Design and Implementation of Concurrent C0 *

Max Willsey
Carnegie Mellon University
mwillsey@cmu.edu

Rokhini Prabhu
Carnegie Mellon University
rokhini@gmail.com

Frank Pfenning
Carnegie Mellon University †
fp@cmu.edu

We describe Concurrent C0, a type-safe C-like language with contracts and session-typed communication over channels. Concurrent C0 supports an operation called forwarding which allows channels to be combined in a well-defined way. The language's type system enables elegant expression of session types and message-passing concurrent programs. We provide a Go-based implementation with language based optimizations that outperforms traditional message passing techniques.

1 Introduction

Message passing is an approach to concurrent programming where processes do not operate directly on a shared state but instead communicate by passing messages over channels. Many modern languages like Go, Rust, and Haskell provide concurrent processes and channels to facilitate safe concurrent programming through message passing, eliminating the need for locks in common communication patterns. Most message passing systems implement *asynchronous* communication, where the channel contains a buffer so the sender can store the message and proceed without waiting for it to be received.

However, conventional channels do not easily enable safe bidirectional communication. Senders must somehow ensure that they do not receive messages that they just sent over the channel. Furthermore, complex protocols involve multiple types of data, so statically typed channels must be created with the sum of those types. The receiver must check the actual type of the value, typically producing errors if the type is unexpected, essentially degenerating to dynamic typing.

We propose the Concurrent C0 language as a tool to enable safer, more efficient concurrent programming. Like other modern languages, it provides concurrent processes that communicate over channels. It uses session typing to guarantee the safety of communication and also to alleviate the burden of manually synchronizing bidirectional communication [3, 5, 9]. Furthermore, Concurrent C0 offers a concise syntax to express session typed protocols and programs adhering to them. The forwarding operation creates ways to write programs not possible in other languages with message passing. These language features provide additional safety and also enable an optimized implementation.

2 Concurrent C0

Concurrent C0 is based on C0, an imperative programming language closely resembling C designed for use in an introductory programming course. C0 intends to have fully specified semantics to avoid the confusion that comes along with C's undefined behavior [1]. C0 provides memory

*An extended version of this paper can be found at http://maxwillsey.com/papers/cc0-thesis.pdf

†This work is partially funded by the FCT (Portuguese Foundation for Science and Technology) through the Carnegie Mellon Portugal Program.

I. Cervesato and M. Fernández (Eds): Fourth International Workshop
on Linearity 2016 (LINEARITY'16)
EPTCS 238, 2017, pp. 73–82, doi:10.4204/EPTCS.238.8

safety by disallowing pointer arithmetic and casting; all pointers come from the built-in `alloc` and all arrays from the built-in `alloc_array`, and they are not interchangeable as in C. The C0 runtime `NULL`-checks pointer accesses and bound-checks array accesses. C0 is garbage collected [2], eliminating the need to explicitly free memory. C0 also supports optional dynamically checked contracts of the familiar forms `@requires`, `@ensures`, and `@assert`. Students use these contracts to learn how to reason about their code; in particular, the special `@loop_invariant` form allows students to reason about their loops. For more on C0, see [6].

Concurrent C0 (CC0) is an extension of C0, providing safety for the sequential aspects of programs written in CC0. The session-typed concurrent extension is delimited from the sequential language, and thus this paper's contributions could be be readily applied to any language with a similar session-typed linear semantics.

2.1 Concurrency

Concurrent C0 extends C0 with the ability to create concurrent processes and channels to communicate between them. In CC0, a *process*[1] is a unit of concurrent execution, and *channels* are effectively[2] unbounded message buffers that allow the processes on either end to communicate asynchronously.

Consider line 1 of Figure 1: `fib` is a *spawning function* that creates and returns the channel `$c` immediately, spawning a concurrent process that will calculate the `n`th Fibonacci number and send it (denoted by the session type `<!int;>`) along `$c`. Spawning functions provide a session type and a name preceded by `$` for the returned channel so that the function body can use it inside the body. The spawned process is referred to as the *provider*, and the caller of the spawning function is referred to as the *client*.

```
<!int;> $c fib(int n) {
    if (n == 0) {
        send($c, 0); close($c);
    } else if (n == 1) {
        send($c, 1); close($c);
    } else {
        <!int;> $c1 = fib(n-1);
        <!int;> $c2 = fib(n-2);
        int f1 = recv($c1); wait($c1);
        int f2 = recv($c2); wait($c2);
        send($c, f1+f2);    close($c);
    }
}
int main() {
    <!int;> $c = fib(10);
    int f = recv($c); wait($c);
    assert(f == 55);
    return 0;
}
```

Figure 1: Naive concurrent Fibonacci.

Figure 1 provides an example with very simple session types to demonstrate CC0's concurrent programming mechanisms. The client, `main()`, spawns a `fib(10)` provider and then receives on the resulting channel. Note that the spawn *does not* block the `main()` process, but the receive does. Communication in CC0 is asynchronous (channels are always buffered), so sends are always non-blocking, but receives have to block until a message is available.

In the non-trivial case, `fib(n)` spawns two concurrent processes to calculate `fib(n-1)` and `fib(n-2)`. After receiving a value, the parent `fib(n)` waits for the children to close their channels (`$c1` and `$c2`) before sending the result back and then closing its own channel `$c`. The compiler statically verifies that sends and receives are performed in the proper order with the proper types according the channel's session type (Section 2.2). Also, the compiler makes sure that all channels are closed and properly waited for (Section 2.3).

[1] In the current implementations, *processes* are units of execution within a single operating system process, and channels are implemented in shared memory. The features of Concurrent C0 generalize to any communicating processes, but this paper focuses on a shared memory implementation. For other applications, see Section 5.

[2] For some session types, a bounded buffer can behave the same as an unbounded one. See Section 3.1.1.

2.2 Session Types

In concurrent programming, communication between two processes is often supposed to follow some sort of protocol. By adding a type discipline to the (untyped) π-calculus, session typing presents a method of encoding the type of this communication: sequences of types represent how the type changes as the communication takes place [3, 5, 9]. Each type in the sequence is designated as sending or receiving, encoding the direction of communication. This captures the temporal aspect of concurrent communication in a way that conventional (monotyped) channels do not: the type actually reflects processes' progress in communicating with one another.

In Concurrent C0, session types are represented as a semicolon-separated sequence of types between angle brackets. Each type is preceded by either `!` or `?` to denote that a message of the given type is sent or received, respectively. For example, a type where the provider sends an *`int`*, then a *`bool`*, then receives an *`int`* would be written as: `<!int; !bool; ?int;>`. The final semicolon is required; it indicates the end of communication along that channel.

Note how, in Figure 1, the provider (`fib`) behaves according to `<!int;>`, but the client (`main`) does the opposite, receiving where the other sends. Concurrent C0 uses dyadic session types to model the client/provider relationship. Because both ends of the channel communicate using the same protocol, it suffices to just give one type; we type the session from the provider's point of view. The client will then have to obey the *dual* of that type. Duality is an important notion in session typing that captures the requirement that communication actions occur in pairs: if a provider sends an *`int`*, the client must receive an *`int`*.

Session typing systems provide *session fidelity*, the property guaranteeing that processes send and receive the correct data in the correct order according to the session type of the channel. For more on session typing, duality, and session fidelity, see [3, 5, 7].

Session types allow bidirectional communication, but only in one direction at a time. Consider process A providing to client B with the type `<?int; ?int; !bool;>`. The direction of communication starts out toward the provider: A is receiving and B is sending. When A has received both *`int`*s, A has received everything B sent but has not sent the *`bool`* yet, so we know the channel buffer must be empty. Also, we know that B has sent both *`int`*s, so its next action will be to receive; both A and B are at the at the same point in the session type.

When session types change direction like this, a *synchronization point* occurs: both processes must be in the same place in the session type and the buffer must be empty, allowing the direction of communication to switch. Synchronization points occur whenever a session type change directions; a formal treatment can be found in [7].

2.2.1 Branching

Many protocols are not characterized by a straightforward sequence of types. CC0 uses the keyword *`choice`* to denote session types that branch into different sequences of types. Choices are declared in a manner similar to *`struct`*s: a list of labels preceded by types. Using these constructs, the C-like syntax can concisely express even complex session types.

Branches are selected by sending and receiving *labels*, the values of *`choice`* types. Labels are sent using dot notation: `$c.Label`. The **`switch`** operator is used to receive values of *`choice`* types: when it takes a channel variable, it receives and cases on the possible labels. The **`case`** branches must follow the appropriate session type, as indicated by the label.

Receiving a label from the provider (`?`*`choice`*) is called an *external choice*, because the client

is making the decision. External choices are a natural way to encode a server request: the client dictates the type of action the server takes. The `empty()` function on line 29 of Figure 2 offers an external choice. Likewise, sending a label from the provider (`!choice`) is an *internal choice*, because the provider specifies which branch the client will take. Internal choices are ideal for encoding a server response, where the client needs to react to different possibilities. The `Deq` branch on line 4 in Figure 2 is an internal choice; the client must handle the `None` case where no element is available.

In Concurrent C0, `choice`s allow the user to name a session type, also giving the ability to specify recursive ones. In Figure 2, `choice queue` is a recursive session type for a provider that offers a queue of integers. Once an element is enqueued, the type dictates that the provider will continue to behave like a queue. Recursive session types can be implemented with tail recursion (Figure 2, line 34) or with loops. CC0 guarantees that a new process is not spawned by a tail recursive call; it is executed in place by the current provider.

2.3 Linearity

Channel variables have linear semantics [3], but with two references: exactly one client and one provider will have a reference to a channel. This ensures that communication is always one-to-one; there can never be a "dangling" channel with no one listening on the other end, nor will there ever be multiple providers or clients fighting to communicate in one direction over a channel[3]. Because a provider can only have one client at a time (initially the caller of the function that spawned it), there is a natural correspondence between the client-provider relationship in a CC0 program and the parent-child relationship in a tree. The `main()` function is a process with no clients and therefore the root of the tree.

The `close` and `wait` primitives let the provider and client satisfy the linear type system. A process providing across channel `$c` must call `close($c)` to terminate. The provider must have already consumed all of its references and be a leaf in the process tree. Before terminating, a client with a reference to a channel `$c` must call `wait($c)` to ensure the provider terminates.

Channel references can be manipulated like other variables, but they are still subject to linearity throughout the whole program. They cannot be copied, only renamed; the old reference cannot be used. When passing a channel into a spawning function, the caller gives up its reference to allow the new process to use the channel. Sending channels along channels works much in the same way: the sender gives up its reference to the receiving process. Linearity ensures that channel references are not leaked or duplicated, so the process tree will remain a tree even with dramatic manipulation of the communication structure.

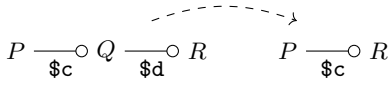
2.4 Forwarding

Concurrent C0 implements an operation not commonly found in other languages with message passing called *forwarding* which allows a process to terminate before its child and remove itself from the process tree. A node with exactly one child[4] can be contracted by the forward operation, allowing its parent and child to communicate directly without it in the middle. Removing the

[3] CC0 implements *linear channels* from [3, 7] which have exactly one client. The same paper provides a notion of *shared channels* which can support multiple clients, but these are not presently in CC0.

[4] Because forwarding terminates the process, linearity dictates that all of its other references must have been properly destroyed at the time of the forward.

inner process effectively merges the two channels; because a process can only forward channels of the same session type, session fidelity is preserved and communication continues as if nothing happened [7, 9].



At a very high level, forwarding can be thought of as setting a channel equal to another channel. To the right, process Q executes the forward `$c = $d`, terminating and combining the two channels into one. It is not obvious how to merge buffers that contain messages; what if `$d` was not empty at the time of the forward? See Figure 3e for an example where concatenation does not work, because messages are temporarily flowing in opposite directions.

We propose an alternate view of forwarding: as a special kind of message. We use the session typing system to infer the direction of communication according to the forwarding process, so a forward sends a special message along the channel in that direction containing a reference to the other channel. This message must be the last one in the buffer because the forwarding process terminated after sending it. When a process receives the forward, it destroys the channel it was sent over and replaces its reference with the new channel from the forward message. Figure 3 contains a more detailed example of how forwarding works.

Section 3.1.2 discusses the details of our implementation, but it's important to note that this interpretation of forwarding allows implementation on any level. This view of forwarding, to the best of our knowledge, is a novel contribution of this work, and could be implemented in any session typed, message passing language.

3 Implementation

Concurrent C0's typing system not only ensures the safety of concurrent code, but it also allows for an efficient parallelizable implementation. Session typing directly enables our implementation to use fewer, smaller buffers than other message passing techniques.

3.1 Compiler

Concurrent C0 enforces linearity and session fidelity to produce safe concurrent code. The compiler typechecks programs to make sure that messages are sent and received according to the appropriate session types, and it also ensures that the linearity of channels is respected. While certainly important to CC0, the typechecker itself is not a novel contribution of this work, and interested readers are referred to [4] for more about typechecking session typed and linear languages. After typechecking, the compiler inserts annotations that inform the runtime about the communication structure. Finally, CC0 source code is compiled to a target language (C or Go) then linked with a runtime implementation written in the same target language.

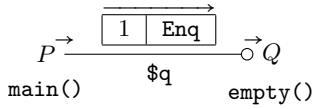
3.1.1 Type Width

Certain session types dictate that only so many values can be buffered at a time. For example, the type `<!bool; ?int;>` could only possibly buffer one value at a time, because the `int` must be sent from the client, which can only occur once the client has received the previous `bool`. This quantity is called the *width* of the type. The CC0 compiler infers widths, allowing the runtime to use small, fixed length circular buffer as queues and not have to worry about ever resizing.

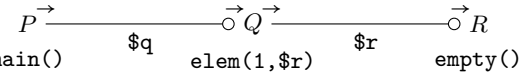
```
1  // external choice for request
2  choice queue {
3    <?int; ?choice queue>  Enq;
4    <!choice queue_elem>   Deq;
5  };
6  typedef <?choice queue> queue;
7
8  // internal choice for response
9  choice queue_elem {
10   <!int; ?choice queue> Some;
11   < >                   None;
12 };
13
14 // provider that holds element x and
15 // points to the rest of the queue $r
16 queue $q elem (int x, queue $r) {
17   switch ($q) {
18   case Enq:
19     int y = recv($q);
20     $r.Enq; send($r, y);
21     $q = elem(x, $r);
22   case Deq:
23     $q.Some; send($q, x);
24     $q = $r;
25   }
26 }
27
28 // provider for the end of the queue
29 queue $q empty () {
30   switch ($q) {
31   case Enq:
32     int y = recv($q);
33     queue $e = empty();
34     $q = elem(y, $e);
35   case Deq:
36     $q.None;
37     close($q);
38   }
39 }
40
41 void dealloc (queue $q) {
42   $q.Deq; switch($q) {
43     case Some:
44       recv($q);
45       dealloc($q);
46       return;
47     case None:
48       wait($q);
49       return;
50   }
51 }
52
53 int main () {
54   queue $q = empty();
55   $q.Enq; send($q, 1);
56   $q.Enq; send($q, 2);
57   dealloc($q);
58   return 0;
59 }
```

Figure 2: `queue.c1`, a queue implementation where each element is a concurrent process.

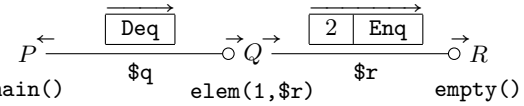
The arrows above the channel contents indicate the actual flow of messages along the channel. The small arrows above channel endpoints indicate the direction of that process' next action along that channel according the session type.



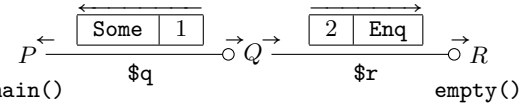
(a) P enqueues 1 [line 55].



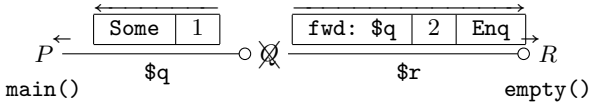
(b) Q gets the enqueue, spawning a new `empty()` process and channel [line 33–34].



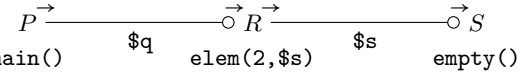
(c) P enqueues 2 [line 56] which Q passes to the back of the queue [line 20]. P sends a dequeue request and waits for the result [line 42].



(d) Q responds to the dequeue [line 23] and is about to forward [line 24].

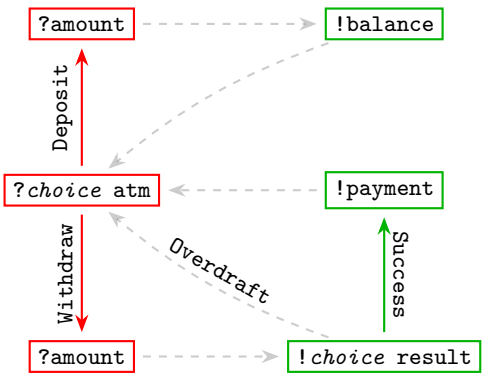


(e) Q forwards `$q = $r` [line 24] by sending a forward message in the direction of communication according to the session type, not the state of the channel buffers. Q terminates, but `$r` must persist because it still has messages. Simply concatenating the buffer here will not work because they have different directions.



(f) R finally gets the enqueue, spawning a new `empty()` process and channel [line 33–34]. When R receives the forward, it deallocates `$r` (which is now safe because `$r` is empty) and will now use `$q` instead.

Figure 3: An illustration using `queue.c1` (Figure 2) demonstrating how treating forwarding as a message resolves communication direction issues.




```
typedef amount  int;
typedef balance int;
typedef payment int;

typedef <?choice atm> atm;

choice atm {
    <?amount; !balance; atm>  Deposit;
    <?amount; !choice result> Withdraw;
};

choice result {
    <!payment; atm> Success;
    <atm>           Overdraft;
};
```

Figure 4: Code and graph of type `atm` with width 2.

Session types can be viewed as a directed graph in which a walk represents a possible sequence of sent or received types. Nodes are colored as sending (green) or receiving (red); see Figure 4. We know that the buffer will only contain messages going in one way at a time, so there are actually two graphs, one red and one green, connected by the dashed gray edges representing synchronization points where we know the buffer will be empty. Thus, the width of the type is the number of nodes in the longest walk in either subgraph.

An ATM is a canonical example in the session typing literature, and Figure 4 shows the code and type graph for a simple ATM protocol. A process providing `<?choice atm>` could clearly stay alive forever: the client may `Deposit` or `Withdraw` an unbounded number of times. However, the width of the type is only 2; so the channel will never need to buffer more than two items.

Note that type width is compatible with forwarding because of the forward-as-message interpretation. Forwards are received *instead of* the intended message, so the forward message will occupy that allocated space.

3.1.2 Forwarding

At each forward call-site, the CC0 compiler infers the direction of communication according the forwarding process's session type. In the generated code, that direction is passed into the forward runtime function. Just like the semantic understanding, the runtime sends a specially tagged message in that direction and then terminates the calling process. Nothing else occurs until the forward is received.

A process attempting to receive another value may see the special forward tag instead. The forward message is guaranteed to be the last message in the buffer, so the receiving process destroys the channel. The forward message contains a reference to the new channel, so the receiving process replaces its own reference to the destroyed channel with the new one, and then it attempts to receive the value it initially expected over that new channel. This ensures the transparency of the forward: this process is still going to receive the value that it expected, and all future interactions over that channel reference will use the new channel instead. Because forwards are deferred, the receiver may need to handle many forwards before getting the expected message.

3.2 Runtime System

The CC0 compiler generates C or Go code that is linked with one of several runtime systems that contain the logic for message passing and manipulating processes. The runtimes have different threading models and synchronization strategies, but they share the same general structure centered around channels. A channel contains a message queue, its direction, a mutex, and a condition variable. The mutex is necessary to protect channel state, and the condition variable is used by receivers to wait on messages to arrive or the queue to change directions.

The runtime consists of four main functions that provide all the necessary functionality for spawning processes and message passing: `NewChannel`, `Send`, `Recv`, and `Forward`. The CC0 functions `close` and `wait` are implemented by sending and receiving a special `DONE` message.

`NewChannel` creates a new concurrent process and the channel along which it will provide, returning a reference to that channel to the caller (client). `NewChannel` takes in the function and arguments for the new provider process as well as the type width and initial direction of the channel as inferred by the compiler, allowing the runtime to create a channel with a bounded ring-buffer when possible.

`Send` sends a given message over a given channel, additionally taking in the message's type and the inferred direction. `Send` locks the channel, enqueues the message with its type, sets the direction of the queue, and unlocks. A receiver may be waiting for the message, so the sender must wake up the potential receiver by signaling the condition variable.

`Recv` receives a message over a given channel, taking in the message's expected type and the inferred direction. `Recv` locks the channel and attempts to receive the message. If the buffer is empty or still flowing in the other direction, then the caller will give up the lock and wait on the condition variable for the sender. If the message is a forward, the receiver handles it, installing the new channel; see Section 3.1.2 for details. `Recv` asserts that the received message is of the expected type, panicking if it does not match (and is not a forward).

`Forward` takes in the two channels involved in the forward and the inferred direction of communication. The forwarding process sends a forward message in the inferred direction using the regular message passing functionality, then it terminates. See Section 2.4 and Section 3.1.2.

Because CC0 encourages highly concurrent programming, most programs spawn many processes. Our early C runtimes used 1:1 or custom M:N threading models; neither performed as well as the Go runtimes. Go is an imperative programming language with a lightweight threading model that provides concurrency and efficient parallelism [8].

4 Experimental Comparison and Analysis

To benchmark Concurrent C0, we created `go0`, a naive, proof-of-concept implementation that uses Go's built-in channels to implement CC0 channels. As CC0 channels provide safe bidirectional communication, two Go channels must be used to implement a CC0 channel without additional synchronization. `go0` serves as a stand-in modeling how message passing is done in other languages (with two large channels intended for one-way communication), but it conforms to the same interface as our other implementations so we can run the same tests against it.

Analysis of both the C and Go runtimes can be found in the extended version of this paper; here we compare `go0` against `go2`, a Go implementation which uses the full suite of language based optimizations detailed in Section 3. Our benchmarking suite[5] consists of many highly concurrent data structures, like the queue in Figure 2. Most of the work done in these tests is

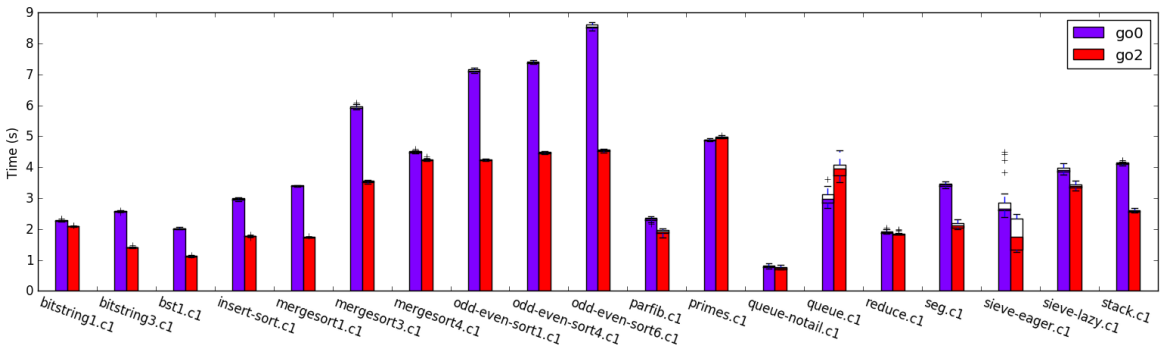


Figure 5: Median benchmark times of the `go0` and `go2` runtimes over 20 samples.

communication, so as to highlight the efficiency of our message passing runtimes. All benchmarks were run on a 2015 MacBook Pro with an Intel Core i7-4870HQ CPU with 4 cores at 2.50GHz.

The `go0` vs. `go2` benchmark in Figure 5 demonstrates the effectiveness of our implementation techniques. Compared to the naive implementation, our optimized version ran 1.38× faster on average[6]. We suspect that the speed up would be even more dramatic if the Go compiler optimized tail recursion. Concurrent C0 encourages a tail recursive style of programming; see Section 2.2.1. The `queue-notail.c1` test case is the same as `queue.c1`, except that it is written with loops as opposed to tail recursion. The more than 2× difference in both Go runtimes' performance indicates that Go's lack of optimization in this case is a serious hindrance. Other test cases like `primes.c1` rely heavily on mutually recursive tail calls, so even though the negative impact is similar, no `-notail` version was written for those cases.

5 Future Work

The knowledge given by session types could improve scheduling decisions. The structure of relationships between communicating process could enable optimizations like co-scheduling providers and clients to increase parallel performance. The same information might also assist the runtime in deciding on granularity; the structure of the process tree could help save the overhead of spawning new concurrent processes in some situations, just running them inline instead.

Session typing and linearity make communication of values safe, and Concurrent C0 (from C0) is memory safe for sequential programs, but the combination of shared memory and concurrency leads to race conditions. Presently, our implementations allow sending and receiving pointer and arrays between processes, but there is no attempt to enforce the safety of accesses and writes. Given that channels already have linear semantics, CC0 could benefit from a linear or affine treatment of shared memory like that of Rust[7].

Session types are traditionally associated with distributed computing, so a distributed implementation of Concurrent C0 could apply some of the contributions of this work. Specifically, the concept of forwarding as a message would be even more beneficial than it was in the shared memory setting, as synchronization is even more challenging on the distributed scale.

[5] See the full suite of tests at `http://maxwillsey.com/assets/cc0-linear16-benchmarks.tgz`

[6] Average is calculated as geometric mean of the ratios of the median benchmark times

[7] `https://doc.rust-lang.org/book/ownership.html`

References

[1] Rob Arnold. "C_0, an Imperative Programming Language for Novice Computer Scientists". Available as Technical Report CMU-CS-10-145. M.S. Thesis. Department of Computer Science, Carnegie Mellon University, Dec. 2010.

[2] Hans-J. Boehm. *A garbage collector for C and C*. URL: http://www.hboehm.info/gc.

[3] Luís Caires and Frank Pfenning. "Session Types as Intuitionistic Linear Propositions". In: *Proceedings of the 21st International Conference on Concurrency Theory (CONCUR 2010)*. Paris, France: Springer LNCS 6269, Aug. 2010, pp. 222–236. DOI: 10.1007/978-3-642-15375-4_16.

[4] Dennis Griffith. "Polarized Substructural Session Types". In preparation. PhD thesis. University of Illinois at Urbana-Champaign, Apr. 2016.

[5] Kohei Honda. "Types for Dyadic Interaction". In: *4th International Conference on Concurrency Theory*. CONCUR'93. Springer LNCS 715, 1993, pp. 509–523. DOI: 10.1007/3-540-57208-2_35.

[6] Frank Pfenning. *C0 Language*. URL: http://c0.typesafety.net.

[7] Frank Pfenning and Dennis Griffith. "Polarized Substructural Session Types". In: *Proceedings of the 18th International Conference on Foundations of Software Science and Computation Structures (FoSSaCS 2015)*. Ed. by A. Pitts. Invited talk. London, England: Springer LNCS, Apr. 2015. DOI: 10.1007/978-3-662-46678-0_1.

[8] *The Go Programming Language*. URL: https://golang.org.

[9] Bernardo Toninho. "A Logical Foundation for Session-based Concurrent Computation". Available as Technical Report CMU-CS-15-109. Ph.D. Thesis. School of Computer Science, Carnegie Mellon University, May 2015.