A New Process Model for Functions

J.R.W. Glauert, L. Leth and B. Thomsen

In: *Term Graph Rewriting: Theory and Practice*, Wiley, 1993

1 Introduction

A number of studies have been made of the relationship between process notations and functional programming. Early work by Kennaway and Sleep used the LNET model [Ken82], while Thomsen used higher-order processes [Tho89, Tho90]. Milner [Mil90], and Leth [Let91], show how an arbitrary λ–expression may be converted to a network of processes whose behaviour simulates reduction of the original λ–expression.

This work has been primarily motivated by a desire to explore the theoretical relationship between the formalisms, rather than to provide an efficient implementation of the λ-Calculus. Such work is of particular interest when considering implementation of a language such as Facile [Gia89] which aims to integrate functional and process styles. By encoding the functional components of a program as a network of processes, the complete language may be converted to processes for semantic purposes at least.

The studies based on extensions to CCS adopt a synchronous model of communication. Honda and Tokoro [Hon91] show that asynchronous communication can be used to the same effect. The work presented here is able to take advantage of such asynchronous models.

This paper describes a new translation of the λ-Calculus into a process notation, supporting a mixture of evaluation strategies using techniques similar to [Bur84]. Milner and Leth consider only the pure λ-Calculus while our new translation handles constant data values and their operations.

Programs in the process language may be translated very naturally into a generalised graph rewriting system (GRS). A translation to the practical GRS language Dactl is presented. The style of execution of such a GRS is rather different from the traditional term-based translation of a functional language.

2 The Lazy- and Eager- λ-Calculus

In this section we shall review some aspects of the Lazy-λ-Calculus [Abr88] and introduce the Eager-λ-Calculus. The syntax of both is slightly unconventional as we use an explicit left-associative operator, @, for application.

The operator will be decorated to indicate which calculus is being considered.

$$e \quad ::= \quad x \mid \lambda x.\, e \mid e \mathbin{@} e$$

where x is taken from a set of variable names.

2.1 The Lazy-λ-Calculus

Terms in the *Lazy-λ-Calculus* may be reduced according to the relation $\rightarrow_L$ defined as follows:

DEFINITION **2.1** *Let $\rightarrow_L$, lazy reduction, be the smallest relation satisfying:*

$$APPL: \quad \frac{e \;\rightarrow_L\; e''}{e \mathbin{@} e' \;\rightarrow_L\; e'' \mathbin{@} e'} \qquad\qquad BETA: \quad (\lambda x.e) \mathbin{@} e' \;\rightarrow_L\; e\,[e'/x]$$

The Lazy-λ-Calculus is inherently sequential. Reduction always occurs at the head of an application sequence, i.e. let $M \equiv M_0 @ M_1 \ldots\; M_n$ $(n > 0)$ where M_0 is not an application, then the only reduction possible is when $n \geq 1$ and $M_0 \equiv \lambda x.\, N$. In this case we have $M_0 @ M_1 @ M_2 \ldots\; M_n \;\rightarrow_L\; N\{M_1/x\} @ M_2 \ldots M_n$ since we will generally consider that expressions to reduce are closed. There will always be a redex on the left spine unless the whole expression is an abstraction.

Note that the Lazy-λ-Calculus does not reduce the bodies of abstractions so it will only reduce expressions to weak head-normal form. This is not a problem when considering functional programming where functional normal forms are not usually of interest.

2.2 The Eager-λ-Calculus

Since we are interested in parallel evaluation of functional programs by translating them into a formalism for concurrent processes, we do not have great hopes if we solely base our work on the Lazy-λ-Calculus. A more eager evaluation strategy is obtained by adding an extra reduction rule $APPR$:

DEFINITION **2.2** *Let $\rightarrow_E$, eager reduction, be the smallest relation satisfying:*

$$APPL: \quad \frac{e \;\rightarrow_E\; e''}{e \mathbin{@} e' \;\rightarrow_E\; e'' \mathbin{@} e'} \qquad\qquad APPR: \quad \frac{e' \;\rightarrow_E\; e''}{e \mathbin{@} e' \;\rightarrow_E\; e \mathbin{@} e''}$$

$$BETA: \quad (\lambda x.e) \mathbin{@} e' \;\rightarrow_E\; e\,[e'/x]$$

This allows reduction in both operator and operand in an application. We shall call this calculus the *Eager-λ-Calculus*. $\rightarrow_L$ is clearly contained in $\rightarrow_E$.

The Eager-λ-Calculus introduces some non-determinism to the reduction rules since for any application with redexes in the operand we may either reduce the operand or perform the reduction in $\rightarrow_L$. Since the Lazy-λ-Calculus is normalising, the Eager-λ-Calculus may converge whenever the lazy calculus converges. However, if we continually choose redexes from $\rightarrow_E$

which are not needed, then a term with a weak head-normal form may not converge.

We should remark that the Eager-λ-Calculus is not the same as the *Call-by-Value-λ-Calculus*. Call-by-value reduction uses a restricted $BETA$ rule in which the operand must be a value, v, in the form $\lambda x.\, e$:

$$BETA: \ (\lambda x.e) \mathbin{@} v \ \rightarrow_V \ e\,[v/x]$$

This means that reduction of operator and operand may proceed non-deterministically, but reduction of a redex may only occur when both operator and operand have converged. Clearly there are terms for which the Call-by-value-λ-Calculus diverges when the Lazy-λ-Calculus converges and the Eager-λ-Calculus may converge.

2.3 A Mixed Calculus

The translations presented in this paper support a mixed language in which the Lazy-λ-Calculus and Eager-λ-Calculus live together in one calculus. In this case we annotate the application by $@_L$ or $@_E$ in $M@_LN$ or $M@_EN$ to indicate which evaluation strategy is to be used. The rules $APPL$ and $BETA$ apply to both forms of application while $APPR$ applies only to $@_E$. The effect is similar to [Bur84]. The intuition is that we may reduce any instance of the $BETA$ rule which can be reached by a path through the operator position of $@_L$ terms or either argument of $@_E$ terms.

Languages such a Concurrent Clean provide facilities which allow for limited eager evaluation and might well be given a semantics in the same way as the mixed calculus. Judicious use of the $@_E$ operator in positions where the operand is needed or strongly terminating will mean that the convergence properties of the program are the same as for the Lazy-λ-Calculus.

3 A Process Notation

We present a process notation into which we will translate programs written in the mixed calculus described in the previous section. The notation has features corresponding to those of ECCS [Eng86], the π-Calculus [MPW89] and LCCS [Let91]. All may be regarded as extensions to CCS allowing the communication of link names and hence allowing dynamic process networks to be generated.

The significant extension to these earlier notations is that we permit, as an atomic event, the communication of terms representing arbitrary tuples of values, although nothing more complicated than pairs is required for our translations below. This is very nearly equivalent to the Polyadic-π-Calculus [Mil91] although we would allow nested terms. We allow a received term to be matched to a pattern of names, or bound to a single name. Further work is required on a sort discipline for the calculus to ensure that names bound to structured values are never used as link names.

We are also very interested in the sub-calculus in which the process following an output action is always the inactive process. This is in essence the language of [Hon91].

3.1 Syntax

The forms of agent, or process, allowed are a subset of those of LCCS with the addition of defined agents from the π-Calculus. In the syntax below, P and Q are agents, A an agent identifier, x a link, and y a value or link name:

Output $x!y.P$
Send value y on link x and continue with behaviour P.

Tupled Output $x!(q,r).P$
Send pair of values q and r on link x and continue with behaviour P.

Input $x?y.P$
Receive a value on link x and bind the value to y in subsequent behaviour P.

Tupled Input $x?(q,r).P$
Receive a pair of values on link x. Bind the first value to q and the second to r in subsequent behaviour P.

Restriction $P\backslash x$
x is the name of a link which may only be used within P. For many purposes this may be seen as the declaration of the link x.

Parallel Composition $(P|Q)$
P and Q continue concurrently and may interact via shared links. Binary composition is illustrated but an arbitrary number of agents may be composed including zero which gives inaction.

Agent Definition $P : A(x_1,\ldots,x_n) = Q$
A is an agent identifier of arity n which may be used in P. $x_1,\ldots,x_n$ may be free names in Q. Q may contain agent identifiers, including A, and free names from P.

Defined Agent $A(y_1,\ldots,y_n)$
A corresponding agent definition of the form $A(x_1,\ldots,x_n) = Q$ must be in scope. The defined agent behaves like $Q\{y_1/x_1,\ldots,y_n/x_n\}$.

Hence parallel composition and repetition are supported, but not summation (choice).

? and \ are name binding constructs yielding the obvious notion of free and bound names. We let $fn(P)$ denote the set of free names in P.

The form $\mathbf{rec}X.P$ is allowed as syntactic sugar for $X : X = P$.

$$\text{Output} \quad x!y.P \xrightarrow{x!y} P$$

$$\text{Input} \quad x?y.P \xrightarrow{x?z} P\{z/y\}$$

$$\text{Res} \quad \frac{P \xrightarrow{\alpha} P'}{P\backslash x \xrightarrow{\alpha} P'\backslash x} \quad , x \neq c(\alpha)$$

$$\text{Open} \quad \frac{P \xrightarrow{y!x} P'}{P\backslash x \xrightarrow{y\backslash z} P'\{z/x\}} \quad , z \notin fn(P\backslash x)$$

$$\text{Par} \quad \frac{P \xrightarrow{\alpha} P'}{P \mid Q \xrightarrow{\alpha} P' \mid Q} \quad , bn(\alpha) \cup fn(Q) = \emptyset$$

$$\text{Com} \quad \frac{P \xrightarrow{x?y} P' \;\; Q \xrightarrow{x!y} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'}$$

$$\text{Close} \quad \frac{P \xrightarrow{x?y} P' \;\; Q \xrightarrow{x\backslash y} Q'}{P \mid Q \xrightarrow{\tau} (P' \mid Q')\backslash y} \quad , y \notin fn(P)$$

$$\text{Agents} \quad \frac{P\{y_1/x_1, \ldots, y_n/x_n\} \xrightarrow{\alpha} P'}{A(y_1, \ldots, y_n) \xrightarrow{\alpha} P'} \quad , A(x_1, \ldots, x_n) = P$$

Figure 1: Operational semantics for Process Notation

3.2 Operational Semantics

The operational semantics of the process notation is given in terms of a labelled transition system. There are four kinds of actions, ranged over by α: input actions $x?y$, output actions $x!z$, output of restricted name $x\backslash y$ and internal actions τ. We let $c(\alpha)$ denote the communication channel of an action, e.g. x above. ? and $\backslash$ are name binders and $bn(\alpha)$ denotes the bound name of an action, e.g. y above.

The transition relation is defined as the smallest relation satisfying the axiom and rules in Figure 1.

4 Translation of Lazy and Eager λ-Calculus

Milner [Mil90] provides encodings of the Lazy-λ-Calculus and the Call-by-value-λ-Calculus in the π-Calculus. Only closed terms are considered. The encodings simulate particular reduction strategies.

We present a new scheme for translating λ–expressions to process networks. The language translated provides two forms of application operator and also supports constants denoted by k in the syntax. If all applications are $@_L$ then a lazy translation results, although it is not quite the same as that of Milner. The use of $@_E$ throughout yields a translation of the Eager-λ-Calculus.

4.1 The New Translation Scheme

$$
\begin{array}{rcl}
[\![x]\!]_u & = & \mathbf{rec}\, X.\, u?v\,.\,(\, x!v.() \mid X\,) \\
[\![k]\!]_u & = & \mathbf{rec}\, X.\, u?v\,.\,(\, v!k.() \mid X\,) \\
[\![\lambda x.\, M]\!]_u & = & \mathbf{rec}\, X.\, u?(x,q)\,.\,(\, [\![M]\!]_q \mid X\,) \\
[\![M \mathbin{@_L} N]\!]_u & = & (\, [\![M]\!]_f \mid f!(t,u).() \mid \mathbf{rec}\, X\,.\, t?v.(\, a!v.() \mid [\![N]\!]_a \mid X\,) \setminus a\,) \setminus f \setminus t \\
[\![M \mathbin{@_E} N]\!]_u & = & (\, [\![M]\!]_f \mid f!(a,u).() \mid [\![N]\!]_a\,) \setminus f \setminus a
\end{array}
$$

The pure λ-Calculus is not of great practical use. Constants, such as boolean and integer data values, and operations on them must be added to make a practical functional language. Data values are represented by a set of names disjoint from those used for links. Functions such as the successor function will be represented by global names. A process network to implement these functions will exist in parallel with the program to be evaluated. The network for *succ* would be as follows:

$$\mathbf{rec}\, X.\, u?(x,q)\,.\,(\, X \mid (\, x!t.() \mid t?n\,.\,q?r\,.\,r!(n+1).()\,) \setminus t\,)$$

An occurrence of the name *succ* will be translated like a variable.

It is assumed that programs will reduce to data values. To retrieve the value from the translation of such a λ–expression it is necessary to send a channel to it and receive back the value on the channel. To extract the value from a program, P, we would produce a network of the form:

$$(\, [\![P]\!]_u \mid \mathit{Read}(\, u\,)\,) \setminus u \;:\; \mathit{Read}(\, w\,) \;=\; (\, w!c.() \mid c?v\,.\,\mathit{Print}(\, v\,)\,) \setminus c$$

and *Print* will display the final result.

4.2 Properties of the New Scheme

The intuition behind the translation is that the translation of each syntactic construct is a process network with a special characteristic link, or *handle*, through which the network will communicate. This link is denoted by u in the translations.

If the process represents a numeric constant, the value received on u will be a link on which the number is to be returned. If the process encodes an

abstraction, u will be sent a pair of links. The first link of the pair is the handle on the argument, while the second link is to be the handle on the process which results when the argument is substituted in the body of the abstraction.

The encoding of an application links up an abstraction with an argument. Considering $@_E$ first, we see that neither the encoding of the operator nor operand are guarded. The process networks concerned will evolve independently. When the operator reaches weak head-normal form it will be waiting to receive input on its handle. The apply glue code in the translation sends the resulting abstraction a pair informing it of the operand and the channel which is to be the handle for the resulting network.

The encoding of $@_L$ guards the operand so that only when the operator actually requests the value of the operand will it be evaluated. The extra glue code acts as a buffer, passing on the request from the operator to the operand network.

A variable simply acts as a buffer, relaying information sent to it to the handle for the operand provided when the abstraction binding the variable was applied. The information sent will depend on the type of the result: if it is a constant then a single return link name will be sent; if it has a function type then a pair will be sent.

There are simplifications gained by communicating pairs of values. If only simple names may be communicated, extra temporary links are needed to prevent interleaving of messages from different agents able to the handle u, as in Milner's call-by-value scheme. The Polyadic-π-Calculus also overcomes these problems, though we must be aware that the sort of apparently similar links will depend on the type of the λ-expressions being translated.

All output actions in a translated network are followed by inaction. As a consequence no process is dependent on synchronised communication, and an asynchronous communication model would suffice. This can significantly reduce the complexity of a low-level implementation of such a process notation as will be shown later. [Hon91] shows that sequentialisation is possible using just asynchronous communication, though our use of tupled communication allows their scheme to be simplified.

4.3 Elimination of Some Recursion

It will be seen that a number of uses of the **rec** construct appear. These are needed to allow for multiple occurrences of the bound variable in the body of abstractions. When such abstractions are applied, the argument may be evaluated more than once with different arguments. These uses of **rec** are not always required. Indeed none would be required for the linear λ-Calculus.

To eliminate some of the superfluous recursion we may use a more sophisticated translation scheme based on the observation that terms in head position do not need recursion. In other words, when a variable or a λ-abstraction occurs as an operator in a λ-expression there is no need for multiple copies, whereas occurrence as an operand needs the possibility of

producing various copies of the variable or the λ-abstraction.

The function application cannot be analysed similarly, so recursion has to be provided in all cases. This gives us eight rules in the translation scheme; two for variables (without and with recursion), two for constants, two for λ-abstraction, and one for each of the applications. The translations involved with recursion are annotated with $*$:

$$\begin{aligned}
[\![x]\!]_u &= u?v\,.\,x!v.() \\
[\![x]\!]^*_u &= \mathbf{rec}X.u?v.(x!v \mid X) \\
[\![k]\!]_u &= u?v\,.\,v!k.() \\
[\![k]\!]^*_u &= \mathbf{rec}\,X.\,u?v\,.\,(\,v!k.() \mid X\,) \\
[\![\lambda x.\,M]\!]_u &= u?(x,q)\,.\,[\![M]\!]_q \\
[\![\lambda x.\,M]\!]^*_u &= \mathbf{rec}\,X.\,u?(x,q)\,.\,(\,[\![M]\!]_q \mid X\,) \\
[\![M \mathbin{@_E} N]\!]_u &= [\![M \mathbin{@_E} N]\!]^*_u \\
&= (\,[\![M]\!]_f \mid f!(a,u).() \mid [\![N]\!]^*_a\,) \setminus f \setminus a \\
[\![M \mathbin{@_L} N]\!]_u &= [\![M \mathbin{@_L} N]\!]^*_u \\
&= (\,[\![M]\!]_f \mid f!(t,u).() \mid \mathbf{rec}\,X\,.\,t?v.(\,a!v.() \mid [\![N]\!]_a \mid X\,) \setminus a\,) \setminus f \setminus t
\end{aligned}$$

Note that the translation of $M \mathbin{@_L} N$ only involves one level of recursion using **rec**. The recursion in $M \mathbin{@_E} N$ comes indirectly through the use of $[\![N]\!]^*$.

5 Process Notation and Graph Rewriting

Process networks such as those described in this paper may be easily translated into generalised graph rewriting systems. The practical graph rewriting language Dactl [Gla91b] is used as the target for our translation.

5.1 A Standard From for Process Networks

In this section we define how to express a process network in a standard form which makes heavy use of defined agents. This form can then be implemented very easily as a GRS.

If A stands for an agent identifier then the forms we will allow are:

$$P ::= x!y.A \;\big|\; x?y.A \;\big|\; (A \mid \ldots \mid A) \;\big|\; P\backslash x \;\big|\; P : A = P$$

Also, we restrict agent definitions to the outermost level, so they may contain no free variables. It should be clear that any process expression can be converted to this form by inserting new agent identifiers and adding appropriate definitions. The transformation is in four stages. We will use forms of $\lambda x.x$ for illustration:

$$\mathbf{rec}\,X.\,u?(x,q)\,.\,(\,X \mid \mathbf{rec}\,Y.\,q?v\,.\,(\,Y \mid x!v.()\,)\,)$$

The first stage is to expand the syntactic sugar for **rec** which was defined such that $\mathbf{rec}X.P \equiv X : X = P$. This gives:

$$X: \quad X = u?(x,q)\,.\,(\,X \,|\, Y : Y = q?v\,.\,(\,Y \,|\, x!v.()\,)\,)$$

The second stage is to add extra agent definitions to satisfy the reduced syntax using the rule that $P \equiv A : A = P$ where A is a new agent identifier:

$$\begin{aligned} X: \quad & X = u?(x,q)\,.\,Z : Z = (\,X \,|\, Y : Y = q?v\,.\,W : \\ & W = (\,Y \,|\, Put : Put = x!v.\,Nil : Nil = ()\,)\,) \end{aligned}$$

The third stage is to add parameters to avoid free variables. This uses the rule that $X : X = P \equiv X(fn(P)) : X(fn(P)) = P\{(fn(P))/X\}$:

$$\begin{aligned} X(u): \quad & X(u) = u?(x,q).\,Z(u,x,q) : Z(u,x,q) = (X(u) \,|\, Y(x,q) : \\ & Y(x,q) = q?v.\,W(x,q,v) : W(x,q,v) = (\,Y(x,q) \,|\, Put(x,v) : \\ & Put(x,v) = x!v.\,Nil : Nil = ()\,)\,) \end{aligned}$$

Finally, definitions can now be pulled to the top level:

$$\begin{aligned} X(u): \quad & X(u) = u?(x,q)\,.\,Z(u,x,q) : \\ & Z(u,x,q) = (\,X(u) \,|\, Y(x,q)\,) : \\ & Y(x,q) = q?v\,.\,W(x,q,v) : \\ & W(x,q,v) = (\,Y(x,q) \,|\, Put(x,v)\,) : \\ & Put(x,v) = x!v.\,Nil : \\ & Nil = () \end{aligned}$$

All output operations are followed by a process denoting inaction. The Put action definition can be used for all output actions.

5.2 The Graph Rewriting Language Dactl

Dactl provides a notation for describing computational objects as directed graphs, and for specifying computation in terms of pattern-directed rewritings of such graphs. It has been used in studies of the implementation of a range of language styles including functional programming, term rewriting languages [Ken90], and concurrent logic languages [Gla88]. Graph rewriting also shows promise for supporting the integration of different programming styles; [Gla91a] reports early work on integration of functional and logic languages, while the work reported here arises from a collaborative study investigating integration of functional and process-based programming.

[Gla91b] provides a thorough background to the Dactl language. Here we review the language features exploited in the translation of process networks to graph rewriting systems in this paper. The nodes of a Dactl graph are labelled with a symbol which indicates that the node plays the role of an *operator* at the root of a rule application, a *data constructor*, or an *overwritable*. The role of operators and constructors will be familiar. The novel

feature is the use of overwritable nodes which may be modified as a side-effect of rule application. This enables Dactl to express more computational models than the term-graph rewriting which is discussed elsewhere in this book. Overwritables may model von Neumann storage cells, semaphores, and the logic variable. Overwritable nodes will be used in this paper to model the contents of communication channels.

A Dactl graph may be represented by listing the definitions of the nodes giving their identifier, symbol, and a sequence of identifiers for the successor nodes. Repetition of identifiers is used to indicate sharing in a graph:

```
c: Chan[ n ],
n: Nil
```

Symbols are integers or identifiers starting in upper-case, while node identifiers start in lower-case. A node definition may replace one of the occurrences of the node identifier, and redundant identifiers may be removed allowing the equivalent shorthand form:

```
c: Chan[ Nil ]
```

Dactl rules contain a *pattern* to be matched and a body, or *contractum*, to replace the occurrence of the pattern in the graph, or *redex*. Patterns are Dactl graphs but may contain node identifiers lacking a definition which will match an arbitrary node.

The contractum of a rule contains new graph structure to be built, which may reference nodes matched by the pattern, and one or more *redirections*, which indicate that the source of the redirection should be overwritten by the target. In classic term rewriting rules there will always be a redirection overwriting the root of the redex with the root of the contractum. In such rules the pattern and contractum are separated by the symbol `=>`. This is shorthand for a form with an explicit overwrite using `->` as separator.

To control the order of evaluation, attempts to match the rules against the graph only begin at *active* nodes, marked with a $*$ in the representation. If more than one rule matches, an arbitrary choice may be made about which rule to apply; fairness is not assumed. The contractum may use markings to nominate further nodes at which rewriting may take place. If multiple active nodes arise they may be considered in any order, or even in parallel if there is no conflict between possible rewritings.

A parallel composition of processes P and Q both using links x and y might be encoded by a graph:

```
*P[x y], *Q[x y]
```

Nodes may also be created *suspended*, indicated by one or more `#` markings, waiting for *notification* that a successor has been rewritten to a stable form. This enables a rule to create a dataflow graph in which certain nodes are active and will produce results which awaken parent nodes once all arguments are available. Each notification removes one suspension, the node becoming active when the last suspension is removed. Notification typically

takes place when an active node has become a constructor or overwritable. Arcs which will form notification paths are marked with `^`.

As a final illustration, we consider the modelling of links in our translation of the process notation. A channel will consist of an overwritable `Chan` whose argument is a list acting as a stack of available values. To output to the channel, a rule simply overwrites the channel to contain a list prefixed with the new value. To input from a channel, a rule must test for available input. If the channel contains the empty list, the operation is suspended until input is available. Otherwise the channel is rewritten containing the tail of the original list.

The rules for `Put` add a value at the head of the stack of values, while `Get` receives a value and then continues with process `Use` which may manipulate the value received:

```
RULE
     Put[c:Chan[v] d] -> c:= *Chan[Cons[d v]];
     Get[c:Chan[Cons[d v]]] -> *Use[d], c:= Chan[v];
     Get[c:Chan[Nil]] -> #Get[^c];
```

Below we illustrate the rules in action. The first attempt to read from the channel suspends on finding the empty channel `Chan[Nil]`. Note the final state with the channel empty again:

```
       *Get[c],  *Put[c 2],  c:Chan[n:Nil]
=>{3}  #Get[^c], *Put[c 2],  c:Chan[n:Nil]
=>{1}  #Get[^c],  c:*Chan[Cons[2 n:Nil]]
=>{-}  *Get[c],   c:Chan[Cons[2 n:Nil]]
=>{2}  *Use[2],   c:Chan[n:Nil]
=>...
```

The initial state of a link is `Chan[Nil]`, a channel with no messages available. Introduction of new channels is associated with restriction operators in process expressions.

The `Put` rule updates the channel `c` to contain the message referred to by `d` as the first message in its buffer. The effect of the active marker on the updated channel will be to unblock any process attempting to receive on this channel.

The two rules for `Get` describe the behaviour of a process guarded by an input action. The first rule corresponds to the case where there is input available. The value found in the channel is passed to the `Use` process which is made active. The channel is updated to reflect the fact that a message has been delivered. The final rule comes into play if the channel is empty. The `#` and `^` markings indicate that the `Get` process should be suspended until the channel `c` is updated by a `Put` process.

5.3 Translation of Standard Form Networks to Dactl

The Dactl rules below illustrate the translation of the function $\lambda x.x$ applied to the constant value 3:

```
RULE
    INITIAL => *Read[z], *A[z], z:NewChan;
    A[z] -> *X[f], *Put[f Cons[a z]], *Const[a 3],
                f:NewChan, a:NewChan;
    X[u:Chan[Cons[Pair[x q] r]]] -> *Z[u x q], u:= Chan[r];
    X[u:Chan[Nil]] -> #X[^u];
    Z[u x q] -> *X[u], *Y[x q];
    Y[x q:Chan[Cons[v r]]] -> *W[x q v], q:= Chan[r];
    Y[x q:Chan[Nil]] -> #Y[x ^q];
    W[x q v] -> *Y[x q], *Put[x v];
    Const[u:Chan[Cons[v r]] k] -> *C[u k v], u:= Chan[r];
    Const[u:Chan[Nil] k] -> #Const[^u k];
    C[u k v] -> *Const[u k], *Put[v k];
    Read[z] => *Get[r], *Put[z r], r:NewChan;
```

All Dactl programs start with a graph containing a single active node with symbol `INITIAL`. This replaces process P above.

Processes generally correspond to GRS terms. The `*` marks an active process or rewritable term. Parallel composition, used by Z and W, is very straightforward. Restriction, used in A, corresponds to declaration of new links denoted by the term `NewChan`. The following pattern defines such new channels:

```
PATTERN
    NewChan = Chan[Nil];
```

All Output actions involve the primitive `Put` described earlier. Processes guarded by an input action become two rules. The first extracts a value from a channel containing input, while the second applies when no input is available and blocks the process until an output action is performed on the channel.

To output a pair, the `Pair` constructor is used to build the pair, as in `A`. When a pair is input, (for example `x` and `q` in `X`), the pattern matches a pair of values for use in the body of the action (`Z` in this case).

The processes `Read` and `Put` are defined as follows:

```
RULE
    Put[c:Chan[v] d] -> c:= *Chan[Cons[d v]];
    Read[z] -> *Put[z r], *Print[r], r: NewChan;
    Print[x:Chan[Cons[v r]]] ->
        *PrintF["Result: %d" v], x:= Chan[r];
    Print[x:Chan[Nil]] -> #Print[^x];
```

The communication scheme is very simple because of the restricted use of the process model in the translation scheme: lack of a choice operator avoids the need to retract communication offers; ability to use asynchronous communication removes the need for synchronising operations; and there is no need to maintain the order of available messages so a stack may be used.

The overall style of execution contrasts strongly with the more conventional Term Graph Rewriting [Bar87] approach. Instead of representing an expression as a single rooted term, this style represents subexpressions as independent unrooted terms linked by shared references to nodes representing channels. In this way, execution corresponds more to the actions of the Chemical Abstract Machine [Ber90] than a traditional graph reduction machine.

6 Results

A translator has been developed which will convert "programs" in an extended λ-Calculus to the process notation. Several different translations from λ-Calculus to processes have been implemented. The process networks are converted to the sublanguage which makes heavy use of agent definitions. This form is then converted to Dactl.

The mapping from process notation to Dactl does not handle non-trivial processes with output guards (only inaction may follow an output guard). This enables us to express the new translation directly, but the π-Calculus translations of Milner cannot be translated directly. A Form of the Lazy-λ-Calculus translation modified in a manner inspired by [Hon91] has been produced. The translation is extended to handle constants. This has been called *PiLazy*:

$$\begin{aligned}
[\![x]\!]_u &= x!u.() \\
[\![k]\!]_u &= u?v\,.\,v!k.() \\
[\![\lambda x.M]\!]_u &= u?d.\,(\,d!a.() \mid a?x.\,u?v.\,[\![M]\!]_v\,)\backslash a \\
[\![M \mathbin{@_L} N]\!]_u &= (\,[\![M]\!]_v \mid v!d.() \mid d?a.\,(a!t.() \mid v!u.() \mid \mathbf{rec}\,X\,.\,t?w.\,([\![N]\!]_w \mid X))\,\backslash t\,)\backslash d\backslash v
\end{aligned}$$

This translations were compared with *NewLazy* and *NewEager*. These correspond to the new translation converting all applications to lazy or eager form correspondingly. In addition, the tupled communication of our notation was exploited to produce an improved version of *PiLazy*, called *ImpPiLazy*:

$$\begin{aligned}
[\![x]\!]_u &= x!u.() \\
[\![k]\!]_u &= u?v\,.\,v!k.() \\
[\![\lambda x.M]\!]_u &= u?(x,v).\,[\![M]\!]_v \\
[\![M \mathbin{@_L} N]\!]_u &= (\,[\![M]\!]_v \mid v!(t,u).() \mid \mathbf{rec}\,X\,.\,t?w.\,([\![N]\!]_w \mid X))\backslash t\backslash v
\end{aligned}$$

The test programs used were:

$$\begin{aligned}
&\text{IdTest}: && (\lambda x.\,x)\,99 \\
&\text{Twice}: && (\lambda twice.\lambda succ.\,twice\,twice\,succ\,0)\;(\lambda f.\lambda x.f\,(f\,x))\;(\lambda n.Succ(n))
\end{aligned}$$

The efficiency measure used is the number of output actions made. In most cases, every output message is consumed.

Translation	*IdTest*	*Twice*
NewEager	4	49
NewLazy	5	80
PiLazy	7	94
ImpPiLazy	4	52

Note that the lazy translations allow no parallelism, so our main focus is on an eager translation, but with the aim of supporting laziness where it is required. Our translations are superior when function application is taking place, as in *Twice*. *PiLazy* does not perform well since we must avoid synchronous communication. However, the improved version *ImpPiLazy* is best for a purely lazy translation. We have not attempted to reduce recursion in the new translation.

7 Conclusions

A new scheme for translating λ-Calculus expressions to process networks has been presented. The new model allows mixing of lazy and call-by-value strategies. Simple data values and their operators may be incorporated in the translation.

Some early experimental results are presented based on a translation of process networks to Dactl. The results show that the new translation is superior for eager computation, and also performs well for lazy computation.

While implementation efficiency is not our primary concern, the final aim of this work is to investigate possible techniques for practical parallel implementation of languages integrating functional and process styles. The process notation we have developed may be related to graph rewriting opening the question of whether graph rewriting can form the basis for such practical implementations.

References

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

[Bar87] H.P. Barendregt, M.C.J.D. van Eekelen, J.R.W. Glauert, J.R. Kennaway, M.J. Plasmeijer, & M.R. Sleep: *Term Graph Rewriting*, Proc. PARLE 87, Springer LNCS 259, p141-158. (1987)

[Ber90] 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 4, p159-174. (1984)

[Eng86] U. Engberg & M. Nielsen: *A Calculus of Communicating Systems with Label Passing* Report DAIMI PB-205, Computer Science Department, University of Aarhus. (1984)

[Gia89] A. Giacalone, P. Mishra, & S. Prasad: *Facile: A Symmetric Integration of Concurrent and Functional Programming* IJPP, Vol 18, No 2, p121-160. (1989)

[Gla88] J.R.W. Glauert, & G.A. Papadopoulos: *A Parallel Implementation of GHC* Proceedings, International Conference on Fifth Generation Computer Systems. ICOT, Tokyo, December 1988. (1988)

[Gla91a] J.R.W. Glauert, & G.A. Papadopoulos: *Unifying Concurrent Logic and Functional Languages in a Graph Rewriting Framework* Proceedings, 3rd Panhellenic Computer Science Conference. Athens, May 1991. (1991)

[Gla91b] 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)

[Hon91] Honda, K., Tokoro, M., *An object calculus for asynchronous communication*, Proceedings, ECOOP'91, Geneva, July 1991. (1991)

[Ken82] J.R. Kennaway and M.R. Sleep: *Expressions as processes* Proceedings, Lisp and FP, Aug 1982, p.21-28. (1982)

[Ken90] J.R. Kennaway: *Implementing Term Rewrite Languages in Dactl* Theor. Comp. Sci. 72, p.225-250. (1990)

[Let91] L. Leth: *Functional Programs as Reconfigurable Networks of Communicating Processes* Ph. D. Thesis, Imperial College, London University, 1991.

[Mil89] R. Milner: *Communication and Concurrency*, Prentice Hall, 1989.

[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 TR ECS-LFCS-89-85, Edinburgh University, June 1989. (1989)

[Tho89] B. Thomsen: *A Calculus of Higher Order Communicating Systems*, Proceedings of POPL 89, pp. 143-154, The Association for Computing Machinery, 1989.

[Tho90] B. Thomsen: *Calculi for Higher Order Communicating Systems*, Ph. D. Thesis, Imperial College, London University, 1990.