School of Informatics, University of Edinburgh

The Laboratory for Foundations of Computer Science

A Typed Language for Distributed Programming

by

Jiansen HE
Supervisor: Prof. Philip Wadler

Informatics Thesis Proposal

School of Informatics **August 2011**
http://www.informatics.ed.ac.uk/

A Typed Language for Distributed Programming

Jiansen HE
Supervisor: Prof. Philip Wadler
SCHOOL *of* INFORMATICS
The Laboratory for Foundations of Computer Science

August 2011

Contents

1 Project Description

1.1 Background and Motivation

Concurrent programming is facing new challenges due to the recent advent of multi-core processors, which rely on parallelism, and the pervasive demands of web applications, which rely on distribution. Theoretic works on concurrency can be traced back to Petri net and other approaches invented in 1960s [5]. Since 1980, more attention has been paid to the study of process calculi (a.k.a. process algebras), which provide a family of formal models for describing and reasoning about concurrent systems [2]. The proliferation of Process Calculi marks the advance in concurrency theory. “The Dreams of Final Theories” [2] for concurrency, however, have not been achieved.

In practice, it is difficult to build a reliable distributed system because of its complexity. Firstly, a distributed system shares features and problems with other concurrent systems. For instance, it is non-deterministic so that traditional input-output semantics is inadequate to be used for reasoning about its behaviour. Secondly, scalability of a distributed system is usually a challenge for its developers. For example, developers of a distributed system are unlikely to know the structure of the system in advance. Moreover, the mobility [10] of a distributed system requires the capability of changing its topological structure on the fly. In addition, the latency of communication could be affected by many unpredictable factors. Thirdly, the system should be tolerant of partial failures. For example, when a site fails to respond to messages, the system should address this failure by invoking a recovery mechanism.

Fortunately, some recent process calculi provide a solid basis for the design of distributed programming languages. For example, the ambient calculus models the movement of processes and devices [10]. Bigraphs, a more abstract model, is aiming at describing spatial aspects and mobility of ubiquitous computing [21]. Lastly, the join-calculus [15] is a remarkable calculus which models features of distributed programming as well as provides a convenient construct, the join patterns, for resources synchronisation. It is also important to note that the syntax of the join-calculus is close to a real programming language and therefore reduces the barrier between understanding the mathematical model and employing this abstract model to guide programming practices.

In addition, good programming principles, such as the OTP design principles[1], have been developed and verified by the long-term implementation practice. OTP is a platform for developing concurrent, distributed, fault tolerant, and non-stop Erlang applications [4]. In the past 20 years, the Erlang language was used to build large reliable applications like RabbitMQ, Twitterfall, and many telecommunications systems.

Nevertheless, Erlang is an untyped functional programming language whereas it is widely believed that applications built in typed languages enjoy the advantages of *reliability* and *maintainability*. In this project, reliability of a distributed system includes 3 aspects: *a*) every worker process should be supervised by a sensor, which is responsible for monitoring its child processes and restarting failed child processes; *b*) all information subscribers interested in the same notification should receive the same message; *c*) if an application is built with different layers, applications built at one layer should only depend on services provided by the layer below, without leaking its internal information to other layers. The maintainability of a distributed system is 3 fold: i) a process or computation may migrate across the network and perform consistent behaviour regardless of the concrete computation framework where it is executed; ii) any process or computation device may join or leave the system peacefully without affecting the rest of the system; iii) the system should be developed in a modular way.

Therefore, this project will aim to investigate how to provide reliable and maintainable distributed applications via embedding good distributed programming principles into a type system for distributed programming. The principles to be considered are the OTP design principles.

[1]OPT is stand for Open Telecom Platform. It provides a set of libraries for developing Erlang applications. For the above reason, it is also known as Erlang/OTP design principles

1.2 Project Aim

The aim of this project is *to formalise an approach that improves the reliability and maintainability of distributed programs.* The project intends to explore factors that encourage programmers to employ good programming principles. This project will demonstrate how types and appropriate models ease the construction of reliable distributed systems.

1.3 Project Objectives

1. To build a library that implements the OTP design principles in a typed language supporting the join-calculus. This piece of work is going to investigate two problems. (a) How types will influence the programmability of distributed applications. In distributed programming based on message passing, untyped channels have the advantage of carrying data which may be rejected by a rigid type system whereas typed channels detect some forms of data misuse earlier. This project will encapsulate "tricks" around using untyped channels into libraries and provide convenient ways for constructing reliable programs. (b) To what extend the join-calculus fits the OTP design principles. Although the OTP design principles have been proposed for the Erlang language, which is based on the Actor model, it should be possible to rephrase those good principles in other languages and models. This project aims to realise general *principles* rather than implement specific library functions.

2. To formalise the approach used in the first objective. The wide applications of our approach will be demonstrated by removing ad-hoc language features used in the first task. The formalism will be based on the join-calculus. New language features will only be cautiously added if the old model is incapable to tackle certain problems. For example, the failure model in the join-calculus might be replaced by exceptions, which is more sophisticate in tracing failures. Moreover, properties of the formalism will be studied. Apart from algebraic properties, a set of criteria for evaluating the maintainability and reliability of distributed systems will be defined and verified.

3. To extend the core calculus to gain engineering benefits. For instance, although encoding the λ-calculus should be possible in our model, which derives from the join-calculus, shipping function closures in the core-calculus could be as expensive as in the core join-calculus. One possible approach to reduce this cost is defining a higher-order join-calculus with λ-abstraction and function application, so that functions would be first class objects in the new model.

1.4 Dissertation Outline

Chapter 1 (*Introduction)* will give a brief introduction to this research, including: (1) the main research topic, (2) the importance of the research, (3) definitions of key concepts, (4) a summary of related work, (5) main novel contributions of this research, and finally (6) a roadmap for the rest of the dissertation.

Chapter 2 (*Process Calculi)* will summarize methodologies and results in the studies of process calculi. Considering the fruitful results in the area of process calculi and the purpose of this dissertation, only those process calculi that directly influenced this research will be introduced.

Chapter 3 (*Other Concurrent Models)* will compare the join-calculus with three widely used models in distributed programming: Petri net, Actor model, and reactive events. This chapter will first define some criteria for evaluating distributed systems. Then, a tiny representative example in distributed programming [2] will be implemented in languages that directly support each model. Lastly, a short summary will be given.

[2]For example, a pingpong program or a simple client-server application.

Chapter 4 (*A Library for the OTP Design Principles in typed languages)* will introduce some OTP design principles, followed by how those principles are employed to build libraries in a typed language that supports the join-calculus. This chapter will enumerate ad-hoc strategies adopted to get around the problem of porting libraries from an untyped language to an typed language.

Chapter 5 (*Supervised Distributed Join-Calculus)* will define a formalism that extends the distributed join-calculus with the supervision principle. It will list properties that ensure the reliability of the extended formalism. The proof for those properties will be given in the content or as an appendix.

Chapter 6 (*Extended Models)* will present some extensions of our model. For each extension, we will discuss the trade-off between retaining the core calculus defined in Chapter 5 and adding new primitives to it.

Chapter 7 (*Conclusion and Future Work)* will evaluate the whole thesis by highlighting the advantages and limitations of the achieved work. The thesis may also elaborate possible trends in the area of distributed programming.

1.5 Dissemination of Results

The main results of this dissertation are expected to be sent for publication to some of the following international conferences: CONCUR, DISC, ICPADS, ICDCS, MOBILITY, OPODIS, PODC, PPoPP, or SIROCCO.

2 Literature Review

Process calculi (or process algebras) is a family of algebraic approaches to study concurrent systems [5]. A process calculus need to address following questions: 1. What are primitive components in a concurrent system? 2. What kind of computations and process behaviours could be modelled in this calculus? 3. What does equivalence between two processes mean in this calculus? In the past 30 years, a number of process calculi have been proposed and studied. Each calculus answers the first two questions from slightly different perspectives. For the third question, it is worth to note that a particular calculus may have more than one interpretation for process equivalences on different levels.

This chapter will present selective results in concurrent theories and their implementations. §2.1 will use the π-calculus as an example to sketch research topics in the area of process calculi, which provides formal basis for the design of concurrent languages. §2.2 and §2.3 will focus on the programmability of the join-calculus and the ambient-calculus. Both sections will introduce how to the subject calculus addresses problems in distributed programming and how ideas in the subject calculus are applied in real programming languages. Algebraic properties of the subject calculus, however, will not be investigated in depth in this short review. Then, §2.4 will introduce OTP design principles which make distributed Erlang programs more reliable and maintainable. §2.5 will present indirect and direct encodings for the λ-calculus in process calculi. Lastly, §2.6 will give a short survey on a general framework for implementing concurrent programming languages.

2.1 The π-calculus

2.1.1 Syntax and Notations

The π-calculus [22, 30] is a name passing calculus, in the sense that computations are represented by passing *names* through *channels*. The π-calculus uses lower-case letter to denote a name, which could be either a channel or a variable.

Syntax of the π-calculus given in Table 1 also denotes part of its semantics. The prefix of a process denotes its capability, which could be (i) receiving names from a channel; (ii) sending names through a channel; or (iii) performing an internal action that cannot be observed from outside. Informally, a process could : (1) evolve to another process after performing an action; (2) evolve as two processes who are running at the same time; (3) evolve as one of the two processes; (4) evolve to another process when two names become equal; (5) a process with a private name; (6) infinite parallel composition of the same process; or (7) an inert process that does nothing. Finally, two processes are structurally congruent, $\equiv$, if they are equal up to structure rules listed in Table 2

In the rest of this section, we distinguish reduction relation($\longrightarrow$) from labelled transition relations($\xrightarrow{\alpha}$). Specifically, the assertion $P \longrightarrow P'$ states that process P can evolve to process P' as a result of an action *within* P. On the other hand, $P \xrightarrow{\alpha} Q$ indicates that P can evolve to Q after performing action α. Reduction relations are formally defined in Table 3 and transition relations are formally defined in Table 4.

α	$::=$		action	P	$::=$		process
	$\mid$	$u(\widetilde{x})$	input		$\mid$	$\alpha.P$	prefix
	$\mid$	$\overline{u}\langle\widetilde{y}\rangle$	output		$\mid$	$P \mid P$	composition
	$\mid$	τ	silent/internal		$\mid$	$P + P$	summation
					$\mid$	$[x = y].P$	match
					$\mid$	$(\nu x).P$	restriction
					$\mid$	$!P$	replication
					$\mid$	0	inert process

Table 1: Syntax of the π-calculus

SC-MAT	$[x = x]\alpha.P$	$\equiv$	$\alpha.P$
SC-SUM-ASSOC	$P_1 + (P_2 + P_3)$	$\equiv$	$(P_1 + P_2) + P_3$
SC-SUM-COMM	$P_1 + P_2$	$\equiv$	$P_2 + P_1$
SC-SUM-INACT	$P + 0$	$\equiv$	P
SC-COMP-ASSOC	$P_1 \mid (P_2 \mid P_3)$	$\equiv$	$(P_1 \mid P_2) \mid P_3$
SC-COMP-COMM	$P_1 \mid P_2$	$\equiv$	$P_1 \mid P_2$
SC-COMP-INACT	$P \mid 0$	$\equiv$	P
SC-RES	$\nu z\ \nu w\ P$	$\equiv$	$\nu w\ \nu z\ P$
SC-RES-INACT	$\nu z\ 0$	$\equiv$	0
SC-RES-COMP	$\nu z\ (P_1 \mid P_2)$	$\equiv$	$P_1 \mid \nu z\ P_2, if\ z \not\in fn(P_1)$
SC-REP	$!P$	$\equiv$	$P \mid !P$

Table 2: The axioms of structural congruence

$$\text{R-INTER}\ \frac{}{(\bar{x}\langle y\rangle.P_1 + M_1) \mid (x(z).P_2 + M_2) \longrightarrow P_1 \mid P_2\{y/z\}}$$

$$\text{R-PAR}\ \frac{P_1 \longrightarrow P_1^{'}}{P_1 \mid P_2 \longrightarrow P_1^{'} \mid P_2} \qquad \text{R-RES}\ \frac{P \longrightarrow P^{'}}{\nu z P \longrightarrow \nu z P'}$$

$$\text{R-STRUCT}\ \frac{P_1 \equiv P_2 \quad P_2 \longrightarrow P_2^{'} \quad P_2^{'} \equiv P_1^{'}}{P_1 \longrightarrow P_1^{'}} \qquad \text{R-TAU}\ \frac{}{\tau P + M \longrightarrow P}$$

Table 3: Reduction rules in the π-calculus

$$\textsc{Out}\ \frac{}{\bar{x}y.P \xrightarrow{\bar{x}y} P} \qquad \textsc{Inp}\ \frac{}{x(z).P \xrightarrow{xy} P\{y/z\}}$$

$$\textsc{Tau}\ \frac{}{\tau.P \xrightarrow{\tau} P} \qquad \textsc{Mat}\ \frac{\pi.P \xrightarrow{\alpha} P'}{[x=x]\pi.P \xrightarrow{\alpha} P'}$$

$$\textsc{Sum-l}\ \frac{P \xrightarrow{\alpha} P'}{P + Q \xrightarrow{\alpha} P'}$$

$$\textsc{Par-l}\ \frac{P \xrightarrow{\alpha} P'}{P \mid Q \xrightarrow{\alpha} P' \mid Q}\ \ \mathsf{bn}(\alpha) \cap \mathsf{fn}(Q) = \emptyset$$

$$\textsc{Comm-l}\ \frac{P \xrightarrow{\bar{x}y} P' \quad Q \xrightarrow{xy} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'}$$

$$\textsc{Close-l}\ \frac{P \xrightarrow{\bar{x}(z)} P' \quad Q \xrightarrow{xz} Q'}{P \mid Q \xrightarrow{\tau} \nu z\,(P' \mid Q')}\ \ z \notin \mathsf{fn}(Q)$$

$$\textsc{Res}\ \frac{P \xrightarrow{\alpha} P'}{\nu z\, P \xrightarrow{\alpha} \nu z\, P'}\ \ z \notin \mathsf{n}(\alpha) \qquad \textsc{Open}\ \frac{P \xrightarrow{\bar{x}z} P'}{\nu z\, P \xrightarrow{\bar{x}(z)} P'}\ \ z \neq x$$

$$\textsc{Rep-act}\ \frac{P \xrightarrow{\alpha} P'}{!P \xrightarrow{\alpha} P' \mid !P}$$

$$\textsc{Rep-comm}\ \frac{P \xrightarrow{\bar{x}y} P' \quad P \xrightarrow{xy} P''}{!P \xrightarrow{\tau} (P' \mid P'') \mid !P}$$

$$\textsc{Rep-close}\ \frac{P \xrightarrow{\bar{x}(z)} P' \quad P \xrightarrow{xz} P''}{!P \xrightarrow{\tau} (\nu z\,(P' \mid P'')) \mid !P}\ \ z \notin \mathsf{fn}(P)$$

Table 4: Transition rules in the π-calculus

2.1.2 Equivalence Relations

Equivalence relations are important concepts in an algebraic system. Unfortunately, the variety of equivalence relations in the π-calculus and other classical process calculi are usually obstacles to learning a calculus. This section will give a gentle introduction to the most important equivalence relation in the π-calculus, the strong barbed bisimulation, followed by short definitions cited from [30] in a designed order. By reading this section, readers will obtain a basic understanding for all equivalences in the Figure 1 and most other common equivalences in literature.

Barbed relations study the behaviour of a process via examining its potential interactions with the environment. Formally,

Definition 2.1. *For each name or co-name μ, the observability predicate $\downarrow_\mu$ is defined by:*
(1) $P \downarrow_\mu$ if P can perform an input action via channel μ
(2) $P \downarrow_{\overline{\mu}}$ if P can perform an output action via channel μ

Strong barbed relations can be defined as follows:

Definition 2.2. *A relation S is a* **strong barbed bisimulation** *if whenever (P,Q) $\in$ S,*
(1) $P \downarrow_\mu$ implies $Q\downarrow_\mu$
(2) $P \xrightarrow{\tau}$ P' implies $Q \xrightarrow{\tau}$ Q' for some Q' with (P', Q') $\in$ S
(3) $Q \downarrow_\mu$ implies $P\downarrow_\mu$
(4) $Q \xrightarrow{\tau}$ Q' implies $P \xrightarrow{\tau}$ P' for some P' with (P', Q') $\in$ S

Definition 2.3. *P and Q are* **strong barbed bisimilar** *if (P,Q) $\in$ S for some barbed bisimulation S.*

Definition 2.4. *P and Q are* **strong barbed congruence** *if C[P] and C[Q] are strong barbed bisimilar for any context C*[3].

Definition 2.5. *P and Q are* **strong barbed equivalent** *if $P \mid R$ and $Q \mid R$ are strong barbed bisimilar for any process R.*

Bisimulation, **bisimilarity**, **congruence**, **and equivalent**, as revealed in the barbed relations, are four related concepts in process calculi. For this reason, in the rest of this subsection, where various of equivalent relations will be introduced, only one of the four concepts in each group will be defined explicitly. The meaning of the rest three relations in each group should be apparent.

Definition 2.6. *A relation S is a* **reduction bisimulation** *if whenever (P,Q) $\in$ S,*
(1) $P \xrightarrow{\tau}$ P' implies $Q \xrightarrow{\tau}$ Q' for some Q' with (P', Q') $\in$ S
(2) $Q \xrightarrow{\tau}$ Q' implies $P \xrightarrow{\tau}$ P' for some P' with (P', Q') $\in$ S

In terms of evaluation strategy, the π-calculus could adopt either the early instantiation scheme or the late instantiation scheme. In the early instantiation scheme, variables are instantiated as soon as an input message is received, more precisely, $\frac{-}{a(x).P \xrightarrow{av} P\{v/x\}}$. On the contrary, in the late instantiation scheme, bound variables of input actions are instantiated only when they are involved in an internal communication.

Definition 2.7. *A binary relation S on processes P and Q is an* **early simulation**, *$\sim_e$, if PSQ implies that*
1. If $P \xrightarrow{\alpha}$ P' and α is a free action[4]*, then for some Q', $Q \xrightarrow{\alpha}$ Q' and P'SQ'*
2. If $P \xrightarrow{x(y)}$ P' and $y \notin n(P,Q)$, then for all w, there is Q' such that $Q \xrightarrow{x(y)}$ Q' and $P\{w/y\}SQ\{w/y\}$
3. If $P \xrightarrow{\bar{x}(y)}$ P' and $y \notin n(P,Q)$, then for some Q', $Q \xrightarrow{\bar{x}(y)}$ Q' and P'SQ'

[3]A context, $C[\cdot]$ is a term written using the same syntax of the π-calculus and an additional constant $\cdot$. Substituting the $\cdot$ by a process P result in a π-calculus term $C[P]$

[4]an internal action or a free output.

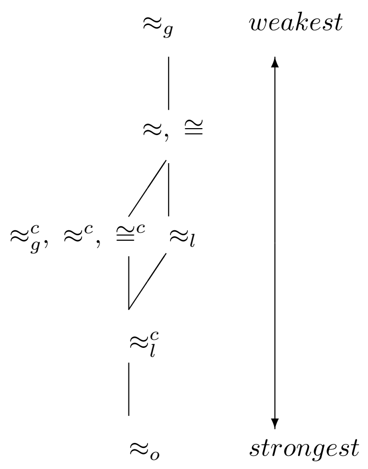
Definition 2.8. *A binary relation S on agents is a* **late simulation**, $\sim_l$, *if PSQ implies that*
1. *If* $P \xrightarrow{\alpha} P'$ *and* α *is a free action, then for some Q', $Q \xrightarrow{\alpha} Q'$ and P'SQ'*
2. *If* $P \xrightarrow{x(y)} P'$ *and* $y \notin n(P,Q)$*, then for some Q',* $Q \xrightarrow{x(y)} Q'$ *and for all w,* $P\{w/y\}SQ\{w/y\}$
3. *If* $P \xrightarrow{\bar{x}(y)} P'$ *and* $y \notin n(P,Q)$*, then for some Q',* $Q \xrightarrow{\bar{x}(y)} Q'$ *and P'SQ'*

Definition 2.9. *P and Q are* **full bisimilar**, $P \approx^c Q$, *if* $P\sigma \approx Q\sigma$ *for every substitution* σ.

Definition 2.10. *A binary relation R over processes is an* **open bisimulation**, $\approx_o$, *if for every pair of elements (p,q)* $\in R$ *and for every name substitution* σ *and every action* α, *whenever* $p\sigma \xrightarrow{\alpha} p'$ *then there exists some q' such that* $q\sigma \xrightarrow{\alpha} q'$ *and (p', q')* $\in R$.

Definition 2.11. *A relation is* **ground bisimulation**, $\approx_g$, *iff whenever* $P \approx_g Q$*, there is* $z \notin fn(P, Q)$ *such that if* $P \xrightarrow{\alpha} Q$*, where* α *is an action, then then* $Q \overset{\alpha}{\Rightarrow} \approx_g P$.

To conclude this subsection, Figure 1, cited from [30], presents the hierarchy of equivalences in the π-calculus.



$\approx_g$:	*ground bisimilarity*	$\approx$:	*bisimilarity*	$\cong$:	*barbed equivalence*
$\approx_g^c$:	*ground congruence*	$\approx^c$:	*full bisimilarity*	$\cong^c$:	*barbed congruence*
$\approx_l$:	*late bisimilarity*	$\approx_l^c$:	*late congruence*	$\approx_o$:	*open bisimilarity*

Figure 1: Hierarchy of equivalences in the π-calculus

2.2 The Join-Calculus and the JoCaml Programming Language

Join-calculus [17] is a name passing process calculus that is designed for the distributed programming. There are two versions of the join-calculus, namely the core join-calculus and the distributed join-calculus. The core join-calculus could be considered as a variant of the π-calculus. It is as expressive as the asynchronous π-calculus in the sense that translations between those two calculi are well formulated. A remarkable construct in the join-calculus is join patterns, which provides a convenient way to express process synchronisations. This feature also makes the join-calculus closer to a real programming language. The distributed join-calculus extends the core calculus with location, process migration, and failure recovery. This proposal uses the short phrase "join-calculus" to refer to the distributed join-calculus which includes all components in the core join-calculus. The syntax (Table 5), the scoping rules (Table 6), and the reduction rules (Table 7) of the join-calculus are well summarized in [16].

P	$::=$		processes	D	$::=$		definition
	$\mid$	$x\langle\widetilde{v}\rangle$	asynchronous message		$\mid$	$J \triangleright P$	local rule
	$\mid$	$\mathbf{def}\ D\ \mathbf{in}\ P$	local definition		$\mid$	$\top$	inert definition
	$\mid$	$P \mid P$	parallel composition		$\mid$	$D \wedge D$	co-definition
	$\mid$	$\mathbf{0}$	inert process		$\mid$	$a\ [D : P]$	**sub-location**
	$\mid$	$go\langle a, \kappa\rangle$	**migration**		$\mid$	$\Omega a\ [D : P]$	**dead sub-location**
	$\mid$	$halt\langle\rangle$	**termination**	J	$::=$		join-pattern
	$\mid$	$fail\langle a, \kappa\rangle$	**failure detection**		$\mid$	$x\langle\widetilde{v}\rangle$	message pattern
					$\mid$	$J \mid J$	synchronous join-pattern

Constructs whose explanation is in **bold** font are only used in the distributed join-calculus. Other constructs are used in both distributed and local join-calculus.

Table 5: Syntax of the distributed join-calculus

$J :$	$dv[x\langle\widetilde{v}\rangle]$	$\overset{def}{=}$ $\{x\}$	$rv[x\langle\widetilde{v}\rangle]$	$\overset{def}{=}$	$\{u \in \widetilde{v}\}$	
	$dv[J \mid J']$	$\overset{def}{=}$ $dv[J] \cup dv[J']$	$rv[J \mid J']$	$\overset{def}{=}$	$rv[J] \uplus rv[J']$	
$D :$	$dv[J \triangleright P]$	$\overset{def}{=}$ $dv[J]$	$rv[J \triangleright P]$	$\overset{def}{=}$	$dv[J] \cup (fv[P] - rv[J])$	
	$dv[\top]$	$\overset{def}{=}$ $\emptyset$	$fv[\top]$	$\overset{def}{=}$	$\emptyset$	
	$dv[D \wedge D']$	$\overset{def}{=}$ $dv[D] \cup dv[D']$	$fv[D \wedge D']$	$\overset{def}{=}$	$fv[D] \cup fv[D']$	
	$dv[a\ [D : P]]$	$\overset{def}{=}$ $\{a\} \uplus dv[D]$	$fv[a\ [D : P]]$	$\overset{def}{=}$	$\{a\} \cup fv[D] \cup fv[P]$	[h]
$P :$	$fv[x\langle\widetilde{v}]$	$\overset{def}{=}$ $\{x\} \cup \{u \in \widetilde{v}\}$	$fv[go\langle a, \kappa\rangle]$	$\overset{def}{=}$	$\{a, \kappa\}$	
	$fv[\mathbf{0}]$	$\overset{def}{=}$ $\emptyset$	$fv[halt\langle\rangle]$	$\overset{def}{=}$	$\emptyset$	
	$fv[P \mid P']$	$\overset{def}{=}$ $fv[P] \cup fv[P']$	$fv[fail\langle a, \kappa\rangle]$	$\overset{def}{=}$	$\{a, \kappa\}$	
	$fv[\mathbf{def}\ D\ \mathbf{in}\ P]$	$\overset{def}{=}$ $(fv[P] \cup fv[D]) - dv[D]$				

Well-formed conditions for D: A location name can be defined only once; a channel name can only appear in the join-patterns at one location.

Table 6: Scopes of the distributed join-calculus

str-join	$\vdash\ P_1 \mid P_2$	$\rightleftharpoons$	$\vdash\ P_1,\ P_2$	
str-null	$\vdash\ \mathbf{0}$	$\rightleftharpoons$	$\vdash$	
str-and	$D_1\ \wedge D_2\ \vdash$	$\rightleftharpoons$	$D_1,\ D_2\ \vdash$	
str-nodef	$\top$	$\rightleftharpoons$	$\vdash$	
str-def	$\vdash\ \mathbf{def}\ D\ \mathbf{in}\ P$	$\rightleftharpoons$	$D\sigma_{dv}\ \vdash\ P\sigma_{dv}$	(range(σ_{dv}) fresh)
str-loc	$\varepsilon a\ [D : P] \vdash_{\varphi}$	$\rightleftharpoons$	$\vdash_{\varphi}\ \Vert\ \{D\}\ \vdash_{\varphi\varepsilon a}\ \{P\}$	(a frozen)
red	$J \triangleright P\ \vdash_{\varphi}\ J\sigma_{rv}$	$\longrightarrow$	$J \triangleright P\ \vdash_{\varphi}\ P\sigma_{rv}$	(φ alive)
comm	$\vdash_{\varphi}\ x\langle\widetilde{v}\rangle\ \Vert\ J \triangleright P \vdash$	$\longrightarrow$	$\vdash_{\varphi}\ \Vert\ J \triangleright P\ \vdash\ x\langle\widetilde{v}\rangle$	($x \in dv[J]$, φ alive)
move	$a[D : P \mid go\langle b, \kappa\rangle]\ \vdash_{\varphi} \Vert \vdash_{\psi\varepsilon b}$	$\longrightarrow$	$\vdash_{\varphi}\ \Vert\ a\ [D : P \mid \kappa\langle\rangle]\ \vdash_{\psi\varepsilon b}$	(φ alive)
halt	$a[D : P \mid halt\langle\rangle]\ \vdash_{\varphi}$	$\longrightarrow$	$\Omega a[D : P]\ \vdash_{\varphi}$	(φ alive)
detect	$\vdash_{\varphi} fail\langle a, \kappa\rangle\ \Vert \vdash_{\psi\varepsilon a}$	$\longrightarrow$	$\vdash_{\varphi}\ \kappa\langle\rangle\ \Vert \vdash_{\psi\varepsilon a}$	($\psi\varepsilon a$ dead, φ alive)

Side conditions: in **str-def**, σ_{dv} instantiates the channel variables $dv[D]$ to distinct, fresh names; in **red**, σ_{rv} substitutes the transmitted names for the received variables $rv[J]$; φ is dead if it contains Ω, and alive otherwise; "a frozen" means that a has no sublocations; εa denotes either a or Ωa

Table 7: The distributed reflexive chemical machine

P	$::=$		processes	D	$::=$		definition
	$\mid$	$x\langle\widetilde{v}\rangle$	asynchronous message		$\mid$	$J \triangleright P$	local rule
	$\mid$	**def** D **in** P	local definition		$\mid$	$\top$	inert definition
	$\mid$	$P \mid P$	parallel composition		$\mid$	$D \wedge D$	co-definition
	$\mid$	**0**	inert process	J	$::=$		join-pattern
	$\mid$	$\mathrm{x}(\widetilde{V}); P$	**sequential composition**		$\mid$	$x\langle\widetilde{v}\rangle$	message pattern
	$\mid$	**let** $\widetilde{u} = \widetilde{V}$ **in** P	**named values**		$\mid$	$J \mid J$	synchronous join-pattern
	$\mid$	**reply** $\widetilde{V}$ **to** x	**implicit continuation**	V	$::=$		values
					$\mid$	x	value name
					$\mid$	$\mathrm{x}(\widetilde{V})$	**synchronous call**

$$\begin{aligned}
\mathrm{x}(\widetilde{v}) &= x\langle\widetilde{v}, \kappa_x\rangle && \text{(in join-patterns)} \\
\textbf{reply } \widetilde{V} \textbf{ to } \mathrm{x} &= \kappa_x\langle\widetilde{V}\rangle && \text{(in process)} \\
x\langle\widetilde{V}\rangle &= \textbf{let } \widetilde{v} = \widetilde{V} \textbf{ in } x\langle\widetilde{v}\rangle \\
\textbf{let } \widetilde{u} = \widetilde{V} \textbf{ in } P &= \textbf{let } u_1 = V_1 \textbf{ in let } u_2 = \cdots \textbf{ in } P \\
\textbf{let } \widetilde{u} = \mathrm{x}(\widetilde{V}) \textbf{ in } P &= \textbf{def } \kappa\langle\widetilde{u}\rangle \triangleright P \textbf{ in } x\langle\widetilde{V}, \kappa\rangle \\
\textbf{let } u = v \textbf{ in } P &= P\{v/u\} \\
\mathrm{x}(\widetilde{V}); P &= \textbf{def } \kappa\langle\rangle \triangleright P \textbf{ in } x\langle\widetilde{V}, \kappa\rangle
\end{aligned}$$

Table 8: The core join-calculus with synchronous channel, sequencing, and let-binding

2.2.1 The Local Reflexive Chemical Machine (RCHAM)

The denotational semantics of the join-calculus is usually described in the domain of a reflexive chemical machine (RCHAM). A local RCHAM consists of two parts: a multiset of definitions D and a multiset of active processes P. Definitions specify possible reductions of processes, while active processes can introduce new names and reaction rules.

The six chemical rules for the local RCHAM are **str-join**, **str-null**, **str-and**, **str-nodef**, **str-def**, and **red** in Table 7. As their names suggest, the first 5 are structure rules whereas the last one is reduction rule. Structure rules correspond to reversible syntactical rearrangements. The reduction rule, **red**, on the other hand, represents an irreversible computation.

Finally, for the ease of writing programs, the local join-calculus could be extended with synchronous channel, sequencing, and let-bindings as in Table 8. The distributed join-calculus could be extended similarly.

2.2.2 Distributed Solutions

Distributed system in the join-calculus is constructed in three steps: first, definitions and processes are partitioned into several local solutions; then, each local solution is attached with a unique location name; finally, location names are organized in a location tree.

A distributed reflexive chemical machine (DRCHAM) is simply a multiset of RCHAMs. It is important to note that message pending to a remotely defined channel will be forwarded to the RCHAM where the channel is defined before applying any **red** rule. The above process is a distinction between the join-calculus and other distributed models. The side effect of this evaluation strategy is that both channel and location names must be pairwise distinct in the whole system. As a consequence, a sophisticate name scheme is required for a language that implements the join-calculus.

To support process migration, a new contract, $go\ \langle b, \kappa\rangle$, is introduced, together with the **move** rule. There are two effects of applying the move rule. Firstly, site a moves from one place (φa) to another ($\psi\varepsilon$a). Secondly, the continuation $\kappa\langle\rangle$ may trigger another computation at the new location.

2.2.3 The Failure Model

A failed location in the join-calculus cannot respond to messages. Reactions inside a failed location or its sub-locations are prevented by the side-condition of reduction rules. Nevertheless, messages and locations are allowed to move into a failed location, but will be frozen in that dead location (str-loc).

To model failure and failure recovery, two primitives $halt\langle\rangle$ and $fail\langle\cdot,\cdot\rangle$ are introduced to the calculus. Specifically speaking, $halt\langle\rangle$ terminates the location where it is triggered (rule halt), whereas $fail\langle a, \kappa\rangle$ triggers the continuation $\kappa\langle\rangle$ when location a fails (rule detect).

2.2.4 The JoCaml Programming Language

The JoCaml programming language is an extension of OCaml. JoCaml supports the join-calculus with similar syntax and more syntactic sugars. When using JoCaml to build distributed applications, users should be aware of following three limitations in the current release (version 3.12) [14]:

1. Functions and closures transmission are not supported. In the join-calculus, distributed calculation is modelled as sending messages to a remotely defined channel. As specified in the **comm** rule, messages sent to a remotely defined channel will be forwarded to the place where the channel is defined. In some cases, however, programmers may want to define an computation at one place but execute the computation elsewhere. The standard JoCaml, unfortunately, does not support code mobility.

2. Distributed channels are untyped. In JoCaml, distributed port names are retrieve by enquiring its registered name (a string) from name service. Since JoCaml encourages modular development, codes supposed to be run at difference places are usually wrote in separated modules and complied independently. The annotated type of a distributed channel, however, is not checked by the name service. Invoking a remote channel whose type is erroneously annotated may cause a run-time error.

3. When mutable value is required over the web, a new copy, rather than a reference to the value, is sent. This may cause problems when a mutable value is referenced at many places across the network.

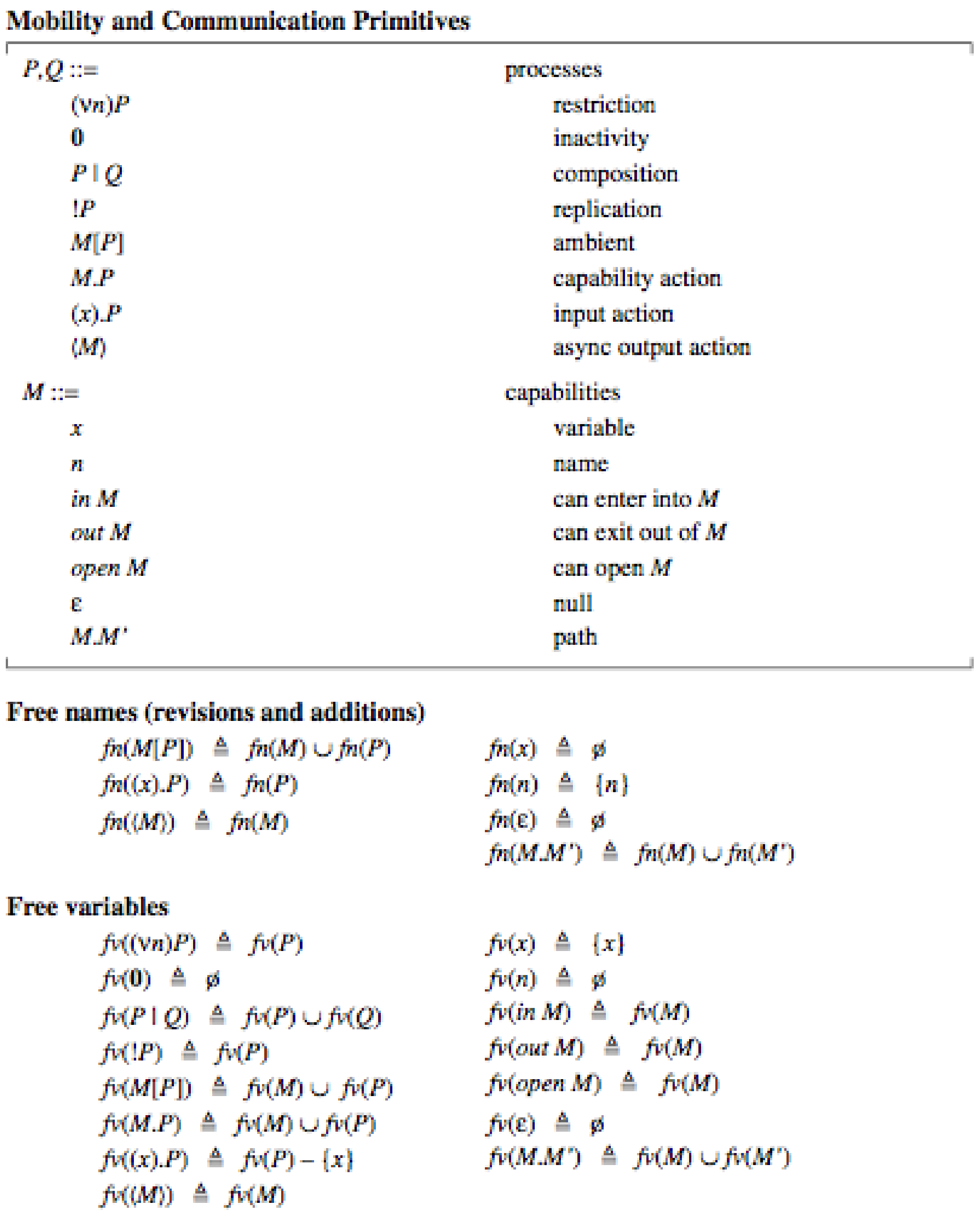
2.3 The Ambient Calculus and the Obliq Programming Language

2.3.1 The Ambient Calculus

The ambient calculus provides a formal basis for describing mobility in concurrent systems. Here mobility refers to both *mobile computing* (computation carried out in mobile devices) and *mobile computation* (code moves between the network sites) [10]. In reality, there is an additional security requirement for mobility, that is, the authorization for an agent to enter or exit certain administrative domain (e.g. a firewall). The ambient calculus solves the above problems with a fundamental concept: ambient. The three key attributes of a ambient are:

- a name for access control (enter, exit, and open the ambient).
- a collection of local processes/agents that control the ambient.
- a collection of sub-ambients.

An atomic computation in the ambient calculus is a one-step movement of an ambient. Although the pure ambient calculus with mobility is Turing-complete [10], communication primitives are necessary to comfort the encoding of other communication based calculi such as the π-calculus. The full calculus is given through Table 9 to 11, cited from [10]. It is important to note that communication in the ambient calculus are local. In other words, value (name or capability) communication only happens between two processes inside the same ambient.



Mobility and Communication Primitives

$P, Q ::=$	processes
$(\nu n)P$	restriction
$\mathbf{0}$	inactivity
$P \mid Q$	composition
$!P$	replication
$M[P]$	ambient
$M.P$	capability action
$(x).P$	input action
$\langle M \rangle$	async output action
$M ::=$	capabilities
x	variable
n	name
$in\ M$	can enter into M
$out\ M$	can exit out of M
$open\ M$	can open M
ε	null
$M.M'$	path

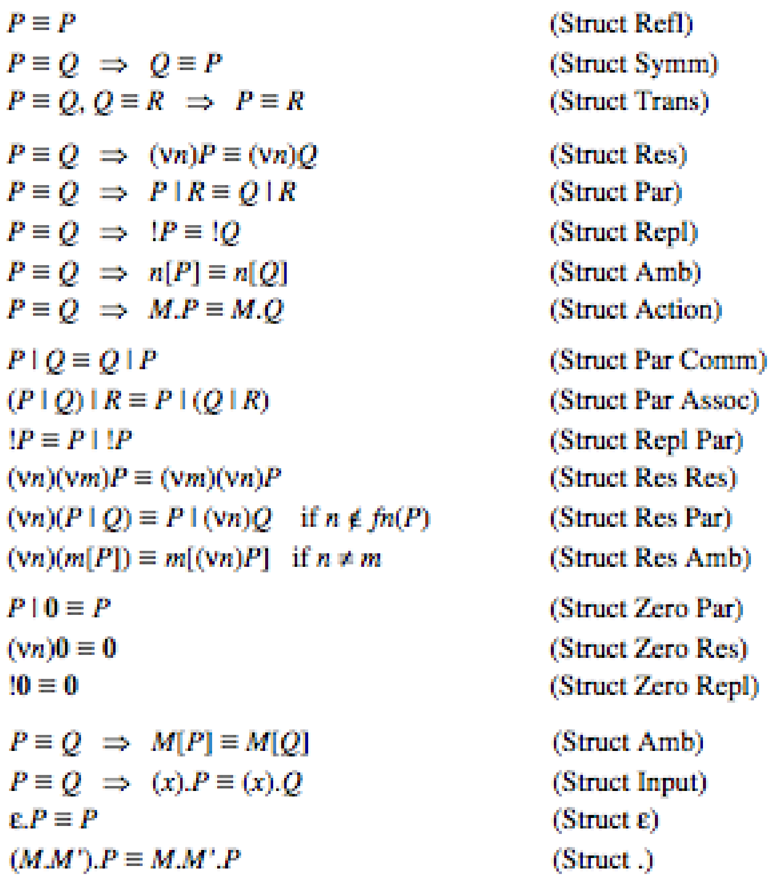
Free names (revisions and additions)

$fn(M[P]) \triangleq fn(M) \cup fn(P)$	$fn(x) \triangleq \emptyset$
$fn((x).P) \triangleq fn(P)$	$fn(n) \triangleq \{n\}$
$fn(\langle M \rangle) \triangleq fn(M)$	$fn(\varepsilon) \triangleq \emptyset$
	$fn(M.M') \triangleq fn(M) \cup fn(M')$

Free variables

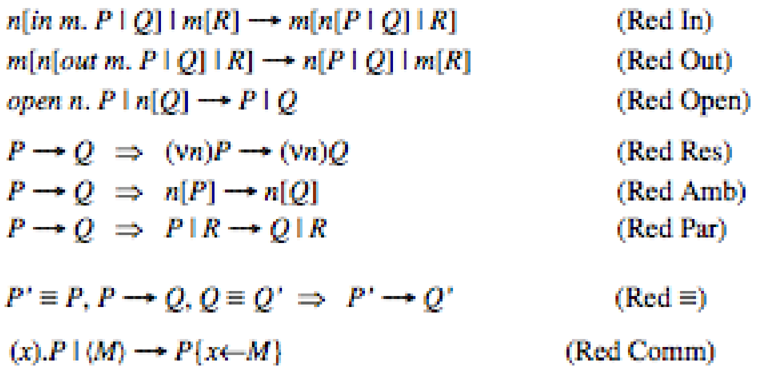
$fv((\nu n)P) \triangleq fv(P)$	$fv(x) \triangleq \{x\}$
$fv(\mathbf{0}) \triangleq \emptyset$	$fv(n) \triangleq \emptyset$
$fv(P \mid Q) \triangleq fv(P) \cup fv(Q)$	$fv(in\ M) \triangleq fv(M)$
$fv(!P) \triangleq fv(P)$	$fv(out\ M) \triangleq fv(M)$
$fv(M[P]) \triangleq fv(M) \cup fv(P)$	$fv(open\ M) \triangleq fv(M)$
$fv(M.P) \triangleq fv(M) \cup fv(P)$	$fv(\varepsilon) \triangleq \emptyset$
$fv((x).P) \triangleq fv(P) - \{x\}$	$fv(M.M') \triangleq fv(M) \cup fv(M')$
$fv(\langle M \rangle) \triangleq fv(M)$	

Table 9: Syntax and scope in the ambient-calculus



$P \equiv P$	(Struct Refl)
$P \equiv Q \Rightarrow Q \equiv P$	(Struct Symm)
$P \equiv Q, Q \equiv R \Rightarrow P \equiv R$	(Struct Trans)
$P \equiv Q \Rightarrow (\nu n)P \equiv (\nu n)Q$	(Struct Res)
$P \equiv Q \Rightarrow P \mid R \equiv Q \mid R$	(Struct Par)
$P \equiv Q \Rightarrow !P \equiv !Q$	(Struct Repl)
$P \equiv Q \Rightarrow n[P] \equiv n[Q]$	(Struct Amb)
$P \equiv Q \Rightarrow M.P \equiv M.Q$	(Struct Action)
$P \mid Q \equiv Q \mid P$	(Struct Par Comm)
$(P \mid Q) \mid R \equiv P \mid (Q \mid R)$	(Struct Par Assoc)
$!P \equiv P \mid !P$	(Struct Repl Par)
$(\nu n)(\nu m)P \equiv (\nu m)(\nu n)P$	(Struct Res Res)
$(\nu n)(P \mid Q) \equiv P \mid (\nu n)Q$ if $n \notin fn(P)$	(Struct Res Par)
$(\nu n)(m[P]) \equiv m[(\nu n)P]$ if $n \neq m$	(Struct Res Amb)
$P \mid \mathbf{0} \equiv P$	(Struct Zero Par)
$(\nu n)\mathbf{0} \equiv \mathbf{0}$	(Struct Zero Res)
$!\mathbf{0} \equiv \mathbf{0}$	(Struct Zero Repl)
$P \equiv Q \Rightarrow M[P] \equiv M[Q]$	(Struct Amb)
$P \equiv Q \Rightarrow (x).P \equiv (x).Q$	(Struct Input)
$\varepsilon.P \equiv P$	(Struct ε)
$(M.M').P \equiv M.M'.P$	(Struct .)

Table 10: Structure congruence in the ambient-calculus



$n[in\ m.\ P \mid Q] \mid m[R] \longrightarrow m[n[P \mid Q] \mid R]$	(Red In)
$m[n[out\ m.\ P \mid Q] \mid R] \longrightarrow n[P \mid Q] \mid m[R]$	(Red Out)
$open\ n.\ P \mid n[Q] \longrightarrow P \mid Q$	(Red Open)
$P \longrightarrow Q \Rightarrow (\nu n)P \longrightarrow (\nu n)Q$	(Red Res)
$P \longrightarrow Q \Rightarrow n[P] \longrightarrow n[Q]$	(Red Amb)
$P \longrightarrow Q \Rightarrow P \mid R \longrightarrow Q \mid R$	(Red Par)
$P' \equiv P, P \longrightarrow Q, Q \equiv Q' \Rightarrow P' \longrightarrow Q'$	(Red $\equiv$)
$(x).P \mid \langle M \rangle \longrightarrow P\{x \leftarrow M\}$	(Red Comm)

Table 11: Reduction in the ambient-calculus

2.3.2 The Obliq Programming Language

At the time of writing this report, there is no real language that implements the ambient-calculus[5]. Instead, this section will introduce the Obliq language, which has certain notions of ambient and influenced the design of the ambient calculus.

Obliq[8] is one of the earliest programming languages which support distributed programming. The language was designed before the pervasive of web applications. It only supports simple object model which is a collection of fields. Each field of an Obliq object could be a value (a constant or a procedure), a method, or an alias to an object. An object could either be constructed directly by specifying its fields, or be cloned from other objects.

The four operations which could be performed on objects are:

- selection: a value field of a object could be selected and transmitted over the web. If the selected value is a constant, the value will be transmitted. By contrast, if the selected value is a method, values of its arguments will be transmitted to the remote site where the method is defined, the computation is performed remotely, and the result or an exception is returned to the site of the selection.
- updating: when an updating operation is performed on an remote object, the selected filed is updated to a value that might be sent across the web. If the selected filed is a method, a transmission of method closure is required.
- cloning: cloning an object will yield a new object which contains all fields of argument objects or raise an error if field names of argument objects conflict.
- aliasing: After executing an aliasing method, a.x := **alias** y **of** b **end**, further operations on x of a will be redirected to y of b.

It is important to note that Obliq, as some other languages in the pre-web era, does not distinguish local values from distributed values. By contrast, Waldo etc. [32] pointed out that distinct views must be adopted for local and distributed objects, due to differences in latency, memory access, partial failure, and concurrency.

2.4 The Erlang Language and The OTP Design principles

Erlang [4] is an untyped functional programming language used for constructing concurrent programs. It is widely used in scalable real-time systems. The reliability of Erlang systems is largely attributed to the five OTP design principles, namely, supervision trees, behaviours, applications, releases, and release handling [1]. The following subsections will give a brief introduction to the three principles closely related to distributed computing.

2.4.1 Supervision Trees

Supervision tree is probably the most important concept in the OTP design principle. A supervision tree consists of workers and supervisors. Workers are processes which carry out actual computations while supervisors are processes which inspect a group of workers or sub-supervisors. Since both workers and supervisors are processes and they are organised in a tree structure, the term *child* is used to refer to any supervised process in literature. An example of the supervision tree is presented in Figure 2 [6], where supervisors are represented by squares and workers are represented by circles.

In principle, a supervisor is accountable for starting, stopping and monitoring its child processes according to a list of *child specifications*. A child specification contains 6 pieces of information [1]: i) a internal name for the supervisor to identify the child. ii) the function call to start the child

[5]Cruz and Aguirre [11] proposed a virtual machine for the ambient calculus

[6]This example is cited from OTP Design Principles/Overview. Restart strategies of each supervisor, however, are removed from the figure since they are not related to the central ides discussed here.

process. iii) whether the child process should be restarted after the termination of its siblings or itself. iv) how to terminate the child process. v) whether the child process is a worker or a supervisor. vi) a singleton list which specifies the name of the callback module.

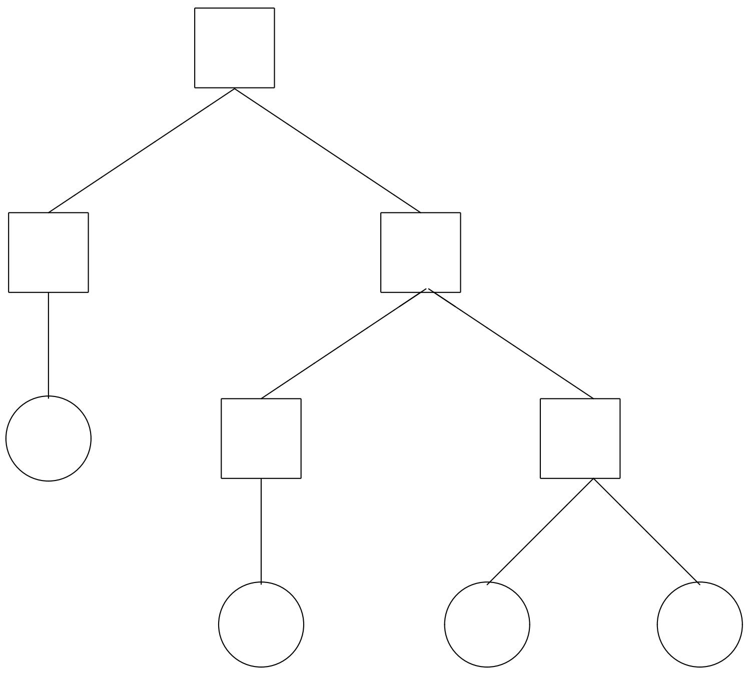


Figure 2: a supervision tree

2.4.2 Behaviours

Behaviours in Erlang, like interface or traits in the objected oriented programming, abstracts common structures and patterns of process implementations. With the help of behaviours, Erlang code can be divided into a generic part, a behaviour module, and a specific part, a callback module. Most processes, including the supervisor in §2.4.1 , could be implemented by realising a set of pre-defined callback functions for one or more behaviours. Although ad-hoc code and programming structures may be more efficient, using consistent general interfaces make code more maintainable and reliable. Standard Erlang/OTP behaviours include: i) *gen_server* for constructing the server of a clientserver paradigm. ii) *gen_fsm* for constructing finite state machines. iii) *gen_event* for implementing event handling functionality. iv) *supervisor* for implementing a supervisor in a supervision tree. Last but not least, users could define their own behaviours in Erlang.

2.4.3 Applications

The OTP platform is made of a group of components called applications. To define an application, users need to annotate the implementation module with "-behaviour(application)" statement and implement the start/2 and the stop/1 functions. Applications without any processes are called library applications. In an Erlang runtime system, all operations on applications are managed by the *application controller* process [7].

This project is concerns in distributed applications which may be used in a distributed system with several Erlang nodes. An Erlang distributed application will be restarted at another node when its current node goes down. A distributed application is controlled by both the application

[7] registered as application_controller

controller and the distributed application controller [8], both of which are part of the *kernel* application. Two configuration parameters must be set before loading and launching a distributed application. First, possible nodes where the distributed application may run must be explicitly pointed. Then, all nodes configured in the last step will be sent a copy of the same configuration which include three parameters: the time for other nodes to start, nodes that must be started in given time, and nodes that can be started in given time.

2.5 Encoding Functions in Process Calculi

Process Calculi are used to describe the structure and behaviour of concurrent processes. As a basis for the design of programming languages, a calculus should be able to encode canonical calculations with functions. Encoding the λ-calculus, the canonical form of functional programming, to a processes calculus could be done either indirectly or directly.

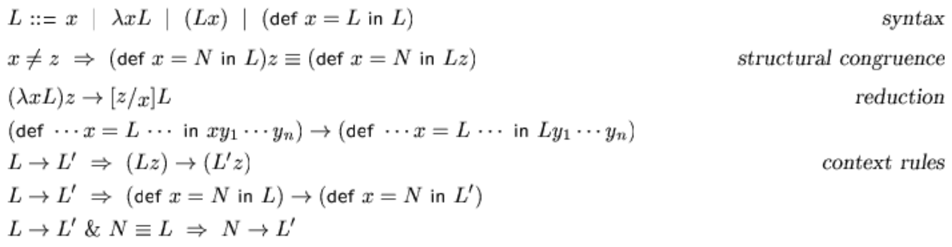
2.5.1 Indirect Encoding

In [20], Milner gave translation rules from both the lazy λ-calculus and the call-by-value λ-calculus to his π-calculus. Based on this work, in an higher-order π-calculus, functions could be encoded into processes and then passed around the network as a value. Later, Sangiorgi pointed out that the higher-order approach is unnecessary since the plain π-calculus could simulate this higher-order feature by passing a name that points to the encoding process[29].

The main drawback of the two indirect encoding approaches is its inefficiency in translation and reduction. Indirect encoding will yield a mass of intermediate variables. Moreover, without a direct representation for functions, complex substitutions of variables for functions are unavoidable during reductions.

2.5.2 The Blue Calculus: Encoding Functions in a Direct Style

Boudol's blue calculus, π^*, is a direct extension of both the λ-calculus and the π-calculus. In fact, Boudol defined two calculi in [7]: a name-passing λ-calculus (λ^*) and an extended π-calculus without summation and matching (π^*). The λ^*-calculus has λ-style syntax and π-style reduction relation (Table 12). It is used as an intermediate language when translating λ-terms to π^*-terms. The π^*-calculus contains primitives from both the λ^*-calculus and the π-calculus so that the translation from both languages is straightforward.



$L ::= x \mid \lambda x L \mid (Lx) \mid (\mathsf{def}\ x = L\ \mathsf{in}\ L)$	*syntax*
$x \neq z \Rightarrow (\mathsf{def}\ x = N\ \mathsf{in}\ L)z \equiv (\mathsf{def}\ x = N\ \mathsf{in}\ Lz)$	*structural congruence*
$(\lambda x L)z \rightarrow [z/x]L$	*reduction*
$(\mathsf{def}\ \cdots x = L \cdots\ \mathsf{in}\ xy_1 \cdots y_n) \rightarrow (\mathsf{def}\ \cdots x = L \cdots\ \mathsf{in}\ Ly_1 \cdots y_n)$	
$L \rightarrow L' \Rightarrow (Lz) \rightarrow (L'z)$	*context rules*
$L \rightarrow L' \Rightarrow (\mathsf{def}\ x = N\ \mathsf{in}\ L) \rightarrow (\mathsf{def}\ x = N\ \mathsf{in}\ L')$	
$L \rightarrow L'\ \&\ N \equiv L \Rightarrow N \rightarrow L'$	

Table 12: The λ^*-calculus

The λ^*-calculus differs from the λ-calculus in two aspects: (i) The argument (N) in an application (MN) must be a variable. (ii) A convenient notation for name declaration, $def\ x\ =\ N\ in\ M$, is allowed. It is important to note that the λ^*-calculus only contains the call-by-name evaluation. This simplifies subsequent studies on relationship between the λ-calculus and other calculi. Translations from λ to λ^* is similar to Launchbury's encoding in [19]:

[8]registered as dist_ac

$$\begin{array}{rcl} x^* & = & x \\ (\lambda x M)^* & = & \lambda x M^* \\ (MN)^* & = & (def\ v = N^*\ in\ (M^* v)) \quad (v \text{ fresh}) \end{array}$$

As mentioned earlier, both the λ^*-calculus and the π-calculus (without summation and matching) could be translated to the π^*-calculus (see Table 14). In addition, a CPS [9] transform from the π^*-calculus to the π-calculus is given in [7] as well. Lastly, Silvano Dal-zilio [12] proposed a implicit polymorphic type sytem for the π^*-calculus as an improvement of the original simple type system in [7].

syntax:				
	P	$::=$	$A \mid D \mid (P \mid P) \mid (\nu x)P$	processes
	A	$::=$	$u \mid (\lambda u)P \mid (Pu)$	agents
	D	$::=$	$\langle u\ =\ P\rangle \mid \langle u\ \Leftarrow P\rangle$	declarations
structural equivalence:				
	$(P \mid Q)$	$\equiv$	$(Q \mid P)$	commutativity
	$((P \mid Q) \mid R)$	$\equiv$	$(P \mid (Q \mid R))$	associativity
	$((\nu u)P \mid Q)$	$\equiv$	$(\nu v)(P \mid Q)\ \ (u\ is\ not\ free\ in\ Q)$	scope migration
	$(P \mid Q)u$	$\equiv$	$(Pu \mid Qu)$	distributivity
	$((\nu u)P)v$	$\equiv$	$(\nu u)(Pv)\ \ (u\ \neq\ v)$	
	Du	$\equiv$	D	
	$\langle u\ =\ P\rangle$	$\equiv$	$\langle u\ \Leftarrow\ (P \mid \langle u\ =\ P\rangle)\rangle$	duplication
reduction:				
	$((\nu u)P)v$	$\rightarrow$	$[v/u]P$	β
	$(u \mid \langle u\ \Leftarrow\ P\rangle)$	$\rightarrow$	P	resource fetching

Table 13: The π^*-Calculus

$[x]u$	$=$	$\overline{x}u$	$[\overline{u}v_1 \cdots v_k]$	$=$	$uv_1 \cdots v_k$
$[\lambda x L]u$	$=$	$u(x,v)[L]v$	$[u(v_1,\cdots,v_k)P]$	$=$	$\langle u\ \Leftarrow\ (\lambda v_1 \cdots v_k)[P]\rangle$
$[Lx]u$	$=$	$(\nu x)([L]v \mid \overline{v}xu)$	$[!u(v_1,\cdots,v_k)P]$	$=$	$\langle u\ =\ (\lambda v_1 \cdots v_k)[P]\rangle$
$[def\ x\ =\ N\ in\ L]u$	$=$	$(vx)([L]u \mid !x(v)[N]v)$	$[P \mid Q]$	$=$	$([P] \mid [Q])$
			$[(\nu u)P]$	$=$	$(\nu u)[P]$

Table 14: Translation from λ^*-calculus and π-calculus to π^*-calculus

2.6 Implementation Strategies

This section will give a short survey on Peter Van Roy's general framework for the implementation of concurrent programming languages [27, 28]. [27] abstracts a four-layer-structure which smoothly applies to four languages designed for different purposes. The four-layer-structure and case studies in [27] are summarized in Table 15. The rest of this section will introduce the main results in [28], which is focused on implementation details.

2.6.1 A General Purpose Virtual Machine

The basic abstract machine in [28] consists of three elements: statement ⟨s⟩, environment E, and assignment store σ. Explanations for concepts related to this VM are given as follows:

[9]continuation passing style

Layer	Erlang	E	Mozart	Oz
strict functional language	Y	Y	Y	Y
deterministic concurrency	N	Y	unknown	Y
asynchronous message passing	Y	Y	Y	Y
global mutable state	Y	N	Y	Y

Table 15: a shared structure for 4 programming languages

- an *assignment store* σ contains a set of variables. Variables could be bound, unbound, or partially bound.
- an *environment* E stores mappings from variable identifiers to entities in σ.
- a *semantic statement* is a pair of $(\langle s \rangle, \mathrm{E})$.
- an *execution state* is a pair of (ST, σ), where ST is a stack of semantic statements.
- a *computation* is a sequence of transitions between execution states.

2.6.2 Extending the Basic VM

The basic VM described in the last section is capable to model most of computation paradigms directly or indirectly. More pleasantly, several restrictions and extensions could be added to this basic VM on demand.

- MST(multiset of semantic stacks). Concurrent programming would be supported via simply replacing the ST by MST. Using the multiset structure for storing semantic stacks gives the flexibility of coordinating identical threads.
- bound or unbound variables. In the book, unbound variables refer to variables which have not been assigned a value. In sequential computing, unbound variables should not be allowed since the program will be halting forever. In concurrent computing, however, unbound variables are acceptable since it might be bound by another thread later.
- single-assignment variable or multiple-assignment variable. Although using single-assignment variable should be encouraged to avoid side-effects, multiple-assignment variable may ease the implementation of imperative languages. Alternatively, both kinds of variable could appear in the same language, such as the Scala language, with different notations.
- variables lifetime. There are three moments in a variables lifetime. i) creation, ii) specification, and iii) evaluation. Adjacent moments may or may not be happen at the same time. Different combinations yields Table 16 cited from [28].
- more kinds of stores. If the state of a shared store could be regarded as the status of a program, evaluation rules on store variables implies features of the programming paradigm. The first three examples in the next subsection demonstrate this idea.
- new statement syntax. When the encoding of certain feature is expensive within the current kernel syntax, adding a new form of statement is one of the easiest way to increase the expressiveness of the language. The gained expressiveness, however, may not be free. For example, the number evaluation rules will be increased dramatically and reasoning about programs will be more complex. Moreover, properties enjoyed by the old model may no longer be hold in the new one.

	sequential with values	sequential with values and dataflow variables	concurrent with values and dataflow variables
eager execution	strict functional programming (e.g., Scheme, ML) (1)&(2)&(3)	declarative model (e.g., Prolog) (1),(2)&(3)	data-driven concurrent model (1),(2)&(3)
lazy execution	lazy functional programming (e.g., Haskell) (1)&(2),(3)	lazy FP with dataflow variables (1),(2),(3)	demand-driven concurrent model (1),(2),(3)

(1): Declare a variable in the store
(2): Specify the function to calculate the variable's value
(3): Evaluate the function and bind the variable

(1)&(2)&(3): Declaring, specifying, and evaluating all coincide
(1)&(2),(3): Declaring and specifying coincide; evaluating is done later
(1),(2)&(3): Declaring is done first; specifying and evaluating are done later and coincide
(1),(2),(3): Declaring, specifying, and evaluating are done separately

Table 16: Popular computation models in languages

2.6.3 Example Paradigms

Following are 5 examples chose from [28]. Each example demonstrates how a popular modern programming feature could be implemented with techniques described in §2.6.2.

- Laziness could be realised by adding a need-predict store.
- Message passing concurrency could be realised by adding a mutable store, together with two primitives NewPort and Send.
- Explicit state could be realised by introducing a mutable cell store. A cell records a mapping from a name value to a store variable. Although a name value might be mapped to different store variables at different time, changes to name-variable mapping will not affect the internal state of a program.
- Relational programming could be realised by adding choice and failure statements to kernel.
- Constraint programming could be realised by adding a constraint store and a batch of primitive operations.

The book [28] provides at least two prominent contributions. Firstly, it provides a framework which unifies most of popular programming paradigms. Secondly, combining different versions of statements, environment and assignment store yields different programming models. To the author of this research proposal, those three components are three dimensions of the "language space". Researchers in the area of programming languages could use this coordinate system is two ways. On the one hand, it provides a convenient formalism to compare different programming languages. On the other hand, regardless of their usages, new programming paradigms could be defined by filling "blank coordinates" of the space. In fact, some models in the book does not correspond to any well known languages.

3 Project Progress and Plans

3.1 Report on Progress

The first academic year was mainly spent on background reading and programming skills enhancing. Besides, I attended lectures and seminars on concurrency and other aspects of informatics. Moreover, I went to several courses of the Transferable Skills Programme, offered by the Informatics Graduate School.

In the first three months(Sep/2010 - Dec/2010), I systematically studied the theoretical aspects and implementation techniques of types and concurrent programming. For theories on types and programming languages, I studied [25] in detail and skimmed main topics in [26]. I learned concurrent programming models and their implementation strategies from [28]. I also scanned recent papers in Concur and PPoPP to develop awareness on topics and trends in the area of concurrent and distributed programming.

During the same period, I did two programming tasks to investigate potential practical issues of concurrent and distributed programming within current main stream technologies. My first task was to write small programs in Scala, Erlang and JoCaml to demonstrate common tasks in a client-server model. I paid particular attention to passing functions between web nodes. After that, I spent a lot of time solving the concordance problem in the Phase I of the SICSA MultiCore Challenge[31]. Although the problem is later proved as IO bounded and limited parallizable, I gained personal benefits from practicing parallel algorithm analysis and implementing parallel algorithms in Scala, JoCaml, Erlang, Haskell, and MPI in C. I attended the full day workshop hosted by the Department of Computer Science, Heriot-Watt University on 13th Dec, 2010.

In the next two months(Dec/2010 - Jan/2011), I focused on the study of process calculi. After summarizing topics in [3], I studied the main results in π-calculus[22, 30], join-calculus[17, 14], ambient calculus[10, 9], fusion calculus[23] and bigraphs[21]. I wrote a short survey on π-calculus, fusion calculus, and join-calculus. Those tasks raised my particular interest in the join-calculus for its programmability. Meanwhile, I studied works on extending languages to support join patterns [6, 18, 24].

During the beginning of 2011(Jan/2011 - April/2011), research directions presented in this proposal became clearer. As a preparing work for further research, I replaced the old Erlang style concurrency in the Links programming language by a join-calculus style one. By doing this, I built a typed join-calculus platform which is under my control.

In early May 2011, my supervisor, Professor Philip Wadler, and I discussed the formal specification of research objectives. We finally agreed on objectives listed in §1.3 and their feasibility. The rest of May was spent on writing the first draft of this report.

In the last two months(Jun/2011 - July/2011) , I reviewed distribution, failure, notification, and exception handling models in both the join calculus and its different implementations. Meanwhile, I proposed a more straightforward implementation for realising join patterns, called neural network. I compared its efficiency with FSM-based and extractor-based implementations. Although the straightforward implementation is more efficient than the extractor-based implementation in theory, the improved extractor-based implementation (Appendix A) turns to be a more adequate library for its simple syntax and reasonable efficiency.

3.2 Plan for the Second Year

The second year will be dedicated to working on the first and second objectives in order. The first a few months will be spent on building a library that support the OTP design principles in a typed language. A suitable host platform would be my Scala library which supports join patterns. After that, a technical report will be produced to document the designed library. The report will cover (i) the functionality provided by the library with moderate size examples; (ii) alternative choices of library design and reasons for my decisions; (iii) benchmark results of selected canonical distributed applications. Later, a formalised model will be defined and verified. The formalism will be based on the join-calculus. The criteria for model verification will include both correctness and reliability.

3.3 Plan for the Third Year

The third year will be dedicated to working on the last objective (extend the model with primitives from other models). The purpose of this work is to demonstrate the extensibility of proposed model. New primitives will be added only when it will significantly reduce the effort of encoding some tasks (e.g. sending function closure). Finally, a dissertation as planed in §1.4 will be produced.

3.4 Potential Risks

As a novel and ambitious research, some tasks may be tougher than my expectation. Also, results of some tasks may differ from my intuition. Risk analyses for each objective are given as follows:

- The difficulty of the first objective depends on how specific host language consist with the OTP design principles. Both Scala and F# support multi-paradigm programming, which provides more ways for library implementation. Besides, the large friendly communities of those two languages will be the back support when encountering difficulties related to language details. I prefer to start the work in Scala because my join library built in that language overcomes some limitations found in other current platforms which support join patterns. Nevertheless, F# is still a good alternative platform if the Scala language turns to not well support the OTP design principles at a certain point. Moreover, the Links languages would be another backup platform which is completely under my control.
- For the second objective, it is not clear what changes should be made to the join-calculus. As stated in early sections, new features will be added only when the old model is incapable to support crucial concepts in the OTP design principles. It is possible that the join-calculus may be consistent with all principles without making any changes. Regardless of whether new model should be defined or not, it would be a novelty of this project to formalise an approach to evaluate distributed programming models.
- Decisions of adding new primitives to the join-calculus might be subjective to some extent. Therefore, criteria for evaluating the new model should be specified in advance. To be more objective, arguments given in this task should be supported by empirical results.

To the best of my knowledge, none of above risks should be the principle hindrance of achieving the three objectives listed in §1.3. With proper plans and the commitment to each objective, the proposed research should be suitable for a PhD project.

A Scala Join (version 0.2) User Manual

The Scala Join Library (version 0.2) improves the scala joins library [18] implemented by Dr. Philipp Haller. In deed, this library inherited its syntax and main implementation strategy for local join calculus. Main advantages of this library are: (i) uniform *and* operator, (ii) theoretically unlimited number of join patterns in a single block (iii) clear typed *unapply* method in extractor objects (iv) simpler structure for the **Join** class, and most importantly, (v) supporting communication on distributed join channels.

A.1 Using the Library

A.1.1 Sending Messages via Channels

An elementary operation in the join calculus is sending a message via a channel. A channel could be either asynchronous or synchronous. At caller's side, sending a message via an asynchronous channel has no obvious effect in the sense that the program will proceed immediately after sending the message. By contrast, when a message is sent via a synchronous channel, the current thread will be suspended until a result is returned.

To send a message m via channel c, users simply apply the message to the channel by calling $c(m)$. For the returned value of a synchronous channel, users may want to assign it to a variable so that it could be used later. For example,

```
val v = c(m) // c is a synchronous channel
```

A.1.2 Grouping Join Patterns

A join definition defines channels and join patterns. Users defines a join definition by initializing the **Join** class or its subclass. It is often the case that a join definition (including channels and join patterns defined inside) should be globally *static*. If this is the case, it is a good programming practice in Scala to declare the join definition as a *singleton object* with following idiom:

```
object join_definition_name extends Join{
  //channel declarations
  //join patterns declaration
}
```

A channel is a singleton object inside a join definition. It extends either class **AsyName**[**ARG**] or class **SynName**[**ARG**, **R**], where **ARG** and **R** are generic type parameters in Scala. Here, **ARG** indicates the type of channel parameter whereas **R** indicates the type of return value of a synchronous channel. The current library only supports unary channels, which only take one parameter. Fortunately, this is sufficient for constructing nullary channels and channels that take more than one parameters. A nullary channel could be edcoded as a channel whose argument is always **Unit** or other constant. For channel that takes more than one parameters, users could pack all arguments in a tuple.

Once a channel is defined, we can use it to define a join pattern in the following form

```
case <pattern> => <statements>
```

The <*pattern*> at the left hand side of ⇒ is made of channels and their formal arguments, connected by the infix operator *and*. The <*statements*> at the right hand side of ⇒ is a sequence of Scala statements. Formal arguments declared in the <*pattern*> must be pairwise distinguished and might be used in the <*statements*> part. In addition, each join definition accepts one and only one group of join pattens as the argument of its *join* method. Lastly, like most implementations for the join calculus, the library does not permit multiple useages of the same channel in a single join pattern. On the other side, using the same channel in arbitrary number of different patterns is allowed.

We conclude this section by presenting a sample code that defines a join pattern.

Listing 1: Example code for defining channels and join patterns (join_test.scala)

```
import join._
import scala.concurrent.ops._ // spawn
object join_test extends App{//for scala 2.9.0
  object myFirstJoin extends Join{
    object echo extends AsyName[String]
    object sq extends SynName[Int, Int]
    object put extends AsyName[Int]
    object get extends SynName[Unit, Int]

    join{
      case echo(str) => println(str)
      case sq(x) => sq reply x*x
      case put(x) and get(_) => get reply x
   }
  }

  spawn {
    val sq3 = myFirstJoin.sq(3)
    println("square (3) = "+sq3)
  }
  spawn { println("get: "+myFirstJoin.get()) }
  spawn { myFirstJoin.echo("Hello World") }
  spawn { myFirstJoin.put(8) }
}
```

One possible result of running above code is:

```
>scalac join_test.scala
>scala join_test
square (3) = 9
Hello World
get: 8
```

A.1.3 Distributed Computation

With the distributed join library, it is easy to construct distributed systems on the top of a local system. This section explains additional constructors in the distributed join library by looking into the code of a simple client-server system, which calculates the square of an integer on request. The server side code is given at Listing 2 and the client side code is given at Listing 3.

Listing 2: Server.scala

```
import join._
object Server extends App{
  val port = 9000
  object join extends DisJoin(port, 'JoinServer){
    object sq extends SynName[Int, Int]
    join{     case sq(x) => println("x:"+x); sq reply x*x }
    registerChannel("square", sq)
  }
  join.start()
}
```

Listing 3: Client.scala

```
object Client{
  def main(args: Array[String]) {
    val server = DisJoin.connect("myServer", 9000, 'JoinServer)
    //val c = new DisSynName[Int, String]("square", server)
    //java.lang.Error: Distributed channel initial error:
            Channel square does not have type Int => java.lang.String.
    //...
    val c = new DisSynName[Int, Int]("square", server)//pass
    val x = args(0).toInt
    val sqr = c(x)
    println("sqr("+x+") = "+sqr)
    exit()
}}
```

```
> scala ServerTest
Server 'JoinServer Started...
                                                    >scala Client 5
x:5
                                                    sqr(5) = 25
                                                    >scala Client 7
x:7
                                                    sqr(7) = 49
```

In Server.scala, we constructed a distributed join definition by extending class **DisJoin**(**Int**, **Symbol**). The integer is the port number where the join definition is going to listen and the symbol is used to identify the join definition. The way to declare channels and join patterns in **DisJoin** is the same as the way in **Join**. In addition, we register channels which might be used at remote site with a memorizable string. At last, different from running a local join definition, a distributed join definition has to be explicitly started.

In Client.scala, we connect to the server by calling DisJoin.connect. The first and second arguments are the hostname and port number where the remote join definition is located. The last argument is the name of the distributed join definition. The hostname is a String which is used for local name server to resolve the IP address of a remote site. The port number and the name of join definition should be exactly the same as the specification of the distributed join definition.

Once the distributed join definition, server, is successfully connected, we can initialize distributed channels as an instance of DisAsyName[ARG](channel_name, server) or DisSynName[ARG, R](channel_name, server). Using an unregistered channel name or declaring a distributed channel whose type is inconsistent with its referring local channel will raise a run-time exception as soon as the distributed channel is initialised. In later parts of program, we are free to use distributed channels to communicate with the remote server. The way to invoke distributed channels and local channels are same.

A.2 Implementation Details

A.2.1 Case Statement, Extractor Objects and Pattern Matching in Scala

In Scala, a partial function is a function with an additional method: $isDefinedAt$, which will return $true$ if the argument is in the domain of this partial function, or $false$ otherwise. The easiest way to define a partial function is using the case statement. For example,

```
scala> val myPF : PartialFunction[Int,String] = {
     |  case 1 => "myPF apply 1"
     | }
myPF: PartialFunction[Int,String] = <function1>

scala> myPF.isDefinedAt(1)
res1: Boolean = true

scala> myPF.isDefinedAt(2)
res2: Boolean = false

scala> myPF(1)
res3: String = myPF apply 1

scala> myPF(2)
scala.MatchError: 2
       at $anonfun$1.apply(<console>:5)
       ...
```

In addition to instances of a numerate class and case classes, the value used between *case* and $\Rightarrow$ could also be an instance of an *extractor object*, object that contains an *unapply* method[13]. For example,

```
scala> object Even {
     |   def unapply(z: Int): Option[Int] = if (z%2 == 0) Some(z/2) else None
     | }
defined module Even

scala>  42 match { case Even(n) => Console.println(n) } // prints 21
21

scala>  41 match { case Even(n) => Console.println(n) } // prints 21
scala.MatchError: 41
       ...
```

In the above example, when a value, say x, attempts to match against a pattern, $Even(n)$, the method $Even.unapply(x)$ is invoked. If $Even.unapply(x)$ returns $Some(v)$, then the formal argument n will be assigned with the value v and statements at the right hand side of $\Rightarrow$ will be executed. By contrast, if $Even.unapply(x)$ returns $None$, then the current case statement is considered not matching the input value, and the pattern examination will move towards the next case statement. If the last case statement still does not match the input value, then the whole partial function is not defined for the input. Applying a value outside the domain of a partial function will rise a *MatchError*.

A.2.2 Implementing Local Channels

Both asynchronous channel and synchronous channel are subclass of trait *NameBase*. The reason why we introduced this implementation free trait is that, although using generic types to restrict the type of messages pending on a specific channel is important for type safety, as we will see later, a unified view for asynchronous and synchronous channels would simplify the implementation at many places.

Listing 4: Code defines local asynchronous channel

```
class AsyName[Arg](implicit owner: Join, argT:ClassManifest[Arg]) extends NameBase{
  var argQ = new Queue[Arg] //queue of arguments pending on this name

  def apply(a:Arg) :Unit = synchronized {
    val isNewMsg = argQ.isEmpty
    argQ += a
    if (isNewMsg){ owner.trymatch() }
  }
  //other code
}
```

Asynchronous channel is implemented as Listing 4. The implicit argument *owner* is the join definition where the channel is defined. The other implicit argument, **argT**, is the descriptor for the run time type of **Arg**. Although **argT** is a duplicate information for **Arg**, it is important for distributed channels, whose erased type parameter might be declared differently between different sites. We postponed this problem until §A.2.4.

As shown in the above code, an asynchronous channel contains an argument queue whose element must has generic type **Arg**. Pending a message on this channel is achieved by calling the *apply* method, so that c(m) could be wrote instead of c.apply(m) for short in Scala. When a message is sent via a channel, it is added to the end of the argument queue. Based on the linear assumption that no channel should appear more than once in a join pattern, reduction is possible only when the number of pending messages on a channel is increased from 0.

Listing 5: Code defines local synchronous channel

```
class SynName[Arg, R](implicit owner: Join, argT:ClassManifest[Arg], resT:ClassManifest[R])
 extends NameBase{
  var argQ = new Queue[Arg]
  var value = new SyncVar[R]

  def apply(a:Arg) :R = synchronized {
    val isNewMsg = argQ.isEmpty
    argQ += a
    if (isNewMsg){ owner.trymatch() }
    fetch
  }

  def reply(r:R){
    assert(!value.isSet, "Internal Error: SyncVar has been set")
    value.put(r)
  }

  private def fetch():R={
    value.take
  }

  //other code
}
```

As listing 5 shows, synchronous channel is similar to asynchronous channel but has more features. Firstly, other processes could send a reply value to a synchronous channel. Secondly, sending a message to a synchronous channel will return a value rather than proceed to the next step of the program. Both features are realised by using a synchronous variable (SynVar), provided by the Scala standard library.

A.2.3 Implementing the Join Patterns Using Extractor Objects

In this library, join patterns are represented as a partial function which has type: (Set[NameBase], Queue[(NameBase, Any)]) $\Rightarrow$ Unit. The trymatch method tries to fire a satisfied pattern, or to restore the system if no pattern is matched at all. The meaning of the arguments will be explained when we look into the details of the unapply method of channels.

Listing 6: Code defines the Join class

```
class Join {
   private var hasDefined = false

   implicit val joinsOwner = this
   private var joinPat: PartialFunction[(Set[NameBase],Boolean), Unit] = _

   def join(joinPat: PartialFunction[(Set[NameBase],Boolean), Unit]) {
    if(!hasDefined){
      this.joinPat = joinPat
      hasDefined = true
    }else{ throw new Exception("Join definition has been set for"+this) }
   }

  def trymatch() = synchronized {
    var names: Set[NameBase] = new HashSet
    var queue = new Queue[(NameBase, Any)]
    try{
      joinPat((names, queue))
    }catch{
      case e:Exception => queue.foreach{ case (c, a) => c.pendArg(a)} // no pattern is matched
    }
  }
}
```

To investigate how extractor objects is used to implement join patterns, first consider a join pattern that only contains a single channel, for example,

```
  case ch(msg) => println("received message "+msg+" from channel "+ch)
```

To examine if this case statement is matched, the library tests if the argument queue of the channel *ch* is empty. If the argument queue is not empty, the unapply method should return the head element of the queue, otherwise the test fails.

Examining a pattern where more than one channel is involved will require more tasks. Firstly, to examine the linearity of each pattern, we use a set to records examined channels in current pattern. Clearly, adding the same channel to the set more than once suggested that the current pattern is not satisfy the linearity requirement. Secondly, since the presence of message on a single channel does not imply the success of the current pattern, a log is required to record examined channels and their popped messages. In the case that a pattern is partly matched, all popped messages should be appended to their original channels.

The last thing is to define an *and* constructor which combines two or more channels in a join pattern. Indeed, this is surprisingly simple to some extent. In the library, *and* is defined as a binary operator which passes the same argument to both operands.

Listing 7: The unapply method in local asynchronous and synchronous channel

```
  def unapply(attr:(Set[NameBase], Queue[(NameBase, Any)])) : Option[Arg]= attr match{
    case (set, ch_arg_q) =>
      assert(!set.contains(this), "\n join pattern non-linear")
      set += this
      if (argQ.isEmpty){//no message is pending on this channel, pattern match fails
        set.clear              // clear the set
        ch_arg_q.foreach{ case (c, a) => c.pendArg(a)} // pending messages back
        None        // signal the failure of pattern matching
      }else{
        val arg = argQ.dequeue
        ch_arg_q += ((this, arg))
        Some (arg)
      }
  }
```

Listing 8: Code defines the and object

```
object and{
  def unapply(attr:(Set[NameBase], Queue[(NameBase, Any)])) = {
      Some(attr,attr)
  }
}
```

A.2.4 Implementing Distributed Join Calculus

The **DisJoin** class extends both the **Join** class, which supports join definitions, and the **Actor** traits, which enables distributed communication. In addition, the distributed join definition manages a name server which maps a string to one of its channels. Comparing to a local join definition, a distributed join definition has two additional tasks: checking if distributed channels used at a remote sites are annotated with correct types and listening messages pending on distributed channels. The code of the **DisJoin** class is presented in Listing 9.

Listing 9: Code defines the DisJoin Class

```
class DisJoin(port:Int, name: Symbol) extends Join with Actor{
  var channelMap = new HashMap[String, NameBase] //work as name server

  def registerChannel(name:String, ch:NameBase){
    assert(!channelMap.contains(name), name+" has been registered.")
    channelMap += ((name, ch))
  }

  def act(){
    RemoteActor.classLoader = getClass().getClassLoader()
    alive(port)
    register(name, self)//
    println("Server "+name+" Started...")

    loop(
      react{
        case JoinMessage(name, arg:Any) => {
          if (channelMap.contains(name)) {
            channelMap(name) match {
              case n : SynName[Any, Any] => sender ! n(arg)
              case n : AsyName[Any] => n(arg)
         }}}
        case SynNameCheck(name, argT, resT) => {
          if (channelMap.contains(name)) {
            sender ! (channelMap(name).argTypeEqual((argT,resT)))
          }else{
            sender ! NameNotFound
          }
        }
        case AsyNameCheck(name, argT) => {
          if (channelMap.contains(name)) {
            sender ! (channelMap(name).argTypeEqual(argT))
          }else{
            sender ! NameNotFound
        }}}
    )
}}
```

In this library, distributed channel is indeed a stub of a remote local channel. When a distributed channel is initialized, its signature is checked at the place where its referring local channel is defined. Later, when a message is sent through this distributed channel, the message and the channel name is forwarded to the remote join definition where the referring local channel is defined. Consistent with the semantic of distributed join calculus, reduction, if any, is performed at the location where the join pattern is defined. If the channel is a distributed synchronous channel, a reply value will be send back to the remote caller. Listing 10 illustrates how distributed synchronous channel is implemented. Distributed asynchronous channel is implemented in a similar way.

Listing 10: Code defines distributed synchronous channel

```
class DisSynName[Arg:Manifest, R:Manifest](n:String, owner:scala.actors.AbstractActor){
  val argT = implicitly[ClassManifest[Arg]]//type of arguments
  val resT = implicitly[ClassManifest[R]]//type of return value

  initial()// type checking etc.

  def apply(arg:Arg) :R = synchronized {
    (owner !? JoinMessage(n, arg)).asInstanceOf[R]
  }

  //check type etc.
  def initial() = synchronized {
    (owner !? SynNameCheck(n, argT, resT)) match {
      case true => Unit
      case false => throw new Error("Distributed channel initial error:"+
                                    "Channel " + n + " does not have type "+
                                    argT+ " => "+resT+".")
      case NameNotFound => throw new Error("name "+n+" is not found at "+owner)
}}}
```

Lastly, the library also provides a function that simplifies the work of connection to a distributed join definition.

```
object DisJoin {
  def connect(addr:String, port:Int, name:Symbol):AbstractActor = {
    val peer = Node(addr, port)//location of the server
    RemoteActor.select(peer, name)
  }
}
```

References

[1] Ericsson AB. Otp design principles user's guide, 2011.

[2] S. Abramsky. What are the fundamental structures of concurrency? we still don't know! In *Algebraic process calculi: the first 25 years and beyond*, BRICS Notes Series NS-05-03, pages 1–5, June 2005.

[3] Luca Aceto and Andrew D. Gordon, editors. *Short Contributions from the Workshop on Algebraic Process Calculi: The First Twenty Five Years and Beyond*, 2005.

[4] Joe Armstrong. *Programming Erlang: Software for a Concurrent World.* Pragmatic Bookshelf, 2007.

[5] J. C. M. Baeten. A brief history of process algebra. Technical report, Theor. Comput. Sci, 2004.

[6] Nick Benton, Luca Cardelli, and Cédric Fournet. Modern concurrency abstractions for c#. *ACM Trans. Program. Lang. Syst.*, 26:769–804, September 2004.

[7] Grard Boudol. The π-calculus in direct style, 1997.

[8] Luca Cardelli. Obliq: A language with distributed scope. *Computing Systems*, 8:27–59, 1995.

[9] Luca Cardelli and Andrew D. Gordon. Types for mobile ambients. In *In Proc. 26th POPL*, pages 79–92. ACM Press, 1999.

[10] Luca Cardelli and Andrew D. Gordon. Mobile ambients. *Theor. Comput. Sci.*, 240(1):177–213, 2000.

[11] Luis Rodrigo Gallardo Cruz and Oscar Olmedo Aguirre. A virtual machine for the ambient calculus. In *Electrical and Electronics Engineering, 2005 2nd International Conference on*, pages 56–59, 2005.

[12] Silvano Dal-Zilio. Implicit polymorphic type system for the blue calculus. Technical report, 1997.

[13] Burak Emir, Martin Odersky, and John Williams. Matching objects with patterns. In *ECOOP 2007 Object-Oriented Programming, volume 4609 of LNCS*, pages 273–298. Springer, 2007.

[14] Cédric Fournet, Fabrice Le Fessant, Luc Maranget, and Alan Schmitt. Jocaml: A language for concurrent distributed and mobile programming. In *In Advanced Functional Programming*, pages 129–158. Springer Verlag, 2003.

[15] Cédric Fournet and Georges Gonthier. The join calculus: A language for distributed mobile programming. In *In Proceedings of the Applied Semantics Summer School (APPSEM), Caminha*, pages 268–332. Springer-Verlag, 2000.

[16] Cédric Fournet, Georges Gonthier, Jean-Jacques Lévy, Luc Maranget, and Didier Rémy. A calculus of mobile agents. In *Proceedings of the 7th International Conference on Concurrency Theory*, CONCUR '96, pages 406–421, London, UK, 1996. Springer-Verlag.

[17] Cédric Fournet, Georges Gonthier, and Inria Rocquencourt. The reflexive cham and the join-calculus. In *In Proceedings of the 23rd ACM Symposium on Principles of Programming Languages*, pages 372–385. ACM Press, 1995.

[18] Philipp Haller and Tom Van Cutsem. Implementing joins using extensible pattern matching. In *Proceedings of the 10th international conference on Coordination models and languages*, COORDINATION'08, pages 135–152, Berlin, Heidelberg, 2008. Springer-Verlag.

[19] John Launchbury. A natural semantics for lazy evaluation. pages 144–154. ACM Press, 1993.

[20] Robin Milner. Functions as processes. In *Proceedings of the 17th International Colloquium on Automata, Languages and Programming*, ICALP '90, pages 167–180, London, UK, 1990. Springer-Verlag.

[21] Robin Milner. *The Space and Motion of Communicating Agents.* Cambridge University Press, New York, NY, USA, 2009.

[22] Robin Milner, J. Parrow, and D. Walker. A calculus of mobile processes. *Journal of Information and Computation*, 100:1–77, 1992.

[23] Joachim Parrow and Bjrn Victor. The fusion calculus: Expressiveness and symmetry in mobile processes (extended abstract). In *LICS'98*, pages 176–185. IEEE Computer Society Press, 1998.

[24] Tomas Petricek. Reactive programming with events. Master's thesis, Charles University in Prague, 2010.

[25] Benjamin C. Pierce. *Types and Programming Languages.* MIT Press, 2002.

[26] Benjamin C. Pierce, editor. *Advanced Topics in Types and Programming Languages.* MIT Press, 2005.

[27] Peter Van Roy. Convergence in language design: a case of lightning striking four times in the same place. flops. In *in the Same Place, 8th International Symposium on Functional and Logic Programming (FLOPS 2006), April 2006, Springer LNCS*, pages 2–12. Springer, 2006.

[28] Peter Van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming.* MIT Press, Cambridge, MA, USA, 2004.

[29] Davide Sangiorgi. From pi-calculus to higher-order pi-calculus - and back. In *Proceedings of the International Joint Conference CAAP/FASE on Theory and Practice of Software Development*, TAPSOFT '93, pages 151–166, London, UK, 1993. Springer-Verlag.

[30] Davide Sangiorgi and David Walker. *The π-Calculus: A Theory of Mobile Processes.* Cambridge University Press, New York, NY, USA, 2001.

[31] SICSA theme on Complex System Engineering. Sicsa multicore challenge: Phase i, December 2010.

[32] Jim Waldo, Geoff Wyant, Ann Wollrath, and Samuel C. Kendall. A note on distributed computing. In *Selected Presentations and Invited Papers Second International Workshop on Mobile Object Systems - Towards the Programmable Internet*, pages 49–64, London, UK, 1997. Springer-Verlag.