Process Semantics of Graph Reduction

Simon Brock and Gerald Ostheimer
Computer Science Division
University of St Andrews
North Haugh
St Andrews
Fife KY16 9SS
{shb,gerald}@dcs.st-and.ac.uk

Abstract

This paper introduces an operational semantics for call-by-need reduction in terms of Milner's π-calculus. The functional programming interest lies in the use of π-calculus as an abstract yet realistic target language. The practical value of the encoding is demonstrated with an outline for a parallel code generator. From a theoretical perspective, the π-calculus representation of computational strategies with shared reductions is novel and solves a problem posed by Milner [13]. The compactness of the process calculus presentation makes it interesting as an alternative definition of call-by-need. Correctness of the encoding is proved with respect to the call-by-need λ-calculus of Ariola *et al.* [3].

1 Introduction

Graph reduction of extended λ-calculi has become a mature field of applied research. The efficiency of the implementations is due in great measure to a technique known as 'sharing', whereby argument values are computed (at most) once and then memoized for future reference. Both the term 'graph reduction' and the original algorithm are due to Wadsworth [20] who also coined the term 'call-by-need' for the parameter passing mechanism. Modern practice is not to perform reduction inside λ-abstractions, *i.e.*, to stop reduction at weak head normal forms, as pointed out by Abramsky [2], who studies the semantics of the resulting 'lazy' λ-calculus.

Despite the maturity of the applications based on Wadsworth's ideas, the underlying theory of λ-calculi with shared reductions is currently undergoing rapid development. Wadsworth proves that his graph-based implementation strategy is equivalent to conventional call-by-name reduction. However his graph-theoretic presentation is idiosyncratic and little apt to serve as a reference for the underlying principles. Recently, various more elegant proposals have been put forward, of which we will identify a few that we consider to be of particular significance. Yoshida's λf-calculus [21] is a λ-calculus extended by a let-construct whose reduction axioms model the sharing of computation. Leftmost reduction in this calculus essentially models call-by-need, although that fact is not established in the paper. Launchbury presents an operational semantics for graph reduction of λ-terms in [11] with a much simpler set of reduction axioms, albeit at the expense of introducing into these axioms an additional layer of global environments which are not part of the source language. Ariola *et al.* [3] present a very concise treatment of a λ-calculus with sharing, doing away with the need for global environments. Like Yoshida [21], standard reduction in their λ_{let}-calculus models a call-by-need reduction strategy. In addition, they also present a 'let-less' system, capable of modelling call-by-need in the context of the standard λ-syntax.

This paper offers a rather different, alternative approach to call-by-need reduction. We describe the encoding of λ-terms in Milner's π-calculus such that the encoded terms reduce according to a call-by-need reduction strategy. The encoding is interesting for being both extremely concise and of clear practical interest as a foundation for provably correct concurrent implementations of graph reduction. In addition, the π-calculus representation of computational strategies with shared reductions is novel and solves a problem posed by Milner in [13]. We prove correctness of the encoding with respect to the call-by-need calculus of Ariola *et al.* [3]. The proof is performed in two steps. We first introduce an extended, 'pragmatic' version of their calculus where we distinguish several 'states' of let-bindings as is the practice in implementations. We prove that the extended λ_{let}-calculus accurately simulates the original version. In the second step, we give encodings for the extensions and show that reduction of the resulting π-terms accurately simulates reduction of the original terms. The correctness of the original encoding then follows directly.

The structure of the rest of the paper is as follows: after discussing some related work in the next section, we introduce the call-by-need calculus of Ariola *et al.* [3] in section three. In the fourth section we present the subset of Milner's π-calculus we require for our encoding and then present the main result of the paper, the encoding itself. Section five outlines a compilation strategy for the simplified π-calculus employed in the encoding, thus establishing the significance of our work for implementations. The extended λ_{let}-calculus and the correctness proofs are found in section six. The last sections discuss some extensions arising from functional programming practice and possible future work suggested by our results.

2 Related work

This work was largely inspired by the original work of Milner [13] who encoded in π-calculus both Abramsky's 'lazy' λ-calculus [2] and a weak parallel call-by-value λ-calculus but these respective encodings are mutually incompatible. In both cases, substitutions are coded as separate processes and in his concluding remarks Milner discusses that a more natural model would be to implement sharing. One of the aims of the work presented here was to solve this problem. We go further than Milner by relating the encodings to implementations. Glauert [6, 7] describes compatible encodings of lazy and eager evaluation in an asynchronous process notation. It is shown the process notation used can be translated into a graph reduction language. In Ostheimer and Davie [16], an encoding for shared reduction strategies was given in the (synchronous) monadic π-calculus but the correspondence to λ-terms was not formalized. Boudol [4] encodes a version of λ-calculus with explicit substitutions which provides sharing in an asynchronous version of π-calculus. The encoding is more complex and lacks the clear implementation route presented here.

Wadsworth [20] introduced graph reduction for λ-terms. In this paper we are interested in the use of graph reduction to implement functional programming and therefore interested in models which have similar properties to the lazy λ-calculus [2]. A number of researchers, including Ariola *et al.* [3], Jeffrey [10], Launchbury [11] and Yoshida [21], have studied operational models of graph reduction for such terms. As mentioned in the introduction, we use results presented by Ariola *et al.* [3] to establish the correctness of our presentation.

3 Operational semantics of graph reduction

As we will make contact with the work of Ariola *et al.* [3] at various points in this paper, it is appropriate to give a quick summary of their λ_{let}-calculus here. This system extends the normal term structure of λ-calculus with a $\mathsf{l}et$ construct and replaces the usual β-reduction step

Syntactic domains

Variables:			x, y, z
Terms:	$M, N \in \mathcal{L}$	$::=$	$x \mid \lambda x.M \mid M\ N \mid \mathsf{let}\ x = N\ \mathsf{in}\ M$
Values:	V	$::=$	$\lambda x.M$
Answers:	A	$::=$	$V \mid \mathsf{let}\ x = M\ \mathsf{in}\ A$
Evaluation Contexts:	E	$::=$	$[\,] \mid EM \mid \mathsf{let}\ x = M\ \mathsf{in}\ E \mid \mathsf{let}\ x = E\ \mathsf{in}\ E[x]$

Reduction rules:

(let_S-I)	$(\lambda x.M)N$	$\rightarrow$	$\mathsf{let}\ x = N\ \mathsf{in}\ M$
(let_S-V)	$\mathsf{let}\ x = V\ \mathsf{in}\ E[x]$	$\rightarrow$	$\mathsf{let}\ x = V\ \mathsf{in}\ E[V]$
(let_S-C)	$(\mathsf{let}\ x = M\ \mathsf{in}\ A)N$	$\rightarrow$	$\mathsf{let}\ x = M\ \mathsf{in}\ AN$
(let_S-A)	$\mathsf{let}\ x = (\mathsf{let}\ y = M\ \mathsf{in}\ A)\ \mathsf{in}\ E[x]$	$\rightarrow$	$\mathsf{let}\ y = M\ \mathsf{in}\ \mathsf{let}\ x = A\ \mathsf{in}\ E[x]$

Figure 1: The standard call-by-need reduction system λ_{let}

$(\lambda x.M)N \rightarrow M[x := N]$ with the let$_S$-I rule that $(\lambda x.M)N \rightarrow$ let $x = N$ in M. There are then three rules that manipulate nested let constructs. The rewrite rules are introduced in a conventional way but *evaluation contexts* [5] are used to define the context for a redex rather than more usual traversal inference rules in a small step operational semantics. The system evaluates terms through to *answers* which are of the form let $x_1 = M_1$ in ... let $x_n = M_n$ in $\lambda x.M$. The final part of an answer is a term in weak head normal form (a *value*), the let bindings being associated with the free variables in this value. As the system does not evaluate "under λ's" these answers are equivalent to *weak head normal forms* in the lazy λ-calculus. The standard reduction strategy in λ_{let}, shown in Figure 1, models call-by-need reduction. When referring to λ_{let} in this paper we will always mean the standard reduction system. We will refer to the set of terms of λ_{let} as $\mathcal{L}$.

Standard reduction is organized in such a way that in let $x = N$ in M, the subterm N is only reduced when it is needed (*i.e.*, when the value associated with x is needed in M). If N then converges to an answer the value part replaces the needing occurrence of x and all subsequent occurrences of x when they need it. It is this use of *need* to drive the reduction prompted Wadsworth [20] to coin the phrase 'call-by-need'.

The reduction rules define a single step reduction relation $\rightarrow$. The reflexive, transitive closure of $\rightarrow$ will be written $\twoheadrightarrow$. We say that: *M converges to M'*, written $M{\downarrow}M'$, if $M \twoheadrightarrow M' \not\rightarrow$; $M{\downarrow}$ means that M converges to some M'; and $M{\uparrow}$ means M *diverges*, *i.e.*, there does not exist M' such that $M{\downarrow}M'$. This notation will be used with all the rewrite relations presented in this paper. Generally rewrite relations will be disambiguated by context but it is sometimes necessary to distinguish the use of a particular rule which will be denoted by subscripting arrow $\rightarrow$ with the name of the rule (*e.g.*, $(\lambda x.M)N \rightarrow_I$ let $x = N$ in M). Closed terms are defined in the usual way and the set of closed terms will be denoted with a superscript '0' *i.e.*, $\mathcal{L}^0$.

Ariola *et al.* show that the evaluators for call-by-need and call-by-name are equivalent. The paper explains in more detail how the reduction system works and gives examples. They also show that this system can be translated into pure λ-terms (*i.e.*, without the let construct) and that these systems are closely related.

4 A π-calculus encoding

The π-calculus was first presented by Milner *et al.* [14] and is designed to allow the specification of processes with dynamically evolving configuration. Due to space limitations we cannot

Syntactic domains

$$\text{Processes:} \quad P, Q, R \in \mathcal{P} \quad ::= \quad \overline{x}y \mid x(y).P \mid \mathbf{0} \mid P|Q \mid !x(y).P \mid (\nu x)P$$

Structural equivalences

$$\begin{array}{ll} P \equiv Q & \text{if } P \equiv_\alpha Q \\ P|\mathbf{0} \equiv P,\ P|Q \equiv Q|P,\ P|(Q|R) \equiv (P|Q)|R & \\ !P \equiv P|!P & \\ (\nu x)\mathbf{0} \equiv \mathbf{0},\ (\nu x)(\nu y)P \equiv (\nu y)(\nu x)P & \\ (\nu x)(P|Q) \equiv P|(\nu x)Q & \text{if } x \text{ not free in } P \end{array}$$

Reduction rules

$$\text{(COM)} \qquad x(y).P|\overline{x}z \to P\{^z/_y\}$$

$$\text{(PAR)} \qquad \frac{P \to P'}{P|Q \to P'|Q}$$

$$\text{(RES)} \qquad \frac{P \to P'}{(\nu y)P \to (\nu y)P'}$$

$$\text{(STRUCT)} \qquad \frac{Q \equiv P \qquad P \to P' \qquad P' \equiv Q'}{Q \to Q'}$$

Figure 2: The π-calculus

present an introduction to the subject here but in Figure 2 we present the subset we are using. Compared to the presentation of Milner [13] we have imposed two syntactic restrictions. Specifically, we prohibit output actions from guarding non-empty processes, leaving the asynchronous subset of the calculus. Further, we permit the replication only of processes guarded by input actions[1]. The motivation for using a restricted version of the calculus is our desire to use π-calculus as a model for a practical compiler target language. We are only interested in those constructs which have simple translations to message-passing computer architectures. (This will be discussed further in section 5.)

We will occasionally require the atomic transmission of pairs. This is easily encoded in the monadic π-calculus as shown in Milner *et al.* [13]. The implementation strategy of the next section will treat the asynchronous communication of tuples directly.

The core idea of the translation is this: given a closed λ-term M and a name o of the π-calculus the encoding $[\![M]\!]o$ will simulate the reduction of M in the following sense. If M converges in the call-by-name λ-calculus then $[\![M]\!]o$ converges in the π-calculus such that the resulting term delivers a representation of the value of M at the channel o. In the translation, names of λ-variables are mapped directly to names of π-calculus channels. In order to be able to represent a call-by-need reduction strategy, these names will not stand for values, rather they provide access to values upon request.

Remembering that in the pure call-by-name λ-calculus weak head normal forms are our only values, we start by considering the encoding of abstractions,

$$[\![\lambda x.M]\!]o \stackrel{\text{def}}{=} (\nu f)(\overline{o}f \mid !f(x,p).[\![M]\!]p)$$

We can see that the representation of the function value, f, is immediately available at o. The

[1]This is called 'lazy replication' in Honda [9]. Note that the permitted process terms are just those of Honda's ν-calculus [9], but we retain the structural equivalences of Milner [13]

replicated term represents an 'activation server' which allows the function value to be applied to an argument; the replication is necessary as a given function may be applied more than once. A process needing to apply a function sends an 'activation' consisting of (access to) the argument x and the place p where the result is required.

To request the value of a variable x, we send a request r along x. Assuming a suitable encoding of function application (see below), we receive the value of x on r. Thus we can provide the following encoding for variable access.

$$[\![x]\!]o \stackrel{\text{def}}{=} (\nu r)(\overline{x}r \mid r(g).\overline{o}g)$$

A more succinct way to achieve the same goal is to send o directly to x, continuation-passing style.

$$[\![x]\!]o \stackrel{\text{def}}{=} \overline{x}o$$

The encoding of an application (MN) completes the picture. We first need to evaluate the function M, then apply it, in the style suggested by the encoding of abstractions, to a suspended form of the argument N.

$$[\![MN]\!]o \stackrel{\text{def}}{=} (\nu m)([\![M]\!]m|m(f).(\nu a)(\overline{f}(a,o)|\mathsf{bind}(a,N)))$$

How is this suspended binding of N to a to be achieved? When the body of M requires the value of its argument, it sends a request along a. So the first action of the binding process must be to receive this request. Only then may the evaluation of N be initiated. Once the value of n is available, the original request must be satisfied and the value of n 'stored' for future reference. We get

$$\mathsf{bind}(a,N) \stackrel{\text{def}}{=} a(r).(\nu n)([\![N]\!]n \mid n(g).(\overline{r}g \mid \mathsf{store}(a,g)))$$

The effect of a (single-assignment) store is achieved with

$$\mathsf{store}(a,g) \stackrel{\text{def}}{=} !a(r).\overline{r}g$$

The solution outlined above, while perfectly workable in practice, turns out to be problematic in that encodings of non-terminating λ-terms may not be confluent. The problem arises from the fact that by consuming the first request for a value, we treat it differently from subsequent requests. While non-confluence only shows up once we add concurrent reduction strategies, distinguishing the initial request complicates the correctness proof in the sequential case.

Fortunately there is a straightforward solution to the problem: after consuming the initial request we can re-generate it, in order to consume it again only when the argument value is available. Incorporating this change we get

$$\mathsf{bind}(a,N) \stackrel{\text{def}}{=} a(r).(\nu n)(\overline{a}(r) \mid [\![N]\!]n \mid n(g).\mathsf{store}(a,g))$$

Figure 3 summarises the encoding. Note that we have moved the declarations of a and n to the top of the application rule. This modification ensures that the term satisfies the 'handler locality' property of section 5 where we discuss implementation.

Let us observe the behaviour of encoded terms with the help of some examples.

Example 4.1 Consider the case of the simplest λ-redex, $(\lambda x.M)N$. We omit ν-declarations to save space.

$$
\begin{aligned}
[\![x]\!]o &\stackrel{\text{def}}{=} \overline{x}o \\
[\![\lambda x.M]\!]o &\stackrel{\text{def}}{=} (\nu f)(\overline{o}f \mid !f(x,p).[\![M]\!]p) \\
[\![MN]\!]o &\stackrel{\text{def}}{=} (\nu a,m,n)([\![M]\!]m \mid m(f).(\overline{f}(a,o) \mid a(r).(\overline{a}r \mid [\![N]\!]n \mid n(g).!a(r).\overline{r}g)))
\end{aligned}
$$

Figure 3: The encoding of closed λ terms

$$
\begin{aligned}
[\![(\lambda x.M)N]\!]o &\stackrel{\text{def}}{=} \left(\begin{array}{l} [\![(\lambda x.M)]\!]m \mid \\ m(f).\left(\overline{f}(x,o) \mid x(r).\left(\begin{array}{l} \overline{x}r \mid \\ [\![N]\!]n \mid \\ n(g).!x(r).\overline{r}g \end{array} \right)\right) \end{array} \right) \\
&\stackrel{\text{def}}{=} \left(\begin{array}{l} \overline{m}f \mid !f(x,p).[\![M]\!]p \mid \\ m(f).\left(\overline{f}(x,o) \mid x(r).\left(\begin{array}{l} \overline{x}r \mid \\ [\![N]\!]n \mid \\ n(g).!x(r).\overline{r}g \end{array} \right)\right) \end{array} \right) \\
&\rightarrow \left(\begin{array}{l} !f(x,p).[\![M]\!]p \mid \\ \overline{f}(x,o) \mid x(r).\left(\begin{array}{l} \overline{x}r \mid \\ [\![N]\!]n \mid \\ n(g).!x(r).\overline{r}g \end{array} \right) \end{array} \right) \\
&\rightarrow \left(\begin{array}{l} !f(x,p).[\![M]\!]p \mid \\ [\![M]\!]o \mid x(r).\left(\begin{array}{l} \overline{x}r \mid \\ [\![N]\!]n \mid \\ n(g).!x(r).\overline{r}g \end{array} \right) \end{array} \right) \qquad (*)
\end{aligned}
$$

Note that the encoding performs two reductions without making any assumptions about the shape of M. Note also that in the case of M being an abstraction, the above process is in normal form and ready to deliver the value of M on o, without performing a copy step. This conveys a strong flavour of 'answers' in the λ_{let}-calulus, which would reduce the above term to $\mathsf{let}\ x = N\ \mathsf{in}\ M$. The analogy will shortly be made more precise.

Example 4.2 For the moment, let us observe the modelling of β-reduction by assuming in the above $M \equiv x$ and N a value, *i.e.*, $[\![N]\!]n \equiv (\overline{n}h \mid Q)$ where Q is the activation server for N accessible through h (ν-declarations have again be omitted). Then $(*)$ reduces as

$$
\begin{aligned}
\left(\begin{array}{l} !f(x,p).[\![x]\!]p \mid \\ [\![x]\!]o \mid x(r).\left(\begin{array}{l} \overline{x}r \mid \\ [\![N]\!]n \mid \\ n(g).!x(r).\overline{r}g \end{array} \right) \end{array} \right) &\stackrel{\text{def}}{=} \left(\begin{array}{l} !f(x,p).[\![x]\!]p \mid \\ \overline{x}o \mid x(r).\left(\begin{array}{l} \overline{x}r \mid \\ (\overline{n}h \mid Q) \mid \\ n(g).!x(r).\overline{r}g \end{array} \right) \end{array} \right) \\
&\rightarrow \left(\begin{array}{l} !f(x,p).[\![x]\!]p \mid \\ \overline{x}o \mid (\overline{n}h \mid Q) \mid n(g).!x(r).\overline{r}g \end{array} \right) \\
&\rightarrow \left(\begin{array}{l} !f(x,p).[\![x]\!]p \mid \\ \overline{x}o \mid Q \mid !x(r).\overline{r}h \end{array} \right) \\
&\rightarrow \left(\begin{array}{l} !f(x,p).[\![x]\!]p \mid \\ \overline{o}h \mid Q \mid !x(r).\overline{r}h \end{array} \right)
\end{aligned}
$$

The latter is in normal form and delivers the value of N at o as required.

As is clear from the examples, an encoded term may traverse several more states before converging than the original. What can we say of the intermediate states? It turns out that some of these have natural analogues in the λ_{let}-calculus. In particular, we can interpret the term in $(*)$ as an encoding of the let-construct. As the activation handler $!f(x,p).[\![M]\!]p$ cannot participate in further reductions and does therefore not interfere with the simulation we can drop it from the encoding and retain

$$[\![\mathsf{let}\ x = N\ \mathsf{in}\ M]\!]o \stackrel{\text{def}}{=} (\nu x)(x(r).(\overline{x}r \mid [\![N]\!]n \mid n(g).!x(r).\overline{r}v) \mid [\![M]\!]o)$$

By reinterpreting the encoding of pure λ-terms as an encoding of the λ_{let}-calculus we gain access to the results of Ariola *et al.* [3]. For the correctness proof of section 6 we shall find it convenient to identify several further intermediate states and define their analogues in a λ-calculus setting by extending λ_{let}.

In summary, we would like to point out that the behaviour of the encoding has very much an operational flavour, befitting a specification of a reduction *strategy*. In particular, since the π-calculus only requires the communication of atomic names, rather than of arbitrarily complex terms, our solution makes substitution explicit in the sense of Abadi *et al.* [1]. Yet the encoding of Figure 3 is arguably as concise as any specification of call-by-need that we have seen.

5 Implementing the π-calculus

While the reader may agree at this point that the encoding of the previous section is interesting on theoretical grounds, the case for its relevance to implementations requires some elaboration. In this section we sketch a strategy for compiling the asynchronous π-calculus to concurrent machine code. The machine model underlying our approach is that of 'active messages' [19][2] of von Eicken *et al.* which is supported on various MIMD machines. This is based on asynchronous message passing where the sender of a message can specify the address of an entry point to be invoked on message arrival. (An SPMD programming model is assumed, *i.e.*, a uniform code image is residing on all processor nodes.) It is worthy of note that, while the encoding of the previous section is completely sequential, we have investigated compatible encodings of concurrent reduction strategies (reported in Ostheimer and Davie [16]) which yield to the implementation strategy outlined below.

5.1 Properties of the encoding

In section 4 we have restricted the π-calculus to asynchronous communication and lazy replication When studying the encoding more closely, we can identify further restrictions being observed which will aid the implementation effort.

To this purpose, let us define *actions* A to be the following subset of π-calculus terms:

$$A \quad ::= \quad \mathbf{0} \mid \overline{x}(\vec{y}) \mid (\nu \vec{x})P \mid A|A$$

Note that we implement restrictions on multiple variables and communication of multiple names directly, hence the vector notation. We refer to the actions above, respectively, as *empty* action, *message* action, *allocation* action and *composite* action. By P we mean here the subset

$$P \quad ::= \quad A \mid x(\vec{y}).P \mid !x(\vec{y}).A \mid P|P$$

We refer to these types of processes, respectively, as *actions*, *transient* handlers, *persistent* handlers and *composite* handlers. Then a simple induction on the structure of the encoding

[2]This is very similar to the programming model for the STAR:DUST architecture described in Ostheimer [15]

shows that we map λ-terms to actions—a message action in the case of encoding a variable and allocation actions in the encodings of λ-abstraction and application.

Further useful properties of the encoding can be expressed with the aid of the following definitions.

Definition 5.1 An allocation action A is said to possess the *handler locality* property if for all subterms $x(y).P$ (or $!x(y).P$, respectively), the occurrence of x is bound by the nearest enclosing allocation action.

In practical terms, this will allow us to identify statically the complete set of handlers which may receive input on any given channel name. In particular, when sending a channel name to a target process, we can be certain that it can only be used as an output channel, but not for input.

Definition 5.2 The *entry points* of a process P, written $\mathrm{EP}(P)$, are defined recursively as

$$\begin{array}{lcl} \mathrm{EP}(A) & := & \emptyset \\ \mathrm{EP}(x(\vec{y}).P)) & := & \{x\} \cup \mathrm{EP}(P) \\ \mathrm{EP}(!x(\vec{y}).A) & := & \{x\} \\ \mathrm{EP}(P_1|P_2) & := & \mathrm{EP}(P_1) \cup \mathrm{EP}(P_2) \end{array}$$

Intuitively, the entry points of P are just the names on which P may receive input, excluding those inside nested allocation actions. For allocation actions $(\nu\vec{x})P$ which have the handler locality property we can say that $\mathrm{EP}(P) \subseteq \vec{x}$.

Definition 5.3 A term is said to possess the *handler separation* property if all subterms of the form $P_1|P_2$ satisfy $\mathrm{EP}(P_1) \cap \mathrm{EP}(P_2) = \emptyset$.

In practical terms, this means that we can uniquely identify, at any point during the reduction process, which handler will be responsible for the next message arriving on a given channel. With respect to the encoding, a simple induction shows that the π-terms produced have both the handler locality and handler separation properties.[3]

5.2 Representations

Fundamental to our implementation is the representation of channels. The value of a channel is represented as a pair (E, C) consisting of a pointer to an environment E and a code entry point C. Transmission of channel names then becomes transmission of pairs. An environment is a structure in a global heap which is allocated upon executing an allocation action (see below). The handler locality property allows us to associate a unique environment with each channel name. The program fragment corresponding to an entry point for a given channel is responsible for receiving all messages transmitted on this channel and executes in the context of the channel's environment.

An allocation action $(\nu\vec{x})P$ gives rise to the allocation of an environment, containing space for all of the following:

1. the 'state' of the environment, specifying which of the transient handlers in P have completed,

2. a pointer for each handler in $\vec{x}$ to a list of messages arriving while the handler is guarded and

[3] It can be shown that more general π-terms can be suitably transformed to satisfy the named properties, but this exceeds the scope of the present paper.

3. the values of all channels which occur free inside any of the object's handlers (but not for the entry points, whose values are known statically).

There is plenty of scope for static analysis to achieve space optimizations. We will not enter into a discussion of this topic here.

5.3 Compilation of actions

We compile a message action $\overline{x}(\vec{y})$ where the value of x is (E, C) into a program fragment which sends an active message to the processor holding the environment E. The code at the entry point C will eventually receive the message and start execution in the context of E.

An allocation action $(\nu\vec{x})P$ is translated to a program fragment which allocates an environment as outlined above, sets up its initial state (e.g., storing the values of free variables) and then performs the unguarded actions in P.

A compound action $A_1|A_2$ is translated into a program fragment which performs the actions in A_1 and those in A_2 in any order. Imposing an arbitrary order is acceptable as actions quickly execute to completion. In particular, actions never wait for synchronization and can neither loop nor recurse.

5.4 Compilation of handlers

For each entry point in a process term we generate a unique entry point in the code. The resulting code sequence is responsible for dealing with incoming messages. Each message carries a pointer to the environment associated with it. Based on the state found in the environment, the code for an entry point dispatches the appropriate handler or, in the case when the next handler is still guarded, it stores the message for future reference.

A transient handler $x(\vec{y}).P$ performs the unguarded actions in P and identifies those handlers in P which have now become unguarded. In the case when any of these handlers already have messages waiting for them, it initiates those handlers directly. Note that the handler locality and handler separation properties guarantee that the newly unguarded handlers each have unique access to the next input from one channel.

A persistent handler $!x(\vec{y}).A$ performs the actions in A and terminates.

5.5 Comments

In the sketch above we have not addressed the important point of physical distribution of processes. For reasons of achieving sufficient granularity, we propose not to distribute terms of the form $P|Q$, but only instances of certain replication terms, *i.e.*, P in terms of the form $!x(y).P$. One can then exploit the resulting locality by implementing some communications constructs in terms of local memory access. The granularity problem can be further alleviated by encoding supercombinators instead of lambda abstractions.

6 Correctness

The system λ_{let} presented in Section 3 describes what it means for a reduction relation on λ-terms to implement 'call-by-need'. In particular, it highlights that the duplication of work inherent in call-by-name does not occur because an argument to a λ expression is only evaluated when it is first needed and subsequently made available as required. Therefore for another reduction system to implement call-by-need it must simulate the system λ_{let}. For the simulation to be evident there must be some point of 'contact' between the two reduction relations. The obvious place to make this contact is the application of the let_S-I rule. Therefore whenever λ_{let}

performs a let_S-I step, the simulating system must perform an equivalent operation but it must not perform any more of these steps. Also, if the simulating system performs extra steps, the number of these extra steps must be bounded.

In this section it will be shown that the encoding presented in Figure 3 does implement call-by-need. To show this, it will be necessary to relate the reduction of the encoding of a λ-term to reduction in λ_{let}. This is complicated by the fact that the encoding includes certain pragmatics details which are not included in λ_{let}. Therefore in the next sub-section, an annotated version of λ_{let} is introduced and then in the following sub-section this new system is shown to implement call-by-need. We then show that the encodings of this extended system occur at points in the reduction of the encoding of a λ-term. Therefore when we show that the encoding of the extended system implements call-by-need, we are showing that the encoding of λ-terms also implements call-by-need.

6.1 A pragmatic extension to λ_{let}

In λ_{let}, given that $M \to N$ then $\mathsf{let}\ x = M\ \mathsf{in}\ E[x] \to \mathsf{let}\ x = N\ \mathsf{in}\ E[x]$. In a message passing implementation, such as our π-calculus encoding, it might be expected that the process encoding the bound x in the evaluation context E sends a message to the process encoding $x = N$ which would then cause N to reduce to an answer. The process encoding the bound x does not have to take any further part in the reduction of N, except to be informed that N has converged. It can therefore be seen that there are three distinct states to a $\mathsf{let}\ x = N\ \mathsf{in}\ M$ term: firstly when M is reducing but does not need x; secondly the reduction of M needs the value associated with x and N is being reduced; and thirdly, N has converged and the value part can be copied into M. These states will be represented by annotating the let with a subscript. The annotated system will be called λ_{let_*} and its terms can be related back to the originals via a simple function that erases the annotations. These states can also be found in some implementations of functional languages, such as the GRIP machine [17].

The initial creation of a let becomes $(\lambda x.M)N \to \mathsf{let}_S\ x = N\ \mathsf{in}\ M$ where the subscript implies the evaluation of N is suspended. Subsequently when M needs x then the reduction $\mathsf{let}_S\ x = N\ \mathsf{in}\ E[x] \to \mathsf{let}_E\ x = N\ \mathsf{in}\ E[x] \to \mathsf{let}_E\ x = N'\ \mathsf{in}\ E[x]$ occurs, the let_E implying the bound term is evaluating. Subsequently, if N converges then let_E can enter a third state in which it can make the value part of the answer available to needing bound occurrences of x.

At this point the role of values must be considered and in particular how they are copied across multiple let clauses. For example, in λ_{let}:

$$\mathsf{let}\ y = V\ \mathsf{in}\ \mathsf{let}\ x = y\ \mathsf{in}\ xN \quad \to^2 \quad \mathsf{let}\ y = V\ \mathsf{in}\ \mathsf{let}\ x = V\ \mathsf{in}\ VN$$

It appears that the value V is duplicated twice when it is only required once at the point of the application, *i.e.*, at VN. A distinction will be made between where copies are produced by evaluation and where they are used, so that values are only copied into the left hand side of applications.

Therefore, if a term bound in a let binding converges to a value, *i.e.*, the result of a reduction sequence in λ_{let_*} produces a binding of the form $\ldots \mathsf{let}_E\ y = V\ \mathsf{in}\ \ldots$ then this becomes a binding of the form $\ldots \mathsf{let}_D\ y = V_f\ \mathsf{in}\ \ldots$ where the name f is fresh. The intuition behind this annotation is that the value V is at a location f. In the subsequent encoding a distinction can now be made between the defining occurrence of the value and the copy of a value. The idea is that a defining occurrence resides at some location and the value is only copied from that location when it is copied into the left hand side of an application. This leads to the reduction sequence:

Syntactic Domains

$$\begin{array}{llcl}
\text{Locations:} & & & f, g \\
\text{Variables:} & & & x, y, z \\
\text{Values:} & V & ::= & \lambda x.M \\
\text{Located values:} & F & ::= & V_f \\
\text{Answers:} & A & ::= & V \mid V_f \mid \mathsf{let}_* \ x = M \ \mathsf{in} \ A \\
\text{Terms:} & M, N \in \mathcal{L}_* & ::= & x \mid \lambda x.M \mid M \ N \mid \mathsf{let}_S \ x = N \ \mathsf{in} \ M \mid \\
 & & & \mathsf{let}_E \ x = N \ \mathsf{in} \ M \mid \mathsf{let}_D \ x = V_f \ \mathsf{in} \ M \\
\text{Evaluation Contexts:} & E & ::= & [\,] \mid EM \mid \mathsf{let}_* \ x = M \ \mathsf{in} \ E \mid \mathsf{let}_E \ x = E \ \mathsf{in} \ E[x] \\
 & & & \hfill \text{for } \mathsf{let}_S, \mathsf{let}_E, \mathsf{let}_D
\end{array}$$

Reduction rules

$$\begin{array}{llcll}
(\mathsf{let}_P\text{-I}) & (\lambda x.M)N & \to & \mathsf{let}_S \ x = N \ \mathsf{in} \ M & \\
 & (\lambda x.M)_f N & \to & \mathsf{let}_S \ x = N \ \mathsf{in} \ M & \\
(\mathsf{let}_P\text{-V}) & \mathsf{let}_E \ x = F \ \mathsf{in} \ E[x] & \to & \mathsf{let}_E \ x = F \ \mathsf{in} \ E[F] & \\
 & \mathsf{let}_D \ x = F \ \mathsf{in} \ E[x] & \to & \mathsf{let}_D \ x = F \ \mathsf{in} \ E[F] & \\
(\mathsf{let}_P\text{-C}) & (\mathsf{let}_* \ x = M \ \mathsf{in} \ A)N & \to & \mathsf{let}_* \ x = M \ \mathsf{in} \ AN & \text{for } \mathsf{let}_S, \mathsf{let}_E, \mathsf{let}_D \\
(\mathsf{let}_P\text{-A}) & \mathsf{let}_E \ x = (\mathsf{let}_S \ y = M \ \mathsf{in} \ A) \ \mathsf{in} \ E[x] & \to & \mathsf{let}_S \ y = M \ \mathsf{in} \ \mathsf{let}_E \ x = A \ \mathsf{in} \ E[x] & \\
 & \mathsf{let}_E \ x = (\mathsf{let}_D \ y = M \ \mathsf{in} \ A) \ \mathsf{in} \ E[x] & \to & \mathsf{let}_D \ y = M \ \mathsf{in} \ \mathsf{let}_E \ x = A \ \mathsf{in} \ E[x] & \\
(\mathsf{let}_P\text{-E}) & \mathsf{let}_S \ x = M \ \mathsf{in} \ E[x] & \to & \mathsf{let}_E \ x = M \ \mathsf{in} \ E[x] & \\
(\mathsf{let}_P\text{-D}) & \mathsf{let}_E \ x = V \ \mathsf{in} \ E[x] & \to & \mathsf{let}_D \ x = V_f \ \mathsf{in} \ E[x] & (f \text{ fresh})
\end{array}$$

Figure 4: The annotated call-by-need reduction system λ_{let_*}

$$\begin{array}{ll}
 & \mathsf{let}_E \ y = V \ \mathsf{in} \ \mathsf{let}_E \ x = y \ \mathsf{in} \ xN \\
\to & \mathsf{let}_D \ y = V_f \ \mathsf{in} \ \mathsf{let}_E \ x = y \ \mathsf{in} \ xN \\
\to & \mathsf{let}_D \ y = V_f \ \mathsf{in} \ \mathsf{let}_E \ x = V_f \ \mathsf{in} \ xN \\
\to & \mathsf{let}_D \ y = V_f \ \mathsf{in} \ \mathsf{let}_E \ x = V_f \ \mathsf{in} \ V_f N
\end{array}$$

The complete reduction system λ_{let_*} is defined in Figure 4. There are two new rules which handle the propagation of need for a value associated with a binding and the formation of a defining occurrence of a value. The rest of the definition parallels that of λ_{let}. For certain rules (such as the definition of evaluation environments and the let_P-A rule) more cases have been added.

It should be noted that the use of locations for values could have been represented in either system by introducing an extra let binding. Therefore the expression $\mathsf{let}_D \ x = V_f \ \mathsf{in} \ M$ in λ_{let_*} could be represented in λ_{let} as $\mathsf{let} \ f = V \ \mathsf{in} \ \mathsf{let} \ x = f \ \mathsf{in} \ M$. From one point of view this is more pleasing as it is in keeping with the nature of the original system but it leads to a number of technical complications. For example, the let_S-I rule becomes

$$\mathsf{let} \ f = \lambda x.M \ \mathsf{in} \ E[fN] \quad \to \quad \mathsf{let} \ f = \lambda x.M \ \mathsf{in} \ E[\mathsf{let} \ x = N \ \mathsf{in} \ M]$$

The definition of evaluation environments must also be extended and there is no longer a simple relationship between terms in $\mathcal{L}_*$ and $\mathcal{L}$.

6.2 The relation between λ_{let_*} and λ_{let}

Clearly reduction in these two systems is closely related. We require that for all terms $M \in \mathcal{L}$ that if $M \downarrow N$ then for some lifting function $\tau : \mathcal{L} \to \mathcal{L}_*$ it is the case that $\tau(M) \downarrow \tau(N)$. Towards that result, a lemma for single step reduction in λ_{let} being simulated in λ_{let_*} is required. As a preliminary, the obvious mapping from $\mathcal{L}_*$ to $\mathcal{L}$ is defined as:

Definition 6.1 If $M' \in \mathcal{L}_*$ then $|M'| \in \mathcal{L}$ is obtained from M' by erasing all subscripts from let's and values.

The method of lifting terms in $\mathcal{L}$ to $\mathcal{L}_*$ is defined in three stages. The first stage is to map let statements in $\mathcal{L}$ to let_S statements in $\mathcal{L}_*$.

Definition 6.2 For all $M \in \mathcal{L}$ then $\sigma(M) \in \mathcal{L}_*$ obtained from M by replacing all let's with let_S's.

The second stage is to impose on a term some regime for the sharing of values. When considering a term lifted from $\mathcal{L}$ to $\mathcal{L}_*$ we cannot know what reductions have occurred to produce this term and how any values in that term have arisen. Therefore we must take into account any valid sharing of these values. We do know, however, that a shared value (which can only arise from a reduction of a term) cannot appear under a λ and if a value is shared then there is a unique defining occurrence for it.

Definition 6.3 For all $M \in \mathcal{L}$ where $M' = \sigma(M)$ then N' imposes a *value sharing regime* on M' if when a value V occurs in some context in M' then

1. V occurs in the similar context in N', or
2. V_f occurs in the similar context in N' (but not under a λ) and
   - it is either a unique defining occurrence of V_f, *i.e.*, it occurs in a sub-term of the form $\mathsf{let}_D\ x = V_f\ \mathsf{in}\ \ldots$, or
   - there is a unique defining occurrence of V_f whose scope covers this occurrence

The third stage is to consider the role of the let_P-E rule which makes a term in a binding eligible for reduction. A single reduction in λ_{let} may require some number of these *propagation of demand* steps in λ_{let_*} before the corresponding reduction can occur. However, for finite terms this propagation is clearly finite, leading to the definition below:

Definition 6.4 If $M' \in \mathcal{L}_*$ then N' is a *propagation* of M' if $M' \twoheadrightarrow_E N'$.

Using the above definitions, a method of lifting $M \in \mathcal{L}$ to a set of terms in $\mathcal{L}_*$ is defined as:

Definition 6.5 For all $M \in \mathcal{L}$, $M' \in \mathcal{L}_*$ we say M' is a member of the set $\mathsf{lift}\ M$ if

- $M' \equiv \sigma(M)$
- M' imposes a value sharing regime on $\sigma(M)$, or
- M' is a propagation of M'' and M'' imposes a value sharing regime on $\sigma(M)$.

The first step to the required simulation lemma is then:

Lemma 6.6 *For all $M \in \mathcal{L}$ if $M \to N$ then for all $M' \in \mathsf{lift}\ M$ such that $M' \twoheadrightarrow N'$ we have $N' \in \mathsf{lift}\ N$.*

Proof:

Reduction in λ_{let} is such that if $M \equiv E[\Delta]$ where E is an evaluation environment and Δ is an instance of the left hand side of a reduction rule (cf a *redex*) then reduction does not affect E. When M is lifted to the corresponding $E'[\Delta']$ it may not be the case that $|E'| \equiv E$ and $|\Delta'| \equiv \Delta$. However, from the definition of lifting and propagation it can be seen that $E'[\Delta'] \twoheadrightarrow_P E''[\Delta'']$ implies $|E''| \equiv E$ and $|\Delta''| \equiv \Delta$. Therefore to prove the hypothesis it is sufficient to show the correspondence to reduction rules of λ_{let}:

$(\lambda x.M)N \rightarrow \textsf{let } x = N \textsf{ in } M$

When the left hand side is lifted, $(\lambda x.M)$ may participate in sharing regime, in which case the second case of the $\textsf{let}_P$-I rule applies. Otherwise the first case applies.

$\textsf{let } x = V \textsf{ in } E[x] \rightarrow \textsf{let } x = V \textsf{ in } E[V]$

The lift of the left hand side can lead to the reduction: $\textsf{let}_S\ x = V \textsf{ in } E[x] \rightarrow \textsf{let}_E\ x = V \textsf{ in } E[x] \rightarrow \textsf{let}_D\ x = V_f \textsf{ in } E[x] \rightarrow \textsf{let}_D\ x = V_f \textsf{ in } E[V_f]$, for a fresh location f. Similarly with a sharing regime, the left hand side can be either $\textsf{let}_D\ x = V_f \textsf{ in } E[x]$, $\textsf{let}_E\ x = V_f \textsf{ in } E[x]$ or $\textsf{let}_S\ x = V_f \textsf{ in } E[x]$. The first two cases follow immediately from the definition of λ_{let_*}. The last case follows after reduction.

$(\textsf{let } x = M \textsf{ in } A)N \rightarrow \textsf{let } x = M \textsf{ in } AN$

Follows from the definition of lifting and the reduction relation.

$\textsf{let } x = (\textsf{let } y = M \textsf{ in } A) \textsf{ in } E[x] \rightarrow \textsf{let } y = M \textsf{ in let } x = A \textsf{ in } E[x]$

After lifting and propagation the leading $\textsf{let}$ will become $\textsf{let}_E$ and can be reduced to produce a lifted right hand side.

□

The main lemma of this section can now be proved:

Lemma 6.7 *For all $M \in \mathcal{L}$ with $M \twoheadrightarrow N$ in λ_{let} there exists $N' \in \mathcal{L}_*$ such that $\sigma(M) \twoheadrightarrow N'$ in λ_{let_*} and $N' \in \textsf{lift}\ N$.*

Proof: By the definition of $\textsf{lift}$ and successive applications of lemma 6.6. □

Corollary 6.8 *For all $M \in \mathcal{L}$, the reduction of $\sigma(M)$ in λ_{let_*} implements call-by-need.*

Proof: By inspecting the proof of Lemma 6.6 it can be seen that a reduction sequence in λ_{let} is simulated by a longer reduction sequence in λ_{let_*}. However, there is a one-to-one correspondence between the reduction steps in λ_{let} and the equivalent ones in λ_{let_*}. The extra reduction steps arise from the use of $\textsf{let}_P$-E and $\textsf{let}_P$-D rules. By inspection it can be seen that for all $M \in \mathcal{L}$ such that $M \twoheadrightarrow N$ in n steps, then $\sigma(M) \twoheadrightarrow N'$ in no more than $3n$ steps. □

6.3 Encoding the extensions

The extensions used in λ_{let_*} can be encoded in π-calculus as shown in Figure 5. Note that these encodings already occur as intermediate states during reduction of terms encoded via the rules in Figure 3.

Example 6.9 In λ_{let} the term $(\textsf{let } x = \lambda y.M \textsf{ in } x) \rightarrow (\textsf{let } x = \lambda y.M \textsf{ in } \lambda y.M)$. Lifting the original term to $\mathcal{L}_*$ leads to the following reduction:

$$\begin{array}{rll} & \textsf{let}_S\ x = \lambda y.M \textsf{ in } x & (1) \\ \rightarrow & \textsf{let}_E\ x = \lambda y.M \textsf{ in } x & (2) \\ \rightarrow & \textsf{let}_D\ x = (\lambda y.M)_f \textsf{ in } x & (3) \\ \rightarrow & \textsf{let}_D\ x = (\lambda y.M)_f \textsf{ in } (\lambda y.M)_f & (4) \end{array}$$

$$
\begin{array}{rcl}
[\![V_f]\!]o & \stackrel{\mathrm{def}}{=} & \overline{o}f \\
[\![\mathsf{let}_S\ x = M\ \mathsf{in}\ N]\!]o & \stackrel{\mathrm{def}}{=} & (\nu x)(x(r).(\overline{x}r \mid [\![M]\!]n \mid n(g).!x(r).\overline{r}g) \mid [\![N]\!]o) \\
[\![\mathsf{let}_E\ x = M\ \mathsf{in}\ N]\!]o & \stackrel{\mathrm{def}}{=} & (\nu x)([\![M]\!]n \mid n(g).!x(r).\overline{r}g \mid [\![N]\!]o) \\
[\![\mathsf{let}_E\ x = V_f\ \mathsf{in}\ N]\!]o & \stackrel{\mathrm{def}}{=} & (\nu x)(!x(r).\overline{r}f \mid [\![N]\!]o) \\
[\![\mathsf{let}_D\ x = (\lambda x.M)_f\ \mathsf{in}\ N]\!]o & \stackrel{\mathrm{def}}{=} & (\nu x)(!x(r).\overline{r}f \mid !f(x,p).[\![M]\!]p \mid [\![N]\!]o)
\end{array}
$$

Figure 5: Encoding the extensions

When this is encoded into π-calculus the following reductions occur, the numbered points corresponding to those above via the encoding in Figure 5.

$$
\begin{array}{rll}
 & [\![\mathsf{let}_S\ x = \lambda y.M\ \mathsf{in}\ x]\!]o & (1) \\
\stackrel{\mathrm{def}}{=} & (\nu x)(x(r).(\overline{x}r \mid [\![\lambda y.M]\!]n \mid n(g).!x(r).\overline{r}g)) \mid \overline{x}o) & \\
\rightarrow & (\nu x)(\overline{x}o \mid [\![\lambda y.M]\!]n \mid n(g).!x(r).\overline{r}g) & (2) \\
\stackrel{\mathrm{def}}{=} & (\nu x)(\nu f)(\overline{x}o \mid \overline{n}f \mid !f(y,p).[\![M]\!]p \mid n(g).!x(r).\overline{r}g)) & \\
\rightarrow & (\nu x)(\nu f)(\overline{x}o \mid !f(y,p).[\![M]\!]p \mid !x(r).\overline{r}f)) & (3) \\
\rightarrow & (\nu x)(\nu f)(!f(y,p).[\![M]\!]p \mid !x(r).\overline{r}f \mid \overline{o}f)) & (4)
\end{array}
$$

This π-calculus reduction sequence would also have arisen if the original λ_{let}-term had been encoded in π-calculus. However, the encoding of the result is:

$$
\begin{array}{rl}
 & [\![\mathsf{let}\ x = \lambda y.M\ \mathsf{in}\ \lambda y.M]\!]o \\
\stackrel{\mathrm{def}}{=} & (\nu\ x)(x(r).(\overline{x}r \mid [\![\lambda y.M]\!]n \mid n(g).!x(r).\overline{r}g) \mid [\![\lambda y.M]\!]o)
\end{array}
$$

showing that the value sharing has been lost.

6.4 Correctness of encoding of λ_{let_*}

Following Milner [13] we require the *r-determinacy* property of the encoding which is similar to the sequential nature of the reduction in λ_{let}.

Definition 6.10 $P \in \mathcal{P}$ is *r-determinate* if whenever $P \twoheadrightarrow Q$ and also $Q \rightarrow Q_1$ and $Q \rightarrow Q_2$, then $Q_1 \equiv Q_2$.

We can now define a correspondence between the *closed* terms $\mathcal{L}^0_*$ which possess value sharing regimes and π-terms:

Definition 6.11 Let the relation $\mathcal{S} \subseteq (\mathcal{L}^0_* \times \mathcal{P})$ contain all pairs $(M', [\![M']\!]o)$ for some channel o such that M' possesses a value sharing regime.

Lemma 6.12 *For all $(M', P) \in \mathcal{S}$ then P is r-determinate and either:*

A. M' is an answer and $P \downarrow P'$

B. For some $(N', Q) \in \mathcal{S}$, $M' \rightarrow N'$ and $P \twoheadrightarrow Q$.

Proof: If M' is an answer then by inspection of the encoding there can be no reduction and clearly $P \downarrow P'$.

If M' is reducible it is of the form $E'[\Delta']$ where E' is an evaluation environment and Δ' is an instance of a left hand side of one of the reduction rules. From the definition of the encoding it can be seen that no reductions will occur in the encoding of the environment and therefore it is sufficient to consider only the left hand sides of the reduction relation. There are therefore six cases to prove:

1. $(\lambda x.M)N \rightarrow \mathsf{let}_S\ x = N\ \mathsf{in}\ M$ or $((\lambda x.M)N)_f \rightarrow \mathsf{let}_S\ x = N\ \mathsf{in}\ M$.
The first case is immediate from the definition of the encoding provided the remnant of the function process is garbage collection. The second case follows similarly as the $((\lambda x.M)N)_f$ occurs with a valid sharing regime.

2. $\mathsf{let}_E\ x = V_f\ \mathsf{in}\ E[x] \rightarrow \mathsf{let}_E\ x = V_f\ \mathsf{in}\ E[V_f]$ or $\mathsf{let}_D\ x = V_f\ \mathsf{in}\ E[x] \rightarrow \mathsf{let}_D\ x = V_f\ \mathsf{in}\ E[V_f]$
In both cases, x in the evaluation context will encode to $\overline{x}m$ for some channel m. This will communicate with the let binding which will send the location f to m which is the encoding of V_f.

3. $(\mathsf{let}_*\ x = M\ \mathsf{in}\ A)N \rightarrow \mathsf{let}_*\ x = M\ \mathsf{in}\ AN$

With a slight abuse of notation, the encoding of the left hand side will be of the form $[\![\mathsf{let}_*\ x = M]\!] | [\![A]\!]m|$
$m(f).(\overline{f}(a,o)|a(r).(\overline{a}r|[\![N]\!]n|n(g).!a(r).\overline{r}g))$. By inspection, this is the encoding of the right hand side.

4. $\mathsf{let}_E\ x = (\mathsf{let}_*\ y = M\ \mathsf{in}\ A)\ \mathsf{in}\ E[x] \rightarrow \mathsf{let}_*\ y = M\ \mathsf{in}\ \mathsf{let}_E\ x = A\ \mathsf{in}\ E[x]$
With a slight abuse of notation, the left hand side encodes to:
$[\![\mathsf{let}_*\ y = M]\!] | [\![A]\!]m|m(v).!x(r).\overline{r}(v)|[\![E[x]]\!]o$
which is also the encoding of the right hand side.

5. $\mathsf{let}_S\ x = M\ \mathsf{in}\ E[x] \rightarrow \mathsf{let}_E\ x = M\ \mathsf{in}\ E[x]$
The encoding of the left hand side reduces to the encoding of the right hand side.

6. $\mathsf{let}_E\ x = V\ \mathsf{in}\ E[x] \rightarrow \mathsf{let}_D\ x = V_f\ \mathsf{in}\ E[x] \quad (f \text{ fresh})$
The encoding of the left hand side reduces to the encoding of the right hand side, the f fresh following from the νf declaration.

In all the cases, the reduction of P has been determined leading to P' with $(M', P') \in \mathcal{S}$ for some M' and therefore P is r-determinate. □

Theorem 6.13 *For all closed $M' \in \mathcal{L}^0_*$, $[\![M']\!]o$ is r-determinate and one of the following conditions holds:*

A. $M' \downarrow N'$ and $[\![M']\!]o \downarrow [\![N']\!]o$

B. $M' \uparrow$ and $[\![M']\!]o \uparrow$.

Proof: by iterating lemma 6.12 starting from $[\![M]\!]o$. □

Corollary 6.14 *The encoding of λ-terms in Figure 3 implements call-by-need.*

Proof: From corollary 6.8, λ_{let_*} implements call-by-need and from lemma 6.13 the π-calculus encoding of $\mathcal{L}_*$ terms with value sharing regimes implements call-by-need. As the set of $\mathcal{L}_*$ terms are a superset of λ-terms, the encoding in Figure 3 implements call-by-need. Recall from corollary 6.8 that if $M \twoheadrightarrow N$ in n steps then the corresponding reduction in λ_{let_*} will be at most $3n$ steps long. The encoding of $\mathcal{L}_*$ terms does not introduce any new reduction rules but each reduction in λ_{let_*} will encode to a number of π-calculus steps. By inspection of the proof of Lemma 6.12 it can be seen that one reduction in λ_{let_*} takes at most two steps in the encoding. □

Syntactic Domains

$$\begin{array}{llcl} \text{Terms} & M, N \in \mathcal{L} & ::= & x \mid \lambda x.M \mid \lambda_V x.M \mid M@N \mid M@_V N \mid \mathsf{let}\ x = N\ \mathsf{in}\ M \\ \text{Evaluation Contexts:} & E & ::= & [\,] \mid EM \mid EE \mid \mathsf{let}\ x = M\ \mathsf{in}\ E \mid \mathsf{let}\ x = E\ \mathsf{in}\ E[x] \end{array}$$

Reduction rules:

$$\begin{array}{lll} (\mathsf{let}_S\text{-I}) & (\lambda x.M)@N & \to \mathsf{let}\ x = N\ \mathsf{in}\ M \\ & (\lambda x.M)@_V V & \to \mathsf{let}\ x = V\ \mathsf{in}\ M \\ & (\lambda_V x.M)@_* N & \to M@_V N \\ (\mathsf{let}_S\text{-V}) & \mathsf{let}\ x = V\ \mathsf{in}\ E[x] & \to \mathsf{let}\ x = V\ \mathsf{in}\ E[V] \\ (\mathsf{let}_S\text{-C}) & (\mathsf{let}\ x = M\ \mathsf{in}\ A)N & \to \mathsf{let}\ x = M\ \mathsf{in}\ AN \\ (\mathsf{let}_S\text{-A}) & \mathsf{let}\ x = (\mathsf{let}\ y = M\ \mathsf{in}\ A)\ \mathsf{in}\ E[x] & \to \mathsf{let}\ y = M\ \mathsf{in}\ \mathsf{let}\ x = A\ \mathsf{in}\ E[x] \end{array}$$

Figure 6: The extensions to λ_{let} to include strictness annotations

$$\begin{array}{rcl} [\![M@_V N]\!]o & \stackrel{\text{def}}{=} & (\nu a, m, n)([\![M]\!]m \mid [\![N]\!]n \mid m(f).n(g).(\overline{f}(a, o) \mid !a(r).\overline{r}g)) \\ [\![\lambda_V x.M]\!]o & \stackrel{\text{def}}{=} & (\nu n, f)(\overline{o}f \mid !f(a, p).(\overline{a}n \mid n(g).(!x(r).r(g) \mid [\![M]\!]p))) \end{array}$$

Figure 7: The extra encodings for strictness annotations

6.5 Actions and observation pre-congruences

Following Milner [13] we need to know how a process interacts with its environment. Milner defines an action relation for the π-calculus which shows the potential interactions a process can perform. This leads to a strengthening of determinacy and the following lemma:

Definition 6.15 P is *determinate* if it is r-determinate and whenever $P \twoheadrightarrow Q$ the only action Q can perform is communication.

Lemma 6.16 *For all $M \in \mathcal{L}$, $[\![M]\!]o$ is determinate.*

Proof: From the proof of Lemma 6.12. □

A notion of atomic observation of a π-term's behaviour can also be defined. Again following Milner, we would expect this to be similar to the precongruence on encoding of lazy λ-terms.

7 Extensions

While the encoding of a pure calculus is interesting from a theoretical standpoint, most functional languages extend this in various ways. Here, three such extensions are considered: combining evaluation strategies; cycles; and constants.

7.1 Mixing evaluations strategies

As the π-calculus is designed to represent concurrency it seems reasonable to extend λ_{let} calculus to include some parallelism and extend the encoding to accomplish this. The usual method of introducing parallelism to a lazy functional language is via strictness annotations, which are either inserted safely by the compiler performing strictness analysis or, potentially less safely, by the programmer. Following Hankin et al [8] we will annotate applications and λ's. The syntax of λ_{let} is then extended with an explicit application symbol @ so that it and λ are annotated with a subscript V when they are to be strict. The resulting system is presented in Figure 6 and the extra encodings are in Figure 7.

The extended version of λ_{let} implements parallel call-by-value such that a let_S-I step for an $@_V$ application only occurs when the right hand side has converged. It does this by allowing both sides of this application to be evaluated in parallel, which is reflected in the encoding. When both sides have converged, the body of the function is activated. The encoding of the call-by-value λ term is slightly more involved: upon receiving an activation it requests the evaluation of the supplied argument before entering the body.

The problem with implementing strictness in the manner shown above is that an application cannot be reduced until both sides have converged to a value. An alternative to this is to use *eager* or *lenient* reduction [18]. Here, both sides of an application are evaluated in parallel (as in parallel call-by-value) but the application can be reduced before the right hand side has converged to a value (as in call-by-name/need). This is achieved by setting up the argument computation as a separate process and passing a handle to this process in place of the argument. Requests for the argument value via the process handle must block until the value is available. The value is then memoized to achieve sharing. We like to refer to the parameter-passing mechanism as *call-by-process*. Representing it as $@_P$ its encoding is:

$$[\![M @_P N]\!]o \stackrel{\text{def}}{=} (\nu a, m, n)([\![M]\!]m \mid [\![N]\!]n \mid m(f).(\overline{f}(a, o) \mid n(g).!a(r).\overline{r}g))$$

This strategy can be accommodated within λ_{let} by enriching the evaluation environments to allow evaluation in a let binding. It should be noted that this style of evaluation is easily incorporated into λ_{let_*} as $(\lambda x.M)@_P N$ could reduce to $\mathsf{let}_E\ x = N\ \mathsf{in}\ M$. The evaluation environment $\mathsf{let}_E\ x = N\ \mathsf{in}\ E[x]$ is then generalized to $\mathsf{let}_E\ x = N\ \mathsf{in}\ M$.

7.2 Recursion

Efficient implementation of recursion in functional programming languages requires support for cycles. Ariola *et al.* [3] incorporate recursion within their system and similar enhancements can be made here. The encoding of a mutually recursive let construct is shown below:

$$[\![\mathsf{letrec}\ x_1 = M_1, \ldots x_n = M_n\ \mathsf{in}\ N]\!]o \stackrel{\text{def}}{=} (\nu\vec{x}\vec{m})\left(\begin{array}{l} (x_1(r).(\overline{x_1}r \mid [\![M_1]\!]m_1 \mid m_1(f).!x_1(s).\overline{s}f)) \mid \\ \ldots \\ (x_n(r).(\overline{x_n}r \mid [\![M_n]\!]m_n \mid m_n(f).!x_n(s).\overline{s}f)) \\ \mid [\![N]\!]o \end{array}\right)$$

The incorporation of recursion into λ_{let} substantially complicates the evaluation environments but it should be possible to extend the proofs presented here to that system.

Many run-time systems include a mechanism to recognize and report a *black hole* which is a detectable self-dependent infinite loop. The encoding given above has a similar property such that the encoding of a black hole causes deadlock, *i.e.*, it is bisimilar to the empty process $\mathbf{0}$.

Example 7.1

$$\begin{array}{ll} & [\![\mathsf{letrec}\ x = x\ \mathsf{in}\ x]\!]o \\ \stackrel{\text{def}}{=} & (\nu m, x)(x(r).(\overline{x}r \mid [\![x]\!]m \mid m(f).!x(s).\overline{s}f) \mid \overline{x}o) \\ \rightarrow & (\nu m, x)(\overline{x}o \mid [\![x]\!]m \mid m(f).!x(s).\overline{s}f) \\ \stackrel{\text{def}}{=} & (\nu m, x)(\overline{x}o \mid \overline{x}m \mid m(f).!x(s).\overline{s}f) \end{array}$$

As can be seen the need for the x causes the let expression to be reduced which results in a need for x again. However, as x has not converged there is no process to accept the request and therefore the process deadlocks.

To prove that the encoding does deadlock for all black holes would require their incorporation into λ_{let} and an extension to the existing correctness proof. We also need to prove that deadlock does not occur under any other circumstances.

7.3 Constants

Constants can be introduced into the encoding at a number of levels. Pre-defined functions can be thought of as let_D declarations at the outermost scope in the usual way. The processes that these binding create could either be direct encodings of λ-terms or direct implementations in π-calculus, as in the data structures examples given by Milner [12]. Alternatively if constants are added to the π-calculus they can be used in a natural way, as shown in Ostheimer and Davie [16].

8 Conclusions

In this paper it has been shown that an encoding of λ-terms in the π-calculus can implement a call-by-need strategy. This encoding is sufficiently simple to be used as a model for code generation for a parallel machine. Further this encoding can be extended to capture many of the requirements of a full implementation of functional programming languages.

9 Acknowledgments.

The authors would like to thank Tony Davie and John Glauert for numerous helpful discussions. Gerald Ostheimer was supported by SERC grant GR-H18739 'Compiler Technology for Scalable Distributed Multiprocessor Architectures'.

References

[1] M. Abadi, L. Cardelli, P. Curien, and J.J.Levy. Explicit substitutions. In *Proceedings of Seventeenth Annual ACM Symposium on the Principles of Programming Languages*, pages 31–46. ACM, ACM, 1990.

[2] S. Abramsky. The lazy λ-calculus. In D. Turner, editor, *Research Topics in Functional Programming*, pages 65–117. Addison-Wesley, 1990.

[3] Z. Ariola, M. Felleisen, J. Maraist, M. Odersky, and P. Wadler. A call-by-need λ-calculus. In *The Conference Record of Principles of Programming Languages*, 1995.

[4] G. Boudol. Some chemical abstract machines. In *A Decade of Concurrency — Reflections and Perspectives, Proceedings of the REX School Symposium*, volume 803 of *LNCS*, pages 92–123. Springer-Verlag, June 1993.

[5] M. Felleisen and D. Friedman. Control operators, the SECD-machine, and the λ-calculus. In *In* 3rd Working Conference on the Formal Description of Programming Concepts*, Ebberup, Denmark*, August 1986.

[6] J. Glauert. Asynchronous mobile processes. In *Proceedings of PARLE 92*, volume 605 of *LNCS*. Springer-Verlag, 1992.

[7] J. Glauert, L. Leth, and B. Thomsen. A new process model for functions. In M. Sleep, M. Plasmeijer, and M. van Eekelen, editors, *Term Graph Rewriting: Theory and Practice*, chapter 20, pages 269–282. Wiley, 1993.

[8] C. Hankin, G. Burn, and S. Peyton-Jones. A safe approach to parallel combinator reduction (extended abstract). In B. Robinet and R.Wilhelm, editors, *Proceedings of ESOP 86*, volume 213 of *LNCS*, pages 99–110. Springer-Verlag, 1986.

[9] K. Honda. Two bisimilarities in ν-calculus. Technical Report 92-002, Keio University, 1992.

[10] A. Jeffrey. Fully abstract semantics for concurrent graph reduction. In *Proceedings, Ninth Annual IEEE Symposium on Logic in Computer Science*, pages 82–91. IEEE, 1994.

[11] J. Launchbury. A natural semantics for lazy evaluation. In *Proceedings of the Conference on Principles of Programming Languages*, 1993.

[12] R. Milner. The polyadic π-calculus: a tutorial. Technical Report ECS-LFCS-91-180., LFCS, Computer Science Department, University of Edinburgh, 1991. Available from ftp.dcs.ed.ac.uk:pub/rm/ppi.ps.Z.

[13] R. Milner. Functions as processes. *Mathematical Structures in Computer Science*, 2:119–141, 1992.

[14] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, parts I and II. Technical report, LFCS, Computer Science Department, University of Edinburgh, 1989.

[15] G. Ostheimer. Parallel functional computation on STAR:DUST. In *Proceedings of the Workshop on the Parallel Implementation of Functional Languages*, 1991.

[16] G. Ostheimer and A. Davie. Modelling parallel graph reduction in the π-calculus. In *Proceedings of the Fifth International Workshop on Implementation of Functional Languages*, 1993.

[17] S. PeytonJones, C. Clack, J. Salkild, and M. Hardie. GRIP — a high performance architecture for parallel graph reduction. In *Proceedings of Functional Programming and Computer Architecture, Portland, Oregon*, volume 274 of *LNCS*, pages 98–112. Springer-Verlag, 1987.

[18] K. R. Traub. *Implementation of Non-Strict Functional Programming Languages.* MIT Press, 1991.

[19] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active messages: a mechanism for integrated communication and computation. In *Proceedings of 19th Annual International Symposium on Computer Architecture*, pages 256–266, 1992.

[20] C. Wadsworth. *Semantics and pragmatics of the λ-calculus.* PhD thesis, University of Oxford, 1971.

[21] N. Yoshida. Optimal reduction in weak lambda calculus with shared environments. In *Proceedings of the ACM Conferences on Functional Languages and Computer Architecture*, pages 243–252, 1993.