

Implementing the Distributed Join Calculus using Actors

Bachelorarbeit

Matthias Heinzl

Matrikelnummer: 359143

Technische Universität Berlin

8. März 2018

Gutachter:

Prof. Dr. Uwe Nestmann

Prof. Dr. Odej Kao

Eidesstattliche Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und eigenhändig sowie ohne unerlaubte fremde Hilfe und ausschließlich unter Verwendung der aufgeführten Quellen und Hilfsmittel angefertigt habe.

Berlin, den 8. März 2018

Matthias Heinzel

Zusammenfassung

Der Join-Kalkül ist eine besser verteilbare Variante des Pi-Kalküls und besitzt eine verteilte Erweiterung mit expliziten Lokalitäten und Migration. Auch wenn diese Eigenschaften vielversprechend für den Einsatz in einem praxis-orientierten Umfeld sind, erfordert die Implementierung von Features wie maschinen-übergreifender Kommunikation und Migration einen hohen Aufwand. Daher untersuchen wir die Möglichkeit, existierende Infrastruktur in aktorenbasierten Systemen zu verwenden.

Wir entwickeln eine verteilbarkeitserhaltende Übersetzung des vereinfachten Core-Join-Kalküls in den Kalkül $A\pi$ (einen typisierten Pi-Kalkül, der das Aktorenmodell repräsentiert) und erstellen daraus die Implementierungen des Join-Kalküls in der aktorenbasierte Programmiersprache Erlang. Der Prototyp wird anschließend um die Primitive des Verteilten Join-Kalküls, einschließlich Migration, erweitert. Durch die Nutzung von Erlang/OTP als Grundlage erhalten wir eine überschaubare und einfache Implementierung.

Abstract

The join-calculus is a better distributable variant of the pi-calculus, with a distributed extension featuring explicit locations and migration. While these properties are promising for use in a practical setting, the implementation of features such as inter-node communication and migration is an ambitious effort. Therefore, we explore the possibility of using existing infrastructure of actor-based systems.

We develop a distributability-preserving encoding from the core join-calculus into $A\pi$ (a typed pi-calculus representing the actor model) and use it to derive an implementation of the join-calculus as a thin layer on top of the actor-based programming language Erlang. Our prototype is then extended to support the primitives of the distributed join-calculus including migration. By using Erlang/OTP as a foundation, we obtain a small and simple implementation.

Contents

Contents	ix
1. Introduction	1
2. Technical Preliminaries	2
2.1. The Join-Calculus	2
2.1.1. The Core Join-Calculus	4
2.1.2. The Distributed Join-Calculus	5
2.2. The Actor Model	6
2.2.1. The Actor Calculus $A\pi$	6
3. Encoding	12
3.1. A Naive Approach	12
3.2. Forwarding	12
3.3. Storing Payloads	13
3.4. Our Encoding	13
3.4.1. Type Correctness	14
3.4.2. Semantic Correctness	15
4. Implementation	20
4.1. Basic Translation	20
4.2. Distributed Implementation	23
4.2.1. Migrating Actors	23
4.2.2. Locations	25
4.2.3. Primitives for Distribution	25
4.2.4. Connecting Nodes	27
5. Conclusion	30
5.1. Future Work	30
Bibliography	31
A. Source Code	33

1. Introduction

Process calculi allow a formal, abstract description of concurrent and distributed systems to simplify mathematical reasoning. In this setting, we are often interested in the *degree of distributability* of a term, which can be intuitively describes as the number of parallel components that can execute independently. However, when implementing these calculi in practice, it is often not possible to preserve their degree of distributability. In the pi-calculus [1], for example, the problem arises from the possible existence of multiple senders and multiple receivers on a channel. A process wanting to send a message to one of many possible receivers has to make sure the receiver did not already receive a messages from somewhere else in the meantime. In this way, it can become necessary to solve a global consensus problem for the correct execution of basic communication primitives.

As presented by Peters et al. [2], this lack of distributability in the asynchronous pi-calculus can be shown by the existence of certain synchronization patterns. At the same time, the *join-calculus* [3] is identified as a calculus that is more distributable. And indeed, one of its main ideas is to enforce that all receivers on a channel reside in the same location (*locality*). It also explores the addition of a notion of locations, which allows it to not only be *distributable*, but also express explicitly *distributed* processes and migration [4]. With these features, it is well-suited to be a practical framework for writing distributed and concurrent programs.

There exist several implementations of the concept of join-patterns^{1 2}, but the only one making use of the distribution primitives is an unmaintained beta version of JoCaml³ that makes heavy modifications to the original OCaml compiler and runtime system [5]. This sacrifices binary compatibility and makes it difficult to incorporate upstream changes. Later versions of JoCaml⁴ dropped support for locations and related primitives.

If it was possible to rely on existing infrastructure, much effort could be avoided. For instance, there are multiple implementations of the actor model providing support for distributed communication and code migration. Also, actors are freely distributable, which should make it possible to find a distributability-preserving translation from the join-calculus. This thesis aims to find such an encoding and, going from there, develop a prototype implementation of the distributed join-calculus.

¹<https://www.microsoft.com/en-us/research/project/comega/>

²<https://github.com/Chymyst/chymyst-core>

³<http://moscova.inria.fr/oldjocaml/index.shtml>

⁴<http://jocaml.inria.fr/>

2. Technical Preliminaries

2.1. The Join-Calculus

Syntax

We use the join-calculus as described by Fournet et al. [4] and Levy [6]. Assuming an infinite set of channel names $\mathcal{N} = \{x, y, z, u, v, \kappa, \dots\}$, we use $\tilde{x}, \tilde{y}, \dots$ for tuples of names and define the syntax as follows.

processes	
$P, Q ::= x\langle\tilde{v}\rangle$	emission of \tilde{v} on x
$\quad \mid \mathbf{def} \ D \ \mathbf{in} \ P$	definition of D in P
$\quad \mid P Q$	parallel composition
$\quad \mid \mathbf{0}$	empty process

definitions	
$D, E ::= J \triangleright P$	elementary clause
$\quad \mid D \wedge E$	simultaneous definitions
$\quad \mid \top$	empty definition

join-patterns	
$J, K ::= x\langle\tilde{v}\rangle$	reception of \tilde{v} on x
$\quad \mid J K$	composed join-pattern

The simplest possible process is the empty process $\mathbf{0}$. Also, we can send a tuple of names \tilde{v} on a channel x and compose any two processes P and Q in parallel. The most interesting construct is the definition $\mathbf{def} \ D \ \mathbf{in} \ P$, which persistently defines a number of elementary clauses. Each elementary clause $J \triangleright R$ consists of a join-pattern J , composed from receptions $x_1\langle\tilde{v}_1\rangle \mid \dots \mid x_n\langle\tilde{v}_n\rangle$ and a process R . When a message is available on each channel x_1, \dots, x_n , they can be consumed atomically, which binds them to the variables $\tilde{v}_1, \dots, \tilde{v}_n$ and starts the process R . The defined names x_1, \dots, x_n are available in both D and P (so they can be used recursively). Note that all received names in a join-pattern must be distinct, making both patterns $x\langle u, u \rangle$ and $x\langle u \rangle | y\langle u \rangle$ invalid.

We will now define the scopes of received, defined and free variables (rv, dv, fv).

$$\begin{aligned}
\text{rv}(x\langle\tilde{v}\rangle) &= \{u \in \tilde{v}\} \\
\text{rv}(J|K) &= \text{rv}(J) \uplus \text{rv}(K) \\
\\
\text{dv}(x\langle\tilde{v}\rangle) &= \{x\} \\
\text{dv}(J|K) &= \text{dv}(J) \cup \text{dv}(K) \\
\\
\text{dv}(J \triangleright P) &= \text{dv}(J) \\
\text{dv}(D \wedge E) &= \text{dv}(D) \cup \text{dv}(E) \\
\text{dv}(\top) &= \emptyset \\
\\
\text{fv}(J \triangleright P) &= \text{dv}(J) \cup (\text{fv}(P) - \text{rv}(J)) \\
\text{fv}(D \wedge E) &= \text{fv}(D) \cup \text{fv}(E) \\
\text{fv}(\top) &= \emptyset \\
\\
\text{fv}(x\langle v \rangle) &= \{x\} \cup \{u \in \tilde{v}\} \\
\text{fv}(\mathbf{def} \ D \ \mathbf{in} \ P) &= (\text{fv}(P) \cup \text{fv}(D)) - \text{dv}(D) \\
\text{fv}(P|Q) &= \text{fv}(P) \cup \text{fv}(Q) \\
\text{fv}(\mathbf{0}) &= \emptyset
\end{aligned}$$

Semantics

The semantics of the join-calculus is described using the reflexive chemical abstract machine (RCHAM) [3]. An RCHAM configuration $\mathcal{R} \vdash \mathcal{M}$ consists of a multiset of active definitions \mathcal{R} , called “reactions”, and a multiset of active processes \mathcal{M} , called “molecules”.

There are multiple reversible heating/cooling rules \rightleftharpoons , corresponding to the structural congruence between processes, and a single reduction rule \longrightarrow . In each of these rules, there is an implicit unchanged context of reactions and molecules which we omit for brevity.

$$\begin{array}{llll}
(\text{str-join}) & \vdash P|Q & \rightleftharpoons & \vdash P, Q \\
(\text{str-null}) & \vdash \mathbf{0} & \rightleftharpoons & \vdash \\
(\text{str-and}) & D \wedge E \vdash & \rightleftharpoons & D, E \vdash \\
(\text{str-nodef}) & \top \vdash & \rightleftharpoons & \vdash \\
(\text{str-def}) & \vdash \mathbf{def} \ D \ \mathbf{in} \ P & \rightleftharpoons & D\sigma_{\text{dv}} \vdash P\sigma_{\text{dv}} \\
\\
(\text{red}) & J \triangleright P \vdash J\sigma_{\text{rv}} & \longrightarrow & J \triangleright P \vdash P\sigma_{\text{rv}}
\end{array}$$

In (str-dev), the substitution σ_{dv} instantiates the defined variables $dv(D)$ to distinct, fresh names that are not free in any reaction or molecule in the configuration. In (red), σ_{rv} simply substitutes the received variables $rv(J)$. Note that (red) makes use of the syntactic similarity between join-patterns and parallel emission of messages in processes.

To apply a reduction, we usually first need to dissolve processes using the heating rules until we can apply (red). Afterwards, the solution can be cooled down again (see example 1).

We can obtain a reduction relation \longrightarrow_J on processes (instead of RCHAM configurations) by defining for any two processes P and P' :

$$P \longrightarrow_J P' \text{ iff } \emptyset \vdash P \rightleftharpoons^* \longrightarrow \rightleftharpoons^* \emptyset \vdash P'$$

Example 1 We can follow the reduction of a simple process in the RCHAM.

$$\begin{aligned}
& \{\} \vdash \{\mathbf{def} \ x\langle u \rangle | y\langle v \rangle \triangleright P \text{ in } x\langle a \rangle | x\langle b \rangle | y\langle c \rangle\} \\
\rightleftharpoons & \{x\langle u \rangle | y\langle v \rangle \triangleright P\} \vdash \{x\langle a \rangle | x\langle b \rangle | y\langle c \rangle\} & (\text{str-def}) \\
\rightleftharpoons & \{x\langle u \rangle | y\langle v \rangle \triangleright P\} \vdash \{x\langle a \rangle, x\langle b \rangle | y\langle c \rangle\} & (\text{str-join}) \\
\longrightarrow & \{x\langle u \rangle | y\langle v \rangle \triangleright P\} \vdash \{x\langle a \rangle, P\{b/u, c/v\}\} & (\text{red}) \\
\rightleftharpoons & \{x\langle u \rangle | y\langle v \rangle \triangleright P\} \vdash \{x\langle a \rangle | P\{b/u, c/v\}\} & (\text{str-join}) \\
\rightleftharpoons & \{\} \vdash \{\mathbf{def} \ x\langle u \rangle | y\langle v \rangle \triangleright P \text{ in } x\langle a \rangle | P\{b/u, c/v\}\} & (\text{str-def})
\end{aligned}$$

and thus

$$\mathbf{def} \ x\langle u \rangle | y\langle v \rangle \triangleright P \text{ in } x\langle a \rangle | x\langle b \rangle | y\langle c \rangle \longrightarrow_J \mathbf{def} \ x\langle u \rangle | y\langle v \rangle \triangleright P \text{ in } x\langle a \rangle | P\{b/u, c/v\}$$

Note the non-determinism in receiving b instead of a on x . Instead of $x\langle a \rangle | P\{b/u, c/v\}$ we could also arrive at $x\langle b \rangle | P\{a/u, c/v\}$.

2.1.1. The Core Join-Calculus

The full calculus can be encoded in a fragment of itself, called core calculus [3]. Since all of its processes are also valid processes in the full calculus, it suffices to describe its semantics using the RCHAM as shown above, only using the rules (str-join), (str-def) and (red).

$$\begin{aligned}
P, Q ::= & x\langle u \rangle \\
& | P|Q \\
& | \mathbf{def} \ x\langle u \rangle | y\langle v \rangle \triangleright P \text{ in } Q
\end{aligned}$$

2.1.2. The Distributed Join-Calculus

In addition to an infinite set of channel names $\mathcal{N} = \{x, y, z, \kappa, \dots\}$, we now use an infinite set of location names $\mathcal{L} = \{a, b, c, \dots\}$. These can also be sent and received, which means that values u, v, \dots in $x\langle\tilde{v}\rangle$ can now be either channel or location names. We extend the syntax of the join-calculus by two new constructs [4]. Locations can be introduced using a new kind of definition:

$$D, E ::= \dots \\ \quad | \quad a[D : P]$$

This initializes a new sublocation a with definitions D and a process P residing there. We demand two additional syntactic restrictions: Firstly, a location can only be defined once in any definition D . Secondly, channel x may only appear in the join-patterns of one location. These conditions make the following definitions invalid:

$$\begin{aligned} &\text{def } a[D : P] \wedge a[E : Q] \triangleright R \text{ in } S \\ &\text{def } a[x\langle u \rangle \triangleright P : Q] \wedge b[x\langle v \rangle \triangleright R : S] \text{ in } T \end{aligned}$$

Also, we provide a construct for migration:

$$P, Q ::= \dots \\ \quad | \quad \text{go}\langle b, \kappa \rangle$$

It migrates the current location to b (becoming a sublocation) and afterwards emits an empty message on κ .

The semantics of the new constructs are defined using a distributed version of the RCHAM, called DRCHAM. It is a multiset of solutions $\mathcal{R} \vdash_{\varphi} \mathcal{M}$ with location *labels* $\varphi, \psi, \dots \in \mathcal{L}^*$, separated by \parallel . We consider trees of locations. For each location a in the tree, the DRCHAM contains a solution with a label consisting (left to right) of the path from the root down to a . This means that φ is a sublocation of ψ if ψ is a prefix of φ .

Each single solution internally evolves by the rules introduced before (with the addition that (str-def) also produces fresh location names). Furthermore, we add three new rules for the two new syntactical constructs:

$$\begin{aligned} (\text{str-loc}) \quad & a[D : P] \vdash_{\varphi} \rightleftharpoons \vdash_{\varphi} \parallel D \vdash_{\varphi a} P \quad (a \text{ frozen}) \\ (\text{comm}) \quad & \vdash_{\varphi} x\langle\tilde{v}\rangle \parallel J \triangleright P \vdash \longrightarrow \vdash_{\varphi} \parallel J \triangleright P \vdash x\langle\tilde{v}\rangle \quad (x \in \text{dv}(J)) \\ (\text{move}) \quad & a[D : P | \text{go}\langle b, \kappa \rangle] \vdash_{\varphi} \parallel \vdash_{\psi b} \longrightarrow \vdash_{\varphi} \parallel a[D : P | \kappa\langle \rangle] \vdash_{\psi b} \end{aligned}$$

In heating direction, (str-loc) initializes a new machine. The side condition “a frozen” applies in the cooling direction and expresses that there may be no solutions at sublocations of a (i.e. with a label containing a). This enforces that the rule is applied to all sublocations of a first to freeze the whole subtree, preparing migration. (comm) allows a message on a channel residing in a different location to move to that location, where it can be received using (red) later. The last rule (move) gives the semantics of migration: a becomes a sublocation of b . The emitted $\kappa\langle \rangle$ can be used to delay execution of a process until migration is completed.

2.2. The Actor Model

The notion of actors was first developed by Hewitt [7] and later formalized by Agha [8] as a model of concurrent computation. As described in [9], a computational system in the actor model, called a *configuration*, consists of a collection of concurrently executing actors and a collection of messages in transit. Each actor has a unique name (the *uniqueness* property) and a *behavior*, and it communicates with other actors via asynchronous messages. Actors are reactive in nature, i.e. they execute only in response to messages receives. An actor's behavior is deterministic in that its response to a message is uniquely determined by the message contents. Message delivery in the actor model is fair. The delivery of a message can only be delayed for a finite but unbounded amount of time.

An actor can perform three basic actions on receiving a message:

- create a finite number of actors with universally fresh names
- send a finite number of messages
- assume a new behavior

In addition to the already mentioned *uniqueness*, there are two other properties we should note: First, actors do not disappear after processing a message (the *persistence* property) and second, actors cannot be created with well known names or names received in a message (the *freshness* property).

The description of a configuration also defines an interface between the configuration and its environment, which constrains interactions between them. An interface is a set of names, called the *receptionist* set, that contains the names of all actors in the configuration that are visible to the environment. The receptionist set may evolve during interactions, as the messages that the configuration sends to the environment may contain names of actors not currently in the receptionist set.

2.2.1. The Actor Calculus $A\pi$

This definition of the actor model has many differences from the join-calculus and similar calculi. While actors are uniquely identified, processes in most calculi are anonymous. The core idea that allows $A\pi$ to resolve this mismatch, is to identify processes by the channel name they receive on. To allow this, several restrictions are enforced by a type system. The definition is directly taken from [9].

Syntax

We assume an infinite set of channel names $\mathcal{N} = \{u, v, w, x, y, z, \dots\}$. The set of actor configurations is defined by the following grammar.

$$\begin{array}{lcl}
P, Q & ::= & \mathbf{0} \\
& | & xy.P \\
& | & \bar{x}y \\
& | & (\nu x)P \\
& | & P \mid Q \\
& | & \mathbf{case } x \mathbf{ of } (y_1 : P_1, \dots, y_n : P_n) \\
& | & B\langle \tilde{x}; \tilde{y} \rangle
\end{array}$$

Order of precedence among the constructs is the order in which they are listed. The simplest terms are $\mathbf{0}$, representing an empty configuration, and output term $\bar{x}y$, which represents a configuration with a single message targeted to x containing a single name y . We will consider polyadic communication towards the end of this chapter.

The input term $x(y).P$ stands for a configuration with an actor x whose behavior is $(y)P$. Its parameter y constitutes the formal parameter list and binds all free occurrences of y in P . The actor x can receive an arbitrary name z and substitute it for y to behave like $P\{z/y\}$. A restricted process $(\nu x)P$ is the same as P , except that x is no longer a receptionist of P . The composition $P_1 \mid P_2$ is a configuration containing all the actors and messages in P_1 and P_2 . The configuration $\mathbf{case } x \mathbf{ of } (y_1 : P_1, \dots, y_n : P_n)$ behaves like P_i if $x = y_i$ for any i , and like $\mathbf{0}$ otherwise. If more than one branch matches, one of them is chosen non-deterministically. Finally, there are behavior instantiations $B\langle \tilde{u}; \tilde{v} \rangle$. Any instantiated identifier B must have a single defining equation of the form $B \stackrel{\text{def}}{=} (\tilde{u}; \tilde{v}) x_1(z).P$, where \tilde{x} is a tuple of one or two distinct names and x_1 denotes its first component. Also, the tuples \tilde{x} and \tilde{y} together must contain exactly the free names in $x_1(z).P$.

Type System

Not all syntactically well-formed configurations are valid actor configurations. For example, the term $x(u).P \mid x(v).Q$ contains two actors with name x , which violates the uniqueness property. Similarly, $x(u).u(v).P$ creates an actor with a name u that is received in a message, violating the freshness property. The actor properties can be enforced, however, by imposing a type system that statically tracks the receptionists of each term.

Since requiring an actor to only ever receive on one single channel is too restrictive, the persistence property is slightly relaxed. Intuitively, an actor is allowed to temporarily assume a new name, which requires the type system to track whether names are regular or only temporarily assumed by an actor.

Before presenting the type system, a few notational conventions are in order. For a tuple \tilde{x} and \tilde{y} , we denote the set of names occurring in \tilde{x} by $\{\tilde{x}\}$ and the result of appending y to x by y, x . Also, we use ϵ for the empty tuple and assume a variable \hat{z} to be either \emptyset or $\{z\}$.

We assume $\perp, * \notin \mathcal{N}$ and for $X \subset \mathcal{N}$ we define $X^* = X \cup \{\perp, *\}$. For $f : X \rightarrow X^*$ we define $f^* : X^* \rightarrow X^*$ as $f^*(x) = f(x)$ for $x \in X$ and $f^*(\perp) = f^*(*) = \perp$.

A typing judgement is of the form $\rho; f \vdash P$, where ρ is the receptionist set of P , and $f : \rho \rightarrow \rho^*$ is a temporary name mapping function that relates actors in P to the temporary names they have currently assumed. Specifically:

- $f(x) = \perp$ means that x is a regular actor name,
- $f(x) = y \notin \{\perp, *\}$ means that actor y has assumed the temporary name x , and
- $f(x) = *$ means x is the temporary name of an actor with a private name (bound by restriction).

The function f has the following properties, which we call *valid mapping* properties. For all $x, y \in \rho$,

- $f(x) \neq x$
- $f(x) = f(y) \notin \{\perp, *\}$ implies $x = y$, because an actor cannot assume more than one temporary name at the same time.
- $f^*(f(x)) = \perp$, because temporary names cannot themselves assume temporary names.

We define a few functions and relations on the temporary name mapping functions. Let $f_1 : \rho_1 \rightarrow \rho_1^*$ and $f_2 : \rho_2 \rightarrow \rho_2^*$.

- We define $f_1 \oplus f_2 : \rho_1 \cup \rho_2 \rightarrow (\rho_1 \cup \rho_2)^*$ as

$$(f_1 \oplus f_2)(x) = \begin{cases} f_1(x) & \text{if } x \in \rho_1, \text{ and } f_1(x) \neq \perp \text{ or } x \notin \rho_2 \\ f_2(x) & \text{otherwise} \end{cases}$$

Note that \oplus is associative.

- If $\rho \subset \rho_1$, we define $f|_\rho : \rho \rightarrow \rho^*$ as

$$(f|_\rho)(x) = \begin{cases} * & \text{if } f(x) \in \rho_1 - \rho \\ f(x) & \text{otherwise} \end{cases}$$

- We say f_1 and f_2 are compatible if $f_1 \oplus f_2 = f_2 \oplus f_1 = f$ has the *valid mapping* properties.
- Also, for a tuple \tilde{x} we define $ch(\tilde{x}) : \{\tilde{x}\} \rightarrow \{\tilde{x}\}^*$ as $ch(\epsilon) = \{\}$ and if $\text{len}(\tilde{x}) = n$, $ch(\tilde{x})(x_i) = x_{i+1}$ for $1 \leq i < n$ and $ch(\tilde{x})(x_n) = \perp$.

Intuitively, $ch(x)$ represents a configuration with a single receptionist x , and $ch(t, x)$ represents a receptionist x that temporarily assumed the name t . Note that $ch(\tilde{x})$ with $\text{len}(\tilde{x}) > 2$ cannot have the *valid mapping* properties.

$$\begin{aligned}
\text{NIL} : & \frac{}{\emptyset; \{\} \vdash \mathbf{0}} \\
\text{MSG} : & \frac{}{\emptyset; \{\} \vdash \bar{x}y} \\
\text{ACT} : & \frac{\rho; f \vdash P}{\{x\} \cup \tilde{z}; ch(x, \tilde{z}) \vdash xy.P} \text{ if } \begin{array}{l} \rho - \{x\} = \tilde{z}, y \notin \rho, \text{ and} \\ f = \begin{cases} ch(x, \tilde{z}) & \text{if } x \in \rho \\ ch(\epsilon, \tilde{z}) & \text{otherwise} \end{cases} \end{array} \\
\text{CASE} : & \frac{\forall 1 \leq i \leq n : \rho_i; f_i \vdash P_i}{(\cup_i \rho_i); (\oplus_i f_i) \vdash \mathbf{case } x \mathbf{ of } (y_1 : P_1, \dots, y_n : P_n)} \text{ if } f_i \text{ are mutually compatible} \\
\text{COMP} : & \frac{\rho_1; f_1 \vdash P_1 \quad \rho_2; f_2 \vdash P_2}{\rho_1 \cup \rho_2; f_1 \oplus f_2 \vdash P_1 \mid P_2} \text{ if } \rho_1 \cap \rho_2 = \emptyset \\
\text{RES} : & \frac{\rho; f \vdash P}{\rho - \{x\}; f|_{\rho - \{x\}} \vdash (\nu x)P} \\
\text{INST} : & \frac{}{\{\tilde{x}\}; ch(\tilde{x}) \vdash B\langle \tilde{x}; \tilde{y} \rangle} \text{ if } \text{len}(\tilde{x}) = 2 \text{ implies } x_1 \neq x_2
\end{aligned}$$

In the ACT rule, if $\hat{z} = \{z\}$, then actor z has assumed temporary name x . The conditions $y \notin \rho$ and $\rho - \{x\} = \hat{z}$ together guarantee the freshness property by ensuring that new actors are created with fresh names. Note that it is possible for x to be a regular name and disappear after receiving a message ($x \notin \rho$). We interpret this as the actor x having assumed a sink behavior, consuming messages without reaction. With this interpretation, the persistence property is not violated.

The compatibility check in CASE prevents errors such as two actors in different branches assuming the same temporary name or the same actor assuming different temporary names in different branches. The COMP rule is critical for the uniqueness property by ensuring that two composed configurations do not contain actors with the same name. For the INST rule to be sound, for every definition $B \stackrel{\text{def}}{=} (\tilde{x}; \tilde{y}) x_1(z).P$ the judgement $\{\tilde{x}\}; ch(x) \vdash x_1(z).P$ must be derivable.

Semantics

The operational semantics of $A\pi$ is formally specified using a labeled transition system (defined modulo alpha-equivalence). The symmetric versions of COM, CLOSE, PAR are not shown. Transition labels, which are also called actions, can be of five forms: τ (silent action), $\bar{x}y$ (free output of message with target x and content y), $\bar{x}(y)$ (bound output), xy (free input), and $x(y)$ (bound input). We let α range over all visible (non- τ) actions \mathcal{L} , and β over all actions.

$$\begin{aligned}
\text{OUT} : & \frac{}{\bar{x}y \xrightarrow{A\pi} \mathbf{0}} \\
\text{INP} : & \frac{}{xy.P \xrightarrow{A\pi} P\{z/y\}} \\
\text{BINP} : & \frac{P \xrightarrow{A\pi} P' \text{ if } y \notin \text{fn}(P)}{P \xrightarrow{A\pi} P'} \\
\text{RES} : & \frac{P \xrightarrow{A\pi} P'}{(\nu y)P \xrightarrow{A\pi} (\nu y)P'} \text{ if } y \notin \text{n}(\alpha) \\
\text{OPEN} : & \frac{P \xrightarrow{A\pi} P' \text{ if } x \neq y}{(\nu y)P \xrightarrow{A\pi} P'} \\
\text{PAR} : & \frac{P_1 \xrightarrow{A\pi} P'_1}{P_1 \mid P_2 \xrightarrow{A\pi} P'_1 \mid P_2} \text{ if } \text{bn}(\alpha) \cap \text{fn}(P_2) = \emptyset \\
\text{COM} : & \frac{P_1 \xrightarrow{A\pi} P'_1 \quad P_2 \xrightarrow{A\pi} P'_2}{P_1 \mid P_2 \xrightarrow{A\pi} P'_1 \mid P_2} \\
\text{CLOSE} : & \frac{P_1 \xrightarrow{A\pi} P'_1 \quad P_2 \xrightarrow{A\pi} P'_2}{P_1 \mid P_2 \xrightarrow{A\pi} (\nu y)P'_1 \mid P_2} \text{ if } y \notin \text{fn}(P_2) \\
\text{BRNCH} : & \frac{}{\text{case } x \text{ of } (y_1 : P_1, \dots, y_n : P_n) \xrightarrow{A\pi} P_i} \text{ if } x = y_i \\
\text{BEHV} : & \frac{(x_1 z.P)\{(\tilde{u}, \tilde{v})/(\tilde{x}, \tilde{y})\} \xrightarrow{A\pi} P'}{B\langle \tilde{x}; \tilde{y} \rangle \xrightarrow{A\pi} P'} \text{ if } B \stackrel{\text{def}}{=} (\tilde{x}; \tilde{y}) x_1 z.P
\end{aligned}$$

Structural Congruence

The structural congruence $\equiv_{A\pi}$ on processes is the smallest possible congruence closed under the following rules [10].

$$\begin{aligned}
& P \mid \mathbf{0} \equiv_{A\pi} P \\
& P \mid Q \equiv_{A\pi} Q \mid P \\
& P \mid (Q \mid R) \equiv_{A\pi} (P \mid Q) \mid R \\
& (\nu x)\mathbf{0} \equiv_{A\pi} \mathbf{0} \\
& (\nu x)(\nu y)\mathbf{0} \equiv_{A\pi} (\nu y)(\nu x)\mathbf{0} \\
& P \mid (\nu x)Q \equiv_{A\pi} (\nu x)(P \mid Q) \text{ if } x \notin \text{fn}(P) \\
& B\langle \tilde{u}; \tilde{v} \rangle \equiv_{A\pi} (x_1(z).P)\{(\tilde{u}, \tilde{v})/(\tilde{x}, \tilde{y}) \text{ if } \begin{cases} B \stackrel{\text{def}}{=} (\tilde{x}; \tilde{y}) x_1(z).P \\ \text{len}(\tilde{u}) = \text{len}(\tilde{x}) \\ \text{len}(\tilde{v}) = \text{len}(\tilde{y}) \end{cases}
\end{aligned}$$

Polyadic Communication

As shown in [9], polyadic communication (messages containing multiple names) can be encoded in monadic $A\pi$. For fresh u, z we have

$$\begin{aligned}
\llbracket \bar{x}\langle y_1, \dots, y_n \rangle \rrbracket_p &= (\nu u)(\bar{x}u \mid S_1\langle u; y_1, \dots, y_n \rangle) \\
S_i &\stackrel{\text{def}}{=} (u; y_1, \dots, y_n) uz. (\bar{z}y_i \mid S_{i+1}\langle u; y_{i+1}, \dots, y_n \rangle) \quad 1 \leq i < n \\
S_n &\stackrel{\text{def}}{=} (u; y_n) uz. \bar{z}y_n \\
\llbracket x(y_1, \dots, y_n).P \rrbracket_p &= xu.(\nu z)(\bar{u}z \mid R_1\langle z, \hat{x}; u, \tilde{a} \rangle) \\
R_i &\stackrel{\text{def}}{=} (z, \hat{x}; u, \tilde{a}) zy_i. (\bar{u}z \mid R_{i+1}\langle z, \hat{x}; u, \tilde{a} \rangle) \quad 1 \leq i < n \\
R_n &\stackrel{\text{def}}{=} (z, \hat{x}; u, \tilde{a}) zy_n. (\bar{u}z \mid \llbracket P \rrbracket_p)
\end{aligned}$$

where $\tilde{a} = \text{fn}(x(y_1, \dots, y_n).P) - \{x\}$, and $\hat{x} = \{x\}$ if $\rho \cup \{x\}; f \vdash \llbracket P \rrbracket_p$ for some ρ, f and $\hat{x} = \emptyset$ otherwise.

In the paper, it is argued that this encoding cannot be used for polyadic reception on temporary names, since they cannot assume temporary names themselves. However, this is too restrictive. The necessary condition for $x(y_1, \dots, y_n).P$ is that P is an actor with a regular name. Therefore, x can be temporary, as long as it is not used in P anymore.

Furthermore, we adapt the MSG and ACT rules of the type system to accomodate polyadic communication. In ACT, we now not only need to prevent the creation of actors with a single name received in a message, but for all of them. Apart from that, the rules remain unchanged.

$$\text{ACT} : \frac{\rho; f \vdash P}{\{x\} \cup \tilde{z}; ch(x, \tilde{z}) \vdash x(y_1, \dots, y_n).P} \text{ if } \begin{cases} \rho - \{x\} = \tilde{z}, y_i \notin \rho \text{ for all } 1 \leq i \leq n, \text{ and} \\ f = \begin{cases} ch(x, \tilde{z}) \text{ if } x \in \rho \\ ch(\epsilon, \tilde{z}) \text{ otherwise} \end{cases} \end{cases}$$

3. Encoding

3.1. A Naive Approach

When introducing the join-calculus, Fournet also gave a simple encoding into the asynchronous pi-calculus [3]. Considering the minor syntactical differences, we can translate it to $A\pi$.

Let us also define \tilde{z}_P for the remainder of this chapter as the tuple of names in $\text{fn}(P) \setminus \{x, y, u, v\}$. This is necessary since P might contain free names that are now also free in the definition of B and thus have to be bound in the parameter list.

$$\begin{aligned} \llbracket P|Q \rrbracket &= \llbracket P \rrbracket \mid \llbracket Q \rrbracket \\ \llbracket x\langle y \rangle \rrbracket &= \bar{x}\langle y \rangle \\ \llbracket \mathbf{def} \ x\langle u \rangle | y\langle v \rangle \triangleright P \mathbf{in} \ Q \rrbracket &= (\nu x)(\nu y) (B\langle x, y; \tilde{z} \rangle \mid \llbracket Q \rrbracket) \\ &\text{with } B \stackrel{\text{def}}{=} (x, y; \tilde{z}) \ x(u).y(v). (B\langle x, y; \tilde{z} \rangle \mid \llbracket P \rrbracket) \end{aligned}$$

The definition of behavior B in this encoding, however, is not well-typed. Assuming that P has no recipients, we can see that $\{x, y\}; ch(x, y) \vdash B\langle x, y; \tilde{z} \rangle \mid \llbracket P \rrbracket$, but then ACT does not apply to $y(v). (B\langle x, y; \tilde{z} \rangle \mid \llbracket P \rrbracket)$, since y is already in the recipients (but not in first position).

Swapping the input actions to arrive at $\{x, y\}; ch(x, y) \vdash x(u). (B\langle x, y; \tilde{z} \rangle \mid \llbracket P \rrbracket)$ does not help, either, as we then have the same problem when prefixing it by $y(v)$. Indeed, this approach is fundamentally flawed.

3.2. Forwarding

The *persistence* property of the actor model is only slightly relaxed in $A\pi$ to temporarily allow assuming another name. It is not possible for an actor to receive on two different channels interchangeably. This restriction can only be circumvented by sending both x and y messages on the same channel, for example using a forwarder.

$$B_{fw} \stackrel{\text{def}}{=} (x; a) \ x(i). (\bar{a}\langle x, i \rangle \mid B_{fw}\langle x; a \rangle)$$

When B_{fw} receives a payload i on a channel x , it forwards it to a , tagged with the name of the channel it was originally sent on. At the same time, it recursively assumes the same behavior again to forward further messages.

3.3. Storing Payloads

When forwarding both x and y to the same actor, a problem arises. Messages from x and y can occur in an arbitrary order. If we, for example, receive multiple messages from x , we must either:

1. respawn all but one of them and wait for a y message. This introduces divergence, since the respawned messages could be received and spawned indefinitely.
2. store them all.

The only available values in $A\pi$ are names, but it is possible to build data structures from actors. A list is an actor and can either be empty or constructed from a single first element (*head*) and a remaining list (*tail*). To find out which of these is the case, we send it a message with two continuations (names) k_n and k_c .

If it is empty, it will send an empty message on k_n :

$$B_{nil} \stackrel{def}{=} (l;) l(k_n, k_c). (\overline{k_n} \langle \rangle \mid B_{nil} \langle l; \rangle)$$

If not, it sends head and tail on k_c :

$$B_{cons} \stackrel{def}{=} (l; h, t) l(k_n, k_c). (\overline{k_c} \langle h, t \rangle \mid B_{cons} \langle l; h, t \rangle)$$

3.4. Our Encoding

Using forwarders and lists, we can define an actor which receives forwarded messages from both channels x and y . If it receives more messages from one of these channels than from the other, it stores the excess payloads in a list l and remembers from which channel these came using a flag f . This is possible since there can always only be excessive messages on one of these channels; otherwise they would have been joined already. We have to make sure the free names in P do not conflict with any of the names $a, f, l, c, p, k_n, k_c, k_P$ and k_a to give following behavior for the actor.

$$\begin{aligned} B_a^P \stackrel{def}{=} & (a; x, y, f, l, \tilde{z}_P) a(c, p). (\nu k_n)(\nu k_c)(\nu k_P)(\nu k_a) \\ & (\overline{l} \langle k_n, k_c \rangle \\ & \mid k_n().(\nu l') (B_{cons} \langle l'; p, l \rangle \mid \overline{k_a} \langle c, l' \rangle) \\ & \mid k_c(h, t). \text{case } f \text{ of } (x : \text{case } c \text{ of } (x : (\nu l') (B_{cons} \langle l'; p, l \rangle \mid \overline{k_a} \langle f, l' \rangle) \\ & \qquad \qquad \qquad y : \overline{k_P} \langle h, p \rangle \mid \overline{k_a} \langle f, t \rangle) \\ & \qquad \qquad \qquad y : \text{case } c \text{ of } (y : (\nu l') (B_{cons} \langle l'; p, l \rangle \mid \overline{k_a} \langle f, l' \rangle) \\ & \qquad \qquad \qquad x : \overline{k_P} \langle p, h \rangle \mid \overline{k_a} \langle f, t \rangle)) \\ & \mid k_P(u, v). \llbracket P \rrbracket \\ & \mid k_a(f', l'). B_a^P \langle a; x, y, f', l', \tilde{z}_P \rangle) \end{aligned}$$

To encode a definition, we now start one such joining actor (with initially no excess payloads), a forwarder for each channel and the encoded subprocess in which the definition is available.

$$\begin{aligned}
\llbracket P|Q \rrbracket &= \llbracket P \rrbracket \mid \llbracket Q \rrbracket \\
\llbracket x\langle y \rangle \rrbracket &= \bar{x}\langle y \rangle \\
\llbracket \mathbf{def} \ x\langle u \rangle | y\langle v \rangle \triangleright P \ \mathbf{in} \ Q \rrbracket &= (\nu a)(\nu x)(\nu y)(\nu l) \ (\ B_a^P \langle a; x, y, x, l, \tilde{z}_P \rangle \\
&\quad \mid B_{fw} \langle x; a \rangle \mid B_{fw} \langle y; a \rangle \\
&\quad \mid B_{nil} \langle l; \rangle \\
&\quad \mid \llbracket Q \rrbracket \)
\end{aligned}$$

As the parallel operator is translated homomorphically and each definition is translated into separate actors, the encoding preserves distributability [2].

3.4.1. Type Correctness

Theorem 1 $\emptyset; \{\} \vdash \llbracket J \rrbracket$ for all terms J in the core join-calculus

Proof We prove the theorem by structural induction over the grammar of the core join-calculus. Let $x, y \in \mathcal{N}$ and J any term in the core join-calculus.

Case 1 (Base Step) $J = x\langle y \rangle$.

Then $\llbracket J \rrbracket = \bar{x}\langle y \rangle$ and thus $\emptyset; \{\} \vdash \llbracket J \rrbracket$ by MSG.

Case 2 $J = P|Q$

Then we know by inductive hypothesis that $\emptyset; \{\} \vdash \llbracket P \rrbracket$ and $\emptyset; \{\} \vdash \llbracket Q \rrbracket$. Since $\llbracket P|Q \rrbracket = \llbracket P \rrbracket \mid \llbracket Q \rrbracket$, we can infer $\emptyset; \{\} \vdash \llbracket J \rrbracket$ by COMP.

Case 3 $J = \mathbf{def} \ x\langle u \rangle | y\langle v \rangle \triangleright P \ \mathbf{in} \ Q$

Then we know by inductive hypothesis that $\emptyset; \{\} \vdash \llbracket P \rrbracket$ and $\emptyset; \{\} \vdash \llbracket Q \rrbracket$.

First, all used definitions have to be well-typed.

$$\begin{aligned}
B_{fw} &\stackrel{def}{=} (x; a) x(i). (\bar{a}\langle x, i \rangle \mid B_{fw}\langle x; a \rangle) & \{x\}; ch(x) \vdash x(i). (\bar{a}\langle x, i \rangle \mid B_{fw}\langle x; a \rangle) \\
B_{nil} &\stackrel{def}{=} (l;) l(k_n, k_c). (\bar{k}_n\langle \rangle \mid B_{nil}\langle l; \rangle) & \{l\}; ch(l) \vdash l(k_n, k_c). (\bar{k}_n\langle \rangle \mid B_{nil}\langle l; \rangle) \\
B_{cons} &\stackrel{def}{=} (l; h, t) l(k_n, k_c). (\bar{k}_c\langle h, t \rangle \mid B_{cons}\langle l; h, t \rangle) & \{l\}; ch(l) \vdash l(k_n, k_c). (\bar{k}_c\langle h, t \rangle \mid B_{cons}\langle l; h, t \rangle)
\end{aligned}$$

$$B_a^P \stackrel{def}{=} (a; x, y, f, l, \tilde{z}_P) a(c, p).(\nu k_n)(\nu k_c)(\nu k_P)(\nu k_a)$$

$$R \left\{ \begin{array}{l} (\bar{l}\langle k_n, k_c \rangle \\ | k_n().(\nu l') (B_{cons}\langle l'; p, l \rangle | \bar{k}_a\langle c, l' \rangle) \\ | k_c(h, t). \mathbf{case} \ f \ \mathbf{of} \ (\ x : \mathbf{case} \ c \ \mathbf{of} \ (\ x : (\nu l') (B_{cons}\langle l'; p, l \rangle | \bar{k}_a\langle f, l' \rangle) \\ \qquad \qquad \qquad y : \bar{k}_P\langle h, p \rangle | \bar{k}_a\langle f, t \rangle) \\ \qquad \qquad \qquad y : \mathbf{case} \ c \ \mathbf{of} \ (\ y : (\nu l') (B_{cons}\langle l'; p, l \rangle | \bar{k}_a\langle f, l' \rangle) \\ \qquad \qquad \qquad \qquad \qquad \qquad x : \bar{k}_P\langle p, h \rangle | \bar{k}_a\langle f, t \rangle)) \\ | k_P(u, v). \llbracket P \rrbracket \\ | k_a(f', l'). B_a^P\langle a; x, y, f', l', \tilde{z}_P \rangle) \end{array} \right.$$

Since none of the branches in the nested case-term have any recipients and the last line has type $ch(k_a, a)$, we can derive:

$$\begin{aligned} \{k_n, k_c, k_P, k_a, a\}; g \vdash R \text{ with } g(k_n) = g(k_c) = g(k_P) = g(a) = \perp, g(k_a) = a \\ \{a\}; ch(a) \vdash (\nu k_n)(\nu k_c)(\nu k_P)(\nu k_a) R \\ \{a\}; ch(a) \vdash a(c, p).(\nu k_n)(\nu k_c)(\nu k_P)(\nu k_a) R \end{aligned}$$

Now that we know that all definitions are correct, we can infer the type of $\llbracket J \rrbracket$:

$$\llbracket J \rrbracket = (\nu a)(\nu x)(\nu y)(\nu l) \underbrace{(B_a^P\langle a; x, y, x, l, \tilde{z}_P \rangle | B_{fw}\langle x; a \rangle | B_{fw}\langle y; a \rangle | B_{nil}\langle l; \rangle | \llbracket Q \rrbracket)}_S$$

Then $\{a, x, y, l\}; f \vdash S$ with $f(a) = f(x) = f(y) = f(l) = \perp$ and $\emptyset; \{\} \vdash \llbracket J \rrbracket$. \square

3.4.2. Semantic Correctness

In the join-calculus, we can receive in a join-pattern using following reduction:

$$\mathbf{def} \ x\langle u \rangle | y\langle v \rangle \triangleright P \ \mathbf{in} \ x\langle b \rangle | y\langle c \rangle | Q \longrightarrow_J \mathbf{def} \ x\langle u \rangle | y\langle v \rangle \triangleright P \ \mathbf{in} \ P\{b/u, c/v\} | Q$$

To see that the proposed encoding allows the same behavior, we can look at the reductions possible for the corresponding $\Lambda\pi$ term. We will make use of the structural congruence to simplify the term between reductions and focus on the interesting aspects.

$$\begin{aligned} \llbracket \mathbf{def} \ x\langle u \rangle | y\langle v \rangle \triangleright P \ \mathbf{in} \ x\langle b \rangle | y\langle c \rangle | Q \rrbracket \equiv_{A\pi} (\nu a)(\nu x)(\nu y)(\nu l) \\ (\ B_{fw}\langle x; a \rangle | B_{fw}\langle y; a \rangle \\ | \bar{x}\langle b \rangle | \bar{y}\langle c \rangle | \llbracket Q \rrbracket \\ | B_{nil}\langle l; \rangle | B_a^P\langle a; x, y, x, l, \tilde{z}_P \rangle) \end{aligned}$$

In this configuration, the messages $\bar{x}\langle b \rangle$ and $\bar{y}\langle c \rangle$ can be received by the forwarders $B_{fw}\langle x; a \rangle$ and $B_{fw}\langle y; a \rangle$ respectively, emitting the forwarded messages $\bar{a}\langle x, b \rangle$ and $\bar{a}\langle y, c \rangle$.

$$\begin{aligned}
& (\nu a)(\nu x)(\nu y)(\nu l) \\
& \quad (\bar{a}\langle x, b \rangle \mid B_{fw}\langle x; a \rangle \mid \bar{a}\langle y, c \rangle \mid B_{fw}\langle y; a \rangle \mid \llbracket Q \rrbracket \mid B_{nil}\langle l; \rangle \\
& \quad \mid B_a^P\langle a; x, y, x, l, \tilde{z}_P \rangle)
\end{aligned}$$

Any of these can now be received by B_a . We pick $\bar{a}\langle x, b \rangle$ first, but the other case works similarly. To receive, we have to inline B_a 's definition, binding all its parameters. Also, we pull all introduced name scopes to the outside.

$$\begin{aligned}
& (\nu a)(\nu x)(\nu y)(\nu l)(\nu k_n)(\nu k_c)(\nu k_P)(\nu k_a) \\
& \quad (B_{fw}\langle x; a \rangle \mid \bar{a}\langle y, c \rangle \mid B_{fw}\langle y; a \rangle \mid \llbracket Q \rrbracket \mid B_{nil}\langle l; \rangle \\
& \quad \mid \bar{l}\langle k_n, k_c \rangle \\
& \quad \mid k_n().(\nu l') (B_{cons}\langle l'; b, l \rangle \mid \overline{k_a}\langle x, l' \rangle) \\
& \quad \mid k_c(h, t). \mathbf{case} \ f \ \mathbf{of} \ (\ x : \mathbf{case} \ x \ \mathbf{of} \ (\ x : (\nu l') (B_{cons}\langle l'; b, l \rangle \mid \overline{k_a}\langle f, l' \rangle) \\
& \quad \quad \quad y : \overline{k_P}\langle h, b \rangle \mid \overline{k_a}\langle f, t \rangle) \\
& \quad \quad \quad y : \mathbf{case} \ x \ \mathbf{of} \ (\ y : (\nu l') (B_{cons}\langle l'; b, l \rangle \mid \overline{k_a}\langle f, l' \rangle) \\
& \quad \quad \quad \quad \quad \quad x : \overline{k_P}\langle b, h \rangle \mid \overline{k_a}\langle f, t \rangle)) \\
& \quad \mid k_P(u, v). \llbracket P \rrbracket \\
& \quad \mid k_a(f', l'). B_a^P\langle a; x, y, f', l', \tilde{z}_P \rangle)
\end{aligned}$$

Now, $\bar{l}\langle k_n, k_c \rangle$ is received by $B_{nil}\langle l; \rangle$, which (since it is empty), emits $k_n\langle \rangle$. We use this opportunity to clean up the term a bit. After this step, the only occurrences of k_c and k_P are where they are received or in a process blocked by such a reception. Using the structural congruence, we can limit the scope of these names and arrive at $(\nu k_c)(\nu k_P) (k_c(h, t).P_{k_c} \mid k_P(u, v).P_{k_P})$ for the corresponding P_{k_c} and P_{k_P} .

This process cannot interact with its context or be reduced any further. To do so, a message on k_c or k_P would be necessary, but it can never occur. We denote this term as $R_{garbage}$.

$$\begin{aligned}
& (\nu a)(\nu x)(\nu y)(\nu l)(\nu k_n)(\nu k_a) \\
& \quad (B_{fw}\langle x; a \rangle \mid \bar{a}\langle y, c \rangle \mid B_{fw}\langle y; a \rangle \mid \llbracket Q \rrbracket \mid \overline{k_n}\langle \rangle \mid B_{nil}\langle l; \rangle \\
& \quad \mid k_n().(\nu l') (B_{cons}\langle l'; b, l \rangle \mid \overline{k_a}\langle x, l' \rangle) \\
& \quad \mid k_a(f', l'). B_a^P\langle a; x, y, f', l', \tilde{z}_P \rangle \\
& \quad \mid R_{garbage})
\end{aligned}$$

We then receive $\overline{k_n}\langle \rangle$ in the third line and pull outwards the scope of l' .

$$\begin{aligned}
& (\nu a)(\nu x)(\nu y)(\nu l)(\nu k_a)(\nu l') \\
& \quad (B_{fw}\langle x; a \rangle \mid \bar{a}\langle y, c \rangle \mid B_{fw}\langle y; a \rangle \mid \llbracket Q \rrbracket \mid B_{nil}\langle l; \rangle \\
& \quad \mid B_{cons}\langle l'; b, l \rangle \mid \overline{k_a}\langle x, l' \rangle \\
& \quad \mid k_a(f', l'). B_a^P\langle a; x, y, f', l', \tilde{z}_P \rangle \\
& \quad \mid R_{garbage})
\end{aligned}$$

The continuation k_a is received next. Afterwards, it does not occur at all, which allows us to completely remove its scope using the structural congruence:

$$(\nu k_a)S \equiv_{A\pi} (\nu k_a)(S \mid \mathbf{0}) \equiv_{A\pi} S \mid (\nu k_a)\mathbf{0} \equiv_{A\pi} S \mid \mathbf{0} \equiv_{A\pi} S \quad (\text{for any } S \text{ without } k_a)$$

$$\begin{aligned} & (\nu a)(\nu x)(\nu y)(\nu l)(\nu l') \\ & \quad (B_{fw}\langle x; a \rangle \mid \bar{a}\langle y, c \rangle \mid B_{fw}\langle y; a \rangle \mid \llbracket Q \rrbracket \mid B_{nil}\langle l; \rangle \mid B_{cons}\langle l'; b, l \rangle \\ & \quad \mid B_a^P\langle a; x, y, x, l', \tilde{z}_P \rangle \\ & \quad \mid R_{garbage}) \end{aligned}$$

At this point, it is important to note that we are almost in the same situation as in the beginning, but B_a successfully received its first message from x and stored the payload b in list l' . If there was another message $\bar{x}\langle d \rangle$, it could be added to the list as well, resulting in $(\nu l'')(\nu l')(\nu l)(\dots \mid B_{cons}\langle l''; d, l' \rangle \mid B_{cons}\langle l'; b, l \rangle \mid B_{nil}\langle l; \rangle)$.

The process can now receive the complimentary message $\bar{a}\langle y, c \rangle$. In its body, we alpha-rename the occurrences of l' to l'' to avoid conflicts.

$$\begin{aligned} & (\nu a)(\nu x)(\nu y)(\nu l)(\nu l')(\nu k_n)(\nu k_c)(\nu k_P)(\nu k_a) \\ & \quad (B_{fw}\langle x; a \rangle \mid B_{fw}\langle y; a \rangle \mid \llbracket Q \rrbracket \mid B_{nil}\langle l; \rangle \mid B_{cons}\langle l'; b, l \rangle \\ & \quad \mid \bar{l}'\langle k_n, k_c \rangle \\ & \quad \mid k_n().(\nu l'') (B_{cons}\langle l''; c, l' \rangle \mid \bar{k}_a\langle y, l'' \rangle) \\ & \quad \mid k_c(h, t). \mathbf{case} \ x \ \mathbf{of} \ (\ x : \mathbf{case} \ y \ \mathbf{of} \ (\ x : (\nu l'') (B_{cons}\langle l''; c, l' \rangle \mid \bar{k}_a\langle x, l'' \rangle) \\ & \quad \quad \quad y : \bar{k}_P\langle h, c \rangle \mid \bar{k}_a\langle x, t \rangle) \\ & \quad \quad \quad y : \mathbf{case} \ y \ \mathbf{of} \ (\ y : (\nu l'') (B_{cons}\langle l''; c, l' \rangle \mid \bar{k}_a\langle x, l'' \rangle) \\ & \quad \quad \quad x : \bar{k}_P\langle c, h \rangle \mid \bar{k}_a\langle x, t \rangle)) \\ & \quad \mid k_P(u, v). \llbracket P \rrbracket \\ & \quad \mid k_a(f', l''). B_a^P\langle a; x, y, f', l'', \tilde{z}_P \rangle \\ & \quad \mid R_{garbage}) \end{aligned}$$

We now perform multiple reductions at once: First, $B_{cons}\langle l'; b, l \rangle$ receives $\bar{l}'\langle k_n, k_c \rangle$. Then (because it is not empty) it sends $\bar{k}_c\langle b, l \rangle$, which is in turn received.

Also, we remove the scope of k_c and add the continuation k_n to the existing garbage (resulting in $R'_{garbage}$).

$$\begin{aligned}
& (\nu a)(\nu x)(\nu y)(\nu l)(\nu l')(\nu k_P)(\nu k_a) \\
& \quad (\ B_{fw}\langle x; a \rangle \mid B_{fw}\langle y; a \rangle \mid \llbracket Q \rrbracket \mid B_{nil}\langle l; \rangle \mid B_{cons}\langle l'; b, l \rangle \\
& \quad \mid \textbf{case } x \textbf{ of } (\ x : \textbf{case } y \textbf{ of } (\ x : (\nu l'') (B_{cons}\langle l''; c, l' \rangle \mid \overline{k_a}\langle x, l'' \rangle) \\
& \quad \quad \quad y : \overline{k_P}\langle b, c \rangle \mid \overline{k_a}\langle x, l \rangle \) \\
& \quad \quad \quad y : \textbf{case } y \textbf{ of } (\ y : (\nu l'') (B_{cons}\langle l''; c, l' \rangle \mid \overline{k_a}\langle x, l'' \rangle) \\
& \quad \quad \quad \quad \quad x : \overline{k_P}\langle c, b \rangle \mid \overline{k_a}\langle x, l \rangle \) \) \\
& \quad \mid k_P(u, v). \llbracket P \rrbracket \\
& \quad \mid k_a(f', l''). B_a^P \langle a; x, y, f', l'', \tilde{z}_P \rangle \\
& \quad \mid R'_{garbage} \)
\end{aligned}$$

At this point it becomes apparent which of the cases applies.

$$\begin{aligned}
& (\nu a)(\nu x)(\nu y)(\nu l)(\nu l')(\nu k_P)(\nu k_a) \\
& \quad (\ B_{fw}\langle x; a \rangle \mid B_{fw}\langle y; a \rangle \mid \llbracket Q \rrbracket \mid B_{nil}\langle l; \rangle \mid B_{cons}\langle l'; b, l \rangle \\
& \quad \mid \overline{k_P}\langle b, c \rangle \mid \overline{k_a}\langle x, l \rangle \\
& \quad \mid k_P(u, v). \llbracket P \rrbracket \\
& \quad \mid k_a(f', l''). B_a^P \langle a; x, y, f', l'', \tilde{z}_P \rangle \\
& \quad \mid R'_{garbage} \)
\end{aligned}$$

The continuation k_P passes the correct parameters to $\llbracket P \rrbracket$.

$$\begin{aligned}
& (\nu a)(\nu x)(\nu y)(\nu l)(\nu l')(\nu k_a) \\
& \quad (\ B_{fw}\langle x; a \rangle \mid B_{fw}\langle y; a \rangle \mid \llbracket Q \rrbracket \mid B_{nil}\langle l; \rangle \mid B_{cons}\langle l'; b, l \rangle \\
& \quad \mid \overline{k_a}\langle x, l \rangle \\
& \quad \mid \llbracket P \rrbracket \{b/u, c/v\} \\
& \quad \mid k_a(f', l''). B_a^P \langle a; x, y, f', l'', \tilde{z}_P \rangle \\
& \quad \mid R'_{garbage} \)
\end{aligned}$$

As a last step, the continuation k_a prepares B_a for receiving further messages, with an empty list of payloads again.

$$\begin{aligned}
& (\nu a)(\nu x)(\nu y)(\nu l)(\nu l') \\
& \quad (\ B_{fw}\langle x; a \rangle \mid B_{fw}\langle y; a \rangle \mid \llbracket Q \rrbracket \mid B_{nil}\langle l; \rangle \mid B_{cons}\langle l'; b, l \rangle \\
& \quad \mid \llbracket P \rrbracket \{b/u, c/v\} \\
& \quad \mid B_a^P \langle a; x, y, x, l, \tilde{z}_P \rangle \\
& \quad \mid R'_{garbage} \)
\end{aligned}$$

We can make two observations: The substitution can be moved into the encoding, since it is name invariant. Also, we can add $(\nu l')B_{cons}\langle l'; b, l \rangle$ to the garbage.

$$\begin{aligned}
& (\nu a)(\nu x)(\nu y)(\nu l) \\
& \quad (\ B_{fw}\langle x; a \rangle \mid B_{fw}\langle y; a \rangle \mid \llbracket Q \rrbracket \mid B_{nil}\langle l; \rangle \\
& \quad \mid \llbracket P\{b/u, c/v\} \rrbracket \\
& \quad \mid B_a^P\langle a; x, y, x, l, \tilde{z}_P \rangle \\
& \quad \mid R''_{garbage} \)
\end{aligned}$$

Comparing this configuration to the encoding of the reduction result in the join-calculus, we see that it is structurally equivalent, with the exception of $R''_{garbage}$. As the garbage only consists of terms that cannot interact anymore, we conclude that both configurations have the same behavior.

$$\begin{aligned}
\llbracket \mathbf{def} \ x\langle u \rangle \mid y\langle v \rangle \triangleright P \ \mathbf{in} \ P\{b/u, c/v\} \mid Q \rrbracket &= (\nu a)(\nu x)(\nu y)(\nu l) \\
& \quad (\ B_{fw}\langle x; a \rangle \mid B_{fw}\langle y; a \rangle \\
& \quad \mid \llbracket P\{b/u, c/v\} \rrbracket \mid \llbracket Q \rrbracket \\
& \quad \mid B_{nil}\langle l; \rangle \mid B_a^P\langle a; x, y, x, l, \tilde{z}_P \rangle \)
\end{aligned}$$

4. Implementation

Based on the presented encoding, we will now create the primitives of the core join-calculus in a language that allows us to describe actor configurations. For this task, we choose *Erlang*. In contrast to other languages offering actor frameworks, Erlang is based on the actor model from its core principles. It allows us to spawn actors, referred to as *Erlang processes*, and asynchronously send messages to them by their *process id* (PID). In contrast, when we use the word *actor*, we refer to the concept from $A\pi$: A name with a behavior and values bound to its parameters.

While PIDs are globally unique and can be used to send messages to processes located on another node, they are not flexible enough for our use case (the reasons are explained in chapter 4.2.1). Thus, we create our own, globally unique names to refer to actors. This is similar to the construct $(\nu x)P$ in $A\pi$. To be able to send messages using such a name, we use our own send function and require every actor to register itself (`register_self`), so we can maintain a registry mapping from names to PIDs. The implementation of these functions will be explored later.

4.1. Basic Translation

Terms of $A\pi$ can be translated into Erlang in a straightforward way. See for example the equivalent of the behavior definition B_{fw} (figure 4.1). The main difference is that the emission $\bar{a}\langle x, i \rangle$ and recursive instantiation $B_{fw}\langle x; a \rangle$ are composed in parallel in $A\pi$, but sequentially in Erlang. However, the message will be sent asynchronously, so its transmission and reception happen concurrently to the instantiation in the next line. Generally, we will spawn multiple Erlang processes to mirror parallel composition in $A\pi$ terms.

The translation of the behavior B_a^P is a bit more complex (figure 4.2). To obtain it, we must first pass in the process P , which is not a formal behavior parameter. The behavior itself then receives the expected parameters. Since Erlang provides built-in lists and case-expressions, we do not need to use church encoding and continuation passing anymore.

```
1 forward(X, A) ->
2   receive
3     I -> join:send(A, {X, I}),
4         forward(X, A)
5   end.
```

Figure 4.1.: behavior definition of a forwarder

```

1 definition(P) ->
2   fun Actor(A, X, Y, Flag, Payloads) ->
3     receive
4       {Channel, Payload} ->
5         case Payloads of
6           [] ->
7             Actor(A, X, Y, Channel, [Payload | Payloads]);
8           [H | T] ->
9             case Flag of
10              X ->
11                case Channel of
12                  X ->
13                    Actor(A, X, Y, Flag, [Payload | Payloads]);
14                  Y ->
15                    spawn(fun() -> P(H, Payload) end),
16                    Actor(A, X, Y, Flag, T)
17                end;
18              Y ->
19                case Channel of
20                  Y ->
21                    Actor(A, X, Y, Flag, [Payload | Payloads]);
22                  X ->
23                    spawn(fun() -> P(Payload, H) end),
24                    Actor(A, X, Y, Flag, T)
25                end
26            end
27          end
28        end
29      end.

```

Figure 4.2.: behavior definition of B_a^P

```

1 join:def(fun(X, Y) -> {
2   fun(U, V) ->
3     % process P
4   end,
5   fun() ->
6     % process Q
7   end
8 } end)

```

Figure 4.3.: creating a join definition

```

1 def(PQ) ->
2   % new names
3   A = join:create_id(act),
4   X = join:create_id(fwX),
5   Y = join:create_id(fwY),
6   Payloads = [],
7   % get user-supplied processes
8   {P, Q} = PQ(X, Y),
9   % spawn in parallel (no need to spawn lists)
10  join:spawn_actor(join_actor:definition(P), A, [X, Y, X, Payloads]),
11  join:spawn_actor(fun join_actor:forward/2, X, [A]),
12  join:spawn_actor(fun join_actor:forward/2, Y, [A]),
13  spawn(Q) .

```

Figure 4.4.: encoding of a join definition

While this is the simplest and conceptually closest translation, the real code uses a queue in place of the list of payloads. This ensures fairness, since messages that reach the actor first will also be joined first.

After defining the necessary behaviors, we can now translate the encoding itself. First we have to decide what the primitives of the join-calculus should look like in Erlang. Sending a message is just a function call and parallel composition is not necessary when all provided actions are asynchronous. A join definition $\mathbf{def} \ x\langle u \rangle | y\langle v \rangle \triangleright P \ \mathbf{in} \ Q$ can be modelled using a function that receives two processes P and Q , where both should have access to the channel names x and y , and P to u and v as well. To model this, we pass a function that receives the shared names and returns two functions (the first one taking the additional names). We can use it as shown in figure 4.3. The syntax is noisier than in the calculus, but the structure is recognizable.

The `def` function (figure 4.4) is again in close correspondence with the formal encoding. We create new names and spawn all necessary actors using the `spawn_actor` function (figure 4.5), which also registers them in our name registry. Note that Q is not registered, since it is not an actor. As shown in section 3.4.1, the encoding of any process in the join-calculus has no recipients.

Example 2 Terms such as $\mathbf{def} \ x\langle u \rangle | y\langle v \rangle \triangleright \text{print}(u, v) \ \mathbf{in} \ x\langle \text{"a"} \rangle | y\langle 1 \rangle | y\langle 2 \rangle$ can now be expressed in Erlang (figure 4.6).

```

1 spawn_actor({Actor, Channel, Args}) ->
2   spawn(fun() ->
3       join:register_self(Channel),
4       apply(Actor, [Channel | Args])
5   end).

```

Figure 4.5.: registering and spawning an actor

```

1 join:def(fun(X, Y) -> {
2   fun(U, V) ->
3     io:format("joined ~p and ~p~n", [U, V])
4   end,
5   fun() ->
6     join:send(X, "a"),
7     join:send(Y, 1),
8     join:send(Y, 2),
9   end
10 } end).

```

Figure 4.6.: join-definition (example 2)

Example 3 Also, we can create a function for single channel join-patterns using the existing primitives (figure 4.7). Such a channel can then be used as a continuation.

4.2. Distributed Implementation

We will now extend the implementation by adding the possibility to create, refer to and migrate locations as in the distributed join-calculus.

4.2.1. Migrating Actors

Running Erlang processes cannot be migrated. However, functions can be serialized, sent somewhere else and spawned there. Since our provided primitives are the only places where actors are created, we can equip them with the ability to “wrap themselves up”: stopping their execution and returning (a) a function capturing their behavior and (b) their current state, which consists only of the values of their parameters. For example, see the extended definition of the `forward` behavior (figure 4.8). The returned data structure can then be sent to another node and re-spawned there.

As this creates a new process with a new PID, we need to generate our own globally unique names that persist across migrations. Fortunately, the Erlang ecosystem already provides global process registries such as *global*¹, which we just wrap in the `join_reg` module.

Note that while actors (and thus their corresponding join-definitions) will be moved to the new location, already spawned non-actor processes cannot be migrated. However, this is not a problem in practice: These processes are usually short-lived; anytime

¹<http://erlang.org/doc/man/global.html>


```

1 def_single(PQ) ->
2   join:def(fun(X, Token) ->
3     % only give it one name
4     {P, Q} = PQ(X),
5     { fun(U, _) ->
6       % after consuming the token, emit a new one
7       join:send(Token, token), P(U)
8     end,
9     fun() ->
10      % initially emit one token
11      join:send(Token, token), Q()
12    end
13  } end).
14
15 % usage
16 def_single(fun(X) -> {
17   fun(U) ->
18     % P
19   end,
20   fun() ->
21     % Q
22   end
23 } end).

```

Figure 4.7.: a function for join-patterns consisting of only one channel

```

1 forward(X, A) ->
2   receive
3     {wrap_up, Pid, Ref} ->
4       % unregister and send itself back as data
5       join_reg:unregister_self(),
6       Pid ! {Ref, {fun forward/2, X, [A]}},
7       % forward all messages to the new location
8       join_forward:forward_on(X);
9
10   I -> join_reg:send(A, {X, I}),
11         forward(X, A)
12 end.

```

Figure 4.8.: the extended forward behavior

control flow is done through sending messages (e.g. continuations), it will happen in the new location. And neither is it from a theoretical perspective: The processes run in parallel to the code responsible for migration. Thus, they could have finished before the migration anyways. The only way to observe the difference is through the temporal order of side effects like printing to the console.

Another problem we have to consider is that a migrating actor, when it handles the `wrap_up` message and halts execution, might still have messages in its inbox. Worse, even after stopping execution, nodes that are not yet aware of the migration could try to send a message to the old PID. To prevent the loss of messages, migrating actors will immediately unregister from their name, wait until it becomes registered somewhere else and stay alive for a limited time to forward any remaining messages to the new process.

4.2.2. Locations

Locations are modelled as processes with registered location names, arranged in a tree. They keep track of their parent location, child locations and actors. These references can be PIDs instead of names to avoid the indirection of looking it up in the registry. To see why PIDs suffice, note that actors and the parent location cannot be migrated without the location itself. When a child location migrates, it is not a child location anymore and can simply be unregistered. Similarly to actors, locations can be wrapped up, unregistering themselves from their parent location and the registry, and recursively wrapping up their child locations and actors.

Locations on the second level of the tree (directly below the root) correspond to physical locations. Each node has to initialize its top location in the beginning using `join_location:create_root()`. These top locations cannot be moved.

4.2.3. Primitives for Distribution

We need to add an additional location parameter to many existing functions:

- All actors need to be spawned at a specified location.
- While the current location is implicit in the distributed join-calculus, we choose to pass it around explicitly for now to simplify the implementation.

Additionally, we implement two new constructs: `def_location` and `go`.

Location Definitions

Since we restricted ourselves to implenting the core join-calculus, we will only add location definitions with a binary join-pattern:

$$\mathbf{def} \ a [x\langle u \rangle | y\langle v \rangle \triangleright P : R] \ \mathbf{in} \ Q$$

Compared to the existing `def`, the corresponding Erlang function `def_location` (figure 4.9) receives an additional location name parameter (which is only used to generate better information for debugging). The function in the existing parameter

```

1 % takes an additional location identifier (just for debugging).
2 % PQ now also receives a location name (as first parameter).
3 def_location(Location, NewLocName, PRQ) ->
4   % new names
5   A = join_util:create_id(act),
6   X = join_util:create_id(fwX),
7   Y = join_util:create_id(fwY),
8   Payloads = queue:new(),
9   % create location
10  NewLocation = join_location:create(Location, NewLocName),
11  % get user-supplied processes
12  {P, R, Q} = PRQ(NewLocation, X, Y),
13  % spawn in parallel (no need to spawn lists)
14  join_location:spawn_actor_at(NewLocation, join_actor:definition(P),
15                               A, [X, Y, X, Payloads]),
16  join_location:spawn_actor_at(NewLocation,
17                               fun join_actor:forward/2, X, [A]),
18  join_location:spawn_actor_at(NewLocation,
19                               fun join_actor:forward/2, Y, [A]),
20  spawn(R),
21  spawn(Q).

```

Figure 4.9.: adapted location definition

now also receives the new, unique location name and returns three processes $\{P, R, Q\}$. While Q is executed at the surrounding location, R will be executed at the new location.

Migration

$go\langle a, \kappa \rangle$

Again, we require to explicitly pass the current location as an additional parameter. The function `go(Location, Destination, Continuation)` then only has to wrap up `Location` and send both the resulting data structure and `Continuation` to `Destination`. There, the data will be re-spawned and the continuation called.

```

1 join:def_location(Location, a, fun(A, X, Y) -> {
2   fun(U, V) ->
3     % P
4   end,
5   fun() ->
6     % R
7   end,
8   fun() ->
9     % Q
10  end
11 } end).

```

Figure 4.10.: using the location definition

```
rebar3 shell --sname node1
> node().
'node1@hostname'
> Root = join_location:create_root().
```

```
rebar3 shell --sname node2
> net_kernel:connect_node('node1@hostname').
true
> Root = join_location:create_root().
```

Figure 4.11.: connecting multiple Erlang shells

4.2.4. Connecting Nodes

To send a message on a channel, we need to know its name. But when we start separate nodes, they don't share any names, so we need a way to exchange some names in the beginning. On these, further names can be exchanged.

In the distributed join-calculus, we can define channels at the root of the location tree (effectively without a location), but this is not realistic for the implementation. In [11], a global name server is used to register and lookup names, but in our implementation, all names are registered in the process registry anyways. This allows us to create globally known channels in `def_globally` by using fixed Erlang atoms as keys (instead of unique, generated names).

Example 4 (Applet) We define a server that provides an applet on the globally available name `cell` (figure 4.12). When a message containing a location `A` and continuation is received on name `cell`, a new location with names `Get` and `Put` will be defined (line 4) and sent to the continuation (line 15), where they can be used. Concurrently, the newly defined location with the cell migrates to location `A` (line 11).

The client (figure 4.13) simply creates a continuation using the cell and sends its name and location to `cell` (line 16).

To make the example work, the Erlang runtimes of the two nodes need to be connected. In the shell, this can be achieved as shown in figure 4.11. Also, they each need to create a root location which can then be passed to the respective definitions. For debugging purposes, locations can print their status including actors and sublocations in a tree (figure 4.14).

```

1 def_single_globally(Location, cell, fun(Cell) -> {
2   fun({A, ContCell}) ->
3     io:format("applet requested~n"),
4     join:def_location(Location, applet, fun(Applet, Get, Put) -> {
5       fun(K, X) ->
6         io:format("got ~p from cell~n", [X]),
7         join:send(K, X)
8       end,
9       fun() ->
10        io:format("moving applet to ~p~n", [A]),
11        join:go(Applet, A, none)
12      end,
13      fun() ->
14        io:format("sending get, put refs to ~p~n", [ContCell]),
15        join:send(ContCell, {Get, Put})
16      end
17    } end)
18  end,
19  fun() ->
20    ok
21  end
22 } end).

```

Figure 4.12.: the applet server

```

1 def_single(Location, fun(Cont) -> {
2   fun({Get, Put}) ->
3     def_single(Location, fun(Print) -> {
4       fun(X) ->
5         io:format("~p~n", [X])
6       end,
7       fun() ->
8         join:send(Put, hello),
9         join:send(Put, world),
10        join:send(Get, Print),
11        join:send(Get, Print)
12      end
13    } end)
14  end,
15  fun() ->
16    join:send(cell, {Location, Cont})
17  end
18 } end).

```

Figure 4.13.: the applet client

```

> join_debug:print_status(Root).
|
|-- location {root,'one@mh-tp-ubu'} (Pid: <0.158.0>, Super: none):
| |   forward {fwY,'one@mh-tp-ubu',4} to {act,'one@mh-tp-ubu',2}
| |   forward {fwX,'one@mh-tp-ubu',3} to {act,'one@mh-tp-ubu',2}
| |   definition {act,'one@mh-tp-ubu',2} (Pid: <0.184.0>):
| |     Flag: {fwY,'one@mh-tp-ubu',4}
| |     Payloads: {[ "3" ], []}
| |
| |   |-- location {{loc,ch2},'one@mh-tp-ubu',6} (Pid: <0.201.0>, Super: <0.158.0>):
| |   |
| |   |   |-- location {{loc,ch3},'one@mh-tp-ubu',7} (Pid: <0.206.0>, Super: <0.201.0>):
| |   |   |
| |   |   |   forward {fwY,'one@mh-tp-ubu',13} to {act,'one@mh-tp-ubu',11}
| |   |   |   forward {fwX,'one@mh-tp-ubu',12} to {act,'one@mh-tp-ubu',11}
| |   |   |   definition {act,'one@mh-tp-ubu',11} (Pid: <0.225.0>):
| |   |   |     Flag: {fwY,'one@mh-tp-ubu',13}
| |   |   |     Payloads: {[unused], []}
| |   |   |   forward {fwY,'one@mh-tp-ubu',10} to {act,'one@mh-tp-ubu',8}
| |   |   |   forward {fwX,'one@mh-tp-ubu',9} to {act,'one@mh-tp-ubu',8}
| |   |   |   definition {act,'one@mh-tp-ubu',8} (Pid: <0.221.0>):
| |   |   |     Flag: {fwX,'one@mh-tp-ubu',9}
| |   |   |     Payloads: {[], [2]}
| |   |
| |   |-- location {{loc,ch1},'one@mh-tp-ubu',5} (Pid: <0.197.0>, Super: <0.158.0>):
| |
|
ok

```

Figure 4.14.: example output after a status request

5. Conclusion

As suspected, we were able to find an encoding from the core join-calculus to $A\pi$ that preserves distributability. While we did not formally prove it to be correct, we showed that the resulting terms can emulate their respective source term and can be reasonably sure that it does not change its behavior. Although the typing rules of $A\pi$ made the encoding relatively complex, it proved to be a good foundation for our prototype and could be translated very directly into Erlang source code.

When implementing the distributed version, we were able to heavily rely on Erlang's infrastructure to create a light-weight prototype in less than 400 lines of code.

5.1. Future Work

After creating a simple prototype for the core join-calculus, it could be interesting to extend it to the complex definitions of the full calculus. Work on compiling these patterns has been done in [12]. Together with support for synchronous calls and a syntax closer to the calculus (e.g. using macros or a preprocessor), it would improve usability quite a lot.

An interesting topic to expand on is the failure model of the distributed join-calculus [4], which handles failures on a per-location basis and allows detecting failure of other locations to recover from it. This approach seems to fit well with the typical pattern in Erlang of creating a *supervision trees* of actors responsible for monitoring each other.

Bibliography

- [1] K. Honda and M. Tokoro, “An object calculus for asynchronous communication,” in *ECOOP’91 European Conference on Object-Oriented Programming*, Lecture Notes in Computer Science, pp. 133–147, Springer, Berlin, Heidelberg, July 1991.
- [2] K. Peters, U. Nestmann, and U. Goltz, “On Distributability in Process Calculi,” in *Proceedings of the 22Nd European Conference on Programming Languages and Systems*, ESOP’13, (Berlin, Heidelberg), pp. 310–329, Springer-Verlag, 2013.
- [3] C. Fournet and G. Gonthier, “The Reflexive CHAM and the Join-calculus,” in *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL ’96, (New York, NY, USA), pp. 372–385, ACM, 1996.
- [4] C. Fournet, G. Gonthier, J.-J. Lévy, L. Maranget, and D. Rémy, “A Calculus of Mobile Agents,” in *Proceedings of the 7th International Conference on Concurrency Theory*, CONCUR ’96, (London, UK, UK), pp. 406–421, Springer-Verlag, 1996.
- [5] S. Conchon and F. Le Fessant, “Jocaml: mobile agents for Objective-Caml,” in *Proceedings. First and Third International Symposium on Agent Systems Applications, and Mobile Agents*, pp. 22–29, 1999.
- [6] J.-J. Lévy, “Some results in the join-calculus,” in *Theoretical Aspects of Computer Software*, Lecture Notes in Computer Science, pp. 233–249, Springer, Berlin, Heidelberg, Sept. 1997.
- [7] C. Hewitt, P. Bishop, and R. Steiger, “A Universal Modular ACTOR Formalism for Artificial Intelligence,” in *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI’73, (San Francisco, CA, USA), pp. 235–245, Morgan Kaufmann Publishers Inc., 1973.
- [8] G. Agha, *Actors: A Model of Concurrent Computation in Distributed Systems*. Cambridge, MA, USA: MIT Press, 1986.
- [9] G. Agha and P. Thati, “An Algebraic Theory of Actors and Its Application to a Simple Object-Based Language,” in *From Object-Oriented to Formal Methods*, Lecture Notes in Computer Science, pp. 26–57, Springer, Berlin, Heidelberg, 2004. DOI: 10.1007/978-3-540-39993-3_4.

- [10] P. Thati, *A theory of testing for asynchronous concurrent systems*. PhD thesis, 2003.
- [11] C. Fournet, F. L. Fessant, L. Maranget, and A. Schmitt, “JoCaml: A Language for Concurrent Distributed and Mobile Programming,” in *Advanced Functional Programming*, Lecture Notes in Computer Science, pp. 129–158, Springer, Berlin, Heidelberg, Aug. 2002.
- [12] L. Maranget and F. Le Fessant, “Compiling Join-Patterns,” in *Electronic Notes in Computer Science*, Elsevier Science Publishers, 1998.
- [13] C. Fournet, *The Join-Calculus: a Calculus for Distributed Mobile Programming*. PhD thesis, 1998.

A. Source Code

The source code ¹ and thesis ² are also provided digitally.

Listing A.1: src/join_actor.erl

```
-module(join_actor).
-include("debug.hrl").
-export([
    definition/1,
    forward/2,
    spawn_actor_here/1,
    wrap_up/1
]).

% an actor is a process that is receiving messages and is registered under
% a channel name in the registry and at a location.
% the concept is based on a behavior in the actor pi calculus.
%
% generally, it knows:
%   - its own channel name
%   - some arguments (usually to keep state)
%
% it can be represented and sent in the form:
%   {Actor, Channel, Args}
% where
%   Actor is a function taking the two arguments Channel and Args.
%
% supported operations:
%   - wrapping itself up into a data structure (ending execution)
%   - (printing its status)
%
% actors can be spawned at a location (see join_location:spawn_actor_at/4),
% which will automatically register them there and in the global registry.

% spawn an actor (as data) and register it in the registry.
% returns pid of spawned process.
spawn_actor_here({Actor, Channel, Args}) ->
    ?DEBUG("spawning actor on ~p", [Channel]),
    spawn(fun() ->
        join_reg:register_self(Channel),
        apply(Actor, [Channel | Args])
    end).

% almost as for locations, but actors don't return errors
wrap_up(Pid) ->
    Ref = make_ref(),
    Pid ! {wrap_up, self(), Ref},
    receive
        {Ref, Data} -> Data
    after 3000 -> timeout
end.
```

¹<https://gitlab.tubit.tu-berlin.de/mheinzel/join-thesis/tree/master/implementation/join>

²https://gitlab.tubit.tu-berlin.de/mheinzel/join-thesis/tree/master/thesis_heinzel.pdf

```

% a definition actor is parametrized over a process P and additionally knows:
% - the two channels it joins
% - the payloads of received, but unjoined messages on these channels
%
% corresponds to B_a in the encoding

definition(P) ->
  fun Actor(A, X, Y, Flag, Payloads) ->
    receive
      {Channel, Payload} ->
        case queue:out(Payloads) of
          {empty, _} ->
            Actor(A, X, Y, Channel, queue:in(Payload, Payloads));
          {{value, H}, T} ->
            case Flag of
              X ->
                case Channel of
                  X ->
                    Actor(A, X, Y, Flag, queue:in(Payload, Payloads));
                  Y ->
                    spawn(fun() -> P(H, Payload) end),
                    Actor(A, X, Y, Flag, T)
                end;
              Y ->
                case Channel of
                  Y ->
                    Actor(A, X, Y, Flag, queue:in(Payload, Payloads));
                  X ->
                    spawn(fun() -> P(Payload, H) end),
                    Actor(A, X, Y, Flag, T)
                end
            end
        end
      end;

    {wrap_up, Pid, Ref} ->
      % unregister and send itself back as data
      join_reg:unregister_self(),
      Pid ! {Ref, {Actor, A, [X, Y, Flag, Payloads]}},
      % forward all messages to the new location
      join_forward:forward_on(A);

    {print_status, Pid, Ref, Level} ->
      io:format(join_util:indentation(Level+1) ++ "definition ~p (Pid: ~p):~n",
        [A, self()]),
      io:format(join_util:indentation(Level+1) ++ " Flag: ~p~n", [Flag]),
      io:format(join_util:indentation(Level+1) ++ " Payloads: ~p~n", [Payloads]),
      Pid ! {Ref, ok},
      Actor(A, X, Y, Flag, Payloads);

    Other ->
      ?WARNING("definition ~p received invalid message: ~p~n",
        [self(), Other]),
      Actor(A, X, Y, Flag, Payloads)
    end
  end.

% a forward actor additionally knows:
% - the channel it has to forward to
%
% corresponds to B_c in the encoding.

forward(X, A) ->
  receive

```

```

{wrap_up, Pid, Ref} ->
    % unregister and send itself back as data
    join_reg:unregister_self(),
    Pid ! {Ref, {fun forward/2, X, [A]}},
    % forward all messages to the new location
    join_forward:forward_on(X);

{print_status, Pid, Ref, IndentLevel} ->
    io:format(join_util:indentation(IndentLevel+1) ++ "forward ~p to ~p~n",
              [X, A]),
    Pid ! {Ref, ok},
    forward(X, A);

I -> join_reg:send(A, {X, I}),
      forward(X, A)
end.

```

Listing A.2: src/join_location.erl

```

-module(join_location).
-include("debug.hrl").
-export([
    create_root/0,
    root_of/1,
    create/2,
    go/3,
    spawn_actor_at/4
]).

% a location knows:
% - its own channel name
% - its superlocation (by pid)
% - its sublocations (by pid)
% - its actors (by pid)
%
% knowing pids is enough, since the tree will be rebuilt on migration anyways.
% we can just store the new pids then.
%
% it can be represented and sent in the form:
% {Name, SubLocations, Actors}
% where
% Name is the channel name,
% SubLocations is a list of location data,
% Actors is a list of actor data.
%
% supported operations:
% - spawning a process
% - wrapping itself up into a data structure (ending execution)
% - re-spawning such a data structure
% - removing a sublocation (when it moves)
% - (printing its status)
%
% it's not possible to remove a spawned process (not actor),
% since it cannot move on its own.

location(Name, Super, Subs, Actors) ->
    receive
        {spawn_actor, Actor} ->
            Pid = join_actor:spawn_actor_here(Actor),
            location(Name, Super, Subs, [Pid | Actors]);

        {spawn_location, Data, Continuation} ->
            SubPid = spawn_here(Data),
            case Continuation of
                none -> ok;
                _ -> join_reg:send(Continuation, ok)
            end
    end

```

```

    end,
    location(Name, Super, [SubPid | Subs], Actors);

{unregister_location, Pid} ->
    location(Name, Super, lists:delete(Pid, Subs), Actors);

{wrap_up, Pid, Ref} ->
    case Super of
        none ->
            Pid ! {Ref, is_root}; % cannot migrate a root location
        _ ->
            Super ! {unregister_location, self()},
            Ss = lists:map(fun wrap_up/1, Subs),
            As = lists:map(fun join_actor:wrap_up/1, Actors),
            join_reg:unregister_self(),
            Pid ! {Ref, {ok, {Name, Ss, As}}},
            join_forward:forward_on(Name)
    end;

{print_status, Pid, Ref, Level} ->
    io:format(join_util:indentation(Level) ++ "~n"),
    io:format(join_util:indentation(Level) ++ "|-- location ~p (Pid: ~p, Super: ~p):~n",
        [Name, self(), Super]),
    lists:foreach(fun(A) -> join_debug:print_status(A, Level+1) end, Actors),
    lists:foreach(fun(S) -> join_debug:print_status(S, Level+1) end, Subs),
    Pid ! {Ref, ok},
    location(Name, Super, Subs, Actors);

Other ->
    ?WARNING("location ~p received invalid message: ~p~n",
        [self(), Other]),
    location(Name, Super, Subs, Actors)
end.

% spawn a given actor at a given location
spawn_actor_at(Location, Actor, Channel, Args) ->
    join_reg:send(Location, {spawn_actor, {Actor, Channel, Args}}).

% migrate a location to another one.
% Location will become a sublocation of Destination.
% at the new location, 'ok' will be sent on the Continuation channel.
% Continuation can also be 'none'.
go(Location, Destination, Continuation) ->
    spawn(fun() ->
        % race condition?
        try join_reg:get_pid(Location) of
            SourcePid ->
                Data = wrap_up(SourcePid),
                join_reg:send(Destination, {spawn_location, Data, Continuation})
        catch
            _:timeout -> ?WARNING("~p cannot find location ~p to go to~n",
                [Location, Destination])
        end
    end).

wrap_up(Pid) ->
    Ref = make_ref(),
    Pid ! {wrap_up, self(), Ref},
    receive
        {Ref, {ok, Data}} ->
            Data;
        {Ref, Error} ->
            io:format("could not wrap up location at ~p: ~p~n", [Pid, Error]),

```

```

        {error, Error}
    after 3000 ->
        ?WARNING("timeout when wrapping up location at ~p~n", [Pid]),
        timeout
    end.

% spawn given location data on the currently executing location process.
% does not register the location at its superlocation, but in the registry.
spawn_here(Data) ->
    Super = self(),
    spawn_at(Super, Data).

% Super is a pid.
% does not register the location at its superlocation, but in the registry.
spawn_at(Super, {Channel, Subs, Actors}) ->
    spawn(fun() ->
        % register in registry
        join_reg:register_self(Channel),
        % register actors
        As = lists:map(fun join_actor:spawn_actor_here/1, Actors),
        % register sub locations
        Ss = lists:map(fun spawn_here/1, Subs),
        location(Channel, Super, Ss, As)
    end).

% create a new location with given name (just for debugging) at a given
% superlocation.
create(Super, Name) ->
    Channel = join_util:create_id({loc, Name}),
    Data = {Channel, [], []},
    join_reg:send(Super, {spawn_location, Data, none}),
    Channel.

% creates the root location of a node.
% this currently has to be run once on each node.
create_root() ->
    Channel = root_of(node()),
    spawn_at(none, {Channel, [], []}),
    Channel.

% returns the location name of the root location of a given node.
root_of(Node) ->
    {root, Node}.

```

Listing A.3: src/join.erl

```

-module(join).
-include("debug.hrl").
-export([
    send/2,
    go/3,
    def/2,
    def_location/3,
    def_globally/4
]).

% the primitives for building join definitions.
%
% NOTE:
% wherever a Location parameter occurs, the innermost enclosing location must
% be supplied!

% re-export

```

```

send(Channel, Payload) ->
    join_reg:send(Channel, Payload).

go(Location, Destination, Continuation) ->
    join_location:go(Location, Destination, Continuation).

% binary join pattern:
%   def X(U) | Y(V) > P
%   in Q
% PQ receives the channel names of X and Y
% and must return two functions {P, Q}.
% P receives the payloads U and V.
% Q has no parameters.
%
def(Location, PQ) ->
    ?DEBUG("~p", [PQ]),
    % new names
    A = join_util:create_id(act),
    X = join_util:create_id(fwX),
    Y = join_util:create_id(fwY),
    Payloads = queue:new(),
    % get user-supplied processes
    {P, Q} = PQ(X, Y),
    % spawn in parallel (no need to spawn lists)
    join_location:spawn_actor_at(Location, join_actor:definition(P), A, [X, Y, X, Payloads]),
    join_location:spawn_actor_at(Location, fun join_actor:forward/2, X, [A]),
    join_location:spawn_actor_at(Location, fun join_actor:forward/2, Y, [A]),
    spawn(Q),
    % return actor name just for debugging
    A.

% takes an additional location identifier (just for debugging).
% PQ now also receives a location name (as first parameter).
def_location(Location, NewLocName, PRQ) ->
    ?DEBUG("~p", [PRQ]),
    % new names
    A = join_util:create_id(act),
    X = join_util:create_id(fwX),
    Y = join_util:create_id(fwY),
    Payloads = queue:new(),
    % create location
    NewLocation = join_location:create(Location, NewLocName),
    % get user-supplied processes
    % def NewLocation [ X | Y > P : R ] in Q
    {P, R, Q} = PRQ(NewLocation, X, Y),
    % spawn in parallel (no need to spawn lists)
    join_location:spawn_actor_at(NewLocation, join_actor:definition(P),
                                A, [X, Y, X, Payloads]),
    join_location:spawn_actor_at(NewLocation,
                                fun join_actor:forward/2, X, [A]),
    join_location:spawn_actor_at(NewLocation,
                                fun join_actor:forward/2, Y, [A]),
    spawn(R),
    spawn(Q),
    % return actor name just for debugging
    A.

% define two channels X and Y globally.
% X and Y must be globally unique and can be used on connected nodes as well.
%
% global names should be registered at the nodes' root location.
def_globally(Location, X, Y, PQ) ->
    ?DEBUG("~p", [PQ]),
    % new names
    A = join_util:create_id(act),

```

```

Payloads = queue:new(),
% get user-supplied processes
{P, Q} = PQ(X, Y),
% spawn in parallel (no need to spawn lists)
join_location:spawn_actor_at(Location, join_actor:definition(P),
                             A, [X, Y, X, Payloads]),
join_location:spawn_actor_at(Location,
                             fun join_actor:forward/2, X, [A]),
join_location:spawn_actor_at(Location,
                             fun join_actor:forward/2, Y, [A]),
spawn(Q),
% return actor name just for debugging
A.

```

Listing A.4: src/join_debug.erl

```

-module(join_debug).
-export([
    print_status/1,
    print_status/2
]).

% synchronous
print_status(Loc) ->
    print_status(Loc, 0).

print_status(Loc, IndentLevel) when is_pid(Loc) ->
    Ref = make_ref(),
    Loc ! {print_status, self(), Ref, IndentLevel},
    receive
        {Ref, ok} -> ok
    after 500 ->
        erlang:error(timeout)
    end;
print_status(Loc, IndentLevel) ->
    try
        Pid = join_reg:get_pid(Loc, 100),
        print_status(Pid, IndentLevel)
    catch
        error:timeout ->
            io:format(join_util:indentation(IndentLevel) ++ "XXX timeout~n")
    end.

```

Listing A.5: src/join_forward.erl

```

-module(join_forward).
-include("debug.hrl").
-export([
    forward_on/1
]).

% just receive anything and forward it to a given channel.
% the idea is to turn actors and locations into a forwarder when migrating, so
% any messages left in the mailbox or still arriving at the old pid reach their
% correct target.
%
% note that these currently broken when using gproc, since the old process
% needs to die immediately to allow the new one to register.
forward_on(Channel) ->
    % leave some time
    timer:sleep(1000),

    % make sure someone new register on the channel
    try join_reg:get_pid(Channel, 200) of
        Pid when Pid==self() ->

```



```

        ?WARNING("~p) still registered on ~p",
                [self(), Channel]),
        forward_on(Channel);
    ->
    % some-one else register, forward everything!
    forward_all(Channel)
catch
    error:timeout ->
        ?WARNING("still no-one registered on ~p",
                [Channel]),
        forward_on(Channel)
end.

% forward messages after someone else registered on the channel
forward_all(Channel) ->
    receive
        Message ->
            ?DEBUG("forwarding on ~p:~n ~p~n",
                    [Channel, Message]),
            join_reg:send(Channel, Message),
            forward_all(Channel)
    after 5000 ->
        ?DEBUG("retiring ~p...", [Channel]),
        ok
    end.
end.

```

Listing A.6: src/join_reg.erl

```

-module(join_reg).
-include("debug.hrl").
-export([
    register_self/1,
    unregister_self/0,
    get_pid/1,
    get_pid/2,
    send/2
]).

% process registry.
% can be backed by either 'gproc' or 'global'.
%
% generally, gproc is the registry we want to use, since registration only
% happens at the local registry at first and is immediately available locally.
% later it is synced to the other nodes.
% its approach of allowing processes to only register themselves fits our use
% case well.
% also, it supports waiting for a name to be registered, which is useful to us.
%
% the problem with gproc is that there is a bug in the function for unregistering
% (goodbye/0).
% processes are only unregistered when they die.
% this makes it impossible to stay around for a while
% and forward everything to the newly registered process.
% it's also impossible to just overwrite the registration in the new process.
% currently, messages in the inbox will be lost when moving (when using gproc),
% but this could be fixed in the future.

% since gproc is currently not working, we have to use the 'global' module
%-define(use_gproc, 1).

-ifdef(use_gproc).
% requires gproc application to be running

register_self(Channel) ->
    try gproc:reg({n, g, Channel})

```

```

catch
  error:_ ->
    ?DEBUG("registering on ~p failed, retrying...p",
           [Channel]),
    timer:sleep(500),
    register_self_again(Channel)
end.

register_self_again(Channel) ->
  try gproc:reg({n, g, Channel})
  catch
    error:Error ->
      % TODO: retry? rethrow?
      ?WARNING("registering on ~p failed with message:~n ~p",
               [Channel, Error]),
      % maybe the channel is still registered somewhere else?
      try gproc:where({n, g, Channel}) of
        Pid ->
          ?WARNING("~p is registered at: ~p", [Channel, Pid])
        catch
          error:badarg ->
            ?WARNING("~p is probably not registered", [Channel])
          end
        end
      end
    end.

unregister_self() ->
  ?DEBUG("self() = ~p", [self()]),
  % seems to have no effect
  gproc:goodbye().

% waits for a registration to appear
% throws error:timeout
get_pid(Channel) ->
  get_pid(Channel, 8000).
get_pid(Channel, Timeout) ->
  {Pid, _} = gproc:await({n, g, Channel}, Timeout),
  Pid.

-else.

register_self(Channel) ->
  % overwrite in case the old one is still around
  case global:re_register_name(Channel, self()) of
    yes -> ok;
    _ -> ?WARNING("registering on ~p failed~p", [Channel])
  end.

unregister_self() ->
  % needs channel name, but we can just not explicitly unregister for now,
  % since a new registration overwrites old ones.
  ok.

% no builtin waiting for registration, so we use an improvised retry mechanism
% throws error:timeout
get_pid(Channel) ->
  get_pid(Channel, [100, 500, 2000]).

% Timeout can be a single time or list of times (in ms) after which we retry
get_pid(Channel, Timeout) when is_number(Timeout) ->
  get_pid(Channel, [Timeout]);
get_pid(Channel, []) ->
  case global:whereis_name(Channel) of
    undefined ->
      erlang:error(timeout);

```

```

    Pid ->
        Pid
    end;
get_pid(Channel, [Timeout | Rest]) ->
    case global:whereis_name(Channel) of
        undefined ->
            ?DEBUG("name ~p not found, retrying in ~p ms", [Channel, Timeout]),
            timer:sleep(Timeout),
            get_pid(Channel, Rest);
        _ -> Pid
    end.
-endif.

% asynchronously looks up channel name and sends message
send(Channel, Payload) ->
    From = self(),
    spawn(fun() ->
        try get_pid(Channel) of
            Pid ->
                ?DEBUG("~p sending to ~p (~p):~n ~p",
                    [From, Channel, Pid, Payload]),
                Pid ! Payload
        catch
            _ -> % just ignore, messages can be lost anyways
                ?error:timeout -> ?WARNING("failed sending to ~p:~n ~p",
                    [Channel, Payload])
        end
    end),
    ok.

```

Listing A.7: src/join_util.erl

```

-module(join_util).
-export([
    create_id/1,
    indentation/1]).

% globally unique identifier
% uses a given tag to help with debugging
create_id(Who) ->
    {Who, node(), erlang:unique_integer([positive, monotonic])}.

% for pretty printing
indentation(Level) ->
    lists:append(lists:duplicate(Level, " | ")).

```

Listing A.8: src/join_examples.erl

```

-module(join_examples).
-compile(export_all).

print(L) ->
    join:def(L, fun(X, Y) -> {
        fun(U, V) ->
            io:format("joined ~p and ~p~n", [U, V])
        end,
        % in
        fun() ->
            join:send(X, "a"),
            join:send(X, "b"),
            join:send(Y, "1"),

```

```

        join:send(Y, "2"),
        join:send(Y, "3")
    end
} end).

% just to make it easier to define continuations
def_single(Location, PQ) ->
    join:def(Location, fun(X, Unused) ->
        {P, Q} = PQ(X),
        { fun(U, _) -> join:send(Unused, unused), P(U) end,
          fun() -> join:send(Unused, unused), Q() end}
    end).

def_single_globally(Location, GlobalName, PQ) ->
    UnusedName = join_util:create_id(unused),
    join:def_globally(Location, GlobalName, UnusedName, fun(X, Unused) ->
        {P, Q} = PQ(X),
        { fun(U, _) -> join:send(Unused, unused), P(U) end,
          fun() -> join:send(Unused, unused), Q() end}
    end).

pi_channel(L) ->
    join:def(L, fun(Send, Receive) -> {
        fun(X, K) ->
            join:send(K, X)
        end,
        % in
        fun() ->
            join:send(Send, 1),
            join:send(Send, 2),
            def_single(L, fun(K) -> {
                fun(V) -> io:format("received ~p~n", [V]) end,
                % in
                fun() -> join:send(Receive, K) end
            } end)
        end
    } end).

% prints "beep" every two seconds.
beeper(L) ->
    def_single(L, fun(Token) -> {
        fun(_) ->
            io:format("beep!~n"),
            timer:sleep(2000),
            join:send(Token, ok)
        end,
        fun() ->
            join:send(Token, ok)
        end
    }
    end).

% simplified applet example
applet_server(Location) ->
    def_single_globally(Location, cell, fun(_Cell) -> {
        fun({A, ContCell}) ->

            io:format("applet requested~n"),

            join:def_location(Location, applet, fun(Applet, Get, Put) -> {
                fun(K, X) ->
                    io:format("got ~p from cell~n", [X]),
                    join:send(K, X)
                end,

```

```

% :
fun() ->
  io:format("moving applet to ~p~n", [A]),
  join:go(Applet, A, none) % can receive messages before migrating
end,
% in
fun() ->
  % reply Get, Put to Cell
  io:format("sending get, put refs of applet to ~p~n", [ContCell]),
  join:send(ContCell, {Get, Put})
end
} end)

end,
%in
fun() ->
  % cell already registered on nameserver
  ok
end
} end) .

applet_client(Location) ->
  applet_client(Location, 2, [hello, world]).

applet_client(Location, NumGet, ToPrint) ->
  def_single(Location, fun(Cont) -> {
    fun({Get, Put}) ->

      def_single(Location, fun(Print) -> {
        fun(X) ->
          io:format("~p~n", [X])
        end,
        % in
        fun() ->
          lists:foreach(fun(V) -> join:send(Put, V) end, ToPrint),
          lists:foreach(fun(K) -> join:send(Get, K) end, lists:duplicate(NumGet, Print))
        end
      } end)

    end,
    % in
    fun() ->
      join:send(cell, {Location, Cont})
    end
  } end) .

```