A Mechanized Theory of Communication Analysis in CML

March 3, 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