Annual Report for CCF 1116725
SHF:Small:Reversible Concurrency

1 Project Goals and Status

The primary deliverables of this research project are an experimental language for reversible computing in a concurrent environment and models for reasoning about this language. The project includes experimental work to implement and test linguistic constructs and theoretical work to provide both formal models for these constructs, and algebraic tools to enable reasoning about programs that utilize them. The three major activities of this project along with their expected fraction of the total effort are:

1. Language design and implementation [45%]
2. Theory and formal models [45%]
3. Language application [10%]

In the first 9 months of this project we have focused upon tasks 1 and 2 and have made significant headway with both, as we outline in this report. We do note that the project, which provides funding for two graduate research assistants, got off to a slow a start with only one RA (William Bowman) working during the period August 2011-May 2012. William has worked on prototyping the language design, which has now reached a fairly stable point; he is however leaving Indiana University to join the Ph.D. program at Northeastern University in Fall 2012. We have attracted a current Ph.D. student at Indiana University, Zachary Sparks, to investigate the formal aspects of the language prototype during Summer 2012. We have also attracted a new Ph.D. student, Edward Amsden, from the Rochester Institute of Technology, starting Fall 2012. We hope to attract another graduate student in the next academic year and we also plan on taking advantage of the funding for undergraduates in the next two summers.

The remainder of this report discusses the current status of our research on formal models of reversible synchronous communication (Sec. 2) and on the language design and implementation (Sec. 3). The last section concludes with detailed plans for the next academic year.

2 Reversible Synchronous Communication

The first focus of our research in the past year has been on designing a high level protocol for reversible synchronous communication.

2.1 Background

To support reversible computing in a distributed environment, we decided to focus upon synchronous channel-based communication as explored in process algebras such as CSP and CCS. Taking the CSP model as an example, suppose processes P and Q share a channel c. Consider a process P which (attempts) to send a value v along channel c and thereafter behave like P':

$$P : c!v \rightarrow P'.$$

Symmetrically, Q might agree to receive a value x over c and thereafter behave like $Q'(x)$:

$$Q : c?x \rightarrow Q'(x)$$

If we consider the parallel composition of P and Q then they behave as:

$$(P \| Q) \rightarrow (P' \| Q'(v))$$

The key idea of this model is that communication involves agreement and hence synchronization between partners. The model becomes more interesting if choice is introduced – for example, we might define a process R which is prepared to accept communication over channels d or e:

$$R : (d?x \rightarrow R'(x)) [\!] (e?x \rightarrow R''(x))$$

If both communication options are offered, then R is permitted to choose which one to accept. The general case, where choice is permitted among arbitrary communication events, is difficult to accomplish in a distributed implementation; however, if we are willing to accept some restrictions on choice, for example, limiting choice to input, then this model can be efficiently implemented. Indeed this is the design decision made for Occam and for various hardware implementations of CSP.

When choice is asymmetric, synchronous communication can be implemented in a delay-insensitive manner. Consider a widely used hardware implementation that utilizes three signals to transmit a value, as illustrated in Figure 2.1. The sender generates request (req) and data, and the receiver generates the signal (ack).

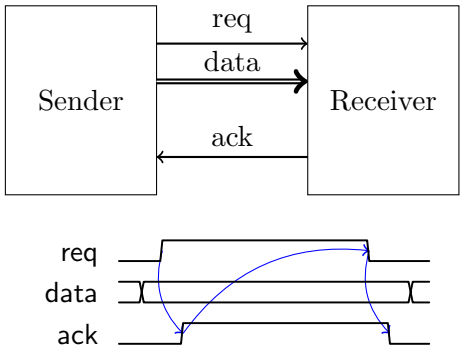

Figure 1: Hardware Handshake

In a message passing environment, the sender sends tuples (req,data) and receives (ack). As long as these are sent along lossless paths and delivered in-order, the same protocol will work.

2.2 Reversibility

At this point we divert from the known to create a protocol suitable for use in a reversible world. Our goal is to implement communication channels that support retraction of communication events. We assume that every process has a local clock that, with forward computation, advances on every communication event; and, with reverse computation, retreats on every withdrawn (negative) communication event. Furthermore, when a forward communication event occurs, both partners advance to a common local time. Similarly, for negative computation events, both partners retreat to common local times. Thus, communication synchronization has the effect of also synchronizing local clocks!

In the following, we ignore the transmitted data and focus solely on the "handshake" portions of the protocol. Rather than binary request and acknowledge signals, we introduce three values (rev, idle, fwd) which are transmitted along with local timestamps. Our working environment is the Symbolic Analysis Laboratory (SAL) which we have been using to verify our protocol development. To give a sense of the model development we present a fragment of a naive model in Listing 1. This fragment shows one transition each for the sender (tx) and receiver (rx). In the rule present, the sender moves to the fwd state (equivalent to req transitioning from 0 to 1 in the hardware protocol). In the process, it selects a new time that is greater than the last time used on the channel. Similarly, the receiver transitions to the fwd state selecting a time that is at least as great as that used by the sender – the protocol does not, and need not capture the relationship between this new time and the local clocks. At the final phase of the communication (not shown), the sender and receiver will have agreed to the new time. Also not shown is the manner in which the protocol handles negative communication.

We have validated the basic protocol from which the illustrated fragment was excerpted – most importantly in the context of communication delay. This first protocol lacks some necessary features. We have extended it to add some of these and are working on further extensions as our understanding of the programming model advances.

3 Language Design and Implementation

The aim is to develop programming constructs which can be used to write reversible communicating processes and whose implementation relies on the protocol described above. In particular, a process should be able to perform a speculative computation, and backtrack when it or some other process decides the computation should be restarted with new values, or a new computation should be done altogether. All communication that happened between the start of the speculative computation and the backtrack signal should be undone, perhaps causing other processes to backtrack as well.

We have approached this problem by considering a core parallel language with synchronous communication and experimenting with various continuation-based operators to model backtracking. Briefly, when a process decides to backtrack to a choice point, it restores the continuation present at the time the choice point was created, using the next choice in the sequence of choices provided to fill the continuation. A choice point might look like choose k e_1 e_2. In this computation, first e_1 would be performed. If this process backtracks to this choice point by using backtrack k, it would then perform e_2 instead.

We now describe in more detail the programming constructs in our current prototype, the small-step reduction semantics, and present several small illustrative examples, and discuss their implementation in the Redex tool.

```
 STATE : TYPE = { rev, idle, fwd };

 tx : MODULE =
  BEGIN
   INPUT  rxstate   : STATE
   OUTPUT txstate   : STATE
   INPUT  ack       : TIME
   OUTPUT req       : TIME

   INITIALIZATION

     txstate = idle;
     req = 0;

   TRANSITION
   [
     (rxstate = idle) AND (txstate = idle) -->
                      txstate' = fwd;
                      req' IN { x : TIME | x > req}

   [] ...
   ]
   END;

rx : MODULE =
    BEGIN
        INPUT  txstate   : STATE
        OUTPUT rxstate   : STATE
        INPUT  req       : TIME
        OUTPUT ack       : TIME

    INITIALIZATION

        rxstate = idle;
        ack = 0;

    TRANSITION
    [

       (txstate = fwd) AND (rxstate = idle)   -->
                       rxstate' = fwd;
                       ack' IN { x : TIME | x >= req}

    [] ...
    ]
   END;
```

Listing 1: Fragment of Naive SAL Model

Syntax and semantics. We augment a core functional sequential language with parallel reductions as follows:

$$\frac{e_1 \rightarrow e_1' \qquad e_2 \rightarrow e_2'}{e_1 \parallel e_2 \Rightarrow e_1' \parallel e_2'}$$

That is, the parallel reduction is simply defined by allowing each process to perform the small-step reduction in parallel. The language is extended with two primitive forms that allow synchronous

communication. A sample reduction rule follows:

$$P_1, E_1[\text{send } P_2\ v] \parallel P_2, E_2[\text{recv } P_1] \Rightarrow P_1, E_1[()] \parallel P_2, E_2[v]$$

where E_1 and E_2 are evaluation contexts which represent the contexts in which the communication is happening in each process.

A choice point creates a continuation with a hole that can be filled with one of a sequence of expressions. The continuation can be invoked with the next expression in the sequence by backtracking to it. Any process can at anytime request that another process backtracks. This attempt will succeed if the process being forced to backtrack has not "committed" (see the description of **commit** below). A choice point will cycle through the sequence of choices as long as the process backtracks to the choice points, and as long as the choice selected choice is not an error. If the selected choice causes an error, or is an error statement, the cycle will stop with the process signaling an error. Semantically, choice points create a new mapping from unique continuation labels to continuations and place them in a process-local store:

$$S; E[\text{choose } k\ e_1\ e_2\ \cdots \parallel P_2 \cdots \Rightarrow S, k \rightarrow \{E[\text{choose}k\ e_2\ \ldots e_1]\}; E[e_1] \parallel P_2 \ldots$$

where S is the process-local store.

A choice point executed by process P introduces a continuation name k in P's local store. In our current prototype, we consider the name k to be global, eventually to be realized using a timestamp mechanism as described in the previous section. In other words, we allow any process to use the name k to request that process P backtracks. Since the backtrack request may be rejected, the process initiating the request specifies an alternative action:

$$S_1; E[\text{backtrack } P\ k_1\ e_1] \parallel S_2, k_1 \rightarrow E_2[e]; e_2 \parallel P_3 \cdots \Rightarrow S_1; E[()] \parallel S_2, k_1 \rightarrow E_2[e]; e \parallel P_3 \ldots$$

The above reduction models the situation in which the backtrack requested is accepted. Otherwise the following reduction applies:

$$S_1; E[\text{backtrack } P\ k_1\ e_1] \parallel S_2; e_2 \parallel P_3 \cdots \Rightarrow S_1; E[e_1] \parallel S_2; e_2 \parallel P_3 \ldots$$

A *commit point* allows a process to remove all continuations from its store prior to a specified point. By treating unique continuation labels as having a linear order, then given k, all mappings $k_1 \rightarrow e_1, \ldots k_n \rightarrow e_n$, where $k_1 < k_n < k$, are removed from the local store. This effectively means the process will never backtrack to a point prior to k. If another process tries to force the committed process to do so, the backtrack will fail and the targeting process will take its alternate path:

$$S, k_1 \rightarrow e_1, S'; E[\text{commit } k_1] \parallel P_1 \cdots \Rightarrow S'; E[()]$$

Examples. This first example shows how three processes can use local speculative choices to try and satisfy a simple global invariant. Each process has a list of numbers from which it selects one number n_i such that to satisfy the global invariant that $n_1 < n_2 < n_3$. Operationally, the processes are linked in a chain. Each process speculatively chooses a number and communicates it to the previous one in the chain. If the received number is out of order, the receiving process tries first to backtrack locally and if it can still not satisfy the invariant asks the sender to backtrack.

```
(define e25
  (par-term
    (id i_0
```

```
        (let x = (recv i_1) in
          (let y = (recv i_2) in
            (let z = (recv i_3) in
              (if (< x y)
                (if (< y z)
                  (err "Success")
                  (err "Fail"))
                (err "Fail"))))))
    (id i_1
        (seq (choose k_0 unit (backtrack i_2 k_1 (err "Error")) (err "Fail"))
             (let x = (recv i_2) in
               (let y = (choose k_1 5 3 (backtrack i_1 k_0 (err "Error")) (err "Fail")) in
                 (if (< y x)
                   (seq (send i_0 y)
                        (commit k_1))
                   (backtrack i_1 k_1 (err "Error")))))))
    (id i_2
        (seq (choose k_0 unit (backtrack i_3 k_1 (err "Error")) (err "Fail"))
             (let x = (recv i_3) in
               (let y = (choose k_1 6 2 (backtrack i_2 k_0 (err "Error")) (err "Fail")) in
                 (if (< y x)
                   (seq (send i_1 y)
                        (send i_0 y)
                        (commit k_1))
                   (backtrack i_2 k_1 (err "Error")))))))
    (id i_3
       (let y = (choose k_1 7 2 (err "Fail")) in
         (seq (send i_2 y)
              (send i_0 y)
              (commit k_1))))
    ))
```

The second example abstracts a scenario in molecular biology investigated by Danos and Krivine. Abstractly speaking, we again have three processes and each process has a list of numbers. The first two processes are willing to send their numbers to the third process. The third process however is only willing to accept a particular sequence of numbers. The example is interesting in our setting because it illustrates that backtracking allows processes to choose among several communication options. In the absence of backtracking, the last process, when offered two possible communications, would never be able to commit to either.

```
(define e28
  (par-term
   (id i_0 (for x in (cns 0 (cns 0 (cns 0 (cns 0 unit))))
             do
             (send i_2 x)))
   (id i_1 (for x in (cns 1 (cns 1 (cns 1 (cns 1 unit))))
             do
             (send i_2 x)))
   (id i_2 (for x in (cns 1 (cns 0 (cns 1 (cns 0 unit))))
             do
             (let y = (choose k (recv i_0) (recv i_1)) in
               (if (eq? x y) unit (backtrack i_2 k (err "what")))))))))
```

Both examples execute as expected in our Redex model. Yet, we believe that there is possible room for improvement in designing the "right" primitives to enable programming larger applications.

4 Goals for next Year

The goals for next academic year are to:

- finalize the Redex model, formalize it, and publish a paper about the language design and its properties in a venue focused on language design, and
- extend the capabilities of the protocol for reversible communication and its proof of correctness and publish the result in a venue focused on automated verification.

If both activities are successful, the next natural steps are to "overlay" the protocol on top of the language model and establish the correctness of a suitably abstracted language implementation.