3C-9

Flong - A Portable Software System for the Inference Machine PIEEE

Martin Nilsson and Hidehiko Tanaka *
The University of Tokyo

Abstract

Flong is a portable software system for developing and executing GHC and Fleng programs running on parallel inference machines. The system is both somewhat like an interpreter and a very small operating system. It contains the essential mechanisms for managing parallel processes and shared resources, like peripheral devices and I/O streams. It also contains a basic set of tools such as an editor and a debugger. We will here briefly describe how goals are delegated from local processors to Flong, how device I/O is handled, and how exceptions are handled.

1 Introduction

Flong is a small runtime system for GHC and Fleng programs on parallel computers, such as PIEEE [KYNT 87]. It is portable in the sense that its requirements of the underlying machine are small, and that it can easily be modified to fit a new machine architecture. Fleng itself is also easily implementable, and can be used to boot the part of the system software which is written in Fleng.

Target systems for Flong are supposed to run GHC and Fleng programs on a large number of parallel processors. Most of these processors are supposedly very simple and can only perform the execution cycle and three basic system primitives [NT 86]. These system primitives are unification, one-argument metacall, and a primitive for computing simple arithmetic, comparison, and type checking.

To be useful, however, the system must for instance also be able to communicate with peripheral devices, modify the program database, and interact with the user. This work is done by different system components which we call monitors. A monitor's main task is to administrate system resources, like setting up a communication paths between processes and peripherals. Monitors have a function similar to that of front-end processors, or the kernel part of an operating system.

The system is made up of monitors, simple processors, and peripheral devices. All of these communicate through streams in the form of shared variables. (From the user's viewpoint, communication is always through streams, but the actual physical implementation may be different.) When a simple processor faces a situation it cannot handle, it passes it to a monitor, which decides on the proper action. In the following, we will assume that the reader is familiar with the concept of stream communication.

2 Monitors Handle Non-Local Goals

Every process is connected to a monitor by a stream. This stream is also distributed to the children of the process. Since this stream is always there, it is tacitly understood, and not explicitly written in program source code. As an example, the clause

```
a(X) :- b(X), c(X).
```

is in fact:

```
a(X,Sys) :- b(X,Sys1), c(X,Sys2),
            merge(Sys1,Sys2,Sys).
```

where `merge` merges the two streams `Sys1` and `Sys2` into `Sys`.

A predicate call which cannot be handled locally can easily be handed over to a monitor by passing it in the stream. E.g., if the predicate `c` above cannot be handled locally:

```
a(X,Sys) :- b(X,Sys1), Sys = [c(X)|Sys1].
```

An example of a system primitive which is handled in this way is the primitive for defining program clauses.

Two other types of situations, that cannot obviously be handled locally by a simple processor, are peripheral device I/O and exceptions. The handling of these constitute a central part of the Flong kernel, so we will describe them in more detail.

3 I/O Interface

All I/O is done through streams. Once a stream is set up, communication can easily be controlled locally by

*The Tanaka Lab., Dept. of Electr. Engineering, Univ. of Tokyo, Bunkyo-ku, Hongo 7-3-1, TOKYO 113

unification. But in order to set up the stream, a local processor needs help from a monitor. The primitive which does this is open, which takes three arguments:

```
open(Descriptor,Result,Stream)
```

`Descriptor` is a list `[Device|ModeList]`, where `Device` is the name of the peripheral device or file, and `ModeList` is an optional list of communication modes. `ModeList` can contain keywords such as `in` and `out`, For random access files and devices like terminals, `ModeList` can contain both `in` and `out`.

A stream contains items, where each item is a record `[Format|ExpressionList]` of two parts, a format specification, and a list of expressions.

```
Stream = [Item | NewStream]
```

For output streams, the expressions are output according to the format, and for input streams, an expression is read according to the format.

The format specification contains formatting keywords. Examples of formatting options for output streams are outputting an expression as an *Ascii character*, as a *term*, or as a *quoted term*. Examples of formatting options for input streams are inputting the *Ascii code* for a character, inputting a *token*, or a *term*.

A very important formatting option is `stream`, which says that the expression list is again a stream of items. This option allows a process to reserve a stream for a consecutive series of outputs.

Input and output is demand driven, so an expression is not output or input until the user defines its format. A file is closed by unifying its toplevel stream with the empty list. End of file can be detected during input if the ExpressionList becomes the empty list.

4 Exceptions

Errors, interrupts and traps are the possible exceptions in Flong. In Flong, they are very similar, and are all implemented in a similar way - they are just calls to Fleng predicates. This is possible in Fleng since processes are very loosely connected, and can be executed very independently of each other.

In other languages exceptions and common procedures are usually quite different and must interact in complicated ways to protect registers and stack contents.

All exception handlers are user-redefinable.

By *interrupts* we mean the asynchronous starting of processes, not directly under user control. Examples of interrupts are clock and timer interrupts, garbage-collection trigged interrupts, and keyboard interrupts.

Some interrupts, like timer interrupts, might need setting up, in order to be trigged later. This can be implemented by creating a stream to the timer device, sending the time interval on the stream to the timer, and then waiting for the timer to send a ready-message back.

Keyboard interrupts are executed when a user pushes a function or control key on his keyboard.

By *traps* we mean synchronous starting of processes by the user, or by some program called by the user. Examples of traps are simulated (user-generated) errors.

The difference between interrupts and traps is quite vague, especially in Fleng. Prolog, as a contrast, cannot allow traps and maintain a logical meaning of execution at the same time.

Errors are generated by errors inside system primitives, or by undefined predicate calls.

Error exceptions generate calls of the form

```
error(Call,Message,Expression)
```

where `Call` is the goal in which the error happened, `Message` is a symbol giving the error message, and `Expression` is the expression which caused the error.

This predicate is user-redefinable, to enable the user to customize action. Suitable action could be to ignore the error; print a message to the user; enter a debugger; or call a spelling-corrector.

5 Results

We have not implemented Flong on PIEEE yet, but we have implemented a pseudo-parallel version of Flong for Vax on top of Unix. This system is written in C, and contains a full implementation of Fleng, although some system software is not finished at the time of this writing. The system is available free for those who would like to experiment with it.

6 References

[KYNT 87] Koike,H.,Yamauchi,T.,Noda,H.,Tanaka,H.: *Parallel Inference Engine Experimental Environment PIEEE - Total System.* In Proc. Logic Progr. Conf. '86, Tokyo. June 1986.

[NT 86] Nilsson,M.: *FLENG Prolog - Turning supercomputers into Prolog machines.* In Proc. Logic Progr. Conf. '86, Tokyo. June 1986.