J. Functional Programming **8** (5): 537–542, September 1998. Printed in the United Kingdom

Book review

ML with Concurrency by Flemming Nielson (ed.), Springer-Verlag, 1997.

Overview

Standard ML has spawned a number of extensions for concurrent, parallel and distributed computing. *ML with Concurrency* (Nielseon, 1997) describes four of them and discusses their implementations and semantics. This book is a snapshot of current research, and is aimed at a researcher or graduate student specializing in concurrent language implementation or theory. It is divided into two parts:

- *Language design* of each of the four languages, discussing the extensions each has made to SML, the type systems, the semantics, and some applications.
- *Semantics* for concurrent functional languages, giving an analysis of the communication capabilities of a program, and a denotational semantics for the resulting judgments. These analyses are carried out for Concurrent ML, but the techniques should be applicable to the other languages.

These are discussed below, although the chapters are reordered to make comparisons easier.

Language design

The papers in this part are:

- *The Essence of Concurrent ML* by Panangaden and Reppy,
- *Concurrency in Poly/ML* by Matthews,
- *CCS Programming in an ML Framework: An Account of LCS* by Berthomieu, and
- *Facile – From Toy to Tool* by Thomsen, Leth and Kuo.

Each of these introduces the language, and discusses the design issues, and either the implementation, the formal semantics, or case studies.

Unfortunately, the main weakness of this part of the book is the lack of comparisons between the languages. Each of the chapters stands in its own right, but they have very different emphasis (for example the LCS chapter contains a large amount of formal type theory, while the Facile chapter is entirely free of mathematics). Combined with the lack of an overview or common examples this makes it quite difficult to compare the four languages.

I found it useful when reading these chapters to look at a simple example program, to see how each language would treat it. The example below is chaining two unbounded buffers together. In CCS (Milner, 1989), the buffer would be defined:

$$\begin{aligned}&\mathit{Buffer}\ [\,] \Leftarrow \mathit{inp}?x.\mathit{Buffer}\ [x]\\&\mathit{Buffer}\ (x :: xs) \Leftarrow (\mathit{inp}?y.\mathit{Buffer}\ (x :: xs@[y])) + (\mathit{outp}!x.\mathit{Buffer}\ xs)\end{aligned}$$

and the chaining operation:

$$P \gg Q \Leftarrow (P[\mathit{mid}/\mathit{outp}] \mid Q[\mathit{mid}/\mathit{inp}]) \setminus \mathit{mid}$$

Two buffers can then be chained as $\mathit{Buffer}\ [\,] \gg \mathit{Buffer}\ [\,]$.

Poly/ML

Abstract Hardware's Poly/ML compiler contains a library for concurrent threads with synchronous communication over typed channels. Threads can be spawned for concurrent computation using *fork*, and external choice between threads is provided using the binary *choice* operator.

For example, an unbounded buffer can be coded:

$$\begin{array}{l}
\textbf{fun}\ \mathit{Buffer}\ []\ (\mathit{inp},\mathit{outp}) = \mathit{Buffer}\ [\mathit{receive}\ \mathit{inp}]\ (\mathit{inp},\mathit{outp}) \\
\quad \mid \mathit{Buffer}\ (x :: \mathit{xs})\ (\mathit{inp},\mathit{outp}) = \mathit{choice}\ (\\
\qquad \textbf{fn}\ () \Rightarrow \mathit{Buffer}\ (x :: \mathit{xs}@[\mathit{receive}\ \mathit{inp}])\ (\mathit{inp},\mathit{outp}), \\
\qquad \textbf{fn}\ () \Rightarrow \mathit{Buffer}\ (\mathit{send}\ (x,\mathit{outp});\ \mathit{xs})\ (\mathit{inp},\mathit{outp}) \\
\quad);
\end{array}$$

with type (for this review, I shall brush the issue of imperative type variables and SML 97 value polymorphism firmly under the carpet):

$$\mathit{Buffer}\ : \alpha\ \mathit{list} \rightarrow (\alpha\ \mathit{channel} * \alpha\ \mathit{channel}) \rightarrow \mathit{unit}$$

Note that in the second clause, the environment is provided with the choice between a *inp* action or an *outp* action. This uses the *choice* function which executes two thunks until one of them performs a communication synchronization, at which point the other is killed:

$$\mathit{choice}\ : (\mathit{unit} \rightarrow \mathit{unit}) * (\mathit{unit} \rightarrow \mathit{unit}) \rightarrow \mathit{unit}$$

Chaining can be defined:

$$\begin{array}{l}
\textbf{fun}\ f \gg g\ (\mathit{inp},\mathit{outp}) = \textbf{let val}\ \mathit{mid} = \mathit{channel}\ () \\
\quad \textbf{in}\ \mathit{fork}\ (\textbf{fn}\ () \Rightarrow f\ (\mathit{inp},\mathit{mid})); g\ (\mathit{mid},\mathit{outp})\ \textbf{end};
\end{array}$$

Apart from some syntax with thunking, this style of programming is very familiar to anyone used to process algebra. However, there is an implementation cost in allowing choices between arbitrary thunks, since it means we can have choices between distributed threads, for example:

$$\mathit{choice}((\textbf{fn}\ () \Rightarrow (\mathit{Buffer}\ [] \gg \mathit{Buffer}\ [])\ (a,b)), (\textbf{fn}\ () \Rightarrow \mathit{Buffer}\ []\ (a,b)))$$

In process algebra terms, we can write general choice, such as $(P \mid Q) + R$, not just guarded choice such as $a.P + b.Q$. Because of general choice, the synchronization algorithm in Poly/ML is quite complex, and is discussed in this chapter in some detail.

This chapter gives an overview of Poly/ML from a programmer's perspective, and discusses the single-heap and multi-heap implementations. There is very little formal semantics.

Concurrent ML

Concurrent ML is a library for SML of New Jersey, which adds concurrent threads in a manner at first sight similar to Poly/ML, for example chaining two processes is:

$$\begin{array}{l}
\textbf{fun}\ f \gg g\ (\mathit{inp},\mathit{outp}) = \textbf{let val}\ \mathit{mid} = \mathit{channel}\ () \\
\quad \textbf{in}\ \mathit{spawn}\ (\textbf{fn}\ () \Rightarrow f\ (\mathit{inp},\mathit{mid})); g\ (\mathit{mid},\mathit{outp})\ \textbf{end};
\end{array}$$

However, there is a subtle difference in the *Buffer* implementation which reveals an important difference in the implementation and design philosophy of CML:

$$\begin{array}{l}
\textbf{fun}\ \mathit{Buffer}\ []\ (\mathit{inp},\mathit{outp}) = \mathit{Buffer}\ [\mathit{recv}\ \mathit{inp}]\ (\mathit{inp},\mathit{outp}) \\
\quad \mid \mathit{Buffer}\ (x :: \mathit{xs})\ (\mathit{inp},\mathit{outp}) = \mathit{sync}\ (\mathit{choose}\ [\\
\qquad \mathit{wrap}\ (\mathit{recvEvt}\ \mathit{inp}, \textbf{fn}\ y \Rightarrow \mathit{Buffer}\ (x :: \mathit{xs}@[y])\ (\mathit{inp},\mathit{outp})), \\
\qquad \mathit{wrap}\ (\mathit{sendEvt}\ (\mathit{outp},x), \textbf{fn}\ () \Rightarrow \mathit{Buffer}\ \mathit{xs}\ (\mathit{inp},\mathit{outp})) \\
\quad]);
\end{array}$$

The difference is caused by the fact that CML only allows guarded choice, not arbitrary choice, and this restriction is enforced by the types of the CML primitives:

$$\begin{array}{l}
\mathit{recvEvt} : \alpha\ \mathit{chan} \rightarrow \alpha\ \mathit{event} \\
\mathit{sendEvt} : \alpha\ \mathit{chan} * \alpha \rightarrow \mathit{unit}\ \mathit{event} \\
\mathit{choose} : \alpha\ \mathit{event}\ \mathit{list} \rightarrow \alpha\ \mathit{event} \\
\mathit{sync} : \alpha\ \mathit{event} \rightarrow \alpha \\
\mathit{wrap} : \alpha\ \mathit{event} * (\alpha \rightarrow \beta) \rightarrow \beta\ \mathit{event}
\end{array}$$

Since *choose* is limited to the *event* type, this means we can only build guarded choices. In particular, we cannot build the equivalent of the CCS term $(P \mid Q) + R$ since $P \mid Q$ cannot be expressed as a value of type *event*.

CML builds on this by allowing computation both before and after the atomic action in an event, thus allowing three-phase protocols to be defined, with a pre-commitment phase, an atomic commitment, and a post-commitment phase. Different protocols can then be put in a common choice.

This chapter gives an overview of the design of the CML language, including a static and dynamic semantics, and a discussion of how CML (almost) fits Moggi's (1991) monadic framework for computation.

Facile

Facile is, on the surface, quite different from Poly/ML and CML since it extends the syntax of SML. It has a *guard* type constructor similar to CML's *event*, so the code for the buffer is almost identical:

```
proc Buffer [] (inp, outp) = activate Buffer [recv inp] (inp, outp)
  | Buffer (x :: xs) (inp, outp) = activate alternative [
      wrapguard (recvguard inp, fn y => Buffer (x :: xs@[y]) (inp, outp)),
      wrapguard (sendguard (outp, x), fn () => Buffer xs (inp, outp))
  ];
```

The alert reader will probably wonder what the new keywords **proc** and **activate** are for. Facile uses a two-level syntax for concurrent programming: SML expressions and Facile behaviours. Facile also introduces a new type *scr* with an isomorphism between behaviours and expressions of type *scr* given by:

- if *e* is an expression of type *scr* then **activate** *e* is a behaviour, and
- if *b* is a behaviour then **script** *e* is an expression of type *scr*.

Programming in Facile seems to involve jumping between these two syntaxes. This is sometimes confusing (as in the above *Buffer* example) but does allow process algebra syntax for concurrency:

```
proc f ≫ g (inp, outp) = let val mid = channel ()
  in activate f (inp, mid) || activate g (mid, outp) end;
```

Once the syntactic hurdle has been overcome, however, programming in the small in Facile is quite similar to programming in CML. However, where the emphasis in CML is on programming in the small, for example in writing communications protocols, Facile emphasizes programming in the large, and in particular distributed client-server systems.

Facile's alternative to object-based languages such as CORBA is to extend the SML module system, and to allow virtual machines to negotiate the signature of a module which is downloaded and linked at run-time. For example, say we wished to make simple queries of a database:

```
signature DBQuerySig = sig val dbQuery : string → string end;
```

We would like to download a module from a remote server satisfying this signature:

$$\textbf{val}\ \mathit{server} = \mathit{initial\_library}\ \texttt{"facile.icl.co.uk"};$$
$$\textbf{structure}\ \mathit{DBQuery} = \textbf{demanding}\ \mathit{server}\ \textbf{with}\ \mathit{DBQuerySig};$$

The above Facile code downloads an appropriate module. Moreover, SML is extended to allow structure declarations at run-time, so code can be downloaded when required, not just at link-time.

This chapter contains some example code (although in order to construct the above examples I had to read the on-line Facile manuals) and descriptions of the distributed environment, together with a discussion of implementation trade-offs and case studies. There is no discussion of static or dynamic semantics.

LCS

LCS also extends SML to include behaviours, with a syntax very similar to that of process calculus. In particular, note that in the buffer process we do *not* specify the input and output channels as parameters:

$$\begin{aligned}&\textbf{fun}\ \mathit{Buffer}\ [\,] = \mathit{inp}?x \Rightarrow \mathit{Buffer}\ [x]\\ &\quad \mid \mathit{Buffer}\ (x :: xs) = (\mathit{inp}?y \Rightarrow \mathit{Buffer}\ (x :: xs@[y])) \vee (\mathit{outp}!x \Rightarrow \mathit{Buffer}\ xs);\end{aligned}$$

The channels of the process are not given as parameters, but instead are represented in the type system, using a type and effect system similar to that discussed by Nielson and Nielson in their chapter. The types are recorded in a style similar to that of extensible records using Wand's row type variables. For example, the buffer is typed:

$$\mathit{Buffer} : \alpha\ \mathit{list} \rightarrow \{\mathit{inp} : \alpha, \mathit{outp} : \alpha\}_b$$

This type records the names of the channels used in the buffer, together with their types. The b extension is a polymorphic variable indicating that the function does not put any other constraints on channel types.

LCS allows visible channels to be renamed and restricted in a style similar to CCS:

$$\textbf{fun}\ p \gg q = (p\{\mathit{mid}/\mathit{outp}\} \wedge q\{\mathit{mid}/\mathit{inp}\})\{/\mathit{mid}\};$$

Note that channels are *not* first class values. For example, there is no way to code up a router which uses a lookup table to decide which channel to forward a packet on; in CML this would be:

$$\begin{aligned}&\textbf{fun}\ \mathit{Router}\ (\mathit{tbl}, \mathit{inp}) = \textbf{let val}\ (\mathit{dest}, \mathit{payload}) = \mathit{recv}\ \mathit{inp}\\ &\quad \textbf{in}\ \mathit{send}\ (\mathit{lookup}\ (\mathit{tbl}, \mathit{dest}), \mathit{payload}); \mathit{Router}\ (\mathit{tbl}, \mathit{inp})\ \textbf{end};\end{aligned}$$

This chapter presents the dynamic and static semantics of LCS, and briefly discusses the concurrent and parallel implementations. A distributed implementation is planned, based on the same architecture as Poly/ML.

Semantics

The papers in the semantics part are:

- *A Semantic Theory for ML Higher-Order Concurrency Primitives* by Debbabi and Bolignano, and
- *Communication Analysis for Concurrent ML* by Nielson and Nielson.

The first paper builds upon techniques defined in the second paper, so I shall consider them in reverse order.

Communication analysis

By comparing the type system of CML with that of LCS we can see that there are two ways to give a type system for a concurrent language. Either the communication possibility of a process are hidden (as in CML) or they are made explicit (as in LCS). In the latter case, it is possible to infer information about the dynamic behaviour of terms from their static semantics.

In a *type and effect* system for concurrent languages, the communication possibilities of a process are recorded in the type system. For example the CML type system has:

$$a : \alpha\ chan, b : \alpha\ chan \vdash Buffer\ [\,]\ (a, b) : unit$$

whereas a type and effect system would record the communication possibilities as:

$$a : \alpha\ chan_{l_1}, b : \alpha\ chan_{l_2} \vdash Buffer\ [\,]\ (a, b) : unit\ \ \&\ l_1?\alpha + l_2!\alpha$$

Here we have tagged the channels with *regions* l_1 and l_2, and we have recorded that the buffer can input on a channel from region l_1 and output on a channel from region l_2.

Nielson and Nielson extend this even further by recording the trace semantics of a process in its type for example:

$$a : \alpha\ chan_{l_1}, b : \alpha\ chan_{l_2} \vdash Buffer\ [\,]\ (a, b) : unit\ \ \&\ \textsc{rec}X\ .\ l_1?\alpha; X + l_2!\alpha; X$$

Compare this with a process which first inputs on a:

$$\begin{aligned}a : \alpha\ chan_{l_1}, b : \alpha\ chan_{l_2} \vdash\ & Buffer\ [recv\ \ a]\ (a, b) : unit\\ &\&\ l_1?\alpha; (\textsc{rec}X\ .\ l_1?\alpha; X + l_2!\alpha; X)\end{aligned}$$

In fact, the type system is even stronger than trace equivalence, since it records the channel creation and concurrent behaviour of the processes. For example chaining two buffers together creates a new channel and spawns off a buffer before running a buffer:

$$\begin{aligned}&a : \alpha\ chan_{l_1}, b : \alpha\ chan_{l_2} \vdash (Buffer\ [\,] \gg Buffer\ [\,])\ (a, b) : unit\\ &\quad \&\ \textsc{chan}_{l_3}\ \alpha; \textsc{fork}_{\pi}(\textsc{rec}X\ .\ l_1?\alpha; X + l_3!\alpha; X); \textsc{rec}X\ .\ l_3?\alpha; X + l_2!\alpha; X\end{aligned}$$

This type system is very strong (for example, it distinguishes between bisimilar terms such as *Buffer* [] and *Buffer* [] $\gg$ *Buffer* []) and the usual definition of subject reduction fails to hold. Instead, the paper shows that as a process evolves, its type also evolves, using a labelled transition system for types.

This paper investigates this type and behaviour inference system in some depth, and concludes with examples of how it might be used in practice for analyzing communication topologies, or for processor scheduling.

Denotational semantics

Debbabi and Bolignano's chapter discusses a denotational semantics for CML based on acceptance trees. This is an active area of research, since providing a denotational semantics for languages which include a unique name generator like CML's *channel* () is non-trivial.

The search for suitable models led Pitts and Stark (1993) to look at functor categories as an appropriate setting for such languages. This approach has had some success, in particular Hennessy (1996) has shown how to lift acceptance trees into this world, and provide a fully abstract model for the π-calculus. Unfortunately, Debbabi and Bolignano do not refer to any of this body of research.

Their semantics is an example of Moggi's (1991) monadic semantics for the computational call-by-value λ-calculus, although the monad T_σ (their notation is $D\_Routine\ (\_, \sigma)$) is parameterized on the side-effects σ of the process. They present Moggi's monadic semantics, and prove that T_σ contains enough structure to define communication, choice and concurrency, which provides them with their model. T_σ also appears to model channel generation, but the relevant clause is missing from the denotational semantics, so it is difficult to tell.

There are interesting ideas in this paper, such as using a monad T_σ parameterized on side-effects, and basing a denotational semantics on a type and effect system, but they are obscured by the presentation.

Conclusion

The language design papers in this book are of a high quality, and will be useful references. Although they do not contain large amounts of new work, they are good summaries of current research, and will be of interest to anyone implementing or providing semantics for concurrent functional languages.

The semantics papers are much more specialized, although still may be worth reading for experts in type and effect systems or denotational semantics for higher-order concurrent languages.

References

Hennessy, M. (1996) A fully abstract denotational semantics for the pi-calculus. Technical report 96:04, University of Sussex.

Milner, R. (1989) *Communication and Concurrency*. Prentice-Hall.

Moggi, E. (1991) Notions of computation and monad. *Infor. and Comput.* **93** 55–92.

Nielson, F. (ed.) (1997) *ML with Concurrency*, Monographs in Computer Science, Springer-Verlag.

Pitts, A. M. and Stark, I. D. B. (1993) Observable properties of higher order functions that dynamically create local names, or: What's new? *Proc. MFCS 93: Lecture Notes in Computer Science 711*, pp. 122–141. Springer-Verlag.

Alan Jeffrey