rabort Rtype-error*)))

— *val dec-diverges* : *forall 'ffi. sem-env v −> state 'ffi −> dec −> bool*
fun *dec-diverges* :: (*v*)*sem-env* ⇒ *'ffi state* ⇒ *dec* ⇒ *bool* **where**
dec-diverges env st (*Dlet locs p e*) = (*Lem-list.allDistinct* (*pat-bindings p* []) ∧ *e-diverges env* ((*refs st*),(*ffi st*)) *e*)
| *dec-diverges env st* (*Dletrec locs funs*) = (*False*)
| *dec-diverges env st* (*Dtype locs tds*) = (*False*)
| *dec-diverges env st* (*Dtabbrev locs tvs tn t1*) = (*False*)
| *dec-diverges env st* (*Dexn locs cn ts*) = (*False*)

inductive
decs-diverges :: (*modN*)*list* ⇒(*v*)*sem-env* ⇒ *'ffi state* ⇒ *decs* ⇒ *bool* **where**

cons1 : ⋀ *mn st env d ds.*
dec-diverges env st d
==>
decs-diverges mn env st (*d* # *ds*)

|

cons2 : ⋀ *mn s1 s2 env d ds new-env.*
evaluate-dec False mn env s1 d (*s2*, *Rval new-env*) ∧
decs-diverges mn (*extend-dec-env new-env env*) *s2 ds*
==>
decs-diverges mn env s1 (*d* # *ds*)

inductive
top-diverges :: (*v*)*sem-env* ⇒ *'ffi state* ⇒ *top0* ⇒ *bool* **where**

tdec : ⋀ *st env d.*
dec-diverges env st d
==>
top-diverges env st (*Tdec d*)

|

tmod : $\bigwedge$ *env s1 ds mn specs.*
¬ ([*mn*] ∈(*defined-mods* *s1*)) ∧
(*no-dup-types ds* ∧
decs-diverges [*mn*] *env s1 ds*)
==>
top-diverges env s1 (*Tmod mn specs ds*)

inductive
prog-diverges :: (*v*)*sem-env* ⇒ ′*ffi state* ⇒ *prog* ⇒ *bool* **where**

cons1 : $\bigwedge$ *st env top0 tops.*
top-diverges env st top0
==>
prog-diverges env st (*top0* # *tops*)

|

cons2 : $\bigwedge$ *s1 s2 env top0 tops new-env.*
evaluate-top False env s1 top0 (*s2*, *Rval new-env*) ∧
prog-diverges (*extend-dec-env new-env env*) *s2 tops*
==>
prog-diverges env s1 (*top0* # *tops*)
end

Chapter 10

Generated by Lem from *semantics/alt-semantics/proofs/bigSmallInvariants.lem*

theory *BigSmallInvariants*

imports
Main
HOL−Library.Datatype-Records
LEM.Lem-pervasives
Lib
Namespace
Ast
SemanticPrimitives
SmallStep
BigStep

begin

— *open import Pervasives*
— *open import Lib*
— *open import Namespace*
— *open import Ast*
— *open import SemanticPrimitives*
— *open import SmallStep*
— *open import BigStep*

— −−−−−− *Auxiliary relations for proving big/small step equivalence* −−−−−−

inductive
evaluate-ctxt :: (v)*sem-env* ⇒ *'ffi state* ⇒ *ctxt-frame* ⇒ *v* ⇒ *'ffi state*∗$((v),(v))$*result* ⇒ *bool* **where**

raise : $\bigwedge$ *env s v1*.

evaluate-ctxt env s (*Craise* ()) *v1* (*s*, *Rerr* (*Rraise v1*))

|

handle : $\bigwedge$ *env s v1 pes*.

evaluate-ctxt env s (*Chandle* () *pes*) *v1* (*s*, *Rval v1*)

|

app1 : $\bigwedge$ *env e v1 vs1 vs2 es env′ bv s1 s2*.
evaluate-list False env s1 es (*s2*, *Rval vs2*) ∧
((*do-opapp* ((*List.rev vs2* @ [*v1*]) @ *vs1*) = *Some* (*env′*,*e*)) ∧
evaluate False env′ s2 e bv)
==>
evaluate-ctxt env s1 (*Capp Opapp vs1* () *es*) *v1 bv*

|

app2 : $\bigwedge$ *env v1 vs1 vs2 es s1 s2*.
evaluate-list False env s1 es (*s2*, *Rval vs2*) ∧
(*do-opapp* ((*List.rev vs2* @ [*v1*]) @ *vs1*) = *None*)
==>
evaluate-ctxt env s1 (*Capp Opapp vs1* () *es*) *v1* (*s2*, *Rerr* (*Rabort Rtype-error*))

|

app3 : $\bigwedge$ *env op0 v1 vs1 vs2 es res s1 s2 new-refs new-ffi*.
(*op0* ≠ *Opapp*) ∧
(*evaluate-list False env s1 es* (*s2*, *Rval vs2*) ∧
(*do-app* ((*refs s2*),(*ffi s2*)) *op0* ((*List.rev vs2* @ [*v1*]) @ *vs1*) = *Some* ((*new-refs*, *new-ffi*) ,*res*)))
==>
evaluate-ctxt env s1 (*Capp op0 vs1* () *es*) *v1* ((*s2* (| *ffi* := *new-ffi*, *refs* := *new-refs* |)), *res*)

|

app4 : $\bigwedge$ *env op0 v1 vs1 vs2 es s1 s2*.
(*op0* ≠ *Opapp*) ∧
(*evaluate-list False env s1 es* (*s2*, *Rval vs2*) ∧
(*do-app* ((*refs s2*),(*ffi s2*)) *op0* ((*List.rev vs2* @ [*v1*]) @ *vs1*) = *None*))
==>
evaluate-ctxt env s1 (*Capp op0 vs1* () *es*) *v1* (*s2*, *Rerr* (*Rabort Rtype-error*))

|

app5 : $\bigwedge$ *env op0 es vs v1 err s s'*.
evaluate-list False env s es (*s'*, *Rerr err*)
==>
evaluate-ctxt env s (*Capp op0 vs* () *es*) *v1* (*s'*, *Rerr err*)

|

log1 : $\bigwedge$ *env op0 e2 v1 e' bv s*.
(*do-log op0 v1 e2* = *Some* (*Exp e'*)) ∧
evaluate False env s e' bv
==>
evaluate-ctxt env s (*Clog op0* () *e2*) *v1 bv*

|

log2 : $\bigwedge$ *env op0 e2 v1 v' s*.
do-log op0 v1 e2 = *Some* (*Val v'*)
==>
evaluate-ctxt env s (*Clog op0* () *e2*) *v1* (*s*, *Rval v'*)

|

log3 : $\bigwedge$ *env op0 e2 v1 s*.
(*do-log op0 v1 e2* = *None*)
==>
evaluate-ctxt env s (*Clog op0* () *e2*) *v1* (*s*, *Rerr* (*Rabort Rtype-error*))

|
if1 : $\bigwedge$ *env e2 e3 v1 e' bv s*.
(*do-if v1 e2 e3* = *Some e'*) ∧
evaluate False env s e' bv
==>
evaluate-ctxt env s (*Cif* () *e2 e3*) *v1 bv*

|

if2 : $\bigwedge$ *env e2 e3 v1 s*.
do-if v1 e2 e3 = *None*
==>
evaluate-ctxt env s (*Cif* () *e2 e3*) *v1* (*s*, *Rerr* (*Rabort Rtype-error*))

|

mat : $\bigwedge$ *env pes v1 bv s err-v*.
evaluate-match False env s v1 pes err-v bv
==>
evaluate-ctxt env s (*Cmat* () *pes err-v*) *v1 bv*

|

lt : $\bigwedge$ $env\ n\ e2\ v1\ bv\ s.$
$evaluate\ False\ (\ env\ (|\ v := (nsOptBind\ n\ v1(v\quad env))\ |))\ s\ e2\ bv$
==>
$evaluate\text{-}ctxt\ env\ s\ (Clet\ n\ ()\quad e2)\ v1\ bv$

|

$con1$: $\bigwedge$ $env\ cn\ es\ vs\ v1\ vs'\ s1\ s2\ v'.$
$do\text{-}con\text{-}check(c\quad env)\ cn\ ((List.length\ vs + List.length\ es) +(\ 1 :: nat)) \wedge$
$((build\text{-}conv(c\quad env)\ cn\ ((List.rev\ vs'\ @\ [v1])\ @\ vs) = Some\ v') \wedge$
$evaluate\text{-}list\ False\ env\ s1\ es\ (s2,\ Rval\ vs'))$
==>
$evaluate\text{-}ctxt\ env\ s1\ (Ccon\ cn\ vs\ ()\quad es)\ v1\ (s2,\ Rval\ v')$

|

$con2$: $\bigwedge$ $env\ cn\ es\ vs\ v1\ s.$
$\neg\ (do\text{-}con\text{-}check(c\quad env)\ cn\ ((List.length\ vs + List.length\ es) +(\ 1 :: nat)))$
==>
$evaluate\text{-}ctxt\ env\ s\ (Ccon\ cn\ vs\ ()\quad es)\ v1\ (s,\ Rerr\ (Rabort\ Rtype\text{-}error))$

|

$con3$: $\bigwedge$ $env\ cn\ es\ vs\ v1\ err\ s\ s'.$
$do\text{-}con\text{-}check(c\quad env)\ cn\ ((List.length\ vs + List.length\ es) +(\ 1 :: nat)) \wedge$
$evaluate\text{-}list\ False\ env\ s\ es\ (s',\ Rerr\ err)$
==>
$evaluate\text{-}ctxt\ env\ s\ (Ccon\ cn\ vs\ ()\quad es)\ v1\ (s',\ Rerr\ err)$

|

$tannot$: $\bigwedge$ $env\ v1\ s\ t0.$

$evaluate\text{-}ctxt\ env\ s\ (Ctannot\ ()\quad t0)\ v1\ (s,\ Rval\ v1)$

|

$lannot$: $\bigwedge$ $env\ v1\ s\ l.$

$evaluate\text{-}ctxt\ env\ s\ (Clannot\ ()\quad l)\ v1\ (s,\ Rval\ v1)$

inductive
$evaluate\text{-}ctxts\ ::\ 'ffi\ state \Rightarrow (ctxt)list \Rightarrow ((v),(v))result \Rightarrow\ 'ffi\ state*((v),(v))result$
$\Rightarrow bool$ **where**

$empty$: $\bigwedge$ $res\ s.$

$evaluate\text{-}ctxts\ s\ []\ res\ (s,\ res)$

|

cons-val : $\bigwedge$ *c1 cs env v1 res bv s1 s2*.
evaluate-ctxt env s1 c1 v1 (*s2*, *res*) ∧
evaluate-ctxts s2 cs res bv
==>
evaluate-ctxts s1 ((*c1*,*env*)# *cs*) (*Rval v1*) *bv*

|

cons-err : $\bigwedge$ *c1 cs env err s bv*.
evaluate-ctxts s cs (*Rerr err*) *bv* ∧
(((∀ *pes*. *c1* ≠ *Chandle* () *pes*)) ∨
((∀ *v1*. *err* ≠ *Rraise v1*)))
==>
evaluate-ctxts s ((*c1*,*env*)# *cs*) (*Rerr err*) *bv*

|

cons-handle : $\bigwedge$ *cs env s s′ res1 res2 pes v1*.
evaluate-match False env s v1 pes v1 (*s′*, *res1*) ∧
evaluate-ctxts s′ cs res1 res2
==>
evaluate-ctxts s ((*Chandle* () *pes*,*env*)# *cs*) (*Rerr* (*Rraise v1*)) *res2*

inductive
evaluate-state :: *′ffi small-state* ⇒ *′ffi state*∗((*v*),(*v*))*result* ⇒ *bool* **where**

exp : $\bigwedge$ *env e c1 res bv ffi0 refs0 st*.
evaluate False env (| *clock* =((*0* :: *nat*)), *refs* = *refs0*, *ffi* = *ffi0*, *defined-types* = ({}), *defined-mods* =
({}) |) *e* (*st*, *res*) ∧
evaluate-ctxts st c1 res bv
==>
evaluate-state (*env*, (*refs0*, *ffi0*), *Exp e*, *c1*) *bv*

|

vl : $\bigwedge$ *env ffi0 refs0 v1 c1 bv*.
evaluate-ctxts (| *clock* =((*0* :: *nat*)), *refs* = *refs0*, *ffi* = *ffi0*, *defined-types* = ({}), *defined-mods* =
({}) |) *c1* (*Rval v1*) *bv*
==>
evaluate-state (*env*, (*refs0*, *ffi0*), *Val v1*, *c1*) *bv*
end

Chapter 11

Generated by Lem from *semantics/evaluate.lem*.

theory *Evaluate*

imports
Main
HOL−Library.Datatype-Records
LEM.Lem-pervasives-extra
Lib
Namespace
Ast
SemanticPrimitives
Ffi

begin

— *open import Pervasives-extra*
— *open import Lib*
— *open import Ast*
— *open import Namespace*
— *open import SemanticPrimitives*
— *open import Ffi*

— *The semantics is defined here using fix-clock so that HOL4 generates ∗ provable termination conditions. However, after termination is proved, we ∗ clean up the definition (in HOL4) to remove occurrences of fix-clock.*

fun *fix-clock* :: $'a\ state \Rightarrow 'b\ state*'c \Rightarrow 'b\ state*'c$ **where**
fix-clock s $(s',res) = ($
$((s'\ (|\ clock := (if(clock\ \ s') \leq(clock\ \ s)$
$then(clock\ \ s')\ else(clock\ \ s))\ |)),res))$

definition *dec-clock* :: $'a\ state \Rightarrow 'a\ state$ **where**

dec-clock s = ((*s* (| *clock* := ((*clock* *s*) −(*1* :: *nat*)) |)))

— *list-result is equivalent to map-result* (*v*. [*v*]) *I*, *where map-result is* ∗ *defined in evalPropsTheory*
fun
list-result :: (′*a*,′*b*)*result* ⇒((′*a list*),′*b*)*result* **where**

list-result (*Rval v2*) = (*Rval* [*v2*])
|
list-result (*Rerr e*) = (*Rerr e*)

— *val evaluate* : *forall* ′*ffi*. *state* ′*ffi* −> *sem-env v* −> *list exp* −> *state* ′*ffi* ∗ *result* (*list v*) *v*
— *val evaluate-match* : *forall* ′*ffi*. *state* ′*ffi* −> *sem-env v* −> *v* −> *list* (*pat* ∗ *exp*) −> *v* −> *state* ′*ffi* ∗ *result* (*list v*) *v*
function (*sequential*,*domintros*)
fun-evaluate-match :: ′*ffi state* ⇒(*v*)*sem-env* ⇒ *v* ⇒(*pat*∗*exp0*)*list* ⇒ *v* ⇒ ′*ffi state*∗(((*v*)*list*),(*v*))*result*
and
fun-evaluate :: ′*ffi state* ⇒(*v*)*sem-env* ⇒(*exp0*)*list* ⇒ ′*ffi state*∗(((*v*)*list*),(*v*))*result*
where

fun-evaluate st env [] = ((*st*, *Rval* []))
|
fun-evaluate st env (*e1* # *e2* # *es*) = (
  (*case fix-clock st* (*fun-evaluate st env* [*e1*]) *of*
    (*st*′, *Rval v1*) =>
      (*case fun-evaluate st*′ *env* (*e2* # *es*) *of*
        (*st*′′, *Rval vs*) => (*st*′′, *Rval* (*List.hd v1* # *vs*))
      | *res* => *res*
)
  | *res* => *res*
  ))
|
fun-evaluate st env [*Lit l*] = ((*st*, *Rval* [*Litv l*]))
|
fun-evaluate st env [*Raise e*] = (
  (*case fun-evaluate st env* [*e*] *of*
    (*st*′, *Rval v2*) => (*st*′, *Rerr* (*Rraise* (*List.hd v2*)))
  | *res* => *res*
  ))
|
fun-evaluate st env [*Handle e pes*] = (
  (*case fix-clock st* (*fun-evaluate st env* [*e*]) *of*
    (*st*′, *Rerr* (*Rraise v2*)) => *fun-evaluate-match st*′ *env v2 pes v2*
  | *res* => *res*
  ))

```
|
fun-evaluate st env [Con cn es] = (
  if do-con-check(c   env) cn (List.length es) then
    (case  fun-evaluate st env (List.rev es) of
      (st', Rval vs) =>
        (case  build-conv(c   env) cn (List.rev vs) of
          Some v2 => (st', Rval [v2])
        | None => (st', Rerr (Rabort Rtype-error))
        )
    | res => res
    )
  else (st, Rerr (Rabort Rtype-error)))
|
fun-evaluate st env [Var n] = (
  (case  nsLookup(v   env) n of
    Some v2 => (st, Rval [v2])
  | None => (st, Rerr (Rabort Rtype-error))
  ))
|
fun-evaluate st env [Fun x e] = ( (st, Rval [Closure env x e]))
|
fun-evaluate st env [App op1 es] = (
  (case  fix-clock st (fun-evaluate st env (List.rev es)) of
    (st', Rval vs) =>
      if op1 = Opapp then
        (case  do-opapp (List.rev vs) of
          Some (env',e) =>
            if(clock   st') =( 0 :: nat) then
              (st', Rerr (Rabort Rtimeout-error))
            else
              fun-evaluate (dec-clock st') env' [e]
        | None => (st', Rerr (Rabort Rtype-error))
        )
      else
        (case  do-app ((refs   st'),(ffi   st')) op1 (List.rev vs) of
          Some ((refs1,ffi1),r) => (( st' (| refs := refs1, ffi := ffi1 |)), list-result r)
        | None => (st', Rerr (Rabort Rtype-error))
        )
  | res => res
  ))
|
fun-evaluate st env [Log lop e1 e2] = (
  (case  fix-clock st (fun-evaluate st env [e1]) of
    (st', Rval v1) =>
      (case  do-log lop (List.hd v1) e2 of
        Some (Exp e) => fun-evaluate st' env [e]
      | Some (Val v2) => (st', Rval [v2])
      | None => (st', Rerr (Rabort Rtype-error))
      )
```

```
  | res => res
  ))
|
fun-evaluate st env [If e1 e2 e3] = (
  (case  fix-clock st (fun-evaluate st env [e1]) of
    (st', Rval v2) =>
      (case  do-if (List.hd v2) e2 e3 of
        Some e => fun-evaluate st' env [e]
      | None => (st', Rerr (Rabort Rtype-error))
      )
  | res => res
  ))
|
fun-evaluate st env [Mat e pes] = (
  (case  fix-clock st (fun-evaluate st env [e]) of
    (st', Rval v2) =>
      fun-evaluate-match st' env (List.hd v2) pes Bindv
  | res => res
  ))
|
fun-evaluate st env [Let xo e1 e2] = (
  (case  fix-clock st (fun-evaluate st env [e1]) of
    (st', Rval v2) => fun-evaluate st' ( env (| v := (nsOptBind xo (List.hd v2)(v
env)) |)) [e2]
  | res => res
  ))
|
fun-evaluate st env [Letrec funs e] = (
  if allDistinct (List.map ( λx .
  (case  x of (x,y,z) => x )) funs) then
    fun-evaluate st ( env (| v := (build-rec-env funs env(v   env)) |)) [e]
  else
    (st, Rerr (Rabort Rtype-error)))
|
fun-evaluate st env [Tannot e t1] = (
  fun-evaluate st env [e])
|
fun-evaluate st env [Lannot e l] = (
  fun-evaluate st env [e])
|
fun-evaluate-match st env v2 [] err-v = ( (st, Rerr (Rraise err-v)))
|
fun-evaluate-match st env v2 ((p,e)# pes) err-v  = (
  if allDistinct (pat-bindings p []) then
    (case  pmatch(c   env)(refs   st) p v2 [] of
     Match env-v' => fun-evaluate st ( env (| v := (nsAppend (alist-to-ns env-v')(v
env)) |)) [e]
    | No-match => fun-evaluate-match st env v2 pes err-v
    | Match-type-error => (st, Rerr (Rabort Rtype-error))
```

```
  )
  else (st, Rerr (Rabort Rtype-error)))
by pat-completeness auto


— val evaluate-decs : forall 'ffi. list modN −> state 'ffi −> sem-env v −> list dec
−> state 'ffi ∗ result (sem-env v) v
fun
fun-evaluate-decs :: (string)list ⇒ 'ffi state ⇒(v)sem-env ⇒(dec)list ⇒ 'ffi state∗(((v)sem-env),(v))result
where

fun-evaluate-decs mn st env [] = ( (st, Rval (| v = nsEmpty, c = nsEmpty |)))
|
fun-evaluate-decs mn st env (d1 # d2 # ds) = (
  (case fun-evaluate-decs mn st env [d1] of
    (st1, Rval env1) =>
    (case fun-evaluate-decs mn st1 (extend-dec-env env1 env) (d2 # ds) of
      (st2,r) => (st2, combine-dec-result env1 r)
    )
  | res => res
  ))
|
fun-evaluate-decs mn st env [Dlet locs p e] = (
  if allDistinct (pat-bindings p []) then
    (case fun-evaluate st env [e] of
      (st', Rval v2) =>
        (st',
         (case pmatch(c env)(refs st') p (List.hd v2) [] of
           Match new-vals => Rval (| v = (alist-to-ns new-vals), c = nsEmpty |)
         | No-match => Rerr (Rraise Bindv)
         | Match-type-error => Rerr (Rabort Rtype-error)
         ))
    | (st', Rerr err) => (st', Rerr err)
    )
  else
    (st, Rerr (Rabort Rtype-error)))
|
fun-evaluate-decs mn st env [Dletrec locs funs] = (
  (st,
   (if allDistinct (List.map ( λx .
  (case x of (x,y,z) => x )) funs) then
     Rval (| v = (build-rec-env funs env nsEmpty), c = nsEmpty |)
   else
     Rerr (Rabort Rtype-error))))
|
fun-evaluate-decs mn st env [Dtype locs tds] = (
  (let new-tdecs = (type-defs-to-new-tdecs mn tds) in
    if check-dup-ctors tds ∧
       (disjnt new-tdecs(defined-types st) ∧
```

```
      allDistinct (List.map ( λx .
  (case  x of (tvs,tn,ctors) => tn )) tds))
    then
      (( st (| defined-types := (new-tdecs ∪(defined-types   st)) |)),
       Rval (| v = nsEmpty, c = (build-tdefs mn tds) |))
    else
      (st, Rerr (Rabort Rtype-error))))
|
fun-evaluate-decs mn st env [Dtabbrev locs tvs tn t1] = (
  (st, Rval (| v = nsEmpty, c = nsEmpty |)))
|
fun-evaluate-decs mn st env [Dexn locs cn ts] = (
  if TypeExn (mk-id mn cn) ∈(defined-types   st) then
    (st, Rerr (Rabort Rtype-error))
  else
    (( st (| defined-types := ({TypeExn (mk-id mn cn)} ∪(defined-types   st)) |)),
     Rval (| v = nsEmpty, c = (nsSing cn (List.length ts, TypeExn (mk-id mn cn)))
|)))
```

definition *envLift* :: *string ⇒ 'a sem-env ⇒ 'a sem-env* **where**

```
      envLift mn env = (
  (| v = (nsLift mn(v   env)), c = (nsLift mn(c   env)) |) )
```

— *val evaluate-tops : forall 'ffi. state 'ffi −> sem-env v −> list top −> state 'ffi ∗ result (sem-env v) v*

fun
evaluate-tops :: *'ffi state ⇒(v)sem-env ⇒(top0)list ⇒ 'ffi state∗(((v)sem-env),(v))result*
where

```
evaluate-tops st env [] = ( (st, Rval (| v = nsEmpty, c = nsEmpty |)))
|
evaluate-tops st env (top1 # top2 # tops) = (
  (case  evaluate-tops st env [top1] of
    (st1, Rval env1) =>
      (case  evaluate-tops st1 (extend-dec-env env1 env) (top2 # tops) of
        (st2, r) => (st2, combine-dec-result env1 r)
      )
  | res => res
  ))
|
evaluate-tops st env [Tdec d] = ( fun-evaluate-decs [] st env [d])
|
evaluate-tops st env [Tmod mn specs ds] = (
  if ¬ ([mn] ∈(defined-mods   st)) ∧ no-dup-types ds
  then
    (case  fun-evaluate-decs [mn] st env ds of
      (st', r) =>
```

((st′ (| defined-mods := ({[mn]} ∪(defined-mods st′)) |)),
(case r of
Rval env′ => Rval (| v = (nsLift mn(v env′)), c = (nsLift mn(c env′))
|)
| Rerr err => Rerr err
))
)
else
(st, Rerr (Rabort Rtype-error)))

— val evaluate-prog : forall ′ffi. state ′ffi −> sem-env v −> prog −> state ′ffi ∗
result (sem-env v) v
definition
fun-evaluate-prog :: ′ffi state ⇒(v)sem-env ⇒(top0)list ⇒ ′ffi state∗(((v)sem-env),(v))result
where

fun-evaluate-prog st env prog = (
if no-dup-mods prog(defined-mods st) ∧ no-dup-top-types prog(defined-types st)
then
evaluate-tops st env prog
else
(st, Rerr (Rabort Rtype-error)))

end

Chapter 12

Generated by Lem from *misc/lem-lib-stub/lib.lem*.

theory *LibAuxiliary*

imports
Main
HOL−Library.Datatype-Records
LEM.Lem-pervasives
LEM.Lem-list-extra
LEM.Lem-string
Coinductive.Coinductive-List
Lib

begin

— **
—
— *Termination Proofs*
—
— **

termination *lunion* **by** *lexicographic-order*

end

Chapter 13

Generated by Lem from *semantics/namespace.lem*.

theory *NamespaceAuxiliary*

imports
Main
HOL−Library.Datatype-Records
LEM.Lem-pervasives
LEM.Lem-set-extra
Namespace

begin

— **
—
— *Termination Proofs*
—
— **

termination *nsMap* **by** *lexicographic-order*

end

Chapter 14

Generated by Lem from *semantics/primTypes.lem*.

theory *PrimTypes*

imports
Main
HOL−Library.Datatype-Records
LEM.Lem-pervasives
Lib
Namespace
Ast
SemanticPrimitives
Ffi
Evaluate

begin

— *open import Pervasives*
— *open import Ast*
— *open import SemanticPrimitives*
— *open import Ffi*
— *open import Namespace*
— *open import Lib*
— *open import Evaluate*

— *val prim-types-program* : *prog*
definition *prim-types-program* :: (*top0*)*list* **where**
prim-types-program = (
[*Tdec* (*Dexn* ((| *row* = *0*, *col* = *0*, *offset* = *0* |), (| *row* = *0*, *col* = *0*, *offset* = *0* |)) (*''Bind''*) []),
Tdec (*Dexn* ((| *row* = *0*, *col* = *0*, *offset* = *0* |), (| *row* = *0*, *col* = *0*, *offset* = *0* |)) (*''Chr''*) []),
Tdec (*Dexn* ((| *row* = *0*, *col* = *0*, *offset* = *0* |), (| *row* = *0*, *col* = *0*, *offset* = *0* |)) (*''Div''*) []),

Tdec (Dexn ((| row = 0, col = 0, offset = 0 |), (| row = 0, col = 0, offset = 0 |)) (''Subscript'') []),
Tdec (Dtype ((| row = 0, col = 0, offset = 0 |), (| row = 0, col = 0, offset = 0 |)) [([], (''bool''), [((''false''), []), ((''true''), [])])]),
Tdec (Dtype ((| row = 0, col = 0, offset = 0 |), (| row = 0, col = 0, offset = 0 |)) [([[(CHR 0x27), (CHR ''a'')]], (''list''), [((''nil''), []), (('':: ''), [Tvar ([(CHR 0x27), (CHR ''a'')]), Tapp [Tvar ([(CHR 0x27), (CHR ''a'')])] (TC-name (Short (''list'')))])])]),
Tdec (Dtype ((| row = 0, col = 0, offset = 0 |), (| row = 0, col = 0, offset = 0 |)) [([[(CHR 0x27), (CHR ''a'')]], (''option''), [((''NONE''), []),((''SOME''), [Tvar ([(CHR 0x27), (CHR ''a'')])])])])])

— *val add-to-sem-env : forall 'ffi. Eq 'ffi => (state 'ffi ∗ sem-env v) −> prog −> maybe (state 'ffi ∗ sem-env v)*
fun *add-to-sem-env :: 'ffi state∗(v)sem-env ⇒(top0)list ⇒('ffi state∗(v)sem-env)option*
where

```
    add-to-sem-env (st, env) prog = (
  (case fun-evaluate-prog st env prog of
    (st', Rval env') => Some (st', extend-dec-env env' env)
  | - => None
  ))
```

— *val prim-sem-env : forall 'ffi. Eq 'ffi => ffi-state 'ffi −> maybe (state 'ffi ∗ sem-env v)*
definition *prim-sem-env :: 'ffi ffi-state ⇒('ffi state∗(v)sem-env)option* **where**

```
    prim-sem-env ffi1 = (
  add-to-sem-env
    ((| clock =(( 0 :: nat)), refs = ([]), ffi = ffi1, defined-types = ({}), defined-mods = ({}) |),
     (| v = nsEmpty, c = nsEmpty |))
       prim-types-program )
```

end

Chapter 15

Generated by Lem from *semantics/semanticPrimitives.lem*.

theory *SemanticPrimitivesAuxiliary*

imports
Main
HOL−Library.Datatype-Records
LEM.Lem-pervasives
LEM.Lem-list-extra
LEM.Lem-string
Lib
Namespace
Ast
Ffi
FpSem
LEM.Lem-string-extra
SemanticPrimitives

begin

— **
—
— *Termination Proofs*
—
— **

termination *pmatch* **by** *lexicographic-order*

termination *do-eq* **by** *lexicographic-order*

termination *v-to-list* **by** *lexicographic-order*

termination *v-to-char-list* **by** *lexicographic-order*

termination *vs-to-string* **by** *lexicographic-order*

end

Chapter 16

Generated by Lem from *semantics/ffi/simpleIO.lem*.

theory *SimpleIO*

imports
Main
HOL−Library.Datatype-Records
LEM.Lem-pervasives
LEM.Lem-pervasives-extra
Lib
Ffi

begin

— *open import Pervasives*
— *open import Pervasives-extra*
— *open import Lib*
— *open import Ffi*

datatype-record *simpleIO* =
input :: *8 word llist*
output0 :: *8 word llist*

— *val isEof : oracle-function simpleIO*
fun *isEof* :: *simpleIO* ⇒(*8 word*)*list* ⇒(*8 word*)*list* ⇒(*simpleIO*)*oracle-result*
where
isEof st conf ([]) = (*Oracle-fail*)
| *isEof st conf* (*x* # *xs*) = (*Oracle-return st* ((*if*(*input st*) = *LNil then of-nat* ((*1* :: *nat*)) *else of-nat* ((*0* :: *nat*)))# *xs*))

— *val getChar : oracle-function simpleIO*
fun *getChar* :: *simpleIO* ⇒(*8 word*)*list* ⇒(*8 word*)*list* ⇒(*simpleIO*)*oracle-result*

```
where
      getChar st conf ([]) = ( Oracle-fail )
| getChar st conf (x # xs) = (
      (case  lhd'(input   st) of
        Some y => Oracle-return (( st (| input := (Option.the (ltl'(input   st))) |)))
(y # xs)
      | - => Oracle-fail
      ))
```

— *val putChar : oracle-function simpleIO*

```
definition putChar :: simpleIO ⇒(8 word)list ⇒(8 word)list ⇒(simpleIO)oracle-result
where
      putChar st conf input1 = (
  (case  input1 of
    [] => Oracle-fail
  | x # - => Oracle-return (( st (| output0 := (LCons x(output0   st)) |))) input1
  ))
```

— *val exit : oracle-function simpleIO*

```
definition exit0 :: simpleIO ⇒(8 word)list ⇒(8 word)list ⇒(simpleIO)oracle-result
where
      exit0 st conf input1 = ( Oracle-diverge )
```

— *val simpleIO-oracle : oracle simpleIO*

```
definition simpleIO-oracle  ::  string ⇒ simpleIO ⇒(8 word)list ⇒(8 word)list
⇒(simpleIO)oracle-result   where
      simpleIO-oracle s st conf input1 = (
  if s = (''isEof'') then
    isEof st conf input1
  else if s = (''getChar'') then
    getChar st conf input1
  else if s = (''putChar'') then
    putChar st conf input1
  else if s = (''exit'') then
    exit0 st conf input1
  else
    Oracle-fail )

end
```

Chapter 17

Generated by Lem from *semantics/tokens.lem*.

theory *Tokens*

imports
Main
HOL−Library.Datatype-Records
LEM.Lem-pervasives-extra

begin

— *open import Pervasives-extra*
— *Tokens for Standard ML. NB, not all of them are used in CakeML*
datatype *token* =
WhitespaceT nat | *NewlineT* | *LexErrorT*
| *HashT* | *LparT* | *RparT* | *StarT* | *CommaT* | *ArrowT* | *DotsT* | *ColonT* | *SealT*
| *SemicolonT* | *EqualsT* | *DarrowT* | *LbrackT* | *RbrackT* | *UnderbarT* | *LbraceT*
| *BarT* | *RbraceT* | *AndT* | *AndalsoT* | *AsT* | *CaseT* | *DatatypeT*
| *ElseT* | *EndT* | *EqtypeT* | *ExceptionT* | *FnT* | *FunT* | *HandleT* | *IfT*
| *InT* | *IncludeT* | *LetT* | *LocalT* | *OfT* | *OpT*
| *OpenT* | *OrelseT* | *RaiseT* | *RecT* | *RefT* | *SharingT* | *SigT* | *SignatureT* | *StructT*
| *StructureT* | *ThenT* | *TypeT* | *ValT* | *WhereT* | *WhileT* | *WithT* | *WithtypeT*
| *IntT int*
| *HexintT string*
| *WordT nat*
| *RealT string*
| *StringT string*
| *CharT char*
| *TyvarT string*
| *AlphaT string*
| *SymbolT string*
| *LongidT string string*
| *FFIT string*

end

Chapter 18

Generated by Lem from *semantics/typeSystem.lem*.

theory *TypeSystem*

imports
Main
HOL−Library.Datatype-Records
LEM.Lem-pervasives-extra
Lib
Namespace
Ast
SemanticPrimitives

begin

— *open import Pervasives-extra*
— *open import Lib*
— *open import Ast*
— *open import Namespace*
— *open import SemanticPrimitives*

— *Check that the free type variables are in the given list. Every deBruijn ∗ variable must be smaller than the first argument. So if it is 0, no deBruijn ∗ indices are permitted.*
— *val check-freevars : nat −> list tvarN −> t −> bool*
function (*sequential*,*domintros*)
check-freevars :: *nat* ⇒(*string*)*list* ⇒ *t* ⇒ *bool* **where**

check-freevars dbmax tvs (*Tvar tv*) = (
Set.member tv (*set tvs*))
|
check-freevars dbmax tvs (*Tapp ts tn*) = (
((∀ *x* ∈ (*set ts*). (*check-freevars dbmax tvs*) *x*)))
|

check-freevars dbmax tvs (*Tvar-db n*) = (*n* < *dbmax*)
by *pat-completeness auto*

— *Simultaneous substitution of types for type variables in a type*
— *val type-subst* : *Map.map tvarN t* −> *t* −> *t*
function (*sequential*,*domintros*)
type-subst :: ((*string*),(*t*))*Map.map* ⇒ *t* ⇒ *t* **where**

type-subst s (*Tvar tv*) = (
  (*case s tv of*
      None => *Tvar tv*
    | *Some*(*t1*) => *t1*
  ))
|
type-subst s (*Tapp ts tn*) = (
  Tapp (*List.map* (*type-subst s*) *ts*) *tn*)
|
type-subst s (*Tvar-db n*) = (*Tvar-db n*)
by *pat-completeness auto*

— *Increment the deBruijn indices in a type by n levels*, *skipping all levels* ∗ *less than skip*.
— *val deBruijn-inc* : *nat* −> *nat* −> *t* −> *t*
function (*sequential*,*domintros*)
deBruijn-inc :: *nat* ⇒ *nat* ⇒ *t* ⇒ *t* **where**

deBruijn-inc skip n (*Tvar tv*) = (*Tvar tv*)
|
deBruijn-inc skip n (*Tvar-db m*) = (
  if m < *skip then*
    Tvar-db m
  else
    Tvar-db (*m* + *n*))
|
deBruijn-inc skip n (*Tapp ts tn*) = (*Tapp* (*List.map* (*deBruijn-inc skip n*) *ts*) *tn*)
by *pat-completeness auto*

— *skip the lowest given indices and replace the next* (*LENGTH ts*) *with the given types and reduce all the higher ones*
— *val deBruijn-subst* : *nat* −> *list t* −> *t* −> *t*
function (*sequential*,*domintros*)
deBruijn-subst :: *nat* ⇒(*t*)*list* ⇒ *t* ⇒ *t* **where**

deBruijn-subst skip ts (*Tvar tv*) = (*Tvar tv*)
|

```
deBruijn-subst skip ts (Tvar-db n) = (
  if ¬ (n < skip) ∧ (n < (List.length ts + skip)) then
    List.nth ts (n − skip)
  else if ¬ (n < skip) then
    Tvar-db (n − List.length ts)
  else
    Tvar-db n )
|
deBruijn-subst skip ts (Tapp ts′ tn) = (
  Tapp (List.map (deBruijn-subst skip ts) ts′) tn )
by pat-completeness auto


— Type environments
datatype tenv-val-exp =
    Empty
  — Binds several de Bruijn type variables
  | Bind-tvar  nat   tenv-val-exp
  — The number is how many de Bruijn type variables the typescheme binds
  | Bind-name  varN   nat   t   tenv-val-exp

— val bind-tvar : nat −> tenv-val-exp −> tenv-val-exp
definition bind-tvar :: nat ⇒ tenv-val-exp ⇒ tenv-val-exp  where
     bind-tvar tvs tenvE = ( if tvs =( 0 :: nat) then tenvE else Bind-tvar tvs tenvE
)


— val opt-bind-name : maybe varN −> nat −> t −> tenv-val-exp −> tenv-val-exp
fun opt-bind-name  :: (string)option ⇒ nat ⇒ t ⇒ tenv-val-exp ⇒ tenv-val-exp
where
     opt-bind-name None tvs t1 tenvE = ( tenvE )
| opt-bind-name (Some n′) tvs t1 tenvE = ( Bind-name n′ tvs t1 tenvE )


— val tveLookup : varN −> nat −> tenv-val-exp −> maybe (nat ∗ t)
fun
tveLookup :: string ⇒ nat ⇒ tenv-val-exp ⇒(nat∗t)option  where

tveLookup n inc Empty = ( None )
|
tveLookup n inc (Bind-tvar tvs tenvE) = ( tveLookup n (inc + tvs) tenvE )
|
tveLookup n inc (Bind-name n′ tvs t1 tenvE) = (
  if n′ = n then
    Some (tvs, deBruijn-inc tvs inc t1)
  else
    tveLookup n inc tenvE )
```

type-synonym *tenv-abbrev* = (*modN*, *typeN*, (*tvarN list* ∗ *t*)) *namespace*
type-synonym *tenv-ctor* = (*modN*, *conN*, (*tvarN list* ∗ *t list* ∗ *tid-or-exn*)) *namespace*
type-synonym *tenv-val* = (*modN*, *varN*, (*nat* ∗ *t*)) *namespace*

datatype-record *type-env* =

v0 :: *tenv-val*

c0 :: *tenv-ctor*

t :: *tenv-abbrev*

— *val extend-dec-tenv* : *type-env* −> *type-env* −> *type-env*
definition *extend-dec-tenv* :: *type-env* ⇒ *type-env* ⇒ *type-env* **where**
extend-dec-tenv tenv′ tenv = (
(| *v0* = (*nsAppend*(*v0* *tenv′*)(*v0* *tenv*)),
c0 = (*nsAppend*(*c0* *tenv′*)(*c0* *tenv*)),
t = (*nsAppend*(*t* *tenv′*)(*t* *tenv*)) |))

— *val lookup-varE* : *id modN varN* −> *tenv-val-exp* −> *maybe* (*nat* ∗ *t*)
fun *lookup-varE* :: ((*string*),(*string*))*id0* ⇒ *tenv-val-exp* ⇒(*nat*∗*t*)*option* **where**

lookup-varE (*Short x*) *tenvE* = (*tveLookup x*((*0* :: *nat*)) *tenvE*)
| *lookup-varE* - *tenvE* = (*None*)

— *val lookup-var* : *id modN varN* −> *tenv-val-exp* −> *type-env* −> *maybe* (*nat* ∗ *t*)
definition *lookup-var* :: ((*modN*),(*varN*))*id0* ⇒ *tenv-val-exp* ⇒ *type-env* ⇒(*nat*∗*t*)*option* **where**
lookup-var id1 tenvE tenv = (
(*case lookup-varE id1 tenvE of*
Some x => *Some x*
| *None* => *nsLookup*(*v0* *tenv*) *id1*
))

— *val num-tvs* : *tenv-val-exp* −> *nat*
fun
num-tvs :: *tenv-val-exp* ⇒ *nat* **where**

num-tvs Empty = ((*0* :: *nat*))
|
num-tvs (*Bind-tvar tvs tenvE*) = (*tvs* + *num-tvs tenvE*)
|

num-tvs (*Bind-name n tvs t1 tenvE*) = (*num-tvs tenvE*)

— *val bind-var-list* : *nat* −> *list* (*varN* ∗ *t*) −> *tenv-val-exp* −> *tenv-val-exp*
fun
bind-var-list :: *nat* ⇒(*string*∗*t*)*list* ⇒ *tenv-val-exp* ⇒ *tenv-val-exp* **where**

bind-var-list tvs [] *tenvE* = (*tenvE*)
|
bind-var-list tvs ((*n*,*t1*)# *binds*) *tenvE* = (
Bind-name n tvs t1 (*bind-var-list tvs binds tenvE*))

— *A pattern matches values of a certain type and extends the type environment* ∗ *with the pattern's binders. The number is the maximum deBruijn type variable* ∗ *allowed.*
— *val type-p* : *nat* −> *type-env* −> *pat* −> *t* −> *list* (*varN* ∗ *t*) −> *bool*

— *An expression has a type*
— *val type-e* : *type-env* −> *tenv-val-exp* −> *exp* −> *t* −> *bool*

— *A list of expressions has a list of types*
— *val type-es* : *type-env* −> *tenv-val-exp* −> *list exp* −> *list t* −> *bool*

— *Type a mutually recursive bundle of functions. Unlike pattern typing*, *the* ∗ *resulting environment does not extend the input environment*, *but just* ∗ *represents the functions*
— *val type-funs* : *type-env* −> *tenv-val-exp* −> *list* (*varN* ∗ *varN* ∗ *exp*) −> *list* (*varN* ∗ *t*) −> *bool*

datatype-record *decls* =

defined-mods0 :: (*modN list*) *set*

defined-types0 :: ((*modN*, *typeN*)*id0*) *set*

defined-exns :: ((*modN*, *conN*)*id0*) *set*

— *val empty-decls* : *decls*
definition *empty-decls* :: *decls* **where**
empty-decls = ((| *defined-mods0* = ({}), *defined-types0* = ({}), *defined-exns* = ({})|))

— *val union-decls* : *decls* −> *decls* −> *decls*
definition *union-decls* :: *decls* ⇒ *decls* ⇒ *decls* **where**
union-decls d1 d2 = (
(| *defined-mods0* = ((*defined-mods0* *d1*) ∪(*defined-mods0* *d2*)),

defined-types0 = ((*defined-types0 d1*) ∪(*defined-types0 d2*)),
defined-exns = ((*defined-exns d1*) ∪(*defined-exns d2*)) |))

— *Check a declaration and update the top−level environments ∗ The arguments are in order*: ∗ − *the module that the declaration is in* ∗ − *the set of all modules*, *and types*, *and exceptions that have been previously declared* ∗ − *the type environment* ∗ − *the declaration* ∗ − *the set of all modules*, *and types*, *and exceptions that are declared here* ∗ − *the environment of new stuff declared here*

— *val type-d* : *bool* −> *list modN* −> *decls* −> *type-env* −> *dec* −> *decls* −> *type-env* −> *bool*

— *val type-ds* : *bool* −> *list modN* −> *decls* −> *type-env* −> *list dec* −> *decls* −> *type-env* −> *bool*
— *val check-signature* : *list modN* −> *tenv-abbrev* −> *decls* −> *type-env* −> *maybe specs* −> *decls* −> *type-env* −> *bool*
— *val type-specs* : *list modN* −> *tenv-abbrev* −> *specs* −> *decls* −> *type-env* −> *bool*
— *val type-prog* : *bool* −> *decls* −> *type-env* −> *list top* −> *decls* −> *type-env* −> *bool*

— *Check that the operator can have type* (*t1* −> ... −> *tn* −> *t*)
— *val type-op* : *op* −> *list t* −> *t* −> *bool*

```
fun type-op :: op0 ⇒(t)list ⇒ t ⇒ bool  where
    type-op (Opn o0) ts t1 = (
  (case (o0,ts) of
      ( -, [Tapp [] TC-int, Tapp [] TC-int]) => (t1 = Tint)
    | (-,-) => False
  ) )
| type-op (Opb o1) ts t1 = (
  (case (o1,ts) of
      ( -, [Tapp [] TC-int, Tapp [] TC-int]) => (t1 =
                                                Tapp []
                                                  (TC-name
                                                     (Short (''bool''))))
    | (-,-) => False
  ) )
| type-op (Opw w o2) ts t1 = (
  (case (w,o2,ts) of
      ( W8, -, [Tapp [] TC-word8, Tapp [] TC-word8]) => (t1 =
                                                         Tapp [] TC-word8)
    | ( W64, -, [Tapp [] TC-word64, Tapp [] TC-word64]) => (t1 =
                                                            Tapp []
                                                              TC-word64)
    | (-,-,-) => False
  ) )
| type-op (Shift w0 s n0) ts t1 = (
  (case (w0,s,n0,ts) of
```

```
    ( W8, -, -, [Tapp [] TC-word8]) => (t1 = Tapp [] TC-word8)
  | ( W64, -, -, [Tapp [] TC-word64]) => (t1 = Tapp [] TC-word64)
  | (-,-,-,-) => False
 ) )
| type-op Equality ts t1 = (
 (case  ts of
    [t11, t2] => (t11 = t2) ∧
                 (t1 = Tapp [] (TC-name (Short (''bool''))))
  | - => False
 ) )
| type-op (FP-cmp f) ts t1 = (
 (case (f,ts) of
    ( -, [Tapp [] TC-word64, Tapp [] TC-word64]) => (t1 =
                                                      Tapp []
                                                       (TC-name
                                                         (Short
                                                           (''bool''))))
  | (-,-) => False
 ) )
| type-op (FP-uop f0) ts t1 = (
 (case (f0,ts) of
    ( -, [Tapp [] TC-word64]) => (t1 = Tapp [] TC-word64)
  | (-,-) => False
 ) )
| type-op (FP-bop f1) ts t1 = (
 (case (f1,ts) of
    ( -, [Tapp [] TC-word64, Tapp [] TC-word64]) => (t1 = Tapp [] TC-word64)
  | (-,-) => False
 ) )
| type-op Opapp ts t1 = (
 (case  ts of
    [Tapp [t2', t3'] TC-fn, t2] => (t2 = t2') ∧ (t1 = t3')
  | - => False
 ) )
| type-op Opassign ts t1 = (
 (case  ts of
    [Tapp [t11] TC-ref, t2] => (t11 = t2) ∧ (t1 = Tapp [] TC-tup)
  | - => False
 ) )
| type-op Opref ts t1 = (
 (case  ts of [t11] => (t1 = Tapp [t11] TC-ref) | - => False ) )
| type-op Opderef ts t1 = (
 (case  ts of [Tapp [t11] TC-ref] => (t1 = t11) | - => False ) )
| type-op Aw8alloc ts t1 = (
 (case  ts of
    [Tapp [] TC-int, Tapp [] TC-word8] => (t1 = Tapp [] TC-word8array)
  | - => False
 ) )
| type-op Aw8sub ts t1 = (
```

```
  (case  ts of
      [Tapp [] TC-word8array, Tapp [] TC-int] => (t1 = Tapp [] TC-word8)
    | - => False
  ) )
| type-op Aw8length ts t1 = (
  (case  ts of [Tapp [] TC-word8array] => (t1 = Tapp [] TC-int) | - => False ) )
| type-op Aw8update ts t1 = (
  (case  ts of
      [Tapp [] TC-word8array, Tapp [] TC-int, Tapp [] TC-word8] => t1 =
                                                                  Tapp
                                                                  []
                                                                  TC-tup
    | - => False
  ) )
| type-op (WordFromInt w1) ts t1 = (
  (case (w1,ts) of
      ( W8, [Tapp [] TC-int]) => t1 = Tapp [] TC-word8
    | ( W64, [Tapp [] TC-int]) => t1 = Tapp [] TC-word64
    | (-,-) => False
  ) )
| type-op (WordToInt w2) ts t1 = (
  (case (w2,ts) of
      ( W8, [Tapp [] TC-word8]) => t1 = Tapp [] TC-int
    | ( W64, [Tapp [] TC-word64]) => t1 = Tapp [] TC-int
    | (-,-) => False
  ) )
| type-op CopyStrStr ts t1 = (
  (case  ts of
      [Tapp [] TC-string, Tapp [] TC-int, Tapp [] TC-int] => t1 =
                                                          Tapp []
                                                            TC-string
    | - => False
  ) )
| type-op CopyStrAw8 ts t1 = (
  (case  ts of
      [Tapp [] TC-string, Tapp [] TC-int, Tapp [] TC-int, Tapp [] TC-word8array,
Tapp [] TC-int] =>
  t1 = Tapp [] TC-tup
    | - => False
  ) )
| type-op CopyAw8Str ts t1 = (
  (case  ts of
      [Tapp [] TC-word8array, Tapp [] TC-int, Tapp [] TC-int] => t1 =
                                                                Tapp
                                                                []
                                                                 TC-string
    | - => False
  ) )
| type-op CopyAw8Aw8 ts t1 = (
```

```
(case ts of
   [Tapp [] TC-word8array, Tapp [] TC-int, Tapp [] TC-int, Tapp [] TC-word8array,
Tapp [] TC-int] =>
 t1 = Tapp [] TC-tup
   | - => False
 ) )
| type-op Ord ts t1 = (
 (case ts of [Tapp [] TC-char] => (t1 = Tint) | - => False ) )
| type-op Chr ts t1 = (
 (case ts of [Tapp [] TC-int] => (t1 = Tchar) | - => False ) )
| type-op (Chopb o3) ts t1 = (
 (case (o3,ts) of
    ( -, [Tapp [] TC-char, Tapp [] TC-char]) => (t1 =
                                               Tapp []
                                                (TC-name
                                                   (Short (''bool''))))
   | (-,-) => False
 ) )
| type-op Implode ts t1 = (
 (case ts of
    [] => False
   | t0 # l0 => (case t0 of
                 Tvar - => False
                | Tvar-db - => False
                | Tapp l1 t4 => (case l1 of
                                 [] => False
                                | t5 # l2 => (case t5 of
                                               Tvar - => False
                                              | Tvar-db - => False
                                              | Tapp l3 t7 => (case l3 of
                                                               [] =>
                                                               (case t7 of
                                                               TC-name - =>
                                                               False
                                                               | TC-int =>
                                                               False
                                                               | TC-char =>
                                                               (case l2 of
                                                               [] =>
                                                               (case t4 of
                                                               TC-name i0 =>
                                                               (case i0 of
                                                               Short s1 =>
                                                               if
                                                               (
                                                               s1 =
                                                               (''list'')) then
                                                               (
                                                               (case l0 of
```

[] =>
t1 =
Tapp
[]
TC-string
| - =>
False
)) *else*
False
| *Long* - - =>
False
)
| *TC-int* =>
False
| *TC-char* =>
False
| *TC-string* =>
False
| *TC-ref* =>
False
| *TC-word8* =>
False
| *TC-word64* =>
False
| *TC-word8array* =>
False
| *TC-fn* =>
False
| *TC-tup* =>
False
| *TC-exn* =>
False
| *TC-vector* =>
False
| *TC-array* =>
False
)
| - # - =>
False
)
| *TC-string* =>
False
| *TC-ref* =>
False
| *TC-word8* =>
False
| *TC-word64* =>
False
| *TC-word8array* =>

```
                                                  False
                                                  | TC-fn =>
                                                  False
                                                  | TC-tup =>
                                                  False
                                                  | TC-exn =>
                                                  False
                                                  | TC-vector =>
                                                  False
                                                  | TC-array =>
                                                  False
                                                  )
                                                  | - # - =>
                                                  False
                                                  )
                                         )
                                 )
                      )
  ) )
| type-op Strsub ts t1 = (
  (case  ts of
     [Tapp [] TC-string, Tapp [] TC-int] => t1 = Tchar
   | - => False
  ) )
| type-op Strlen ts t1 = (
  (case  ts of [Tapp [] TC-string] => t1 = Tint | - => False ) )
| type-op Strcat ts t1 = (
  (case  ts of
     [] => False
   | t10 # l6 => (case  t10 of
                    Tvar - => False
                  | Tvar-db - => False
                  | Tapp l7 t12 => (case  l7 of
                                      [] => False
                                    | t13 # l8 => (case  t13 of
                                                     Tvar - => False
                                                   | Tvar-db - =>
                                                  False
                                                   | Tapp l9 t15 =>
                                                  (case  l9 of
                                                     [] => (case  t15 of
                                                              TC-name - =>
                                                          False
                                                            | TC-int =>
                                                          False
                                                            | TC-char =>
                                                          False
                                                            | TC-string =>
                                                          (case  l8 of
```

```
    [] =>
(case  t12 of
    TC-name i3 =>
(case  i3 of
    Short s3 =>
if(s3 =
     (''list'')) then
  ((case  l6 of
     [] =>
   t1 =
     Tapp
     []
     TC-string
     | - =>
   False
   )) else
  False
  | Long - - =>
False
)
  | TC-int =>
False
  | TC-char =>
False
  | TC-string =>
False
  | TC-ref =>
False
  | TC-word8 =>
False
  | TC-word64 =>
False
  | TC-word8array =>
False
  | TC-fn =>
False
  | TC-tup =>
False
  | TC-exn =>
False
  | TC-vector =>
False
  | TC-array =>
False
)
  | - # - =>
False
)
  | TC-ref =>
```

```
False
  | TC-word8 =>
False
  | TC-word64 =>
False
  | TC-word8array =>
False
  | TC-fn =>
False
  | TC-tup =>
False
  | TC-exn =>
False
  | TC-vector =>
False
  | TC-array =>
False
)
| - # - => False
)
)
)
)
) )
| type-op VfromList ts t1 = (
  (case  ts of
     [] => False
   | t18 # l12 => (case  t18 of
        Tvar - => False
      | Tvar-db - => False
      | Tapp l13 t20 => (case  l13 of
           [] => False
         | t21 # l14 => (case  l14 of
              [] => (case  t20 of
                 TC-name i5 =>
                (case  i5 of
                 Short s5 =>
                if(
                 s5 =
                 (''list'')) then
                 (
                 (case
                 (t21,l12) of
                 (-,[]) =>
                 t1 =
                 Tapp
                 [t21]
                 TC-vector
                 | (-,-) =>
```

```
False
)) else
False
| Long _ _ =>
False
)
| TC_int =>
False
| TC_char =>
False
| TC_string =>
False
| TC_ref =>
False
| TC_word8 =>
False
| TC_word64 =>
False
| TC_word8array =>
False
| TC_fn =>
False
| TC_tup =>
False
| TC_exn =>
False
| TC_vector =>
False
| TC_array =>
False
)
| _ # _ =>
False
)
)
)
) )
| type_op Vsub ts t1 = (
(case ts of
    [Tapp [t11] TC_vector, Tapp [] TC_int] => t1 = t11
  | _ => False
) )
| type_op Vlength ts t1 = (
(case ts of [Tapp [t11] TC_vector] => (t1 = Tapp [] TC_int) | _ => False ) )
| type_op Aalloc ts t1 = (
(case ts of
    [Tapp [] TC_int, t11] => t1 = Tapp [t11] TC_array
  | _ => False
) )
```

```
| type-op AallocEmpty ts t1 = (
  (case  ts of
      [Tapp [] TC-tup] =>( ∃ t10. t1 = Tapp [t10] TC-array)
    | - => False
  ) )
| type-op Asub ts t1 = (
  (case  ts of
      [Tapp [t11] TC-array, Tapp [] TC-int] => t1 = t11
    | - => False
  ) )
| type-op Alength ts t1 = (
  (case  ts of [Tapp [t11] TC-array] => t1 = Tapp [] TC-int | - => False ) )
| type-op Aupdate ts t1 = (
  (case  ts of
      [Tapp [t11] TC-array, Tapp [] TC-int, t2] => (t11 = t2) ∧
                                              (t1 = Tapp [] TC-tup)
    | - => False
  ) )
| type-op ConfigGC ts t1 = (
  (case  ts of
      [Tapp [] TC-int, Tapp [] TC-int] => t1 = Tapp [] TC-tup
    | - => False
  ) )
| type-op (FFI s0) ts t1 = (
  (case (s0,ts) of
      (-, [Tapp [] TC-string, Tapp [] TC-word8array]) => t1 = Tapp [] TC-tup
    | (-,-) => False
  ) )
```

— *val check-type-names : tenv-abbrev −> t −> bool*
function (*sequential*,*domintros*)
check-type-names :: ((*string*),(*string*),((*string*)*list*∗*t*))*namespace* ⇒ *t* ⇒ *bool* **where**

```
check-type-names tenvT (Tvar tv) = (
  True )
|
check-type-names tenvT (Tapp ts tn) = (
  (case  tn of
     TC-name tn =>
       (case  nsLookup tenvT tn of
           Some (tvs, t1) => List.length tvs = List.length ts
         | None => False
       )
   | - => True
  ) ∧
  ((∀  x ∈ (set ts).  (check-type-names tenvT) x)))
|
```

```
check-type-names tenvT (Tvar-db n) = (
  True )
by pat-completeness auto


— Substitution of type names for the type they abbreviate
— val type-name-subst : tenv-abbrev −> t −> t
function (sequential,domintros)
type-name-subst :: ((string),(string),((string)list∗t))namespace ⇒ t ⇒ t   where

type-name-subst tenvT (Tvar tv) = ( Tvar tv )
|
type-name-subst tenvT (Tapp ts tc) = (
  (let args = (List.map (type-name-subst tenvT) ts) in
    (case  tc of
        TC-name tn =>
          (case  nsLookup tenvT tn of
               Some (tvs, t1) => type-subst (map-of (Lem-list-extra.zipSameLength
tvs args)) t1
            | None => Tapp args tc
          )
      | - => Tapp args tc
    )))
|
type-name-subst tenvT (Tvar-db n) = ( Tvar-db n )
by pat-completeness auto


— Check that a type definition defines no already defined types or duplicate ∗ con-
structors, and that the free type variables of each constructor argument ∗ type are
included in the type's type parameters. Also check that all of the ∗ types mentioned
are in scope.
— val check-ctor-tenv : tenv-abbrev −> list (list tvarN ∗ typeN ∗ list (conN ∗ list
t)) −> bool
definition check-ctor-tenv :: ((modN),(typeN),((tvarN)list∗t))namespace ⇒((tvarN)list∗typeN∗(conN∗(t)list
⇒ bool   where
      check-ctor-tenv tenvT tds = (
  check-dup-ctors tds ∧
  (((∀  x ∈ (set tds).  ( λx .
  (case  x of
      (tvs,tn,ctors) =>
  Lem-list.allDistinct tvs ∧
    ((∀  x ∈ (set ctors).
       ( λx .  (case  x of
                           (cn,ts) => ((∀  x ∈ (set ts).
                                         (check-freevars (( 0 :: nat)) tvs) x))
                                        ∧
                                        ((∀  x ∈ (set ts).
                                           (check-type-names tenvT) x))
```

```
                    )) x))
 )) x)) ∧
 Lem-list.allDistinct (List.map ( λx .
 (case  x of (-,tn,-) => tn )) tds)))
```

— val build-ctor-tenv : list modN −> tenv-abbrev −> list (list tvarN ∗ typeN ∗ list (conN ∗ list t)) −> tenv-ctor

definition *build-ctor-tenv* :: (string)list ⇒((modN),(typeN),((tvarN)list∗t))namespace ⇒((tvarN)list∗string∗(string∗(t)list)list)list ⇒((string),(string),((tvarN)list∗(t)list∗tid-or-exn))namespace
where

```
     build-ctor-tenv mn tenvT tds = (
 alist-to-ns
   (List.rev
     (List.concat
       (List.map
          ( λx .
 (case  x of
     (tvs,tn,ctors) =>
 List.map
   ( λx .  (case  x of
                    (cn,ts) => (cn,(tvs,List.map (type-name-subst tenvT)
                                        ts, TypeId (mk-id mn tn)))
                 )) ctors
 ))
          tds))))
```

— Check that an exception definition defines no already defined (or duplicate) ∗ constructors, and that the arguments have no free type variables.
— val check-exn-tenv : list modN −> conN −> list t −> bool

definition *check-exn-tenv* :: (modN)list ⇒ string ⇒(t)list ⇒ bool **where**

```
     check-exn-tenv mn cn ts = (
 ((∀  x ∈ (set ts).  (check-freevars(( 0 :: nat)) []) x)))
```

— For the value restriction on let−based polymorphism
— val is-value : exp −> bool

function (*sequential*,*domintros*)
is-value :: *exp0* ⇒ *bool* **where**

```
is-value (Lit -) = ( True )
|
is-value (Con - es) = ( ((∀  x ∈ (set es).  is-value x)))
|
is-value (Var -) = ( True )
|
is-value (Fun - -) = ( True )
|
```

is-value (*Tannot e* -) = (*is-value e*)
|
is-value (*Lannot e* -) = (*is-value e*)
|
is-value - = (*False*)
by *pat-completeness auto*

— *val tid-exn-to-tc* : *tid-or-exn* −> *tctor*
fun *tid-exn-to-tc* :: *tid-or-exn* ⇒ *tctor* **where**
tid-exn-to-tc (*TypeId tid*) = (*TC-name tid*)
| *tid-exn-to-tc* (*TypeExn* -) = (*TC-exn*)

inductive
type-ps :: *nat* ⇒ *type-env* ⇒(*pat*)*list* ⇒(*t*)*list* ⇒(*varN*∗*t*)*list* ⇒ *bool*
and
type-p :: *nat* ⇒ *type-env* ⇒ *pat* ⇒ *t* ⇒(*varN*∗*t*)*list* ⇒ *bool* **where**

pany : ⋀ *tvs tenv t0*.
check-freevars tvs [] *t0*
==>
type-p tvs tenv Pany t0 []

|

pvar : ⋀ *tvs tenv n t0*.
check-freevars tvs [] *t0*
==>
type-p tvs tenv (*Pvar n*) *t0* [(*n*,*t0*)]

|

plit-int : ⋀ *tvs tenv n*.

type-p tvs tenv (*Plit* (*IntLit n*)) *Tint* []

|

plit-char : ⋀ *tvs tenv c1*.

type-p tvs tenv (*Plit* (*Char c1*)) *Tchar* []

|

plit-string : ⋀ *tvs tenv s*.

type-p tvs tenv (*Plit* (*StrLit s*)) *Tstring* []

|

plit-word8 : $\bigwedge$ *tvs tenv w*.

type-p tvs tenv (*Plit* (*Word8 w*)) *Tword8* []

|

plit-word64 : $\bigwedge$ *tvs tenv w*.

type-p tvs tenv (*Plit* (*Word64 w*)) *Tword64* []

|

pcon-some : $\bigwedge$ *tvs tenv cn ps ts tvs' tn ts' bindings*.
(($\forall$ *x* $\in$ (*set ts'*). (*check-freevars tvs* []) *x*)) $\wedge$
((*List.length ts'* = *List.length tvs'*) $\wedge$
(*type-ps tvs tenv ps* (*List.map* (*type-subst* (*map-of* (*Lem-list-extra.zipSameLength*
tvs' ts'))) *ts*) *bindings* $\wedge$
(*nsLookup*(*c0* *tenv*) *cn* = *Some* (*tvs'*, *ts*, *tn*))))
==>
type-p tvs tenv (*Pcon* (*Some cn*) *ps*) (*Tapp ts'* (*tid-exn-to-tc tn*)) *bindings*

|

pcon-none : $\bigwedge$ *tvs tenv ps ts bindings*.
type-ps tvs tenv ps ts bindings
==>
type-p tvs tenv (*Pcon None ps*) (*Tapp ts TC-tup*) *bindings*

|

pref : $\bigwedge$ *tvs tenv p t0 bindings*.
type-p tvs tenv p t0 bindings
==>
type-p tvs tenv (*Pref p*) (*Tref t0*) *bindings*

|

ptypeannot : $\bigwedge$ *tvs tenv p t0 bindings*.
check-freevars((*0* :: *nat*)) [] *t0* $\wedge$
(*check-type-names*(*t* *tenv*) *t0* $\wedge$
type-p tvs tenv p (*type-name-subst*(*t* *tenv*) *t0*) *bindings*)
==>
type-p tvs tenv (*Ptannot p t0*) (*type-name-subst*(*t* *tenv*) *t0*) *bindings*

|

empty : $\bigwedge$ *tvs tenv*.

type-ps tvs tenv [] [] []

|

cons : $\bigwedge$ *tvs tenv p ps t0 ts bindings bindings'*.
type-p tvs tenv p t0 bindings ∧
type-ps tvs tenv ps ts bindings'
==>
type-ps tvs tenv (*p* # *ps*) (*t0* # *ts*) (*bindings'*@*bindings*)

inductive
type-funs :: *type-env* ⇒ *tenv-val-exp* ⇒(*varN*∗*varN*∗*exp0*)*list* ⇒(*varN*∗*t*)*list* ⇒ *bool*
and
type-es :: *type-env* ⇒ *tenv-val-exp* ⇒(*exp0*)*list* ⇒(*t*)*list* ⇒ *bool*
and
type-e :: *type-env* ⇒ *tenv-val-exp* ⇒ *exp0* ⇒ *t* ⇒ *bool* **where**

lit-int : $\bigwedge$ *tenv tenvE n*.

type-e tenv tenvE (*Lit* (*IntLit n*)) *Tint*

|

lit-char : $\bigwedge$ *tenv tenvE c1*.

type-e tenv tenvE (*Lit* (*Char c1*)) *Tchar*

|

lit-string : $\bigwedge$ *tenv tenvE s*.

type-e tenv tenvE (*Lit* (*StrLit s*)) *Tstring*

|

lit-word8 : $\bigwedge$ *tenv tenvE w*.

type-e tenv tenvE (*Lit* (*Word8 w*)) *Tword8*

|

lit-word64 : $\bigwedge$ *tenv tenvE w*.

type-e tenv tenvE (*Lit* (*Word64 w*)) *Tword64*

|

raise : $\bigwedge$ *tenv tenvE e t0*.
check-freevars (*num-tvs tenvE*) [] *t0* ∧
type-e tenv tenvE e Texn
==>
type-e tenv tenvE (*Raise e*) *t0*

|

handle : $\bigwedge$ *tenv tenvE e pes t0*.
type-e tenv tenvE e t0 ∧ (¬ (*pes* = []) ∧
((∀ (*p*,*e*) ∈
List.set pes. (∃ *bindings*.
Lem-list.allDistinct (*pat-bindings p* []) ∧
(*type-p* (*num-tvs tenvE*) *tenv p Texn bindings* ∧
type-e tenv (*bind-var-list*((*0* :: *nat*)) *bindings tenvE*) *e t0*)))))
==>
type-e tenv tenvE (*Handle e pes*) *t0*

|

con-some : $\bigwedge$ *tenv tenvE cn es tvs tn ts′ ts*.
((∀ *x* ∈ (*set ts′*). (*check-freevars* (*num-tvs tenvE*) []) *x*)) ∧
((*List.length tvs* = *List.length ts′*) ∧
(*type-es tenv tenvE es* (*List.map* (*type-subst* (*map-of* (*Lem-list-extra.zipSameLength*
tvs ts′))) *ts*) ∧
(*nsLookup*(*c0 tenv*) *cn* = *Some* (*tvs*, *ts*, *tn*))))
==>
type-e tenv tenvE (*Con* (*Some cn*) *es*) (*Tapp ts′* (*tid-exn-to-tc tn*))

|

con-none : $\bigwedge$ *tenv tenvE es ts*.
type-es tenv tenvE es ts
==>
type-e tenv tenvE (*Con None es*) (*Tapp ts TC-tup*)

|

var : $\bigwedge$ *tenv tenvE n t0 targs tvs*.
(*tvs* = *List.length targs*) ∧
(((∀ *x* ∈ (*set targs*). (*check-freevars* (*num-tvs tenvE*) []) *x*)) ∧
(*lookup-var n tenvE tenv* = *Some* (*tvs*,*t0*)))
==>
type-e tenv tenvE (*Var n*) (*deBruijn-subst*((*0* :: *nat*)) *targs t0*)

|

fn : $\bigwedge$ *tenv tenvE n e t1 t2*.

check-freevars (*num-tvs tenvE*) [] *t1* ∧
type-e tenv (*Bind-name n*((*0* :: *nat*)) *t1 tenvE*) *e t2*
==>
type-e tenv tenvE (*Fun n e*) (*Tfn t1 t2*)

|

app : $\bigwedge$ *tenv tenvE op0 es ts t0*.
type-es tenv tenvE es ts ∧
(*type-op op0 ts t0* ∧
check-freevars (*num-tvs tenvE*) [] *t0*)
==>
type-e tenv tenvE (*App op0 es*) *t0*

|

log : $\bigwedge$ *tenv tenvE l e1 e2*.
type-e tenv tenvE e1 (*Tapp* [] (*TC-name* (*Short* (''*bool*'')))) ∧
type-e tenv tenvE e2 (*Tapp* [] (*TC-name* (*Short* (''*bool*''))))
==>
type-e tenv tenvE (*Log l e1 e2*) (*Tapp* [] (*TC-name* (*Short* (''*bool*''))))

|

if' : $\bigwedge$ *tenv tenvE e1 e2 e3 t0*.
type-e tenv tenvE e1 (*Tapp* [] (*TC-name* (*Short* (''*bool*'')))) ∧
(*type-e tenv tenvE e2 t0* ∧
type-e tenv tenvE e3 t0)
==>
type-e tenv tenvE (*If e1 e2 e3*) *t0*

|

mat : $\bigwedge$ *tenv tenvE e pes t1 t2*.
type-e tenv tenvE e t1 ∧ (¬ (*pes* = [])) ∧
((∀ (*p*,*e*) ∈
  List.set pes. (∃ *bindings*.
   Lem-list.allDistinct (*pat-bindings p* []) ∧
   (*type-p* (*num-tvs tenvE*) *tenv p t1 bindings* ∧
   type-e tenv (*bind-var-list*((*0* :: *nat*)) *bindings tenvE*) *e t2*)))))
==>
type-e tenv tenvE (*Mat e pes*) *t2*

|

— *let-poly* : *forall tenv tenvE n e1 e2 t1 t2 tvs*. *is-value e1* && *type-e tenv* (*bind-tvar tvs tenvE*) *e1 t1* && *type-e tenv* (*opt-bind-name n tvs t1 tenvE*) *e2* *t2* ==> *type-e tenv tenvE* (*Let n e1 e2*) *t2 and*

let-mono : $\bigwedge$ *tenv tenvE n e1 e2 t1 t2*.
type-e tenv tenvE e1 t1 ∧
type-e tenv (*opt-bind-name n*((*0* :: *nat*)) *t1 tenvE*) *e2 t2*
==>
type-e tenv tenvE (*Let n e1 e2*) *t2*

— *and letrec* : *forall tenv tenvE funs e t tenv′ tvs*. *type-funs tenv* (*bind-var-list 0 tenv′* (*bind-tvar tvs tenvE*)) *funs tenv′* && *type-e tenv* (*bind-var-list tvs tenv′ tenvE*) *e t* ==> *type-e tenv tenvE* (*Letrec funs e*) *t*

|

letrec : $\bigwedge$ *tenv tenvE funs e t0 bindings*.
type-funs tenv (*bind-var-list*((*0* :: *nat*)) *bindings tenvE*) *funs bindings* ∧
type-e tenv (*bind-var-list*((*0* :: *nat*)) *bindings tenvE*) *e t0*
==>
type-e tenv tenvE (*Letrec funs e*) *t0*

|

typeannot: $\bigwedge$ *tenv tenvE e t0*.
check-freevars((*0* :: *nat*)) [] *t0* ∧
(*check-type-names*(*t tenv*) *t0* ∧
type-e tenv tenvE e (*type-name-subst*(*t tenv*) *t0*))
==>
type-e tenv tenvE (*Tannot e t0*) (*type-name-subst*(*t tenv*) *t0*)

|

locannot: $\bigwedge$ *tenv tenvE e l t0*.
type-e tenv tenvE e t0
==>
type-e tenv tenvE (*Lannot e l*) *t0*

|

empty : $\bigwedge$ *tenv tenvE*.

type-es tenv tenvE [] []

|

cons : $\bigwedge$ *tenv tenvE e es t0 ts*.
type-e tenv tenvE e t0 ∧
type-es tenv tenvE es ts
==>
type-es tenv tenvE (*e* # *es*) (*t0* # *ts*)

|

no-funs : $\bigwedge$ *tenv tenvE*.

type-funs tenv tenvE [] []

|

funs : $\bigwedge$ *tenv tenvE fn n e funs bindings t1 t2*.
check-freevars (*num-tvs tenvE*) [] (*Tfn t1 t2*) ∧
(*type-e tenv* (*Bind-name n*((*0* :: *nat*)) *t1 tenvE*) *e t2* ∧
(*type-funs tenv tenvE funs bindings* ∧
(*Map.map-of bindings fn* = *None*)))
==>
type-funs tenv tenvE ((*fn*, *n*, *e*)# *funs*) ((*fn*, *Tfn t1 t2*)# *bindings*)

— *val tenv-add-tvs* : *nat* −> *alist varN t* −> *alist varN* (*nat* ∗ *t*)
definition *tenv-add-tvs* :: *nat* ⇒(*string*∗*t*)*list* ⇒(*string*∗(*nat*∗*t*))*list* **where**
tenv-add-tvs tvs bindings = (
List.map (λ*x* .
(*case x of* (*n*,*t1*) => (*n*,(*tvs*,*t1*)))) *bindings*)

— *val type-pe-determ* : *type-env* −> *tenv-val-exp* −> *pat* −> *exp* −> *bool*
definition *type-pe-determ* :: *type-env* ⇒ *tenv-val-exp* ⇒ *pat* ⇒ *exp0* ⇒ *bool*
where
type-pe-determ tenv tenvE p e = ((
∀ *t1*.
∀ *tenv1*.
∀ *t2*.
∀ *tenv2*.
(*type-p*((*0* :: *nat*)) *tenv p t1 tenv1* ∧ (*type-e tenv tenvE e t1* ∧
(*type-p*((*0* :: *nat*)) *tenv p t2 tenv2* ∧ *type-e tenv tenvE e t2*)))
⟶
(*tenv1* = *tenv2*)))

— *val tscheme-inst* : (*nat* ∗ *t*) −> (*nat* ∗ *t*) −> *bool*
fun *tscheme-inst* :: *nat*∗*t* ⇒ *nat*∗*t* ⇒ *bool* **where**
tscheme-inst (*tvs-spec*, *t-spec*) (*tvs-impl*, *t-impl*) = ((
∃ *subst*.
(*List.length subst* = *tvs-impl*) ∧
(*check-freevars tvs-impl* [] *t-impl* ∧
(((∀ *x* ∈ (*set subst*). (*check-freevars tvs-spec* []) *x*)) ∧
(*deBruijn-subst*((*0* :: *nat*)) *subst t-impl* = *t-spec*)))))

inductive
type-d :: *bool* ⇒(*modN*)*list* ⇒ *decls* ⇒ *type-env* ⇒ *dec* ⇒ *decls* ⇒ *type-env* ⇒ *bool* **where**

dlet-poly : $\bigwedge$ *extra-checks tvs mn tenv p e t0 bindings decls locs.*
is-value e ∧
(*Lem-list.allDistinct* (*pat-bindings p* [])) ∧
(*type-p tvs tenv p t0 bindings* ∧
(*type-e tenv* (*bind-tvar tvs Empty*) *e t0* ∧
(*extra-checks* ⟶
((∀ *tvs′*. ∀ *bindings′*. ∀ *t′*.
(*type-p tvs′ tenv p t′ bindings′* ∧
type-e tenv (*bind-tvar tvs′ Empty*) *e t′*) ⟶
list-all2 tscheme-inst (*List.map snd* (*tenv-add-tvs tvs′ bindings′*)) (*List.map snd* (*tenv-add-tvs tvs bindings*))))))))
==>
type-d extra-checks mn decls tenv (*Dlet locs p e*)
empty-decls (| *v0* = (*alist-to-ns* (*tenv-add-tvs tvs bindings*)), *c0* = *nsEmpty*, *t* = *nsEmpty* |)

|

dlet-mono : $\bigwedge$ *extra-checks mn tenv p e t0 bindings decls locs.*
— *The following line makes sure that when the value restriction prohibits generalisation, a type error is given rather than picking an arbitrary instantiation. However, we should only do the check when the extra-checks argument tells us to.*
(*extra-checks* ⟶ (¬ (*is-value e*) ∧ *type-pe-determ tenv Empty p e*)) ∧
(*Lem-list.allDistinct* (*pat-bindings p* [])) ∧
(*type-p*((*0* :: *nat*)) *tenv p t0 bindings* ∧
type-e tenv Empty e t0))
==>
type-d extra-checks mn decls tenv (*Dlet locs p e*)
empty-decls (| *v0* = (*alist-to-ns* (*tenv-add-tvs*((*0* :: *nat*)) *bindings*)), *c0* = *nsEmpty*, *t* = *nsEmpty* |)

|

dletrec : $\bigwedge$ *extra-checks mn tenv funs bindings tvs decls locs.*
type-funs tenv (*bind-var-list*((*0* :: *nat*)) *bindings* (*bind-tvar tvs Empty*)) *funs bindings* ∧
(*extra-checks* ⟶
((∀ *tvs′*. ∀ *bindings′*.
type-funs tenv (*bind-var-list*((*0* :: *nat*)) *bindings′* (*bind-tvar tvs′ Empty*)) *funs bindings′* ⟶
list-all2 tscheme-inst (*List.map snd* (*tenv-add-tvs tvs′ bindings′*)) (*List.map snd* (*tenv-add-tvs tvs bindings*)))))
==>
type-d extra-checks mn decls tenv (*Dletrec locs funs*)
empty-decls (| *v0* = (*alist-to-ns* (*tenv-add-tvs tvs bindings*)), *c0* = *nsEmpty*, *t* = *nsEmpty* |)

|

dtype : $\bigwedge$ *extra-checks mn tenv tdefs decls defined-types′ decls′ tenvT locs.*
check-ctor-tenv (*nsAppend tenvT*(*t tenv*)) *tdefs* ∧
((*defined-types′* = *List.set* (*List.map* (*λx* .
(*case x of* (*tvs*,*tn*,*ctors*) => (*mk-id mn tn*))) *tdefs*)) ∧
(*disjnt defined-types′*(*defined-types0 decls*) ∧
((*tenvT* = *alist-to-ns* (*List.map* (*λx* .
(*case x of*
(*tvs*,*tn*,*ctors*) => (*tn*, (*tvs*, *Tapp* (*List.map Tvar tvs*)
(*TC-name* (*mk-id mn tn*))))
)) *tdefs*)) ∧
(*decls′* = (| *defined-mods0* = ({}), *defined-types0* = *defined-types′*, *defined-exns* = ({}) |)))))
==>
type-d extra-checks mn decls tenv (*Dtype locs tdefs*)
decls′ (| *v0* = *nsEmpty*, *c0* = (*build-ctor-tenv mn* (*nsAppend tenvT*(*t tenv*)) *tdefs*), *t* = *tenvT* |)

|

dtabbrev : $\bigwedge$ *extra-checks mn decls tenv tvs tn t0 locs.*
check-freevars((*0* :: *nat*)) *tvs t0* ∧
(*check-type-names*(*t tenv*) *t0* ∧
Lem-list.allDistinct tvs)
==>
type-d extra-checks mn decls tenv (*Dtabbrev locs tvs tn t0*)
empty-decls (| *v0* = *nsEmpty*, *c0* = *nsEmpty*,
t = (*nsSing tn* (*tvs*,*type-name-subst*(*t tenv*) *t0*)) |)

|

dexn : $\bigwedge$ *extra-checks mn tenv cn ts decls decls′ locs.*
check-exn-tenv mn cn ts ∧
(¬ (*mk-id mn cn* ∈(*defined-exns decls*)) ∧
(((∀ *x* ∈ (*set ts*). (*check-type-names*(*t tenv*)) *x*)) ∧
(*decls′* = (| *defined-mods0* = ({}), *defined-types0* = ({}), *defined-exns* = ({*mk-id mn cn*}) |)))))
==>
type-d extra-checks mn decls tenv (*Dexn locs cn ts*)
decls′ (| *v0* = *nsEmpty*,
c0 = (*nsSing cn* ([], *List.map* (*type-name-subst*(*t tenv*)) *ts*, *TypeExn* (*mk-id mn cn*))),
t = *nsEmpty* |)

inductive
type-ds :: *bool* ⇒(*modN*)*list* ⇒ *decls* ⇒ *type-env* ⇒(*dec*)*list* ⇒ *decls* ⇒ *type-env* ⇒ *bool* **where**

empty : $\bigwedge$ *extra-checks mn tenv decls.*

type-ds extra-checks mn decls tenv []
  empty-decls (| *v0* = *nsEmpty*, *c0* = *nsEmpty*, *t* = *nsEmpty* |)

|

cons : $\bigwedge$ *extra-checks mn tenv d ds tenv1 tenv2 decls decls1 decls2*.
type-d extra-checks mn decls tenv d decls1 tenv1 ∧
type-ds extra-checks mn (*union-decls decls1 decls*) (*extend-dec-tenv tenv1 tenv*) *ds decls2 tenv2*
==>
type-ds extra-checks mn decls tenv (*d* # *ds*)
 (*union-decls decls2 decls1*) (*extend-dec-tenv tenv2 tenv1*)

inductive
type-specs :: (*modN*)*list* ⇒ *tenv-abbrev* ⇒ *specs* ⇒ *decls* ⇒ *type-env* ⇒ *bool*
where

empty : $\bigwedge$ *mn tenvT*.

type-specs mn tenvT []
  empty-decls (| *v0* = *nsEmpty*, *c0* = *nsEmpty*, *t* = *nsEmpty* |)

|

sval : $\bigwedge$ *mn tenvT x t0 specs tenv fvs decls subst*.
check-freevars((*0* :: *nat*)) *fvs t0* ∧
(*check-type-names tenvT t0* ∧
(*type-specs mn tenvT specs decls tenv* ∧
(*subst* = *map-of* (*Lem-list-extra.zipSameLength fvs* (*List.map Tvar-db* (*genlist* (λ *x* . *x*) (*List.length fvs*)))))))
==>
type-specs mn tenvT (*Sval x t0* # *specs*)
  decls
  (*extend-dec-tenv tenv*
   (| *v0* = (*nsSing x* (*List.length fvs*, *type-subst subst* (*type-name-subst tenvT t0*))),
     c0 = *nsEmpty*,
     t = *nsEmpty* |))

|

stype : $\bigwedge$ *mn tenvT tenv td specs decls′ decls tenvT′*.
(*tenvT′* = *alist-to-ns* (*List.map* (λ*x* .
  (*case x of*
    (*tvs*,*tn*,*ctors*) => (*tn*, (*tvs*, *Tapp* (*List.map Tvar tvs*)
                  (*TC-name* (*mk-id mn tn*))))
  )) *td*)) ∧
(*check-ctor-tenv* (*nsAppend tenvT′ tenvT*) *td* ∧
(*type-specs mn* (*nsAppend tenvT′ tenvT*) *specs decls tenv* ∧

$(decls' = (|\ defined\text{-}mods0\ =\ (\{\}),$
$\quad defined\text{-}types0\ =\ (List.set\ (List.map\ (\ \lambda x\ .$
$(case\ \ x\ of\ (tvs,tn,ctors)\ =>\ (mk\text{-}id\ mn\ tn)\ ))\ td)),$
$\quad defined\text{-}exns\ =\ (\{\})\ |))))$
$==>$
$type\text{-}specs\ mn\ tenvT\ (Stype\ td\ \#\ specs)$
$(union\text{-}decls\ decls\ decls')$
$(extend\text{-}dec\text{-}tenv\ tenv$
$(|\ v0\ =\ nsEmpty,$
$c0\ =\ (build\text{-}ctor\text{-}tenv\ mn\ (nsAppend\ tenvT'\ tenvT)\ td),$
$t\ =\ tenvT'\ |))$

|

$stabbrev:\ \bigwedge\ mn\ tenvT\ tenvT'\ tvs\ tn\ t0\ specs\ decls\ tenv.$
$Lem\text{-}list.allDistinct\ tvs\ \wedge$
$(check\text{-}freevars((\ 0\ ::\ nat))\ tvs\ t0\ \wedge$
$(check\text{-}type\text{-}names\ tenvT\ t0\ \wedge$
$((tenvT'\ =\ nsSing\ tn\ (tvs,type\text{-}name\text{-}subst\ tenvT\ t0))\ \wedge$
$type\text{-}specs\ mn\ (nsAppend\ tenvT'\ tenvT)\ specs\ decls\ tenv)))$
$==>$
$type\text{-}specs\ mn\ tenvT\ (Stabbrev\ tvs\ tn\ t0\ \#\ specs)$
$decls\ (extend\text{-}dec\text{-}tenv\ tenv\ (|\ v0\ =\ nsEmpty,\ c0\ =\ nsEmpty,\ t\ =\ tenvT'\ |))$

|

$sexn:\ \bigwedge\ mn\ tenvT\ tenv\ cn\ ts\ specs\ decls.$
$check\text{-}exn\text{-}tenv\ mn\ cn\ ts\ \wedge$
$(type\text{-}specs\ mn\ tenvT\ specs\ decls\ tenv\ \wedge$
$((\forall\ \ x\in(set\ ts).\ \ (check\text{-}type\text{-}names\ tenvT)\ x)))$
$==>$
$type\text{-}specs\ mn\ tenvT\ (Sexn\ cn\ ts\ \#\ specs)$
$(union\text{-}decls\ decls\ (|\ defined\text{-}mods0\ =\ (\{\}),\ defined\text{-}types0\ =\ (\{\}),\ defined\text{-}exns\ =$
$(\{mk\text{-}id\ mn\ cn\})\ |))$
$(extend\text{-}dec\text{-}tenv\ tenv$
$(|\ v0\ =\ nsEmpty,$
$c0\ =\ (nsSing\ cn\ ([],\ List.map\ (type\text{-}name\text{-}subst\ tenvT)\ ts,\ TypeExn\ (mk\text{-}id$
$mn\ cn))),$
$t\ =\ nsEmpty\ |))$

|

$stype\text{-}opq:\ \bigwedge\ mn\ tenvT\ tenv\ tn\ specs\ tvs\ decls\ tenvT'.$
$Lem\text{-}list.allDistinct\ tvs\ \wedge$
$((tenvT'\ =\ nsSing\ tn\ (tvs,\ Tapp\ (List.map\ Tvar\ tvs)\ (TC\text{-}name\ (mk\text{-}id\ mn\ tn))))$
$\wedge$
$type\text{-}specs\ mn\ (nsAppend\ tenvT'\ tenvT)\ specs\ decls\ tenv)$
$==>$
$type\text{-}specs\ mn\ tenvT\ (Stype\text{-}opq\ tvs\ tn\ \#\ specs)$

(*union-decls decls* (| *defined-mods0* = ({}), *defined-types0* = ({*mk-id mn tn*}), *defined-exns* = ({}) |))
(*extend-dec-tenv tenv* (| *v0* = *nsEmpty*, *c0* = *nsEmpty*, *t* = *tenvT′* |))

— *val weak-decls* : *decls* −> *decls* −> *bool*
definition *weak-decls* :: *decls* ⇒ *decls* ⇒ *bool* **where**
weak-decls decls-impl decls-spec = (
((*defined-mods0 decls-impl*) =(*defined-mods0 decls-spec*)) ∧
(((*defined-types0 decls-spec*) ⊆(*defined-types0 decls-impl*)) ∧
((*defined-exns decls-spec*) ⊆(*defined-exns decls-impl*))))

— *val weak-tenvT* : *id modN typeN* −> (*list tvarN* ∗ *t*) −> (*list tvarN* ∗ *t*) −> *bool*
fun *weak-tenvT* :: ((*modN*),(*typeN*))*id0* ⇒(*string*)*list*∗*t* ⇒(*string*)*list*∗*t* ⇒ *bool* **where**
weak-tenvT n (*tvs-spec*, *t-spec*) (*tvs-impl*, *t-impl*) = (
(
— *For simplicity*, *we reject matches that differ only by renaming of bound type variablestvs-spec* = *tvs-impl*) ∧
((*t-spec* = *t-impl*) ∨
(
— *The specified type is opaquet-spec* = *Tapp* (*List.map Tvar tvs-spec*) (*TC-name n*))))

definition *tscheme-inst2* :: ′*a* ⇒ *nat*∗*t* ⇒ *nat*∗*t* ⇒ *bool* **where**
tscheme-inst2 - *ts1 ts2* = (*tscheme-inst ts1 ts2*)

— *val weak-tenv* : *type-env* −> *type-env* −> *bool*
definition *weak-tenv* :: *type-env* ⇒ *type-env* ⇒ *bool* **where**
weak-tenv tenv-impl tenv-spec = (
nsSub tscheme-inst2(*v0 tenv-spec*)(*v0 tenv-impl*) ∧
(*nsSub* (*λx* .
(*case x of* - => *λ x y* . *x* = *y*))(*c0 tenv-spec*)(*c0 tenv-impl*) ∧
nsSub weak-tenvT(*t tenv-spec*)(*t tenv-impl*)))

inductive
check-signature :: (*modN*)*list* ⇒ *tenv-abbrev* ⇒ *decls* ⇒ *type-env* ⇒(*specs*)*option* ⇒ *decls* ⇒ *type-env* ⇒ *bool* **where**

none : ⋀ *mn tenvT decls tenv*.

check-signature mn tenvT decls tenv None decls tenv

|

some : $\bigwedge$ *mn specs tenv-impl tenv-spec decls-impl decls-spec tenvT*.
weak-tenv tenv-impl tenv-spec ∧
(*weak-decls decls-impl decls-spec* ∧
type-specs mn tenvT specs decls-spec tenv-spec)
==>
check-signature mn tenvT decls-impl tenv-impl (*Some specs*) *decls-spec tenv-spec*

definition *tenvLift* :: *string* ⇒ *type-env* ⇒ *type-env* **where**
tenvLift mn tenv = (
(| *v0* = (*nsLift mn*(*v0* *tenv*)), *c0* = (*nsLift mn*(*c0* *tenv*)), *t* = (*nsLift mn*(*t* *tenv*)) |))

inductive
type-top :: *bool* ⇒ *decls* ⇒ *type-env* ⇒ *top0* ⇒ *decls* ⇒ *type-env* ⇒ *bool* **where**

tdec : $\bigwedge$ *extra-checks tenv d tenv′ decls decls′*.
type-d extra-checks [] *decls tenv d decls′ tenv′*
==>
type-top extra-checks decls tenv (*Tdec d*) *decls′ tenv′*

|

tmod : $\bigwedge$ *extra-checks tenv mn spec ds tenv-impl tenv-spec decls decls-impl decls-spec*.
¬ ([*mn*] ∈(*defined-mods0* *decls*)) ∧
(*type-ds extra-checks* [*mn*] *decls tenv ds decls-impl tenv-impl* ∧
check-signature [*mn*](*t* *tenv*) *decls-impl tenv-impl spec decls-spec tenv-spec*)
==>
type-top extra-checks decls tenv (*Tmod mn spec ds*)
(*union-decls* (| *defined-mods0* = ({[*mn*]}), *defined-types0* = ({}), *defined-exns* = ({}) |) *decls-spec*)
(*tenvLift mn tenv-spec*)

inductive
type-prog :: *bool* ⇒ *decls* ⇒ *type-env* ⇒(*top0*)*list* ⇒ *decls* ⇒ *type-env* ⇒ *bool* **where**

empty : $\bigwedge$ *extra-checks tenv decls*.

type-prog extra-checks decls tenv [] *empty-decls* (| *v0* = *nsEmpty*, *c0* = *nsEmpty*, *t* = *nsEmpty* |)

|

cons : $\bigwedge$ *extra-checks tenv top0 tops tenv1 tenv2 decls decls1 decls2*.
type-top extra-checks decls tenv top0 decls1 tenv1 ∧
type-prog extra-checks (*union-decls decls1 decls*) (*extend-dec-tenv tenv1 tenv*) *tops decls2 tenv2*

==>
type-prog extra-checks decls tenv (*top0* # *tops*)
(*union-decls decls2 decls1*) (*extend-dec-tenv tenv2 tenv1*)
end

Chapter 19

Generated by Lem from *semantics/typeSystem.lem*.

theory *TypeSystemAuxiliary*

imports
Main
HOL−Library.Datatype-Records
LEM.Lem-pervasives-extra
Lib
Namespace
Ast
SemanticPrimitives
TypeSystem

begin

— **
—
— *Termination Proofs*
—
— **

termination *check-freevars* **by** *lexicographic-order*

termination *type-subst* **by** *lexicographic-order*

termination *deBruijn-inc* **by** *lexicographic-order*

termination *deBruijn-subst* **by** *lexicographic-order*

termination *check-type-names* **by** *lexicographic-order*

termination *type-name-subst* **by** *lexicographic-order*

termination *is-value* **by** *lexicographic-order*

end

Part II

Proofs ported from HOL4

Chapter 20

Adaptations for Isabelle

```
theory Semantic-Extras
imports
  generated/CakeML/BigStep
  generated/CakeML/SemanticPrimitivesAuxiliary
  generated/CakeML/AstAuxiliary
  generated/CakeML/Evaluate
  HOL−Library.Simps-Case-Conv
begin

type-synonym exp = exp0

hide-const (open) sem-env.v

code-pred
  (modes: evaluate: i ⇒ i ⇒ i ⇒ i ⇒ o ⇒ bool as compute
     and evaluate-list: i ⇒ i ⇒ i ⇒ i ⇒ o ⇒ bool
     and evaluate-match: i ⇒ i ⇒ i ⇒ i ⇒ i ⇒ i ⇒ o ⇒ bool) evaluate .

code-pred (modes: i ⇒ i ⇒ i ⇒ i ⇒ i ⇒ o ⇒ bool) evaluate-dec .
code-pred (modes: i ⇒ i ⇒ i ⇒ i ⇒ i ⇒ o ⇒ bool) evaluate-decs .
code-pred (modes: i ⇒ i ⇒ i ⇒ i ⇒ o ⇒ bool) evaluate-top .
code-pred (modes: i ⇒ i ⇒ i ⇒ i ⇒ o ⇒ bool as compute-prog) evaluate-prog .

termination pmatch-list by lexicographic-order
termination do-eq-list by lexicographic-order

lemma all-distinct-alt-def: allDistinct = distinct
proof
  fix xs :: 'a list
  show allDistinct xs = distinct xs
    by (induct xs) auto
qed

lemma find-recfun-someD:
```

assumes *find-recfun n funs* = *Some* (*x*, *e*)
shows (*n*, *x*, *e*) ∈ *set funs*
using *assms*
by (*induct funs*) (*auto split*: *if-splits*)

lemma *find-recfun-alt-def*[*simp*]: *find-recfun n funs* = *map-of funs n*
by (*induction funs*) *auto*

lemma *size-list-rev*[*simp*]: *size-list f* (*rev xs*) = *size-list f xs*
by (*auto simp*: *size-list-conv-sum-list rev-map*[*symmetric*])

lemma *do-if-cases*:
obtains
(*none*) *do-if v e1 e2* = *None*
| (*true*) *do-if v e1 e2* = *Some e1*
| (*false*) *do-if v e1 e2* = *Some e2*
unfolding *do-if-def*
by *meson*

case-of-simps *do-log-alt-def*: *do-log.simps*
case-of-simps *do-con-check-alt-def*: *do-con-check.simps*
case-of-simps *list-result-alt-def*: *list-result.simps*

context begin

private fun-cases *do-logE*: *do-log op v e* = *res*

lemma *do-log-exp*: *do-log op v e* = *Some* (*Exp e'*) ⟹ *e* = *e'*
by (*erule do-logE*)
(*auto split*: *v.splits option.splits if-splits tid-or-exn.splits id0.splits list.splits*)

end

lemma *c-of-merge*[*simp*]: *c* (*extend-dec-env env2 env1*) = *nsAppend* (*c env2*) (*c env1*)
by (*cases env1*; *cases env2*; *simp add*: *extend-dec-env-def*)

lemma *v-of-merge*[*simp*]: *sem-env.v* (*extend-dec-env env2 env1*) = *nsAppend* (*sem-env.v env2*) (*sem-env.v env1*)
by (*cases env1*; *cases env2*; *simp add*: *extend-dec-env-def*)

lemma *nsEmpty-nsAppend*[*simp*]: *nsAppend e nsEmpty* = *e nsAppend nsEmpty e* = *e*
by (*cases e*; *auto simp*: *nsEmpty-def*)+

lemma *do-log-cases*:
obtains
(*none*) *do-log op v e* = *None*
| (*val*) *v'* **where** *do-log op v e* = *Some* (*Val v'*)

| (*exp*) *do-log op v e* = *Some* (*Exp e*)
proof (*cases do-log op v e*)
  case *None*
  then show *?thesis* **using** *none* **by** *metis*
next
  case (*Some res*)
  with *val exp* **show** *?thesis*
    by (*cases res*) (*metis do-log-exp*)+
qed

context begin

private fun-cases *do-opappE*: *do-opapp vs* = *Some res*

lemma *do-opapp-cases*:
  assumes *do-opapp vs* = *Some* (*env'*, *exp'*)
  obtains (*closure*) *env n v0*
    where *vs* = [*Closure env n exp'*, *v0*]
      env' = (*env* ⦇ *sem-env.v* := *nsBind n v0* (*sem-env.v env*) ⦈)
  | (*recclosure*) *env funs name n v0*
    where *vs* = [*Recclosure env funs name*, *v0*]
    and *allDistinct* (*map* (λ(*f*, -, -). *f*) *funs*)
    and *find-recfun name funs* = *Some* (*n*, *exp'*)
    and *env'* = (*env* ⦇ *sem-env.v* := *nsBind n v0* (*build-rec-env funs env* (*sem-env.v env*)) ⦈)
proof −
  show *thesis*
    using *assms*
    apply (*rule do-opappE*)
    apply (*rule closure*; *auto*)
    apply (*auto split*: *if-splits option.splits*)
    apply (*rule recclosure*)
    apply *auto*
    done
qed

end

lemmas *evaluate-induct* =
  evaluate-match-evaluate-list-evaluate.inducts[*split-format*(*complete*)]

lemma *evaluate-clock-mono*:
  evaluate-match ck env s v pes v' (*s'*, *r1*) ⟹ *clock s'* ≤ *clock s*
  evaluate-list ck env s es (*s'*, *r2*) ⟹ *clock s'* ≤ *clock s*
  evaluate ck env s e (*s'*, *r3*) ⟹ *clock s'* ≤ *clock s*
by (*induction rule*: *evaluate-induct*)
  (*auto simp del*: *do-app.simps simp*: *datatype-record-update split*: *state.splits if-splits*)

lemma *evaluate-list-singleton-valE*:
  assumes *evaluate-list ck env s* [*e*] (*s′*, *Rval vs*)
  obtains *v* **where** *vs* = [*v*] *evaluate ck env s e* (*s′*, *Rval v*)
using *assms*
by (*auto elim*: *evaluate-list.cases*)

lemma *evaluate-list-singleton-errD*:
  assumes *evaluate-list ck env s* [*e*] (*s′*, *Rerr err*)
  shows *evaluate ck env s e* (*s′*, *Rerr err*)
using *assms*
by (*auto elim*: *evaluate-list.cases*)

lemma *evaluate-list-singleton-cases*:
  assumes *evaluate-list ck env s* [*e*] *res*
  obtains (*val*) *s′ v* **where** *res* = (*s′*, *Rval* [*v*]) *evaluate ck env s e* (*s′*, *Rval v*)
    | (*err*) *s′ err* **where** *res* = (*s′*, *Rerr err*) *evaluate ck env s e* (*s′*, *Rerr err*)
using *assms*
apply −
apply (*ind-cases evaluate-list ck env s* [*e*] *res*)
apply *auto*
apply (*ind-cases evaluate-list ck env s2* [] (*s3*, *Rval vs*) **for** *s2 s3 vs*)
apply *auto*
apply (*ind-cases evaluate-list ck env s2* [] (*s3*, *Rerr err*) **for** *s2 s3 err*)
done

lemma *evaluate-list-singletonI*:
  assumes *evaluate ck env s e* (*s′*, *r*)
  shows *evaluate-list ck env s* [*e*] (*s′*, *list-result r*)
using *assms*
by (*cases r*) (*auto intro*: *evaluate-match-evaluate-list-evaluate.intros*)

lemma *prod-result-cases*:
  obtains (*val*) *s v* **where** *r* = (*s*, *Rval v*)
    | (*err*) *s err* **where** *r* = (*s*, *Rerr err*)
apply (*cases r*)
subgoal for - *b*
  apply (*cases b*)
  by *auto*
done

lemma *do-con-check-build-conv*: *do-con-check* (*c env*) *cn* (*length es*) ⟹ *build-conv* (*c env*) *cn vs* ≠ *None*
by (*cases cn*) (*auto split*: *option.splits*)

fun *match-result* :: (*v*)*sem-env* ⇒ *′ffi state* ⇒ *v* ⇒(*pat*∗*exp*)*list* ⇒ *v* ⇒ (*exp* × (*char list* × *v*) *list*, *v*)*result* **where**
match-result - - - [] *err-v* = *Rerr* (*Rraise err-v*) |
match-result env s v0 ((*p*, *e*) # *pes*) *err-v* =
  (*if Lem-list.allDistinct* (*pat-bindings p* []) *then*

(*case pmatch* (*sem-env.c env*) (*refs s*) *p v0* [] *of*
Match env′ ⇒ *Rval* (*e*, *env′*) |
No-match ⇒ *match-result env s v0 pes err-v* |
Match-type-error ⇒ *Rerr* (*Rabort Rtype-error*))
else
Rerr (*Rabort Rtype-error*))

case-of-simps *match-result-alt-def*: *match-result.simps*

lemma *match-result-sound*:
case match-result env s v0 pes err-v of
Rerr err ⇒ *evaluate-match ck env s v0 pes err-v* (*s*, *Rerr err*)
| *Rval* (*e*, *env′*) ⇒
∀ *bv*.
evaluate ck (*env* (| *sem-env.v* := *nsAppend* (*alist-to-ns env′*)(*sem-env.v env*) |)) *s e bv* ⟶
evaluate-match ck env s v0 pes err-v bv
by (*induction rule*: *match-result.induct*)
(*auto intro*: *evaluate-match-evaluate-list-evaluate.intros split*: *match-result.splits result.splits*)

lemma *match-result-sound-val*:
assumes *match-result env s v0 pes err-v* = *Rval* (*e*, *env′*)
assumes *evaluate ck* (*env* (| *sem-env.v* := *nsAppend* (*alist-to-ns env′*)(*sem-env.v env*) |)) *s e bv*
shows *evaluate-match ck env s v0 pes err-v bv*
proof −
note *match-result-sound*[**where** *env* = *env* **and** *s* = *s* **and** *?v0.0* = *v0* **and** *pes* = *pes* **and** *err-v* = *err-v*, *unfolded assms result.case prod.case*]
with *assms* **show** *?thesis* **by** *blast*
qed

lemma *match-result-sound-err*:
assumes *match-result env s v0 pes err-v* = *Rerr err*
shows *evaluate-match ck env s v0 pes err-v* (*s*, *Rerr err*)
proof −
note *match-result-sound*[**where** *env* = *env* **and** *s* = *s* **and** *?v0.0* = *v0* **and** *pes* = *pes* **and** *err-v* = *err-v*, *unfolded assms result.case prod.case*]
then show *?thesis* **by** *blast*
qed

lemma *match-result-correct*:
assumes *evaluate-match ck env s v0 pes err-v* (*s′*, *bv*)
shows *case bv of*
Rval v ⇒
∃ *e env′*. *match-result env s v0 pes err-v* = *Rval* (*e*, *env′*) ∧ *evaluate ck* (*env* (| *sem-env.v* := *nsAppend* (*alist-to-ns env′*) (*sem-env.v env*) |)) *s e* (*s′*, *Rval v*)
| *Rerr err* ⇒

(*match-result env s v0 pes err-v* = *Rerr err*) ∨
(∃ *e env′*. *match-result env s v0 pes err-v* = *Rval* (*e*, *env′*) ∧ *evaluate ck* (*env* ⦇ *sem-env.v* := *nsAppend* (*alist-to-ns env′*) (*sem-env.v env*) ⦈) *s e* (*s′*, *Rerr err*))
using *assms*
proof (*induction pes*)
 case (*Cons pe pes*)
 from *Cons.prems* **show** *?case*
  proof *cases*
   case (*mat-cons1 env′ p e*)
   then show *?thesis*
    by (*cases bv*) *auto*
  next
   case (*mat-cons2 p e*)
   then show *?thesis*
    using *Cons.IH*
    by (*cases bv*) *auto*
  qed *auto*
qed (*auto elim*: *evaluate-match.cases*)

end

Chapter 21

Functional big-step semantics

21.1 Termination proof

theory *Evaluate-Termination*
  imports *Semantic-Extras*
begin

case-of-simps *fix-clock-alt-def*: *fix-clock.simps*

primrec *size-exp′* :: *exp* ⇒ *nat* **where**
size-exp′ (*Raise e*) = *Suc* (*size-exp′ e*) |
[*simp del*]: *size-exp′* (*Handle e pes*) = *Suc* (*size-exp′ e* + *size-list* (λ(*p*, *es*). *Suc* (*size p* + *es*)) (*map* (*map-prod id size-exp′*) *pes*)) |
size-exp′ (*Con* - *es*) = *Suc* (*size-list id* (*map size-exp′ es*)) |
size-exp′ (*Fun* - *e*) = *Suc* (*size-exp′ e*) |
size-exp′ (*App* - *es*) = *Suc* (*size-list id* (*map size-exp′ es*)) |
size-exp′ (*Log* - *e f*) = *Suc* (*size-exp′ e* + *size-exp′ f*) |
size-exp′ (*If e f g*) = *Suc* (*size-exp′ e* + *size-exp′ f* + *size-exp′ g*) |
[*simp del*]: *size-exp′* (*Mat e pes*) = *Suc* (*size-exp′ e* + *size-list* (λ(*p*, *es*). *Suc* (*size p* + *es*)) (*map* (*map-prod id size-exp′*) *pes*)) |
size-exp′ (*Let* - *e f*) = *Suc* (*size-exp′ e* + *size-exp′ f*) |
[*simp del*]: *size-exp′* (*Letrec defs e*) = *Suc* (*size-exp′ e* + *size-list* (λ(-, -, *es*). *Suc* (*Suc es*)) (*map* (*map-prod id* (*map-prod id size-exp′*)) *defs*)) |
size-exp′ (*Tannot e* -) = *Suc* (*size-exp′ e*) |
size-exp′ (*Lannot e* -) = *Suc* (*size-exp′ e*) |
size-exp′ (*Lit* -) = *0* |
size-exp′ (*Var* -) = *0*

lemma [*simp*]:
  size-exp′ (*Mat e pes*) = *Suc* (*size-exp′ e* + *size-list* (*size-prod size size-exp′*) *pes*)
apply (*simp add*: *size-exp′.simps size-list-conv-sum-list*)
apply (*rule arg-cong*[**where** *f* = *sum-list*])
apply *auto*
done

lemma [*simp*]:
size-exp′ (*Handle e pes*) = *Suc* (*size-exp′ e* + *size-list* (*size-prod size size-exp′*) *pes*)
apply (*simp add*: *size-exp′.simps size-list-conv-sum-list*)
apply (*rule arg-cong*[**where** *f* = *sum-list*])
apply *auto*
done

lemma [*simp*]:
size-exp′ (*Letrec defs e*) = *Suc* (*size-exp′ e* + *size-list* (*size-prod* (λ-. *0*) (*size-prod* (λ-. *0*) *size-exp′*)) *defs*)
apply (*simp add*: *size-exp′.simps size-list-conv-sum-list*)
apply (*rule arg-cong*[**where** *f* = *sum-list*])
apply *auto*
done

context begin

private definition *fun-evaluate-relation* **where**
fun-evaluate-relation = *inv-image* (*less-than* <∗*lex*∗> *less-than*) (λ*x*.
case x of
Inr (*s*, -, *es*) ⇒ (*clock s*, *size-list size-exp′ es*)
| *Inl* (*s*,-,-,*pes*,-) ⇒ (*clock s*, *size-list* (*size-prod size size-exp′*) *pes*))

termination *fun-evaluate*
by (*relation fun-evaluate-relation*;
auto
simp: *fun-evaluate-relation-def fix-clock-alt-def dec-clock-def do-if-def do-log-alt-def*
simp: *datatype-record-update*
split: *prod.splits state.splits lop.splits v.splits option.splits if-splits tid-or-exn.splits id0.splits list.splits*)

end

end

21.2 Simplifiying the definition

theory *Evaluate-Clock*
imports *Evaluate-Termination*
begin

hide-const (**open**) *sem-env.v*

lemma *fix-clock*:
fix-clock s1 (*s2*, *x*) = (*s*, *x*) ⟹ *clock s* ≤ *clock s1*
fix-clock s1 (*s2*, *x*) = (*s*, *x*) ⟹ *clock s* ≤ *clock s2*
unfolding *fix-clock-alt-def* **by** *auto*

lemma *dec-clock*[*simp*]: *clock* (*dec-clock st*) = *clock st* − *1*
unfolding *dec-clock-def* **by** *auto*

context begin

private lemma *fun-evaluate-clock0*:
  clock (*fst* (*fun-evaluate-match s1 env v p v′*)) ≤ *clock s1*
  clock (*fst* (*fun-evaluate s1 env e*)) ≤ *clock s1*
proof (*induction rule*: *fun-evaluate-match-fun-evaluate.induct*)
  case (*2 st env e1 e2 es*)

  obtain *st′ r* **where** ∗[*simp*]: *fix-clock st* (*fun-evaluate st env* [*e1*]) = (*st′*, *r*)
    by *force*

  show *?case*
    apply (*auto split*: *prod.splits result.splits*)
    subgoal
      using *2*(*2*)[*OF* ∗[*symmetric*]]
      by (*metis* ∗ *fix-clock*(*1*) *fix-clock.simps fst-conv le-trans prod.collapse*)
    subgoal
      using *2*(*2*)[*OF* ∗[*symmetric*]]
      by (*metis* ∗ *fix-clock*(*1*) *fix-clock.simps fst-conv le-trans prod.collapse*)
    subgoal
      by (*metis* ∗ *fix-clock*(*1*) *fix-clock.simps prod.collapse prod.sel*(*2*))
    done
next
  case (*5 st env e pes*)

  obtain *st′ r* **where** ∗[*simp*]: *fix-clock st* (*fun-evaluate st env* [*e*]) = (*st′*, *r*)
    by *force*

  show *?case*
    apply (*auto split*: *prod.splits result.splits*)
    subgoal
      by (*metis* ∗ *fix-clock*(*1*) *fix-clock.simps prod.collapse prod.sel*(*2*))
    subgoal
      using *5*(*2*)[*OF* ∗[*symmetric*]]
      by (*smt* (*verit*) ∗ *5.IH*(*1*) *dual-order.trans eq-fst-iff error-result.exhaust error-result.simps*(*5*) *error-result.simps*(*6*) *fix-clock*(*2*) *fix-clock.simps*)
    done
next
  case (*9 st env op1 es*)

  obtain *st′ r* **where** ∗[*simp*]: *fix-clock st* (*fun-evaluate st env* (*rev es*)) = (*st′*, *r*)
    by *force*

  note *do-app.simps*[*simp del*]

  show *?case*

apply (*auto split*: *prod.splits result.splits option.splits if-splits*)
subgoal
by (*metis* ∗ *fix-clock*(*1*) *fix-clock.simps prod.collapse prod.sel*(*2*))
subgoal
by (*metis* ∗ *fix-clock*(*1*) *fix-clock.simps prod.collapse prod.sel*(*2*))
subgoal
by (*smt* ∗ *9.IH*(*2*) *One-nat-def Suc-pred dec-clock dual-order.trans fix-clock*(*1*) *fix-clock.simps fst-conv le-imp-less-Suc nat-less-le prod.collapse*)
subgoal
by (*metis* ∗ *fix-clock*(*1*) *fix-clock.simps fst-conv prod.collapse*)
subgoal
using *9*(*2*)[*OF* ∗[*symmetric*], *simplified*]
by (*metis* ∗ *Suc-pred dual-order.trans fix-clock*(*1*) *fix-clock.simps le-imp-less-Suc less-irrefl-nat nat-le-linear prod.collapse prod.sel*(*2*))
subgoal
by (*metis* ∗ *fix-clock*(*1*) *fix-clock.simps prod.collapse prod.sel*(*2*))
subgoal
using *9*(*2*)[*OF* ∗[*symmetric*], *simplified*]
by (*metis* ∗ *Suc-pred dual-order.trans fix-clock*(*1*) *fix-clock.simps le-imp-less-Suc less-irrefl-nat nat-le-linear prod.collapse prod.sel*(*2*))
done
next
case (*10 st env lop e1 e2*)

obtain *st′ r* **where** ∗[*simp*]: *fix-clock st* (*fun-evaluate st env* [*e1*]) = (*st′*, *r*)
by *force*

show *?case*
apply (*auto split*: *prod.splits result.splits option.splits exp-or-val.splits*)
subgoal
by (*metis* ∗ *fix-clock*(*1*) *fix-clock.simps fst-conv prod.collapse*)
subgoal
using *10*(*2*)[*OF* ∗[*symmetric*]]
by (*metis* (*no-types*, *lifting*) ∗ *dual-order.trans fix-clock*(*1*) *fix-clock.simps fstI prod.collapse*)
subgoal
by (*metis* ∗ *fix-clock*(*1*) *fix-clock.simps prod.collapse snd-conv*)
subgoal
by (*metis* ∗ *fix-clock*(*1*) *fix-clock.simps fst-conv prod.exhaust-sel*)
done
next
case (*11 st env e1 e2 e3*)

obtain *st′ r* **where** ∗[*simp*]: *fix-clock st* (*fun-evaluate st env* [*e1*]) = (*st′*, *r*)
by *force*

show *?case*
apply (*auto split*: *prod.splits result.splits option.splits*)
subgoal

by (*metis* ∗ *fix-clock*(*1*) *fix-clock.simps fst-conv prod.collapse*)
subgoal
using *11*(*2*)[*OF* ∗[*symmetric*]]
by (*metis* (*no-types*, *lifting*) ∗ *dual-order.trans eq-fst-iff fix-clock*(*1*) *fix-clock.simps*)
subgoal
by (*metis* ∗ *fix-clock*(*1*) *fix-clock.simps fst-conv prod.exhaust-sel*)
done
next
case (*12 st env e pes*)

obtain *st′ r* **where** ∗[*simp*]: *fix-clock st* (*fun-evaluate st env* [*e*]) = (*st′*, *r*)
by *force*

show *?case*
apply (*auto split*: *prod.splits result.splits option.splits*)
subgoal
using *12*(*2*)[*OF* ∗[*symmetric*]]
by (*metis* (*no-types*, *lifting*) ∗ *dual-order.trans eq-fst-iff fix-clock*(*1*) *fix-clock.simps*)
subgoal
by (*metis* ∗ *fix-clock*(*1*) *fix-clock.simps fst-conv prod.exhaust-sel*)
done
next
case (*13 st env xo e1 e2*)

obtain *st′ r* **where** ∗[*simp*]: *fix-clock st* (*fun-evaluate st env* [*e1*]) = (*st′*, *r*)
by *force*

show *?case*
apply (*auto split*: *prod.splits result.splits option.splits*)
subgoal
using *13*(*2*)[*OF* ∗[*symmetric*]]
by (*metis* (*no-types*, *lifting*) ∗ *dual-order.trans eq-fst-iff fix-clock*(*1*) *fix-clock.simps*)
subgoal
by (*metis* ∗ *fix-clock*(*1*) *fix-clock.simps fst-conv prod.exhaust-sel*)
done
qed (*auto split*: *prod.splits result.splits option.splits match-result.splits*)

lemma *fun-evaluate-clock*:
fun-evaluate-match s1 env v p v′ = (*s2*, *r*) ⟹ *clock s2* ≤ *clock s1*
fun-evaluate s1 env e = (*s2*, *r*) ⟹ *clock s2* ≤ *clock s1*
using *fun-evaluate-clock0* **by** (*metis fst-conv*)+

end

lemma *fix-clock-evaluate*[*simp*]:
fix-clock s1 (*fun-evaluate s1 env e*) = *fun-evaluate s1 env e*
unfolding *fix-clock-alt-def*
using *fun-evaluate-clock* **by** (*fastforce split*: *prod.splits*)

declare *fun-evaluate.simps*[*simp del*]
declare *fun-evaluate-match.simps*[*simp del*]

lemmas *fun-evaluate-simps*[*simp*] =
  fun-evaluate.simps[*unfolded fix-clock-evaluate*]
  fun-evaluate-match.simps[*unfolded fix-clock-evaluate*]

lemmas *fun-evaluate-induct* =
  fun-evaluate-match-fun-evaluate.induct[*unfolded fix-clock-evaluate*]

lemma *fun-evaluate-length*:
  fun-evaluate-match s env v pes err-v = (*s'*, *res*) ⟹ (*case res of Rval vs* ⇒ *length vs* = *1* | - ⇒ *True*)
  fun-evaluate s env es = (*s'*, *res*) ⟹ (*case res of Rval vs* ⇒ *length vs* = *length es* | - ⇒ *True*)
proof (*induction arbitrary*: *s' res* **and** *s' res rule*: *fun-evaluate-match-fun-evaluate.induct*)
  case (*9 st env op1 es*)
  then show *?case*
    supply *do-app.simps*[*simp del*]
    apply (*fastforce*
      split: *if-splits prod.splits result.splits option.splits exp-or-val.splits match-result.splits error-result.splits*
        simp: *list-result-alt-def*)
    done
qed (*fastforce*
      split: *if-splits prod.splits result.splits option.splits exp-or-val.splits*
          match-result.splits error-result.splits)+

lemma *fun-evaluate-matchE*:
  assumes *fun-evaluate-match s env v pes err-v* = (*s'*, *Rval vs*)
  obtains *v* **where** *vs* = [*v*]
using *fun-evaluate-length*(*1*)[*OF assms*]
by (*cases vs*) *auto*

end

21.3 Simplifiying the definition: no mutual recursion

theory *Evaluate-Single*
imports *Evaluate-Clock*
begin

fun *evaluate-list* ::
  (*'ffi state* ⇒ *exp* ⇒ *'ffi state*∗(*v*, *v*) *result*) ⇒
    'ffi state ⇒ *exp list* ⇒ *'ffi state*∗(*v list*, *v*) *result* **where**

Nil:

evaluate-list eval s [] = (*s*, *Rval* []) |

Cons:
evaluate-list eval s (*e*#*es*) =
  (*case fix-clock s* (*eval s e*) *of*
    (*s'*, *Rval v*) ⇒
      (*case evaluate-list eval s' es of*
        (*s''*, *Rval vs*) ⇒ (*s''*, *Rval* (*v*#*vs*))
      | *res* ⇒ *res*)
  | (*s'*, *Rerr err*) ⇒ (*s'*, *Rerr err*))

lemma *evaluate-list-cong*[*fundef-cong*]:
  assumes ⋀*e s*. *e* ∈ *set es1* ⟹ *clock s* ≤ *clock s1* ⟹ *eval1 s e* = *eval2 s e s1* = *s2 es1* = *es2*
  shows *evaluate-list eval1 s1 es1* = *evaluate-list eval2 s2 es2*
using *assms* **by** (*induction es1 arbitrary*: *es2 s1 s2*) (*fastforce simp*: *fix-clock-alt-def split*: *prod.splits result.splits*)+

function (*sequential*)
evaluate :: *v sem-env* ⇒'*ffi state* ⇒ *exp* ⇒ '*ffi state*∗(*v*,*v*) *result* **where**

Lit:
evaluate env s (*Lit l*) = (*s*, *Rval* (*Litv l*)) |

Raise:
evaluate env s (*Raise e*) =
  (*case evaluate env s e of*
    (*s'*, *Rval v*) ⇒ (*s'*, *Rerr* (*Rraise* (*v*)))
  | *res* ⇒ *res*) |

Handle:
evaluate env s (*Handle e pes*) =
  (*case evaluate env s e of*
    (*s'*, *Rerr* (*Rraise v*)) ⇒
      (*case match-result env s' v pes v of*
        (*Rval* (*e'*, *env'*)) ⇒
          evaluate (*env* ⦇ *sem-env.v* := *nsAppend* (*alist-to-ns env'*) (*sem-env.v env*) ⦈) *s' e'*
      | (*Rerr err*) ⇒ (*s'*, *Rerr err*))
  | *res* ⇒ *res*) |

Con:
evaluate env s (*Con cn es*) =
  (*if do-con-check* (*c env*) *cn* (*length es*) *then*
    (*case evaluate-list* (*evaluate env*) *s* (*rev es*) *of*
      (*s'*, *Rval vs*) ⇒
        (*case build-conv* (*c env*) *cn* (*rev vs*) *of*
          Some v ⇒ (*s'*, *Rval v*)
        | *None* ⇒ (*s'*, *Rerr* (*Rabort Rtype-error*)))

```
    | (s′, Rerr err) ⇒ (s′, Rerr err))
  else (s, Rerr (Rabort Rtype-error))) |

Var:
evaluate env s (Var n) =
  (case nsLookup (sem-env.v env) n of
    Some v ⇒ (s, Rval v)
  | None ⇒ (s, Rerr (Rabort Rtype-error))) |

Fun:
evaluate env s (Fun n e) = (s, Rval (Closure env n e)) |

App:
evaluate env s (App op0 es) =
  (case evaluate-list (evaluate env) s (rev es) of
    (s′, Rval vs) ⇒
      (if op0 = Opapp then
        (case do-opapp (rev vs) of
          Some (env′, e) ⇒
            (if (clock s′ = 0) then
              (s′, Rerr (Rabort Rtimeout-error))
            else
              evaluate env′ (dec-clock s′) e)
        | None ⇒ (s′, Rerr (Rabort Rtype-error)))
      else
        (case do-app (refs s′, ffi s′) op0 (rev vs) of
          Some ((refs′,ffi′), res) ⇒ (s′ ⦇refs:=refs′,ffi:=ffi′⦈, res)
        | None ⇒ (s′, Rerr (Rabort Rtype-error))))
  | (s′, Rerr err) ⇒ (s′, Rerr err)) |

Log:
evaluate env s (Log op0 e1 e2) =
  (case evaluate env s e1 of
    (s′, Rval v) ⇒
      (case do-log op0 v e2 of
        Some (Exp e′) ⇒ evaluate env s′ e′
      | Some (Val bv) ⇒ (s′, Rval bv)
      | None ⇒ (s′, Rerr (Rabort Rtype-error)))
  | res ⇒ res) |

If:
evaluate env s (If e1 e2 e3) =
  (case evaluate env s e1 of
    (s′, Rval v) ⇒
      (case do-if v e2 e3 of
        Some e′ ⇒ evaluate env s′ e′
      | None ⇒ (s′, Rerr (Rabort Rtype-error)))
  | res ⇒ res) |
```

Mat:
evaluate env s (*Mat e pes*) =
  (*case evaluate env s e of*
    (*s'*, *Rval v*) ⇒
      (*case match-result env s' v pes Bindv of*
        Rval (*e'*, *env'*) ⇒
          evaluate (*env* ⦇ *sem-env.v* := *nsAppend* (*alist-to-ns env'*) (*sem-env.v env*)
⦈) *s' e'*
      | *Rerr err* ⇒ (*s'*, *Rerr err*))
  | *res* ⇒ *res*) |

Let:
evaluate env s (*Let n e1 e2*) =
  (*case evaluate env s e1 of*
    (*s'*, *Rval v*) ⇒
      evaluate (*env* ⦇ *sem-env.v* := (*nsOptBind n v*(*sem-env.v env*)) ⦈) *s' e2*
  | *res* ⇒ *res*) |

Letrec:
evaluate env s (*Letrec funs e*) =
  (*if distinct* (*List.map* (λ*x*. (*case x of* (*x*,*y*,*z*) => *x*)) *funs*) *then*
    evaluate (*env* ⦇ *sem-env.v* := (*build-rec-env funs env*(*sem-env.v env*)) ⦈) *s e*
  else
    (*s*, *Rerr* (*Rabort Rtype-error*))) |

Tannot:
evaluate env s (*Tannot e t0*) = *evaluate env s e* |

Lannot:
evaluate env s (*Lannot e l*) = *evaluate env s e*
  by *pat-completeness auto*

context
  notes *do-app.simps*[*simp del*]
begin

lemma *match-result-elem*:
  assumes *match-result env s v0 pes err-v* = *Rval* (*e*, *env'*)
  shows ∃*pat*. (*pat*, *e*) ∈ *set pes*
  using *assms* **proof** (*induction pes*)
  case *Nil*
  then show *?case* **by** *auto*
next
  case (*Cons pe pes*)
  then obtain *p e* **where** *pe* = (*p*, *e*) **by** *force*
  show *?case*
    using *Cons*(*2*)
    apply (*simp add*:*match-result-alt-def*)
    unfolding ‹*pe* = -›

apply (*cases allDistinct* (*pat-bindings p* []))
apply (*cases pmatch* (*c env*) (*refs s*) *p v0* [])
using *Cons*(*1*) **by** *auto*+
qed

private lemma *evaluate-list-clock-monotone*: *clock* (*fst* (*evaluate-list eval s es*)) ≤ *clock s*
apply (*induction es arbitrary*: *s*)
apply (*auto split*:*prod.splits result.splits simp add*:*fix-clock-alt-def dest*!:*fstI intro*:*le-trans*)
apply (*metis state.record-simps*(*1*))+
done

lemma *i-hate-words-helper*:
$i \leq (j - k :: nat) \Longrightarrow i \leq j$
by *simp*

thm *i-hate-words-helper* [*THEN le-trans*, *no-vars*]

private lemma *evaluate-clock-monotone*:
‹*clock* (*fst* (*evaluate env s e*)) ≤ *clock s*›
if ‹*evaluate-dom* (*env*, *s*, *e*)›
proof −
have ∗: ‹$i \leq j - k \Longrightarrow j \leq r \Longrightarrow i \leq r$› **for** *i j k r* :: *nat*
by *arith*
from *that* **show** *?thesis*
by *induction* (*fastforce simp add*: *evaluate.psimps do-con-check-build-conv evaluate-list-clock-monotone*
split: *prod.splits result.splits option.splits exp-or-val.splits error-result.splits*
dest: *fstI intro*: ∗)+
qed

private definition *fun-evaluate-single-relation* **where**
fun-evaluate-single-relation = *inv-image* (*less-than* <∗*lex*∗> *less-than*) (λ*x*.
case x of (-, *s*, *e*) ⇒ (*clock s*, *size-exp′ e*))

private lemma *pat-elem-less-size*:
(*pat*, *e*) ∈ *set pes* ⟹ *size-exp′ e* < (*size-list* (*size-prod size size-exp′*) *pes*)
by (*induction pes*) *auto*

private lemma *elem-less-size*: *e* ∈ *set es* ⟹ *size-exp′ e* ≤ *size-list size-exp′ es*
by (*induction es*) *auto*

lemma *evaluate-total*: *All evaluate-dom*
proof (*relation fun-evaluate-single-relation*, *unfold fun-evaluate-single-relation-def*, *goal-cases*)
case *7*
then show *?case*
using *evaluate-list-clock-monotone 7*(*1*)[*symmetric*]

by (*auto dest*!: *fstI simp add*:*evaluate-list-clock-monotone Suc-le-lessD*)
qed (*auto simp add*: *less-Suc-eq-le Suc-le-lessD do-if-def do-log-alt-def evaluate-list-clock-monotone elem-less-size*
split:*lop.splits v.splits option.splits tid-or-exn.splits if-splits id0.splits list.splits*
dest!:*evaluate-clock-monotone match-result-elem fstI dest*:*sym pat-elem-less-size intro*:*le-neq-implies-less*)

termination *evaluate* **by** (*rule evaluate-total*)

lemma *evaluate-clock-monotone'*: *evaluate eval s e* = (*s'*, *r*) $\Longrightarrow$ *clock s'* $\leq$ *clock s*
using *fst-conv evaluate-clock-monotone evaluate-total*
by *metis*

fun *evaluate-list'* :: *v sem-env* ⇒ *'ffi state* ⇒ *exp list* ⇒ *'ffi state*∗(*v list*, *v*) *result*
where
evaluate-list' env s [] = (*s*, *Rval* []) |
evaluate-list' env s (*e*#*es*) =
(*case evaluate env s e of*
(*s'*, *Rval v*) ⇒
(*case evaluate-list' env s' es of*
(*s''*, *Rval vs*) ⇒ (*s''*, *Rval* (*v*#*vs*))
| *res* ⇒ *res*)
| (*s'*, *Rerr err*) ⇒ (*s'*, *Rerr err*))

lemma *fix-clock-evaluate*[*simp*]: *fix-clock s* (*evaluate eval s e*) = *evaluate eval s e*
unfolding *fix-clock-alt-def*
apply (*auto simp*: *datatype-record-update split*: *state.splits prod.splits*)
using *evaluate-clock-monotone'* **by** *fastforce*

lemma *evaluate-list-eq*[*simp*]: *evaluate-list* (*evaluate env*) = *evaluate-list' env*
apply (*rule ext*)+
subgoal for *s es*
by (*induction rule*:*evaluate-list'.induct*) (*auto split*:*prod.splits result.splits*)
done

declare *evaluate-list.simps*[*simp del*]

lemma *fun-evaluate-equiv*:
fun-evaluate-match s env v pes err-v = (*case match-result env s v pes err-v of*
Rerr err ⇒ (*s*, *Rerr err*)
| *Rval* (*e*, *env'*) ⇒ *evaluate-list* (*evaluate* (*env* ⦇ *sem-env.v* := (*nsAppend* (*alist-to-ns env'*) (*sem-env.v env*)) ⦈)) *s* [*e*])
fun-evaluate s env es = *evaluate-list* (*evaluate env*) *s es*
by (*induction rule*: *fun-evaluate-induct*)
(*auto split*: *prod.splits result.splits match-result.splits option.splits exp-or-val.splits*
if-splits match-result.splits error-result.splits
simp: *all-distinct-alt-def*)

corollary *fun-evaluate-equiv′*:
  evaluate env s e = *map-prod id* (*map-result hd id*) (*fun-evaluate s env* [*e*])
by (*subst fun-evaluate-equiv*) (*simp split*: *prod.splits result.splits add*: *error-result.map-id*)

end

end

Chapter 22

Relational big-step semantics

22.1 Determinism

theory *Big-Step-Determ*
imports *Semantic-Extras*
begin

lemma *evaluate-determ*:
  evaluate-match ck env s v pes v′ r1a ⟹ *evaluate-match ck env s v pes v′ r1b* ⟹
r1a = *r1b*
  evaluate-list ck env s es r2a ⟹ *evaluate-list ck env s es r2b* ⟹ *r2a* = *r2b*
  evaluate ck env s e r3a ⟹ *evaluate ck env s e r3b* ⟹ *r3a* = *r3b*
proof (*induction arbitrary*: *r1b* **and** *r2b* **and** *r3b rule*: *evaluate-match-evaluate-list-evaluate.inducts*)
  case (*raise1 ck s1 e env s2 v1*)
  then show *?case*
    by − (*ind-cases evaluate ck env s1* (*Raise e*) *r3b*, *auto*)
next
  case (*raise2 ck s1 e env s2 err*)
  then show *?case*
    by − (*ind-cases evaluate ck env s1* (*Raise e*) *r3b*, *auto*)
next
  case (*handle1 ck env s2 s1 e v1 pes*)
  then show *?case*
    by − (*ind-cases evaluate ck env s1* (*Handle e pes*) *r3b*, *auto*)
next
  case (*handle2 ck s1 s2 env e pes v1 bv*)
  then show *?case*
    by − (*ind-cases evaluate ck env s1* (*Handle e pes*) *r3b*; *fastforce*)
next
  case (*handle3 ck s1 s2 env e pes a*)
  then show *?case*
    by − (*ind-cases evaluate ck env s1* (*Handle e pes*) *r3b*; *auto*)
next
  case (*con1 ck env cn es vs s s′ v1*)
  then show *?case*

by − (*ind-cases evaluate ck env s* (*Con cn es*) *r3b*; *fastforce*)
next
case (*con2 ck env cn es s*)
then show *?case*
by − (*ind-cases evaluate ck env s* (*Con cn es*) *r3b*, *auto*)
next
case (*con3 ck env cn es err s s'*)
then show *?case*
by − (*ind-cases evaluate ck env s* (*Con cn es*) *r3b*, *auto*)
next
case (*app1 ck env es vs env' e bv s1 s2*)
then show *?case*
by − (*ind-cases evaluate ck env s1* (*App Opapp es*) *r3b*; *fastforce*)
next
case (*app2 ck env es vs env' e s1 s2*)
then show *?case*
by − (*ind-cases evaluate ck env s1* (*App Opapp es*) *r3b*; *force*)
next
case (*app3 ck env es vs s1 s2*)
then show *?case*
by − (*ind-cases evaluate ck env s1* (*App Opapp es*) *r3b*; *force*)
next
case (*app4 ck env op0 es vs res s1 s2 refs' ffi'*)
then show *?case*
by − (*ind-cases evaluate ck env s1* (*App op0 es*) *r3b*; *fastforce*)
next
case (*app5 ck env op0 es vs s1 s2*)
then show *?case*
by − (*ind-cases evaluate ck env s1* (*App op0 es*) *r3b*; *force*)
next
case (*app6 ck env op0 es err s1 s2*)
then show *?case*
by − (*ind-cases evaluate ck env s1* (*App op0 es*) *r3b*; *force*)
next
case (*log1 ck env op0 e1 e2 v1 e' bv s1 s2*)
then show *?case*
by − (*ind-cases evaluate ck env s1* (*Log op0 e1 e2*) *r3b*; *fastforce*)
next
case (*log2 ck env op0 e1 e2 v1 bv s1 s2*)
then show *?case*
by − (*ind-cases evaluate ck env s1* (*Log op0 e1 e2*) *r3b*; *force*)
next
case (*log3 ck env op0 e1 e2 v1 s1 s2*)
then show *?case*
by − (*ind-cases evaluate ck env s1* (*Log op0 e1 e2*) *r3b*; *force*)
next
case (*log4 ck env op0 e1 e2 err s s'*)
then show *?case*
by − (*ind-cases evaluate ck env s* (*Log op0 e1 e2*) *r3b*; *auto*)

next
  case (*if1 ck env e1 e2 e3 v1 e′ bv s1 s2*)
  then show *?case*
    by − (*ind-cases evaluate ck env s1* (*If e1 e2 e3*) *r3b*; *fastforce*)
next
  case (*if2 ck env e1 e2 e3 v1 s1 s2*)
  then show *?case*
    by − (*ind-cases evaluate ck env s1* (*If e1 e2 e3*) *r3b*; *force*)
next
  case (*if3 ck env e1 e2 e3 err s s′*)
  then show *?case*
    by − (*ind-cases evaluate ck env s* (*If e1 e2 e3*) *r3b*, *auto*)
next
  case (*mat1 ck env e pes v1 bv s1 s2*)
  then show *?case*
    by − (*ind-cases evaluate ck env s1* (*Mat e pes*) *r3b*; *fastforce*)
next
  case (*mat2 ck env e pes err s s′*)
  then show *?case*
    by − (*ind-cases evaluate ck env s* (*Mat e pes*) *r3b*, *auto*)
next
  case (*let1 ck env n e1 e2 v1 bv s1 s2*)
  then show *?case*
    by − (*ind-cases evaluate ck env s1* (*Let n e1 e2*) *r3b*; *fastforce*)
next
  case (*let2 ck env n e1 e2 err s s′*)
  then show *?case*
    by − (*ind-cases evaluate ck env s* (*Let n e1 e2*) *r3b*, *auto*)
next
  case (*letrec1 ck env funs e bv s*)
  then show *?case*
    by − (*ind-cases evaluate ck env s* (*Letrec funs e*) *r3b*; *fastforce*)
next
  case (*letrec2 ck env funs e s*)
  then show *?case*
    by − (*ind-cases evaluate ck env s* (*Letrec funs e*) *r3b*, *auto*)
next
  case (*tannot ck env e t0 s bv*)
  then show *?case*
    by − (*ind-cases evaluate ck env s* (*Tannot e t0*) *r3b*, *auto*)
next
  case (*locannot ck env e l s bv*)
  then show *?case*
    by − (*ind-cases evaluate ck env s* (*Lannot e l*) *r3b*, *auto*)
next
  case (*cons1 ck env e es v1 vs s1 s2 s3*)
  then show *?case*
    by − (*ind-cases evaluate-list ck env s1* (*e # es*) *r2b*, *auto*)
next

case (*cons2 ck env e es err s s'*)
then show *?case*
by – (*ind-cases evaluate-list ck env s* (*e* # *es*) *r2b*, *auto*)
next
case (*cons3 ck env e es v1 err s1 s2 s3*)
then show *?case*
by – (*ind-cases evaluate-list ck env s1* (*e* # *es*) *r2b*, *auto*)
next
case (*mat-cons1 ck env env' v1 p pes e bv err-v s*)
then show *?case*
by – (*ind-cases evaluate-match ck env s v1* ((*p*, *e*) # *pes*) *err-v r1b*; *fastforce*)
next
case (*mat-cons2 ck env v1 p e pes bv s err-v*)
then show *?case*
by – (*ind-cases evaluate-match ck env s v1* ((*p*, *e*) # *pes*) *err-v r1b*; *fastforce*)
next
case (*mat-cons3 ck env v1 p e pes s err-v*)
then show *?case*
by – (*ind-cases evaluate-match ck env s v1* ((*p*, *e*) # *pes*) *err-v r1b*, *auto*)
next
case (*mat-cons4 ck env v1 p e pes s err-v*)
then show *?case*
by – (*ind-cases evaluate-match ck env s v1* ((*p*, *e*) # *pes*) *err-v r1b*, *auto*)
qed (*auto elim*: *evaluate.cases evaluate-list.cases evaluate-match.cases*)

end

22.2 Totality

theory *Big-Step-Total*
imports *Semantic-Extras*
begin

context begin

private lemma *evaluate-list-total0*:
fixes s :: *'a state*
assumes $\bigwedge e\ env\ s'{::}'a\ state.\ e \in set\ es \Longrightarrow clock\ s' \leq clock\ s \Longrightarrow \exists\, s''\ r.\ evaluate\ True\ env\ s'\ e\ (s'', r)$
shows $\exists\, s'\ r.\ evaluate\text{-}list\ True\ env\ s\ es\ (s', r)$
using *assms* **proof** (*induction es arbitrary*: *env s*)
case *Nil*
show *?case* **by** (*metis evaluate-match-evaluate-list-evaluate.empty*)
next
case (*Cons e es*)
then obtain *s' r* **where** *e*: *evaluate True env s e* (*s'*, *r*)
by *fastforce*
then have *clock*: $clock\ s' \leq clock\ s$
by (*metis evaluate-clock-mono*)

```
  show ?case
    proof (cases r)
      case (Rval v)

      have ∃ s'' r. evaluate-list True env s' es (s'', r)
        using Cons clock by auto
      then obtain s'' r where evaluate-list True env s' es (s'', r)
        by auto

      with e Rval show ?thesis
        by (cases r)
        (metis evaluate-match-evaluate-list-evaluate.cons1 evaluate-match-evaluate-list-evaluate.cons3)+
    next
      case Rerr
      with e show ?thesis by (metis evaluate-match-evaluate-list-evaluate.cons2)
    qed
qed

private lemma evaluate-match-total0:
  fixes s :: 'a state
  assumes ⋀p e env s'::'a state. (p, e) ∈ set pes ⟹ clock s' ≤ clock s ⟹ ∃ s'' r.
evaluate True env s' e (s'', r)
  shows ∃ s' r. evaluate-match True env s v pes v' (s', r)
using assms proof (induction pes arbitrary: env s)
  case Nil
  show ?case by (metis mat-empty)
next
  case (Cons pe pes)
  then obtain p e where pe = (p, e) by force

  show ?case
    proof (cases allDistinct (pat-bindings p []))
      case distinct: True
      show ?thesis
        proof (cases pmatch (c env) (refs s) p v [])
          case No-match

          have ∃ s' r. evaluate-match True env s v pes v' (s', r)
            apply (rule Cons)
            apply (rule Cons)
            by auto
          then obtain s' r where evaluate-match True env s v pes v' (s', r)
            by auto

          show ?thesis
            unfolding ‹pe = -›
            apply (intro exI)
            apply (rule mat-cons2)
```

```
          apply safe
          by fact+
      next
        case Match-type-error
        then show ?thesis
          unfolding ‹pe = -› by (metis mat-cons3)
      next
        case (Match env')

          have ∃ s' r. evaluate True (env ⦇ sem-env.v := (nsAppend (alist-to-ns
env') (sem-env.v env)) ⦈) s e (s', r)
          apply (rule Cons)
          unfolding ‹pe = -› by auto
         then obtain s' r where evaluate True (env ⦇ sem-env.v := (nsAppend
(alist-to-ns env') (sem-env.v env)) ⦈) s e (s', r)
          by auto

        show ?thesis
          unfolding ‹pe = -›
          apply (intro exI)
          apply (rule mat-cons1)
          apply safe
          apply fact+
          done
      qed
  next
    case False
    then show ?thesis
      unfolding ‹pe = -› by (metis mat-cons4)
  qed
qed

lemma evaluate-total: ∃ s' r. evaluate True env s e (s', r)
proof −
  have wf (less-than <*lex*> measure (size::exp ⇒ nat))
    by auto
  then show ?thesis
    proof (induction (clock s, e) arbitrary: env s e)
      case less
      show ?case
        proof (cases e)
          case (Raise e')
          then have ∃ s' r. evaluate True env s e' (s', r)
            using less by auto
          then obtain s' r where evaluate True env s e' (s', r)
            by auto
          then show ?thesis
            unfolding Raise by (cases r) (metis raise1 raise2)+
      next
```

case (*Con cn es*)
show *?thesis*
proof (*cases do-con-check* (*c env*) *cn* (*length es*))
case *True*
have $\exists\, s'$ *vs. evaluate-list True env s* (*rev es*) (s', *vs*)
apply (*rule evaluate-list-total0*)
apply (*rule less*)
unfolding *Con*
apply *auto*
using *Con* **apply** (*auto simp*: *less-eq-Suc-le*)
apply (*rule size-list-estimation'*)
apply *assumption*
by *simp*
then obtain *r* s' **where** *es*: *evaluate-list True env s* (*rev es*) (s', *r*)
by *auto*

show *?thesis*
proof (*cases r*)
case (*Rval vs*)
moreover obtain *v* **where** *build-conv* (*c env*) *cn* (*rev vs*) = *Some v*
using *True*
by (*cases cn*) (*auto split*: *option.splits*)
ultimately show *?thesis*
using *True es* **unfolding** *Con* **by** (*metis con1*)
next
case *Rerr*
with *True es* **show** *?thesis* **unfolding** *Con* **by** (*metis con3*)
qed
next
case *False*
with *Con* **show** *?thesis* **by** (*metis con2*)
qed
next
case (*Var n*)
then show *?thesis*
by (*cases nsLookup* (*sem-env.v env*) *n*) (*metis var1 var2*)+
next
case (*App op es*)
have $\exists\, s'$ *vs. evaluate-list True env s* (*rev es*) (s', *vs*)
apply (*rule evaluate-list-total0*)
apply (*rule less*)
unfolding *App* **apply** (*auto simp*: *less-eq-Suc-le*)
apply (*rule size-list-estimation'*)
apply *assumption*
by *simp*
then obtain *r s2* **where** *es*: *evaluate-list True env s* (*rev es*) (*s2*, *r*)
by *auto*
then have *clock*: *clock s2* $\leq$ *clock s*
by (*metis evaluate-clock-mono*)

```
show ?thesis
  proof (cases r)
    case (Rval vs)
    show ?thesis
      proof (cases op = Opapp)
        case opapp: True
        show ?thesis
          proof (cases do-opapp (rev vs))
            case None
            with App opapp Rval es show ?thesis by (metis app3)
          next
            case (Some r)
            obtain env' e' where r = (env', e')
              by (metis surj-pair)

            show ?thesis
              proof (cases clock s2 = 0)
                case True
                show ?thesis
                  unfolding ‹op = -› App
                  apply (intro exI)
                  apply (rule app2)
                  apply (intro conjI)
                  using es unfolding Rval apply assumption
                  using Some unfolding ‹r = -› apply assumption
                  apply fact ..
              next
                case False

                have ∃s' r. evaluate True env' (s2 (| clock := clock s2 − Suc
0 |)) e' (s', r)
                  apply (rule less)
                    using False clock by (auto simp: datatype-record-update
split: state.splits)
                then obtain s' r' where evaluate True env' (s2 (| clock :=
clock s2 − Suc 0 |)) e' (s', r')
                  by auto

                show ?thesis
                  unfolding ‹op = -› App
                  apply (intro exI)
                  apply (rule app1)
                  apply (intro conjI)
                  using es unfolding Rval apply assumption
                  using Some unfolding ‹r = -› apply assumption
                  using False apply metis
                  apply simp
                  apply fact
```

```
                done
              qed
            qed
          next
            case False
            show ?thesis
              proof (cases do-app ((refs  s2),(ffi  s2)) op (rev vs))
                case None
                show ?thesis
                  unfolding App
                  apply (intro exI)
                  apply (rule app5)
                  apply (intro conjI)
                  using es unfolding Rval apply assumption
                  by fact+
              next
                case (Some r)
                obtain refs′ ffi′ res where r = ((refs′, ffi′), res)
                  by (metis surj-pair)

                show ?thesis
                  unfolding App
                  apply (intro exI)
                  apply (rule app4)
                  apply (intro conjI)
                  using es unfolding Rval apply assumption
                  using Some unfolding ‹r = -› apply assumption
                  by fact
              qed
          qed
      next
        case Rerr
        with es App show ?thesis by (metis app6)
      qed
  next
    case (Log op e1 e2)
    with less have ∃ s′ r. evaluate True env s e1 (s′, r) by simp
    then obtain s′ r where e1: evaluate True env s e1 (s′, r)
      by blast
    then have clock: clock s′ ≤ clock s
      by (metis evaluate-clock-mono)

    show ?thesis
      proof (cases r)
        case (Rval v)
        with e1 Log show ?thesis
          proof (cases op v e2 rule: do-log-cases)
            case none
            then show ?thesis
```

```
          unfolding Log
          using e1 Rval by (metis log3)
      next
        case val
        then show ?thesis
          unfolding Log
          using e1 Rval by (metis log2)
      next
        case exp
        have ∃s'' r. evaluate True env s' e2 (s'', r)
          apply (rule less)
          using clock Log by auto
        then obtain s'' r where evaluate True env s' e2 (s'', r)
          by auto
        show ?thesis
          unfolding Log
          apply (intro exI)
          apply (rule log1)
          apply (intro conjI)
          using Rval e1 apply force
          by fact+
      qed
  next
    case Rerr
    with e1 show ?thesis
      unfolding Log by (metis log4)
  qed
next
  case (If e1 e2 e3)
  with less have ∃s' r. evaluate True env s e1 (s', r) by simp
  then obtain s' r where e1: evaluate True env s e1 (s', r) by auto
  then have clock: clock s' ≤ clock s
    by (metis evaluate-clock-mono)

  show ?thesis
    proof (cases r)
      case (Rval v1)
      show ?thesis
        proof (cases v1 e2 e3 rule: do-if-cases)
          case none
          show ?thesis
            unfolding If
            apply (intro exI)
            apply (rule if2)
            apply (intro conjI)
            using Rval e1 apply force
            by fact
        next
          case true
```

```
          have ∃s'' r. evaluate True env s' e2 (s'', r)
            apply (rule less)
            using clock If by auto
          then obtain s'' r where evaluate True env s' e2 (s'', r)
            by auto
          show ?thesis
            unfolding If
            apply (intro exI)
            apply (rule if1)
            apply (intro conjI)
            using Rval e1 apply force
            by fact+
        next
          case false
          have ∃s'' r. evaluate True env s' e3 (s'', r)
            apply (rule less)
            using clock If by auto
          then obtain s'' r where evaluate True env s' e3 (s'', r)
            by auto
          show ?thesis
            unfolding If
            apply (intro exI)
            apply (rule if1)
            apply (intro conjI)
            using Rval e1 apply force
            by fact+
        qed
    next
      case Rerr
      with e1 show ?thesis unfolding If by (metis if3)
    qed
next
  case (Handle e' pes)
  with less have ∃s' r. evaluate True env s e' (s', r) by simp
  then obtain s' r where e': evaluate True env s e' (s', r) by auto
  then have clock: clock s' ≤ clock s
    by (metis evaluate-clock-mono)

  show ?thesis
    proof (cases r)
      case Rval
      with e' show ?thesis
        unfolding Handle by (metis handle1)
    next
      case (Rerr err)
      show ?thesis
        proof (cases err)
          case (Rraise exn)
```

have $\exists s''\ r.$ *evaluate-match True env s′ exn pes exn* (*s″*, *r*)
apply (*rule evaluate-match-total0*)
apply (*rule less*)
using *Handle clock* **apply** (*auto simp*: *less-eq-Suc-le*)
apply (*rule trans-le-add1*)
apply (*rule size-list-estimation′*)
apply *assumption*
by *auto*
then obtain *s″ r* **where** *evaluate-match True env s′ exn pes exn* (*s″*, *r*)
by *auto*

show *?thesis*
unfolding *Handle*
apply (*intro exI*)
apply (*rule handle2*)
apply *safe*
using *e′* **unfolding** *Rerr Rraise* **apply** *assumption*
by *fact*
next
case (*Rabort x2*)
with *e′ Rerr* **show** *?thesis*
unfolding *Handle*
by (*metis handle3*)
qed
qed
next
case (*Mat e′ pes*)
with *less* **have** $\exists s'\ r.$ *evaluate True env s e′* (*s′*, *r*) **by** *simp*
then obtain *s′ r* **where** *e′*: *evaluate True env s e′* (*s′*, *r*) **by** *auto*
then have *clock*: *clock s′* $\leq$ *clock s*
by (*metis evaluate-clock-mono*)

show *?thesis*
proof (*cases r*)
case (*Rval v*)

have $\exists s''\ r.$ *evaluate-match True env s′ v pes* (*Conv* (*Some* (*″Bind″*, *TypeExn* (*Short ″Bind″*))) []) (*s″*, *r*)
apply (*rule evaluate-match-total0*)
apply (*rule less*)
unfolding *Mat* **using** *clock* **apply** (*auto simp*: *less-eq-Suc-le*)
apply (*rule trans-le-add1*)
apply (*rule size-list-estimation′*)
apply *assumption*
by *auto*
then obtain *s″ r* **where** *evaluate-match True env s′ v pes* (*Conv* (*Some* (*″Bind″*, *TypeExn* (*Short ″Bind″*))) []) (*s″*, *r*)
by *auto*

show *?thesis*
unfolding *Mat*
apply (*intro exI*)
apply (*rule mat1*)
apply *safe*
using *e'* **unfolding** *Rval*
apply *assumption*
apply *fact*
done
next
case *Rerr*
with *e'* **show** *?thesis*
unfolding *Mat*
by (*metis mat2*)
qed
next
case (*Let n e1 e2*)
then have $\exists\, s'\ r.$ *evaluate True env s e1* (s', r)
using *less* **by** *auto*
then obtain $s'\ r$ **where** *e1*: *evaluate True env s e1* (s', r)
by *auto*
then have *clock*: *clock* $s' \leq$ *clock s*
by (*metis evaluate-clock-mono*)
show *?thesis*
proof (*cases r*)
case (*Rval v*)
have $\exists\, s''\ r.$ *evaluate True* (*env* ⦇ *sem-env.v* := *nsOptBind n v* (*sem-env.v env*) ⦈) s' *e2* (s'', r)
apply (*rule less*)
using *Let clock* **by** *auto*
then show *?thesis*
unfolding *Let*
using *e1 Rval* **by** (*metis let1*)
next
case *Rerr*
with *e1* **show** *?thesis*
unfolding *Let*
by (*metis let2*)
qed
next
case (*Letrec funs e'*)
then have $\exists\, s'\ r.$ *evaluate True* (*env* ⦇ *sem-env.v* := *build-rec-env funs env* (*sem-env.v env*) ⦈) *s e'* (s', r)
using *less* **by** *auto*
then show *?thesis*
unfolding *Letrec*
by (*cases allDistinct* (*map* ($\lambda x.$ *case x of* $(x, y, z) \Rightarrow x$) *funs*))
(*metis letrec1 letrec2*)+

```
      next
        case (Tannot e′)
        with less have ∃ s′ r. evaluate True env s e′ (s′, r) by simp
        then show ?thesis
          unfolding ‹e = -›
          by (fastforce intro: evaluate-match-evaluate-list-evaluate.intros)
      next
        case (Lannot e′)
        with less have ∃ s′ r. evaluate True env s e′ (s′, r) by simp
        then show ?thesis
          unfolding ‹e = -›
          by (fastforce intro: evaluate-match-evaluate-list-evaluate.intros)
      qed (fastforce intro: evaluate-match-evaluate-list-evaluate.intros)+
    qed
qed

end
```

The following are pretty much the same proofs as above, but without additional assumptions; instead using *evaluate-total* directly.

```
lemma evaluate-list-total: ∃ s′ r. evaluate-list True env s es (s′, r)
proof (induction es arbitrary: env s)
  case Nil
  show ?case by (metis evaluate-match-evaluate-list-evaluate.empty)
next
  case (Cons e es)
  obtain s′ r where e: evaluate True env s e (s′, r)
    by (metis evaluate-total)
  show ?case
    proof (cases r)
      case (Rval v)
      have ∃ s′′ r. evaluate-list True env s′ es (s′′, r)
        using Cons by auto
      then obtain s′′ r where evaluate-list True env s′ es (s′′, r)
        by auto

      with e Rval show ?thesis
        by (cases r)
        (metis evaluate-match-evaluate-list-evaluate.cons1 evaluate-match-evaluate-list-evaluate.cons3)+
    next
      case Rerr
      with e show ?thesis
        by (metis evaluate-match-evaluate-list-evaluate.cons2)
    qed
qed

lemma evaluate-match-total: ∃ s′ r. evaluate-match True env s v pes v′ (s′, r)
proof (induction pes arbitrary: env s)
  case Nil
```

```
  show ?case by (metis mat-empty)
next
  case (Cons pe pes)
  then obtain p e where pe = (p, e) by force

  show ?case
    proof (cases allDistinct (pat-bindings p []))
      case distinct: True
      show ?thesis
        proof (cases pmatch (c env) (refs s) p v [])
          case No-match

          have ∃ s′ r. evaluate-match True env s v pes v′ (s′, r)
            by (rule Cons)
          then obtain s′ r where evaluate-match True env s v pes v′ (s′, r)
            by auto

          show ?thesis
            unfolding ‹pe = -›
            apply (intro exI)
            apply (rule mat-cons2)
            apply safe
            by fact+
        next
          case Match-type-error
          then show ?thesis
            unfolding ‹pe = -› by (metis mat-cons3)
        next
          case (Match env′)

            have ∃ s′ r. evaluate True (env (| sem-env.v := (nsAppend (alist-to-ns
env′) (sem-env.v env)) |)) s e (s′, r)
            by (metis evaluate-total)
           then obtain s′ r where evaluate True (env (| sem-env.v := (nsAppend
(alist-to-ns env′) (sem-env.v env)) |)) s e (s′, r)
            by auto

          show ?thesis
            unfolding ‹pe = -›
            apply (intro exI)
            apply (rule mat-cons1)
            apply safe
            apply fact+
            done
        qed
    next
      case False
      then show ?thesis
        unfolding ‹pe = -› by (metis mat-cons4)
```

qed
qed

end

22.3 Equivalence to the functional semantics

theory *Big-Step-Fun-Equiv*
imports
Big-Step-Determ
Big-Step-Total
Evaluate-Clock
begin

locale *eval* =
fixes
eval :: *v sem-env* ⇒ *exp* ⇒ *'a state* ⇒ *'a state* × (*v*, *v*) *result* **and**
eval-list :: *v sem-env* ⇒ *exp list* ⇒ *'a state* ⇒ *'a state* × (*v list*, *v*) *result* **and**
eval-match :: *v sem-env* ⇒ *v* ⇒ (*pat* × *exp*) *list* ⇒ *v* ⇒ *'a state* ⇒ *'a state* × (*v*, *v*) *result*

assumes
valid-eval: *evaluate True env s e* (*eval env e s*) **and**
valid-eval-list: *evaluate-list True env s es* (*eval-list env es s*) **and**
valid-eval-match: *evaluate-match True env s v pes err-v* (*eval-match env v pes err-v s*)
begin

lemmas *eval-all* = *valid-eval valid-eval-list valid-eval-match*

lemma *evaluate-iff*:
evaluate True env st e r ⟷ (*r* = *eval env e st*)
evaluate-list True env st es r' ⟷ (*r'* = *eval-list env es st*)
evaluate-match True env st v pes v' r ⟷ (*r* = *eval-match env v pes v' st*)
by (*metis eval-all evaluate-determ*)+

lemma *evaluate-iff-sym*:
evaluate True env st e r ⟷ (*eval env e st* = *r*)
evaluate-list True env st es r' ⟷ (*eval-list env es st* = *r'*)
evaluate-match True env st v pes v' r ⟷ (*eval-match env v pes v' st* = *r*)
by (*auto simp*: *evaluate-iff*)

lemma *other-eval-eq*:
assumes *Big-Step-Fun-Equiv.eval eval' eval-list' eval-match'*
shows *eval'* = *eval eval-list'* = *eval-list eval-match'* = *eval-match*
proof −
interpret *other*: *Big-Step-Fun-Equiv.eval eval' eval-list' eval-match'* **by** *fact*

show *eval'* = *eval*

```
    apply (rule ext)+
    using evaluate-iff other.evaluate-iff
    by (metis evaluate-determ)

  show eval-list′ = eval-list
    apply (rule ext)+
    using evaluate-iff other.evaluate-iff
    by (metis evaluate-determ)

  show eval-match′ = eval-match
    apply (rule ext)+
    using evaluate-iff other.evaluate-iff
    by (metis evaluate-determ)
qed

lemma eval-list-singleton:
  eval-list env [e] st = map-prod id list-result (eval env e st)
proof −
  define res where res = eval-list env [e] st
  then have e: evaluate-list True env st [e] res
    by (metis evaluate-iff)
  then obtain st′ r where res = (st′, r)
    by (metis surj-pair)
  then have map-prod id list-result (eval env e st) = (st′, r)
    proof (cases r)
      case (Rval vs)
      with e obtain v where vs = [v] evaluate True env st e (st′, Rval v)
        unfolding ‹res = (st′, r)› by (metis evaluate-list-singleton-valE)
      then have eval env e st = (st′, Rval v)
        by (metis evaluate-iff-sym)
      then show ?thesis
        unfolding ‹r = -› ‹vs = -› by auto
    next
      case (Rerr err)
      with e have evaluate True env st e (st′, Rerr err)
        unfolding ‹res = (st′, r)› by (metis evaluate-list-singleton-errD)
      then have eval env e st = (st′, Rerr err)
        by (metis evaluate-iff-sym)
      then show ?thesis
        unfolding ‹r = -› by (cases err) auto
    qed
  then show ?thesis
    using res-def ‹res = (st′, r)›
    by metis
qed

lemma eval-eqI:
  assumes ⋀r. evaluate True env st1 e1 r ⟷ evaluate True env st2 e2 r
  shows eval env e1 st1 = eval env e2 st2
```

using *assms* **by** (*metis evaluate-iff*)

lemma *eval-match-eqI*:
assumes $\bigwedge r$. *evaluate-match True env1 st1 v1 pes1 err-v1 r* $\longleftrightarrow$ *evaluate-match True env2 st2 v2 pes2 err-v2 r*
shows *eval-match env1 v1 pes1 err-v1 st1* = *eval-match env2 v2 pes2 err-v2 st2*
using *assms* **by** (*metis evaluate-iff*)

lemma *eval-tannot*[*simp*]: *eval env* (*Tannot e t1*) *st* = *eval env e st*
by (*rule eval-eqI*) (*auto elim*: *evaluate.cases intro*: *evaluate-match-evaluate-list-evaluate.intros*)

lemma *eval-lannot*[*simp*]: *eval env* (*Lannot e t1*) *st* = *eval env e st*
by (*rule eval-eqI*) (*auto elim*: *evaluate.cases intro*: *evaluate-match-evaluate-list-evaluate.intros*)

lemma *eval-match*[*simp*]:
  eval env (*Mat e pes*) *st* =
    (*case eval env e st of*
      (*st′*, *Rval v*) ⇒ *eval-match env v pes Bindv st′*
    | (*st′*, *Rerr err*) ⇒ (*st′*, *Rerr err*))
apply (*subst evaluate-iff-sym*[*symmetric*])
apply (*simp only*: *split*!: *prod.splits result.splits*)
subgoal
  apply (*subst* (*asm*) *evaluate-iff-sym*[*symmetric*])
  apply (*rule mat1*, *rule*)
  apply *assumption*
  apply (*subst Bindv-def*)
  apply (*metis valid-eval-match*)
  done
subgoal
  apply (*subst* (*asm*) *evaluate-iff-sym*[*symmetric*])
  by (*auto intro*: *evaluate-match-evaluate-list-evaluate.intros*)
done

lemma *eval-match-empty*[*simp*]: *eval-match env v2* [] *err-v st* = (*st*, *Rerr* (*Rraise err-v*))
by (*subst evaluate-iff-sym*[*symmetric*]) (*auto intro*: *evaluate-match-evaluate-list-evaluate.intros*)

end

lemma *run-eval*: ∃ *run-eval*. ∀ *env e s*. *evaluate True env s e* (*run-eval env e s*)
proof −
  define *f* **where** *f env-e-s* = (*case env-e-s of* (*env*, *e*, *s*::′*a state*) ⇒ *evaluate True env s e*) **for** *env-e-s*
  have ∃ *g*. ∀ *env-e-s*. *f env-e-s* (*g env-e-s*)
    proof (*rule choice*, *safe*, *unfold f-def prod.case*)
      fix *env e*
      fix *s* :: ′*a state*
      obtain *s′ r* **where** *evaluate True env s e* (*s′*, *r*)
        by (*metis evaluate-total*)

then show ∃ *r*. *evaluate True env s e r*
by *auto*
qed
then show *?thesis*
unfolding *f-def*
by *force*
qed

lemma *run-eval-list*: ∃ *run-eval-list*. ∀ *env es s*. *evaluate-list True env s es* (*run-eval-list env es s*)
proof −
define *f* **where** *f env-es-s* = (*case env-es-s of* (*env*, *es*, *s*::′*a state*) ⇒ *evaluate-list True env s es*) **for** *env-es-s*
have ∃ *g*. ∀ *env-es-s*. *f env-es-s* (*g env-es-s*)
proof (*rule choice*, *safe*, *unfold f-def prod.case*)
fix *env es*
fix *s* :: ′*a state*
obtain *s*′ *r* **where** *evaluate-list True env s es* (*s*′, *r*)
by (*metis evaluate-list-total*)
then show ∃ *r*. *evaluate-list True env s es r*
by *auto*
qed
then show *?thesis*
unfolding *f-def*
by *force*
qed

lemma *run-eval-match*: ∃ *run-eval-match*. ∀ *env v pes err-v s*. *evaluate-match True env s v pes err-v* (*run-eval-match env v pes err-v s*)
proof −
define *f* **where** *f env-v-pes-err-v-s* = (*case env-v-pes-err-v-s of* (*env*, *v*, *pes*, *err-v*, *s*::′*a state*) ⇒ *evaluate-match True env s v pes err-v*) **for** *env-v-pes-err-v-s*
have ∃ *g*. ∀ *env-es-s*. *f env-es-s* (*g env-es-s*)
proof (*rule choice*, *safe*, *unfold f-def prod.case*)
fix *env v pes err-v*
fix *s* :: ′*a state*
obtain *s*′ *r* **where** *evaluate-match True env s v pes err-v* (*s*′, *r*)
by (*metis evaluate-match-total*)
then show ∃ *r*. *evaluate-match True env s v pes err-v r*
by *auto*
qed
then show *?thesis*
unfolding *f-def*
by *force*
qed

global-interpretation *run*: *eval*
SOME f. ∀ *env e s*. *evaluate True env s e* (*f env e s*)
SOME f. ∀ *env es s*. *evaluate-list True env s es* (*f env es s*)

SOME f. ∀ env v pes err-v s. evaluate-match True env s v pes err-v (f env v pes err-v s)
defines
run-eval = SOME f. ∀ env e s. evaluate True env s e (f env e s) **and**
run-eval-list = SOME f. ∀ env es s. evaluate-list True env s es (f env es s) **and**
run-eval-match = SOME f. ∀ env v pes err-v s. evaluate-match True env s v pes err-v (f env v pes err-v s)
proof (*standard*, *goal-cases*)
case *1*
show *?case*
using *someI-ex*[*OF run-eval*, *rule-format*] **.**
next
case *2*
show *?case*
using *someI-ex*[*OF run-eval-list*, *rule-format*] **.**
next
case *3*
show *?case*
using *someI-ex*[*OF run-eval-match*, *rule-format*] **.**
qed

hide-fact *run-eval*
hide-fact *run-eval-list*
hide-fact *run-eval-match*

lemma *fun-evaluate*:
evaluate-match True env s v pes err-v (map-prod id (map-result hd id) (fun-evaluate-match s env v pes err-v))
evaluate-list True env s es (fun-evaluate s env es)
proof (*induction rule*: *fun-evaluate-induct*)
case (*5 st env e pes*)
from *5(1)* **show** *?case*
apply (*rule evaluate-list-singleton-cases*)
subgoal
apply *simp*
apply (*rule evaluate-match-evaluate-list-evaluate.cons1*)
apply (*intro conjI*)
apply (*rule handle1*)
apply *assumption*
apply (*rule evaluate-match-evaluate-list-evaluate.empty*)
done
subgoal for *s′ err*
apply (*simp split*!: *error-result.splits*)
subgoal for *exn*
apply (*cases fun-evaluate-match s′ env exn pes exn rule*: *prod-result-cases*; *simp only*:)
subgoal premises *prems*
using *prems(4)*
apply (*rule fun-evaluate-matchE*)

apply *simp*
apply (*rule evaluate-match-evaluate-list-evaluate.cons1*)
apply (*intro conjI*)
apply (*rule handle2*)
apply (*intro conjI*)
apply (*rule prems*)
using *5(2)*[*OF prems(1)*[*symmetric*] *refl refl*, *unfolded prems(4)*]
apply *simp*
by (*rule evaluate-match-evaluate-list-evaluate.empty*)
subgoal premises *prems*
apply (*rule evaluate-match-evaluate-list-evaluate.cons2*)
apply (*rule handle2*)
apply (*intro conjI*)
apply (*rule prems*)
supply *error-result.map-ident*[*simp*]
using *5(2)*[*OF prems(1)*[*symmetric*] *refl refl*, *unfolded prems(4)*, *simplified*]
.
done
subgoal
apply (*rule evaluate-match-evaluate-list-evaluate.cons2*)
apply (*rule handle3*)
by *assumption*
done
done
next
case (*6 st env cn es*)
show *?case*
proof (*cases do-con-check* (*c env*) *cn* (*length es*))
case *True*
then show *?thesis*
apply *simp*
apply (*frule 6*)
apply (*cases fun-evaluate st env* (*rev es*) *rule*: *prod-result-cases*; *simp*)
subgoal for - *vs*
apply (*frule do-con-check-build-conv*[**where** *vs* = *rev vs*], *auto split*: *option.splits*)
apply (*rule evaluate-match-evaluate-list-evaluate.cons1*)
apply (*intro conjI*)
apply (*rule con1*)
apply (*intro conjI*)
apply *assumption+*
by (*rule evaluate-match-evaluate-list-evaluate.empty*)
subgoal
by (*auto intro*: *evaluate-match-evaluate-list-evaluate.intros*)
done
next
case *False*
then show *?thesis*
apply *simp*

```
      apply (rule evaluate-match-evaluate-list-evaluate.cons2)
      apply (rule con2)
      by assumption
    qed
next
  case (9 st env op es)
  note do-app.simps[simp del]
  show ?case
    apply (cases fun-evaluate st env (rev es) rule: prod-result-cases; simp)
    subgoal
      apply (safe; simp split!: option.splits)
      subgoal using 9 by (auto intro: evaluate-match-evaluate-list-evaluate.intros)
      subgoal premises prems
        apply (rule conjI)
        using 9 prems apply (fastforce intro: evaluate-match-evaluate-list-evaluate.intros)
        apply safe
        using 9(2)[OF prems(1)[symmetric] refl prems(2) prems(3) refl]
        apply (cases rule: evaluate-list-singleton-cases)
        subgoal by simp
        subgoal
          apply simp
          apply (rule evaluate-match-evaluate-list-evaluate.cons1)
          using 9 prems
          by (auto intro: evaluate-match-evaluate-list-evaluate.intros simp: dec-clock-def)
        subgoal
          apply simp
          apply (rule evaluate-match-evaluate-list-evaluate.cons2)
          using 9 prems
          by (auto intro: evaluate-match-evaluate-list-evaluate.intros simp: dec-clock-def)
        done
      subgoal using 9 by (auto intro: evaluate-match-evaluate-list-evaluate.intros)
      subgoal
        apply (rule evaluate-list-singletonI)
        apply (rule app4)
        apply (intro conjI)
        using 9 by auto
      done
    subgoal
      apply (rule evaluate-match-evaluate-list-evaluate.cons2)
      apply (rule app6)
      using 9(1) by simp
    done
next
  case (12 st env e pes)
  from 12(1) show ?case
    apply (rule evaluate-list-singleton-cases)
    subgoal for s' v
      apply simp
      apply (cases fun-evaluate-match s' env v pes Bindv rule: prod-result-cases;
```

```
simp only:)
      subgoal premises prems
        using prems(3)
        apply (rule fun-evaluate-matchE)
        apply simp
        apply (rule evaluate-match-evaluate-list-evaluate.cons1)
        apply (intro conjI)
        apply (rule mat1)
        apply (fold Bindv-def)
        apply (intro conjI)
        apply (rule prems)
        supply error-result.map-ident[simp]
         using 12(2)[OF prems(1)[symmetric] refl, simplified, unfolded prems(3),
simplified]
        apply simp
        by (rule evaluate-match-evaluate-list-evaluate.empty)
      subgoal premises prems
        apply (rule evaluate-match-evaluate-list-evaluate.cons2)
        apply (rule mat1)
        apply (fold Bindv-def)
        apply (intro conjI)
        apply (rule prems)
        supply error-result.map-ident[simp]
         using 12(2)[OF prems(1)[symmetric] refl, simplified, unfolded prems(3),
simplified] .
      done
    subgoal
      apply simp
      apply (rule evaluate-match-evaluate-list-evaluate.cons2)
      apply (rule mat2)
      by assumption
    done
next
  case (14 st env funs e)
  then show ?case
    by (cases allDistinct (map (λ(x, y, z). x) funs))
     (fastforce intro: evaluate-match-evaluate-list-evaluate.intros elim: evaluate-list-singleton-cases)+
next
  case (18 st env v2 p e pes err-v)
  show ?case
    proof (cases allDistinct (pat-bindings p []))
      case True
      show ?thesis
        proof (cases pmatch (c env) (refs st) p v2 [])
          case No-match
          with True show ?thesis
            apply (simp del: id-apply)
            apply (rule mat-cons2)
            apply (intro conjI)
```

```
      apply assumption+
      apply (rule 18)
      apply assumption+
      done
    next
      case Match-type-error
      with True show ?thesis
        apply simp
        apply (rule mat-cons3)
        apply assumption+
        done
    next
      case Match
      with True show ?thesis
        apply (simp del: id-apply)
        apply (rule mat-cons1)
        apply (intro conjI)
        apply assumption+
        using 18(2)
        apply (rule evaluate-list-singleton-cases)
        apply assumption+
        apply (auto simp: error-result.map-ident)
        done
    qed
  next
    case False
    show ?thesis
      using False by (auto intro: mat-cons4)
  qed
qed (fastforce
      intro: evaluate-match-evaluate-list-evaluate.intros
      elim: evaluate-list-singleton-cases
      split: option.splits prod.splits result.splits if-splits exp-or-val.splits)+

global-interpretation fun: eval
  λenv e s. map-prod id (map-result hd id) (fun-evaluate s env [e])
  λenv es s. fun-evaluate s env es
  λenv v pes err-v s. map-prod id (map-result hd id) (fun-evaluate-match s env v
pes err-v)
proof (standard, goal-cases)
  case (1 env s e)
  have evaluate-list True env s [e] (fun-evaluate s env [e])
    by (metis fun-evaluate)
  then show ?case
    by (rule evaluate-list-singleton-cases) (auto simp: error-result.map-id)
next
  case (2 env s es)
  show ?case
    by (rule fun-evaluate)
```

next
  case (*3 env s v pes err-v*)
  show *?case*
    by (*rule fun-evaluate*)
qed

lemmas *big-fun-equivalence* =
  fun.other-eval-eq[*OF run.eval-axioms*]
—

run-eval =
(*λenv e s. map-prod id* (*map-result hd id*) (*fun-evaluate s env* [*e*]))
run-eval-list = (*λenv es s. fun-evaluate s env es*)
run-eval-match =
(*λenv v pes err-v s.*
    map-prod id (*map-result hd id*) (*fun-evaluate-match s env v pes err-v*))

end

22.4 A simpler version with no clock parameter and factored-out matching

theory *Big-Step-Unclocked*
imports
  Semantic-Extras
  Big-Step-Determ
begin

inductive

evaluate-list :: (*v*)*sem-env* ⇒ ′*ffi state* ⇒(*exp*)*list* ⇒ ′*ffi state*∗(((*v*)*list*),(*v*))*result* ⇒ *bool*
    and
evaluate :: (*v*)*sem-env* ⇒ ′*ffi state* ⇒ *exp* ⇒ ′*ffi state*∗((*v*),(*v*))*result* ⇒ *bool*
where

lit : ⋀ *env l s.*

evaluate env s (*Lit l*) (*s*, *Rval* (*Litv l*))

|

raise1 : ⋀ *env e s1 s2 v1*.
evaluate s1 env e (*s2*, *Rval v1*)
==>
evaluate s1 env (*Raise e*) (*s2*, *Rerr* (*Rraise v1*))

|

raise2 : $\bigwedge$ *env e s1 s2 err*.
evaluate s1 env e (*s2*, *Rerr err*)
==>
evaluate s1 env (*Raise e*) (*s2*, *Rerr err*)

|

handle1 : $\bigwedge$ *s1 s2 env e v1 pes*.
evaluate s1 env e (*s2*, *Rval v1*)
==>
evaluate s1 env (*Handle e pes*) (*s2*, *Rval v1*)

|

handle2 : $\bigwedge$ *s1 s2 env e pes v1 bv*.
evaluate env s1 e (*s2*, *Rerr* (*Rraise v1*)) $\Longrightarrow$
match-result env s2 v1 pes v1 = *Rval* (*e'*, *env'*) $\Longrightarrow$
evaluate (*env* ⦇ *sem-env.v* := *nsAppend* (*alist-to-ns env'*) (*sem-env.v env*) ⦈) *s2 e'* *bv*
==>
evaluate env s1 (*Handle e pes*) *bv*

|

handle2b : $\bigwedge$ *s1 s2 env e pes v1*.
evaluate env s1 e (*s2*, *Rerr* (*Rraise v1*)) $\Longrightarrow$
match-result env s2 v1 pes v1 = *Rerr err*
==>
evaluate env s1 (*Handle e pes*) (*s2*, *Rerr err*)

|

handle3 : $\bigwedge$ *s1 s2 env e pes a*.
evaluate env s1 e (*s2*, *Rerr* (*Rabort a*))
==>
evaluate env s1 (*Handle e pes*) (*s2*, *Rerr* (*Rabort a*))

|

con1 : $\bigwedge$ *env cn es vs s s' v1*.
do-con-check(*c* *env*) *cn* (*List.length es*) $\Longrightarrow$
build-conv(*c* *env*) *cn* (*List.rev vs*) = *Some v1* $\Longrightarrow$
evaluate-list env s (*List.rev es*) (*s'*, *Rval vs*)
==>
evaluate env s (*Con cn es*) (*s'*, *Rval v1*)

|

con2 : $\bigwedge$ *env cn es s.*
$\neg$ (*do-con-check*(*c* *env*) *cn* (*List.length es*))
==>
evaluate env s (*Con cn es*) (*s*, *Rerr* (*Rabort Rtype-error*))

|

con3 : $\bigwedge$ *env cn es err s s'.*
do-con-check(*c* *env*) *cn* (*List.length es*) $\Longrightarrow$
evaluate-list env s (*List.rev es*) (*s'*, *Rerr err*)
==>
evaluate env s (*Con cn es*) (*s'*, *Rerr err*)

|

var1 : $\bigwedge$ *env n v1 s.*
nsLookup(*sem-env.v* *env*) *n* = *Some v1*
==>
evaluate env s (*Var n*) (*s*, *Rval v1*)

|

var2 : $\bigwedge$ *env n s.*
nsLookup(*sem-env.v* *env*) *n* = *None*
==>
evaluate env s (*Var n*) (*s*, *Rerr* (*Rabort Rtype-error*))

|

fn : $\bigwedge$ *env n e s.*

evaluate env s (*Fun n e*) (*s*, *Rval* (*Closure env n e*))

|

app1 : $\bigwedge$ *env es vs env' e bv s1 s2.*
evaluate-list env s1 (*List.rev es*) (*s2*, *Rval vs*) $\Longrightarrow$
do-opapp (*List.rev vs*) = *Some* (*env'*, *e*) $\Longrightarrow$
evaluate env' s2 e bv
==>
evaluate env s1 (*App Opapp es*) *bv*

|

app3 : $\bigwedge$ *env es vs s1 s2.*
evaluate-list env s1 (*List.rev es*) (*s2*, *Rval vs*) $\Longrightarrow$
(*do-opapp* (*List.rev vs*) = *None*)
==>

evaluate env s1 (App Opapp es) (s2, Rerr (Rabort Rtype-error))

|

app4 : $\bigwedge$ *env op0 es vs res s1 s2 refs′ ffi′*.
evaluate-list env s1 (List.rev es) (s2, Rval vs) $\Longrightarrow$
do-app ((refs s2),(ffi s2)) op0 (List.rev vs) = Some ((refs′,ffi′), res) $\Longrightarrow$
op0 $\neq$ *Opapp*
==>
evaluate env s1 (App op0 es) ((s2 (| refs := refs′, ffi :=ffi′ |)), res)

|

app5 : $\bigwedge$ *env op0 es vs s1 s2*.
evaluate-list env s1 (List.rev es) (s2, Rval vs) $\Longrightarrow$
do-app ((refs s2),(ffi s2)) op0 (List.rev vs) = None $\Longrightarrow$
op0 $\neq$ *Opapp*
==>
evaluate env s1 (App op0 es) (s2, Rerr (Rabort Rtype-error))

|

app6 : $\bigwedge$ *env op0 es err s1 s2*.
evaluate-list env s1 (List.rev es) (s2, Rerr err)
==>
evaluate env s1 (App op0 es) (s2, Rerr err)

|

log1 : $\bigwedge$ *env op0 e1 e2 v1 e′ bv s1 s2*.
evaluate env s1 e1 (s2, Rval v1) $\Longrightarrow$
do-log op0 v1 e2 = Some (Exp e′) $\Longrightarrow$
evaluate env s2 e′ bv
==>
evaluate env s1 (Log op0 e1 e2) bv

|

log2 : $\bigwedge$ *env op0 e1 e2 v1 bv s1 s2*.
evaluate env s1 e1 (s2, Rval v1) $\Longrightarrow$
(do-log op0 v1 e2 = Some (Val bv))
==>
evaluate env s1 (Log op0 e1 e2) (s2, Rval bv)

|

log3 : $\bigwedge$ *env op0 e1 e2 v1 s1 s2*.
evaluate env s1 e1 (s2, Rval v1) $\Longrightarrow$
(do-log op0 v1 e2 = None)

==>
evaluate env s1 (Log op0 e1 e2) (s2, Rerr (Rabort Rtype-error))

|

log4 : $\bigwedge$ *env op0 e1 e2 err s s′.*
evaluate env s e1 (s′, Rerr err)
==>
evaluate env s (Log op0 e1 e2) (s′, Rerr err)

|

if1 : $\bigwedge$ *env e1 e2 e3 v1 e′ bv s1 s2.*
evaluate env s1 e1 (s2, Rval v1) $\Longrightarrow$
do-if v1 e2 e3 = Some e′ $\Longrightarrow$
evaluate env s2 e′ bv
==>
evaluate env s1 (If e1 e2 e3) bv

|

if2 : $\bigwedge$ *env e1 e2 e3 v1 s1 s2.*
evaluate env s1 e1 (s2, Rval v1) $\Longrightarrow$
(do-if v1 e2 e3 = None)
==>
evaluate env s1 (If e1 e2 e3) (s2, Rerr (Rabort Rtype-error))

|

if3 : $\bigwedge$ *env e1 e2 e3 err s s′.*
evaluate env s e1 (s′, Rerr err)
==>
evaluate env s (If e1 e2 e3) (s′, Rerr err)

|

mat1 : $\bigwedge$ *env e pes v1 bv s1 s2.*
evaluate env s1 e (s2, Rval v1) $\Longrightarrow$
match-result env s2 v1 pes Bindv = Rval (e′, env′) $\Longrightarrow$
evaluate (env $(\!|$ *sem-env.v := nsAppend (alist-to-ns env′) (sem-env.v env)* $|\!)$*) s2 e′*
bv
==>
evaluate env s1 (Mat e pes) bv

|

mat1b : $\bigwedge$ *env e pes v1 s1 s2.*
evaluate env s1 e (s2, Rval v1) $\Longrightarrow$
match-result env s2 v1 pes Bindv = Rerr err

==>
evaluate env s1 (Mat e pes) (s2, Rerr err)

|

mat2 : $\bigwedge$ *env e pes err s s'.*
evaluate env s e (s', Rerr err)
==>
evaluate env s (Mat e pes) (s', Rerr err)

|

let1 : $\bigwedge$ *env n e1 e2 v1 bv s1 s2.*
evaluate env s1 e1 (s2, Rval v1) $\Longrightarrow$
evaluate (env (| sem-env.v := (nsOptBind n v1(sem-env.v env)) |)) s2 e2 bv
==>
evaluate env s1 (Let n e1 e2) bv

|

let2 : $\bigwedge$ *env n e1 e2 err s s'.*
evaluate env s e1 (s', Rerr err)
==>
evaluate env s (Let n e1 e2) (s', Rerr err)

|

letrec1 : $\bigwedge$ *env funs e bv s.*
distinct (List.map (λx .
(case x of (x,y,z) => x)) funs) $\Longrightarrow$
evaluate (env (| sem-env.v := (build-rec-env funs env(sem-env.v env)) |)) s e bv
==>
evaluate env s (Letrec funs e) bv

|

letrec2 : $\bigwedge$ *env funs e s.*
¬ *(distinct (List.map (λx .*
(case x of (x,y,z) => x)) funs))
==>
evaluate env s (Letrec funs e) (s, Rerr (Rabort Rtype-error))

|

tannot : $\bigwedge$ *env e t0 s bv.*
evaluate env s e bv
==>
evaluate env s (Tannot e t0) bv

|

locannot : $\bigwedge$ *env e l s bv*.
evaluate env s e bv
==>
evaluate env s (*Lannot e l*) *bv*

|

empty : $\bigwedge$ *env s*.

evaluate-list env s [] (*s*, *Rval* [])

|

cons1 : $\bigwedge$ *env e es v1 vs s1 s2 s3*.
evaluate env s1 e (*s2*, *Rval v1*) $\Longrightarrow$
evaluate-list env s2 es (*s3*, *Rval vs*)
==>
evaluate-list env s1 (*e* # *es*) (*s3*, *Rval* (*v1* # *vs*))

|

cons2 : $\bigwedge$ *env e es err s s'*.
evaluate env s e (*s'*, *Rerr err*)
==>
evaluate-list env s (*e* # *es*) (*s'*, *Rerr err*)

|

cons3 : $\bigwedge$ *env e es v1 err s1 s2 s3*.
evaluate env s1 e (*s2*, *Rval v1*) $\Longrightarrow$
evaluate-list env s2 es (*s3*, *Rerr err*)
==>
evaluate-list env s1 (*e* # *es*) (*s3*, *Rerr err*)

lemma *unclocked-sound*:
  evaluate-list v s es bv $\Longrightarrow$ *BigStep.evaluate-list False v s es bv*
  evaluate v s e bv' $\Longrightarrow$ *BigStep.evaluate False v s e bv'*
proof (*induction rule*: *evaluate-list-evaluate.inducts*)
  case (*handle2 e' env' s1 s2 env e pes v1 bv*)
  show *?case*
    apply (*rule BigStep.handle2*, *intro conjI*)
    apply *fact*
    apply (*rule match-result-sound-val*)
    apply *fact+*
    done
next
  case (*handle2b err s1 s2 env e pes v1*)

show *?case*
apply (*rule BigStep.handle2*, *intro conjI*)
apply *fact*
apply (*rule match-result-sound-err*)
apply *fact*
done
next
case (*mat1 e′ env′ env e pes v1 bv s1 s2*)
show *?case*
apply (*rule BigStep.mat1*, *fold Bindv-def*, *intro conjI*)
apply *fact*
apply (*rule match-result-sound-val*)
apply *fact+*
done
next
case (*mat1b err env e pes v1 s1 s2*)
show *?case*
apply (*rule BigStep.mat1*, *fold Bindv-def*, *intro conjI*)
apply *fact*
apply (*rule match-result-sound-err*)
apply *fact*
done
qed (*fastforce simp*: *all-distinct-alt-def*[*symmetric*] *intro*: *evaluate-match-evaluate-list-evaluate.intros*)+

context begin

private lemma *unclocked-complete0*:
BigStep.evaluate-match ck env s v0 pes err-v (*s′*, *bv*) ⟹ ¬ *ck* ⟹ (
case bv of
Rval v ⇒
∃ *e env′*.
match-result env s v0 pes err-v = *Rval* (*e*, *env′*) ∧
evaluate (*env* ⦇ *sem-env.v* := *nsAppend* (*alist-to-ns env′*) (*sem-env.v env*)
⦈) *s e* (*s′*, *Rval v*)
| *Rerr err* ⇒
(*match-result env s v0 pes err-v* = *Rerr err*) ∨
(∃ *e env′*.
match-result env s v0 pes err-v = *Rval* (*e*, *env′*) ∧
evaluate (*env* ⦇ *sem-env.v* := *nsAppend* (*alist-to-ns env′*) (*sem-env.v env*)
⦈) *s e* (*s′*, *Rerr err*)))
BigStep.evaluate-list ck v s es (*s′*, *bv0*) ⟹ ¬ *ck* ⟹ *evaluate-list v s es* (*s′*, *bv0*)
BigStep.evaluate ck v s e (*s′*, *bv*) ⟹ ¬ *ck* ⟹ *evaluate v s e* (*s′*, *bv*)
proof (*induction rule*: *evaluate-induct*)
case (*handle2 ck s1 s2 env e pes v1 s3 bv*)
show *?case*
proof (*cases bv*)
case (*Rval v*)
with *handle2* **obtain** *e env′* **where**
match-result env s2 v1 pes v1 = *Rval* (*e*, *env′*)

```
      evaluate (env (| sem-env.v := nsAppend (alist-to-ns env') (sem-env.v env)
|)) s2 e (s3, Rval v)
      by auto

    show ?thesis
      unfolding ‹bv = -›
      apply (rule evaluate-list-evaluate.handle2)
      using handle2 apply blast
      by fact+
  next
    case (Rerr err)
    with handle2 consider
      (match-err) match-result env s2 v1 pes v1 = Rerr err |
      (eval-err) e env' where
        match-result env s2 v1 pes v1 = Rval (e, env')
        evaluate (env (| sem-env.v := nsAppend (alist-to-ns env') (sem-env.v env)
|)) s2 e (s3, Rerr err)
      by auto
    then show ?thesis
      proof cases
        case match-err
        then have evaluate-match ck env s2 v1 pes v1 (s2, Rerr err)
          by (metis match-result-sound-err)
        moreover have evaluate-match ck env s2 v1 pes v1 (s3, Rerr err)
          using handle2 unfolding ‹bv = -› by blast
        ultimately have s2 = s3
          by (metis evaluate-determ fst-conv)

        show ?thesis
          unfolding ‹bv = -›
          apply (rule evaluate-list-evaluate.handle2b)
          using handle2 unfolding ‹s2 = -› apply blast
          using match-err unfolding ‹s2 = -› .
      next
        case eval-err
        show ?thesis
          unfolding ‹bv = -›
          apply (rule evaluate-list-evaluate.handle2)
          using handle2 apply blast
          by fact+
      qed
  qed
next
  case (mat1 ck env e pes v1 s3 v' s1 s2)
  then show ?case

    apply (auto split: result.splits simp: Bindv-def[symmetric])
    subgoal by (rule evaluate-list-evaluate.mat1) auto
    subgoal
```

```
      apply (frule match-result-sound-err)
      apply (subgoal-tac s2 = s3)
      apply (rule evaluate-list-evaluate.mat1b)
      apply force
      apply force
      apply (drule evaluate-determ)
      apply assumption
      by auto
    subgoal by (rule evaluate-list-evaluate.mat1) auto
    done
next
  case (mat-cons1 ck env env' v1 p pes e a b err-v s)
  then show ?case
    by (auto split: result.splits)
next
  case (mat-cons2 ck env v1 p e pes a b s err-v)
  then show ?case
    by (auto split: result.splits)
qed (fastforce simp: all-distinct-alt-def intro: evaluate-list-evaluate.intros)+

lemma unclocked-complete:
  BigStep.evaluate-list False v s es bv' ⟹ evaluate-list v s es bv'
  BigStep.evaluate False v s e bv ⟹ evaluate v s e bv
apply (cases bv'; metis unclocked-complete0)
apply (cases bv; metis unclocked-complete0)
done

end

lemma unclocked-eq:
  evaluate-list = BigStep.evaluate-list False
  evaluate = BigStep.evaluate False
by (auto intro: unclocked-sound unclocked-complete intro!: ext)

lemma unclocked-determ:
  evaluate-list env s es r2a ⟹ evaluate-list env s es r2b ⟹ r2a = r2b
  evaluate env s e r3a ⟹ evaluate env s e r3b ⟹ r3a = r3b
by (metis unclocked-eq evaluate-determ)+

end
```

22.5 Lemmas about the clocked semantics

```
theory Big-Step-Clocked
  imports
    Semantic-Extras
    Big-Step-Total
    Big-Step-Determ
begin
```

— From HOL4 bigClockScript.sml

lemma *do-app-no-runtime-error*:
  assumes *do-app* (*refs s*, *ffi s*) *op0* (*rev vs*) = *Some* ((*refs*′, *ffi*′), *res*)
  shows *res* ≠ *Rerr* (*Rabort Rtimeout-error*)
using *assms*
apply (*auto*
    split: *op0.splits list.splits v.splits lit.splits if-splits word-size.splits*
      eq-result.splits option.splits store-v.splits
    simp: *store-alloc-def store-assign-def call-FFI-def Let-def*)
by (*auto split*: *oracle-result.splits if-splits*)

context
  notes *do-app.simps*[*simp del*]
begin

private lemma *big-unclocked0*:
  evaluate-match ck env s v pes err-v r1 ⟹ *ck* = *False* ⟹ *snd r1* ≠ *Rerr* (*Rabort Rtimeout-error*) ∧ (*clock s*) = (*clock* (*fst r1*))
  evaluate-list ck env s es r2 ⟹ *ck* = *False* ⟹ *snd r2* ≠ *Rerr* (*Rabort Rtimeout-error*) ∧ (*clock s*) = (*clock* (*fst r2*))
  evaluate ck env s e r3 ⟹ *ck* = *False* ⟹ *snd r3* ≠ *Rerr* (*Rabort Rtimeout-error*) ∧ (*clock s*) = (*clock* (*fst r3*))
  by (*induction rule*: *evaluate-match-evaluate-list-evaluate.inducts*)
    (*auto intro*!: *do-app-no-runtime-error*)

corollary *big-unclocked-notimeout*:
  evaluate-match False env s v pes err-v (*s*′, *r1*) ⟹ *r1* ≠ *Rerr* (*Rabort Rtimeout-error*)
  evaluate-list False env s es (*s*′, *r2*) ⟹ *r2* ≠ *Rerr* (*Rabort Rtimeout-error*)
  evaluate False env s e (*s*′, *r3*) ⟹ *r3* ≠ *Rerr* (*Rabort Rtimeout-error*)
using *big-unclocked0* **by** *fastforce*+

corollary *big-unclocked-unchanged*:
  evaluate-match False env s v pes err-v (*s*′, *r1*) ⟹ *clock s* = *clock s*′
  evaluate-list False env s es (*s*′, *r2*) ⟹ *clock s* = *clock s*′
  evaluate False env s e (*s*′, *r3*) ⟹ *clock s* = *clock s*′
using *big-unclocked0* **by** *fastforce*+

private lemma *big-unclocked1*:
 evaluate-match ck env s v pes err-v r1 ⟹ ∀ *st*′ *r*. *r1* = (*st*′, *r*) ∧ *r* ≠ *Rerr* (*Rabort Rtimeout-error*)
  ⟶ *evaluate-match False env* (*s* ⦇ *clock* := *cnt* ⦈) *v pes err-v* ((*st*′ ⦇ *clock* := *cnt* ⦈), *r*)
 evaluate-list ck env s es r2 ⟹ ∀ *st*′ *r*. *r2* = (*st*′, *r*) ∧ *r* ≠ *Rerr* (*Rabort Rtimeout-error*)
  ⟶ *evaluate-list False env* (*s* ⦇ *clock* := *cnt* ⦈) *es* ((*st*′ ⦇ *clock* := *cnt* ⦈), *r*)
 evaluate ck env s e r3 ⟹ ∀ *st*′ *r*. *r3* = (*st*′, *r*) ∧ *r* ≠ *Rerr* (*Rabort Rtimeout-error*)

$\longrightarrow$ *evaluate False env* (*s* ⦇ *clock* := *cnt* ⦈) *e* ((*st′* ⦇ *clock* := *cnt* ⦈), *r*)
by (*induction arbitrary*: *cnt* **and** *cnt* **and** *cnt rule*: *evaluate-match-evaluate-list-evaluate.inducts*)
(*auto intro*: *evaluate-match-evaluate-list-evaluate.intros split*:*if-splits*)

lemma *big-unclocked-ignore*:
evaluate-match ck env s v pes err-v (*st′*, *r1*) $\Longrightarrow$ *r1* $\neq$ *Rerr* (*Rabort Rtimeout-error*) $\Longrightarrow$
evaluate-match False env (*s* ⦇ *clock* := *cnt* ⦈) *v pes err-v* (*st′* ⦇ *clock* := *cnt* ⦈, *r1*)
evaluate-list ck env s es (*st′*, *r2*) $\Longrightarrow$ *r2* $\neq$ *Rerr* (*Rabort Rtimeout-error*) $\Longrightarrow$
evaluate-list False env (*s* ⦇ *clock* := *cnt* ⦈) *es* (*st′* ⦇ *clock* := *cnt* ⦈, *r2*)
evaluate ck env s e (*st′*, *r3*) $\Longrightarrow$ *r3* $\neq$ *Rerr* (*Rabort Rtimeout-error*) $\Longrightarrow$
evaluate False env (*s* ⦇ *clock* := *cnt* ⦈) *e* (*st′* ⦇ *clock* := *cnt* ⦈, *r3*)
by (*rule big-unclocked1*[*rule-format*]; (*assumption* | *simp*))+

lemma *big-unclocked*:
assumes *evaluate False env s e* (*s′*,*r*) $\Longrightarrow$ *r* $\neq$ *Rerr* (*Rabort Rtimeout-error*)
assumes *evaluate False env s e* (*s′*,*r*) $\Longrightarrow$ *clock s* = *clock s′*
assumes *evaluate False env* (*s* ⦇ *clock* := *count1* ⦈) *e* ((*s′* ⦇ *clock* := *count1* ⦈),*r*)
shows *evaluate False env* (*s* ⦇ *clock* := *count2* ⦈) *e* ((*s′* ⦇ *clock* := *count2* ⦈),*r*)
using *assms big-unclocked0*(*3*) *big-unclocked-ignore*(*3*) **by** *fastforce*

private lemma *add-to-counter0*:
evaluate-match ck env s v pes err-v r1 $\Longrightarrow$ $\forall$ *s′ r′ extra*. (*r1* = (*s′*,*r′*)) $\wedge$ (*r′* $\neq$ *Rerr* (*Rabort Rtimeout-error*)) $\wedge$ (*ck* = *True*)
$\longrightarrow$ *evaluate-match True env* (*s* ⦇ *clock* := (*clock s*)+*extra* ⦈) *v pes err-v* ((*s′* ⦇ *clock* := (*clock s′*)+ *extra*⦈),*r′*)
evaluate-list ck env s es r2 $\Longrightarrow$ $\forall$ *s′ r′ extra*. (*r2* = (*s′*,*r′*)) $\wedge$ (*r′* $\neq$ *Rerr* (*Rabort Rtimeout-error*)) $\wedge$ (*ck* = *True*)
$\longrightarrow$ *evaluate-list True env* (*s* ⦇ *clock* := (*clock s*)+*extra* ⦈) *es* ((*s′* ⦇ *clock* := (*clock s′*)+ *extra*⦈),*r′*)
evaluate ck env s e r3 $\Longrightarrow$ $\forall$ *s′ r′ extra*. (*r3* = (*s′*,*r′*)) $\wedge$ (*r′* $\neq$ *Rerr* (*Rabort Rtimeout-error*)) $\wedge$ (*ck* = *True*)
$\longrightarrow$ *evaluate True env* (*s* ⦇ *clock* := (*clock s*)+*extra* ⦈) *e* ((*s′* ⦇ *clock* := (*clock s′*)+ *extra*⦈),*r′*)
by (*induction rule*: *evaluate-match-evaluate-list-evaluate.inducts*)
(*auto intro*: *evaluate-match-evaluate-list-evaluate.intros*)

corollary *add-to-counter*:
evaluate-match True env s v pes err-v (*s′*, *r1*) $\Longrightarrow$ *r1* $\neq$ *Rerr* (*Rabort Rtimeout-error*) $\Longrightarrow$
evaluate-match True env (*s* ⦇ *clock* := *clock s* + *extra* ⦈) *v pes err-v* ((*s′* ⦇ *clock* := *clock s′* + *extra* ⦈), *r1*)
evaluate-list True env s es (*s′*, *r2*) $\Longrightarrow$ *r2* $\neq$ *Rerr* (*Rabort Rtimeout-error*) $\Longrightarrow$
evaluate-list True env (*s* ⦇ *clock* := (*clock s*)+*extra* ⦈) *es* ((*s′* ⦇ *clock* := (*clock s′*)+ *extra*⦈),*r2*)
evaluate True env s e (*s′*, *r3*) $\Longrightarrow$ *r3* $\neq$ *Rerr* (*Rabort Rtimeout-error*) $\Longrightarrow$
evaluate True env (*s* ⦇ *clock* := (*clock s*)+*extra* ⦈) *e* ((*s′* ⦇ *clock* := (*clock s′*)+ *extra*⦈),*r3*)

by (*rule add-to-counter0*[*rule-format*]; (*assumption* | *simp*))+

lemma *add-clock*:
evaluate-match ck env s v pes err-v r1 $\Longrightarrow \forall s'\ r'.$ (*r1* = (s', r') ∧ *ck* = *False*
⟶ (∃ *c*. *evaluate-match True env* (*s* (| *clock* := *c* |)) *v pes err-v* ((s' (| *clock* := *0* |)),r')))
evaluate-list ck env s es r2 $\Longrightarrow \forall s'\ r'.$ (*r2* = (s', r') ∧ *ck* = *False*
⟶ (∃ *c*. *evaluate-list True env* (*s* (| *clock* := *c* |)) *es* ((s' (| *clock* := *0* |)),r')))
evaluate ck env s e r3 $\Longrightarrow \forall s'\ r'.$ (*r3* = (s', r') ∧ *ck* = *False*
⟶ (∃ *c*. *evaluate True env* (*s* (| *clock* := *c* |)) *e* ((s' (| *clock* := *0* |)),r')))
proof (*induction rule*:*evaluate-match-evaluate-list-evaluate.inducts*)
case *app1*
then show *?case*
apply *clarsimp*
subgoal for s' r' *c* c'
apply (*drule add-to-counter*(*2*)[**where** *extra* = c'+*1*])
by (*auto intro*!: *evaluate-match-evaluate-list-evaluate.intros*)
done
qed (*force intro*: *evaluate-match-evaluate-list-evaluate.intros dest*:*add-to-counter*(*3*))+

lemma *clock-monotone*:
evaluate-match ck env s v pes err-v r1 $\Longrightarrow \forall s'\ r'.$ *r1* = (s',r') ∧ (*ck*=*True*) ⟶ (*clock* s') ≤ (*clock* *s*)
evaluate-list ck env s es r2 $\Longrightarrow \forall s'\ r'.$ *r2* = (s',r') ∧ (*ck* = *True*) ⟶ (*clock* s') ≤ (*clock* *s*)
evaluate ck env s e r3 $\Longrightarrow \forall s'\ r'.$ *r3* = (s',r') ∧ (*ck* = *True*) ⟶ (*clock* s') ≤ (*clock* *s*)
by (*induction rule*:*evaluate-match-evaluate-list-evaluate.inducts*) *auto*

lemma *big-clocked-unclocked-equiv*:
evaluate False env s e (s',*r1*) =
(∃ *c*. *evaluate True env* (*s* (| *clock* := *c* |)) *e* ((s' (| *clock* := *0* |)),*r1*) ∧
r1 ≠ *Rerr* (*Rabort Rtimeout-error*) ∧ (*clock* *s*) = (*clock* s')) (**is** *?lhs* = *?rhs*)
proof
assume *?lhs*
then show *?rhs*
using *big-unclocked-unchanged*(*3*) **by** (*fastforce simp add*: *big-unclocked-unchanged big-unclocked-notimeout add-clock*)
next
assume *?rhs*
then show *?lhs*
apply −
apply (*elim conjE exE*)
apply (*drule big-unclocked-ignore*(*3*))
apply *auto*
by (*metis big-unclocked state.record-simps*(*7*))
qed

lemma *big-clocked-timeout-0*:
evaluate-match ck env s v pes err-v r1 $\Longrightarrow$ $\forall s'$. *r1* = (*s'*,*Rerr* (*Rabort Rtimeout-error*)) $\wedge$ *ck* = *True* $\longrightarrow$ (*clock s'*) = *0*
evaluate-list ck env s es r2 $\Longrightarrow$ $\forall s'$. *r2* = (*s'*,*Rerr* (*Rabort Rtimeout-error*)) $\wedge$ *ck* = *True* $\longrightarrow$ (*clock s'*) = *0*
evaluate ck env s e r3 $\Longrightarrow$ $\forall s'$. *r3* = (*s'*,*Rerr* (*Rabort Rtimeout-error*)) $\wedge$ *ck* = *True* $\longrightarrow$ (*clock s'*) = *0*
proof(*induction rule*:*evaluate-match-evaluate-list-evaluate.inducts*)
  case *app4*
  then show *?case* **by** (*auto dest*!:*do-app-no-runtime-error*)
qed(*auto*)

lemma *big-clocked-unclocked-equiv-timeout*:
  ($\forall r$. $\neg$*evaluate False env s e r*) =
  ($\forall c$. $\exists s'$. *evaluate True env* (*s* ⦇ *clock* := *c* ⦈) *e* (*s'*,*Rerr* (*Rabort Rtimeout-error*)) $\wedge$ (*clock s'*) = *0*) (**is** *?lhs* = *?rhs*)
proof *rule*
  assume *l*:*?lhs*
  show *?rhs*
  proof
    fix *c*
    obtain *s' r* **where** *e*:*evaluate True env* (*update-clock* (λ-. *c*) *s*) *e* (*s'*,*r*)
      using *evaluate-total* **by** *blast*
    have *r*:*r* = *Rerr* (*Rabort Rtimeout-error*)
      using *l big-unclocked-ignore*(*3*)[*OF e*, *simplified*]
      by (*metis state.record-simps*(*7*))
    moreover have (*clock s'*) = *0*
      using *r e big-clocked-timeout-0*(*3*) **by** *blast*
    ultimately show $\exists s'$. *evaluate True env* (*update-clock* (λ-. *c*) *s*) *e* (*s'*, *Rerr* (*Rabort Rtimeout-error*)) $\wedge$ *clock s'* = *0*
      using *e* **by** *blast*
  qed
next
  assume *?rhs*
  then show *?lhs*
    by (*metis big-clocked-unclocked-equiv eq-snd-iff evaluate-determ*(*3*))
qed

lemma *sub-from-counter*:
  evaluate-match ck env s v pes err-v r1 $\Longrightarrow$
   $\forall$ *count count' s' r'*.
    (*clock s*) = *count* + *extra1* $\wedge$
    r1 = (*s'*,*r'*) $\wedge$
    (*clock s'*) = *count'* + *extra1* $\wedge$
    ck = *True* $\longrightarrow$
     evaluate-match True env (*s* (| *clock* := *count* |)) *v pes err-v* ((*s'* (| *clock* := *count'* |)),*r'*)
  evaluate-list ck env s es r2 $\Longrightarrow$
   $\forall$ *count count' s' r'*.

(*clock s*) = *count* + *extra2* ∧
r2 = (*s'*,*r'*) ∧
(*clock s'*) = *count'* + *extra2* ∧
ck = *True* ⟶
evaluate-list True env (*s* (| *clock* := *count* |)) *es* ((*s'* (| *clock* := *count'* |)),*r'*)
evaluate ck env s e r3 ⟹
∀ *count count' s' r'*.
(*clock s*) = *count* + *extra3* ∧
r3 = (*s'*,*r'*) ∧
(*clock s'*) = *count'* + *extra3* ∧
ck = *True* ⟶
evaluate True env (*s* (| *clock* := *count* |)) *e* ((*s'* (| *clock* := *count'* |)),*r'*)
proof (*induction arbitrary*:*extra1* **and** *extra2* **and** *extra3 rule*:*evaluate-match-evaluate-list-evaluate.inducts*)
case (*handle2 ck s1 s2 env e pes v1 bv*)
then show *?case*
apply *clarsimp*
apply (*subgoal-tac* (*clock s2*) ≥ *extra3*)
apply (*drule spec*)
apply (*drule spec*)
apply (*drule spec*)
apply (*drule-tac x*=(*clock s2*)−*extra3* **in** *spec*)
apply *rule*
apply *force*
by (*auto dest*:*clock-monotone*(*1*))
next
case (*app1 ck env es vs env' e bv s1 s2*)
then show *?case*
apply *clarsimp*
apply (*subgoal-tac* (*clock s2*)−*1*≥*extra3*)
apply (*drule spec*)
apply (*drule spec*)
apply (*drule spec*)
apply (*drule-tac x*=(*clock s2*)−*extra3*−*1* **in** *spec*)
apply *rule*
apply *force*
by (*auto dest*:*clock-monotone*(*3*))
next
case (*log1 ck env op0 e1 e2 v1 e' bv s1 s2*)
then show *?case*
apply *clarsimp*
apply (*subgoal-tac* (*clock s2*)≥*extra3*)
apply (*drule spec*)
apply (*drule spec*)
apply (*drule spec*)
apply (*drule-tac x*=(*clock s2*)−*extra3* **in** *spec*)
apply *rule*
apply *force*
by (*auto dest*:*clock-monotone*(*3*))
next

```
  case (if1 ck env e1 e2 e3 v1 e' bv s1 s2)
  then show ?case
    apply clarsimp
    apply (subgoal-tac (clock  s2)≥extra3)
     apply (drule spec)
     apply (drule spec)
     apply (drule spec)
     apply (drule-tac x=(clock  s2)−extra3 in spec)
     apply rule
     apply force
   by (auto intro: evaluate-match-evaluate-list-evaluate.intros dest:clock-monotone(3))
next
  case (mat1 ck env e pes v1 bv s1 s2)
  then show ?case
    apply clarsimp
    apply (subgoal-tac (clock  s2)≥extra3)
     apply (drule spec)
     apply (drule spec)
     apply (drule spec)
     apply (drule-tac x=(clock  s2)−extra3 in spec)
     apply rule
     apply force
    by (auto dest:clock-monotone(1))
next
  case (let1 ck env n e1 e2 v1 bv s1 s2)
  then show ?case
    apply clarsimp
    apply (subgoal-tac (clock  s2)≥extra3)
     apply (drule spec)
     apply (drule spec)
     apply (drule spec)
     apply (drule-tac x=(clock  s2)−extra3 in spec)
     apply rule
    apply force
    by (auto dest:clock-monotone(3))
next
  case (cons1 ck env e es v1 vs s1 s2 s3)
  then show ?case
    apply clarsimp
    apply (subgoal-tac (clock  s2)≥extra2)
     apply (drule spec)
     apply (drule spec)
     apply (drule spec)
     apply (drule-tac x=(clock  s2)−extra2 in spec)
     apply rule
     apply force
    by (auto dest:clock-monotone(2))
next
  case (cons3 ck env e es v1 err s1 s2 s3)
```

then show *?case*
apply *clarsimp*
apply (*subgoal-tac* (*clock s2*)≥*extra2*)
apply (*drule spec*)
apply (*drule spec*)
apply (*drule spec*)
apply (*drule-tac x*=(*clock s2*)−*extra2* **in** *spec*)
apply *rule*
apply *force*
by (*auto dest*:*clock-monotone*(*2*))
qed(*fastforce intro*:*evaluate-match-evaluate-list-evaluate.intros*)+

lemma *clocked-min-counter*:
assumes *evaluate True env s e* (*s'*,*r'*)
shows *evaluate True env* (*s* (| *clock* := (*clock s*) − (*clock s'*) |)) *e* ((*s'* (| *clock* := *0* |)),*r'*)
proof −
from *assms* **have** (*clock s*) ≥ (*clock s'*)
by (*fastforce intro*:*clock-monotone*(*3*)[*rule-format*])
then show *?thesis*
thm *sub-from-counter*(*3*)[*rule-format*]
using *assms* **by** (*auto intro*!:*sub-from-counter*(*3*)[*rule-format*])
qed

lemma *dec-evaluate-not-timeout*:
evaluate-dec False mn env s d (*s'*,*r*) ⟹ *r* ≠ *Rerr* (*Rabort Rtimeout-error*)
by (*ind-cases evaluate-dec False mn env s d* (*s'*, *r*), *auto dest*: *big-unclocked-notimeout*)

lemma *dec-unclocked-ignore*:
evaluate-dec ck mn env s d res ⟹
∀ *s' r count*. *res* = (*s'*,*r*) ∧ *r* ≠ *Rerr* (*Rabort Rtimeout-error*) ⟶
evaluate-dec False mn env (*s* (| *clock* := *count* |)) *d* (*s'* (| *clock* := *count* |),*r*)
proof (*induction rule*:*evaluate-dec.inducts*)
case *dtype1*
then show *?case*
apply *auto*
using *evaluate-dec.intros state.record-simps*(*4*)
by *metis*
next
case *dexn1*
then show *?case*
apply *auto*
using *evaluate-dec.intros state.record-simps*(*4*)
by (*metis Un-insert-left sup-bot.left-neutral*)
qed (*force intro*:*evaluate-dec.intros simp add*:*big-unclocked-ignore*(*3*))+

private lemma *dec-unclocked-1*:
assumes *evaluate-dec False mn env s d* (*s'*,*r*)
shows (*r* ≠ *Rerr* (*Rabort Rtimeout-error*)) ∧ (*clock s*) = (*clock s'*)

using *assms* **by** *cases* (*auto dest*: *big-unclocked-notimeout big-unclocked-unchanged*)

private lemma *dec-unclocked-2*:
assumes *evaluate-dec False mn env* (*s* ⦇ *clock* := *count1* ⦈) *d* ((*s'* ⦇ *clock* := *count1* ⦈),*r*)
shows *evaluate-dec False mn env* (*s* ⦇ *clock* := *count2* ⦈) *d* ((*s'* ⦇ *clock* := *count2* ⦈),*r*)
proof −
from *assms* **have** *r* ≠ *Rerr* (*Rabort Rtimeout-error*)
using *dec-evaluate-not-timeout* **by** *blast*
then show *?thesis*
using *assms dec-unclocked-ignore*[*rule-format*]
by *fastforce*
qed

lemma *dec-unclocked*:
(*evaluate-dec False mn env s d* (*s'*,*r*) ⟶ (*r* ≠ *Rerr* (*Rabort Rtimeout-error*)) ∧ (*clock s*) = (*clock s'*)) ∧
(*evaluate-dec False mn env* (*s* (| *clock* := *count1* |)) *d* ((*s'* (| *clock* := *count1* |)),*r*) ⟶
evaluate-dec False mn env (*s* (| *clock* := *count2* |)) *d* ((*s'* (| *clock* := *count2* |)),*r*))
using *dec-unclocked-1 dec-unclocked-2* **by** *blast*

corollary *big-clocked-unclocked-equiv-timeout-1*:
(∀ *r*. ¬ *evaluate False env s e r*) ⟹
(∀ *c*. ∃ *s'*. *evaluate True env* (*update-clock* (λ-. *c*) *s*) *e* (*s'*, *Rerr* (*Rabort Rtimeout-error*)) ∧ *clock s'* = *0*)
using *big-clocked-unclocked-equiv-timeout* **by** *blast*

lemma *not-evaluate-dec-timeout*:
assumes ∀ *r*. ¬*evaluate-dec False mn env s d r*
shows ∃ *r*. *evaluate-dec True mn env s d r* ∧ *snd r* = *Rerr* (*Rabort Rtimeout-error*)
proof (*cases d*)
case (*Dlet locs p e*)
have ¬ *evaluate False env s e r* **for** *r*
apply *rule*
apply (*cases Lem-list.allDistinct* (*pat-bindings p* []))
subgoal
apply (*cases r*)
apply *hypsubst-thin*
subgoal for *s' r*
apply (*cases r*; *hypsubst-thin*)
subgoal for *v*
apply (*cases pmatch*(*c env*)(*refs s'*) *p v* [])
using *assms* **unfolding** *Dlet* **by** (*metis evaluate-dec.intros*)+
subgoal
using *assms* **unfolding** *Dlet* **by** (*metis dlet5*)

```
      done
    done
  subgoal
    using assms unfolding Dlet by (metis dlet4)
  done
note big-clocked-unclocked-equiv-timeout-1[rule-format, OF this]
 then obtain s' where evaluate True env (update-clock (λ-. clock s) s) e (s',
Rerr (Rabort Rtimeout-error))
  by blast
then have evaluate True env s e (s', Rerr (Rabort Rtimeout-error))
  by simp

have Lem-list.allDistinct (pat-bindings p [])
  apply (rule ccontr)
  apply (drule dlet4)
  using assms Dlet by blast

show ?thesis
  unfolding Dlet
  apply (intro exI conjI)
   apply (rule dlet5)
   apply rule
    apply fact+
  apply simp
  done
qed (metis evaluate-dec.intros assms)+

lemma dec-clocked-total: ∃ res. evaluate-dec True mn env s d res
proof (cases d)
  case (Dlet locs p e)
  obtain s' r where e:evaluate True env s e (s', r) by (metis evaluate-total)
  show ?thesis
    unfolding Dlet
    apply (cases Lem-list.allDistinct (pat-bindings p []))
    subgoal
      using e apply (cases r)
      subgoal for v
        apply hypsubst-thin
        apply (cases pmatch(c   env)(refs   s') p v [])
        using evaluate-dec.intros by metis+
      using evaluate-dec.intros by metis
    using evaluate-dec.intros by metis
qed (blast intro: evaluate-dec.intros)+

lemma dec-clocked-min-counter:
  evaluate-dec ck mn env s d res ⟹ ck = True ⟹
   evaluate-dec ck mn env (s (| clock := (clock   s) − (clock   (fst res))|)) d (((fst
res) (| clock := 0|)), snd res)
proof (induction rule:evaluate-dec.inducts)
```

next
  case *dtype1*
  then show *?case*
    apply *auto*
    using *state.record-simps*(*4*) *evaluate-dec.intros*
    by *metis*
next
  case *dexn1*
  then show *?case*
    apply *auto*
    using *evaluate-dec.intros state.record-simps*(*4*)
    by (*metis Un-insert-left sup-bot.left-neutral*)
qed (*force intro*:*evaluate-dec.intros simp add*:*clocked-min-counter*)+

lemma *dec-sub-from-counter*:
  evaluate-dec ck mn env s d res $\Longrightarrow$
  ($\forall$ *count count′ s′ r*. (*clock s*) = *count* + *extra* $\wedge$ (*clock s′*) = *count′* + *extra* $\wedge$ *res* = (*s′*,*r*) $\wedge$ *ck* = *True* $\longrightarrow$
    evaluate-dec ck mn env (*s* (| *clock* := *count* |)) *d* ((*s′* (| *clock* := *count′* |)),*r*))
proof (*induction rule*:*evaluate-dec.inducts*)
next
  case *dtype1*
  then show *?case*
    apply *auto*
    using *evaluate-dec.intros state.record-simps*(*4*)
    by (*metis*)
next
  case *dtype2*
  then show *?case*
    apply *rule*
    by (*auto intro*!: *evaluate-dec.intros*)
next
  case *dexn1*
  then show *?case*
    apply (*auto*)
    using *evaluate-dec.intros state.record-simps*(*4*)
    by (*metis Un-insert-left sup-bot.left-neutral*)
qed (*force intro*:*evaluate-dec.intros simp add*:*sub-from-counter*)+

lemma *dec-clock-monotone*:
  evaluate-dec ck mn env s d res $\Longrightarrow$ *ck* = *True* $\Longrightarrow$ (*clock* (*fst res*)) $\leq$ (*clock s*)
  by (*induction rule*:*evaluate-dec.inducts*)
    (*auto simp add*:*clock-monotone*)

lemma *dec-add-clock*:
  evaluate-dec ck mn env s d res $\Longrightarrow$
  $\forall$ *s′ r*. *res* = (*s′*,*r*) $\wedge$ *ck* = *False* $\longrightarrow$ ($\exists$ *c*. *evaluate-dec True mn env* (*s* (| *clock* := *c* |)) *d* ((*s′* (| *clock* := *0* |)),*r*))
proof (*induction rule*: *evaluate-dec.inducts*)

```
case dlet1
  then show ?case
    apply rule
    apply (drule add-clock(3))
    by (auto|rule)+
next
  case dlet2
  then show ?case
    apply rule
    apply (drule add-clock(3))
    apply auto
    by rule+ auto
next
  case dlet3
  then show ?case
    apply rule
    apply (drule add-clock(3))
    apply auto
    by rule+ auto
next
  case dlet4
  then show ?case
    by (auto intro:evaluate-dec.intros)
next
  case dlet5
  then show ?case
    apply rule
    apply (drule add-clock(3))
    apply auto
    by rule+ auto
next
  case dletrec1
  then show ?case
    apply auto
    by rule+ auto
next
  case dletrec2
  then show ?case
    apply auto
    by rule+ auto
next
  case dtype1
  then show ?case
    apply auto
    by (metis (full-types) evaluate-dec.dtype1 state.record-simps(4))
next
  case dtype2
  then show ?case
    apply clarsimp
```

by *rule+ auto*
next
case *dtabbrev*
then show *?case*
apply *auto*
by *rule+*
next
case *dexn1*
then show *?case*
apply *auto*
apply (*rule exI*[**where** *x=0*])
using *evaluate-dec.intros state.record-simps*(*4*)
by (*metis Un-insert-left sup-bot.left-neutral*)
next
case *dexn2*
then show *?case*
apply *auto*
apply *rule*
apply *rule*
by *auto*
qed

lemma *dec-add-to-counter*:
evaluate-dec ck mn env s d res $\Longrightarrow$
$\forall s'\ r\ extra.\ res = (s',r) \wedge ck = True \wedge r \neq Rerr\ (Rabort\ Rtimeout\text{-}error) \longrightarrow$
evaluate-dec True mn env $(s\ (\mid clock := (clock\quad s) + extra \mid))\ d\ ((s'\ (\mid clock := (clock\quad s') + extra \mid)),r)$
proof (*induction rule*:*evaluate-dec.inducts*)
next
case *dtype1*
then show *?case*
apply *auto*
using *evaluate-dec.intros state.record-simps*(*4*)
by (*metis*)
next
case *dexn1*
then show *?case*
apply *auto*
using *evaluate-dec.intros state.record-simps*(*4*)
by (*metis Un-insert-left sup-bot.left-neutral*)
qed (*force intro*:*evaluate-dec.intros simp add*:*add-to-counter*(*3*))+

lemma *dec-unclocked-unchanged*:
evaluate-dec ck mn env s d r $\Longrightarrow ck = False \Longrightarrow (snd\ r) \neq Rerr\ (Rabort\ Rtimeout\text{-}error) \wedge (clock\quad s) = (clock\quad (fst\ r))$
by (*induction rule*: *evaluate-dec.inducts*)
(*auto simp*: *big-unclocked-notimeout big-clocked-unclocked-equiv*)

lemma *dec-clocked-unclocked-equiv*:

evaluate-dec False mn env s1 d (*s2*,*r*) =
(∃ *c*. *evaluate-dec True mn env* (*s1* (| *clock* := *c* |)) *d* ((*s2* (| *clock* := *0* |)),*r*) ∧
r ≠ *Rerr* (*Rabort Rtimeout-error*) ∧ (*clock s1*) = (*clock s2*)) (**is** *?lhs* = *?rhs*)
proof
assume *?lhs*
then show *?rhs*
by (*auto dest*:*dec-unclocked-unchanged dec-add-clock*)
next
assume *?rhs*
then show *?lhs*
using *dec-unclocked-ignore*
proof −

obtain *nn* :: *nat* **where**
f1: *evaluate-dec True mn env* (*update-clock* (λ*n*. *nn*) *s1*) *d* (*update-clock* (λ*n*. *0*) *s2*, *r*) ∧ *r* ≠ *Rerr* (*Rabort Rtimeout-error*) ∧ *clock s1* = *clock s2*
using ‹∃ *c*. *evaluate-dec True mn env* (*update-clock* (λ-. *c*) *s1*) *d* (*update-clock* (λ-. *0*) *s2*, *r*) ∧ *r* ≠ *Rerr* (*Rabort Rtimeout-error*) ∧ *clock s1* = *clock s2*› **by** *blast*
then have ∀ *n*. *evaluate-dec False mn env* (*update-clock* (λ*na*. *n*) *s1*) *d* (*update-clock* (λ*na*. *n*) *s2*, *r*)
using *dec-unclocked-ignore*
by *fastforce*
then show *?thesis*
using *f1*
by (*metis* (*full-types*) *state.record-simps*(*7*))
qed
qed

lemma *decs-add-clock*:
evaluate-decs ck mn env s ds res ⟹
∀ *s'* *r*. *res* = (*s'*,*r*) ∧ *ck* = *False* ⟶ (∃ *c*. *evaluate-decs True mn env* (*s* (| *clock* := *c* |)) *ds* (*s'* (| *clock* := *0* |),*r*))
proof (*induction rule*:*evaluate-decs.inducts*)
case *cons1*
then show *?case*
using *dec-add-clock evaluate-decs.cons1* **by** *blast*
next
case *cons2*
then show *?case*
apply *auto*
apply (*drule dec-add-clock*)
using *dec-add-to-counter*[*rule-format*] *evaluate-decs.cons2*
by *fastforce*
qed (*auto intro*:*evaluate-decs.intros*)

lemma *decs-evaluate-not-timeout*:
evaluate-decs ck mn env s ds r ⟹
∀ *s'* *r'*. *ck* = *False* ∧ *r* = (*s'*,*r'*) ⟶ *r'* ≠ *Rerr* (*Rabort Rtimeout-error*)

by (*induction rule*:*evaluate-decs.inducts*)
(*case-tac r*;*fastforce dest*:*dec-evaluate-not-timeout*)+

lemma *decs-unclocked-unchanged*:
evaluate-decs ck mn env s ds r $\Longrightarrow$
$\forall s'\ r'.\ ck = False \wedge r = (s',r') \longrightarrow r' \neq Rerr\ (Rabort\ Rtimeout\text{-}error) \wedge (clock\ s) = (clock\ \ s')$
by (*induction rule*:*evaluate-decs.inducts*)
(*case-tac r*;*fastforce simp add*:*dec-unclocked-unchanged dest*:*dec-evaluate-not-timeout*)+

lemma *decs-unclocked-ignore*:
evaluate-decs ck mn env s d res $\Longrightarrow \forall s'\ r\ count.\ res = (s',r) \wedge r \neq Rerr\ (Rabort\ Rtimeout\text{-}error) \longrightarrow$
evaluate-decs False mn env (*s* (| *clock* := *count* |)) *d* ((*s'* (| *clock* := *count* |)),*r*)
by (*induction rule*:*evaluate-decs.inducts*)
(*auto intro*!:*evaluate-decs.intros simp add*:*dec-unclocked-ignore*)

private lemma *decs-unclocked-2*:
assumes *evaluate-decs False mn env* (*s* (| *clock* := *count1* |)) *ds* ((*s'* (| *clock* := *count1* |)),*r*)
shows *evaluate-decs False mn env* (*s* (| *clock* := *count2* |)) *ds* ((*s'* (| *clock* := *count2* |)),*r*)
using *decs-unclocked-ignore*[*rule-format*] *assms decs-evaluate-not-timeout* **by** *fastforce*

lemma *decs-unclocked*:
(*evaluate-decs False mn env s ds* (*s'*,*r*) $\longrightarrow r \neq Rerr\ (Rabort\ Rtimeout\text{-}error) \wedge (clock\ \ s) = (clock\ \ s')) \wedge$
(*evaluate-decs False mn env* (*s* (| *clock* := *count1* |)) *ds* ((*s'* (| *clock* := *count1* |)),*r*) =
evaluate-decs False mn env (*s* (| *clock* := *count2* |)) *ds* ((*s'* (| *clock* := *count2* |)),*r*))
by (*auto simp add*:*decs-unclocked-unchanged decs-unclocked-2*)

lemma *not-evaluate-decs-timeout*:
assumes $\forall r.\ \neg$*evaluate-decs False mn env s ds r*
shows $\exists r.$ *evaluate-decs True mn env s ds r* $\wedge$ (*snd r*) = *Rerr* (*Rabort Rtimeout-error*)
using *assms* **proof** (*induction ds arbitrary*:*mn env s*)
case *Nil*
then show *?case*
using *assms evaluate-decs.intros* **by** *blast*
next
case (*Cons d ds*)
obtain *s' r* **where** *d*:*evaluate-dec True mn env s d* (*s'*,*r*)
using *dec-clocked-total* **by** *force*
then show *?case*
proof (*cases r*)
case (*Rval new-env*)

have ¬ *evaluate-decs False mn* (*extend-dec-env new-env env*) *s′ ds* (*s3*, *r*) **for** *s3 r*

proof
assume *evaluate-decs False mn* (*extend-dec-env new-env env*) *s′ ds* (*s3*, *r*)
then have *evaluate-decs False mn* (*extend-dec-env new-env env*) (*s′* (|*clock* := (*clock s*)|)) *ds* ((*s3* (|*clock* := (*clock s*)|)), *r*)
using *decs-unclocked decs-unclocked-ignore* **by** *fastforce*
moreover from *d* **have** *evaluate-dec False mn env s d* ((*s′* (|*clock* := (*clock s*)|)), *Rval new-env*)
using *dec-unclocked-ignore*
unfolding *Rval*
by (*metis* (*full-types*) *result.distinct*(*1*) *state.record-simps*(*7*))
ultimately show *False*
using *evaluate-decs.cons2 Cons*
by *blast*
qed

then show *?thesis*
using *Cons.IH*[*simplified*] *evaluate-decs.cons2 d*
unfolding *Rval*
by (*metis combine-dec-result.simps*(*1*) *snd-conv*)
next
case (*Rerr e*)
have *e* = *Rabort Rtimeout-error*

proof (*rule ccontr*)
assume *e* ≠ *Rabort Rtimeout-error*
then obtain *s′* **where** *evaluate-dec False mn env s d* (*s′*,*r*)
using *dec-unclocked-ignore*[*rule-format*, **where** *count*=*clock s*] *d Rerr state.simps*
by *fastforce*
thus *False*
unfolding *Rerr*
using *Cons evaluate-decs.cons1* **by** *blast*
qed

then show *?thesis*
using *d evaluate-decs.cons1 Rerr* **by** *fastforce*
qed
qed

lemma *decs-clocked-total*: ∃ *res*. *evaluate-decs True mn env s ds res*
proof (*induction ds arbitrary*:*mn env s*)
case *Nil*
then show *?case* **by** (*auto intro*:*evaluate-decs.intros*)
next
case (*Cons d ds*)
obtain *s′ r* **where** *d*:*evaluate-dec True mn env s d* (*s′*,*r*)

using *dec-clocked-total*
by *force*
then obtain s'' r' **where** *ds*:*evaluate-decs True mn env* s' *ds* (s'',r')
using *Cons* **by** *force*
from *d ds* **show** *?case*
using *evaluate-decs.intros Cons* **by** (*cases r*;*fastforce*)+
qed

lemma *decs-clock-monotone*:
evaluate-decs ck mn env s d res ⟹ *ck* = *True* ⟹ (*clock* (*fst res*)) ≤ (*clock s*)
by (*induction rule*:*evaluate-decs.inducts*) (*fastforce dest*:*dec-clock-monotone*)+

lemma *decs-sub-from-counter*:
evaluate-decs ck mn env s d res ⟹
∀ *extra count count′ s′ r′*.
(*clock s*) = *count* + *extra* ∧ (*clock s′*) = *count′* + *extra* ∧
res = (*s′*,*r′*) ∧ *ck* = *True* ⟶ *evaluate-decs ck mn env* (*s* ⦇ *clock* := *count* ⦈)
d ((*s′* ⦇ *clock* := *count′* ⦈),*r′*)
proof (*induction rule*:*evaluate-decs.inducts*)
case (*cons2 ck mn s1 s2 s3 env d ds new-env r*)
then show *?case*
apply *auto*
subgoal for *extra*
apply (*subgoal-tac clock s2*≥*extra*)
apply (*metis dec-sub-from-counter diff-add-inverse2 eq-diff-iff evaluate-decs.cons2 le-add2*)
using *decs-clock-monotone* **by** *fastforce*
done
qed (*auto intro*!:*evaluate-decs.intros simp add*:*dec-sub-from-counter*)

lemma *decs-clocked-min-counter*:
assumes *evaluate-decs ck mn env s ds res ck* = *True*
shows *evaluate-decs ck mn env* (*s* ⦇ *clock* := *clock s* − (*clock* (*fst res*))⦈) *ds* (((*fst res*) ⦇ *clock* := *0* ⦈),(*snd res*))
proof −
from *assms* **have** *clock* (*fst res*) ≤ *clock s*
using *decs-clock-monotone* **by** *fastforce*
with *assms* **show** *?thesis*
by (*auto elim*!: *decs-sub-from-counter*[*rule-format*])
qed

lemma *decs-add-to-counter*:
evaluate-decs ck mn env s d res ⟹ ∀ *s′ r extra*. *res* = (*s′*,*r*) ∧ *ck* = *True* ∧ *r* ≠ *Rerr* (*Rabort Rtimeout-error*) ⟶
evaluate-decs True mn env (*s* ⦇ *clock* := *clock s* + *extra* ⦈) *d* ((*s′* ⦇ *clock* := *clock s′* + *extra* ⦈),*r*)
proof (*induction rule*:*evaluate-decs.inducts*)
case *cons2*
then show *?case*

using *dec-add-to-counter evaluate-decs.cons2* **by** *fastforce*
qed (*auto intro*!:*evaluate-decs.intros simp add*:*dec-add-to-counter*)

lemma *top-evaluate-not-timeout*:
evaluate-top False env s tp $(s',r) \Longrightarrow r \neq$ *Rerr* (*Rabort Rtimeout-error*)
by (*ind-cases evaluate-top False env s tp* (s',r)) (*fastforce dest*:*dec-evaluate-not-timeout decs-evaluate-not-timeout*)+

lemma *top-unclocked-ignore*:
assumes *evaluate-top ck env s tp* (s',r) $r \neq$ *Rerr* (*Rabort Rtimeout-error*)
shows *evaluate-top False env* (*s* ⦇ *clock* := *cnt* ⦈) *tp* ((*s′* ⦇ *clock* := *cnt* ⦈),*r*)
using *assms* **proof** (*cases*)
case (*tmod1 s2 ds mn specs new-env*)
then show *?thesis*
proof −
from *tmod1* **have** [*mn*] ∉ *defined-mods* (*update-clock* (λ-. *cnt*) *s*)
by *fastforce*
moreover from *tmod1* **have** *evaluate-decs False* [*mn*] *env* (*update-clock* (λ-. *cnt*) *s*) *ds* (*update-clock* (λ-. *cnt*) *s2*, *Rval new-env*)
using *decs-unclocked-ignore* **by** *fastforce*
ultimately show *?thesis*
unfolding *tmod1*
apply −
apply (*drule evaluate-top.tmod1*[*OF conjI*])
using *tmod1* **by** *auto*
qed
next
case *tmod2*
then show *?thesis* **using** *assms*
apply *auto*
apply (*subst state.record-simps*(*5*)[*symmetric*])
by (*fastforce simp add*:*decs-unclocked-ignore intro*:*evaluate-top.tmod2*[*simplified*])
next
case (*tmod3 ds mn specs*)
then show *?thesis* **by** (*auto intro*:*evaluate-top.intros*)
qed(*auto intro*!:*evaluate-top.intros simp add*:*dec-unclocked-ignore*)

lemma *top-unclocked*:
(*evaluate-top False env s tp* $(s',r) \longrightarrow (r \neq$ *Rerr* (*Rabort Rtimeout-error*)) ∧ (*clock* *s*) = (*clock* *s′*)) ∧
(*evaluate-top False env* (*s* (| *clock* := *count1* |)) *tp* ((*s′* (| *clock* := *count1* |)),*r*) =
evaluate-top False env (*s* (| *clock* := *count2* |)) *tp* ((*s′* (| *clock* := *count2* |)),*r*))
(**is** *?P* ∧ *?Q*)
proof
show *?P*
apply (*auto simp add*:*top-evaluate-not-timeout*)
by (*ind-cases evaluate-top False env s tp* (s',r))

(*auto simp add*:*dec-unclocked decs-unclocked top-evaluate-not-timeout*)
next
show *?Q*
using *top-unclocked-ignore*[*rule-format*] *top-evaluate-not-timeout* **by** *fastforce+*
qed

lemma *not-evaluate-top-timeout*:
assumes ∀ *r*. ¬*evaluate-top False env s tp r*
shows ∃ *r*. *evaluate-top True env s tp r* ∧ (*snd r*) = *Rerr* (*Rabort Rtimeout-error*)
proof (*cases tp*)
case (*Tmod mn specs ds*)
have *ds*:*no-dup-types ds*
using *Tmod assms tmod3* **by** *blast*
have *mn*:[*mn*] ∉ *defined-mods s*
using *Tmod assms tmod4* **by** *blast*
have ¬ *evaluate-decs False* [*mn*] *env s ds* (*s'*, *r*) **for** *s' r*
apply (*cases r*)
using *ds mn Tmod assms tmod1 tmod2* **by** *blast+*
then obtain *s'* **where** *evaluate-decs True* [*mn*] *env s ds* (*s'*, *Rerr* (*Rabort Rtimeout-error*))
by (*metis* (*full-types*) *not-evaluate-decs-timeout*[*simplified*])
show *?thesis*
unfolding *Tmod*
apply (*intro exI conjI*)
apply (*rule tmod2*)
apply (*intro conjI*)
apply *fact+*
apply *simp*
done
next
case (*Tdec d*)
have ¬ *evaluate-dec False* [] *env s d* (*s'*, *r*) **for** *s' r*
apply (*cases r*)
using *tdec1 tdec2 assms Tdec* **by** *blast+*
then obtain *s'* **where** *evaluate-dec True* [] *env s d* (*s'*, *Rerr* (*Rabort Rtimeout-error*))
using *not-evaluate-dec-timeout*[*simplified*] **by** *blast*
show *?thesis*
unfolding *Tdec*
apply (*intro exI conjI*)
apply *rule*
apply *fact*
apply *simp*
done
qed

lemma *top-clocked-total*:
∃ *r*. *evaluate-top True env s tp r*
proof (*cases tp*)

```
  case (Tmod mn specs ds)
  have ds:∃ s' r. evaluate-decs True [mn] env s ds (s',r)
    using decs-clocked-total[simplified] by blast
  from Tmod show ?thesis
    apply (cases no-dup-types ds)
     prefer 2
    using tmod3 apply blast
    apply (cases [mn] ∈(defined-mods s))
    using tmod4 apply blast
    using ds apply auto
    subgoal for s' r
      apply (cases r)
      using evaluate-top.tmod1 evaluate-top.tmod2 by blast+
    done
next
  case (Tdec d)
  have d:∃ s' r. evaluate-dec True [] env s d (s',r)
    using dec-clocked-total[simplified] by blast
  show ?thesis
    unfolding Tdec
    using d apply auto
    subgoal for s' r
      apply (cases r)
      using evaluate-top.tdec1 evaluate-top.tdec2 by blast+
    done
qed

lemma top-clocked-min-counter:
  assumes evaluate-top ck env s tp (s',r) ck
  shows evaluate-top ck env (s (| clock := clock s − clock s' |)) tp (s' (| clock := 0
|),r)
  using assms proof (cases)
  case tmod1
  then show ?thesis
    apply auto
    apply (subst state.record-simps(5)[symmetric])
    apply (rule evaluate-top.tmod1[simplified])
    using assms by (auto dest:decs-clocked-min-counter)
next
  case tmod2
  then show ?thesis
    apply auto
    apply (subst state.record-simps(5)[symmetric])
    apply (rule evaluate-top.tmod2[simplified])
    using assms by (auto dest:decs-clocked-min-counter)
qed (fastforce intro:evaluate-top.intros dest:dec-clocked-min-counter)+

lemma top-add-clock:
  assumes evaluate-top ck env s tp (s',r) ¬ck
```

shows ∃ *c*. *evaluate-top True env* (*s* (| *clock* := *c* |)) *tp* ((*s′* (| *clock* := *0* |)),*r*)
using *assms* **proof** (*cases*)
case (*tdec1 d*)
then obtain *c* **where** *evaluate-dec True* [] *env* (*update-clock* (λ-. *c*) *s*) *d* (*update-clock* (λ-. *0*) *s′*, *r*)
using *dec-add-clock assms* **by** *metis*
then show *?thesis*
unfolding *tdec1*
by − *rule+*
next
case (*tdec2 d*)
then obtain *c* **where** *evaluate-dec True* [] *env* (*update-clock* (λ-. *c*) *s*) *d* (*update-clock* (λ-. *0*) *s′*, *r*)
using *dec-add-clock assms* **by** *metis*
then show *?thesis*
unfolding *tdec2*
by − *rule+*
next
case (*tmod1 s2 ds mn specs new-env*)
then obtain *c* **where** *evaluate-decs True* [*mn*] *env* (*update-clock* (λ-. *c*) *s*) *ds* (*update-clock* (λ-. *0*) *s2*, *Rval new-env*)
using *decs-add-clock assms* **by** *metis*
then show *?thesis*
unfolding *tmod1*
apply *auto*
apply (*subst state.record-simps*(*5*)[*symmetric*])
apply *rule+*
apply (*rule evaluate-top.tmod1*[*simplified*])
using *tmod1* **by** *auto*
next
case (*tmod2 s2 ds mn specs err*)
then obtain *c* **where** *evaluate-decs True* [*mn*] *env* (*update-clock* (λ-. *c*) *s*) *ds* (*update-clock* (λ-. *0*) *s2*, *Rerr err*)
using *decs-add-clock assms* **by** *metis*
then show *?thesis*
unfolding *tmod2*
apply *auto*
apply (*subst state.record-simps*(*5*)[*symmetric*])
apply *rule+*
apply (*rule evaluate-top.tmod2*[*simplified*])
using *tmod2* **by** *auto*
next
case *tmod3*
then show *?thesis* **by** (*auto intro*:*evaluate-top.intros*)
next
case *tmod4*
then show *?thesis*
unfolding *tmod4*
apply −

apply *rule*
apply *rule*
by *simp*
qed

lemma *top-clocked-unclocked-equiv*:
evaluate-top False env s tp (s',r) =
(∃ *c*. *evaluate-top True env* (*s* ⦇ *clock* := *c* ⦈) *tp* ((*s'* ⦇ *clock* := *0* ⦈),*r*) ∧ *r* ≠ *Rerr* (*Rabort Rtimeout-error*) ∧
(*clock s*) = (*clock s'*)) (**is** *?P* = *?Q*)
proof
assume *?P*
then show *?Q*
by (*auto simp add*:*top-add-clock top-unclocked dest*:*top-evaluate-not-timeout*)
next
assume *?Q*
then show *?P*
using *top-unclocked-ignore*

proof −
obtain *nn* :: *nat* **where**
f1: *evaluate-top True env* (*update-clock* (λ*n*. *nn*) *s*) *tp* (*update-clock* (λ*n*. *0*) *s'*, *r*) ∧ *r* ≠ *Rerr* (*Rabort Rtimeout-error*) ∧ *clock s* = *clock s'*
using ‹∃ *c*. *evaluate-top True env* (*update-clock* (λ-. *c*) *s*) *tp* (*update-clock* (λ-. *0*) *s'*, *r*) ∧ *r* ≠ *Rerr* (*Rabort Rtimeout-error*) ∧ *clock s* = *clock s'*› **by** *presburger*
then have ∀ *n*. *evaluate-top False env* (*update-clock* (λ*na*. *n*) *s*) *tp* (*update-clock* (λ*na*. *n*) *s'*, *r*)
using *top-unclocked-ignore*
by *fastforce*
then show *?thesis*
using *f1*
by (*metis state.record-simps*(*7*))
qed
qed

lemma *top-clock-monotone*:
evaluate-top ck env s tp (*s'*,*r*) ⟹ *ck* = *True* ⟹ (*clock s'*) ≤ (*clock s*)
by (*ind-cases evaluate-top ck env s tp* (*s'*,*r*)) (*fastforce dest*:*dec-clock-monotone decs-clock-monotone*)+

lemma *top-sub-from-counter*:
assumes *evaluate-top ck env s tp* (*s'*,*r*) *ck* = *True* (*clock s*) = *cnt* + *extra* (*clock s'*) = *cnt'* + *extra*
shows *evaluate-top ck env* (*s* (| *clock* := *cnt* |)) *tp* ((*s'* (| *clock* := *cnt'* |)),*r*)
using *assms* **proof** (*cases*)
case *tmod1*
then show *?thesis*
using *assms* **apply** *auto*
apply (*subst state.record-simps*(*5*)[*symmetric*])

apply (*rule evaluate-top.tmod1*[*simplified*])
by (*auto dest*:*decs-sub-from-counter*)
next
case *tmod2*
then show *?thesis*
using *assms* **apply** *auto*
apply (*subst state.record-simps*(*5*)[*symmetric*])
apply (*rule evaluate-top.tmod2*[*simplified*])
by (*auto dest*:*decs-sub-from-counter*)
qed (*fastforce intro*:*evaluate-top.intros simp add*:*dec-sub-from-counter*)+

lemma *top-add-to-counter*:
assumes *evaluate-top True env s d* (s',*r*) $r \neq$ *Rerr* (*Rabort Rtimeout-error*)
shows *evaluate-top True env* (*s* (| *clock* := (*clock s*) + *extra* |)) *d* ((s' (| *clock* := (*clock* s') + *extra* |)),*r*)
using *assms* **proof** *cases*
case *tmod1*
then show *?thesis*
apply *auto*
apply (*subst state.record-simps*(*5*)[*symmetric*])
apply (*rule evaluate-top.tmod1*[*simplified*])
by (*auto dest*:*decs-add-to-counter*)
next
case *tmod2*
then show *?thesis*
using *assms* **apply** *auto*
apply (*subst state.record-simps*(*5*)[*symmetric*])
apply (*rule evaluate-top.tmod2*[*simplified*])
by (*auto dest*:*decs-add-to-counter*)
qed (*fastforce intro*:*evaluate-top.intros dest*:*dec-add-to-counter*)+

lemma *prog-clock-monotone*:
evaluate-prog ck env s prog res $\Longrightarrow$ *ck* $\Longrightarrow$ (*clock* (*fst res*)) $\leq$ (*clock s*)
by (*induction rule*:*evaluate-prog.inducts*) (*auto dest*:*top-clock-monotone*)

lemma *prog-unclocked-ignore*:
evaluate-prog ck env s prog res $\Longrightarrow$ $\forall$ *cnt* s' *r*. *res* = (s',*r*) $\wedge$ $r \neq$ *Rerr* (*Rabort Rtimeout-error*)
$\longrightarrow$ *evaluate-prog False env* (*s* (| *clock* := *cnt* |)) *prog* ((s' (| *clock* := *cnt* |)),*r*)
by (*induction rule*:*evaluate-prog.inducts*) (*auto intro*!:*evaluate-prog.intros dest*:*top-unclocked-ignore*)

lemma *prog-unclocked-unchanged*:
evaluate-prog ck env s prog res $\Longrightarrow$ $\neg$*ck* $\Longrightarrow$(*snd res*) $\neq$ *Rerr* (*Rabort Rtimeout-error*) $\wedge$ (*clock* (*fst res*)) = (*clock s*)
proof (*induction rule*:*evaluate-prog.inducts*)
case (*cons1 ck s1 s2 s3 env top0 tops new-env r*)
then have $r \neq$ *Rerr* (*Rabort Rtimeout-error*)
by *simp*
moreover from *cons1* **have** *clock s1* = *clock s2*

using *top-unclocked* **by** *force*
ultimately show *?case*
using *combine-dec-result.simps cons1* **by** (*cases r*;*auto*)
qed (*auto simp add*: *top-clocked-unclocked-equiv*)

private lemma *prog-unclocked-1*:
assumes *evaluate-prog False env s prog* (*s'*,*r*)
shows $r \neq$ *Rerr* (*Rabort Rtimeout-error*) $\wedge$ (*clock s* = *clock s'*)
proof –
from *assms* **show** *?thesis*
using *prog-unclocked-unchanged* **by** *fastforce*
qed

private lemma *prog-unclocked-2*:
assumes *evaluate-prog False env* (*s* ⦇ *clock* := *cnt1* ⦈) *prog* (*s'* ⦇ *clock* := *cnt1* ⦈,*r*)
shows *evaluate-prog False env* (*s* ⦇ *clock* := *cnt2* ⦈) *prog* (*s'* ⦇ *clock* := *cnt2* ⦈,*r*)
proof –
from *assms* **have** $r \neq$ *Rerr* (*Rabort Rtimeout-error*)
using *prog-unclocked-1* **by** *blast*
then show *?thesis*
using *prog-unclocked-ignore assms* **by** *fastforce*
qed

lemma *prog-unclocked*:
(*evaluate-prog False env s prog* (*s'*,*r*) $\longrightarrow$ $r \neq$ *Rerr* (*Rabort Rtimeout-error*) $\wedge$ (*clock s* = *clock s'*)) $\wedge$
(*evaluate-prog False env* (*s* ⦇ *clock* := *cnt1* ⦈) *prog* (*s'* ⦇ *clock* := *cnt1* ⦈,*r*) =
evaluate-prog False env (*s* ⦇ *clock* := *cnt2* ⦈) *prog* (*s'* ⦇ *clock* := *cnt2* ⦈,*r*))
using *prog-unclocked-1 prog-unclocked-2* **by** *blast*

lemma *not-evaluate-prog-timeout*:
assumes $\forall$ *res*. $\neg$*evaluate-prog False env s prog res*
shows $\exists$ *r*. *evaluate-prog True env s prog r* $\wedge$ *snd r* = *Rerr* (*Rabort Rtime-out-error*)
using *assms* **proof** (*induction prog arbitrary*:*env s*)
case *Nil*
then show *?case*
using *evaluate-prog.intros*(*1*) **by** *blast*
next
case (*Cons top0 tops*)
obtain *s' r* **where** *top0*:*evaluate-top True env s top0* (*s'*,*r*)
using *top-clocked-total*[*simplified*] **by** *blast*
then show *?case*
proof (*cases r*)
case (*Rval new-env*)
have $\neg$ *evaluate-prog False* (*extend-dec-env new-env env*) *s' tops* (*s3*, *r*) **for** *s3 r*
proof

assume *tops*:*evaluate-prog False* (*extend-dec-env new-env env*) *s′ tops* (*s3*, *r*)
then have *r* ≠ *Rerr* (*Rabort Rtimeout-error*)
using *prog-unclocked* **by** *fastforce*
moreover from *top0* **have** *evaluate-top False env s top0* (*update-clock* (λ-. *clock s*) *s′*, *Rval new-env*)
unfolding *Rval* **using** *top-unclocked-ignore*
by (*metis* (*full-types*) *result.distinct*(*1*) *state.record-simps*(*7*))
ultimately show *False*
using *prog-unclocked-ignore*[*rule-format*] *Cons.prems evaluate-prog.cons1 tops* **by** *fastforce*
qed
then show *?thesis*
using *Cons.IH*[*simplified*] *evaluate-prog.cons1 top0* **unfolding** *Rval*
by (*metis combine-dec-result.simps*(*1*) *snd-conv*)
next
case (*Rerr err*)
have *err* = *Rabort Rtimeout-error*
using *Cons top0 top-unclocked-ignore* **unfolding** *Rerr*
by (*metis evaluate-prog.cons2 result.inject*(*2*) *state.record-simps*(*7*))
then show *?thesis*
using *top0* **unfolding** *Rerr*
by (*meson evaluate-prog.cons2 snd-conv*)
qed
qed

lemma *not-evaluate-whole-prog-timeout*:
assumes ∀ *res*. ¬*evaluate-whole-prog False env s prog res*
shows ∃ *r*. *evaluate-whole-prog True env s prog r* ∧ *snd r* = *Rerr* (*Rabort Rtimeout-error*) (**is** *?P*)
proof −
show *?P*
apply (*cases no-dup-mods prog* (*defined-mods s*))
apply (*cases no-dup-top-types prog* (*defined-types s*))
using *not-evaluate-prog-timeout assms* **by** *fastforce+*
qed

lemma *prog-add-to-counter*:
evaluate-prog ck env s prog res ⟹ ∀ *s′ r extra*. *res* = (*s′*,*r*) ∧ *ck* = *True* ∧ *r* ≠ *Rerr* (*Rabort Rtimeout-error*) ⟶
evaluate-prog True env (*s* (| *clock* := (*clock s*) + *extra* |)) *prog* ((*s′* (| *clock* := (*clock s′*) + *extra* |)),*r*)
by (*induction rule*:*evaluate-prog.inducts*) (*auto intro*!:*evaluate-prog.intros dest*:*top-add-to-counter*)

lemma *prog-sub-from-counter*:
evaluate-prog ck env s prog res ⟹
∀ *extra cnt cnt′ s′ r*.
(*clock s*) = *extra* + *cnt* ∧ (*clock s′*) = *extra* + *cnt′* ∧ *res* = (*s′*,*r*) ∧ *ck* = *True* ⟶
evaluate-prog ck env (*s* (| *clock* := *cnt* |)) *prog* ((*s′* (| *clock* := *cnt′* |)),*r*)

proof (*induction rule*:*evaluate-prog.inducts*)
  case (*cons1 ck s1 s2 s3 env top0 tops new-env r*)
  then show *?case*
    apply (*auto*)
    subgoal for *extra*
      apply (*subgoal-tac clock s2* $\geq$ *extra*)
       apply (*drule-tac x*=*extra* **in** *spec*)
       apply (*drule-tac x*=(*clock s2*) − *extra* **in** *spec*)
       apply *rule*+
      by (*auto simp add*:*top-sub-from-counter dest*:*prog-clock-monotone*)
    done
qed (*auto intro*!:*evaluate-prog.intros simp add*:*top-sub-from-counter*)

lemma *prog-clocked-min-counter*:
  assumes *evaluate-prog True env s prog* (s', *r*)
  shows *evaluate-prog True env* (*s* (| *clock* := (*clock s*) − (*clock* s') |)) *prog* (((s')
(| *clock* := *0* |)), *r*)
using *assms*
apply −
apply (*frule prog-clock-monotone*)
using *prog-sub-from-counter* **by** *force*+

lemma *prog-add-clock*:
  evaluate-prog False env s prog (s', *res*) $\Longrightarrow \exists$ *c*. *evaluate-prog True env* (*s* (| *clock*
:= *c* |)) *prog* ((s' (| *clock* := *0* |)),*res*)
proof (*induction False env s prog* s' *res rule*: *evaluate-prog.induct*[*split-format*(*complete*)])
  case *cons1*
  then show *?case*
    apply *auto*
    apply (*drule top-add-clock*)
    apply *auto*
    subgoal for *c* c'
      apply (*drule top-add-to-counter*[**where** *extra* = *c*])
      by (*auto simp add*:*add.commute intro*: *evaluate-prog.intros*)
    done
qed (*auto intro*: *evaluate-prog.intros dest*: *top-add-clock*)

lemma *prog-clocked-unclocked-equiv*:
  evaluate-prog False env s prog (s',*r*) =
    ($\exists$ *c*. *evaluate-prog True env* (*s* (| *clock* := *c* |)) *prog* ((s' (| *clock* := *0* |)),*r*) $\wedge$
        r $\neq$ *Rerr* (*Rabort Rtimeout-error*) $\wedge$ (*clock s*) = (*clock* s')) (**is** *?lhs* =
?rhs)
proof *rule*
  assume *?lhs*
  then show *?rhs*
    using *prog-add-clock*
    by (*fastforce simp*: *prog-unclocked*)
next
  assume *?rhs*

then show *?lhs*
apply (*auto simp*: *prog-unclocked*)

proof −
fix *c* :: *nat*
assume *a1*: *evaluate-prog True env* (*update-clock* (λ-. *c*) *s*) *prog* (*update-clock* (λ-. *0*) *s'*, *r*)
assume *a2*: *r* ≠ *Rerr* (*Rabort Rtimeout-error*)
assume *a3*: *clock s* = *clock s'*
have ∀ *n. evaluate-prog False env* (*update-clock* (λ*na. n*) *s*) *prog* (*update-clock* (λ*na. n*) *s'*, *r*)
using *a2 a1 prog-unclocked-ignore*
by *fastforce*
then show *?thesis*
using *a3* **by** (*metis* (*no-types*) *state.record-simps*(*7*))
qed
qed

end

lemma *clocked-evaluate*:
(∃ *k. BigStep.evaluate True env* (*update-clock* (λ-. *k*) *s*) *e* (*s'*, *r*) ∧ *r* ≠ *Rerr* (*Rabort Rtimeout-error*)) =
(∃ *k. BigStep.evaluate True env* (*update-clock* (λ-. *k*) *s*) *e* ((*update-clock* (λ-. *0*) *s'*), *r*) ∧ *r* ≠ *Rerr* (*Rabort Rtimeout-error*))
apply *auto*
apply (*frule clock-monotone*)
subgoal for *k*
by (*force dest*: *sub-from-counter*(*3*)[*rule-format*, **where** *count'* = *0* **and** *count* = *k* − (*clock s'*)])
by (*force dest*: *add-to-counter*[**where** *extra* = *clock s'*])

end

22.6 An even simpler version without mutual induction

theory *Big-Step-Unclocked-Single*
imports *Big-Step-Unclocked Big-Step-Clocked Evaluate-Single Big-Step-Fun-Equiv*
begin

inductive *evaluate-list* ::
(*'ffi state* ⇒ *exp* ⇒ *'ffi state*∗(*v*,*v*) *result* ⇒ *bool*) ⇒
'ffi state ⇒ *exp list* ⇒ *'ffi state*∗(*v list*, *v*)*result* ⇒ *bool* **for** *P* **where**
empty:
evaluate-list P s [] (*s*,*Rval* []) |

cons1:

P s1 e (*s2*, *Rval v*) $\Longrightarrow$
evaluate-list P s2 es (*s3*, *Rval vs*) $\Longrightarrow$
evaluate-list P s1 (*e*#*es*) (*s3*, *Rval* (*v*#*vs*)) |

cons2:
P s1 e (*s2*, *Rerr err*) $\Longrightarrow$
evaluate-list P s1 (*e*#*es*) (*s2*, *Rerr err*) |

cons3:
P s1 e (*s2*, *Rval v*) $\Longrightarrow$
evaluate-list P s2 es (*s3*, *Rerr err*) $\Longrightarrow$
evaluate-list P s1 (*e*#*es*) (*s3*, *Rerr err*)

lemma *evaluate-list-mono-strong*[*intro?*]:
assumes *evaluate-list R s es r*
assumes $\bigwedge$*s e r*. *e* $\in$ *set es* $\Longrightarrow$ *R s e r* $\Longrightarrow$ *Q s e r*
shows *evaluate-list Q s es r*
using *assms* **by** (*induction*; *fastforce intro*: *evaluate-list.intros*)

lemma *evaluate-list-mono*[*mono*]:
assumes *R* $\leq$ *Q*
shows *evaluate-list R* $\leq$ *evaluate-list Q*
using *assms* **unfolding** *le-fun-def le-bool-def*
by (*metis evaluate-list-mono-strong*)

inductive *evaluate* :: *v sem-env* $\Rightarrow$ *′ffi state* $\Rightarrow$ *exp* $\Rightarrow$ *′ffi state*∗(*v*,*v*) *result* $\Rightarrow$ *bool*
where

lit:
evaluate env s (*Lit l*) (*s*, *Rval* (*Litv l*)) |

raise1:
evaluate env s1 e (*s2*, *Rval v*) $\Longrightarrow$
evaluate env s1 (*Raise e*) (*s2*, *Rerr* (*Rraise v*)) |

raise2:
evaluate env s1 e (*s2*, *Rerr err*) $\Longrightarrow$
evaluate env s1 (*Raise e*) (*s2*, *Rerr err*) |

handle1:
evaluate env s1 e (*s2*, *Rval v*) $\Longrightarrow$
evaluate env s1 (*Handle e pes*) (*s2*, *Rval v*) |

handle2:
evaluate env s1 e (*s2*, *Rerr* (*Rraise v*)) $\Longrightarrow$
match-result env s2 v pes v = *Rval* (*e′*, *env′*) $\Longrightarrow$
evaluate (*env* (| *sem-env.v* := *nsAppend* (*alist-to-ns env′*) (*sem-env.v env*) |)) *s2*
e′ bv $\Longrightarrow$
evaluate env s1 (*Handle e pes*) *bv* |

handle2b:
  evaluate env s1 e (*s2*, *Rerr* (*Rraise v*)) ⟹
   match-result env s2 v pes v = *Rerr err* ⟹
   evaluate env s1 (*Handle e pes*) (*s2*, *Rerr err*) |

handle3:
  evaluate env s1 e (*s2*, *Rerr* (*Rabort a*)) ⟹
   evaluate env s1 (*Handle e pes*) (*s2*, *Rerr* (*Rabort a*)) |

con1:
  do-con-check (*c env*) *cn* (*length es*) ⟹
   build-conv (*c env*) *cn* (*rev vs*) = *Some v* ⟹
   evaluate-list (*evaluate env*) *s1* (*rev es*) (*s2*, *Rval vs*) ⟹
   evaluate env s1 (*Con cn es*) (*s2*, *Rval v*) |

con2:
  ¬(*do-con-check* (*c env*) *cn* (*length es*)) ⟹
   evaluate env s (*Con cn es*) (*s*, *Rerr* (*Rabort Rtype-error*)) |

con3:
  do-con-check (*c env*) *cn* (*length es*) ⟹
   evaluate-list (*evaluate env*) *s1* (*rev es*) (*s2*, *Rerr err*) ⟹
   evaluate env s1 (*Con cn es*) (*s2*, *Rerr err*) |

var1:
  nsLookup (*sem-env.v env*) *n* = *Some v* ⟹
   evaluate env s (*Var n*) (*s*, *Rval v*) |

var2:
  nsLookup (*sem-env.v env*) *n* = *None* ⟹
   evaluate env s (*Var n*) (*s*, *Rerr* (*Rabort Rtype-error*)) |

fn:
  evaluate env s (*Fun n e*) (*s*, *Rval* (*Closure env n e*)) |

app1:
  evaluate-list (*evaluate env*) *s1* (*rev es*) (*s2*, *Rval vs*) ⟹
   do-opapp (*rev vs*) = *Some* (*env′*, *e*) ⟹
   evaluate env′ s2 e bv ⟹
   evaluate env s1 (*App Opapp es*) *bv* |

app3:
  evaluate-list (*evaluate env*) *s1* (*rev es*) (*s2*, *Rval vs*) ⟹
   (*do-opapp* (*rev vs*) = *None*) ⟹
   evaluate env s1 (*App Opapp es*) (*s2*, *Rerr* (*Rabort Rtype-error*)) |

app4:
  evaluate-list (*evaluate env*) *s1* (*rev es*) (*s2*, *Rval vs*) ⟹

do-app (refs s2, ffi s2) op0 (rev vs) = Some ((refs′,ffi′), res) $\Longrightarrow$
op0 $\neq$ Opapp $\Longrightarrow$
evaluate env s1 (App op0 es) (s2 ⦇refs:=refs′,ffi:=ffi′⦈, res) |

app5:
evaluate-list (evaluate env) s1 (rev es) (s2, Rval vs) $\Longrightarrow$
do-app (refs s2, ffi s2) op0 (rev vs) = None $\Longrightarrow$
op0 $\neq$ Opapp $\Longrightarrow$
evaluate env s1 (App op0 es) (s2, Rerr (Rabort Rtype-error)) |

app6:
evaluate-list (evaluate env) s1 (rev es) (s2, Rerr err) $\Longrightarrow$
evaluate env s1 (App op0 es) (s2, Rerr err) |

log1:
evaluate env s1 e1 (s2, Rval v1) $\Longrightarrow$
do-log op0 v1 e2 = Some (Exp e′) $\Longrightarrow$
evaluate env s2 e′ bv $\Longrightarrow$
evaluate env s1 (Log op0 e1 e2) bv |

log2:
evaluate env s1 e1 (s2, Rval v1) $\Longrightarrow$
(do-log op0 v1 e2 = Some (Val bv)) $\Longrightarrow$
evaluate env s1 (Log op0 e1 e2) (s2, Rval bv) |

log3:
evaluate env s1 e1 (s2, Rval v1) $\Longrightarrow$
(do-log op0 v1 e2 = None) $\Longrightarrow$
evaluate env s1 (Log op0 e1 e2) (s2, Rerr (Rabort Rtype-error)) |

log4:
evaluate env s e1 (s′, Rerr err) $\Longrightarrow$
evaluate env s (Log op0 e1 e2) (s′, Rerr err) |

if1:
evaluate env s1 e1 (s2, Rval v1) $\Longrightarrow$
do-if v1 e2 e3 = Some e′ $\Longrightarrow$
evaluate env s2 e′ bv $\Longrightarrow$
evaluate env s1 (If e1 e2 e3) bv |

if2:
evaluate env s1 e1 (s2, Rval v1) $\Longrightarrow$
(do-if v1 e2 e3 = None) $\Longrightarrow$
evaluate env s1 (If e1 e2 e3) (s2, Rerr (Rabort Rtype-error)) |

if3:
evaluate env s e1 (s′, Rerr err) $\Longrightarrow$
evaluate env s (If e1 e2 e3) (s′, Rerr err) |

mat1:
  evaluate env s1 e (*s2*, *Rval v1*) $\Longrightarrow$
   match-result env s2 v1 pes Bindv = *Rval* (*e'*, *env'*) $\Longrightarrow$
   evaluate (*env* ⦇ *sem-env.v* := *nsAppend* (*alist-to-ns env'*) (*sem-env.v env*) ⦈) *s2 e' bv* $\Longrightarrow$
   evaluate env s1 (*Mat e pes*) *bv* |

mat1b:
  evaluate env s1 e (*s2*, *Rval v1*) $\Longrightarrow$
   match-result env s2 v1 pes Bindv = *Rerr err* $\Longrightarrow$
   evaluate env s1 (*Mat e pes*) (*s2*, *Rerr err*) |

mat2:
  evaluate env s e (*s'*, *Rerr err*) $\Longrightarrow$
   evaluate env s (*Mat e pes*) (*s'*, *Rerr err*) |

let1:
  evaluate env s1 e1 (*s2*, *Rval v1*) $\Longrightarrow$
   evaluate (*env* ⦇ *sem-env.v* := (*nsOptBind n v1*(*sem-env.v env*)) ⦈) *s2 e2 bv* $\Longrightarrow$
   evaluate env s1 (*Let n e1 e2*) *bv* |

let2:
  evaluate env s e1 (*s'*, *Rerr err*) $\Longrightarrow$
   evaluate env s (*Let n e1 e2*) (*s'*, *Rerr err*) |

letrec1:
  distinct (*List.map* (λx .
     (*case x of* (*x*,*y*,*z*) => *x*)) *funs*) $\Longrightarrow$
   evaluate (*env* ⦇ *sem-env.v* := (*build-rec-env funs env*(*sem-env.v env*)) ⦈) *s e bv* $\Longrightarrow$
   evaluate env s (*Letrec funs e*) *bv* |

letrec2:
  ¬ (*distinct* (*List.map* (λx .
     (*case x of* (*x*,*y*,*z*) => *x*)) *funs*)) $\Longrightarrow$
   evaluate env s (*Letrec funs e*) (*s*, *Rerr* (*Rabort Rtype-error*)) |

tannot:
  evaluate env s e bv $\Longrightarrow$
   evaluate env s (*Tannot e t0*) *bv* |

locannot:
  evaluate env s e bv $\Longrightarrow$
   evaluate env s (*Lannot e l*) *bv*

lemma *unclocked-single-list-sound*:
 evaluate-list (*Big-Step-Unclocked.evaluate v*) *s es bv* $\Longrightarrow$ *Big-Step-Unclocked.evaluate-list v s es bv*
by (*induction rule*: *evaluate-list.induct*) (*auto intro*: *evaluate-list-evaluate.intros*)

lemma *unclocked-single-sound*:
evaluate v s e bv $\Longrightarrow$ *Big-Step-Unclocked.evaluate v s e bv*
by (*induction rule*:*evaluate.induct*)
(*auto simp del*: *do-app.simps intro*: *Big-Step-Unclocked.evaluate-list-evaluate.intros unclocked-single-list-sound*
evaluate-list-mono-strong)

lemma *unclocked-single-complete*:
Big-Step-Unclocked.evaluate-list v s es bv1 $\Longrightarrow$ *evaluate-list* (*evaluate v*) *s es bv1*
Big-Step-Unclocked.evaluate v s e bv2 $\Longrightarrow$ *evaluate v s e bv2*
by (*induction rule*: *evaluate-list-evaluate.inducts*)
(*auto intro*: *evaluate.intros evaluate-list.intros*)

corollary *unclocked-single-eq*:
evaluate = *Big-Step-Unclocked.evaluate*
by (*rule ext*)+ (*metis unclocked-single-sound unclocked-single-complete*)

corollary *unclocked-single-eq'*:
evaluate = *BigStep.evaluate False*
by (*simp add*: *unclocked-single-eq unclocked-eq*)

corollary *unclocked-single-determ*:
evaluate env s e r3a $\Longrightarrow$ *evaluate env s e r3b* $\Longrightarrow$ *r3a* = *r3b*
by (*metis unclocked-single-eq unclocked-determ*)

lemma *unclocked-single-fun-eq*:
(($\exists$ *k*. *Evaluate-Single.evaluate env* (*s* ⦇ *clock*:= *k* ⦈) *e* = (*s'*, *r*)) $\wedge$ *r* $\neq$ *Rerr* (*Rabort Rtimeout-error*) $\wedge$ (*clock s*) = (*clock s'*)) =
evaluate env s e (*s'*,*r*)
apply (*subst fun-evaluate-equiv'*)
apply (*subst unclocked-single-eq*)
apply (*subst unclocked-eq*)
apply (*subst fun.evaluate-iff-sym*(*1*)[*symmetric*])
apply (*subst big-clocked-unclocked-equiv*)
using *clocked-evaluate* **by** *metis*

end

Chapter 23

Matching adaptation

theory *Matching*
imports *Semantic-Extras*
begin

context begin

qualified fun *fold2* **where**
fold2 f err [] [] init = init |
fold2 f err (x # xs) (y # ys) init = fold2 f err xs ys (f x y init) |
fold2 - err - - - = err

qualified lemma *fold2-cong*[*fundef-cong*]:
  assumes *init1 = init2 err1 = err2 xs1 = xs2 ys1 = ys2*
  assumes $\bigwedge$*init x y. x* ∈ *set xs1* ⟹ *y* ∈ *set ys1* ⟹ *f x y init = g x y init*
  shows *fold2 f err1 xs1 ys1 init1 = fold2 g err2 xs2 ys2 init2*
using *assms*
by (*induction f err1 xs1 ys1 init1 arbitrary*: *init2 xs2 ys2 rule*: *fold2.induct*) *auto*

fun *pmatch-single* :: ((*string*),(*string*),(*nat*∗*tid-or-exn*))*namespace* ⇒((*v*)*store-v*)*list*
⇒ *pat* ⇒ *v* ⇒(*string*∗*v*)*list* ⇒((*string*∗*v*)*list*)*match-result* **where**
pmatch-single envC s Pany v′ env = (*Match env*) |
pmatch-single envC s (*Pvar x*) *v′ env* = (*Match* ((*x*,*v′*)# *env*)) |
pmatch-single envC s (*Plit l*) (*Litv l′*) *env* = (
  if l = *l′ then*
    Match env
  else if lit-same-type l l′ then
    No-match
  else
    Match-type-error) |
pmatch-single envC s (*Pcon* (*Some n*) *ps*) (*Conv* (*Some* (*n′*, *t′*)) *vs*) *env* =
  (*case nsLookup envC n of*
    Some (*l*, *t1*) =>
      if same-tid t1 t′ ∧ (*List.length ps* = *l*) *then*
        if same-ctor (*id-to-n n*, *t1*) (*n′*,*t′*) *then*

fold2 (*λp v m. case m of*
Match env ⇒ *pmatch-single envC s p v env*
| *m* ⇒ *m*) *Match-type-error ps vs* (*Match env*)
else
No-match
else
Match-type-error
| *- => Match-type-error*
) |
pmatch-single envC s (*Pcon None ps*) (*Conv None vs*) *env* = (
if List.length ps = *List.length vs then*
fold2 (*λp v m. case m of*
Match env ⇒ *pmatch-single envC s p v env*
| *m* ⇒ *m*)
Match-type-error ps vs (*Match env*)
else
Match-type-error) |
pmatch-single envC s (*Pref p*) (*Loc lnum*) *env* =
(*case store-lookup lnum s of*
Some (*Refv v2*) *=> pmatch-single envC s p v2 env*
| *Some - => Match-type-error*
| *None => Match-type-error*
) |
pmatch-single envC s (*Ptannot p t1*) *v2 env* = *pmatch-single envC s p v2 env* |
pmatch-single envC - - - env = *Match-type-error*

private lemma *pmatch-list-length-neq*:
length vs ≠ *length ps* ⟹ *fold2*(*λp v m. case m of*
Match env ⇒ *pmatch-single cenv s p v env*
| *m* ⇒ *m*) *Match-type-error ps vs m* = *Match-type-error*
by (*induction ps vs arbitrary*:*m rule*:*List.list-induct2′*) *auto*

private lemma *pmatch-list-nomatch*:
length vs = *length ps* ⟹ *fold2*(*λp v m. case m of*
Match env ⇒ *pmatch-single cenv s p v env*
| *m* ⇒ *m*) *Match-type-error ps vs No-match* = *No-match*
by (*induction ps vs rule*:*List.list-induct2′*) *auto*

private lemma *pmatch-list-typerr*:
length vs = *length ps* ⟹ *fold2*(*λp v m. case m of*
Match env ⇒ *pmatch-single cenv s p v env*
| *m* ⇒ *m*) *Match-type-error ps vs Match-type-error* = *Match-type-error*
by (*induction ps vs rule*:*List.list-induct2′*) *auto*

private lemma *pmatch-single-eq0*:
length ps = *length vs* ⟹ *pmatch-list cenv s ps vs env* = *fold2*(*λp v m. case m of*
Match env ⇒ *pmatch-single cenv s p v env*
| *m* ⇒ *m*) *Match-type-error ps vs* (*Match env*)
pmatch cenv s p v0 env = *pmatch-single cenv s p v0 env*

proof (*induction rule*: *pmatch-list-pmatch.induct*)
case (*4 envC s n ps n′ t′ vs env*)
then show *?case*
by (*auto split*:*option.splits match-result.splits dest*!:*pmatch-list-length-neq*[**where** *m* = *Match env* **and** *cenv* = *envC* **and** *s* = *s*])
qed (*auto split*:*option.splits match-result.splits store-v.splits simp*:*pmatch-list-nomatch pmatch-list-typerr*)

lemma *pmatch-single-equiv*: *pmatch* = *pmatch-single*
by (*rule ext*)+ (*simp add*: *pmatch-single-eq0*)

end

export-code *pmatch-single* **checking** *SML*

end

Chapter 24

Code generation

theory *CakeML-Code*
imports
  Evaluate-Single
  Matching
  generated/CakeML/PrimTypes
begin

hide-const (**open**) *Lib.the*

declare *evaluate-list-eq*[*code-unfold*]
declare *fix-clock-evaluate*[*code-unfold*]
declare *fun-evaluate-equiv*[*code*]
declare *pmatch-single-equiv*[*code*]

declare [[*code abort*: *failwith fp64-negate fp64-sqrt fp64-sub fp64-mul fp64-div fp64-add fp64-abs*]]

definition *empty-ffi-state* :: *unit ffi-state* **where**
empty-ffi-state = *initial-ffi-state* (λ- - - -. *Oracle-fail*) ()

context begin

private definition *prim-sem-res* **where**
prim-sem-res = *Option.the* (*prim-sem-env empty-ffi-state*)

local-setup ‹*fn lthy =>*
  let
    val thm = Code-Simp.dynamic-conv lthy @{*cterm prim-sem-res*}
    val (-, *lthy′*) = *Local-Theory.note* ((@{*binding prim-sem-res-code*}, @{*attributes* [*code*]}), [*thm*]) *lthy*
  in lthy′ end
›

value [*simp*] *prim-sem-res*

definition *prim-sem-env* **where** *prim-sem-env* = *snd prim-sem-res*
definition *prim-sem-state* **where** *prim-sem-state* = *fst prim-sem-res*

end

export-code *evaluate fun-evaluate fun-evaluate-prog prim-sem-env*
 checking *SML*

— Test
lemma *snd* (*evaluate prim-sem-env prim-sem-state* (*Lit* (*IntLit 1*))) = *Rval* (*Litv* (*IntLit 1*))
 by *simp*

end

Chapter 25

Quickcheck setup (fishy)

theory *CakeML-Quickcheck*
imports
generated/CakeML/SemanticPrimitives
begin

datatype-compat *namespace*
datatype-compat *t*
datatype-compat *pat*
datatype-compat *sem-env*

context begin

qualified definition *handle-0* **where**
handle-0 n = *Handle n* []

qualified definition *handle-1* **where**
handle-1 n p1 e1 = *Handle n* [(*p1*, *e1*)]

qualified definition *handle-2* **where**
handle-2 n p1 e1 p2 e2 = *Handle n* [(*p1*, *e1*), (*p2*, *e2*)]

qualified definition *con-0* **where**
con-0 n = *Con n* []

qualified definition *con-1* **where**
con-1 n e1 = *Con n* [*e1*]

qualified definition *con-2* **where**
con-2 n e1 e2 = *Con n* [*e1*, *e2*]

qualified definition *app-0* **where**
app-0 n = *App n* []

qualified definition *app-1* **where**

app-1 n e1 = *App n* [*e1*]

qualified definition *app-2* **where**
app-2 n e1 e2 = *App n* [*e1*, *e2*]

qualified definition *mat-0* **where**
mat-0 n = *Mat n* []

qualified definition *mat-1* **where**
mat-1 n p1 e1 = *Mat n* [(*p1*, *e1*)]

qualified definition *mat-2* **where**
mat-2 n p1 e1 p2 e2 = *Mat n* [(*p1*, *e1*), (*p2*, *e2*)]

qualified definition *conv-0* **where**
conv-0 n = *Conv n* []

qualified definition *conv-1* **where**
conv-1 n v1 = *Conv n* [*v1*]

qualified definition *conv-2* **where**
conv-2 n v1 v2 = *Conv n* [*v1*, *v2*]

qualified definition *closure-dummy* **where**
closure-dummy es var = *Closure* (| *v* = *Bind* [] [], *c* = *Bind* [] [] |) *es var*

qualified definition *recclosure-dummy* **where**
recclosure-dummy es var = *Recclosure* (| *v* = *Bind* [] [], *c* = *Bind* [] [] |) *es var*

qualified definition *vectorv-0* **where**
vectorv-0 = *Vectorv* []

qualified definition *vectorv-1* **where**
vectorv-1 v1 = *Vectorv* [*v1*]

qualified definition *vectorv-2* **where**
vectorv-2 v1 v2 = *Vectorv* [*v1*, *v2*]

end

quickcheck-generator *exp0*
  constructors:
    Raise,
    CakeML-Quickcheck.handle-0,
    CakeML-Quickcheck.handle-1,
    CakeML-Quickcheck.handle-2,
    Lit,
    CakeML-Quickcheck.con-0,
    CakeML-Quickcheck.con-1,

CakeML-Quickcheck.con-2,
Var,
Fun,
CakeML-Quickcheck.app-0,
CakeML-Quickcheck.app-1,
CakeML-Quickcheck.app-2,
Log,
If,
CakeML-Quickcheck.mat-0,
CakeML-Quickcheck.mat-1,
CakeML-Quickcheck.mat-2,
Let,
Tannot,
Lannot

quickcheck-generator *v*
constructors:
Litv,
CakeML-Quickcheck.conv-0,
CakeML-Quickcheck.conv-1,
CakeML-Quickcheck.conv-2,
CakeML-Quickcheck.closure-dummy,
CakeML-Quickcheck.recclosure-dummy,
Loc,
CakeML-Quickcheck.vectorv-0,
CakeML-Quickcheck.vectorv-1,
CakeML-Quickcheck.vectorv-2

quickcheck-generator *dec*
constructors: *Dlet*, *Dletrec*, *Dtype*, *Dtabbrev*, *Dexn*

lemma
fixes *t* :: *dec*
shows $t \neq t$
quickcheck [*expect* = *counterexample*, *timeout* = *90*]
quickcheck [*random*, *expect* = *counterexample*, *timeout* = *90*]
oops

lemma
fixes *t* :: *v*
shows $t \neq t$
quickcheck [*expect* = *counterexample*, *timeout* = *90*]
quickcheck [*random*, *expect* = *counterexample*, *timeout* = *90*]
oops

end

Chapter 26

CakeML Compiler

theory *CakeML-Compiler*
imports
  generated/CakeML/Ast
  Show.Show-Instances
keywords *cakeml* :: *diag*
begin

hide-const (**open**) *Lem-string.concat*

ML-file ‹*Tools/cakeml-sexp.ML*›
ML-file ‹*Tools/cakeml-compiler.ML*›

end