1 Introduction

The language we're trying to develop is a simple parallel language, with synchronous communication, and the ability to backtrack. A process should be able to perform a speculative computation, and backtrack when some 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've approached this problem by creating choice points, each with multiple choices. 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:

$$E[\text{choose } k\ e_1\ e_2]$$

In this computation, first $E[e_1]$ would be performed. If this process backtracks to this choice point, it would then perform $E[e_2]$.

A process could backtrack via

$$E[\text{backtrack } k]$$

.

2 Current Language

The current language as modeled in redex is a simple scheme-like language, extended to a parallel language with synchronous communication, choice points, 'failing backtracking', sync points, and commit points.

2.1 Parallelism

The language is extended to be parallel by creating a reduction for a simple scheme-like language, and defining a new reduction 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.

2.2 Synchronous communication

The parallel language is extend with two primitive forms that allow synchronous communication. A sample reduction rule follows:

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

2.3 Choice points

A choice points 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 try to force another process to backtrack, including itself. This attempt will succeed if and only if the process being forced to backtrack has not 'commited' (see 2.6).

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, of course the cycle will stop with the process signaling an error.

Choice points create a new mapping from 'k-vars' (a unique name for the continuations) to continuations and place them in a process-local store.

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

2.4 Failing backtracking

Any process can causes another to backtrack at anytime. In previous version of the language/model, this always causes the target process to backtrack. In this version, however, if the target process has committed, the backtrack will fail. The backtrack form thus now requires an alternate expression, to be executed if the backtrack fails.

To target a process and force it to backtrack, it's necessary to specify the process ID and the k-var denoting where the process should backtrack to.

$$S_1; I_1; E[\text{backtrack } I_2\ k_1\ e_1] \parallel S_2, k_1 \rightarrow E_2[e]; I_2; e_2 \parallel P_3 \cdots \Rightarrow S_1; I_1; E[()] \parallel S_2, k_1 \rightarrow E_2[e]; I_2; e \parallel P_3$$

$$S_1; I_1; E[\text{backtrack } I_2\ k_1\ e_1] \parallel S_2; I_2; e_2 \parallel P_3 \cdots \Rightarrow S_1; I_1; E[e_1] \parallel S_2; I_2; e_2 \parallel P_3 \ldots$$

if $k_1 \notin S_2$

2.5 Sync points

Intuitively, sync points allow two process to synchronize, and specify some expression to be executed if the process backtracks to a specified choice point. We used sync points in an earlier model which lacked choice points, but allowed backtracking to a point without a new choose to execute. It's not obvious if we still need sync points, but they seem to provide a nice abstraction, so I've left them in. They also seem to give a bit of expressiveness that we can't from the other primitives.

Sync points are implemented by modified the stored continuation and tacking on the expression to execute at the front of the continuation. Thus, if the sync point is executed, and the process backtracks, it will perform some extra work before executing the continuation at it's choice point.

$$S_1, k_1 \to e_1; I_1; E_1[\text{sync } I_2\ k_1\ e_3] \parallel S_2, k_2 \to e_2; I_2; E_2[\text{sync } I_1\ k_2\ e_4] \parallel P_3 \cdots \Rightarrow$$

$$S_1, k_1 \to (\text{begin } e_3\ e_1); I_1; E_1[()] \parallel S_2, k_2 \to (\text{begin } e_4\ e_2); I_2; E_2[()] \parallel P_3 \ldots$$

2.6 Commit points

A commit point allows a process to remove all continuations from it's store prior to a specified point. By treating k-vars as having a linear order, then given k, all mappings $k_1 \to e_1, \ldots k_n \to e_n$, where $k1 < k_n < k$, 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 it's alternate path.

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