Design and Implementation of Concurrent C0

Max Willsey

mwillsey@cmu.edu

Advisor: Frank Pfenning

Carnegie Mellon University

School of Computer Science

Senior Honors Thesis

April 29, 2016

Abstract

Concurrent C0 is a type-safe, C-like language with session-typed communication that makes writing highly concurrent, message passing programs easier, safer, and more efficient than other languages. Concurrent C0 presents a novel interpretation of a forwarding operation which allows channels to be combined in a well-defined way. We provide C- and Go-based implementations with language based optimizations that outperform traditional message passing techniques.

Contents

List of Figures

Chapter 1

Introduction

Message passing is an approach to concurrent programming where processes[1] do not operate directly on shared state but instead communicate by passing messages over channels. Many modern languages like Go, Rust, and Haskell provide channels implemented in shared memory to facilitate safe concurrent programming through message passing.

Channels make concurrent programming easier by eliminating the need for locks in common communication patterns. They enable the programmer to easily send or receive data across the channel in a *synchronous* or *asynchronous* manner. Synchronous communication means a send operation will not complete until the process on other end of the channel receives. In an asynchronous model, the channel contains a buffer so the sender can store the message and proceed without it being received right away; for this reason, asynchronous communication is also called buffered communication.

However, conventional asynchronous channels do not easily enable safe bidirectional communication. If a process A sends a message to process B and wishes to receive a response, simply receiving from that same channel may result in A receiving *its own* message (see Figure 1.1). Clearly, A intended to receive a response from B, but, due to the asynchronous nature of communication, there's no guarantee that B receives A's message before A attempts to receive B's response. There are a number of existing approaches used by modern programming languages to address this

[1]Throughout this paper, a *process* refers to a unit of execution abstracted from its implementation. Process implementations can range from coroutines within a single thread to machines communicating over a network.

receive-your-own-message problem:

- The language could simply leave it up to the programmer, allowing arbitrary send and receive operations. Haskell takes this approach[2].
- To provide some amount of safety, the language could allow the programmer to locally constrain channels as one-way. Go has bidirectional channels by default, but it allows this restriction[3]. Constraining channels allows two entities (a channel for each direction) to safely enable bidirectional communication, although constraining the direction is left up to the programmer.
- For complete safety, channels should be limited to unidirectional communication. In this case, a channel requires distinct endpoints or handles for sending and receiving, as well a type system that safely prevents aliasing of these handles. Rust takes this approach[4], providing truly safe bidirectional communication, but requiring four entities (two channels, two endpoints each) to do so.

Figure 1.1 shows a Go program where communicating a very basic client/server protocol over a single channel goes awry. The problem arises when the client—after sending a "hash" request to the server—tries to receive the response. Because Go channels are bidirectional by default[5], the client may instead receive *its own* message (`0xbadc0de` in this case). Not only is this a very poor hash, but now the server may be waiting indefinitely for a message from the client. In a more sophisticated protocol, this could lead to deadlock. In Go, this could be alleviated by using an unbuffered channel (implementing synchronous communications), using two one-way channels, or implementing some kind of synchronization to prevent receiving your own message; but each of these entails additional complexity or loss of concurrency.

Furthermore, channels are typically either dynamically typed or statically given a single type that represents what can be sent. When communication involves multiple types of data, as is often

[2] `http://hackage.haskell.org/package/base-4.8.2.0/docs/Control-Concurrent-Chan.html`
[3] `https://golang.org/ref/spec#Channel_types`
[4] `http://doc.rust-lang.org/std/sync/mpsc/`
[5] Go does allow one-way channels of a sort, but does not offer truly distinct one-way endpoints like Rust. One-way channels would solve this particular problem at the expense of having to use two separate channels.

```
func client(ch chan int) int {
        ch <- 0xbadc0de  // send request
        response := <-ch // get response
        return response
}

func hashServer(ch chan int) {
        request := <-ch // get request
        response := (request & 37892) * 232341
        ch <- response // send response
}

func main() {
        ch := make(chan int, 1) // buffered chan
        go hashServer(ch)       // run concurrently
        response := client(ch)
        fmt.Printf("Response: %x\n", response)
        // output: "Response: badc0de"
}
```

Figure 1.1: Go program where the client may receive its own request instead of the server's response.

the case when adhering to a complex protocol, typed channels must be created with the sum of those types. To receive from sum-typed channel, the receiver must check the actual type of the value, typically producing errors if the received type was not what was expected (see Figure 2.2a). This approach is verbose and essentially degenerates to that of a dynamically typed language, because typed channels do not encode any temporal properties of the protocol.

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, 6, 10]. 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 not only provide additional safety, but also enable an efficient implementation with memory and performance optimizations.

In Chapter 2, we provide a high-level overview of the Concurrent C0 language, including the

benefits of its session typing system and a demonstration of how it facilitates safe concurrent programming. In Chapter 3, we explain the challenges for an implementation and our solution. Chapter 4 analyzes the performance benefits of our implementation with benchmarks. Finally, Chapter 5 explores potential uses and extensions of this work.

Chapter 2

Concurrent C0

Concurrent C0 is an imperative language in the C style: for sequential programs, Concurrent C0 is nearly identical C in its syntax and semantics, but it avoids undefined and implementation-defined behaviors (see Section 2.1). Concurrent C0 facilitates concurrent programming by allowing process spawning and manipulation with safe message passing over channels[1]. Lightweight, C-like syntax enables users to concisely specify protocols as session types, and the session type system with linearity ensures the safety of concurrent code.

The semantics of Concurrent C0 provides many guarantees for concurrent programs: they will not deadlock, communicated values are guaranteed to be of the specified type, and process and channel resources cannot be leaked. We also introduce a novel interpretation of a *forward* operation that allows programs to safely manipulate the communication structure in such way that would be impossible or cumbersome to do so without it.

2.1 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

[1]In the current implementations of Concurrent C0, 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 Chapter 5.

avoid the confusion that comes along with C's undefined behavior [1].

C0 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 [7].

As memory management is a common stumbling block for students, C0 provides complete memory safety. Pointers and arrays are distinct, and accesses are `NULL`-checked and bounds-checked, respectively. C0 provides only two allocation primitives: `alloc` for structures and pointers, and `alloc_array` for arrays. Allocation always zeroes memory first, so the user cannot access uninitialized memory. Large types (structs and arrays) cannot be stack-allocated in C0, and there is no address-of operator or pointer arithmetic, so one cannot obtain a reference to anything that did not result from a call to `alloc` or `alloc_array`. Finally, C0 uses the Boehm-Demers-Weiser conservative garbage collector [2], eliminating the need to explicitly free memory.

The C0 compiler generates human-readable C code with these additional safety features that is then sent to a C compiler.

C0 is used at Carnegie Mellon in the *Principles of Imperative Computation* course (15-122). In this first year course, students learn the basics of imperative data structures and algorithms in C0, and then transition into C later in the the course. In the Compiler Design course (15-411), students design and build compilers for a subset of C0, starting with arithmetic operations and finishing with an optimizing compiler for nearly the complete language.

Concurrent C0 (CC0) is a strict extension of C0, so all C0 programs still compile. This provides safety for the sequential aspects of programs written in CC0, and it also enabled us to base our implementation on the C0 compiler. The session-typed concurrent extension is delimited from the sequential language, and thus this paper's contributions could be readily applied to any language with a similar session-typed linear semantics.

2.2 Concurrency

Concurrent C0 extends C0 with the ability to create concurrent processes and channels to communicate between them. A *process* in Concurrent C0 is a unit of concurrent execution, distinct from

the notion of a process in operating systems. *Channels* behave as unbounded buffers[2] and allow the processes on either end to communicate asynchronously. Consider the following declaration for a function that will concurrently calculate the nth Fibonacci number:

```
<!int;> $c fib(int n);
```

The right-hand side looks familiar; this is a function named `fib` that accepts an `int` parameter. As expected in a C-like language, the return type is on the left-hand side. This return type states that `fib` returns a channel `$c` with the session type `<!int;>`. Session types in Concurrent C0 will be covered detail in Section 2.3, but for now this type denotes that an `int` will be sent along `$c`. Spawning functions must provide a name (always preceded by a `$`) as well as a session type for the returned channel, so the channel can be referred to inside the function body. Spawning functions create and return a channel immediately, spawning a concurrent process that will communicate along the returned channel. The spawned process is referred to as the *provider* (or server) and the caller of the spawning function is referred to as the *client*. Jumping ahead to Section 2.3, it is now up to the provider to communicate along the returned channel according to the specified session type.

Figure 2.1 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 trivial cases, the `fib(n)` will send back the appropriate value across the channel (without blocking) and then close the channel. The close operation must be a provider's final action on a channel, notifying the client that the communication across the channel is now complete. In other cases, the `fib(n)` process will spawn two concurrent processes to calculate `fib(n-1)` and `fib(n-2)`. After receiving a value, the parent `fib(n)` will wait for the children to terminate before sending the result back.

[2]For some session types, we can use a bounded buffer in place of an unbounded one. See Section 3.1.2.

```
<!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 2.1: Naive concurrent Fibonacci in CC0.

The compiler statically verifies that sends and receives are performed in the proper order with the proper types according the channel's session type. Also, the compiler makes sure that all channels are closed and properly waited for. Section 2.3 and Section 2.4 covering session types and linearity will go into more detail on this.

2.3 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, 6, 10]. 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.

```haskell
data Protocol = AInt Int
              | ABool Bool

server toClient toServer = do
  writeChan toClient (AInt 42)
  writeChan toClient (ABool True)
  shouldBeInt2 <- readChan toServer
  case shouldBeInt2 of
    AInt i2 -> print i2
    ABool b -> error "should be an int"

main = do
  toClient <- newChan
  toServer <- newChan
  -- forkIO will run server concurrently
  forkIO (server toClient toServer)
  shouldBeInt1 <- readChan toClient
  case shouldBeInt1 of
    ABool b -> error "should be an int"
    AInt i1 -> do
      shouldBeBool <- readChan toClient
      case shouldBeBool of
        AInt i2 -> error "should be bool"
        ABool b ->
          writeChan toServer (AInt (i1+1))
```

(a) Implementation in Haskell.

```
typedef <!int; !bool; ?int;> protocol;

protocol $c server() {
  send($c, 42);
  send($c, true);
  printint(recv($c)); // receive i2
  close($c);
}

int main() {
  protocol $c = server();
  int i1 = recv($c);
  bool b = recv($c);
  send($c, i1+1);
  wait($c);
  return 0;
}
```

(b) Implementation in Concurrent C0.

Figure 2.2: A simple protocol where the server sends an *int*, then a *bool*, then receives an *int*.

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.

Being able to give a type to even very simple protocols can drastically simplify communication code. Figure 2.2a and Figure 2.2b provide programs in Haskell and CC0 that implement the `<!`*int*`; !`*bool*`; ?`*int*`;>` protocol. The Haskell program uses two channels to implement bidirectional communication and avoid the receive-your-own-message problem mentioned earlier. Note also that Haskell, a language with a very powerful type system including sum types, degenerates to the approach of dynamically typed language: it must check each received value to safely assert that it is of the expected type.

Note how, in Figure 2.2b, the provider (`server`) behaves according to `<!`*int*`; !`*bool*`; ?`*int*`;>`, but the client (`main`) does the opposite, receiving where the other sends. Concurrent C0 uses

dyadic session types that 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 is sending an `int`, the client must be receiving an `int`. Readers interested in the theory behind duality in Concurrent C0's session types are referred to [3].

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. Session fidelity is the theoretical basis on which CC0 achieves type safe, deadlock-free concurrent execution [4, 6]. Because session types track the expected type and direction of communication as it takes place, we can provide errors concisely stating how and when a protocol is violated. Consider if the line `bool b = recv($c)` were removed from the Figure 2.2b example. Receiving that value does not actually affect the program execution: sending back the integer does not depend on this boolean. Still though, the client `main()` is not adhering to the agreed upon protocol of `$c`, and the CC0 compiler will reject this program with the following error message:

```
client sending along channel providing output $c : <!bool;?int;>
  send($c, i1+1);
  ~~~~~~~~~~~~~~
Compilation failed
```

In this case, the client should have received a `bool` along `$c` (`!bool` from the provider's point of view, as is conventional), but instead the client attempted to send an `int` (meaning the provider would have received, `?int`). In this way, the compiler can represent a protocol adherence error as a "type mismatch" error that should be familiar to most programmers.

Presently, session types can include "small" (word-sized) types: `int`s, `bool`s, etc. Channel references and function pointers are also small types, so they can safely be sent over channels as well. The limitation to small types serves only to simplify our implementations; there's no technical reason that large types (`structs`) could not be supported. Because pointers and arrays are small types, they can be sent over channels; however there is currently no way to enforce the safety (statically or dynamically) of accesses to memory shared between CC0 processes. This issue is not

```
choice calc {
  <?int; !bool;>                    IsEven;
  <?int; ?int; !int;>               Mult;
  <?int; ?int; !choice div_result> Div;
}

choice div_result {
  <!int;> DivOk;
  < >     DivByZero;
}

<?choice calc> $c calculator() {
  switch ($c) {
  case IsEven: ... break;
  case Mult:   ... break;
  case Div:    ... break;
  ...
```

Figure 2.3: A calculator protocol with internal and external choice.

yet addressed in the current implementation, and Chapter 5 discusses potential solutions.

2.3.1 Branching

Many protocols are not characterized by a straightforward sequence of types. A provider may need to respond to various kinds of requests from the client, or respond in different ways. The concept of *choice* encodes this: parties send and receive *labels* indicating which branch of the protocol to take. To denote session types that branch, CC0 uses the keyword `choice`. Choices are declared in a manner similar to `struct`s: a list of labels preceded by types. Using these constructs, one can concisely express even complex session types in a way fits with the rest of the C-like syntax.

Consider the definition of `choice calc` in Figure 2.3. A process would provide this service along a channel of type `<?choice calc>`, with the `?` denoting that it will receive a label from the client indicating which branch to take. We call this 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.

Note the lack of a semicolon at the end of `<?choice calc>` because the types within the `choice`

are semicolon terminated, indicating the end of the type.

Branches are selected by sending and receiving labels, the values of *choice* types. Labels are sent using dot notation: `$c.Label`. In CC0, 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. For example, in Figure 2.3, the compiler will make sure that the code in the `IsEven` branch sends an *int* then receives a *bool*.

To add a division operation to our simple calculator protocol, just adding a branch that looks like `<?`*int*`; ?`*int*`; !`*int*`;> Div;` would be insufficient; we need a way to tell the client that division failed if they requested a divide-by-zero. This introduces the *internal choice*, where the provider specifies which branch the client will take. Intuitively—as this is where the provider sends the choice—we specify the internal choice with `!`*choice*. Figure 2.3 provides a definition of *choice* `div_result`, an internal choice meant to be sent back to the client as the response to a `Div` request.

In Concurrent C0, *choice*s allow the user to name a session type, also giving the ability to specify recursive ones. Without recursion, the language would only permit one-time-use protocols. In Figure 2.4, *choice* `ints` is a recursive session type for a provider that offers a stream of integers. Like a conventional server, a process providing *choice* `ints` will persist and serve multiple requests until the client requests its termination using the `Stop` label.

Because Concurrent C0 is an imperative language, users can write loops to implement providers that adhere to recursive session types. CC0 also provides tail recursion for processes; a provider can "become" a new provider of the same session type: `$c = from(n+1);`. CC0 guarantees that a new process is not spawned in this case, the call is executed in place by the current provider. Note that a tail call does not return; the caller terminates and is effectively replaced by the callee. Figure 2.4 shows both styles of implementing recursive session types.

```
choice ints {
  <!int; ?choice ints> Next;
  < >                  Stop;
};

typedef <?choice ints> ints;

ints $c from(int n) {
  switch ($c) {
    case Next:
      send($c,n);
      $c = from(n+1);
    case Stop:
      close($c);
  }
}
```

```
ints $c from(int n) {
  while(true) {
    switch ($c) {
      case Next:
        send($c,n);
        n += 1; break;
      case Stop:
        close($c);
    }
  }
}
```

Figure 2.4: A session-typed stream of natural numbers with an imperative-style implementation using a loop and a tail recursive implementation.

2.3.2 Synchronization Points

Session types allow bidirectional communication, but only in one direction at a time. Consider process A providing to client B (denoted $A \multimap B$) with the type `<?int; ?int; !bool;>`. The direction of communication starts out toward the provider: A is receiving and B is sending. Because of the asynchronous, concurrent nature of communication in CC0, this can happen a number of ways. B might run first, sending both `int`s before A has even called `recv`. Likewise, A might run first, call `recv`, and block before B even gets a chance to run. Session fidelity promises that the communication will work out in the end, but there are many possible concurrent interleavings for even this very simple session type.

Despite the many interleavings for the two sends, consider when A has received both `int`s and is about to send the `bool`. A has received everything B sent, but A has not sent anything 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. Once A sends the `bool`, the flow of communication is towards the client.

When session types change direction (from `?int` to `!bool` in the previous example), we call

this a *synchronization point*: 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—even recursive ones—change directions; a more formal treatment can be found in [8].

Synchronization points enable Concurrent C0 to provide bidirectional communication from the programmer's perspective, but messages really only go one way at a time. This design is driven by the assumption that bidirectional communication is only useful when synchronized; in other words, one party can only say something useful once it has heard everything the other party has to say. This notion is supported by the sequential nature of session types: a process must perform the actions in order.

2.4 Linearity

Linearity plays a key role in CC0's session typing system [3], most directly influencing the use of channels. CC0 channel variables have linear semantics, but with two references: at all times exactly one client and one provider will have a reference to a channel. If $A \multimap B$ along the channel `$c`, then only processes A and B have a reference to `$c`. 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 a tree. The `main()` function is a process with no clients and therefore the root of the tree. If $A \multimap B$, then A is a child of B in the tree. Figure 2.5 gives the process tree for naive concurrent Fibonacci. The process tree also includes the Fibonacci number sent back across the channel. Recall that each node in the tree is a concurrently executing process, and each edge is a channel.

CC0 provides the `close` and `wait` primitives so the provider and client can satisfy the linear type

[3] CC0 implements *linear channels* from [3, 8] 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.

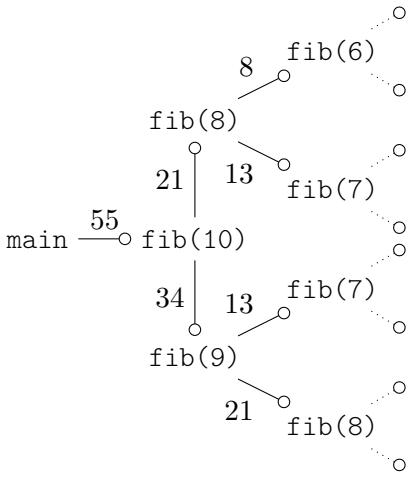


Figure 2.5: Process tree for naive Fibonacci (code in Figure 2.1).

system. When done providing across a channel `$c`, a provider must call `close($c)` to terminate. The provider must have already consumed all of its references and be a leaf in the process tree. Likewise, a client with a reference to a channel `$c` must call `wait($c)` to ensure the provider terminates before destroying the channel.

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.

2.5 Forwarding

So far, the concurrency primitives are similar to the fork/join model where parents must survive to collect their children. Spawning grows the process tree by adding providers as children, and the `wait` operation allows parents to "join" their terminated children, reducing the process tree from the bottom up. Linearity ensures that a parent cannot terminate while it has running children.

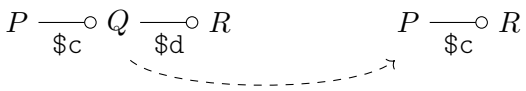


Figure 2.6: Process Q executes the forward `$c = $d` when messages are flowing to the right.

Concurrent C0 implements an operation called *forwarding* which allows a process to terminate before its child and remove itself from the process tree. Forwarding is not present in other languages with message passing (Go, Haskell, Rust, etc.). Consider a process Q providing across channel `$c` that has a child providing to it along `$d` (see Figure 2.6) where `$c` and `$d` are of the same session type. If Q has no more work to do, instead of staying alive just to pass messages between `$c` and `$d`, it can execute a forward: `$c = $d`. The forward makes channel `$c` behave like `$d`, and Q terminates. Because forwarding terminates the process, linearity dictates that all of its other references must have been properly destroyed when it forwards. Nodes like Q with one exactly child can be contracted by the forward operation, allowing Q's parent and child to communicate directly without Q in the middle. This enables the programmer to shrink the process tree from locations other than the leaves.

Forwarding is transparent to other ends of the channels involved in the forward. Neither P nor R can tell when Q forwards. The channels `$c` and `$d` still behave according to their session type even though someone else is on the other end. Because a process can only forward channels of the same session type, session fidelity is preserved and communication continues as if nothing happened [8]. The `Deq` case in Figure A.1 line 23 demonstrates how telling channel to transparently "behave like" another channel is a useful: dequeuing should make a queue behave exactly like the rest of the queue.

At a very high level, forwarding can be thought of as setting a channel equal to another channel (and the `$c = $d` notation comes from this intuition), but its semantics and implementation are more complicated. For example, in Figure 2.6, how do we choose `$c` to persist instead of `$d`? What happened to the messages in `$d`?

Concurrency is another challenge to understanding forwarding. Because of asynchronous communication, a provider and client may disagree on the session type of a channel. Even though the forwarding process sees the two channels have the same session type, the outer processes may still

be waiting on messages. Forwarding terminates the process and closes one of the channels, but it is not obvious how to deal with buffers that contain messages. Simply combining the channels by concatenating the buffers does not work; see Figure A.2e for an example where messages are flowing in opposite directions.

We propose an alternate view of forwarding: as a special kind of message. Treating forwards in this way allows their execution to be deferred until the processes agree on the session types and one of channels involved is empty. We use the session typing system to send a special forward message in the direction of communication according to the forwarding process. In the Figure 2.6 example, Q would send a message along `$d` containing its reference to `$c` before terminating. When R receives the message—and we know it will be receiving because Q sent the message in the direction of communication—`$d` must be empty because because Q terminated after sending the forward message. R will then destroy the empty channel (`$d`) and change its own channel reference to the one from the forward message (`$c`). This maintains the transparency of forwarding: the session type of `$d` from R's perspective when it receives the forward is the same as the type of `$c` from Q's perpective when it executed the forward, so session fidelity is still preserved.

Figure A.2 contains a more detailed example of how forwarding works, including a situation where, because of the asynchronous and concurrent nature of communication, the actual direction of the channel buffers does not coincide with the direction of communication according to the session type (see Figure A.2e). Figure A.3 contains a more complex program where forwarding, combined with sending and receiving channels, allows restructuring of communication to model the restructuring of a list.

Interpreting forwarding as a message allows us to understand forwarding in the context of concurrency, which is important for a shared memory implementation. Section 3.1.3 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. Chapter 5 discusses potential future work in the distributed setting where the forward-as-message interpretation would be imperative for a correct, efficient implementation.

Chapter 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 [5] 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 Shifts

As seen in Figure A.2, the direction of asynchronous communication is not a straightforward notion. There are actually three relevant directions (that may not all agree): the actual direction of messages in the channel buffer and the directions of the next actions of the two processes. The session

typing system for Concurrent C0 as described in [8] uses polarized logic to encode these three directions. Session types are polarized as positive or negative when the provider is sending or receiving, respectively. To transition from a positive to negative type, a process sends a *shift*, and receiving a shift takes the type from negative to positive. Polarity captures the mismatch between provider and client perspectives; a provider that sends a message and then a shift will have negative polarity, but the client will have a positive polarity until it receives the shift. When a shift is received, we know that the provider and client have the same polarity and that the buffer is empty. Receiving a shift is a synchronization point (Section 2.3.2).

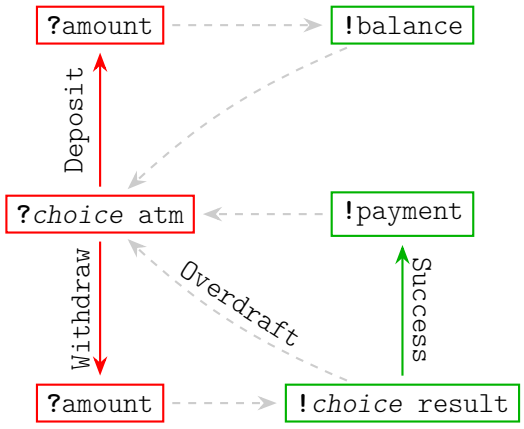
Shifts allow our implementations to keep track communication direction and process polarity without burdening the programmer. The CC0 compiler infers where shifts should go, sending or receiving shifts appropriately when a session type goes from sending to receiving. Shift inference allows programmers to use CC0 without knowing about shifts at all.

In the logic, there are shifts that convey information other than direction changes (explicit synchronization, affine channels), but those features are not presently in Concurrent C0 [8]. Because the CC0 compiler can statically infer the direction of communication, shifts are optimized out of the implementations; they are simply no-ops.

3.1.2 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 the `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.

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 3.1. 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 longest walk in




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

typedef <?choice atm> atm;

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

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

Figure 3.1: Code and graph of type `atm` with width 2.

either the red or green subgraph.

An ATM is a canonical example in the session typing literature, and Figure 3.1 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.

Because width is derived from longest *walk* not and longest path, it can be efficiently calculated. First, decompose the graph into the two subgraphs: sending and receiving. If a cycle is found in either subgraph, the width is infinite, and the communication may buffer an unbounded number of values. If not, then both subgraphs are DAGs, and the longest path can be obtained using a shortest path algorithm on the graph with negated edge weights. Note that the longest walk will never cross between the sending and receiving subgraphs, because those edges represent synchronization points. The compiler passes type width information to the runtime, which uses small, fixed size buffers when possible.

Note that type width is compatible with forwarding because of the forward-as-message interpretation. Section 3.1.3 details how forwards are received *instead of* the intended message, so the forward message will occupy that allocated space.

3.1.3 Forwarding

Forwarding is not typically part of a message passing system, so implementing it safely and efficiently was not obvious. Early implementations of the runtime attempted to carry out forwards as soon as they were executed. These implementations suffered from deadlocks and race conditions from the challenges of forwarding in an asynchronous environment. Section 2.5 presents the forward-as-a-message interpretation that solves those concurrency issues. This view of forwarding made for a much simpler implementation that—because forwarding is now just a message—takes advantage of the message passing functionality already present in the system.

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 to 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. Figure 3.2 shows the definition of the channel data structure. As mentioned in

```
typedef enum {
    TO_PROVIDER,
    TO_CLIENT
} channel_dir_e;

struct channel {
    channel_dir_e queue_dir;
    queue_t* msgs;

    pthread_mutex_t m;
    pthread_cond_t c;
};
```

Figure 3.2: Channel data structure definition in C.

Section 3.1.1, shifts are currently not needed in CC0 because the compiler can the infer the direction of every action. Therefore, the channel data structure only maintains the direction of the message queue. A 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.

3.2.1 Interface

The generated code calls into the runtime for all message passing and process manipulation. The compiler-runtime interface is four main functions and is consistent across all of our runtime implementations.

- `cc0_spawn` creates a new concurrent process and the channel along which it will provide, returning a reference to that channel to the caller (client). `cc0_spawn` 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.

- `cc0_send` sends a given message over a given channel, additionally taking in the message's type and the inferred direction. `cc0_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.

Runtime	Threading	Synchronization
`concur2`	1:1	`pthread` locks and condition variables
`concur3`	1:1	C11 lockless primitives
`concur5`	M:N	Coroutine scheduling and `pthread` locks
`concur6`	M:1	Coroutine scheduling
`go0`	Go M:N	Naive use of Go channels
`go2`	Go M:N	Locks and condition variable design from `concur2`

Figure 3.3: CC0 runtime implementations.

- `cc0_recv` receives a message over a given channel, taking in the message's expected type and the inferred direction. `cc0_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 3.1.3 for details. `cc0_recv` asserts that the received message is of the expected type, panicking if it does not match and is not a forward.

- `cc0_forward` takes in the two channels involved in the forward and the inferred direction of communication. As discussed in Section 2.5, a forward message is sent in the inferred direction containing a reference to the channel in the other direction. `cc0_forward` sends this message using the regular message passing functionality of the runtime and then terminates the calling process.

The `close` and `wait` operations are implemented by sending a receiving a message of special `DONE` type. In the C implementations, `wait` (a `cc0_recv` of the `DONE` type) will deallocate channel and process resources upon receiving the `close`. Go is garbage collected, so this is unnecessary in those implementations.

3.2.2 C Implementations

Figure 3.3 briefly lists the differences in our runtime implementations, specifically the different threading models and synchronization strategies.

`concur2` was our first functional runtime for CC0, and largely determined the runtime-compiler interface. The `pthread` library enables concurrency and parallelism on multi-core machines, and

it also provides mutexes and condition variables. CC0 processes map 1-to-1 onto pthreads which typically map 1-to-1 onto system threads: spawning essentially becomes `pthread_create`, and `wait` becomes `pthread_join`. Channels are protected by a mutex and condition variable, so sending, receiving, and forwarding all require a channel lock.

`concur3` is a variation of `concur2` that uses C11 lockless primitives. Threading is still provided by `pthread`, but channels are not locked. Processes are never blocked in this implementations (there are no condition variables to wait on), instead they use atomic operations to repeatedly poll if queue contains messages or is in the correct direction.

Because CC0 encourages highly concurrent programming, most programs spawn many processes. `concur5` represents an attempt reduce the runtime overhead associated with process management by using an M:N threading model, where M cooperatively scheduled coroutines are multiplexed on top of N system threads. The runtime only spawns as many system threads (using `pthread`) as there are CPU cores, so M is far greater than N, diminishing the overhead associated with rapidly creating tens of thousands of processes. For `concur5`, we created our own M:N threading solution, and we reimplemented the necessary synchronization primitives (mutexes and condition variables) as well.

However, creating an efficient M:N scheduler proved difficult (see Section 4.1), so the `concur6` runtime is a version of `concur5` that maps all of processes to a single system thread. This vastly simplified scheduling and synchronization, and it is worth discussing in its own right. While simpler and ultimately faster than `concur5`, the cost of `concur6` was the loss of parallelism, one of Concurrent C0's primary goals.

3.2.3 Go Implementations

The performance of the single-threaded `concur6` runtime demonstrated that lightweight threading was indeed an effective way to implement Concurrent C0. Instead of further pursuing a state-of-the-art M:N threading system, we retargeted the CC0 compiler to Go. Go is imperative programming language with a lightweight threading model that provides concurrency and efficient parallelism [9]. Go also features message passing over channels, which provided both a tool to implement

Concurrent C0 and something to compare our more specialized implementation against.

The `go2` runtime is nearly a direct port of the `concur2` runtime, but instead of spawning processes with `pthread_create`, it uses the `go` command that spawns a new goroutine, the unit of concurrent execution in Go. Because each CC0 processes is mapped to a goroutine, this implementation benefits from Go's high performance M:N scheduler. Like `concur2`, `go2` uses mutexes and condition variables for synchronization; it does not use the channels Go provides by default. Section 4.2 compares the performance of Go's default channels and CC0's channels implemented in Go.

Chapter 4

Experimental Comparison and Analysis

The language based optimizations detailed in Chapter 3 allow our implementations to make tangible performance improvements over other message passing techniques. We analyze the performance of our C runtimes in Section 4.1 which demonstrates the effectiveness (or lack thereof) of the different threading models and synchronization techniques in those implementations. In Section 4.2, we compare our highest performing runtime, `go2`, against a specially created runtime that mimics how message passing is typically done in Go.

To benchmark Concurrent C0, we created a benchmarking suite (Figure 4.1) consisting of many highly concurrent data structures, like the queue in Figure A.1. Most of the work done in these tests is communication, highlighting 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.

4.1 C Implementation Analysis

The benchmarking results for the C runtimes (Figure 4.2) are limited by the poor performance of `concur3` and `concur5`. Note that the plot uses a log scale; `concur5` performs orders of magnitude worse than other runtimes in several cases. Due to the long running times, some liberties were

`bitstring1.c1`	w/ external choice	`parfib.c1`	parallel naive Fibonacci
`bitstring3.c1`	w/ internal choice	`primes.c1`	prime sieve (sequential)
`bst1.c1`	BST reduce operation	`queue-notail.c1`	queues without tailcalls
`insert-sort.c1`	insertion sort using list	`queue.c1`	queues written naturally
`mergesort1.c1`	w/ Fibonacci trees	`reduce.c1`	reduce/scan parallel seqs
`mergesort3.c1`	w/ binary trees	`seg.c1`	list segments
`mergesort4.c1`	w/ binary trees, seq. merge	`sieve-eager.c1`	eager prime sieve
`odd-even-sort1.c1`	odd/even sort, v1	`sieve-lazy.c1`	lazy prime sieve
`odd-even-sort4.c1`	odd/even sort, v4	`stack.c1`	a simple stack
`odd-even-sort6.c1`	odd/even sort, v6		

Figure 4.1: A short explanation of the tests in the CC0 benchmarking suite. The tests and benchmark results can be found at `http://maxwillsey.com/assets/cc0-thesis-benchmarks.tgz`

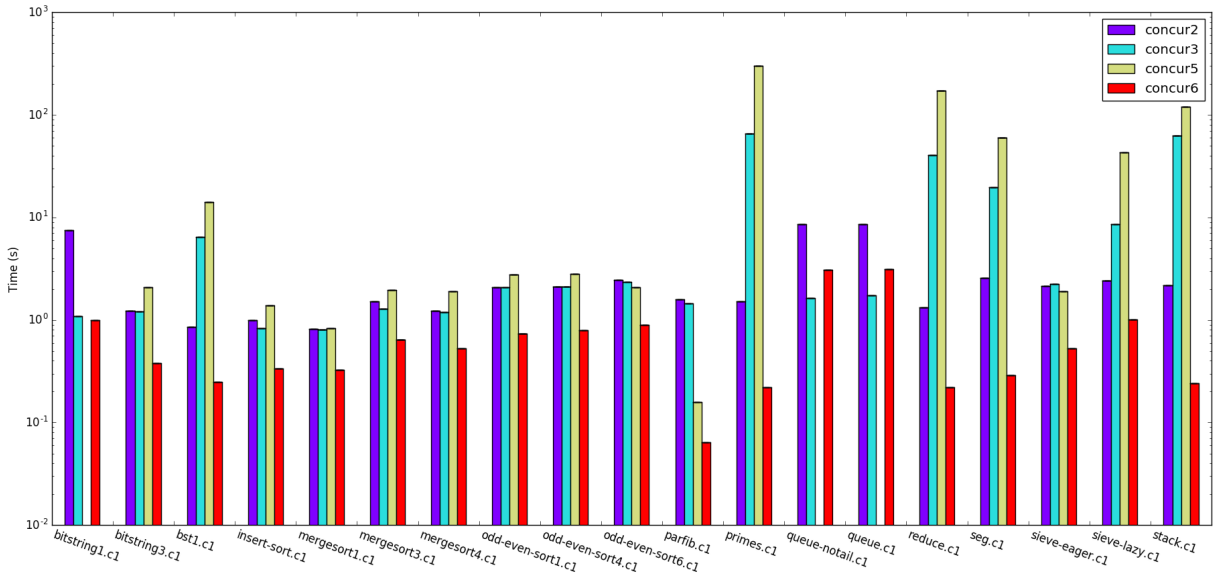


Figure 4.2: C runtime benchmarks on a log scale. Note that only one sample was taken, and some results omitted completely, because of the poor performance of `concur5`.

taken with benchmarking the C runtimes: the parameters of the tests were tweaked to make them shorter, only one sample was taken, and `concur5`'s results for `bitstring1.c1`, `queue-notail.c1`, and `queue-tail.c1` were omitted for not terminating in a reasonable amount of time. The more "interesting" runtimes from a performance perspective, `concur6` and the Go runtimes, are given a more rigorous analysis in Section 4.2.

Comparing `concur2` and `concur3` shows that the lockless approach yielded mixed results. In some cases with high parallelism, where most processes are runnable all the time (like `queue.c1`), the lockless approach saw considerable improvement. However, in cases where many threads are blocked, the spinning associated with the lockless synchronization led to significant regressions. In `primes.c1`, where only one thread is runnable at any time, `concur3` regressed almost two orders of magnitude from `concur2`. These results led us to abandon the lockless strategy in future implementations.

The performance of `concur6` highlights how efficient lightweight threading is as an implementation for Concurrent C0. The disparity between `concur5` and `concur6`, which are largely similar, demonstrates the challenges to efficiently implementing M:N threading. `concur6` implements the M:1 model, forgoing the potential parallelism of 4 cores in favor of a much simpler scheduling and synchronization strategy, because all the processes are running on one system thread. We speculate that the dramatic slowdown of `concur5` can be attributed to poor scheduling, specifically extremely high contention on the scheduler queue. By eliminating that contention, the admittedly simple `concur6` runtime significantly outperforms every other C runtime, even by orders of magnitude in many cases.

4.2 Go Implementation Analysis

To compare Concurrent C0 to more typical message passing techniques, 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 commu-

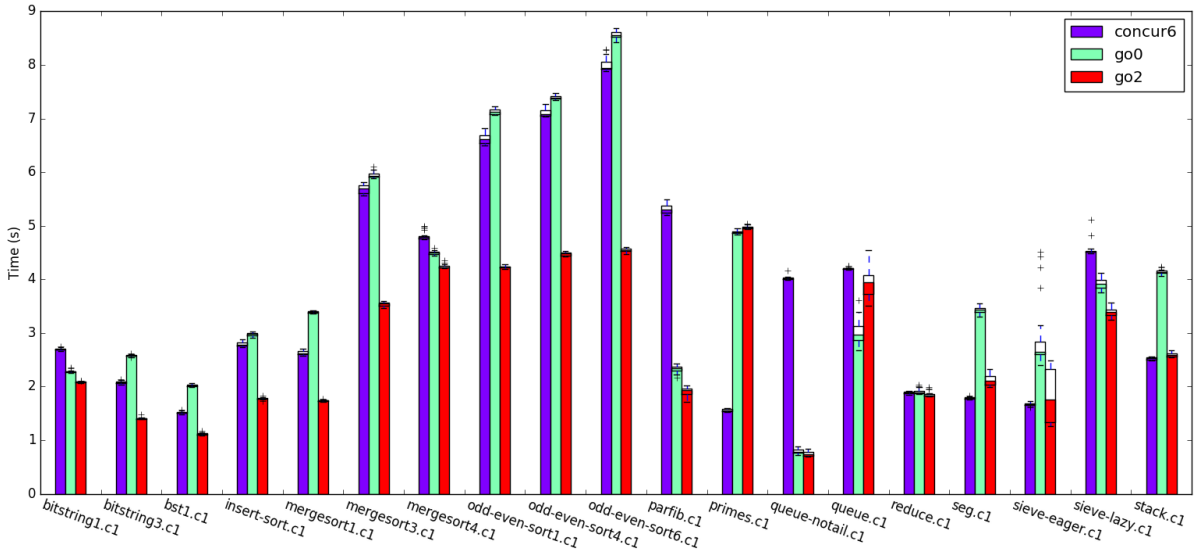


Figure 4.3: Go runtime benchmarks with `concur6`. The bars are drawn at the median of 20 samples, and the overlaid boxplots additionally display the maximum, minimum, 1st and 3rd quartiles, and outliers of the results. Result data can be found at `http://maxwillsey.com/assets/cc0-thesis-benchmarks.tgz`

nication), but it conforms to the same interface as our other implementations so we can run the same tests against it.

Figure 4.3 shows the results of the Go runtimes `go0` and `go2` benchmarked with `concur6` for comparison. Recall that `go0` closely models conventional message passing (using Go channels), and `go2` leverages all of our language based optimizations: type width for buffer bounding, using only one buffer, and automatically synchronizing the direction of communication. Both `go0` and `go2` use the same threading model, so comparing them demonstrates the effectiveness of our implementation techniques. Compared to the naive implementation, our optimized version ran 1.38× faster on average[1].

These results also demonstrate that our implementation takes advantage of parallelism when it exists. `concur6` and `go2` share the same language based optimizations, but `concur6` uses a custom M:1 threading model and `go2` uses Go's M:N threading model. In test cases with high parallelism like `queue-notail.c1` and `parfib.c1`, where many processes are runnable most of the time, we see between 2× and 4× speed up from the sequential `concur6` to the parallel `go2`.

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

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 2.3.1. Because Go does not optimize tail calls, their extensive use causes the stack to grow rapidly. Go starts goroutines (which are used to implement CC0 processes) with a very small stack, copying the stack to another location when it needs to grow. Profiling results suggest that up to half of the running time of some of our test cases is spent by Go copying stacks.

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 on these tests indicates that Go's lack of tail call optimization 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.

Chapter 5

Conclusions and Future Work

Concurrent C0 makes writing highly concurrent programs using message passing easier, safer, and more efficient than other languages. A concise, C-like syntax allows programmers to use a familiar imperative style to safely create and manipulate concurrent processes. The forwarding operation and linear channel variable semantics enable safe modifications to the process tree, even outside of fork/join style operations. Session types ensure that message passing adheres to the expected protocols, and they also give insight into the overall communication structure of the program. Our implementations exploit this knowledge to make optimizations that shrink some buffers and eliminate others, increasing performance while still providing bidirectional communication.

The knowledge given by session types could also be used to inform scheduling decisions at runtime. 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 pointers 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[1], which would disallow multiple CC0 processes manipulating shared state.

Session types are traditionally discussed in the setting of distributed computing, so a distributed implementation of Concurrent C0 could carry some of the contributions of this work into that space. 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. Message passing is already common in distributed systems, so our forwarding interpretation could easily be implemented in existing systems.

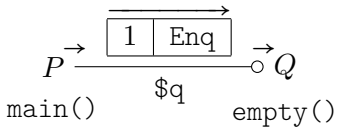
[1] `https://doc.rust-lang.org/book/ownership.html`

Appendix A

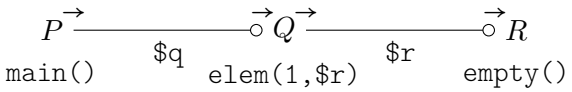
Examples

```
// external choice for request
choice queue {
  <?int; ?choice queue>  Enq;
  <!choice queue_elem>   Deq;
};

// internal choice for response
choice queue_elem {
  <!int; ?choice queue> Some;
  < >                   None;
};

typedef <?choice queue> queue;

// provider holds element x and
// points to rest of the queue $r
queue $q elem (int x, queue $r) {
  switch ($q) {
  case Enq:
    int y = recv($q);
    $r.Enq; send($r, y);
    $q = elem(x, $r);
  case Deq:
    $q.Some; send($q, x);
    $q = $r;
  }
}

// provider for end of queue
queue $q empty () {
  switch ($q) {
  case Enq:
    int y = recv($q);
    queue $e = empty();
    $q = elem(y, $e);
  case Deq:
    $q.None;
    close($q);
  }
}

void dealloc (queue $q) {
  $q.Deq; switch($q) {
    case Some:
      recv($q);
      dealloc($q);
      return;
    case None:
      wait($q);
      return;
  }
}

int main () {
  queue $q = empty();
  $q.Enq; send($q, 1);
  $q.Enq; send($q, 2);
  dealloc($q);
  return 0;
}
```

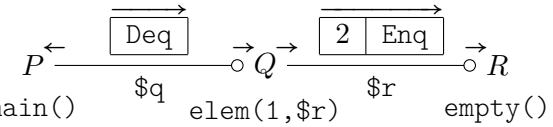
Figure A.1: `queue.c1`, a queue implementation where each element is a concurrent process.



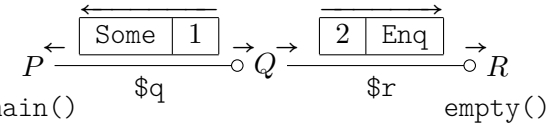
(a) P enqueues 1 [line 56].



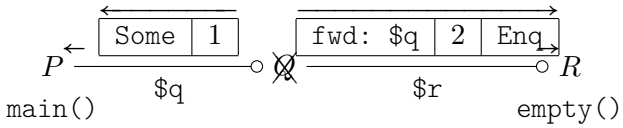
(b) Q gets the enqueue, spawning a new `empty()` process and channel [line 34–35].



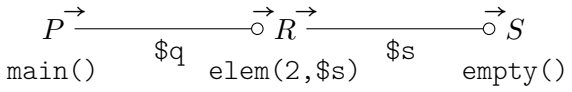
(c) P enqueues 2 [line 57] which Q passes to the back of the queue [line 21]. P sends a dequeue request and waits for the result [line 43].



(d) Q responds to the dequeue [line 24] and is about to forward [line 25].



(e) Q forwards `$q = $r` [line 25] by sending a forward message in the direction of communication according to the session type, not the state of the channel buffers. Q terminates, but `$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 34–35]. When R receives the forward, it deallocates `$r` (which is now safe because `$r` is empty) and will now use `$q` instead.

Figure A.2: An illustration using `queue.c1` (Figure A.1) demonstrating how treating forwarding as a message resolves communication direction issues. 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.

```
choice list {
  <!int; !choice list> Cons;
  <>                   Nil;
};

// a Lisp-style list
typedef <!choice list> list;

// first requests the list tail
// then behaves like the whole list
typedef <?list; list> seg;

list $c nil() {
  $c.Nil;
  close($c);
}

list $c cons(int x, list $d) {
  $c.Cons;
  send($c, x);
  $c = $d;
}

seg $c empty() {
  list $tail = recv($c);
  $c = $tail;
}

seg $c concat(seg $d, seg $e) {
  list $tail = recv($c);
  send($e, $tail);
  send($d, $e);
  $c = $d;
}

seg $c prepend(int x, seg $e) {
  list $tail = recv($c);
  send($e, $tail);
  $c = cons(x, $e);
}

seg $c append(seg $d, int x) {
  list $tail = recv($c);
  list $e = cons(x, $tail);
  send($d, $e);
  $c = $d;
}

int read_elements(seg $d) {
  list $nil = nil();
  send($d, $nil);
  int sum = 0;
  while (true) {
    switch ($d) {
    case Nil:
      wait($d);
      return sum;
    case Cons:
      sum += recv($d);
      break;
    }
  }
}

int main() {
  int n = 500;
  seg $c = empty();
  seg $d = empty();
  for (int i = 0; i < n; i++) {
    $c = append($c, i);
    $d = prepend(n-i-1, $d);
  }
  seg $e = concat($c, $d);
  int sum = read_elements($e);
  assert(sum == n*(n-1));

  return 0;
}
```

Figure A.3: `seg.c1`, a list implementation supporting prepending, appending, and concatenation where each element is a concurrent process. The `nil` and `cons` functions provide the `list` type, an internal choice representing a singly-linked list in the style of Lisp. The `seg` type is higher-level abstraction that first receives a list then behaves like a list.

Bibliography

[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.

[4] Mariangiola Dezani-Ciancaglini and Ugo de'Liguoro. "Sessions and session types: An overview". In: *Web Services and Formal Methods*. Springer, 2010, pp. 1–28.

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

[6] Kohei Honda. "Types for Dyadic Interaction". In: *4th International Conference on Concurrency Theory*. CONCUR'93. Springer LNCS 715, 1993, pp. 509–523.

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

[8] 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. To appear. London, England: Springer LNCS, Apr. 2015.

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

[10] 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.