On the Execution of Ambients

Matthew Sackman[1,2]

Department of Computing
Imperial College
London
England

Susan Eisenbach[3]

Department of Computing
Imperial College
London
England

Abstract

Successfully harnessing multi-threaded programming has recently received renewed attention. The GHz war of the last years has been replaced with a parallelism war, each manufacturer seeking to produce CPUs supporting a greater number of threads in parallel execution.
The Ambient calculus offers a simple yet powerful means to model communication, distributed computation and mobility. However, given its first class support for concurrency, we sought to investigate the utility of the Ambient calculus for practical programming purposes.
Although too low-level to be considered as a general-purpose programming language itself, the Ambient calculus is nevertheless a suitable virtual machine for the execution of mobile and distributed higher-level languages. We present the Glint Virtual Machine: an interpreter for the Safe Boxed Ambient calculus. The GlintVM provides an effective platform for mobile, distributed and parallel computation and should ease some of the difficulties of writing compilers for languages that can exploit the new thread-parallel architectures.

Keywords: Concurrency, Ambients, Parallelism

1 Introduction

Whilst process algebra are well studied and understood as a means of specifying interaction between concurrent processes, there have been few implementations. Even fewer implementations of Mobile Ambient calculi [8,5,17] are available [7,23,12]. The Mobile Ambient calculi are particularly interesting given the recent explosion of mobile devices and our insatiable desire to be able to access all of our data all of the time from anywhere. Being able to describe movement and encapsulation of processes offers a more suitable model for such mobile environments than the

[1] We would like to thank the SLURP team at Imperial College for their review and input to this paper.
[2] Email: ms@doc.ic.ac.uk
[3] Email: sue@doc.ic.ac.uk

π-calculus [18] or similar process algebra by catering for the modelling of limited range communication and of movement, in sympathy with real-world constraints.

Although conceived as a modelling tool, the Mobile Ambient calculus can be considered an assembly language, offering first-class support for concurrency, movement and communication. As such they make an attractive target language for higher-level languages which have similar goals of concurrency and mobility as the gap between source language and target language becomes much smaller for these features. Additionally, it becomes infeasible to execute by hand large programs in any process algebra.

We implemented one of the many variants of the Mobile Ambient calculus the Safe Boxed Ambient calculus [17]. The Safe Boxed Ambient calculus has several desirable features from both a theoretical and implementation perspective. It does not contain the `open` x instruction, hence the *boxed*. This means that Ambient boundaries cannot be dissolved, simplifying both implementation and type-systems. Indeed, the type-systems presented in [17] statically assert several desirable properties such as type-safety of communication and mobility, hence the *safe*. Our implementation, called the GlintVM, interprets the Ambient calculus program and spawns a thread for each process created by the program. As such, it is able to make good use of multi-threaded hardware. We have not implemented the type-systems for the Safe Boxed Ambient calculus.

In order to test the suitability of the Safe Boxed Ambient calculus as an assembly language or byte-code, we developed an Actor-based language which compiles to the Safe Boxed Ambient calculus for execution on the GlintVM. This is a useful way to test the utility of the GlintVM platform. This language and the compiler are covered in appendix A.

The rest of this paper is organised as follows: Section 2 briefly introduces the Safe Boxed Ambient calculus and demonstrates by example why it is a suitable language for expressing interactions between concurrent processes. Section 3 examines the implementation of the GlintVM in Java and explains the difficulties of implementing the Safe Boxed Ambient calculus. Section 4 evaluates the implementation, comparing the GlintVM with related work and other tools in the area. Finally, section 5 concludes.

2 Worked Example

In Ambient calculi, Ambients are named and contain processes. The processes consist of instructions which can perform communication with other processes or movement of the Ambient itself. The instructions can cause further Ambients to be created, processes to be spawned in parallel or replicated, all within the containing Ambient. Ambients can be nested and when an Ambient moves all of the processes within it move too: thus the code and data of the program are mobile rather than just the data. Finally, the range of communication is limited. A communication instruction either targets a named child Ambient or the parent of the current Ambient, or allows itself to be paired with any matching communication within the current Ambient, immediate parent Ambient or immediate child Ambient.

The syntax of the Safe Boxed Ambient calculus is shown in figure 1 whilst the

Fig. 1. The syntax of the Safe Boxed Ambient calculus

Names: $n, m, \ldots, x, y, \ldots \in \mathbf{N}$

Locations:

$\eta ::=$	n	names
	$\mid \uparrow$	parent Ambient
	$\mid *$	local

Processes:

$P ::=$	0	nil process
	$\mid (\, P_1 \mid P_2 \,)$	parallel composition
	$\mid (\nu\, n)(\, P \,)$	restriction
	$\mid\, !P$	replication
	$\mid V[\, P \,]$	Ambient
	$\mid V \,.\, P$	prefixing
	$\mid (\overrightarrow{x})^{\eta} \,.\, P$	input
	$\mid \langle \overrightarrow{V} \rangle^{\eta} \,.\, P$	output

Values:

$V, U ::=$	n		name
	$\mid \mathsf{in}\ V$		enter into V
	$\mid \mathsf{out}\ V$		exit into V
	$\mid \overline{\mathsf{in}}\ \alpha$	$\alpha \in \{n, *\}$	allow entry of n or of all
	$\mid \overline{\mathsf{out}}\ \alpha$	$\alpha \in \{n, *\}$	allow exit of n or of all
	$\mid V_1 \,.\, V_2$		path

reduction rules are shown in figure 2; see [17] for a full discussion of the operational semantics. Communication is synchronous as per [17] and polyadic, an extension we have made. As usual, communications without any explicit location annotation default to local $(*)$ communications.

Consider accessing a distributed database where the data is distributed across several hosts and one central server is able to direct queries to the relevant host. The process that wishes to query the dataset should ask the central server which host to move to before moving and interrogating the data held by that host. Assume *lookupHost* is a function that finds the correct host for the given query and binds the *host* variable to that value. The central server could have a structure as follows:

$$!(\, querier \,) \,.\, lookupHost \,.\, \langle host \rangle \,.\, 0$$

Thus there is a replicated process which, through communication, receives some details of the query to be performed, invokes the *lookupHost* function and finally outputs the *host* value. However, given the use of replication, many of these processes could exist in parallel. As such, the central server must be sure that the result gets back to the correct *querier*:

$$central[\, !(\, querier \,) \,.\, \overline{\mathsf{in}}\, querier \,.\, lookupHost \,.\, \langle host \rangle^{querier} \,.\, 0 \,]$$

Now the *querier* is allowed into the *central* Ambient and so as a child of that Ambient can be targeted by communication. Note that in moving into the *central*

Fig. 2. Reduction in the Safe Boxed Ambient calculus

Mobility:

$$
\begin{array}{lll}
n[\,\mathsf{in}\,m\,.\,P \mid Q\,] \mid m[\,\overline{\mathsf{in}}\,\alpha\,.\,R \mid S\,] & & \\
\quad\rightarrow m[\,n[\,P \mid Q\,] \mid R \mid S\,] & \text{for } \alpha \in \{*, n\} & \text{RED IN} \\
m[\,n[\,\mathsf{out}\,m\,.\,P \mid Q\,] \mid R\,] \mid \overline{\mathsf{out}}\,\alpha\,.\,S & & \\
\quad\rightarrow n[\,P \mid Q\,] \mid m[\,R\,] \mid S & \text{for } \alpha \in \{*, n\} & \text{RED OUT}
\end{array}
$$

Communication:

$$
\begin{array}{rcll}
(\overrightarrow{x})\,.\,P \mid \langle\overrightarrow{V}\rangle\,.\,Q & \rightarrow & P\{\overrightarrow{V}/\overrightarrow{x}\} \mid Q & \text{RED COMM LOCAL} \\
(\overrightarrow{x})^{n}\,.\,P \mid n[\,\langle\overrightarrow{V}\rangle\,.\,Q \mid R\,] & \rightarrow & P\{\overrightarrow{V}/\overrightarrow{x}\} \mid n[\,Q \mid R\,] & \text{RED COMM INPUT } n \\
(\overrightarrow{x})\,.\,P \mid n[\,\langle\overrightarrow{V}\rangle^{\uparrow}\,.\,Q \mid R\,] & \rightarrow & P\{\overrightarrow{V}/\overrightarrow{x}\} \mid n[\,Q \mid R\,] & \text{RED COMM OUTPUT } \uparrow \\
\langle\overrightarrow{V}\rangle^{n}\,.\,P \mid n[\,(\overrightarrow{x})\,.\,Q \mid R\,] & \rightarrow & P \mid n[\,Q\{\overrightarrow{V}/\overrightarrow{x}\} \mid R\,] & \text{RED COMM OUTPUT } n \\
\langle\overrightarrow{V}\rangle\,.\,P \mid n[\,(\overrightarrow{x})^{\uparrow}\,.\,Q \mid R\,] & \rightarrow & P \mid n[\,Q\{\overrightarrow{V}/\overrightarrow{x}\} \mid R\,] & \text{RED COMM INPUT } \uparrow \\
 & & & \text{all where } |\overrightarrow{x}| = |\overrightarrow{V}|
\end{array}
$$

Congruence:

$$
P \equiv Q \qquad Q \rightarrow R \qquad R \equiv S \qquad \text{implies} \qquad P \rightarrow S \qquad \text{RED STRUCT}
$$

Ambient, the code of the *querier* is mobile. But we must consider how to get the name of the *querier* Ambient in an output, to communicate with the input in the *central* Ambient. Communication cannot occur between processes in sibling Ambients which means either the *central* Ambient must be a child of the *querier* or the *querier* must be a child of the *central* Ambient. Given the architecture of the existing program, the *central* Ambient should not move, thus we choose the latter:

$$
\mathit{central}[\,!\overline{\mathsf{in}}\ *\,.\,(\,\mathit{querier}\,)\,.\,\mathit{lookupHost}\,.\,\langle \mathit{host}\rangle^{\mathit{querier}}\,.\,0\,]
$$

Now the *central* Ambient allows any Ambient in, then receives the name of the *querier* Ambient (pairing with an output within the *querier* that targets the parent, *central*, Ambient), before computing the host containing the data requested and sending that host name to the *querier* Ambient. However, there are still some remaining problems. Firstly there is no reason why you could not have more than one Ambient within the *central* Ambient with the same name. Consequently, the results of the *lookupHost* function could get misrouted. Secondly, having entered the *central* Ambient, the *querier* must be allowed out, thus an $\overline{\mathsf{out}}$ *querier* process must exist outside of the *central* Ambient. The final program is shown in figure 3 along with a possible *querier* process which interacts with the *central* Ambient.

Three top-level processes are shown, and are discussed in the order they appear. The first top-level process is replicated. It receives any Ambient name and allows that Ambient out of some sibling Ambient. Note that this is used twice, once by the process inside the *myQuery* Ambient, and once by the *central* Ambient's process

Fig. 3. The final program for the initial database host query

$$
\begin{aligned}
&!(\,letMeOut\,)\,.\,\overline{\mathsf{out}}\,letMeOut\,.\,0\\
&\mid central[\,!\overline{\mathsf{in}}\;*\,.\,(\,querier\,)\,.\,lookupHost\,.\,\langle host\rangle^{querier}\,.\,\langle querier\rangle^{\uparrow}\,.\,0\,]\\
&\mid myQuery[\,(\nu\, querier)(\\
&\qquad \langle querier\rangle^{\uparrow}\,.\,querier[\,\mathsf{out}\,myQuery\;.\;\mathsf{in}\,central\;.\;\langle querier\rangle^{\uparrow}\;.\\
&\qquad\qquad (\,targetHost\,)\;.\;\mathsf{out}\,central\;.\;\mathsf{in}\,myQuery\;.\;\langle targetHost\rangle\;.\;0\,]\\
&\qquad \mid \overline{\mathsf{in}}\,querier\;.\;(\,targetHost\,)^{querier}\;.\;P\,)\,]
\end{aligned}
$$

as discussed above. Other than for this addition, the second top-level process, the *central* Ambient, is unchanged.

The third top-level process is the *myQuery* Ambient. This works by creating a unique name for the *querier*, thus avoiding any future name clashes. It then arranges for the *querier* to be allowed out and then constructs the *querier* Ambient. This moves out of *myQuery* and into the *central* Ambient. When it comes back it moves back into *myQuery* and then communicates its findings to the process within *myQuery* which will then go on to perform the rest of the query. Given two Ambients with the same *myQuery* name, the $\overline{\mathsf{in}}$ *querier* instruction in each *myQuery*'s process names the *querier* and as each *querier* has a unique name, the *querier* is guaranteed to return to its originating Ambient.

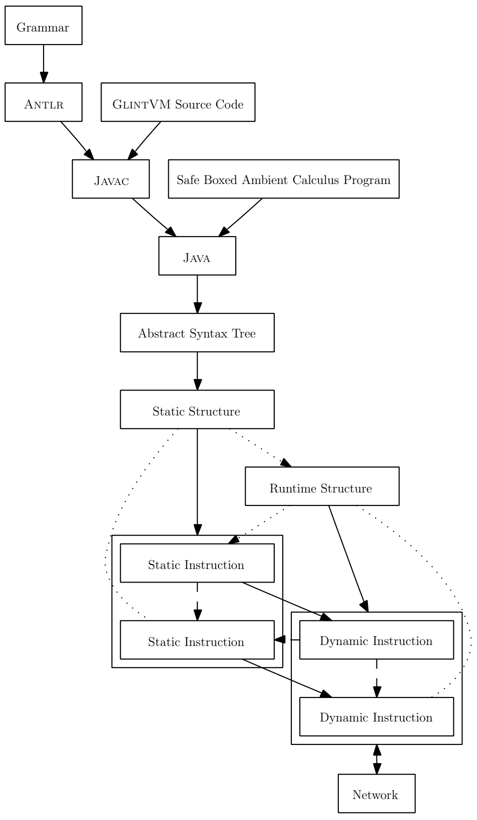
Even with such a simple example, making the program behaviour correct in light of concurrent queries requires some care. It should also be clear that the program is already getting to a size which is unpleasant to reduce by hand. As such, an interpreter becomes a requirement for development of any reasonably sized programs.

3 Implementation

Programs are written in plain ASCII files using a simple translation of the syntax shown in appendix B. The basic design of the GlintVM is as follows: having parsed the input program, the AST formed is traversed. The AST consists of objects representing the static form of each instruction. On interpretation, the current instruction is translated to a dynamic representation of the same instruction which is then executed. Once complete, the next static instruction is similarly translated and executed. In this way, the static instructions can be reused, e.g. as part of a replicated process, containing no state related to the execution of the instruction and are immutable after construction. The dynamic representation exists to keep the state necessary for the execution of the instruction and is never reused or shared. This design is shown in figure 4, where the static structure indicates the ordering of instructions and the runtime structure indicates the relationship between the dynamic instructions and their association with the static instructions.

Parallel composition, restriction, Ambient creation and replication are all forking instructions, potentially containing multiple processes in parallel. When executed, additional threads may be spawned. All other instructions require a pairing with another process. Movement instructions require pairing an action ($\mathsf{in}\ x$, $\mathsf{out}\ x$) with

Fig. 4. The overall design of the GLINTVM



a co-action ($\overline{\mathsf{in}}\ x$, $\overline{\mathsf{out}}\ x$) whilst communication instructions require the pairing of an output and an input. The execution of a variable can only require pairings as a variable can only be bound to an instruction via input and the values which can be communicated can only consist of movement or communication or concatenation of movement or communication instructions.

The rest of this section is organised as follows: Section 3.1 explains how the model of Ambients causes issues when mapped onto physical hosts: the inability of physical hosts to be nested within other hosts creates problems for the movement of Ambients between hosts. Section 3.2 discusses infinite replication in Ambient calculi and how we implemented the desired lazy replication. Section 3.3 explains how we achieve pairings between processes safely, for communication and movement. Finally, section 3.4 discusses how inter-host movement of processes is achieved through serialisation.

3.1 Hosts and Ambients

Ambients can be moved into and out of other Ambients. For this to work there must either be a top-most *root* Ambient that contains all other Ambients or some top level environment in which other processes and Ambients can be executed. This, unfortunately, does not map naturally onto physical hosts and no suitable mappings are presented with the calculi.

The simplest mapping from the calculi to physical hosts is that each host is an immoveable Ambient within which other Ambients and processes can exist. Thus an Ambient could either represent hardware or software. However, by being immoveable, for inner Ambients to move between hosts they must move outside of the *host* Ambient before entering the target *host* Ambient. This presents a problem as, having moved outside of the originating *host* Ambient, the mobile Ambient is no longer in any host and thus it is not clear on which machine any further instructions should be processed; in particular, the next movement instruction which should take the mobile Ambient into the target *host* Ambient.

To solve this problem, we have augmented the semantics to deal specifically with inter-host movement. An Ambient which is a direct child of the *host* Ambient can specifically target another *host* Ambient for inter-host movement. For example:

$$\mathsf{host}\colon \mathsf{\mathit{rose}}[\, a[\, \mathsf{in\ host}\colon \mathsf{\mathit{holly}} \,.\, P \,]] \mid \mathsf{host}\colon \mathsf{\mathit{holly}}[\, \overline{\mathsf{in}}\ \mathsf{host}\colon \mathsf{\mathit{rose}} \,{:}\, a \,.\, Q \,]$$

Here, there are two Ambients representing hosts called *rose* and *holly* and one non-host Ambient, a. With these additional semantics, this reduces to:

$$\mathsf{host}\colon \mathsf{\mathit{rose}}[] \mid \mathsf{host}\colon \mathsf{\mathit{holly}}[\, Q \mid a[\, P \,]]$$

Thus for inter-host movement, the moving Ambient directly targets the destination host and does not first move out of the *host* Ambient it is within. Such a movement instruction can only be executed if the process trying to execute the instruction is within an Ambient that is a direct child of the *host* Ambient. In all other cases, the inter-host movement instruction will block. Thus the reduction rule for inter-host movement is:

$$\begin{array}{l} \mathsf{host}\colon \mathsf{\mathit{x}}[\, n[\, \mathsf{in\ host}\colon \mathsf{\mathit{y}} \,.\, P \mid Q \,] \mid R \,] \mid \mathsf{host}\colon \mathsf{\mathit{y}}[\, \overline{\mathsf{in}}\ \mathsf{host}\colon \mathsf{\mathit{x}} \,{:}\alpha \ .\ S \mid T \,] \\ \quad \to \mathsf{host}\colon \mathsf{\mathit{x}}[\, R \,] \mid \mathsf{host}\colon \mathsf{\mathit{y}}[\, S \mid T \mid n[\, P \mid Q \,]] \quad \text{for } \alpha \in \{*, n\} \quad \text{Red In Host} \end{array}$$

Note that only the $\mathsf{in\ host}\colon \mathsf{\mathit{x}}$ and $\overline{\mathsf{in}}\ \mathsf{host}\colon \mathsf{\mathit{x}}{:}y$ actions are needed: the out and $\overline{\mathsf{out}}$ instructions have no inter-host variation. Also note that the $\mathsf{in\ host}\colon$ instruction indicates only a host to move to whilst the co-action, $\overline{\mathsf{in}}\ \mathsf{host}\colon$, indicates both a source host and an Ambient name. The source host must be specified, but the Ambient name can be $*$ to indicate accepting any Ambient from the host. Finally, $\mathsf{host}\colon \mathsf{\mathit{x}}$ and $\mathsf{host}\colon \mathsf{\mathit{x}}{:}y$ are considered values and thus can be communicated between processes.

This is similar to the separation between physical and logical location in [23]. There, the authors eliminate nesting of Ambients within the same host. Their terminology corresponds *node* with a *host* Ambient and *agent* loosely with a direct child of a *host* Ambient. All *agents* are directly within the same *node* and the *node* is responsible for the creation of new *agents.* Our model allows for Ambients to create new child Ambients without any interaction with the *host* Ambient. Their *immobile* Ambients are the same as our *host* Ambients.

3.2 Replication

A process of the form $!P$ represents infinite copies of the process P in parallel: $(P \mid P \mid \ldots)$. Of course, with the finite resources of a computer, it is unwise to try and create infinite copies of the replicated process. Instead the copies should be made lazily, each copy being produced only when the previous copy has performed some sort of reduction.

A process P can be incapable of reducing by itself, for example $(x) . 0$. Such a process must only consist of processes that require pairings to reduce that the other processes within P cannot provide *in their current location*. If such a process, P, is placed in parallel with itself, $P \mid P$, then the parallel composition can only reduce if the composition allows for pairings between processes from each copy of P. $n[\, \mathsf{in}\, n . R \mid \overline{\mathsf{in}}\, n . Q \,]$ is such process: on its own it cannot reduce, but in parallel with itself it will reduce to $n[\, \mathsf{in}\, n . R \mid Q \mid n[\, R \mid \overline{\mathsf{in}}\, n . Q \,]]$. Thus the replication of the process must start with more than a single copy of P, but how many is enough? As reductions are achieved through pairings, two copies are sufficient in order to be able to provide both parts of the pairing. Therefore, we must consider the differences between an eager unrolling of $!P$ and a lazy unrolling starting from $P \mid P$ where the first reduction of any copy of P causes a new fresh copy of P to be created.

If $P \mid P$ cannot reduce then neither can an infinite parallel composition of P. To see this, consider that reduction is achieved through pairing and that by definition pairing requires two processes. Every process within P occurs twice within $P \mid P$ thus if no pairing can occur in $P \mid P$ then no pairings can occur in an infinite parallel composition of P because adding additional copies of P does not create any configurations of processes that are not present in $P \mid P$. From this we can see that a lazy unrolling does not prevent divergence that is possible with an eager unrolling: because the eager unrolling of $!P$ does not create any configurations that are not present in $P \mid P$, any reduction that can occur in an eager unrolling can also occur in a lazy unrolling, starting from $P \mid P$.

Consequently, the lazy unrolling of $!P$ cannot prevent reductions from occurring: if $P|P$ cannot reduce internally then neither can an eager unrolling of $!P$; if P or $P|P$ can reduce internally, then a lazy unrolling of $!P$ does not preclude any reduction that could occur with an eager unrolling of $!P$. Our implementation initially unrolls $!P$ to $P \mid P \mid !P$ and then only when P reduces does $!P$ next unroll, thus reducing to $P' \mid P \mid P \mid !P$ or $P' \mid P'' \mid P \mid P \mid !P$ depending on the reduction that has occurred. However, detecting the reduction can be tricky. Consider the following process:

$$b[\, (x) . \mathsf{in}\, x . 0 \,] \mid !(\nu\, a)(a[\, \overline{\mathsf{in}}\, * . 0 \,] \mid \langle a \rangle^b . 0)$$

Whilst the $(\nu\, a)(P)$ instruction can always be reduced to P,[4] it should not trigger a new copy of the replicated process to be created. In this case, it is only the reduction of $\langle a \rangle^b . 0$ that should cause the copy of the replicated process to be created. This is achieved by the following means: whenever a static instruction generates a dynamic representation there is an observer pattern that receives notification. Thus the replication can observe whenever any of the instructions within

[4] Ambient calculi do not reduce restriction. Our implementation executes restriction by generating a unique name and binding it to the variable indicated.

the process that is replicated are made into their dynamic equivalents. Also, all instructions know their *container*. This may be an Ambient, a parallel composition, a replication or a restriction. Whenever a dynamic instruction is generated, the replication updates a mapping from the instruction's container to the container's container. This will recursively lead back to the replication. At this point the replication subscribes to further observers within the dynamic instruction to be notified at the completion of execution of the dynamic instruction.

This means that whenever an instruction within the replicated process is executed, the replication will be informed. It will again find the container of the instruction. It queries the mapping with this container and if it finds a value will recurse and query the mapping with the value found. For each value found in the map, it removes the key from the map. Only if the final value it finds is the replication itself does it know that the instruction is the first instruction to be executed and so the replicated process should be duplicated.

As soon the first instruction executed is found, all subscriptions to the observer patterns in the now-reduced process are cancelled. The mapping is necessary because the first instruction completed could be a movement instruction which would mean that finding the parent of the container of the instruction may well now not be within the replication, for example the process $b[\,\overline{\mathsf{in}}\ a\,.\,0\,]\mid !a[\,\mathsf{in}\ b\,.\,0\,]$: after the execution of the $\mathsf{in}\ b$ instruction, the container of the a Ambient is the b Ambient rather than the replication. Note that a container is not considered to be complete until all of the processes within it have reduced to 0. Therefore the process $(\nu\ x)(P \mid Q \mid R)$ will not complete until P and Q and R are all reduced to 0 or have otherwise moved outside of the restriction. Further, the observer patterns are combined with visitor patterns which allow the replication to choose carefully which reductions it is informed of. For example, it does not care about the completion of the null (0) process. This means that $!0$ will not spin and $!a[\,0 \mid P\,]$ will ignore the trivial completion of 0.

3.3 Local Movement and Communication

One of the features of the Ambient calculi is that in addition to being able to communicate data between processes, Ambients and the processes they contain (hence, code and data) can be moved.

Communication and movement both involve pairing instructions between processes. In the case of movement, the obvious location for the pairing to take place is in the parent Ambient of the Ambients that are pairing. For example, in the process:

$$a[\ b[\,\overline{\mathsf{in}}\,c\,.\,0\,]\mid(\,c[\,\mathsf{in}\,b\,.\,\mathsf{out}\,b\,.\,0\,]\mid\overline{\mathsf{out}}\,c\,.\,0\,)\ ]$$

All the pairings between the actions and co-actions take place in the a Ambient. In the case of the $\mathsf{out}\ b$ instruction, this means that the process must find the grandparent of its Ambient. For communication the pairing always takes place in the Ambient containing the *un-targeted* communication. The *un-targeted* communication is either of (a) or $\langle a\rangle$, whereas communications that indicate the parent or a child ambient are *targeted*. A *targeted* communication can only reduce with an *un-targeted* communication.

$$\langle x \rangle^{a} \,.\, 0 \mid a[\, (m) \,.\, P \mid \langle y \rangle \,.\, 0 \mid b[\, \langle z \rangle^{\uparrow} \,.\, 0 \,]\,]$$

All the outputs in the above process will use the a Ambient to seek pairing with the input. Substituting outputs for inputs and vice versa, the same occurs: the a Ambient is the site for the pairings to be established. Multiple pairings can be sought at the same time and when this involves movement there is the possibility that movement occurs whilst another pairing is being sought which invalidates the pairing. Therefore the algorithm used must ensure that at the time the pairing is made the two processes are in a valid relationship with each other for the pairing to take place. If the pairing is invalidated then it must be guaranteed that both processes know the pairing is invalid, that the pairing is discarded and that both processes begin searching for a pairing afresh.

$$a[\, \mathsf{in}\; b \,.\, 0 \mid \mathsf{in}\; c \,.\, 0 \,] \mid b[\, \overline{\mathsf{in}}\; a \,.\, 0 \,] \mid c[\, \overline{\mathsf{in}}\; a \,.\, 0 \,]$$

In the above process, either the process within the b Ambient reduces or the process within the c Ambient reduces but not both. Similarly of the two processes within the a Ambient and of course the correct process must reduce given the movement that occurs.

$$a[\, \mathsf{in}\; b \,.\, 0 \mid \mathsf{in}\; c \,.\, 0 \mid (m) \,.\, 0 \,] \mid b[\, \overline{\mathsf{in}}\; a \,.\, 0 \mid \langle x \rangle^{a} \,.\, 0 \,] \mid c[\, \overline{\mathsf{in}}\; a \,.\, 0 \mid \langle y \rangle^{a} \,.\, 0 \,] \mid \langle z \rangle^{a} \,.\, 0$$

The pairings must ensure that only one of the possible movement pairings occur. For the communication, again only one of the pairings can occur. It is perfectly valid for the communication involving the output $\langle z \rangle^{a}$ to complete after the movement has occurred on the grounds that the pairing of the output and input was established and confirmed to be valid before the movement took place, but only completed after the movement. What must be guaranteed is that if the output $\langle z \rangle^{a}$ reduces then none of the other outputs can reduce and that if the output $\langle z \rangle^{a}$ does not reduce then the only output that can reduce is the output within the Ambient into which the a Ambient moves.

The GlintVM uses a finite state machine algorithm to achieve safe pairings. This has been proven to be deadlock free by modelling the algorithm using the Ltsa tool [16]. The key component is to create the pairing within a cell where neither party can determine the other until the cell is full and *sealed*. If the pairing is invalidated then it must be impossible for either party to determine the other, forcing both parties to seek a fresh pairing. Thus both parties must populate the cell, must seal the cell only if there is no error and if there is an error, must indicate that the cell is invalid. Only once the cell is sealed by *both* parties can each uncover with whom they are pairing. Both parties then obtain locks preventing movement of the Ambients they are contained within and any other relevant Ambients before performing final checks that the pairing is still valid and then actually performing the movement. See [21] for a fuller discussion of this algorithm.

3.4 Inter-host Movement

Inter-host movement is the movement of Ambients between hosts. Our implementation needs to distinguish between intra-host and inter-host types of movement where the Ambient calculi do not. Inter-host movement of Ambients amounts to being able to move running code between hosts which is a very attractive feature for migrating applications between mobile devices. This is strong mobility [6] as it is the code and the entire execution state of the process, which is transported between hosts.

Given that the instructions, both static and dynamic, are represented by objects, Java's serialisation of objects is the simplest way to transport instructions between hosts. Before transportation however, it must be guaranteed that all processes within the Ambient that is to be transported have stopped and accurately recorded their state such that after transportation to the target host, no instructions will be executed more than once and no instructions will be skipped.

Having achieved a pairing with the target host, the inter-host movement instruction first takes the movement lock recursively on the Ambient that is to move. This prevents any other Ambients from entering or leaving the moving Ambient. Without this step, Ambients could escape and possibly be duplicated, or could move into the moving Ambient and possibly be left behind. Every process within the moving Ambient is then told to *freeze*. This amounts to either completing the current instruction being executed or to abandoning it. All instructions that are blocked waiting for a pairing will be awoken at this point, will notice the *freeze* request and will save the state of the current instruction as if it had never started. Having *frozen* all the processes within the moving Ambient, the threads that were executing those processes die, leaving just the thread executing the inter-host movement instruction within the Ambient alive.

Now the serialisation can occur, safe in the knowledge that nothing can interfere with the *frozen* Ambient. The serialisation must ensure that only the Ambient and its processes and their state are serialised. This requires some care given the way in which Java serialisation works. Consider, for example:

$$\mathsf{host}\colon \textsf{\textit{rose}}[\,\overline{\mathsf{in}}\ \mathsf{host}\colon \textsf{\textit{holly}}\,{:}a\,.\,0\,] \mid \mathsf{host}\colon \textsf{\textit{holly}}[\,!\overline{\mathsf{out}}\ b\,.\,0 \mid a[\,\mathsf{in}\ \mathsf{host}\colon \textsf{\textit{rose}}\ .\,0 \mid !b[\,\mathsf{out}\ a\,.\,0\,]]]$$

Given the way replication is implemented (see section 3.2), there will be pointers from the right-most replication to the instructions within the b Ambients, potentially until just after they have moved out of the a Ambient. It is vital that these pointers are not followed when serialising the replication otherwise not only will the a Ambient be transported but so will some, if not all of the b Ambients that have moved outside of the a Ambient. Careful consideration of the state of instructions, where the *freezing* mechanism can `null` out field values along with use of the `transient` field modifier is sufficient to achieve this.

Having been successfully transported to the target *host* Ambient, the moved Ambient must be *thawed*. This is very similar to simply starting up the Ambient initially only the dynamic instructions need not be generated from the static instructions as they already exist.

4 Evaluation and Related Work

The GlintVM is a distributed implementation of the Safe Boxed Ambient calculus. We have testing the GlintVM with programs of several thousand instructions running in a distributed setting across up to eight nodes. There are some limitations in our implementation. Firstly the replication of processes is implemented via a lazy unrolling in order to prevent exhaustion of resources. However, the implementation does not attempt to analyse whether or not a reduction that is achieved is entirely internal to the replicated process. Consequently, replicated processes that reduce internally, for example $!((v) \,.\, 0 \mid \langle x \rangle \,.\, 0)$ will endlessly execute.

However, even detecting internal reductions would not be sufficient as it is possible to construct a process which has replicated internal processes where each of the internal processes cannot reduce internally, but reduces by pairing externally to the replicated process. Externally, the process is unobservable. E.g.:

$$(\nu\, x)((!\langle a \rangle^{x} \,.\, 0) \mid (!x[\,(v) \,.\, 0\,]))$$

This leads on to a more fundamental problem, the inability to detect and garbage collect stuck processes. A stuck process is a process that cannot reduce any further but is blocking, waiting for some sort of pairing that will never occur, for example $(\nu\, x)(\langle a \rangle^{x} \,.\, P)$. A process which is unobservable may be reducing, but will never perform an action or communication which an external process can pair with. Such processes should also be considered for garbage collection.

The inability to detect and remove these processes causes resource leaks often both in terms of memory and threads. However, detecting stuck processes in general is very challenging, especially in a distributed setting. It requires tracking communications and detecting when restricted names become unknown. This is very hard to achieve without some sort of global registry which would create significant problems itself.

Finally, the decision to use one thread per process is limiting. The overhead of using operating-system level threads is high and inter-thread communication such as via `wait()` and `notify()` in Java require a kernel-trap in order to update the state of the threads. However, the benefit is the implementation is simplified and can readily take advantage of thread-parallel architectures. The obvious alternative would be to use just one thread and maintain processes that can be reduced in data structures, switching between the available processes. This would not be able to take advantage of thread-parallel hardware at all. The happy medium is to have the same number of threads as can be executed by the hardware in parallel and then for those threads to choose from data structures representing the available reducible processes. This however would be a more complex design.

Whilst we know of no other implementations of the Safe Boxed Ambient calculus, there are a few implementations of Ambient calculi. In [7], Cardelli implements the Ambient calculus in Java. However, the implementation is not distributed, limited to reducing Ambients on a single machine. The threading model used is the same as ours where each Ambient process is executed by a separate thread.

Rather than treating an Ambient calculus as a target language for a compiler, the Channel Ambient System [20] instead adds functionality directly to the Channel

Ambient calculus in order to produce a more expressive and powerful language. This is implemented formally in an abstract reducing machine and then practically in OCaml. The Channel Ambient System has a number of highly useful primitives in the calculus such as communication between processes in sibling Ambients. In practice, this turns out to be very convenient and simplifies a lot of applications.

The approach used in [23] is similar to that of [20], in which the Safe Ambient calculus [15] is implemented. Their implementation is similarly formally treated with an abstract reducing machine and they then sketch their Java-based implementation. Their implementation uses a thread per Ambient. The thread is responsible for executing, on a round-robin basis, the local processes within the Ambient. This avoids the need to spawn potentially thousands of threads for large applications, but for smaller applications risks being unable to make best use of thread-parallel hardware. The Safe Ambient calculus avoids what the authors call "grave interferences" by the use of co-actions, similar to the Safe Boxed Ambient calculus and a type system. This prevents, for example, multiple processes within an Ambient attempting to move the ambient to different destinations concurrently. In our experience, avoiding interferences such as these is important though the use of restriction tends to result in there never being competing movement instructions that can both reduce.

Finally, [12] implements the Mobile Ambient calculus [8] via translation to JoCaml [9], a variant of OCaml extended with the Join calculus [11]. This is a highly concurrent implementation which makes good use of the concurrent features of the Join calculus and thus JoCaml. This is particularly interesting given the close relationship between the Join calculus and Actors, examined in [10].

5 Conclusions and Future Work

We have presented the GlintVM, an interpreter for the Safe Boxed Ambient calculus. The implementation makes few departures from the calculus, the only significant change is the separation of inter-host movement from intra-host movement due to the mapping between Ambients and physical hosts.

Using the Ambient calculus as an assembly language is limiting for general purpose languages, given the focus of the Ambient calculus on mobility and concurrency. However, we believe it is a foundation upon which additional primitives can be added, closing the gap with higher-level languages. In order to safely harness the increasingly thread-parallel hardware being produced, paradigms without shared mutable state must be considered. Erlang[4] is one such language and is loosely based on the Actor paradigm. Ambients and process algebra in general sit in this domain, as state is encapsulated within processes and thus can only be modified by the process itself.

We hope to address the limitations of our implementation of replication so that we can detect that a reduction has occurred entirely within the replicated process, thus delaying the unrolling of the replication. We are considering ways of tracking the range of restricted values so as to be able to reason about unobservable and stuck processes. We also are considering the type systems presented with the Safe Boxed Ambient calculus and how they can be implemented in an extensible way.

Finally, we believe that adding additional primitives to the calculus would allow for a simpler and more efficient encoding of our Actor-based toy language into the Safe Boxed Ambient calculus.

References

[1] *Antlr*, http://www.antlr.org/.

[2] *Haskell*, http://www.haskell.org/onlinereport/.

[3] Agha, G., "Actors: A Model of Concurrent Computation in Distributed Systems," MIT Press, Cambridge, MA, USA, 1986.

[4] Armstrong, J., *The development of erlang*, in: *ICFP '97: Proceedings of the second ACM SIGPLAN international conference on Functional programming* (1997), pp. 196–203.

[5] Bugliesi, M., G. Castagna and S. Crafa, *Boxed ambients*, , **2215 of Lecture Notes In Computer Science** (2001), pp. 38–63.

[6] Cabri, G., L. Leonardi and F. Zambonelli, *Weak and strong mobility in mobile agent applications*, in: *Proceedings of the 2nd International Conference and Exhibition on The Practical Application of Java (PA JAVA 2000)*, Manchester (UK), 2000.

[7] Cardelli, L., *Mobile ambient synchronization*, Technical Report SRC Tech Note 1997-013" (1997).

[8] Cardelli, L. and A. D. Gordon, *Mobile ambients*, Theoretical Computer Science (TCS) **240** (2000), pp. 177–213.

[9] Fessant, F. L., *The jocaml system prototype* (1998), http://join.inria.fr/jocaml.

[10] Fournet, C. and G. Gonthier, *The join calculus: A language for distributed mobile programming*, in: *Applied Semantics, International Summer School, APPSEM 2000, Caminha, Portugal, September 9-15, 2000, Advanced Lectures* (2002), pp. 268–332.

[11] Fournet, C., G. Gonthier, J.-J. Lévy, L. Maranget and D. Rémy, *A calculus of mobile agents*, in: *CONCUR '96: Proceedings of the 7th International Conference on Concurrency Theory* (1996), pp. 406–421.

[12] Fournet, C., J.-J. Levy and A. Schmitt, *An asynchronous, distributed implementation of mobile ambients*, in: *IFIP TCS*, 2000, pp. 348–364.

[13] Greif, I. and C. Hewitt, *Actor semantics of planner-73*, in: *POPL '75: Proceedings of the 2nd ACM SIGACT-SIGPLAN symposium on Principles of programming languages* (1975), pp. 67–77.

[14] Hewitt, C., P. Bishop and R. Steiger, *A universal modular actor formalism for artificial intelligence*, in: *IJCAI*, Stanford, California, 1973, pp. 235–245.

[15] Levi, F. and D. Sangiorgi, *Controlling interference in ambients*, in: *Proceedings of POPL'00* (2000), pp. 352–364.

[16] Magee, J. and J. Kramer, "Concurrency: state models & Java programs," John Wiley & Sons, Inc., New York, NY, USA, 1999.

[17] Merro, M. and V. Sassone, *Typing and subtyping mobility in boxed ambients*, , **2421 of Lecture Notes In Computer Science** (2002), pp. 304–320.

[18] Milner, R., "Communicating and Mobile Systems: the π-calculus," CUP, 1999.

[19] Odersky, M., P. Altherr, V. Cremet, B. Emir, S. Maneth, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman and M. Zenger, *An overview of the scala programming language*, Technical report, EPFL (2004).

[20] Phillips, A. N. J. B., "Specifying and Implementing Secure Mobile Applications in the Channel Ambient System," Ph.D. thesis, Imperial College, London (2005).

[21] Sackman, M., "Glint: Breeding Mobile Ambients with Actors," Master's thesis, Imperial College (2006), http://wellquite.org/non-blog/glint/glintReport.pdf.

[22] Sackman, M., *Glint software* (2006), http://wellquite.org/index.php/glint/.

[23] Sangiorgi, D. and A. Valente, *A distributed abstract machine for safe ambients*, in: *ICALP '01: Proceedings of the 28th International Colloquium on Automata, Languages and Programming,* (2001), pp. 408–420.

A Employing the GlintVM

To test the GlintVM as a byte-code interpreter for a higher level language, we designed a toy language with first-class concurrency and mobility primitives. Our preference was for a language in which only one model must be maintained mentally whilst programming, joining concurrency and data abstractions.[5] The Actor paradigm [14,3,13] is suitable for our purposes as threads of execution are trapped within Actors (which encapsulate state similarly to Objects) and can be extended quite naturally to include mobility.

A simple model of Actors is as follows. An Actor is created by combining a new thread with a *behaviour*. The *behaviour* specifies how to act on messages that are sent to the Actor. The actions performed upon the receipt of a message can include creating new Actors, sending messages or specifying a new *behaviour*. In this way the *behaviour* of an Actor can change over time upon receipt of messages. Messages that are sent to an Actor accumulate in the Actor's mailbox. Messages are not required to be processed in the order they arrive in the mailbox and it is valid to send a message to an Actor that will never be processed. A *behaviour* only specifies, typically by some form of pattern matching, which messages it understands. For a thorough discussion of Actors, see [3].

Our toy language distinguishes between two types of Actor definitions: definitions that can be combined many times with threads: instantiable Actors; and definitions that can only ever be combined with one thread per host: singleton Actors. Instantiable Actor definitions are analogous to Classes in an Object Oriented language in that they have to be constructed explicitly before use. Singleton Actor definitions are analogous to `static` methods in a Class in Java or to singleton types in Scala [19], in that they have a name which is statically known and are not constructed prior to use. In our toy language, we have developed a small library of Actors. In this library, the Actor definition representing the number 1 is implemented as a singleton Actor, whereas the Actor definition for a countdown latch is an instantiable Actor.

An Actor definition of either type can specify a method body which is to be executed upon receipt of any message, or, one or more methods. These methods contain bodies which will be executed only upon receipt of a message containing the name of the method. In this way, the toy languages coerces method calls to message sending: calling a method on an Actor is translated to sending a message to that Actor with the first value in the message being the name of the method called. Method calls are synchronous as this allows them to return a result. Whilst such semantics prevent the need to use continuation passing style they also destroy concurrency, so there are mechanisms within the language to allow an immediate reply to a synchronous message before the body has been executed. For a full discussion of the relationship between asynchronous communication and continuation passing, see [21].

The body consists of several components. An optional become statement speci-

[5] The Object Oriented model suffers in this respect, for having completely separate graphs for objects and for concurrency, leading to multi-threaded programs in which the concurrency is difficult to comprehend as it is disjoint from the syntax of the language. Adding mobility to the Object Oriented paradigm would make programs yet more obscure.

Fig. A.1. An Instantiable Counter Actor

```
define Counter (Number value)
  inc ()
  inc
    become new Counter<value.plus<One>>

  dec ()
  dec
    become new Counter<value.minus<One>>
```

fies a replacement *behaviour*. If this does not exist then the *behaviour* of the Actor (i.e. the definition which governs how the Actor responds to messages) does not change. The body can specify a return statement. If this is not specified then the method call will complete immediately and the remaining parts of the body will then execute concurrently with the subsequent statements of the caller. Finally, a where clause can be specified. This is very similar to a *Haskell*-style [2] where clause in that assignments can be specified in any order and it is up to the compiler to determine the correct order of execution based on the dependencies between the assignments.

Figure A.1 shows the code for a counter. In order to create an Actor based on this definition, the initial value must be specified. This is indicated by the (Number value) on the first line. In keeping with the Ambient calculi, parenthesis are used to represent input whilst angle-brackets represent output. Two methods are defined in Counter, inc and dec. Neither method requires any parameters and both specify only a become statement. The become statements in both methods are creating new Counter instances which will replace the existing Counter instance. The new instance will have a different initial value, achieved through the plus or minus method calls. Thus the state of the instance is updated through replacing the instance.

A.1 Movement

Whilst distributed Actors are considered in [3], mobile Actors are not and are an addition we have made to the Actor model. We permit an Actor to move between hosts when it specifies a replacement *behaviour*. This is the only movement permitted: Actor instances cannot be nested as Ambients can be, but all exist within the same top-level namespace. More complex nestings of Actors are an interesting idea and are subject to future work. When moving to a host, the names of Actors that the moving Actor knows of will be invalid on the target host as Actors with those names will not exist on the target host. There is no automatic forwarding of method calls back to the originating host. However, given a standard library which makes use of singleton Actors and is loaded onto both hosts at startup, the moving Actor will be able to access the singleton Actors on the target host.

Whilst there is no forwarding from the target host back to the originating host, there is forwarding to the target host. Other Actors on the originating host may not know that the Actor has moved to a new host, and given the synchronous method calls, it would be disastrous if method calls to the moved Actor block indefinitely. Therefore a forwarding system allows method calls to the moved Actor

Fig. A.2. A Mobile Ball

```
define Ball (Number x, Number y)
 draw ()
 draw x'.greaterThan<Zero> and x'.lessThan<Ten>
    become new Ball<x',y>
    where
       any = Screen.clear<>.drawCircleAt<x',y>
 draw x'.equals<Zero>
    ...deal with bouncing ...
 draw
    become new Ball<Zero,y> in host:otherHost
    where
       otherHost = Screen.getHostAt<x',y>
 where
    x' = x.plus<One>
```

to be forwarded to the destination host and for results of the method call to be returned. This works transitively, so if an Actor makes multiple movements between hosts then messages sent to that Actor within the very first host will still reach it and results will still be returned correctly.

Figure A.2 shows a definition for a Ball that moves between hosts. The intuition is that each host has one Screen (defined as a singleton). If the Ball moves off the side of the Screen then the Screen tells the Ball which host to move to in order to continue on moving in that direction. Only the key parts of the Ball definition are shown: the full definition is substantially longer. This example also shows how a method body can have multiple cases, all but the last being guarded by a predicate. The predicates are tested in the order they are defined and the body guarded by the first succeeding predicate is the only body executed.

A.2 Toy Language Compiler

The compiler is implemented in two halves. The first half uses ANTLR's [1] very powerful AST walkers to progressively refine the AST, translating parts of the language so that they are expressed in terms of other, simpler language primitives. Having translated the input program to a subset of the toy language's primitives, the code is then translated, using a set of templates of Safe Boxed Ambient code.

In writing the compiler for our toy language it became very clear how difficult it is to implement what may seem to be quite simple semantics of Actors in the Safe Boxed Ambient calculi. Similarly, relatively minor differences in behaviour between Actors and the calculi resulted in considerable complexity in the encoding. As a consequence, the code generated by the compiler typically expands a few tens of lines of toy language code to several hundreds of instructions of Safe Boxed Ambient calculus. The program shown in figure A.1 compiles to 502 instructions of Safe Boxed Ambient code and, along with other examples, is available from the website accompanying this paper [22].

Expanding eight lines of code to 502 instructions is an untypically large expansion for a compiler: JAVA for example has a much lower rate of expansion, a method call in Java will be expanded to just a few Java byte-code instructions dependent on the number of arguements, whereas a method call in our toy language gets expanded to around 20 Safe Boxed Ambient calculus instructions. But the Java byte-code has a far more complete set of general-purpose operations, for example branches, numeric operations and object creation which are not catered for by the Safe Boxed Ambient calculus. Consequently, we have had to encode these types of operations within the Safe Boxed Ambient calculus which results in a complex and large encoding.

In light of how challenging it is to write strongly mobile applications in Java and how obscure and complex the resulting programs are, we feel that the size and complexity of the encoding is a minor issue. Being able to write applications in languages with first-class support for mobility and concurrency is a significant increase in power of expression. We do nevertheless believe that the encoding of our Actor-based language into the Safe Boxed Ambient calculus can be simplified, in particular by adding additional primitives to the calculus to support our toy Actor language more closely.

B Translating Safe Boxed Ambient calculus into ASCII

$$
\begin{aligned}
\langle\!\langle n \rangle\!\rangle &\mapsto \texttt{n} \text{ where } n \in \mathbf{N} \\
\langle\!\langle \uparrow \rangle\!\rangle &\mapsto \texttt{\^{}} \\
\langle\!\langle * \rangle\!\rangle &\mapsto \begin{cases} \texttt{*} & \text{where used in mobility} \\ & \text{where used in communication} \end{cases} \\
\langle\!\langle \mathsf{in}\, V \rangle\!\rangle &\mapsto \texttt{in } \langle\!\langle V \rangle\!\rangle \\
\langle\!\langle \mathsf{out}\, V \rangle\!\rangle &\mapsto \texttt{out } \langle\!\langle V \rangle\!\rangle \\
\langle\!\langle \overline{\mathsf{in}}\, V \rangle\!\rangle &\mapsto \texttt{IN } \langle\!\langle V \rangle\!\rangle \\
\langle\!\langle \overline{\mathsf{out}}\, V \rangle\!\rangle &\mapsto \texttt{OUT } \langle\!\langle V \rangle\!\rangle \\
\langle\!\langle V_1 \,.\, V_2 \rangle\!\rangle &\mapsto \langle\!\langle V_1 \rangle\!\rangle \texttt{ . } \langle\!\langle V_2 \rangle\!\rangle \\
\langle\!\langle 0 \rangle\!\rangle &\mapsto \texttt{0} \\
\langle\!\langle (\, P_1 \mid P_2 \,) \rangle\!\rangle &\mapsto \texttt{(} \langle\!\langle P_1 \rangle\!\rangle \texttt{ | } \langle\!\langle P_2 \rangle\!\rangle \texttt{)} \\
\langle\!\langle (\nu\, n)(P) \rangle\!\rangle &\mapsto \texttt{(new } \langle\!\langle n \rangle\!\rangle \texttt{)(} \langle\!\langle P \rangle\!\rangle \texttt{)} \\
\langle\!\langle !P \rangle\!\rangle &\mapsto \texttt{! } \langle\!\langle P \rangle\!\rangle \\
\langle\!\langle V[\, P \,] \rangle\!\rangle &\mapsto \langle\!\langle V \rangle\!\rangle \texttt{ [} \langle\!\langle P \rangle\!\rangle \texttt{]} \\
\langle\!\langle V \,.\, P \rangle\!\rangle &\mapsto \langle\!\langle V \rangle\!\rangle \texttt{ . } \langle\!\langle P \rangle\!\rangle \\
\langle\!\langle (x)^{\eta} \,.\, P \rangle\!\rangle &\mapsto \texttt{(} \langle\!\langle x \rangle\!\rangle \texttt{) } \langle\!\langle \eta \rangle\!\rangle \texttt{ . } \langle\!\langle P \rangle\!\rangle \\
\langle\!\langle \langle V \rangle^{\eta} \,.\, P \rangle\!\rangle &\mapsto \texttt{<} \langle\!\langle V \rangle\!\rangle \texttt{>} \langle\!\langle \eta \rangle\!\rangle \texttt{ . } \langle\!\langle P \rangle\!\rangle \\
\langle\!\langle \mathsf{host}\!: a : n \rangle\!\rangle &\mapsto \langle\!\langle \mathsf{host}\!: a \rangle\!\rangle \texttt{:} \langle\!\langle n \rangle\!\rangle \\
\langle\!\langle \mathsf{host}\!: a \rangle\!\rangle &\mapsto \texttt{host:} \langle\!\langle a \rangle\!\rangle
\end{aligned}
$$