Carnegie Mellon University

Design and Implementation of Concurrent C0

Max Willsey Advisor: Frank Pfenning

Concurrent C0

Concurrent C0 (CC0) is an extension to the C0 programming language that supports safe concurrent message passing using session types. Session typing allows us to encode the type of a communication protocol: sequences of types represent how the type changes as the program executes. Each type in a sequence is designated as sending (`!`) or receiving (`?`), encoding the direction of communication. Communication occurs between pairs of processes in which one is called the *provider* and the other is the *client*. 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.

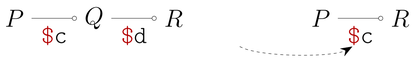
The fundamental unit of concurrency in Concurrent C0 is the *process*. Processes are *spawned* by functions that return a channel variable (denoted with the `$` sign). When a spawning function is called, a new channel is returned immediately to the caller, who is the client. A new process is then created and executes the function concurrently, providing from the other end of the channel. The following code sample implements queues in CC0 with fine grained concurrency, where each element is contained in its own process:

```
// 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;
}
```

Forwarding

Forwarding allows a process to terminate, letting the provider and client communicate directly. The middle process Q combines two channels of the same session type by executing the forward `$c = $d`, so the two endpoints can communicate without the process in the middle. At a very high level, forwarding can be thought of as setting a channel equal to another channel, but its semantics and implementation are more complicated. How do we choose `$c` to persist? What happened to the messages in `$d`?



We implement forwarding as a special kind of message, deferring the execution 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. 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 communication is still safe.

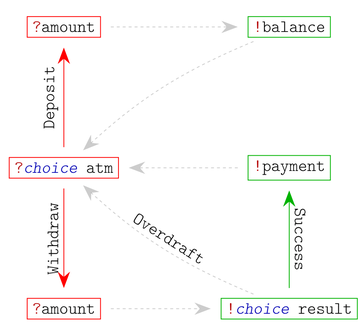
Optimizations

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

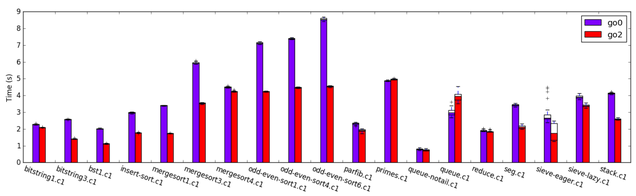
Session types dictate that communication is only in one direction at a time, and in certain cases only so many values can be buffered at a time. For example, the type `<!bool; ?int;>` could only possibly buffer one value at a time, because the `int` must be sent from the client, which can only occur 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. We know that the buffer will only contain messages going in one way at a time, so there are actually two graphs, one sending (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 either the red or green subgraph. The ATM protocol shown to the right has a width of 2.

```
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;
};
```



Results



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

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.

We compare `go0` against `go2`, a Go implementation which uses the full suite of language based optimizations. Our benchmarking suite consists of many highly concurrent data structures, like the queue on the left. Most of the work done in these tests is communication, so as to highlight the efficiency of our message passing runtimes.

The `go0` vs. `go2` benchmark demonstrates the effectiveness of our implementation techniques. Compared to the naive implementation, our optimized version ran 1.38× faster on average. We suspect that the speed up would be even more dramatic if the Go compiler optimized tail calls, because CC0 encourages a tail recursive style of programming. 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.

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.

[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.