Thread and Memory-Safe Programming with CLASS

Luís Caires
Instituto Superior Técnico (U Lisboa) / INESC-ID
luis.caires@tecnico.ulisboa.pt

CLASS is a proof-of-concept general purpose linear programming language, flexibly supporting realistic concurrent programming idioms, and featuring an expressive linear type system ensuring that programs (1) never misuse or leak stateful resources or memory, (2) never deadlock, and (3) always terminate. The design of CLASS and the strong static guarantees of its type system originates in its Linear Logic and proposition-as-types foundations. However, instead of focusing on its theoretical foundations, this paper briefly illustrates, in a tutorial form, an identifiable CLASS session-based programming style where strong correctness properties are automatically ensured by type-checking. Our more challenging examples include concurrent thread and memory-safe mutable ADTs, lazy stream programming, and manipulation of linear digital assets as used in smart contracts.

1 Introduction

The interpretation of linear logic as a session-typed π-calculus [7, 9, 31], capturing full session types [15, 16, 14], has been intensively developed since its proposal. As particular relevant themes for programming language design, we may refer to polymorphism [6, 31], inductive and co-inductive types [27, 29, 23], integration with higher-order functional programming and dependent types [26, 28], or control effects [5]. In [23, 24], we have introduced program constructs inspired by DiLL [13], which allow stateful shared state computation to be expressed, while keeping compatibility with the core ideology of proposition-as-types. We believe that, pretty much like the lambda calculus is considered a canonical typed model for functional sequential computation with pure values, the linear logic typed session calculus may be fairly considered a canonical typed model for linear stateful concurrent computation, also motivating programming language design and implementation. Programming experience with sessions eventually led to the evolution and implementation of CLASS [23, 24, 22], a proof-of-concept general purpose linear programming language, featuring an expressive type system ensuring that programs never misuse or leak resources or memory, never deadlock, and always terminate. Prototype implementations have been and keep being produced, based on a fully concurrent execution model [25], motivating our recent proposal for a fully sequential coroutine-based execution model, the Session Abstract Machine [8, 10].

The strength of static guarantees often comes at the expense of a programming language's expressiveness. We believe that this is not the case for CLASS, which robustly supports many interesting "real-world" programming idioms, involving sessions, higher-order computation, concurrency, and shared mutable state, combined elegantly under a lazy computation discipline. The aim of this paper is thus to illustrate, using examples in a tutorial format, the particular style of CLASS programming, for which all of its rather strong correctness guarantees are automatically ensured at type-checking time.

1.1 Hello World

Since Ritchie [19], every programming language introductory tutorial should start with the "hello world" program. It simply prints out a greeting message and terminates. CLASS programs are collections of process definitions. On the right, we illustrate program execution at the CLASS top level REPL prompt.

Submitted to:
PLACES25

© Luís Caires
This work is licensed under the
Creative Commons Attribution License.

```
proc main() {
  println("hello world "+(2*3));() };;
```

```
> main();;
hello world 6
```

1.2 Basic Linear Session Programming

While the core construct of functional programming is the function, in CLASS every object is a session, where a function is just a particular case of a session. Like functions, sessions are objects that may accept and return values. Unlike functions, sessions can interleave multiple steps of input and output, and offer and exercise choices. Moreover, with sessions there is no asymmetry between caller and callee, a session describes a bidirectional interaction between two symmetric partners, described by dual types. For every session type `A` , there is its dual type `~A`. Typically, we use the positive form `A` for the session side that produces the object, and the negated form `~A` for the session side that consumes the object.

We illustrate with the code for an arithmetic server object and its clients, where process `menu` implements the server body. The protocol of the server is defined by the type `tmenu`.

```
type tmenu {
  offer of {
    | #Dup: recv ~lint;
            send lint; wait
    | #Add: recv ~lint; recv ~lint;
            send lint; wait
}};;
```

```
type dtmenu {
  case of {
    | #Dup: send lint;
            recv ~lint; close
    | #Add: send lint; send lint;
            recv ~lint; close
}};;
```

It offers options `#Dup` and `#Add` as defined by the specific protocol. For `#Dup`, it reads one integer, sends back its double and closes the session, while for `#Add`, it reads two integers, sends back its addition and closes. We give the explicit definition of the dual type `dtmenu` of `tmenu`, but we could have just used `~tmenu`. Notice how the code for each branch option in `menu` complies with the corresponding session type branch, and `recv` and `send` operations implement session input and output.

```
proc menu(m:tmenu) {
  case m of {
    | #Dup: recv m(n);
       send m(2*n);
       wait m; ()
    | #Add: recv m(n1);
       recv m(n2);
       send m(n1+n2);
       wait m; ()
  }
};;
```

```
proc alice0(c:~tmenu) {
  #Dup c; c <- 2; c -> m;
  println("alice got " +m);
  close c
};;

proc main0(){
  cut { menu(s)
        |s:~tmenu|
       alice0(s) }
};;
```

```
proc bob0(c:~tmenu){
  #Add c;
  c <- 4; c <- 3; c -> m;
  println("bob got " +m);
  close c
};;

proc main1(){
  letc s:tmenu { menu(s) };
  bob0(s)
};;
```

Code for `main0` and `main1` exemplify the two usages of the server, one for `#Dup` and other for `#Add`. They essentially launch the server and compose it with client code. Sessions are composed by the `cut` construct. `main0` shows the basic CLASS cut notation from Linear Logic, but in `main1` we use the equivalent sugared `letc` notation. It defines a new interaction point (`s`) and lets the two parts of the code (server and client) interact. There is a fundamental difference between CLASS `letc` binding, and the more familiar **let** binding of functional languages, where the let expression is evaluated, bound to an identifier, and then used in the body. In `letc c:A {P}; Q` the processes `P` and `Q` conceptually execute in in several interaction steps via `c`, in parallel or co-routining, as defined by the session type `A`. Namely, in `main1()`, `s` is used at type `tmenu` in `menu(s)`, but at type `~tmenu` in `bob0(s)`. Notice in `alice0` and `bob0` the sugared notation for send (`send s(2)` / `s <- 2`) and receive (`recv s(r)` / `s -> r`), where the received object is bound to the fresh identifier `r` in the continuation. Unlike in functional languages,

where function parameters always represent “input" values, CLASS process parameters denote general session interaction points, where i/o information flow is defined at a fine grain by the session type.

1.3 Replicated Session Programming

Replicated (non-linear) sessions, defined in CLASS using exponential types, may be shared like regular objects in a language like Python, used an unbounded (possibly zero) number of times. We illustrate here the code for a replicated arithmetic server, callable an unbounded number of times, defined by process `rserver`, and created by `rserver(s)` in `main`. Each time `s` is called (via a client `call s(m)`), a new fresh (linear) session `m` to execute `menu(m)` at server side is spawned. In `main`, the code for `alice` and `bob` call the same replicated server `s` (at type `?~tmenu`), while requesting different operations.

```
proc rserver(sm:!tmenu) {
    !sm(m); menu(m)
};;
```

```
proc bob(; rs:~tmenu) {
  call rs(b); bob0(b)
};;

proc alice(; rs:~tmenu) {
  call rs(a); alice0(a)
};;
```

```
proc main() {
  letc s:!tmenu {
     rserver(s)
  };
  par { alice(;s)
        || bob(;s) }
};;
```

Notice in process definitions the presence of parameters of implicit exponential type `?A`, declared after the ";" in the (optional) exponential context. Exponential parameters are handled non-linearly (cf. `s` in `alice(;s)` and `bob(;s)`); the only available usage for an exponential parameter `r` is `call r(-)`.

1.4 Pure Inductive Data Types and Generics

Session-based programming promotes an interaction based lazy programming style for programs that create and manipulate linear and replicated data types. Using recursive types and generics (universal polymorphism), we may define in CLASS a type for lists of objects of an arbitrary (session) type `A`.

```
type rec List(A){
    choice of {
        |#Nil: close
        |#Cons: pair A; List(A) }
};;

proc nil<A>(l:List(A)){
    #Nil l; close l
};;

proc cons<A>(a:~A, l:~List(A), nl:List(A)){
    #Cons nl; nl <- a; fwd nl l
};;
```

```
proc rec concat<A>( a:~List(A),
                    b:~List(A),
                    ab:List(A)) {
  case a of {
  |#Nil: wait a; fwd b ab
  |#Cons:
    a -> val;
    letc lx:List(A) {
      concat<A>(a,b,lx)
    };
    cons<A>(val,lx,ab)
  }
};;
```

The session `send` type may be seen as a linear pair constructor (cf. in the linear logic semantics, `send` corresponds to the $A \otimes B$ type). This explains our usage of keyword `pair` in the type of `List` as a sugared alias for `send`. Notice how negation / duality ($\sim$) in the parameters types cleanly express "input" parameters (objects consumed, rather than produced). The constructors `nil` and `cons` produce canonical linear `List(A)` values. In the `cons` case, the new list at `l` signals `#Cons`, then exposes the element of type A, and then continues with the tail list `l`, as implemented by forwarding `fwd`. The concat “function" process resembles the code we would write in a functional language, where the linear prefix list `a` is recursively destroyed and reconstructed. However, due to the underlying session execution model, a function like concat is executed lazily, following a concurrent or demand driven co-routing semantics.

1.5 Lazy and Stream-based Computation

We showcase co-recursive and affine types in a famous scenario of lazy computation: Turner's [30] filter network sieve of Eratosthenes, coded in session programming style. We first define the type `AIntStream` of infinite streams of (non-linear) integers; a session of `affine` type may either by used linearly, or discarded (using `drop`). For convenience, we define `CIntStream`, the one step unfolding of `AIntStream`. Processes `intsfm` and `intsfm2` generate respectively the stream of all integers from `k` and from 2.

```
type corec AIntStream {
  affine send !lint; AIntStream
};;

proc rec intsfm(nk: AIntStream; k:~ lint)
{
  affine nk; nk <- k; intsfm(nk;k+1)
};;
```

```
type CIntStream {
  affine send !lint; AIntStream
};;

proc intsfm2(n2: AIntStream)
{
  intsfm(n2;2)
};;
```

Below we define the `filter` and `sieve` processes; `sieve` consumes the stream `sins` $n_0, n_1, \ldots$ of integers, and returns 0 for each non-prime n_i and n_p for each prime n_p (creating a new filter for n_p).

```
proc rec filter(fouts:AIntStream,
                fins:~ CIntStream; n:~ lint)
{
    fins -> v;
    if (v mod n == 0) then {
        affine fouts;
        fouts <- 0;
        filter(fouts,fins;n)
    } else {
       affine fouts;
       fouts <- v;
       filter(fouts, fins; n)
    }
};;
```

```
proc rec sieve(souts:AIntStream,
               sins:~ CIntStream) {
  sins -> p;
  affine souts;
  souts <- p;
  if (p == 0) {
    sieve(souts,sins)
  } else {
    letc outp:AIntStream {
      filter(outp,sins;p)
    };
    sieve(souts,outp)
  }
};;
```

We now present sample driver code, in this case `main_sa(;n)` prints the primes up to `n`.

```
proc primesN(lp:AIntStream)
{
  letc ln:AIntStream { intsfm2(ln) };
  sieve(lp,ln)
};;

proc main_sa(;n:~ lint)
{
  letc lp:AIntStream
    { primesN(lp) };
  print2k(lp;n)
};;
```

```
proc gen_rec print2k(il:~ AIntStream;
                     k :~ lint)
{
  if(k==1) then { println(""); drop il }
  else {
    il -> n;
    if (n==0) {
      print2k(il;k-1)
    } else { print(n+"␣"); print2k(il;k-1)
    }
  }
};;
```

While `filter` and `sieve` are strictly inductively defined processes (termination-safe by typing), we illustrate in `printup2k` the use of general recursion in a "while" type iteration. CLASS allows `gen_rec` as an unsafe escape mechanism, useful to write non-terminating or general recursive programs. Using `gen_rec` we developed a library of well-typed termination-safe iterators and generators (cf. Python's range()), which we prefer not to use here for clarity in showcasing pure CLASS code.

1.6 Shared Mutable State

A key feature of CLASS is safe manipulation of shared linear state [23, 22, 24], supporting behaviourally typed reference cells. These turn out to match a typed version of Concurrent Haskell's MVars [18]. The type `state A` denotes the type of cells holding objects of type `A`, with dual type `usage`∼`A`. Cells are manipulated with `take`, `put` and `drop` operations: `take` moves the cell-stored object to the reader (linear move semantics, emptying the cell), `put` moves an object from the caller into the cell (linear move semantics, filling the cell), and `drop` releases cell reference usage (cf. session `close`), causing the cell to be deallocated when no more references to it exist (our implementation uses reference counting). Types ensure any cell alias is used according to a linear protocol suggested by the regex `(take;put)*;drop`.

```
proc main0m() {
  letc m:state Int {
    cell m(42)
  };
  take m(x);
  println(x);
  put m(x);
  drop m
}
;;
```

```
proc main1m() {
  letc m:state !lint {
    cell m(2)
  };
  share m {
    take m(x); put m(x+1); println(x); drop m
    ||
    take m(x); put m(x-1); println(x); drop m
  }
};;
```

The code `main0m` exemplifies a simple linear usage of a reference cell. The code for `main1m` illustrates sharing of reference cells, made here explicit by the `share` construct. The reference cell `m` is shared by two independent threads, that concurrently interleave cell operations non-deterministically, where shared `take`s require acquiring a cell mutex, to be eventually released by `put`. Hence, `main1m` prints 2 followed by either 3 or 1. Type-checking of `share` (resp. `cut`) ensures that each of the two independent threads involved can at most share one reference cell (resp. session). This discipline is crucial to ensure deadlock absence and does not hinder expressiveness [24]. When all threads using some shared cell drop it, the cell is deallocated (as well as its disposable content, required to be either of `affine` or `state` type). Sharing annotations do not have any special operational meaning, and may be inferred by the CLASS interpreter elaboration phase, but here we manifest in the code all occurrences of sharing via `share`, for clarity. Notice that `share` is not a static scoping construct; cell references may be freely passed around, like any all other session objects, as we exemplify in the code snippets below.

```
type Mint { state lint };;
type HO { send Mint; wait };;

proc sender(s:HO) {
  letc m:Mint { cell m(2) };
  share m {
    s <- m; wait s; ()
    ||
    take m(v); put m(v+1); drop m
  }
};;
```

```
proc receiver(s:~HO) {
  s -> c;
  take c(v);
  println (v);
  put c(0);
  drop c;
  close s
};;
```

```
proc pass() {
  letc s:HO {
    sender(s)
  };
  receiver(s)
};;
```

Program `pass` composes `sender` and `receiver` via a session of type `HO` (standing for HandOver). The `sender` allocates a fresh cell `m` and sends (an alias of) `m` to `receiver`, while locally increments it via the retained alias `m` before dropping it. Concurrently, the `receiver` reads the cell contents, sets it to 0, and drops it. When the program terminates, neither the cell has been leaked, nor any reference to it became dangling, as ensured by construction of the typed CLASS code.

1.7 A Concurrent Barrier for *N* Threads

We implement in CLASS a generic barrier abstraction (see e.g., Rust std::sync::Barrier [20]). The data representation of the barrier object is a (shared) memory cell storing a pair (type `BState`) of an integer counting the number of threads yet to synchronise and a list of waiting thread continuations (we reuse `List(A)` from Section 1.4). We model a thread continuation as a session object of type `Cont`, simply waiting for a `close` signal to run. Process `barrier` creates a new barrier `b` for `nt` threads, auxiliary process `init` sets up the initial state, with the counter set to `nt` and an empty waiting list. The process `awakeall` is called when all threads reach the barrier; it traverses the waiting list `ws` and launches (via `close w`) each pending continuation.

```
type Cont { affine wait };;
type BState
  { pair !lint; affine List(Cont) };;

type Barrier { state BState };;

proc init(rep: BState;n:~lint){
    rep <- n;
    affine rep;
    nil<Cont>(rep)
};;
```

```
proc barrier(b:Barrier;nt:~lint){
    cell b(r. affine r; init(r;nt))
};;

proc rec awakeAll(ws:~List(Cont)){
  case ws of {
    | #Nil : wait ws;()
    | #Cons: recv ws(w);
             close w;
             awakeAll(ws)
    }};;
```

The core of the barrier code is process `bwait`. Any thread about to register in the barrier calls `bwait`, passing its owned shared reference to the barrier object cell at `b`, and its own continuation at `cont`.

```
proc bwait(b:~Barrier, cont:~Cont) {
  take b(ws);
  ws -> n;
  if n==1 then {
    par { awakeAll(ws)
        ||
        put b(nw. affine nw; init(nw;0));
        close cont; release b
    }
  } else {
   letc nw: affine BState {
     affine nw; nw <- n-1;
     affine nw; cons<Cont>(cont,ws,nw) };
     put b(nw); drop b
  }
};;
```

```
proc thread(b:~Barrier; i:~lint) {
  println("thread␣" +i +"␣started.");
  sleep 99; // work before barrier
  letc cont: ~affine wait {
    println("thread␣" +i +"␣on␣wait");
    bwait(b,cont) // call barrier wait
  };
  affine cont;
  wait cont; // wait here
  println("thread␣" +i +"␣wake␣up.");
  sleep 101; // work after barrier
  println("thread␣" +i +"␣terminates.");
  ()
}
;;
```

Each time `bwait` is called, it takes the barrier state (acquiring the mutex). If the calling thread is the last one (`n==1`), it concurrently awakes all waiting threads, while, in parallel, launches the calling thread and drops its ownership of the barrier (for simplicity, we assume the barrier to be single use).

Otherwise, it adds the continuation to the waiting queue (using `cons`), decrements the count of threads that did not reach the barrier yet, and updates the state accordingly using `put b(nw)`, which releases the barrier mutex. We simulate the job of each thread `i` with process `thread`, which does some prior work (`sleep 99`), and calls `bwait` with a continuation to do some after work (`sleep 101`). Notice that some types are declared `affine`; recall that a session of `affine` type may either by used linearly, or discarded, and that values storable in cells are required to be of `affine` or `state` type. We conclude our concurrent barrier example with some client code.

```
proc gen_rec spawnall(b:~Barrier; i:~lint, n:~lint) {
  if n == 0 then { drop b }
  else
    { share b { thread(b;i) || spawnall(b;i+1,n-1) }
  }
};;
```

```
proc mainb(;nt:~lint) {
    letc c: Barrier {
        barrier(c;nt)
    };
    spawnall(c;0,nt)
};;
```

The main program `mainb` creates a new barrier for `nt` threads and concurrently launches the code for each one, calling `spawnall`. All the core code for the barrier above is deemed thread-safe by typing: no deadlocks, livelocks, or memory leaks may arise.

1.8 (Lazy) Mutable Data Structures

We show an implementation for lists of linked memory cells, using a tail sentinel node. Each list element is a memory cell with content as specified by type `ANode(A)` (CLASS annotates the argument type of a `state` type constructor automatically as `affine` except already explicitly typed `affine` or `state`). Notice that definition of types `LList(A)` and `Node(A)` is mutually recursive.

```
type rec LList(A) { state Node(A) }
  and Node(A) { choice of {
                 | #Nil : close
                 | #Next : pair affine A; LList(A)
                }
};;

proc cons<A>(a:~affine A, t:~LList(A), l: ANode(A)){
  affine l; #Next l; l <- a; fwd l t
};;
```

```
type ANode(A) {
    affine Node(A)
};;


proc nil<A>(l: ANode(A)) {
    affine l;
    #Nil l;
    close l
};;
```

The code for `cons` builds at `l` a new `ANode`, pairing a value `a` of type `A` with a reference to the list tail `l`.

```
proc rec concat<A>(a:~LList(A), b:~LList(A), ab: LList(A)){
    take a(node);
    case node of {
        | #Nil: put a(n.nil<A>(n)); fwd b ab
        | #Next: node -> val;
                letc nodeb:LList(A) { concat<A>(node,b,nodeb) };
                put a(node. cons<A>(val,nodeb,node)); fwd a ab
    }
};;
```

This `concat` “function" process resembles the code we would write in an imperative C-like language. Due to the take / put move semantics, the list is traversed recursively by following the references by taking cells on down calls and putting back on returns, until the last node is reached, causing the sentinel to be updated in place. However, due to the lazy session execution model, which uniformly follows a demand driven co-routing semantics, the concatenation of two lists is done in O(1) time, with reconstruction of the possibly shared imperative structure amortised in future transversals. We challenge the reader to wonder what happens when a well typed shared `LList(A)` is concurrently concatenated with other lists.

1.9 An Abstract Data Type of Digital Assets

In this example, we illustrate state encapsulation using behavioral interfaces to code a linear abstract data type representing a leak-free wallet of digital tokens, as used e.g., in a blockchain app. Types

`IWallet(X)` and `Ans(X)` are mutually defined. We pick `List(X)` as representation type (for generic token type `X`), and define the external interface by the co-recursive type `IWallet(X)`.

```
type corec IWallet(X) {
    offer of {
        |#Count: send !lint; IWallet(X)
        |#Add: recv ~X; IWallet(X)
        |#Get: Ans(X)
    }
} and Ans(X) {
    choice of {
        |#Some: send X; IWallet(X)
        |#None: close
    }
};;
```

Notice that "method" `#Count` returns the number of stored tokens, `#Add` adds a new token, and `#Get` extracts a token, if the wallet is empty, `#Get` returns `#None` and the wallet terminates (is disposed).

```
proc rec tokens_imp<A>(tm:IWallet(A),
                       st:~List(A)) {
  case tm of {
  |#Count:
    letc rc: { len<A>(st,rc) };
    rc -> ns; tm <- rc;
    tokens_imp<A>(tm,ns)
  |#Add:
    tm -> val;
    letc ns: { cons<A>(val,st,ns) };
    tokens_imp<A>(tm,ns)
   |#Get:
     case st of {
     |#Nil: wait st; #None tm; close tm
     |#Cons: st -> val; #Some tm; tm <- val;
       tokens_imp<A>(tm,st)
     }
  }
};;
```

```
proc rec len<A>(a:~List(A),
                ao:pair List(A);!lint)
{
...
};;

proc newTokens(tm:IWallet(lstring)) {
    letc s: { nil<lstring>(s) };
    tokens_imp<lstring>(tm,s)
};;

proc test(tk:IWallet(lstring)) {
    letc t: { newTokens(t) };
    #Add t; t <- "NFT@A36D54F89606A";
    #Count t;
    t -> n;
    println ("balance␣=␣"+n);
    fwd t tk
};;
```

The wallet behaviour is implemented by the recursive process `tokens_imp` at session `tm` , at each step it branches on the selected "method", executes the operations and recurses updating the state passed in `st`. We leave as exercise to the reader the definition of the code for procedure `len`: given the linear list `a`, it should return at `ao` the same list and its length (as a `!lint`). Usage by client code of an object of type `IWallet` is only possible via its interface type, as illustrated in `test`. The depicted code ensures that the representation state is never tampered with, and tokens never duplicated, erased or double spent, by linearity and parametricity, since no such capabilities are exported by the ADT.

CLASS type system also supports existential types, which may be used to express more flexible modes of information hiding, but which we are unable to cover in the present brief overview.

1.10 A Mutable Shared Queue

We now address a more challenging concurrent programming exercise, sometimes used as a benchmark for formal verification techniques. We code in CLASS a shared concurrent LIFO queue offering O(1) enqueue and dequeue operations using the mutable linked list data structure of Section 1.8. For simplicity, we assume that the queue stores `Aint` typed values. It implements two separate usage interfaces: one of type `EnqI` (for enqueing) and other of type `DeqI` (for dequeing), encapsulating shared state and code. Both (corecursively typed) views allow clients to call the operations until droping (`#Drop`) the reference.

```
type corec EnqI {
  offer of {
    | #Enq: recv ~affine lint; EnqI
    | #Drop: wait
  }};;
```

```
type corec DeqI {
  offer of {
    | #Deq: pair Opt(Aint); DeqI
    | #Drop: wait
  }};;
```

In the case for #Deq, we return an option type: when the queue is empty, the dequeue operation will return #None (see the the short definition of the Opt(A) type below). The queue representation will maintain two cells of type Ptr, one for the head of the list, for dequeing, and other for the last sentinel (empty) list element (for enqueuing). Recall that the list will always contains a "dummy" sentinel element at the tail. Each list node will be an object of type state Node(lint).

```
type Aint
  { affine lint };;

type Opt(A) {
  affine choice of {
    | #None : close
    | #Some : A
  }
};;
```

```
proc None(o: Opt(Aint)) {
  affine o;
  #None o;close o
};;

proc Some(val:~Aint,
          o:Opt(Aint)) {
  affine o;
  #Some o; fwd val o
};;
```

```
type Ptr
  { state
    state Node(lint) };;

proc free(p:~statel
            Node(lint))
{
  put p(c. nil<lint>(c));
  drop p
};;
```

The code for deq accesses the contents lh of the cell hp at the head, and inspects it. If set to #Nil, it is the sentinel node (empty queue); the content of hp is reset, and #None returned (the sugared closure notation rv <- {r.Node(r)} represents send rv {r.Node(r)};). The code for enq allocates a fresh sentinel node nn. It then stores the value to enqueue v and the reference to nn at the sentinel node sn (using cons, and an update in place). It also stores the reference nn as the new sentinel node in tl. At the end, nn is shared by the #Next field of the last queue node and by the tail cell tl. Notice the crucial use of share, ensuring safe dynamic sharing of state between the interfering access paths from queue head and tail.

```
proc deq(hd:~Ptr, rv:pair Opt(Aint);Ptr) {
  take hd(hp);
  take hp(lh);
  case lh of {
    |#Nil : wait lh;
          put hp(c. nil<lint>(c));
          put hd(hp);
          rv <- { r. None(r) }; fwd hd rv
    |#Next :
          recv lh(val);
          rv <- { r. Some(val,r) };
          put hd(lh); free(hp); fwd hd rv
    }
};;
```

```
proc enq(tl:~Ptr,
         v:~affine lint, tlo:Ptr) {
  letc nn:LList(lint) {
    cell nn (c. nil<lint>(c))
  };
  take tl(sn);
  share nn {
    take sn(lp);
    put sn(c. cons<lint>(v,nn,c));
    drop sn; discard lp
    ||
    put tl(nn); fwd tl tlo
  }
};;
```

The queue constructor lqueue builds the initial structure of a (empty) queue. It allocates and initialises the (empty) sentinel list node sn, and stores a shared reference to it in the head hd and tail hd cells. The queue interfaces are then offered by the deqop and enqop processes, which "bind" the DeqI and EnqI session protocols to the actual implementation of the queue operations deq and enc.

```
proc lqueue(ienq:EnqI,ideq:DeqI) {
  letc sn: LList(lint) {
    cell sn (c.nil<lint>(c)) };
  share sn {
       letc hd:Ptr { cell hd (sn) }; deqop(ideq,hd)
       ||
       letc tl:Ptr { cell tl (sn) }; enqop(ienq,tl)
    }
};;
```

```
proc rec deqop(deci:DeqI,tl:~Ptr) {
  case deci of {
    |#Deq: letc tnext:pair Opt(Aint); Ptr {
             deq(tl,tnext)
           };
           recv tnext (val);
           deci <- val;
           deqop(deci,tnext)
     |#End: wait deci;
           drop tl
    }
};;
```

```
proc rec enqop(enqi:EnqI,tl:~Ptr) {
  case enqi of {
    |#Enq: recv enqi(item);
           letc tnext:Ptr {
             enq(tl,item,tnext)
           };
           enqop(enqi,tnext)
    |#End: wait enqi;
           drop tl
    }

};;
```

This last example highlights some key insights about how CLASS type system compositionally and implicitly captures non-interference and acyclicity in programs' data and control structures.

The code for all examples in this paper may be found at the web site [4].

2 Concluding Remarks

We have presented a brief tutorial on the key design principles and features of CLASS language, illustrating its expressiveness in realistic concurrent session-based and shared-state programs. Details more technically focused on the development and foundations of CLASS may be found in [23, 24, 22, 4].

Besides the examples in this paper, many more functional, imperative and concurrent shared state CLASS code has been developed, and automatically type-checked for the strong safety and liveness properties ensured by its linear type system. Our experience has shown that CLASS type system, despite its expressive power, is flexible enough to deal, without resorting to unsafe features, with complex resource acquisition protocols, such as the dynamic dining philosophers, or the complete "low-level" implementation of Hoare-style monitors with condition variables.

Linear types are becoming more and more relevant in computing practice, as witnessed by the widespread adoption of programming languages such as Rust [20], for general systems programming, Move [32], for blockchain smart contracts, Linear Haskell [3], and other [17, 1, 12, 11, 21]. We expect that some of the ideas introduced in CLASS to be of quite wide application. More information about CLASS, its foundations, current versions of various implementations, and coding examples has been maintained in our evolving web site [4]. Ongoing work on a CLASS compiler for a LLVM/CLANG backend is expected to support a fair assessment of CLASS' performance [2].

I would like to thank Bernardo Toninho, Frank Pfenning, Pedro Rocha, Ricardo Antunes, Vasco T. Vasconcelos, Philip Wadler, Sam Lindley, Jorge A. Perez, Nobuko Yoshida, Stephanie Balzer, and Peter Thiemann for many related discussions, the anonymous referees for very useful comments, and project BIG H22020 Grant ID 952226 for supporting this research.

References

[1] Bernardo Almeida, Andreia Mordido & Vasco T. Vasconcelos (2019): *FreeST: Context-free Session Types in a Functional Language*. In Francisco Martins & Dominic Orchard, editors: *Proceedings Programming Language Approaches to Concurrency- and Communication-cEntric Software, PLACES@ETAPS 2019, Prague, Czech Republic, 7th April 2019*, *EPTCS* 291, pp. 12–23, doi:10.4204/EPTCS.291.2.

[2] Ricardo Antunes (forthcoming): *Efficient Compilation for the Linear Language CLASS*. *MSc Thesis, Tecnico Lisboa, Department of Computer Science*.

[3] Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R. Newton, Simon Peyton Jones & Arnaud Spiwack (2018): *Linear Haskell: practical linearity in a higher-order polymorphic language*. *Proc. ACM Program. Lang.* 2(POPL), pp. 5:1–5:29, doi:10.1145/3158093.

[4] Luís Caires (2025): *CLASS: Classical Linear Logical with Affine Shared State*. Available at `https://luiscaires.org/software/`.

[5] Luís Caires & Jorge A. Pérez (2017): *Linearity, Control Effects, and Behavioral Types*. In Hongseok Yang, editor: *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017*, *Lecture Notes in Computer Science* 10201, Springer, pp. 229–259, doi:10.1007/978-3-662-54434-1_9.

[6] Luís Caires, Jorge A. Pérez, Frank Pfenning & Bernardo Toninho (2013): *Behavioral Polymorphism and Parametricity in Session-Based Communication*. In: *Proceedings of the 22nd European Conference on Programming Languages and Systems*, ESOP'13, Springer-Verlag, Berlin, Heidelberg, p. 330–349, doi:10.1007/978-3-642-37036-6_19.

[7] Luís Caires & Frank Pfenning (2010): *Session Types as Intuitionistic Linear Propositions*. In Paul Gastin & François Laroussinie, editors: *CONCUR 2010 - Concurrency Theory*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 222–236, doi:10.1007/978-3-642-15375-4_16.

[8] Luís Caires & Bernardo Toninho (2024): *The Session Abstract Machine*. In Stephanie Weirich, editor: *Programming Languages and Systems - 33rd European Symposium on Programming, ESOP 2024*, *Lecture Notes in Computer Science* 14576, Springer, pp. 206–235, doi:10.1007/978-3-031-57262-3_9.

[9] Luís Caires, Frank Pfenning & Bernardo Toninho (2016): *Linear logic propositions as session types*. *Mathematical Structures in Computer Science* 26(3), p. 367–423, doi:10.1017/S0960129514000218.

[10] Luís Caires & Bernardo Toninho (2024): *The Session Abstract Machine (Artifact)*. doi:10.5281/zenodo.10459455.

[11] R. Chen, S. Balzer & B. Toninho (2022): *Ferrite: A Judgmental Embedding of Session Types in Rust*. In K. Ali & J. Vitek, editors: *36th European Conference on Object-Oriented Programming, ECOOP 2022*, *LIPIcs* 222, pp. 22:1–22:28, doi:10.4230/LIPICS.ECOOP.2022.22.

[12] A. Das & F. Pfenning (2022): *Rast: A Language for Resource-Aware Session Types*. *Log. Methods Comput. Sci.* 18(1), doi:10.46298/LMCS-18(1:9)2022.

[13] Thomas Ehrhard (2018): *An introduction to differential linear logic: proof-nets, models and antiderivatives*. *Mathematical Structures in Computer Science* 28(7), pp. 995–1060, doi:10.1017/S0960129516000372.

[14] S. Gay & M. Hole (2005): *Subtyping for Session Types in the Pi Calculus*. *Acta Informatica* 42(2-3), pp. 191–225, doi:10.1007/S00236-005-0177-Z.

[15] Kohei Honda (1993): *Types for dyadic interaction*. In Eike Best, editor: *CONCUR'93*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 509–523, doi:10.1007/3-540-57208-2_35.

[16] Kohei Honda, Vasco T. Vasconcelos & Makoto Kubo (1998): *Language primitives and type discipline for structured communication-based programming*. In Chris Hankin, editor: *Programming Languages and Systems*, Springer, pp. 122–138, doi:10.1007/BFB0053567.

[17] Jules Jacobs & Stephanie Balzer (2023): *Higher-Order Leak and Deadlock Free Locks*. *Proc. ACM Program. Lang.* 7(POPL), pp. 1027–1057, doi:10.1145/3571229.

[18] Simon Peyton Jones, Andrew Gordon & Sigbjorn Finne (1996): *Concurrent Haskell*. In: *POPL*, 96, Citeseer, pp. 295–308, doi:10.1145/237721.237794.

[19] Brian W. Kernighan & Dennis Ritchie (1978): *The C Programming Language*. Prentice-Hall.

[20] Steve Klabnik & Carol Nichols (2021): *The Rust Programming Language*.

[21] Julien Lange, Nicholas Ng, Bernardo Toninho & Nobuko Yoshida (2017): *Fencing off go: liveness and safety for channel-based programming*. In Giuseppe Castagna & Andrew D. Gordon, editors: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, ACM, pp. 748–761, doi:10.1145/3009837.3009847.

[22] Pedro Rocha (2022): *CLASS: A Logical Foundation for Typeful Programming with Shared State*. Ph.D. thesis, NOVA University Lisbon.

[23] Pedro Rocha & Luís Caires (2021): *Propositions-as-types and Shared State*. *Proceedings of the ACM on Programming Languages* 5(ICFP), pp. 1–30, doi:10.1145/3473584.

[24] Pedro Rocha & Luís Caires (2023): *Safe Session-Based Concurrency with Shared Linear State*. In Thomas Wies, editor: *Programming Languages and Systems - 32nd European Symposium on Programming, ESOP 2023*, *LNCS* 13990, Springer, pp. 421–450, doi:10.1007/978-3-031-30044-8_16.

[25] Pedro Rocha & Luís Caires (2023): *Safe Session-Based Concurrency with Shared Linear State (Artifact)*. doi:10.5281/zenodo.7506064.

[26] Bernardo Toninho, Luis Caires & Frank Pfenning (2013): *Higher-Order Processes, Functions, and Sessions: A Monadic Integration*. In Matthias Felleisen & Philippa Gardner, editors: *Programming Languages and Systems*, Springer, pp. 350–369, doi:10.1007/978-3-642-37036-6_20.

[27] Bernardo Toninho, Luís Caires & Frank Pfenning (2014): *Corecursion and non-divergence in session-typed processes*. In: *International Symposium on Trustworthy Global Computing*, Springer, pp. 159–175, doi:10.1007/978-3-662-45917-1_11.

[28] Bernardo Toninho, Luís Caires & Frank Pfenning (2021): *A Decade of Dependent Session Types*. In Niccolò Veltri, Nick Benton & Silvia Ghilezan, editors: *PPDP 2021: 23rd International Symposium on Principles and Practice of Declarative Programming*, ACM, pp. 3:1–3:3, doi:10.1145/3479394.3479398.

[29] Bernardo Toninho & Nobuko Yoshida (2021): *On Polymorphic Sessions and Functions: A Tale of Two (Fully Abstract) Encodings*. *ACM Trans. Program. Lang. Syst.* 43(2), doi:10.1145/3457884.

[30] D. A. Turner (1976): *SASL language manual*. Technical Report Technical Report CS/75/1.

[31] Philip Wadler (2014): *Propositions as Sessions*. *Journal of Functional Programming* 24(2-3), pp. 384–418, doi:10.1017/S095679681400001X.

[32] Jingyi Emma Zhong, Kevin Cheang, Shaz Qadeer, Wolfgang Grieskamp, Sam Blackshear, Junkil Park, Yoni Zohar, Clark W. Barrett & David L. Dill: *The Move Prover*. In Shuvendu K. Lahiri & Chao Wang, editors: *Computer Aided Verification - 32nd International Conference, CAV*, doi:10.1007/978-3-030-53288-8_7.