Sketch design for a Prolog multiple-process facility

Fernando Pereira
10/5/86

1. Origins

This design, which I will be working on for a while (ie., the current version is almost certainly not complete) is based on ideas from CSP, Delta-Prolog and Luca Cardelli's rendezvous facility for Amber. In the proposed design, interprocess communication (IPC) is synchronous.

Backtracking is a tricky issue when designing an IPC scheme for Prolog. In this design, IPC like [nonbacktrackable] input/output, that, is, there is no way to undo a rendezvous.

Status

This version (5/10) has a first (incomplete) sketch of the external channel mechanism and a fairly detailed discussion of a timeout mechanism. I still need to complete the external channel specification and work out critical regions and error handling.

2. Concepts

(For more details see Luca's paper).

Process

An independent Prolog instruction stream and associated state. Processes are dynamically created by other processes and communicate by channels (q.v.). A process is characterized by its state (program counter, registers, stacks). In the design below, processes are manipulated (by manipulating their state) but processes are not first-class data objects, that is, there are no Prolog-accessible process objects.

In this document, the term "process" always means a Prolog process within a given executing Prolog image. Operating-system processes will be called "OS processes". An alternative convention very common in the Unix world is to call "lightweight process" to what we call "process" here and "process" to Unix processes.

Channel

Channels are dynamically-created objects that can be passed around as Prolog data objects. Channels transfer Prolog data objects between processes (see details below).

Channels come in two major types, *internal* channels that allow communication between Prolog processes and *external* channels that allow communication with the rest of the operating environment (operating system, I/O system, other

operating-system processes). External channels are further subdivided into *stream* channels, which are associated to a Prolog I/O stream and *programmed* channels, which are associated to some specialized communication code supplied by the user.

Request

A request corresponds to an attempt of a process to communicate. A request has an associated process, a resumption program counter for the process, and, when the request is for output, an output value. Requests are classified, according to the type of channel they belong to, into internal and external, and external requests into stream and programmed. An input and an output request on the same channel are said to be *complementary*. The operation of matching complementary requests with the appropriate data transfer is called *rendezvous*.

Pool

A pool is a set of requests, channels or processes with some common property, eg. waiting for input on a given channel. Luca (and I) use the term "pool" rather than "queue" to avoid the ordering connotations of the latter. Pools may be represented in the machine in any convenient way, eg. doubly-linked lists. In the descriptions below, I use several operations on pools, `insert(Elem,Pool)` that inserts an element into a pool, `remove(Pool)` that removes an arbitrary element from a nonempty pool and returns it, `remove(Elem,Pool)` that removes `Elem` from `Pool`, and also a constant `empty` for the empty pool.

3. Basic data structures

Channel

`channel.Type`	The channel type, one of `Internal`, `Stream` or `Programmed`.
`channel.State`	The channel state, one of `Open`, `Draining` or `Closed`.
`channel.InputPool`	Pool of input requests for this channel.
`channel.OutputPool`	Pool of output requests for this channel.
`channel.RendezVous`	The channel rendezvous function. Called to do the actual data transfer for each rendezvous operation on the channel.
`channel.Private`	A private value for external channels, passed in calls to the rendezvous function. It is irrelevant for internal channels; for stream channels, it is the associated stream; for programmed channels it is specified by the user.

Request

`request.Process`	The process that created this request.
`request.Channel`	Backpointer to this request's channel.
`request.Resumption`	Where to resume request.Process if the request is satisfied.
`request.Mutex`	Pool of mutually-exclusive requests (see the discussion of selection below).
`request.Value`	The output value for this request, or the address in which to store an input value. Output values are skeletal Prolog terms, that is, they are in the form of input terms.

Global Objects

`ActivePool`	Pool of all those channels with nonempty input or output pools.
`ReadyPool`	Pool of all the processes that are neither running nor waiting on a request.
`CurrentProc`	Which process is currently running.
`PreemptFlag`	This is used to request the emulator to deactivate (q.v.) the running process so that an external request can be satisfied.
`NullProc`	A dummy process that never runs but has all the process data structures. When `CurrentProc == NullProc`, the scheduler is intered. Conceptually, the scheduler is `NullProc` (maybe this can be used to clean up some of the dispatch code).

4. State Transitions

As Luca, I will describe the operational details of this design by specifying state transitions. Some of the state transitions correspond to specific emulator instructions, others to internal scheduler actions.

`Stop`

Description

Kills `CurrentProc`. Clearly, `CurrentProc` cannot be in any pool, so no pool tidying is required. However, this might be a good time to get rid of the process' state (eg. stack group). After a `stop`, the scheduler takes over.

Effect

```
CurrentProc = NullProc;
```

Start

Description

Takes a Prolog goal G and creates a new process with that initial goal. The process is placed in the ready pool. The current process continues executing.

Effect

```
insert(NewProcess(G), ReadyPool);
```

Language

```
fork(+Goal)
```

Implementation

I'm not sure how this should be implemented. It could be a new emulator instruction:

```
<put argument into Ai>
fork Ai
```

but it would probably be a rather complicated instruction. Alternatively, it could be implemented in pseudo-Prolog or C with some emulator assist.

Channel

Description

Creates a new channel.

Effect

```
chan = NewChannel();
chan.InputPool = chan.OutputPool = empty;
return chan;
```

Language

```
make_channel(-Channel)
stream_to_channel(+Stream,-Channel)
```

For `make_channel/1` the resulting channel is internal, for `stream_to_channel/2` it is external. The details of external streams are discussed below.

Implementation

To Prolog, channels should be "magic cookies" like streams and database references are now. It would be more efficient to have a reserved basic datatype for these beasties. I think that the original decision not to have a primitive datatype (and tag) for such magic creatures was a bad one. is it too late to change?

Close

Description

Closes down a channel. All subsequent attempts to output to the channel fail. When all the outstanding requests on the channel are satisfied, the channel is marked as closed and all requests on the channel fail.

Effect

```
channel->State = Draining;
```

Language

```
close_channel(+Channel)
```

Implementation

Channel-acess operators and the scheduler must check for channels being closed down. In particular, the scheduler must notice when the input and output pools are drained so that the channel structure may be freed and its descriptor invalidated (in the same way as old stream descriptors or database references are).

Select

Description

Request communication on a set of channels simultaneously. The first request to be satisfied is honored, and all the others are discarded. The requests to be created are given by a set R of tuples <C,D,V,N> where C is the channel, D the direction (input or output), V the value for output requests and N the continuation (address of next instruction to execute).

Effect

```
mutex = NewPool();
for (<C,D,V,N> in R) {
      req = NewRequest();
      req->Process = CurrentProc;
      req->Resumption = N;
      if (D == Input) {
```

```
                req->Value = V; */reference to unbound var */
                insert(req, C->InputPool);
        }
        else {
                req->Value = copy(V);
                insert(req, C->OutputPool);
        }
        insert(req, mutex);
}
CurrentProc = NullProc;
/* This activates the scheduler */
```

Language

$(G_1 \Rightarrow B_1 \mid \ldots \mid G_n \Rightarrow B_n)$

where each B_i is an arbitrary Prolog goal and each G_i is a communication goal of one of the two forms

`Channel?Term`	input request
`Channel!Term`	output request

At the time a communication goal is executed, the `Channel` argument must be instantiated to a valid channel. The `Term` argument can be any term. The decision on whether to honor a request is independent of the form of `Term`, that is, an input (output) on a channel is paired with an output (input) on the same channel and the corresponding branch of the selection is taken independently of whether the terms given as arguments to the input/output pair unify. Thus, a selection branch of the form

```
Channnel?Term => Goal
```

behaves exactly as

```
Channel?X => X = Term, Goal
```

where `X` is a new variable.

Implementation

We need four instructions

```
mutex Ai
read Ai, Aj, Ak, C
write Ai, Aj, Ak, C
wait
```

The `mutex` instruction establishes a mutual exclusion pool and returns a pointer to it on `Ai`. The `read` instruction establishes an input request for the [unbound] variable in `Aj`

on the channel in Ai with the mutual exclusion pool given in Ak and continuation at the instruction at offset C. The write instruction establishes an output request for the value in Aj on the channel in Ai, with the mutual exclusion given in Ak and continuation at instruction at offset C. The argument checking for these instructions, including the conversion of channel constants from Prolog terms to pointers to channel objects should be done by previous goals and instructions. Finally, the wait instruction suspends the execution of the current process and hands over to the scheduler.

Deactivate

Description

Moves the current process to the ready pool and hands over to the scheduler.

Effect

```
temp = CurrentProc;
CurrentProc = NullProc;
insert(temp, ReadyPool);
```

Language

```
suspend
```

The suspend/0 predicate can be used by any busy process to make sure that it does not lock out other activity. However, for external requests there is a suplementary mechanism that deactivates the current process at procedure calls or in the general unification main loop whenever PreemptFlag is true. This flag is set by the interrupt-driven part of the scheduler discussed below.

Implementation

A single instruction

```
deactivate
```

5. The Scheduler

Apart from the mess with external events discussed below, the scheduler is very simple. There are only two state transitions, activate that chooses a process from the ready pool to be the current process and rendezvous that matches an input request to an output request. If both activate and rendezvous are possible, the choice of which to do is arbitrary wrt. correctness, but it matters in terms of scheduling. I propose that rendezvous be always preferred to activate (eager communication), at least when PreemptFlag is true.

The scheduler always resets the preemption flag, so that a preemption request is not acted upon more than once.

Activate

Description

Removes a process from the ready pool and makes it the current process.

Effect

```
assert(ReadyPool != empty).
CurrentProc = remove(ReadyPool);
/* Continue from here executing CurrentProc */
```

RendezVous

Description

If the active pool is nonempty and contains a channel C with nonempty input and output pools, requests I and O are removed from C->InputPool and C->OutputPool respectively, the unbound variable pointed to by I.Value is bound to O->Value (or a heap copy of O->Value for nonground values), all requests in the mutex pools of I and O are removed from the respective pools and discarded. Channels without requests after this are removed from the active pool. I->Process is made current process and O->Process is placed in the ready pool. Note that I->Value does not need to be trailed because the compiler ensures that this is the first occurrence of that variable.

Effect

```
assert(ActivePool != empty);
assert(exists C in ActivePool
            with C->InputPool != empty
            and  C->OutputPool != empty);
for (C in ActivePool)
     if (C->InputPool != empty && C->OutputPool != empty) {
           remove(C, ActivePool);
           I = remove(C->InputPool);
           O = remove(C->OutputPool);
           (*C->RendezVous)(&val, O->Value, C->Private);
           if (atomic(val))
                 *I->Value = val;
           else
                 *I->Value = heapcopy(val, input->Process);
           PI = I->Process;
           PO = O->Process;
           PI->program_counter = I->Resumption;
           PO->program_counter = O->Resumption;
```

```
                for (E in I->Mutex) discard(E);
                for (E in O->Mutex) discard(E);
                for (D in ActivePool)
                      if (D->InputPool == empty
                       && D->OutputPool == empty)
                remove(D, ActivePool);
                if (PO != NullProc)
                      insert(PO, ReadyPool);
                CurrentProc = PI;
                break;
        }
        /* Resume CurrentProc */
        if (CurrentProc != NullProc)
              resume CurrentProc;
        else
              loop back to scheduler entry point;
```

For internal channels, the rendezvous function is simply defined by

```
(void) internal_rendezvous(input, output)
caddr_t *input, output;
{
      *input = output;
}
```

In practice the calling code for the rendezvous function need sto be somewhat more complicated to ensure datatype safety between Prolog and foreign code. The output value passed to the rendezvous function of a programmed channel must be converted to a foreign datatype and the value returned through the input pointer must be converted back to Prolog representation. Rendezvous functions for external channels are discussed in the next section.

The `heapcopy` builds a copy of its first argument (a skeletal term) on the heap of its second argument (a process). The `discard` function cancels a request, removing it from all the pools it belongs to. Furthermore, discarding the last remaining request on a channel also removes the channel from the active pool.

6. External Channels

External channels provide the communication between Prolog processes and the operating environment outside the running Prolog image, eg. the Unix operating system. For conceptual uniformity, we may think of external channels as connecting with a collection of permanently-running *external* processes supplied by the Prolog environment.

Typical operating systems do not use rendezvous as their primary means of communication between programs and the operating environment, so we need an *adaptor* layer between the redezvous mechanism and the external facilities. For simplicity, the adaptor layer should avoid dealing with rendezvous directly, but only

post appropriate requests into channel pools. Nevertheless, the code for the rendezvous operation will need to recognize certain special cases and deal with them appropriately.

In the remainder of this section, I will discuss how to implement external channels mainly for Berkeley Unix. At appropriate points I will give suggestions on how to adapt the design to System V.2. As for other operating systems (VAX/VMS, MVS), I don't know enough about them at this point to ensure that the design fits their peculiarities.

Types of External Channels

Stream Channels

A stream channel is created from a Prolog stream by a call to `stream_to_channel/2`. I don't see a good way of associating channels to programmed or biderectional streams, so `stream_to_channel` should only succeed for "straight" streams associated to operating-system input or output streams. A stream channel associated to an input (output) stream is an *input* (*output*) channel.

Input and output streams differ from internal streams only in the possible events the streams. An input request on an input channel may rendezvous with a complementary request by the I/O system, and an output request on an output channel may rendezvous with a complementary request posted by the I/O system. Nevertheless, it is also possible for requests on stream channels to rendezvous with internally generated complementary requests.

Programmed Channels

[[to be redone]]
Programmed channels are to channels what QP programmed streams are to streams. Their main purpose is to allow users to write foreign code that posts event requests, eg. to report Unix signals or window-system events.

The library function `QP_make_channel`, with declaration

```
typedef Bool (*BoolFn)();

Bool QP_make_channel(handle,chanfn, chan)
caddr_t handle; BoolFn chanfn; Int *chan;
```

attempts to build a channel with descriptor `*chan` with channel function `chanfn`. The channel descriptor is an integer that can be converted to a Prolog channel name with the predicate `channel_code/2` (analogous to `stream_code/2`). The channel function is called by the scheduler to check for available requests and to prepare for a scheduler wait.

Implementation

The basic implementation question is how to decouple process scheduling and rendezvous from operating-system vagaries. To this effect, I define the following scheduler predicates and pools

`Ready`	The ready pool is not emtpy.
`Active`	At least one rendezvous operation is possible.
`ExternalPool`	Pool of unsatisfied external requests
`Waiting`	The external pool is nonempty

With these, we can define several classes of scheduler states

stuck	`¬Ready & ¬Active`
blocked	stuck `& Waiting`
deadlocked	stuck `& ¬Waiting`
running	`¬`stuck `& ¬Waiting`
interruptible	`¬`stuck `& Waiting`

Blocked states

When blocked, the scheduler's only alternative is to satisfy some request in the wait pool. No Prolog-generated rendezvous can take us out of this state, only external intervention can. The available interventions are I/O operations on streams associated to stream channels and calls to the channel functions of programmed channels.

The Unix system call select is then used to wait for activity on external channels (both I/O and signals)

Deadlocked States

In a deadlocked state, there are no external requests whose satisfaction would cause the execution to proceed. This is an error, and Prolog should abort back to a well-defined start state with all requests canceled and all processes stopped.

Running States

In a running state, the scheduler only needs to take the appropriate (rendezvous or activate) action.

Interruptible States

An interruptible state should proceed like a running state with a rendezvous or activate action. However, there are pending external requests that may be satisfied. In order not to block out the satisfaction of these requests, is must be possible for external activity to set the preemption flag `PreemptFlag`. For requests on programmed channels, preemption is achieved by calling the `QP_post_event` function discussed above. For requests on stream channels, it is necessary to install an appropriate `SIGIO` handler on the Unix stream attached to the channel. This handler just raises the preemption flag to allow the scheduler to collect pending I/O requests.

7. Timeouts

Description

A process facility needs a timeout mechanism for various purposes, for instance to break out of selections after some period has elapsed. The timeout mechanism I propose here uses as much as possible of the machinery already developed. A timeout is implemented as a rendezvous on a special predefined *time* channel.

Effect

Timer request

```
wakeup = Now + interval;
if (wakeup  < NextWakeUp) {
      NextWakeUp = wakeup;
      set interval timer to signal at NextWakeUp;
}
enter an output request for TimeChannel with V = wakeup;
```

Wakeup call

```
for (O in TimeChannel->OutputPool)
if (O->Value <= Now) {
      req = NewRequest();
      req->Process = NullProc;
      req->Mutex = empty;
      req->Value = &dummy;
      insert(req, TimeChannel->InputPool);
}
NextWakeUp = Omega; /* the end of time */
for (O in TimeChannel->OutputPool)
      if (O->Value > Now && O->Value < NextWakeUp)
            NextWakeUp = O->Value;
if (NextWakeUp < Omega)
      set timer call for NextWakeUp;
PreemptFlag = true;
```

Language

```
(... time(+Interval) => ...)
```

Implementation

A timer request is simulated by an output request on the timer channel with the absolute time of the request as value. Unix timer wakeup calls are used to interrupt Prolog at the time of the closest request(s). When a wakeup occurs, matching input requests for all the timer channel output requests for earlier than the current time are created, a new wakeup call is established for the next timer request and preemption is initiated.