Parallel Implementation of Functional Languages Using Small Processes

John Glauert[*]
School of Information Systems
University of East Anglia
Norwich NR4 7TJ, UK
jrwg@sys.uea.ac.uk

Parallel Implementation Workshop, Aachen, September 1992

Abstract

We report work in progress on the implementation of languages that integrate concurrent and functional programming styles.

A translation scheme is presented for mapping such languages into process networks using a simple notation based on communication between named processes. The translation of functional code is sequential, but parallelism arises from use of concurrency constructs in the source language.

Early implementation experiments have used graph rewriting techniques, although work on a more direct implementation is in progress.

1 Introduction and Background

This paper reports work done at ECRC in close collaboration with Bent Thomsen and Lone Leth. The work has focussed on techniques for implementing the Facile language [GPM89] which enhances the λ–calculus with primitives for process spawning and channel-based communication in the style of CCS [Mil80].

Recent work by Milner using the π–calculus [MPW89] and its polyadic form [Mil91] shows that a λ–expression may be translated to a process network in the polyadic π–calculus which simulates the λ–expression [Mil90]. Similar work was done by Leth [Leth91].

The approach taken by the Facile group at ECRC has been to develop translations of both the functional and concurrent features of Facile into networks of small processes. In such networks there is a uniform representation of all features of the language. The ultimate goal is to exploit the inherent concurrency of the process networks in an implementation on a parallel machine.

Our first approach used process notations that are close to the Polyadic π–Calculus [Mil91] in which communication is channel-based. This work is reported in [GLT91] and [Gla92]. In such models the channels can be thought of as independent entities which act as brokers, arranging communications between processes. Similar ideas appear in [Wis92].

However, it turned out that in most cases a channel used in the translations was closely associated with a particular process which was the only process to receive messages from the channel. Further, the process would not receive messages from any other channel.

[*]Work done on sabbatical at ECRC, Munich

This led us to investigate a model of communicating agents using process-based, or point-to-point, communication. This is in contrast to the channel-based communication of CCS, and the π–calculus. The model can be seen as a very minimal version of Paragon [ABK92],

with a very direct correspondence to the graph-rewriting framework in which Paragon has been presented. Later we present a mapping from the new notation to the graph-rewriting language Dactl [GKS91].

The process-communication model results in a very low-level translation of Facile features in which potential implementation data-structures become visible. For example, Facile channels can be represented as objects which are manipulated in exactly the same way as in the Chemical Abstract Machine [BeBo90] proposed for Facile in [LT92].

2 A Source Language

For the purpose of investigation we have used a very small language in which we can represent all the features of core Facile along with facilities for handling constants and operators as in ML.

2.1 Source Language Syntax

The language is the λ–calculus with constants and pairing. The trivial value and pairing operator could have been considered as normal constants.

$$e \quad ::= \quad x \mid \lambda x.\, e \mid e\, e \mid c \mid () \mid (e, e)$$

c stands for a constant and x for a variable.

The features of Facile are encoded in our source language as special constants in very much the same way as in the sequential and distributed implementations based on SML [Kuo92, KC91, Kna91]. The functionality is hidden in the definition of the constants *channel*, *send*, *receive*, and *spawn*.

The function *channel* takes a trivial value and returns a new channel value. Every invocation of *channel* returns a distinct channel value.

The function *send* takes a pair containing a channel and a value to be sent on the channel. A trivial value is returned when the output action has succeeded. The function has the effect of the SML/Facile code ($\mathbf{fn}\,(x, y)\ =>\ x!y$)

receive takes a channel as argument and returns a value received as the result of an input action. It has the effect ($\mathbf{fn}\, x\ =>\ x?$).

The essence of Facile behaviour expressions, the scripts of processes, is provided by using delay closures, functions of type $() \rightarrow ()$. *spawn* takes such a function and returns a trivial value having created a process which runs in parallel. The spawned process executes the behaviour expression by applying its representation function to the trivial value, terminating if the application terminates.

Although the original Facile language provides more programming flexibility, it is sufficient for our implementation experiments to concentrate on the four primitives described above.

2.2 Source Language Semantics

Although the language treated in this paper is slightly different from the original Facile language, it is clear that there is a direct correspondence between expressions in this language and in the original. Hence we may use the semantics found in [GPM89]. We will assume that programs are always closed terms.

It is worth recalling the reduction relation for the λ–calculus sublanguage of Facile. The language is given by:

$$\begin{array}{rcl} e & ::= & x \mid \lambda x.\, e \mid e\, e \\ v & ::= & \lambda x.\, e \end{array}$$

v is the class of weak head-normal forms for closed terms.

$$APPL: \ \frac{e \to e''}{e\ e' \to e''\ e'} \qquad APPR_v: \ \frac{e' \to e''}{v\ e' \to v\ e''} \qquad BETA_v: \ (\lambda x.e)\ v \to e\,[v/x]$$

We will call the reduction relation defined as the least relation satisfying these rules the *sequential* strategy. To reduce an application, first the operator will be reduced to weak head-normal form then the operand, and only when both are abstractions will the limited beta-reduction rule be applied. No reduction is performed under λs so only a weak head-normal form will be produced.

The sequential strategy is very close to the *call-by-value* strategy given by the rules:

$$APPL: \ \frac{e \to e''}{e\ e' \to e''\ e'} \qquad APPR: \ \frac{e' \to e''}{e\ e' \to e\ e''} \qquad BETA_v: \ (\lambda x.e)\ v \to e\,[v/x]$$

This is a non-deterministic strategy since operator and operand may be reduced in either order before beta-reduction.

For the pure λ–calculus both of these strategies converge for the same set of terms, but for Facile as in SML, it is convenient to use the sequential strategy in order to be able to reason about any side effect resulting from evaluation. The lack of implicit concurrency in the expression part of Facile contrasts with the explicit concurrency provided by the process part.

It would be possible to exploit safe implicit concurrency, where the evaluation order of operator and operand does not affect the observable behaviour of an expression. The approach we have explored is to decorate the hidden apply operators in expressions in the manner of [Bur84]. Sequential application, denoted by $@_s$, is reduced according to $APPR_v$ while call-by-value application, $@_v$, uses $APPR$. The other rules apply to both forms. The combination gives a reduction relation for expressions with a mixture of application styles.

To complete the discussion we note that our earlier work [GLT91] proposed the combination of the *normal-order* strategy of the Lazy-λ–Calculus [Abr88] and an *eager* strategy. The first has just $APPL$ and the general $BETA$ rule:

$$APPL: \ \frac{e \to e''}{e\ e' \to e''\ e'} \qquad BETA: \ (\lambda x.e)\ e' \to e\,[e'/x]$$

The eager strategy adds $APPR$:

$$APPL: \ \frac{e \to e''}{e\ e' \to e''\ e'} \qquad APPR: \ \frac{e' \to e''}{e\ e' \to e\ e''} \qquad BETA: \ (\lambda x.e)\ e' \to e\,[e'/x]$$

The normal-order strategy is sequential and, as is well known, is strongly normalising for terms with a weak head-normal form. The relation specified by the eager strategy contains the same normal forms, but it is a larger relation, allowing speculative evaluation of operands. There may be infinite reduction sequences even for terms with normal forms. We may combine these strategies using $@_n$ and $@_e$ for normal-order and eager application operators respectively; $APPR$ only applies when the application operator is $@_e$.

Many approaches to parallel implementation of functional programming languages can be seen as combinations of these strategies, with the addition of concurrent reduction which corresponds to use of consistent multi-step stategies. Call-by-value reduction mixed with normal-order reduction may change convergence properties and does not support pipelining. Mixing normal-order and eager evaluation retains convergence properties as long as the choice of reduction steps is fair. In either case if the reductions other than the normal-order step are *"needed"*, convergence will be maintained. This corresponds to only varying the reduction strategy in the presence of strict arguments.

3 A Simple Process Notation

In this section we introduce a very simple and highly restrictive process model based on communication with named processes. The model has adequate expressive power for our purpose of translating a language such as Facile, but might not seem attractive for general purpose use.

3.1 Informal Description of the Process Model

The process model manipulates a world of terms forming a many-sorted algebraic structure:

- Terms are built up from primitive data values, including a domain of process names, and constructor symbols with fixed arities.
- A process has a name and is associated with a term which holds all the process state.
- Messages are terms which are directed to named processes.

Process names are almost pointers to terms, hence the representation of a process network is only a step away from a directed-graph representation.

The following examples are taken from a more complete example later and illustrate the creation of a process with name l whose name is known to the new process k. The initial state of l is a term involving subterms d and i not shown here. A message containing a numeric value is sent to the process k.

$$l : C(d,i),\ \ k : F(l),\ \ k\,!\,Unit(3) \tag{1}$$

The behaviour of processes is determined by pattern-directed rules which describe what happens when a message is received by a process:

- Patterns are left-linear open terms, hence they may contain variable names, but such names may not be repeated.
- Matching of rules is based on two patterns. The sets of variable names in the two patterns are disjoint. Neither pattern may be a solitary variable, hence every pattern has at least a root symbol.

- The first pattern of a rule matches the state of the process, the other matches the incoming message. Pattern variables are bound to corresponding subterms during matching.
- Every message sent to a process must match exactly one rule.
- Actions are performed when a rule is matched, substituting values of bound variables in the normal way. Rules contain no free variables.

The following example shows the rule which will be used by processes k above:

$$F(l)\ ?\ Unit(m) \quad \rightarrow \quad (\,p:\ E(l,m)\,|\,p\,!\,Unit(4)\,)$$

The condition that a process must handle every incoming message avoids the implication that an implementation might need to store messages and retry them later if the process state changes. An alternative semantics would be for unmatched messages to be discarded. Also, the requirement that exactly one rule is matched avoids the need to specify the semantics of overlapping rules. It is clear that pattern matching could be translated into nested matching operations revealing any potential non-determinism.

When a message is received:

- A process may create new terms.
- A process may create new processes and has access to their names.
- A process may send messages to processes it can name.
- A process may copy the name of a process it can name. Hence it can send such names to other processes.
- A process may change its own state. The process name subsequently refers to the new state.

Note that a process may not inspect the state of another process; it may only send it messages or pass on its name. Messages must be handled completely by their target process before another message is considered.

3.2 Syntax and Informal Semantics of the Process Model

In the proposed syntax, lower-case identifiers are bound variable names. Other identifiers are constructor symbols. The most general form for a rule is a pair of patterns, followed by actions to be performed if a process whose state matches the first pattern receives a message matching the second pattern:

$$ProcState\ ?\ MsgPattern \quad \rightarrow \quad (\,Action\,|\cdots|\,Action\,)\ :=\ NewState$$

The actions are zero or more process creations, zero or more message creations, and an optional revised state for the process. The syntax $p\,!\,Msg$ denotes sending a message to a named process, while $p : InitState$ creates a process with an initial state, as illustrated above.

Processes might be better called actors or objects since they only act when messages are sent to them. When a message arrives at a process exactly one rule should match. We have required that the message is handled completely by a process before another message is considered. This is very conservative: if a process involves no state change there should be no need for such serialisation; if a process does involve a state change then once all new terms and processes referenced in the new state have been created, and the state has been changed, then other messages could be handled.

Notionally, at least, the actions all occur simultaneously. However, in practice there are some dependencies:

- Any new processes and terms are built.
- The process state is changed if required.
- Sending of any messages is initiated.

Clearly it would be meaningless for any messages sent as a result of invoking a rule to be processed before processes and terms they reference have been created.

3.3 An Example of the Process Model

As an example we will address the hardest case which occurs in the translation of the source language given earlier. This concerns communication which is the only language feature requiring synchronisation.

The chosen representation of a Facile channel is exactly like that in the Facile Chemical Abstract Machine of [LT92]. The state of a channel holds a queue of pending output offers and a queue of pending input requests. At least one of the queues will be empty.

A channel receives input and output requests and either satisfies the request or queues it. Requests contain continuations in the form of process names. Hence a receiver sends an $In(r)$ message with the identity of the process to handle the received message. Senders send $Out(v, s)$ where v is the value to be output and s a process to receive acknowledgement that the communication has succeeded.

Messages are sent to both sender and receiver when a communication is closed.

$$\begin{array}{rcl}
Chan(iq, NOQ)\ ?\ In(r) & := & Chan(IQ(iq, r), NOQ) \\
Chan(NIQ, oq)\ ?\ Out(v, s) & := & Chan(NIQ, OQ(oq, v, s)) \\
Chan(iq, OQ(oq, v, s))\ ?\ In(r) & \rightarrow & (\, r\,!\,v \,|\, s\,!\,Triv\,)\ :=\ Chan(iq, oq) \\
Chan(IQ(iq, r), oq)\ ?\ Out(v, s) & \rightarrow & (\, r\,!\,v \,|\, s\,!\,Triv\,)\ :=\ Chan(iq, oq)
\end{array}$$

When a process representing a new channel is created, the initial state has both queues empty:

$$ch : Chan(NIQ, NOQ)$$

It is synchronisation of concurrent computations which requires state changes. None of the rules generated for the λ–calculus part of Facile needs to involve a change in process state.

4 A Translation Scheme for Facile

The translation is based on the syntax of a term. The translation of a syntactic form takes a parameter which will be the name of the process to which a result is to be sent. In the translation, we will make use of a function $fv()$ which yields a list of the free variables in a λ–expression.

4.1 Translation of the Source Language

The translation of a syntactic construct creates some actions which will form part of the body of a rule, and may also introduce new constructor symbols and associated rules:

$$
\begin{array}{rcl}
[\![x]\!]_u & = & u\,!\,Unit(x) \\
[\![c]\!]_u & = & u\,!\,Unit(c) \\
[\![()]\!]_u & = & u\,!\,Triv \\
[\![\lambda x.\, e]\!]_u & = & u\,!\,Unit(y) \,|\, y:\ Y(fv(\lambda x.\, e)) \\
 & & \text{where} \quad Y(fv(\lambda x.\, e))\ ?\ Doub(b,x)\ \cdot\ (\,[\![e]\!]_b\,) \\
[\![e\, e']\!]_u & = & [\![e]\!]_a \,|\, a:\ A(u, fv(e')) \\
 & & \text{where} \quad A(u, fv(e'))\ ?\ Unit(y)\ \cdot\ (\,b:\ B(u,y) \,|\, [\![e']\!]_b\,) \\
 & & \text{and} \quad B(u,y)\ ?\ Unit(z)\ \cdot\ (\,y\,!\,Doub(u,z)\,) \\
[\![(e, e')]\!]_u & = & [\![e]\!]_a \,|\, a:\ A(u, fv(e')) \\
 & & \text{where} \quad A(u, fv(e'))\ ?\ Unit(y)\ \cdot\ (\,b:\ B(u,y) \,|\, [\![e']\!]_b\,) \\
 & & \text{and} \quad B(u,y)\ ?\ Unit(z)\ \cdot\ (\,u\,!\,Doub(y,z)\,)
\end{array}
$$

All the constructor names introduced are distinct. All the process names are distinct and are different from all the variables used in the source λ–expression, with the exception of x in the translation of an abstraction which stands for itself.

The translation of the various constant functions for arithmetic and the operators of Facile is as follows:

$$
\begin{array}{rcl}
[\![succ]\!]_u & = & u\,!\,Unit(f) \,|\, f:\ Succ \\
 & & \text{where} \quad Succ\ ?\ Doub(b,x)\ \cdot\ (\,b\,!\,Unit(x+1)\,) \\
[\![mul]\!]_u & = & u\,!\,Unit(f) \,|\, f:\ Mul \\
 & & \text{where} \quad Mul\ ?\ Doub(b, Pair(x,y))\ \cdot\ (\,b\,!\,Unit(x*y)\,) \\
[\![channel]\!]_u & = & u\,!\,Unit(f) \,|\, f:\ Channel \\
 & & \text{where} \quad Channel\ ?\ Doub(b, Triv)\ \cdot\ (\,b\,!\,Unit(c) \,|\, c:\ Chan(NIQ, NOQ)\,) \\
[\![send]\!]_u & = & u\,!\,Unit(f) \,|\, f:\ Send \\
 & & \text{where} \quad Send\ ?\ Doub(b, Pair(c,v))\ \cdot\ (\,c\,!\,Out(v,s)\,) \\
[\![receive]\!]_u & = & u\,!\,Unit(f) \,|\, f:\ Receive \\
 & & \text{where} \quad Receive\ ?\ Doub(b,c)\ \cdot\ (\,c\,!\,In(b)\,) \\
[\![spawn]\!]_u & = & u\,!\,Unit(f) \,|\, f:\ Spawn \\
 & & \text{where} \quad Spawn\ ?\ Doub(b,p)\ \cdot\ (\,b\,!\,Unit(Triv) \,|\, p\,!\,Doub(Triv,s) \,|\, s:\ Sink\,) \\
 & & \text{and} \quad Sink\ ?\ Unit(Triv)\ \cdot\ (\ )
\end{array}
$$

The rules for $Chan$ were given earlier.

It should be noted that the types are not really correct here; a typed source language is required, as before. However, the constructors used correctly reflect the arities of the terms.

4.2 An Example of a Translated Expression

The following is the translation of the expression $(Mul(3,4),())$ under the new scheme:

$$
\begin{array}{rcl}
Start\ ?\ Triv & \to & (\,d:\ B(a) \,|\, c:\ D(d) \,|\, c\,!\,Unit(b) \,|\, b:\ Mul \,|\, a:\ Print\,) \\
A(a,e)\ ?\ Unit(f) & \to & (\,g:\ PairB \,|\, g\,!\,Trip(a,e,f)\,)
\end{array}
$$

$$\begin{array}{rcl}
B(a) \mathbin{?} Unit(e) & \rightarrow & (\, h : A(a,e) \,|\, h \mathbin{!} Unit(Triv) \,) \\
C(d,i) \mathbin{?} Unit(j) & \rightarrow & (\, i \mathbin{!} Doub(d,j) \,) \\
D(d) \mathbin{?} Unit(i) & \rightarrow & (\, l : C(d,i) \,|\, k : F(l) \,|\, k \mathbin{!} Unit(3) \,) \\
E(l,m) \mathbin{?} Unit(n) & \rightarrow & (\, o : PairB \,|\, o \mathbin{!} Trip(l,m,n) \,) \\
F(l) \mathbin{?} Unit(m) & \rightarrow & (\, p : E(l,m) \,|\, p \mathbin{!} Unit(4) \,) \\
Mul \mathbin{?} Doub(b, Pair(x,y)) & \rightarrow & (\, b \mathbin{!} Unit(x * y) \,) \\
PairB \mathbin{?} Trip(u,x,y) & \rightarrow & (\, u \mathbin{!} Unit(Pair(x,y)) \,)
\end{array}$$

We illustrate the first few steps of the elaboration of a program which starts with an initial process with state $Start$ being sent the message $Triv$:

$$\begin{array}{ll}
 & s : Start,\ s \mathbin{!} Triv \qquad (2) \\
\rightarrow & a : Print,\ d : B(a),\ b : Mul,\ c : D(d),\ c \mathbin{!} Unit(b) \qquad (3) \\
\rightarrow & a : Print,\ d : B(a),\ b : Mul,\ l : C(d,b),\ k : F(l),\ k \mathbin{!} Unit(3) \qquad (4) \\
\rightarrow & a : Print,\ d : B(a),\ b : Mul,\ l : C(d,b),\ p : E(l,3),\ p \mathbin{!} Unit(4) \qquad (5) \\
\rightarrow & a : Print,\ d : B(a),\ b : Mul,\ l : C(d,b),\ o : PairB,\ o \mathbin{!} Trip(l,3,4) \qquad (6) \\
\rightarrow & a : Print,\ d : B(a),\ b : Mul,\ l : C(d,b),\ l \mathbin{!} Unit(Pair(3,4)) \qquad (7) \\
\rightarrow & a : Print,\ d : B(a),\ b : Mul,\ b \mathbin{!} Doub(d, Pair(x,y)) \qquad (8) \\
\rightarrow & a : Print,\ d : B(a),\ d \mathbin{!} Unit(12) \qquad (9)
\end{array}$$

The values 3 and 4 are gathered together to form a pair by step 6. This pair is sent to the Mul process which sends on the product to the process given by the first argument to $Doub$.

5 Implementation of the Process Model by Graph Rewriting

An implementation of the process model, and hence of Facile programs translated into the model, can be made using graph rewriting techniques.

The model is based on very similar principles to Paragon [ABK92]. Indeed we could have used a subset of the syntax of Paragon, although our model is much simpler overall.

It is also possible to map the process model to a graph rewriting system in the extended graph rewriting language Dactl [GKS91]. An system has been written in SML which translates source programs into an internal form of the process notation, and also generates an executable Dactl program. The examples in this paper are taken straight from the output of the system.

Creation of a process maps to creation of a named graph term. Sending a message to a process involves creating an active graph term with the symbol `Call` and subterms representing the destination process and the message to be sent. Apart from a small number of utility rules, all the rules are for the symbol `Call`.

The rules for communication, using the symbol `Chan`, are the only ones involving a change of state. These rules, in the utility module $AProcs$, overwrite the contents of the destination process state.

Below is the module for the Dactl translation of the earlier example, followed by the utility module used by all programs, with apologies to those unfamiliar with the language.

```
MODULE Facile;
IMPORTS AProcs;
```

```
SYMBOL OVERWRITABLE  Start_; A; B; C; D; E; F;

RULE
  INITIAL -> *Call_[Start_ Triv_];
  Call_[ h_:Start_ Triv_ ] ->
             d:B[a], c:D[d], *Call_[c Unit_[b]], b:Mul_, a:Print_;

  Call_[ h_:A[a e] Unit_[f] ] -> g:PairB_, *Call_[g Trip_[a e f]];
  Call_[ h_:B[a] Unit_[e] ] -> h:A[a e], *Call_[h Unit_[Triv_]];
  Call_[ h_:C[d i] Unit_[j] ] -> *Call_[i Doub_[d j]];
  Call_[ h_:D[d] Unit_[i] ] -> l:C[d i], k:F[l], *Call_[k Unit_[3]];
  Call_[ h_:E[l m] Unit_[n] ] -> o:PairB_, *Call_[o Trip_[l m n]];
  Call_[ h_:F[l] Unit_[m] ] -> p:E[l m], *Call_[p Unit_[4]];
ENDMODULE Facile;


MODULE AProcs;
IMPORTS OS_Core; Arithmetic;

SYMBOL CREATABLE PUBLIC CREATABLE Triv_; Unit_; Doub_; Trip_; Print_;
SYMBOL CREATABLE PUBLIC CREATABLE Channel_; Send_; Receive_; Spawn_;
SYMBOL CREATABLE PUBLIC CREATABLE Pair_; Succ_; Add_; Sub_; Mul_; Div_;
SYMBOL CREATABLE PUBLIC CREATABLE PairB_; AddB_; SubB_; MulB_; DivB_;
PATTERN PUBLIC SuccB_ = Succ_;

SYMBOL REWRITABLE PUBLIC REWRITABLE Call_;
SYMBOL REWRITABLE Print; Sink;
SYMBOL OVERWRITABLE Chan;
SYMBOL CREATABLE NIQ; IQ; NOQ; OQ; In; Out; Terminate;

RULE
  Call_[ PairB_ Trip_[u x y] ] -> *Call_[ u Unit_[Pair_[x y]] ];

  Call_[ c: Chan[iq:NIQ oq] Out[v s] ] -> c := Chan[iq OQ[oq v s]];
  Call_[ c: Chan[IQ[iq r] oq] Out[v s] ] ->
             *Call_[r Unit_[v]], *Call_[s Unit_[Triv_]], c := Chan[ iq oq
];
  Call_[ c: Chan[iq oq:NOQ] In[r] ] -> c := Chan[ IQ[iq r] oq ];
  Call_[ c: Chan[iq OQ[oq v s]] In[r] ] ->
             *Call_[r Unit_[v]], *Call_[s Unit_[Triv_]], c := Chan[ iq oq
];

  Call_[ Channel_ Doub_[u Triv_] ] -> *Call_[ u Unit_[Chan[NIQ NOQ]] ];
  Call_[ Send_ Doub_[s Pair_[c v]] ] -> *Call_[ c Out[v s] ];
  Call_[ Receive_ Doub_[r c] ] -> *Call_[c In[r]];
  Call_[ Spawn_ Doub_[u p] ] ->
             *Call_[u Unit_[Triv_]], s: Sink, *Call_[p Doub_[Triv_ s]];
  Call_[ Sink Unit_[Triv_] ] -> *Terminate;

  Call_[ Succ_ Doub_[u n] ] -> #Call_[ u ^#Unit_[^*IAdd[n 1]] ];
```

```
  Call_[ Add_ Doub_[u Pair_[x y]] ] -> #Call_[ u ^#Unit_[^*IAdd[x y]] ];
  Call_[ Sub_ Doub_[u Pair_[x y]] ] -> #Call_[ u ^#Unit_[^*ISub[x y]] ];
  Call_[ Mul_ Doub_[u Pair_[x y]] ] -> #Call_[ u ^#Unit_[^*IMul[x y]] ];
  Call_[ Div_ Doub_[u Pair_[x y]] ] -> #Call_[ u ^#Unit_[^*IDiv[x y]] ];

  Call_[ AddB_ Trip_[u x y] ] -> #Call_[ u ^#Unit_[^*IAdd[x y]] ];
  Call_[ SubB_ Trip_[u x y] ] -> #Call_[ u ^#Unit_[^*ISub[x y]] ];
  Call_[ MulB_ Trip_[u x y] ] -> #Call_[ u ^#Unit_[^*IMul[x y]] ];
  Call_[ DivB_ Trip_[u x y] ] -> #Call_[ u ^#Unit_[^*IDiv[x y]] ];

  Call_[ Print_ Unit_[v] ] -> *SEQ[PrintF["Result: "] Print[v]];
  Print[Pair_[l r]] =>
             *SEQ[PrintF["("] Print[l] PrintF[","] Print[r] PrintF[")"]];
  Print[Triv_] => *PrintF["()"];
  Print[n:INT] => *PrintF["%d" n];
ENDMODULE AProcs;
```

6 Direct Implementation of the Process Model

The next stage of the work aims to produce an efficient sequential implementation of a functional language with the features of Facile using this model. Later work will address parallel implementation as well.

6.1 Optimisation

Many of the rules in the automatically generated code create a process and immediately send it a message. Since most of the rules generated only match a single pattern each, it is possible to expand them out and build rules with larger numbers of actions.

Most of the code is inherently sequential, so it will reduce to something corresponding to normal expression evaluation using a continuation-passing style.

6.2 Processing Messages

The rules are like very primitive methods in an object-oriented model. Hence the processing of a message is like a procedure call. The pattern of process state term and the message determines what code is executed. All the state of a process is in its term representation. There is no environment to worry about.

If the actions of a rule generate no messages, a thread of control terminates. See the first two rules for $Chan$ and the rule $Sink$.

If the actions of a rule generate one message, a sequential thread continues. The new message can be prepared and control may jump to the code indicated by the symbol of the target process.

If the actions of a rule generate multiple messages, additional threads of control are required. This applies to the last two rules for $Chan$ and code for *spawn*. A stack-based implementation on a single processor would have the effect of remembering the work to be done, and returning to it when the current thread completes (possibly because an attempted communication blocks). The advantage is that no record of the thread needs to be placed on some form of run queue. With a parallel machine, the problem is that this potential work is not available for evaluation on other processors.

There is no need to use a stack at all if messages representing new threads are created as heap objects and linked into a rudimentary run queue. The number of actions generating multiple messages is expected to be a small proportion of all messages, so the cost of creating these very simple thread descriptors may be acceptable even on a sequential machine.

7 Conclusions

The ideas presented here represent work which is still very much in progress, although early indications are that for straightforward functional code it will be possible to produce an implementation comparable with existing high-quality compilers.

The benefit of the technique should be that there will be little penalty for using the process facilities of a language like Facile. By comparison, with existing implementations of the language, process spawning is only beneficial if large grains of computation are available.

The implementation model used appears to match well with recent techniques for programming multiprocessors [ECGS92]. Hence there is the promise of an effective parallel implementation in future.

References

[ABK92] P. Anderson, D. Bolton & P. Kelly: *Paragon Specifications: Structure, Analysis and Implementation*, Proc PARLE'92. LNCS 605. June 1992. (1992)

[Abr88] S. Abramsky: *The Lazy Lambda Calculus*, Chapter 4 in D. Turner (ed.), Research Topics in Functional Programming, pp. 65-116, Addison Wesley, 1988.

[BeBo90] G. Berry & G. Boudol: *The Chemical Abstract Machine*, Proc. POPL 90, p 81-94. (1990)

[Bur84] F.W. Burton: *Annotations to Control Parallelism and Reduction Order in the Distributed Evaluation of Functional Programs*, TOPLAS, Vol 6, No 2, p159-174. (1984)

[ECGS92] T. v Eicken, D.E. Culler, S.C. Goldstein & K.E. Schauser, *Active Messages: a Mechanism for Integrated Communication and Computation*, International Symposium on Computer Architecture '92, ACM. (1992).

[GKS91] J.R.W. Glauert, J.R. Kennaway, & M.R. Sleep: *Dactl: An Experimental Graph Rewriting Language*, Proc. 4th International Workshop on Graph Grammars, Bremen, 1990. Springer LNCS 532. (1991)

[Gla92] J.R.W. Glauert: *Asynchronous Mobile Processes and Graph Rewriting*, Proc PARLE'92. LNCS 605. June 1992. (1992)

[GLT91] J.R.W. Glauert, L. Leth & B. Thomsen: *A New Translation of Functions as Processes*, SemaGraph '91 Symposium, December 1991. (1991)

[GPM89] A. Giacalone, P. Mishra, & S. Prasad: *Facile: A Symmetric Integration of Concurrent and Functional Programming*, IJPP, Vol 18, No 2, p121-160. (1989)

[KC91] A. Kramer & F. Cosquer: *Distributing Facile*, MAGIC Note 12, ECRC. October 1991. (1991)

[Kna91] F. Knabe: *A distributed protocol for the generalized select command*, MAGIC Note 11, ECRC. July 1991. (1991)

[Kuo92] Kuo, T.-M.: *Magic Facile version 0.3*, MAGIC Note 22, ECRC. March 1992. (1992)

[Leth91] L. Leth: *Functional Programs as Reconfigurable Networks of Communicating Processes*, Ph.D. Thesis, ICSTM. (1991)

[LT92] L. Leth & B. Thomsen: *Some Facile Chemistry*, ECRC Technical Report ECRC-92-14. May 1992. (1992)

[Mil80] R. Milner: *A Calculus of Communicating Systems*, Springer LNCS 92. (1980)

[Mil90] R. Milner: *Functions as Processes*, Automata, Languages, and Programming. Springer LNCS 443. (1990) Also: Technical Report INRIA Sophia Antipolis, June 1989.

[Mil91] R. Milner: *The Polyadic π−Calculus: A Tutorial*, Technical Report ECS-LFCS-91-180, Edinburgh University, October 1991. (1991)

[MPW89] R. Milner, J. Parrow, & D. Walker: *A Calculus of Mobile Processes*, Parts I and II, Technical Report ECS-LFCS-89-85, Edinburgh University, June 1989. (1989)

[Wis92] M. Wise: *Message Brokers*, Proc PARLE'92. LNCS 605. June 1992. (1992)