

Towards a formal semantics for paraML

Peter Bailey

Department of Computer Science
The Australian National University
Canberra ACT 0200 AUSTRALIA

Peter.Bailey@cs.anu.edu.au

Abstract

The original design and implementation of paraML, a parallel extension to Standard ML, was an organic exercise in language development, responding to the suggestions of users and addressing performance limitations. A complete redesign of the language extensions has taken place over the last year. In contrast to the previous experiment, the redesign has taken place simultaneously with the development of a formal semantics for the extensions. The implementation of the new language is in progress, and has been simplified by the semantic separation of core and derived extensions. The semantics define a λ calculus with rules for sequential and parallel evaluation. The core extensions of paraML are defined by the parallel evaluation rules, which describe how processes and communication ports are created, and how values are passed from processes to ports. Further work is required to develop an appropriate model of evaluation traces and properties of the configuration of processes, and to prove the type soundness of the calculus.

Keywords Formal semantics, concurrency, parallel languages, ML.

1 Introduction

The language paraML [2] is an extension of Standard ML [9], and has been designed for use with multicomputer systems. This experiment is the second iteration for the design and implementation of paraML. There was never a formal semantics developed for the original design and implementation of the language, which was constructed with a pragmatic eye to performance and addressing the desires of users. Unlike most other extensions to ML, in paraML each process evaluates against a unique environment. The language Facile [15] (a distributed extension of ML) is the closest in spirit to paraML, since it allows processes to

evaluate against both shared and individual environments, but it is targeted much more at distributed computation and must address such problems as node failure. This paper explores some steps in the development of a formal semantics for paraML. The development has taken place simultaneously with a redesign of the language, reflecting a more principled understanding of the requirements for parallel evaluation.

The formal semantics loosely follows the style developed by Reppy for CML [13, 14] (a concurrent extension of ML), which in turn is based primarily on the style developed by Wright and Felleisen [16]. This style of operational semantics is most useful for reasoning about entire programs, not about program fragments. As such, it assists in providing a close modelling of actual program execution and requirements on the compiler/runtime system for paraML. The development of a theory to enable equational reasoning about program fragments is more easily managed with a labelled transition system semantics, as Ferreira, Hennessy and Jeffrey [6] have done for CML. Their work also proves an equivalence between their labelled transition system semantics and Reppy's formalism. A similar approach should be possible for the semantics presented here.

The issue of formally establishing the type soundness of the extensions is not presented in this paper. Instead, the extensions are given a brief static semantic treatment in the notation of the initial basis of Standard ML in Section 2. The core of the work is presented in Section 3, which defines the sequential and parallel evaluation relations for λ_{pv} . This calculus models the essential properties of the extensions of paraML. Some future directions of the work are presented in Section 4.

2 Static semantics

The extensions that paraML makes to ML encompass seven operations for process creation and communication. Two new types for process names and port names are also required, together with some exceptions for indicating error conditions. The static semantics for these additions can be captured simplistically as extensions to the initial static basis of Standard ML as defined in [9]. Ultimately, it is

var	\mapsto	σ
$process$	\mapsto	$unit \rightarrow ProcessName$
$port$	\mapsto	$\forall 'a. ProcessName \rightarrow 'a PortName$
$execute$	\mapsto	$ProcessName \times (unit \rightarrow unit) \rightarrow unit$
$self_id$	\mapsto	$unit \rightarrow ProcessName$
$send$	\mapsto	$\forall 'a. 'a PortName \times 'a \rightarrow unit$
$recv$	\mapsto	$\forall 'a. 'a PortName \rightarrow 'a$
$probe$	\mapsto	$\forall 'a. 'a PortName \rightarrow bool$

Figure 1 - extensions to the initial value environment.

$tycon$	\mapsto	$\{\Theta, \{con_1 \mapsto \sigma_1, \dots, con_n \mapsto \sigma_n\} (n \geq 0)\}$
$process$	\mapsto	$\{ProcessName, \{process \mapsto unit \rightarrow ProcessName\}\}$
$port$	\mapsto	$\{PortName, \{port \mapsto \forall 'a. ProcessName \rightarrow 'a PortName\}\}$

Figure 2 - extensions to the initial type constructor environment.

desirable to formulate the issues involved in the static semantics uniformly with those of the dynamic operational semantics; this will involve defining a polymorphic type discipline for λ_{pv} and proving that it is sound.

2.1 New types

The two new types are `ProcessName` for the names of processes and `PortName` for the names of ports. The new definition of the set of initial type names T_0 is:

$$T_0 = \{bool, int, real, string, list, \\ ref, exn, instream, ostream, \\ ProcessName, PortName\}$$

2.2 New operations

The new operations can be specified as extensions to the initial value environment VE_0 with the nonfix definitions given in Figure 1. These operations perform the following actions:

1. **process** - create a new process, and return its name.
2. **port** - create a port on a process, and return its name.
3. **execute** - request a process to execute a thunk.
4. **self_id** - allow a process to determine its own name.
5. **send** - asynchronously send a message to a port.
6. **recv** - allow a process to perform a blocking receive from one of its ports.
7. **probe** - allow a process to query whether a `recv` operation would be successful or not.

2.3 New type constructors

The initial type constructor environment TE_0 is extended with the definitions given in Figure 2. These encode the `process` and `port` constructors, which are the only two operations which generate objects of new types in paraML. Note that the `PortName` type is an imperative one (otherwise referred to as weakly polymorphic - denoted by $'_a$), in order to provide a degree of polymorphism, whilst still retaining safe static typing of communications.

2.4 New exceptions

The domain of the initial exception environment EE_0 is the set of basic exception names (`BasExName`). `BasExName` is extended with the following exception constructors: `NoPortCreated`, `ProcessExecuting`, `NoSuchProcess`, `NoSuchPort`, `PortNotOwned`. These exceptions are used to indicate error conditions in the use of the extension operations, and are typically parameterised by the process name or port name used in the operation that generated the error.

3 Dynamic semantics

The following section characterises the dynamic semantics of paraML programs by introducing a parallel evaluation relation for a calculus called λ_{pv} . This work draws in part on the work of Reppy [13] in developing an operational semantics for CML. The section proceeds by providing some basic definitions in Section 3.1, followed by a brief overview of sequential evaluation in Section 3.2. The parallel evaluation relation is defined in Section

x	\in	VAR	variables
c	\in	CONST = BCONST \cup FCONST	constants
		BCONST = {(), true, false, 0, 1, ...}	base constants
		FCONST = {+, -, ...}	function constants
ex	\in	EXNNAME	exception names
π	\in	PROCESSNAME	process names
σ	\in	PORTNAME	port names
e	\in	EXP	expressions in the language
v	\in	VAL \subset EXP	values in the language

Figure 3 - Basic syntactic definitions.

e	$::=$	v	value
		$e_1 e_2$	application
		$(e_1.e_2)$	pair
		let $x = e_1$ in e_2	let
		exception x in e	exception
		raise $e_1 e_2$	raise exception
		e_1 handle $x e_2$	exception handle
		exn	exception packet
		process e	process creation
		port e	port creation
		execute e	execute request
		self_id e	own process name
		send e	send value to port
		recv e	receive value from port
		probe e	test ability to receive from port
exn	$::=$	$[ex, v]$	exception packet
v	$::=$	c	constant
		x	variable
		$(v_1.v_2)$	pair value
		$\lambda x(e)$	λ -abstraction
		ex	exception name
		π	process name
		σ	port name

Figure 4 - Grammar for expressions, exceptions and values.

3.3, and some remarks on the properties of configurations of processes in Section 3.4.

Reppy's work describes the syntax, sequential evaluation and concurrent evaluation of λ_{cv} , a concurrent extension of the λ_v calculus of Plotkin [12]. It also draws on the chemical abstract machine model developed by Berry and Boudol [5]. The λ_v calculus is simpler than ML in that

exceptions, references and continuations are not incorporated. Wright and Felleisen [16] present various extensions to the λ_v calculus, which progressively build up to encompass all of these additional language features in ML. Reppy uses a different notation for exceptions and shows that threads and channels can be used to implement references and recursion. The goal in this work is to define a parallel extension to λ_v , called λ_{pv} , which

$ \begin{aligned} E ::= & [] \mid E e \mid v E \mid (E.e) \mid (v.E) \mid \\ & \text{let } x = E \text{ in } e \mid \text{raise } E e \mid \text{raise } ex E \mid e \text{ handle } ex E \mid E \text{ handle } ex v \mid \\ & \text{process } E \mid \text{port } E \mid \text{execute } E \mid \text{self_id } E \mid \text{send } E \mid \text{recv } E \mid \text{probe } E \end{aligned} $

Figure 5 - Grammars for contexts.

$E[c v]$	\rightarrow	$E[\delta(c,v)]$	$(\lambda_{pv}\delta)$
$E[\lambda x(e) v]$	\rightarrow	$E[e[x \mapsto v]]$	$(\lambda_{pv}\beta)$
$E[\text{let } x=v \text{ in } e]$	\rightarrow	$E[e[x \mapsto v]]$	$(\lambda_{pv}\text{-let})$
$E[\text{raise } ex v]$	\rightarrow	$E[[ex, v]]$	$(\lambda_{pv}\text{-raise})$
$E[[ex, v] \text{ handle } ex v']$	\rightarrow	$E[v' v]$	$(\lambda_{pv}\text{-handle})$
$E[[ex_1, v] \text{ handle } ex_2 v']$	\rightarrow	$E[[ex_1, v]] \quad ex_1 \neq ex_2$	$(\lambda_{pv}\text{-reraise})$
$E[[ex, v] e]$	\rightarrow	$E[[ex, v]]$	λ_{pv} exception
$E[v [ex, v]]$	\rightarrow	$E[[ex, v]]$	propagation
$E[[ex, v].e]$	\rightarrow	$E[[ex, v]]$	rules
$E[(v'.ex, v)]$	\rightarrow	$E[[ex, v]]$	
$E[\text{let } x=[ex, v] \text{ in } e]$	\rightarrow	$E[[ex, v]]$	
$E[\text{raise } [ex, v] e]$	\rightarrow	$E[[ex, v]]$	
$E[\text{raise } ex [ex, v]]$	\rightarrow	$E[[ex, v]]$	
$E[e \text{ handle } ex' [ex, v]]$	\rightarrow	$E[[ex, v]]$	
$E[\text{process } [ex, v]]$	\rightarrow	$E[[ex, v]]$	
$E[\text{port } [ex, v]]$	\rightarrow	$E[[ex, v]]$	
$E[\text{execute } [ex, v]]$	\rightarrow	$E[[ex, v]]$	
$E[\text{self_id } [ex, v]]$	\rightarrow	$E[[ex, v]]$	
$E[\text{send } [ex, v]]$	\rightarrow	$E[[ex, v]]$	
$E[\text{recv } [ex, v]]$	\rightarrow	$E[[ex, v]]$	
$E[\text{probe } [ex, v]]$	\rightarrow	$E[[ex, v]]$	

Figure 6 - Sequential evaluation relation.

will be used to model the dynamic semantics of paraML. Eventually, this may also be used in proving the type soundness of paraML's extensions.

3.1 Definitions

The λ_{pv} calculus is split into sequential and parallel evaluation relations. The sequential evaluation relation defines what goes on inside a process, and the parallel evaluation relation defines how processes come into existence and how they exchange information. The sequential side of the language is in the standard applicative style, with constants, variables and call-by-value higher-order functions; this is given formal definition in the λ_v calculus. This has been extended to include pairs and parameterised exceptions to characterise the extensions faithfully; exceptions are defined as in Reppy [13]. Basic syntactic definitions are given in Figure 3.

The three syntactic classes of terms are expressions, values and exception packets. The

latter are irreducible terms, but are not values. Exception packets and exception names are terms that appear as intermediate forms, and hence do not appear in programs. Notice that the operations of paraML are defined as expressions, not function constants, since they will be given meaning by a parallel evaluation relation, not by a sequential evaluation relation like the other function constants. At present, references and continuations are not incorporated into the language; it is intended that they will be in future. The syntactic classes are all defined by the grammar given in Figure 4.

Note that the type system will guarantee that the arguments to the operations involving processes and ports are of the correct type. The free variables ($FV(e)$) and bound variables ($BV(e)$) of an expression are defined in the usual way, with **let** and λ binding variables with the restriction that **let** does not bind x in e_1 , only in e_2 . The standard techniques of Barandregt [4] are used to keep bound variables distinct from free variables in different expressions.

χ	\equiv	$\{ex_1, ex_2, \dots, ex_N\} \subset \text{EXNNAME}$	exception names
q	\equiv	$[v_1, v_2, \dots, v_N] \in \text{QUEUE}$	queues of values
(σ_N)	\equiv	$\{\sigma_1, \sigma_2, \dots, \sigma_N\}$	a set of port names
$Q(\sigma_N)$	\equiv	$\{\sigma_1 \mapsto q_1, \sigma_2 \mapsto q_2, \dots, \sigma_N \mapsto q_N\}$	a set of maps from port name to queue such that $\text{dom}(Q(\sigma_N)) = (\sigma_N)$
ε	\in	$\varepsilon \times E[e] \times [v]$	evaluation states
$p = \langle \pi; Q(\sigma_N); \varepsilon \rangle$	\in	PROC	process states
χ, P	\in	$\text{Fin}(\text{EXNNAME}), \text{Fin}(\text{PROC})$	process configurations
Terminal (P)	\subseteq	P , such that $\langle \pi; Q(\sigma_N); \varepsilon \rangle \in \text{Terminal}(P)$ iff $\varepsilon = [v]$	
Awaiting (P)	\subseteq	P , such that $\langle \pi; Q(\sigma_N); \varepsilon \rangle \in \text{Awaiting}(P)$ iff $\varepsilon = \varepsilon$	

Figure 7 - Syntactic definitions for exception names, queues, evaluation states, and process configurations.

The set VAL° is the set of closed value terms - those without any free variables.

3.2 Sequential evaluation

The sequential semantics of λ_{pv} are basically identical to those presented for λ_v by Reppy [13] and Wright and Felleisen [16]. We refer the reader to these works for more detailed information, and outline the main issues below. The semantics of the sequential function constants is given by the definition of the partial function δ , which applies function constants to closed values and produces a closed value:

$$\delta : \text{FCONST} \times \text{VAL}^\circ \rightarrow \text{VAL}^\circ \cup \{\text{exception } x \text{ in raise } x \mid v \in \text{VAL}^\circ\}$$

As mentioned before, evaluation of functions is call-by-value with a leftmost-outermost ordering. Terms within the language are partitioned into evaluation contexts and redexes. The grammar for the evaluation contexts E (expressions with one subexpression replaced by a hole, denoted $[]$) of λ_{pv} is given in Figure 5. The use of evaluation contexts prevents any possibility of capture of free variables, as the hole does not appear inside either of the two binding constructs. They also enforce the desired leftmost-outermost evaluation ordering. Filling the hole with a redex produces a new expression, $E[e]$.

The semantics of the sequential evaluation relation is the smallest relation, \rightarrow , satisfying the rules given in Figure 6. The λ_{pv} - β and λ_{pv} -**let** rules result in substitution for the bound variable in the redex expression. Raised exceptions are handled by their dynamically closest handler. The majority of rules are concerned with propagating exception packets through evaluation contexts to a surrounding handler, at which point either the λ_{pv} -**handle** or λ_{pv} -**reraise** rules will be applied.

3.3 Parallel evaluation

Reppy's concurrent evaluation relation is built around a chemical abstract machine model [5], except without any heating or cooling transitions.

The machine is embodied at any one time by a *configuration* which consists of finite sets of *process states*, exception names and channel names. The same general approach is taken for λ_{pv} - each process is tagged with a unique name; similarly all ports are tagged with a unique name. A process state is given as the 3-tuple consisting of the process name, the current state of any ports attached to the process, and the *evaluation state* of the process. Tagging each process state with a unique process name avoids having to use the multisets of a chemical abstract machine. As in Reppy's model, exceptions are bound in an implicit global environment that forms part of a process configuration. This technique guarantees that exception names are unique across all processes, despite their ability to escape their binding site by message transmission. The requirement of a global environment for exceptions is unfortunate as they have no bearing on parallelism; ideally only process names and port names would have constituted an implicit global environment. Process configurations thus consist of finite sets of exception names and process states.

Communication in CML is synchronous; in paraML, communication is asynchronous, and ports queue messages destined for a particular process. This requires some additional notation which is used to capture the state of communication for a port. The empty queue is given by $[]$; and the infix operation $@$ is used to indicate insertion into a queue. The normal semantics of queue operations are observed. The notation for port queues is given in Figure 7.

The evaluation state of a process can be in one of three different forms: awaiting execution to take place (denoted by ε); evaluating a term (the sequential evaluation contexts defined earlier in Section 3.2); or having completed evaluation to a basic value $[v]$. *Terminal* processes of a configuration are those which have evaluated to a basic value, and *awaiting* processes are those still awaiting an `execute` request. Implicit in the

$\frac{e \rightarrow e'}{\chi, P + \langle \pi; Q(\sigma_N); e \rangle \Rightarrow \chi, P + \langle \pi; Q(\sigma_N); e' \rangle}$	$(\lambda_{pv}\text{-seq})$
$\frac{ex \notin \chi}{\chi, P + \langle \pi; Q(\sigma_N); E[\text{exception } x \text{ in } e] \rangle \Rightarrow \chi + ex, P + \langle \pi; Q(\sigma_N); E[e[x \mapsto ex]] \rangle}$	$(\lambda_{pv}\text{-ex})$
$\frac{\pi' \notin \text{PROCN}(P) \cup \{\pi\}}{\chi, P + \langle \pi; Q(\sigma_N); E[\text{process } ()] \rangle \Rightarrow \chi, P + \langle \pi; Q(\sigma_N); E[\pi'] \rangle + \langle \pi'; \emptyset; \varepsilon \rangle}$	$(\lambda_{pv}\text{-process})$
$\frac{}{\chi, P + \langle \pi; Q(\sigma_N); E[\text{self_id } ()] \rangle \Rightarrow \chi, P + \langle \pi; Q(\sigma_N); E[\pi] \rangle}$	$(\lambda_{pv}\text{-self_id})$

Figure 8 - Parallel evaluation rules for: a) sequential evaluation; b) exception binding; c) **process**; d) **self_id**.

definition of possible evaluation states is that processes will only service at most one `execute` request. The definitions for these attributes of process configurations are given in Figure 7.

The empty set is given by \emptyset . The usual set union rule for the left associative operator $+$ is given, so that $P + p \equiv P \cup \{p\}$. Similarly with $Q(\sigma_N) + (\sigma_{N+1} \mapsto q_{N+1})$. The set of process names from the process states of a configuration is given as $\text{PROCN}(P)$, and the set of port names is given as $\text{PORTN}(P)$. The initial configuration of a paraML system is the singleton process set $\{\langle \pi_0; \emptyset; E[e] \rangle\}$, where no port names have been allocated. The initial process name is identified as π_0 to distinguish it from processes created by the `process` operation. This configuration is well-formed.

The symbol for the parallel evaluation relation is \Rightarrow . Evaluation proceeds by transitions between configurations, according to the parallel evaluation relation rules. For some of the rules, particularly those connected with the `execute`, `send` and `recv` operations, there are an additional two operations needed. These operations - `copy` and `uncopy` - are used to capture a value complete with any information needed to interpret it in a different evaluation context. (They are not new expressions as they are not part of the language, just a technique for describing value transmission.) Thus the `copy` operation takes a tuple argument consisting of an evaluation context and the value itself, and produces a new value. When these new values are supplied as argument to an `uncopy` operation (for example, as a result of a `recv` operation on a non-empty port queue), they are disinterred from their source evaluation context and placed in the hole of the current evaluation context with appropriate alterations. Since there are no free variables in a copied closed value, there are no variable capture problems. However, although references and continuations are not currently part of λ_{pv} , their inclusion in the future is expected, and will require

the use of `copy` and `uncopy` to adequately account for the semantics of their transmission.

The general form of the parallel evaluation rules is to show preconditions above a horizontal line. Below the line are two configurations connected by the parallel evaluation relation, corresponding to a transition between the two. One or two processes are selected on the left-hand side of the relation, and their process state is given in full. The remainder of the processes in the configuration are unaffected by the transition.

3.3.1 Sequential evaluation

The first rule in Figure 8.a states that sequential evaluation can be deduced in the presence of parallel evaluation. Note that if $e = E[e_1]$ and $e' = [v]$, then after the parallel evaluation rule has completed, $\pi \in \text{PROCN}(\text{Terminal}(P))$. The selected process in this transition is identified by π .

3.3.2 Exception binding

Creating a new exception name is given meaning in the parallel evaluation rules as the new exception name must be unique across the entire process configuration. The new name is added to the implicit global environment of exception names χ , and substitution of the new exception name for the identifier is performed, as given in Figure 8.b.

3.3.3 process

Process creation requires picking a new process name and creating a new process without any ports initially. The evaluation state of the new process is *awaiting* an execution request to be sent. The evaluation rule is given in Figure 8.c.

3.3.4 self_id

The `self_id` operation, given in Figure 8.d, is extremely straightforward, simply filling the evaluation context hole with the process identifier.

$\sigma_i \notin \text{PORTN}(\text{P}) \cup (\sigma_N)$	$(\lambda_{\text{pv-port1}})$
$\chi, \text{P} + \langle \pi; Q(\sigma_N); E[\text{port } \pi] \rangle \Rightarrow \chi, \text{P} + \langle \pi; Q(\sigma_N) + (\sigma_i \mapsto []); E[\sigma_i] \rangle$	
$\sigma'_i \notin \text{PORTN}(\text{P}) \cup (\sigma_N) \cup (\sigma'_{N'}) \quad es \neq [v]$	$(\lambda_{\text{pv-port2}})$
$\chi, \text{P} + \langle \pi; Q(\sigma_N); E[\text{port } \pi'] \rangle + \langle \pi'; Q(\sigma'_{N'}); es \rangle$ $\Rightarrow \chi, \text{P} + \langle \pi; Q(\sigma_N); E[\sigma'_i] \rangle + \langle \pi'; Q(\sigma'_{N'}) + (\sigma'_i \mapsto []); es \rangle$	
$\pi' \in \text{PROCN}(\text{Terminal}(\text{P}))$	$(\lambda_{\text{pv-port3}})$
$\chi, \text{P} + \langle \pi; Q(\sigma_N); E[\text{port } \pi'] \rangle$ $\Rightarrow \chi, \text{P} + \langle \pi; Q(\sigma_N); E[\text{raise NoPortCreated } \pi'] \rangle$	

Figure 9 - Parallel evaluation rules for **port**.

$\pi' \notin \text{PROCN}(\text{P}) \cup \{\pi\}$	$(\lambda_{\text{pv-exec1}})$
$\chi, \text{P} + \langle \pi; Q(\sigma_N); E[\text{execute } (\pi'.v)] \rangle + \langle \pi'; Q(\sigma'_{N'}); \varepsilon \rangle$ $\Rightarrow \chi, \text{P} + \langle \pi; Q(\sigma_N); E[()] \rangle + \langle \pi'; Q(\sigma'_{N'}); [(\text{uncopy } (\text{copy } (E, v))) \rangle]$	
$\pi' \notin \text{PROCN}(\text{P}) \cup \{\pi\}$	$(\lambda_{\text{pv-exec2}})$
$\chi, \text{P} + \langle \pi; Q(\sigma_N); E[\text{execute } (\pi'.v)] \rangle + \langle \pi'; Q(\sigma'_{N'}); E'[e] \rangle$ $\Rightarrow \chi, \text{P} + \langle \pi; Q(\sigma_N); E[\text{raise ProcessExecuting } \pi'] \rangle + \langle \pi'; Q(\sigma'_{N'}); E'[e] \rangle$	
$\pi' \in \text{PROCN}(\text{Terminal}(\text{P}))$	$(\lambda_{\text{pv-exec3}})$
$\chi, \text{P} + \langle \pi; Q(\sigma_N); E[\text{execute } (\pi'.v)] \rangle$ $\Rightarrow \chi, \text{P} + \langle \pi; Q(\sigma_N); E[\text{raise NoSuchProcess } \pi'] \rangle$	

Figure 10 - Parallel evaluation rules for **execute**.

3.3.5 port

Port creation requires picking a new port name, and a new mapping in the designated process from port name to queue. There are three basic situations: rule $(\lambda_{\text{pv-port1}})$ describes what happens when the designated process is the same as the port requester; rule $(\lambda_{\text{pv-port2}})$ describes what happens when the designated process is not the same as the port requester; and rule $(\lambda_{\text{pv-port3}})$ describes what happens when the designated process belongs to the terminal set of the configuration. Note that the use of a subscripted i in denoting the new port name does not mean it exists in the set of port names (σ_N) - the latter is a set, not a sequence. The three rules for the **port** operation are given in Figure 9.

3.3.6 execute

There are three rules also for the **execute** operation. These cover the following situations: rule

$(\lambda_{\text{pv-exec1}})$ is where an execution request is accepted by another process; rule $(\lambda_{\text{pv-exec2}})$ describes what happens if the designated process is already evaluating some expression; and rule $(\lambda_{\text{pv-exec3}})$ describes what happens if the process belongs to the terminal set of the configuration. The three rules for **execute** are given in Figure 10.

3.3.7 send

There are three different rules for sending: rule $(\lambda_{\text{pv-send1}})$ covers the situation where a message is sent to a port in the same process; rule $(\lambda_{\text{pv-send2}})$ details sending to a port on a remote process; and rule $(\lambda_{\text{pv-send3}})$ is the error situation when the port belongs to a process in the terminal set of the configuration. The use of **copy** in rule $(\lambda_{\text{pv-send1}})$ is important as it permits a uniform treatment of values in a port's queue by the **recv** operation. The three rules for the **send** operation are given in Figure 11.

$\sigma_i \notin \text{PORTN}(\text{P}) \cup (\sigma_N)$	$(\lambda_{\text{pv-send1}})$
$\chi, \text{P} + \langle \pi; Q(\sigma_N) + (\sigma_i \mapsto q_i); E[\text{send}(\sigma_i.v)] \rangle$	
$\Rightarrow \chi, \text{P} + \langle \pi; Q(\sigma_N) + (\sigma_i \mapsto q_i @ (\text{copy}(E, v))); E[()] \rangle$	
$\sigma'_i \notin \text{PORTN}(\text{P}) \cup (\sigma_N) \cup (\sigma'_{N'}) \quad es \neq [v]$	$(\lambda_{\text{pv-send2}})$
$\chi, \text{P} + \langle \pi; Q(\sigma_N); E[\text{send}(\sigma'_i.v)] \rangle + \langle \pi'; Q(\sigma'_{N'}) + (\sigma'_i \mapsto q'_i); es \rangle$	
$\Rightarrow \chi, \text{P} + \langle \pi; Q(\sigma_N); E[()] \rangle + \langle \pi'; Q(\sigma'_{N'}) + (\sigma'_i \mapsto q'_i @ (\text{copy}(E, v))); es \rangle$	
$\sigma_i \in \text{PORTN}(\text{Terminal}(\text{P}))$	$(\lambda_{\text{pv-send3}})$
$\chi, \text{P} + \langle \pi; Q(\sigma_N); E[\text{send}(\sigma_i.v)] \rangle$	
$\Rightarrow \chi, \text{P} + \langle \pi; Q(\sigma_N); E[\text{raise NoSuchPort } \sigma_i] \rangle$	

Figure 11 - Parallel evaluation rules for **send**.

$\sigma_i \notin \text{PORTN}(\text{P}) \cup (\sigma_N) \quad q_i \neq []$	$(\lambda_{\text{pv-recv1}})$
$\chi, \text{P} + \langle \pi; Q(\sigma_N) + (\sigma_i \mapsto [v_1, v_2, \dots, v_N]); E[\text{recv } \sigma_i] \rangle$	
$\Rightarrow \chi, \text{P} + \langle \pi; Q(\sigma_N) + (\sigma_i \mapsto [v_2, \dots, v_N]); E[\text{uncopy } v_1] \rangle$	
$\sigma_i \notin \text{PORTN}(\text{P}) \cup (\sigma_N) \cup (\sigma'_{N'}) \quad q_i = []$	$(\lambda_{\text{pv-recv2}})$
$\chi, \text{P} + \langle \pi; Q(\sigma_N) + (\sigma_i \mapsto []); E[\text{recv } \sigma_i] \rangle + \langle \pi'; Q(\sigma'_{N'}); E'[\text{send}(\sigma_i.v)] \rangle$	
$\Rightarrow \chi, \text{P} + \langle \pi; Q(\sigma_N) + (\sigma_i \mapsto []); E[\text{uncopy}(\text{copy}(E', v))] \rangle + \langle \pi'; Q(\sigma'_{N'}); E'[()] \rangle$	
$\sigma_i \notin (\sigma_N)$	$(\lambda_{\text{pv-recv3}})$
$\chi, \text{P} + \langle \pi; Q(\sigma_N); E[\text{recv } \sigma_i] \rangle$	
$\Rightarrow \chi, \text{P} + \langle \pi; Q(\sigma_N); E[\text{raise PortNotOwned } \sigma_i] \rangle$	

Figure 12 - Parallel evaluation rules for **recv**.

$\sigma_i \notin \text{PORTN}(\text{P}) \cup (\sigma_N) \quad q_i \neq []$	$(\lambda_{\text{pv-probe1}})$
$\chi, \text{P} + \langle \pi; Q(\sigma_N) + (\sigma_i \mapsto q_i); E[\text{probe } \sigma_i] \rangle$	
$\Rightarrow \chi, \text{P} + \langle \pi; Q(\sigma_N) + (\sigma_i \mapsto q_i); E[\text{true}] \rangle$	
$\sigma_i \notin \text{PORTN}(\text{P}) \cup (\sigma_N) \quad q_i = []$	$(\lambda_{\text{pv-probe2}})$
$\chi, \text{P} + \langle \pi; Q(\sigma_N) + (\sigma_i \mapsto q_i); E[\text{probe } \sigma_i] \rangle$	
$\Rightarrow \chi, \text{P} + \langle \pi; Q(\sigma_N) + (\sigma_i \mapsto q_i); E[\text{false}] \rangle$	
$\sigma_i \notin (\sigma_N)$	$(\lambda_{\text{pv-probe3}})$
$\chi, \text{P} + \langle \pi; Q(\sigma_N); E[\text{probe } \sigma_i] \rangle$	
$\Rightarrow \chi, \text{P} + \langle \pi; Q(\sigma_N); E[\text{raise PortNotOwned } \sigma_i] \rangle$	

Figure 13 - Parallel evaluation rules for **probe**.

3.3.8 recv

There are three rules for receiving messages. The first situation is when the named port has a non-empty queue, in which case the first message is dequeued, given in rule (λ_{pv} -**recv1**). Rule (λ_{pv} -**recv2**) is the most interesting rule yet, as it covers the case where a process is *blocked* awaiting input on a named port. Evaluation can only proceed if there is a matching send to the same port. (Rule (λ_{pv} -**recv2**) can be considered superfluous, in that the successive application of rules (λ_{pv} -**send2**) and (λ_{pv} -**recv1**) is equivalent to the actions taking place, but is included for completeness with respect to the state of a port's queue.) Rule (λ_{pv} -**recv3**) covers the error condition, where the port is owned by another process (or was before it terminated). The three rules for the `recv` operation are given in Figure 12.

3.3.9 probe

The rules for the operation `probe` look very similar to the rules for receiving, except that a simple boolean truth value is returned. Error conditions are again detected in rule (λ_{pv} -**probe3**). The three rules are given in Figure 13.

3.4 Properties of configurations

It is useful to consider some properties about configurations and evaluations. A paraML evaluation proceeds by progressively transforming the process configuration, commencing with the initial well-formed configuration of $\{\langle \pi_0; \emptyset; E[e] \rangle\}$, where $E[e]$ constitutes the program. Note that by examination of the definitions for parallel evaluation rules, if χ, P is well-formed and $\chi, P \Rightarrow \chi', P'$ then the following also hold:

1. χ', P' is well-formed
2. $\chi \subseteq \chi'$
3. $\text{PROCN}(P) \subseteq \text{PROCN}(P')$
4. $\text{PORTN}(P) \subseteq \text{PORTN}(P')$

Processes have been characterised according to their evaluation states as either *awaiting* (prior to execution), *terminal* (after evaluation to a closed value), or evaluating some term. If a process is currently unable to proceed with evaluation (typically, it is seeking to apply rule (λ_{pv} -**recv2**), but there is no matching process prepared to send a value to its desired port), then it is termed *blocked*. The set of *ready* or *enabled* processes (denoted $\text{Ready}(P)$) consists of all processes that are not *awaiting*, *terminal*, or *blocked*. The following definitions about a process configuration then hold.

1. If $\text{PROCN}(P) = \text{PROCN}(\text{Terminal}(P) + \text{Awaiting}(P))$, then the configuration χ, P is *complete*.
2. If $\text{Ready}(P) = \emptyset$ and the configuration is not complete, then the configuration χ, P is *deadlocked*.
3. Otherwise, the computation is running.

A trace is defined as a sequence (possibly infinite) of configurations $[\chi_1, P_1, \chi_2, P_2, \dots, \chi_N, P_N]$

such that $\chi_1, P_1 \Rightarrow \chi_2, P_2 \Rightarrow \dots \Rightarrow \chi_N, P_N$. One program may have many different possible traces for its evaluation. A trace T is a *computation* if it is infinite or if it is finite and the final configuration is complete. However this definition permits traces that are unfair - those in which some processes fail to make progress despite the fact that they are enabled. A computation T is *acceptable* if it ends in a configuration that is complete, or if T satisfies *strong process fairness* constraints as defined in Kwiatkowska's survey of fairness issues [7]. This definition requires that any process that is enabled infinitely often is selected infinitely often. The actions that can be taken by an enabled process are defined by the parallel evaluation rules and the selected processes are those that are the subject of one of the transitions of these rules. In practice, with the implementation of this language it is fairly straightforward to guarantee fairness properties due to execution on a parallel computer. Also, the properties of ports as queues of values under asynchronous sending semantics helps to ensure that processes once enabled can make progress.

4 Future work

At this point in time, not enough consideration has been given to the meaning of a paraML program, since many processes may compute to produce basic values. Also there is no account taken of the side-effecting nature of true parallel programs which may alter state in the form of output files or streams. The proposal of Mitchell [10] for identifying a main thread of computation, from which other processes are created, is an intuitively attractive approach. In his work, the overall effect of the computation is the canonical value produced by the main thread and computation is deemed to be complete when this occurs. A similar characterisation is expected to be formally integrated with this work in the near future.

The formal definitions of `copy` and `uncopy` are yet to be complete. Earlier work explored a characterisation of the operations in a fragment of ML, but this was not wholly satisfactory. The treatment of exceptions and references proved to be particularly problematic. It is clear that the λ_v -variants eliminate the problems associated with transmission of closed values and will provide a more rigorous and appropriate abstraction for the transmission of references and continuations between evaluation contexts.

Lastly, it would be intriguing to attempt a translation of the paraML extensions into the PICT language [11], an implementation of the π calculus [8]. Such a translation would be valuable for the insights into how a mobile process calculus differs in explaining parallel behaviour from a λ calculus extension. Work by Amadio [1] on the semantics of Facile has provided some interesting leads to explore in this respect. As explored in our earlier work [3] we consider the use of a fully concurrent language such as PICT as the "sequential" language within processes to be

essential for both implementation and semantic aspects of parallel language development. The integration of parallel and concurrent semantics would pose some challenging problems.

5 Conclusion

This paper has outlined the current directions towards formalising the semantic basis of the language paraML. The separation in the semantics of typing issues from runtime evaluation follows that expounded for Standard ML. The type rules for the language are captured in a slightly informal manner as extensions to the initial static basis of Standard ML. In contrast, the dynamic semantics of paraML has been described through the definition of a λ calculus extension called λ_{pv} . The rules for reduction of terms in this calculus are captured by sequential and parallel evaluation relations. The rules for the parallel evaluation relation capture the basic operational semantics of how processes and communication ports are created and how values are passed from processes to ports within programs in the paraML language. The work is significant as paraML does not rely on shared environments for the execution of processes, making it particularly appropriate for characterising such types of languages on the high performance multicomputer systems found today.

Acknowledgments

I would particularly like to record my appreciation for the support of my supervisors: Malcolm Newey, Robin Stanton, and Chris Johnson. Conversations with Benjamin Pierce, Thierry le Sergent and Andy Gordon helped considerably in encouraging my interests in a redesign and formalisation of the language.

References

- [1] R. M. Amadio. Translating Core Facile. ECRC, ECRC-1994-3, 1994.
- [2] P. Bailey and M. Newey. An Extension of ML for Distributed Memory Multicomputers. In *Proceedings of the Sixteenth Australian Computer Science Conference*, pp. 387-396, Brisbane, Australia, 1993.
- [3] P. Bailey, M. Newey, D. Sitsky and R. Stanton. Supporting Coarse and Fine Grain Parallelism in an Extension of ML. In *Proceedings of the CONPAR'94 - VAPP VI*, pp. 593-604, Linz, Austria, 1994.
- [4] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. 1984.
- [5] G. Berry and G. Boudol. The chemical abstract machine. *Theoretical Computer Science* 96, (1992) pp. 217-248.
- [6] W. Ferreira, M. Hennessy and A. Jeffrey. A Theory of Weak Bisimulation for Core CML. School of Cognitive and Computing Sciences, University of Sussex, 05/95, 1995.
- [7] M. Z. Kwiatkowska. Survey of fairness notions. *Information and Software Technology* 31, 7 (1989) pp. 371-386.
- [8] R. Milner, J. Parrow and D. Walker. A Calculus of Mobile Processes I & II. *Information and Computation* 100, 1 (1992) pp. 1-77.
- [9] R. Milner, M. Tofte and R. Harper. *The Definition of Standard ML*. MIT Press, Cambridge, Massachusetts, 1990.
- [10] K. Mitchell. Concurrency in a Natural Semantics. LFCS, University of Edinburgh, ECS-LFCS-94-311, 1994.
- [11] B. C. Pierce and D. N. Turner. PICT user manual. *available electronically* <http://www.cl.cam.ac.uk/users/bcp1000/1994>.
- [12] G. D. Plotkin. Call-by-name, Call-by-value and the λ -Calculus. *Theoretical Computer Science* 1, (1975) pp. 125-159.
- [13] J. H. Reppy. An Operational Semantics of First-class Synchronous Operations. Cornell University, TR 91-1232, 1991.
- [14] J. H. Reppy. Higher-Order Concurrency. PhD dissertation, University of Cornell University, 1992.
- [15] B. Thomsen, L. Leth, S. Prasad, T.-M. Kuo, A. Kramer, F. Knabe and A. Giacalone. Facile Antigua Release Programming Guide. ECRC, ECRC-93-20, 1993.
- [16] A. K. Wright and M. Felleisen. A Syntactic Approach to Type Soundness. Dept. Computer Science, Rice University, TR91-160, 1991.