A Mechanized Theory of Communication Analysis in CML

March 4, 2019

Concurrent ML

- extension of Standard ML
- concurrency and synchronization
- synchronized communication over channels: send event, receive event
- composition of events: choose event, wrap event ...

Concurrent ML

```
type thread_id
val spawn : (unit -> unit) -> thread_id

type 'a chan
val channel : unit -> 'a chan

type 'a event
val sync: 'a event -> 'a

val recvEvt: 'a chan -> 'a event
val sendEvt: 'a channel * 'a -> unit event

val send: 'a chan * 'a -> unit
fun send (ch, v) = sync (sendEvt (ch, v))

val recv: 'a chan -> 'a
fun recv ch = sync (recvEvt ch)
```

Concurrent ML

```
structure Serv : SERV =
struct
  datatype serv = S of (int * int chan)
     chan

  fun make () =
  let
    val reqCh = channel ()
    fun loop state =
    let
      val (v, replCh) = recv reqCh
      val () = send (replCh, state)
    in
      loop v
    end
    val () = spawn (fn () => loop 0)
  in
    S reqCh
  end

  fun call (server, v) =
  let
    val S reqCh = server
    val replCh = channel ()
    val () = send (reqCh, (v, replCh))
  in
    recv replCh
  end
end

signature SERV =
sig
  type serv
  val make : unit -> serv
  val call : serv * int -> int
end
```

Isabelle/HOL

- interactive theorem proving assistant; proof assistant
- unification and rewriting
- simply typed terms
- propositions as boolean typed terms
- higher order terms
- computable functions
- inductive data
- inductive reasoning
- tactics and composition

Isabelle/HOL

```
⊢ P1 ∨ P2 ⟶ Q
proof
  assume P1 ∨ P2:
    case P1:
      have ⊢ P1 ⟶ Q by A
      have ⊢ Q by modus ponens
    case P2:
      have ⊢ P2 ⟶ Q by B
      have ⊢ Q by modus ponens
    have P1 ⊢ Q, P2 ⊢ Q
    have ⊢ Q by disjunction elimination
  have P1 ∨ P2 ⊢ Q
  have ⊢ P1 ∨ P2 ⟶ Q
    by implication introduction
qed
```

```
⊢ P1 ∨ P2 ⟶ Q
apply (rule impI)
  P1 ∨ P2 ⊢ Q
apply (erule disjE)
  P1 ⊢ Q
∗ P2 ⊢ Q
apply (insert A)
  P1, P1 ⟶ Q ⊢ Q
∗ P2 ⊢ Q
apply (erule mp)
  P1 ⊢ P1
∗ P2 ⊢ Q
apply assumption
  P2 ⊢ Q
apply (insert B)
  P2, P2 ⟶ Q ⊢ Q
apply (erule mp)
  P2 ⊢ P2
apply assumption
done
```

Isabelle/HOL

```
datatype nat = Z | S nat

predicate lte: nat -> nat -> bool where
  eq: n .
  ⊢ lte n n
* lt: n1 n2 .
    lte n1 n2
  ⊢ lte n1 (S n2)
```

```
datatype 'a list =
  Nil
| Cons 'a ('a list)

predicate sorted:
  ('a -> 'a -> bool) -> 'a list -> bool
where
  nil: r .
  ⊢ sorted r Nil
* uni: r x .
  ⊢ sorted r (Cons x Nil)
* cons: r x y ys .
    r x y,
    sorted r (Cons y ys)
  ⊢ sorted r (Cons x (Cons y ys))
```

Isabelle/HOL

```
⊢ sorted lte (Cons (Z) (Cons (S Z) (Cons (S Z) (Cons (S (S (S Z))) Nil))))
apply (rule cons)
  ⊢ lte Z (S Z)
* ⊢ sorted lte (Cons (S Z) (Cons (S Z) (Cons (S (S (S Z))) Nil)))
apply (rule lt)
  ⊢ lte Z Z
* ⊢ sorted lte (Cons (S Z) (Cons (S Z) (Cons (S (S (S Z))) Nil)))
apply (rule eq)
  ⊢ sorted lte (Cons (S Z) (Cons (S Z) (Cons (S (S (S Z))) Nil)))
apply (rule cons)
  ⊢ lte (S Z) (S Z)
* ⊢ sorted lte (Cons (S Z) (Cons (S (S (S Z))) Nil))
apply (rule eq)
  ⊢ sorted lte (Cons (S Z) (Cons (S (S (S Z))) Nil))
apply (rule cons)
  ⊢ lte (S Z) (S (S (S Z)))
* ⊢ sorted lte (Cons (S (S (S Z))) Nil)
apply (rule lt)
  ⊢ lte (S Z) (S (S Z))
* ⊢ sorted lte (Cons (S (S (S Z))) Nil)
apply (rule lt)
  ⊢ lte (S Z) (S Z)
* ⊢ sorted lte (Cons (S (S (S Z))) Nil)
apply (rule eq)
  ⊢ sorted lte (Cons (S (S (S Z))) Nil)
apply (rule uni)
done
```

Analysis

- communication classification: one-shot, one-to-many, many-to-one, many-to-many
- control flow analysis
- channel liveness
- algorithm vs constraints
- structural recursion vs fixpoint accumulation
- performance improvements
- safety

Synchronization

- uniprocessor; dispatch scheduling
- multiprocessor; mutex and compare-and-swap
- synchronization state
- sender and receiver thread containers
- message containers

Syntax

```
datatype name = Nm string

datatype term =
  Bind name complex term
| Rslt name

and complex =
  Unt
| MkChn
| Atom atom
| Spwn term
| Sync name
| Fst name
| Snd name
| Case name name term name term
| App name name

and atom =
  SendEvt name name
| RecvEvt name
| Pair name name
| Lft name
| Rht name
| Fun name name term
```

Dynamic Semantics

```
datatype dynamic_step =
  DSeq name
| DSpwn name
| DCll name
| DRtn name

type dynamic_path =
  dynamic_step list

datatype chan =
  Chan dynamic_path name

datatype dynamic_value =
  VUnt
| VChn chan
| VAtm atom (name -> dynamic_value option)

type environment =
  name -> dynamic_value option
```

Dynamic Semantics

```
predicate seqEval: complex -> environment -> dynamic_value -> bool where

  unit: env .
  ⊢ seqEval Unt env VUnt

* atom: a env .
  ⊢ seqEval (Atom a) env (VAtm a env)

* first: env n_p n_1 n_2 env_p v .
    env n_p = Some (VAtm (Pair n_1 n_2) env_p),
    env_p n_1 = Some v
  ⊢ seqEval (Fst n_p) env v

* second: env n_p n_1 n_2 env_p v .
    env n_p = Some (VAtm (Pair n_1 n_2) env_p),
    env_p n_2 = Some v
  ⊢ seqEval (Snd n_p) env v
```

Dynamic Semantics

```
predicate callEval: complex -> env -> term -> env -> bool where

  distincLeft: env n_s n_c env_s v n_l t_l n_r t_r .
    env n_s = Some (VAtm (Lft n_c) env_s),
    env_s n_c = Some v
  ⊢ callEval (Case n_s n_l t_l n_r t_r) env t_l (env(n_l -> v))

* distincRight: env n_s n_c env_s v n_l t_l n_r t_r .
    env n_s = Some (VAtm (Rht n_c) env_s),
    env_s n_c = Some v
  ⊢ callEval (Case n_s n_l t_l n_r t_r) env t_r (env(n_r -> v))

* application: env n_f n_f' n_p t_b env_f n_a v .
    env n_f = Some (VAtm (Fun n_f' n_p t_b) env_f),
    env n_a = Some v
  ⊢ callEval
    (App n_f n_a) env t_b
    (env_f(
      n_f' -> (VAtm (Fun n_f' n_p t_b) env_f),
      n_p -> v
    ))
```

Dynamic Semantics

```
datatype contin = Ctn name tm env

type stack = contin list

datatype state =
  Stt program env stack

type pool =
  dynamic_path -> state option

predicate leaf: pool -> dynamic_path -> bool where
  intro: pool path stt .
    pool path = Some stt,
    (∄ path' stt' .
      pool path' = Some stt',
      strictPrefix path path'
    )
  ⊢ leaf pool path

type corresp = dynamic_path * chan * dynamic_path

type communication = corresp set
```

Dynamic Semantics

```
predicate dynamicEval:
  pool -> communication -> pool -> communication -> bool
where

  return: pool path n env n_k t_k env_k stack' v comm .
    leaf pool path,
    pool path = Some (Stt (Rslt n) env ((Ctn n_k t_k env_k) # stack')),
    env n = Some v
  ⊢ dynamicEval
    pool comm
    (pool(
      path @ [DRtn n] ->
        (Stt t_k env_k(n_k -> v) stack')
    ))
    comm

* seq: pool path n c t' env stack v .
    leaf pool path,
    pool path = Some (Stt (Bind n c t') env stack),
    seqEval c env v
  ⊢ dynamicEval
    pool comm
    (pool(
      path @ [DSeq n] -> (Stt t' (env(n -> v)) stack)
    ))
    comm
```

Dynamic Semantics

```
* call: pool path n c t' env stack t_c env_c comm .
    leaf pool path,
    pool path = Some (Stt (Bind n c t') env stack),
    callEval c env t_c env_c
  ⊢ dynamicEval
    pool comm
    (pool(
      path @ [DCll n] -> (Stt t_c env_c ((Ctn n t' env) # stack))
    )) comm

* makeChan: pool path n t' env stack .
    leaf pool path,
    pool path = Some (Stt (Bind n MkChn t') env stack)
  ⊢ dynamicEval pool comm
    (pool(
      path @ [DSeq n] ->
        (Stt t' (env(n -> (VChn (Chan path n)))) stack)
    )) comm

* spawn: pool path n t_c t' env stack comm .
    leaf pool path,
    pool path = Some (Stt (Bind n (Spwn t_c) t') env stack)
  ⊢ dynamicEval pool comm
    (pool(
      path @ [DSeq n] -> (Stt t' (env(n -> VUnt)) stack),
      path @ [DSpwn n] -> (Stt t_c env [])
    )) comm
```

Dynamic Semantics

```
* sync: pool path_s n_s n_se t_s env_s stack_s n_sc n_m
  env_se path_r n_r n_re t_r env_r stack_r n_rc env_re chan comm .
    leaf pool path_s,
    pool path_s = Some
      (Stt (Bind n_s (Sync n_se) t_s) env_s stack_s),
    env_s n_se = Some
      (VAtm (SendEvt n_sc n_m) env_se),
    leaf pool path_r,
    pool path_r = Some
      (Stt (Bind n_r (Sync n_re) t_r) env_r stack_r),
    env_r n_re = Some
      (VAtm (RecvEvt n_rc) env_re),
    env_se n_sc = Some (VChn chan),
    env_re n_rc = Some (VChn chan),
    env_se n_m = Some v_m
  ⊢ dynamicEval
    pool comm
    (pool(
      path_s @ [DSeq n_s] -> (Stt t_s (env_s(n_s -> VUnt)) stack_s),
      path_r @ [DSeq n_r] -> (Stt t_r (env_r(n_r -> v_m)) stack_r)
    ))
    (comm ∪ {(path_s, chan, path_r)})
```