Project 2
Join Calculus

CMSC 32102

Version 1.2
May 27, 2003

1 Introduction

The join caluclus is a concurrent programming model proposed by Fournet and Gonthier [FG96]. The join calculus is based on asynchronous message passing. In the abstract, the join calculus has the following form:

Processes

$$\begin{array}{rcl} P & ::= & c(v) \\ & | & \texttt{let}\ D\ \texttt{in}\ P \\ & | & P \mid P \end{array}$$

Definitions

$$\begin{array}{rcl} D & ::= & J{=}{>}P \\ & | & D\ \texttt{or}\ D \end{array}$$

Join patterns

$$\begin{array}{rcl} J & ::= & c(x) \\ & | & J\ \texttt{and}\ J \end{array}$$

where c is a channel name, v is a value, and x is a variable. A process P is either a message, a definition, or the parallel composition of two processes. A definition consists of one or more guarded processes, where the guards are join patterns. A join pattern is the conjunction of one or more message patterns. A guard is enabled when there are messages available that match all of its patterns (messages are matched by channel name). For example, the following process waits for messages on two out of three channels and sends the result:

```
signature JOIN =
  sig

    type 'a jchan
    type 'a join

    val channel : unit -> 'a jchan

    val recv : 'a jchan -> 'a join

    val join : ('a join * 'b join) -> ('a * 'b) join

    val send : 'a jchan * 'a -> unit

    val match    : 'a join -> 'a
    val matchEvt : 'a join -> 'a CML.event

  end
```

Figure 1: Join interface

```
let j1(a) and j2(b) => res(a, b)
 or j2(a) and j3(b) => res(a, b)
 or j3(a) and j1(b) => res(a, b)
 in ...
```

Notice how the declaration form is similar to a CML `select`.

2 The project

For this project, you must implement a join abstraction in CML (*i.e.*, an abstraction that represents the *J* part of the Join calculus). You should package your implementation in a module name `Join` that matches the interface in Figure 1. The semantics of this interface is as follows:

`type` `'a jchan` is the type of a join channel.

`type` `'a join` is the type of a join of one or more join channels.

`channel ()` returns a new Join channel.

`recv jch` returns the singleton join for receiving messages on the join channel `jch`.

`join (`*`j1`*`, `*`j2`*`)` returns the join of the two joins *`j1`* and *`j2`*.

`send (`*`jch`*`, `*`msg`*`)` sends the message *msg* on the join channel *jch*.

`matchEvt` *j* returns an event value for synchronizing on matching the join *j*.

`match` *j* blocks the calling thread on the join *j* until there is a match.

An example of the use of this API, the following code defines a function (`f`) that waits for messages on any two of three given join channels:

```
structure J = Join

fun f (j1, j2, j3) = CML.select [
            matchEvt(J.join(J.recv j1, J.recv j2)),
            matchEvt(J.join(J.recv j2, J.recv j3)),
            matchEvt(J.join(J.recv j3, J.recv j1))
      ]
```

This code implements the same pattern as the example in Section 1.

The complete project is due on June 10th. You should mail me a compressed tarball of your sources with a `README` file that gives an overview of your solution.

3 Hints

The implementation of the `matchEvt` function will require using the `withNack` combinator to create a "transaction manager" thread at synchronization time. The transaction manager will have to have to query the channels involved in the join for their initial state and then monitor the changes in state (the state of a channel can be viewed as the number of available messages).

One tricky part of the problem is getting the types right. You will have to separate the tracking of channel states from the assembly of the final result. Also, your implementation should be able to handle patterns of the following form:

```
J.join(J.recv j, J.recv j)
```

which requires two messages on channel `j`.

You implementation should be built on CML primitives (you do not need operations from the `Unsafe` module or continuations). Furthermore, your implementation should *not* be monolithic (*i.e.*, do not use a single server thread to manage all join transactions).

4 Document history

1.2 (5/27/03) Fixed a couple of typos and clarified project description.

1.1 (5/20/03) Fixed example code to match changes in API.

1.0 (5/20/03) Initial release.

References

[FG96] C. Fornet and G. Gonthier. The reflexive chemical abstract machine and the join-calculus. In *POPL'96*, January 1996.