Implementing the Distributed Join Calculus using Actors

Bachelorarbeit

Matthias Heinzel
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 Aπ (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 Aπ (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

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 synchonization 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 JoCaml[3] 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 JoCaml[4] 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 $\mathcal{N} = \{x, y, z, u, v, \kappa, \ldots\}$, we use $\tilde{x}, \tilde{y}, \ldots$ for tuples of names and define the syntax as follows.

			processes
P, Q	$::=$	$x\langle\tilde{v}\rangle$	emission of $\tilde{v}$ on x
	$\vert$	**def** D **in** P	definition of D in P
	$\vert$	$P\vert Q$	parallel composition
	$\vert$	$\mathbf{0}$	empty process
			definitions
D, E	$::=$	$J \triangleright P$	elementary clause
	$\vert$	$D \wedge E$	simultaneous definitions
	$\vert$	$\top$	empty definition
			join-patterns
J, K	$::=$	$x\langle\tilde{v}\rangle$	reception of $\tilde{v}$ on x
	$\vert$	$J\vert K$	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 **def** D **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 $x_1\langle\tilde{v}_1\rangle|\cdots|x_n\langle\tilde{v}_n\rangle$ and a process R. When a message is available on each channel $x_1, \ldots, x_n$, they can be consumed atomically, which bindes them to the variables $\tilde{v}_1, \ldots, \tilde{v}_n$ and starts the process R. The defined names $x_1, \ldots, x_n$ 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).

$$\begin{aligned}
\mathrm{rv}(x\langle\tilde{v}\rangle) &= \{u \in \tilde{v}\}\\
\mathrm{rv}(J|K) &= \mathrm{rv}(J) \uplus \mathrm{rv}(K)\\[1em]
\mathrm{dv}(x\langle\tilde{v}\rangle) &= \{x\}\\
\mathrm{dv}(J|K) &= \mathrm{dv}(J) \cup \mathrm{dv}(K)\\[1em]
\mathrm{dv}(J \rhd P) &= \mathrm{dv}(J)\\
\mathrm{dv}(D \wedge E) &= \mathrm{dv}(D) \cup \mathrm{dv}(E)\\
\mathrm{dv}(\top) &= \emptyset\\[1em]
\mathrm{fv}(J \rhd P) &= \mathrm{dv}(J) \cup (\mathrm{fv}(P) - \mathrm{rv}(J))\\
\mathrm{fv}(D \wedge E) &= \mathrm{fv}(D) \cup \mathrm{fv}(E)\\
\mathrm{fv}(\top) &= \emptyset\\[1em]
\mathrm{fv}(x\langle v\rangle) &= \{x\} \cup \{u \in \tilde{v}\}\\
\mathrm{fv}(\mathbf{def}\ D\ \mathbf{in}\ P) &= (\mathrm{fv}(P) \cup \mathrm{fv}(D)) - \mathrm{dv}(D)\\
\mathrm{fv}(P|Q) &= \mathrm{fv}(P) \cup \mathrm{fv}(Q)\\
\mathrm{fv}(\mathbf{0}) &= \emptyset
\end{aligned}$$

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.

$$\begin{array}{llcr}
\text{(str-join)} & \vdash P|Q & \rightleftharpoons & \vdash P, Q\\
\text{(str-null)} & \vdash \mathbf{0} & \rightleftharpoons & \vdash\\
\text{(str-and)} & D \wedge E \vdash & \rightleftharpoons & D, E \vdash\\
\text{(str-nodef)} & \top \vdash & \rightleftharpoons & \vdash\\
\text{(str-def)} & \vdash \mathbf{def}\ D\ \mathbf{in}\ P & \rightleftharpoons & D\sigma_{\mathrm{dv}} \vdash P\sigma_{\mathrm{dv}}\\[1em]
\text{(red)} & J \rhd P \vdash J\sigma_{\mathrm{rv}} & \longrightarrow & J \rhd P \vdash P\sigma_{\mathrm{rv}}
\end{array}$$

In (str-dev), the substitution σ_{dv} instantiates the defined variables $\mathrm{dv}(D)$ 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 $\mathrm{rv}(J)$. 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':

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

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

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

and thus

$$\textbf{def } x\langle u\rangle | y\langle v\rangle \rhd P \textbf{ in } x\langle a\rangle | x\langle b\rangle | y\langle c\rangle \;\longrightarrow_J\; \textbf{def } x\langle u\rangle | y\langle v\rangle \rhd P \textbf{ in } x\langle a\rangle | P\{b/u, c/v\}$$

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).

$$\begin{array}{rcl}
P, Q & ::= & x\langle u\rangle \\
 & | & P|Q \\
 & | & \textbf{def } x\langle u\rangle | y\langle v\rangle \rhd P \textbf{ in } Q
\end{array}$$

2.1.2. The Distributed Join-Calculus

In addition to an infinite set of channel names $\mathcal{N} = \{x, y, z, \kappa, \ldots\}$, we now use an infinite set of location names $\mathcal{L} = \{a, b, c, \ldots\}$. These can also be sent and received, which means that values $u, v, \ldots$ 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:

$$\begin{aligned} D, E \ &::= \ \ldots \\ &\ \ \big| \ \ a\,[D : P] \end{aligned}$$

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:

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

Also, we provide a construct for migration:

$$\begin{aligned} P, Q \ &::= \ \ldots \\ &\ \ \big| \ \ \mathrm{go}\langle b, \kappa\rangle \end{aligned}$$

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* $\varphi, \psi, \ldots \in \mathcal{L}^*$, separated by $||$. 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:

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

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 Aπ

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 Aπ 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 $\mathcal{N} = \{u, v, w, x, y, z, \ldots\}$. The set of actor configurations is defined by the following grammar.

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

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 $\overline{x}y$, 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 occurences 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 **case** x **of** $(y_1 : P_1, \ldots, y_n : P_n)$ 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 behavor instantiations $B\langle \tilde{u}; \tilde{v} \rangle$. Any instantiated identifier B must have a single defining equation of the form $B \overset{def}{=} (\tilde{u}; \tilde{v})\, x_1(z).P$, 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 occuring 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 $\bot, * \notin \mathcal{N}$ and for $X \subset \mathcal{N}$ we define $X^* = X \cup \{\bot, *\}$. For $f : X \to X^*$ we define $f^* : X^* \to X^*$ as $f^*(x) = f(x)$ for $x \in X$ and $f^*(\bot) = f^*(*) = \bot$.

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

- $f(x) = \bot$ means that x is a regular actor name,
- $f(x) = y \notin \{\bot, *\}$ 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 $x, y \in \rho$,

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

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

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

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

  Note that $\oplus$ is associative.

- If $\rho \subset \rho_1$, we define $f|_\rho : \rho \to \rho^*$ 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 $ch(\tilde{x}) : \{\tilde{x}\} \to \{\tilde{x}\}^*$ as $ch(\epsilon) = \{\}$ and if $\text{len}(\tilde{x}) = n$, $ch(\tilde{x})(x_i) = x_{i+1}$ for $1 \leq i < n$ and $ch(\tilde{x})(x_n) = \bot$.

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.

$$\text{NIL}: \quad \frac{}{\emptyset; \{\} \vdash \mathbf{0}}$$

$$\text{MSG}: \quad \frac{}{\emptyset; \{\} \vdash \overline{x}y}$$

$$\text{ACT}: \quad \frac{\rho; f \vdash P}{\{x\} \cup \tilde{z}; ch(x, \tilde{z}) \vdash xy.P} \text{ if } \begin{array}{l} \rho - \{x\} = \tilde{z}, y \notin \rho, \text{ and} \\ f = \begin{cases} ch(x, \tilde{z}) \text{ if } x \in \rho \\ ch(\epsilon, \tilde{z}) \text{ otherwise} \end{cases} \end{array}$$

$$\text{CASE}: \quad \frac{\forall\ 1 \leq i \leq n : \rho_i; f_i \vdash P_i}{(\cup_i \rho_i); (\oplus_i f_i) \vdash \textbf{case } x \textbf{ of } (y_1 : P_1, \ldots, y_n : P_n)} \text{ if } f_i \text{ are mutually compatible}$$

$$\text{COMP}: \quad \frac{\rho_1; f_1 \vdash P_1 \qquad \rho_2; f_2 \vdash P_2}{\rho_1 \cup \rho_2; f_1 \oplus f_2 \vdash P_1 \mid P_2} \text{ if } \rho_1 \cap \rho_2 = \emptyset$$

$$\text{RES}: \quad \frac{\rho; f \vdash P}{\rho - \{x\}; f|_{\rho - \{x\}} \vdash (\nu x)P}$$

$$\text{INST}: \quad \frac{}{\{\tilde{x}\}; ch(\tilde{x}) \vdash B\langle \tilde{x}; \tilde{y} \rangle} \text{ if len}(\tilde{x}) = 2 \text{ implies } x_1 \neq x_2$$

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 $B \overset{def}{=} (\tilde{x}; \tilde{y})\, x_1(z).P$ the judgement $\{\tilde{x}\}; ch(x) \vdash x_1(z).P$ must be derivable.

Semantics

The operational semantics of Aπ 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), $\overline{x}y$ (free output of message with target x and content y), $\overline{x}(y)$ (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.

$$\text{OUT}: \quad \frac{}{\overline{x}y \xrightarrow{\overline{x}y}_{A\pi} \mathbf{0}}$$

$$\text{INP}: \quad \frac{}{xy.P \xrightarrow{xz}_{A\pi} P\{z/y\}}$$

$$\text{BINP}: \quad \frac{P \xrightarrow{xy}_{A\pi} P'}{P \xrightarrow{x(y)}_{A\pi} P'} \text{ if } y \notin \text{fn}(P)$$

$$\text{RES}: \quad \frac{P \xrightarrow{\alpha}_{A\pi} P'}{(\nu y)P \xrightarrow{\alpha}_{A\pi} (\nu y)P'} \text{ if } y \notin \text{n}(\alpha)$$

$$\text{OPEN}: \quad \frac{P \xrightarrow{\overline{x}y}_{A\pi} P'}{(\nu y)P \xrightarrow{\overline{x}(y)}_{A\pi} P'} \text{ if } x \neq y$$

$$\text{PAR}: \quad \frac{P_1 \xrightarrow{\alpha}_{A\pi} P_1'}{P_1 \mid P_2 \xrightarrow{\alpha}_{A\pi} P_1' \mid P_2} \text{ if } \text{bn}(\alpha) \cap \text{fn}(P_2) = \emptyset$$

$$\text{COM}: \quad \frac{P_1 \xrightarrow{\overline{x}y}_{A\pi} P_1' \qquad P_2 \xrightarrow{xy}_{A\pi} P_2'}{P_1 \mid P_2 \xrightarrow{\tau}_{A\pi} P_1' \mid P_2}$$

$$\text{CLOSE}: \quad \frac{P_1 \xrightarrow{\overline{x}(y)}_{A\pi} P_1' \qquad P_2 \xrightarrow{xy}_{A\pi} P_2'}{P_1 \mid P_2 \xrightarrow{\tau}_{A\pi} (\nu y)P_1' \mid P_2} \text{ if } y \notin \text{fn}(P_2)$$

$$\text{BRNCH}: \quad \frac{}{\textbf{case } x \textbf{ of } (y_1 : P_1, \ldots, y_n : P_n) \xrightarrow{\tau}_{A\pi} P_i} \text{ if } x = y_i$$

$$\text{BEHV}: \quad \frac{(x_1 z.P)\{(\tilde{u}, \tilde{v})/(\tilde{x}, \tilde{y})\} \xrightarrow{\alpha}_{A\pi} P'}{B\langle\tilde{x}; \tilde{y}\rangle \xrightarrow{\alpha}_{A\pi} P'} \text{ if } B \overset{def}{=} (\tilde{x}; \tilde{y})\, x_1 z.P$$

Structural Congruence

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

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

Polyadic Communication

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

$$\begin{aligned}
[\![\overline{x}\langle y_1, \ldots, y_n\rangle]\!]_p &= (\nu u)\,(\overline{x}u \mid S_1\langle u; y_1, \ldots, y_n\rangle) \\
S_i &\overset{def}{=} (u; y_1, \ldots, y_n)\, uz.\,(\overline{z}y_i \mid S_{i+1}\langle u; y_{i+1}, \ldots, y_n\rangle) \qquad 1 \leq i < n \\
S_n &\overset{def}{=} (u; y_n)\, uz.\overline{z}y_n \\
[\![x(y_1, \ldots, y_n).P]\!]_p &= xu.(\nu z)\,(\overline{u}z \mid R_1\langle z, \hat{x}; u, \tilde{a}\rangle) \\
R_i &\overset{def}{=} (z, \hat{x}; u, \tilde{a})\, zy_i.\,(\overline{u}z \mid R_1\langle z, \hat{x}; u, \tilde{a}\rangle) \qquad 1 \leq i < n \\
R_n &\overset{def}{=} (z, \hat{x}; u, \tilde{a})\, zy_n.\,(\overline{u}z \mid [\![P]\!]_p)
\end{aligned}$$

where $\tilde{a} = \text{fn}(x(y_1, \ldots, y_n).P) - \{x\}$, and $\hat{x} = \{x\}$ if $\rho \cup \{x\}; f \vdash [\![P]\!]_p$ 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 $x(y_1, \ldots, y_n).P$ is that P is an actor with a regular name. Therefore, x can be temporary, as long is it 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.

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

3. Encoding

3.1. A Naive Approach

When introduction 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 Aπ.

Let us also define $\tilde{z}_P$ for the remainder of this chapter as the tuple of names in $\mathrm{fn}(P) \setminus \{x, y, u, v\}$. 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.

$$\begin{aligned}
[\![P|Q]\!] &= [\![P]\!] \mid [\![Q]\!] \\
[\![x\langle y\rangle]\!] &= \overline{x}\langle y\rangle \\
[\![\mathbf{def}\ x\langle u\rangle|y\langle v\rangle \rhd P \ \mathbf{in}\ Q]\!] &= (\nu x)(\nu y)\,(B\langle x, y; \tilde{z}\rangle \mid [\![Q]\!]) \\
&\quad \text{with } B \overset{def}{=} (x, y; \tilde{z})\, x(u).y(v).\,(B\langle x, y; \tilde{z}\rangle \mid [\![P]\!])
\end{aligned}$$

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

Swapping the input actions to arrive at $\{x, y\}; ch(x, y) \vdash x(u).\,(B\langle x, y; \tilde{z}\rangle \mid [\![P]\!])$ 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 Aπ 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.

$$B_{fw} \overset{def}{=} (x; a)\, x(i).\,(\overline{a}\langle x, i\rangle \mid B_{fw}\langle x; a\rangle)$$

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 Aπ 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:

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

If not, it sends head and tail on k_c:

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

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.

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

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.

$$\begin{aligned}
[\![P|Q]\!] &= [\![P]\!] \mid [\![Q]\!] \\
[\![x\langle y\rangle]\!] &= \overline{x}\langle y\rangle \\
[\![\mathbf{def}\ x\langle u\rangle|y\langle v\rangle \rhd P \ \mathbf{in}\ Q]\!] &= (\nu a)(\nu x)(\nu y)(\nu l)\ \big(\ B_a^P\langle a; x, y, x, l, \tilde{z}_P\rangle \\
&\qquad\qquad\qquad\qquad\quad \mid B_{fw}\langle x; a\rangle \mid B_{fw}\langle y; a\rangle \\
&\qquad\qquad\qquad\qquad\quad \mid B_{nil}\langle l; \rangle \\
&\qquad\qquad\qquad\qquad\quad \mid [\![Q]\!]\ \big)
\end{aligned}$$

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 $\emptyset; \{\} \vdash [\![J]\!]$ *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 $[\![J]\!] = \overline{x}\langle y\rangle$ and thus $\emptyset; \{\} \vdash [\![J]\!]$ by MSG.

Case 2 $J = P|Q$
Then we know by inductive hypothesis that $\emptyset; \{\} \vdash [\![P]\!]$ and $\emptyset; \{\} \vdash [\![Q]\!]$. Since $[\![P|Q]\!] = [\![P]\!] \mid [\![Q]\!]$, we can infer $\emptyset; \{\} \vdash [\![J]\!]$ by COMP.

Case 3 $J = \mathbf{def}\ x\langle u\rangle|y\langle v\rangle \rhd P\ \mathbf{in}\ Q$
Then we know by inductive hypothesis that $\emptyset; \{\} \vdash [\![P]\!]$ and $\emptyset; \{\} \vdash [\![Q]\!]$.

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

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

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

$$R \begin{cases} \big(\ \bar{l}\langle k_n, k_c\rangle \\ \mid k_n().(\nu l')\,\big(B_{cons}\langle l'; p, l\rangle \mid \overline{k_a}\langle c, l'\rangle\big) \\ \mid k_c(h,t).\,\textbf{case}\ f\ \textbf{of}\ \big(\ x:\ \textbf{case}\ c\ \textbf{of}\ \big(\ x: (\nu l')\,\big(B_{cons}\langle l'; p, l\rangle \mid \overline{k_a}\langle f, l'\rangle\big) \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad y: \overline{k_P}\langle h, p\rangle \mid \overline{k_a}\langle f, t\rangle\ \big) \\ \qquad\qquad\qquad\qquad\qquad\ \ y:\ \textbf{case}\ c\ \textbf{of}\ \big(\ y: (\nu l')\,\big(B_{cons}\langle l'; p, l\rangle \mid \overline{k_a}\langle f, l'\rangle\big) \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad x: \overline{k_P}\langle p, h\rangle \mid \overline{k_a}\langle f, t\rangle\ \big)\ \big) \\ \mid k_P(u,v).[\![P]\!] \\ \mid k_a(f', l').B_a^P\langle a; x, y, f', l', \tilde{z}_P\rangle\ \big) \end{cases}$$

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:

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

Now that we know that all definitions are correct, we can infer the type of $[\![J]\!]$:

$$[\![J]\!] = (\nu a)(\nu x)(\nu y)(\nu l)\, \underbrace{\big(B_a^P\langle a; x, y, x, l, \tilde{z}_P\rangle \mid B_{fw}\langle x; a\rangle \mid B_{fw}\langle y; a\rangle \mid B_{nil}\langle l;\rangle \mid [\![Q]\!]\big)}_{S}$$

Then $\{a, x, y, l\}; f \vdash S$ with $f(a) = f(x) = f(y) = f(l) = \bot$ and $\emptyset; \{\} \vdash [\![J]\!]$. □

3.4.2. Semantic Correctness

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

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

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

$$\begin{aligned} [\![\textbf{def}\ x\langle u\rangle|y\langle v\rangle \triangleright P\ \textbf{in}\ x\langle b\rangle|y\langle c\rangle|Q]\!] \equiv_{A\pi}\ & (\nu a)(\nu x)(\nu y)(\nu l) \\ & \big(\ B_{fw}\langle x; a\rangle \mid B_{fw}\langle y; a\rangle \\ & \mid \overline{x}\langle b\rangle \mid \overline{y}\langle c\rangle \mid [\![Q]\!] \\ & \mid B_{nil}\langle l;\rangle \mid B_a^P\langle a; x, y, x, l, \tilde{z}_P\rangle\ \big) \end{aligned}$$

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

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

Any of these can now be received by B_a. We pick $\overline{a}\langle x,b\rangle$ 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.

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

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 occurences 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 $R_{garbage}$.

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

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

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

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:

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

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

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 $\overline{x}\langle d\rangle$, it could be added to the list as well, resulting in $(\nu l'')(\nu l')(\nu l)\,(\cdots \mid B_{cons}\langle l''; d, l'\rangle \mid B_{cons}\langle l'; b, l\rangle \mid B_{nil}\langle l;\rangle)$.

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

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

We now perform multiple reductions at once: First, $B_{cons}\langle l'; b, l\rangle$ receives $\overline{l'}\langle k_n, k_c\rangle$. Then (because it is not empty) it sends $\overline{k_c}\langle b, l\rangle$, 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 $R'_{garbage}$).

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

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

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

The continuation k_P passes the correct parameters to $[\![P]\!]$.

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

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

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

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.

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

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.

$$
\begin{aligned}
[\![\mathbf{def}\ x\langle u\rangle|y\langle v\rangle \triangleright P\ \mathbf{in}\ P\{b/u,c/v\}|Q]\!] = {}&(\nu a)(\nu x)(\nu y)(\nu l)\\
&\big(\ B_{fw}\langle x;a\rangle \mid B_{fw}\langle y;a\rangle\\
&\mid [\![P\{b/u,c/v\}]\!] \mid [\![Q]\!]\\
&\mid B_{nil}\langle l;\rangle \mid B^P_a\langle a;x,y,x,l,\tilde{z}_P\rangle\ \big)
\end{aligned}
$$

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 asynchronically send messages to them by their *process id* (PID). In contrast, when we use the word *actor*, we refer to the concept from Aπ: 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 Aπ. 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 Aπ 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 $\overline{a}\langle x, i\rangle$ and recursive instantiation $B_{fw}\langle x; a\rangle$ are composed in parallel in Aπ, 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 Aπ 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.

```
forward(X, A) ->
  receive
    I -> join:send(A, {X, I}),
         forward(X, A)
  end.
```

Figure 4.1.: behavior definition of a forwarder

```
definition(P) ->
  fun Actor(A, X, Y, Flag, Payloads) ->
    receive
      {Channel, Payload} ->
        case Payloads of
          [] ->
            Actor(A, X, Y, Channel, [Payload | Payloads]);
          [H | T] ->
            case Flag of
              X ->
                case Channel of
                  X ->
                    Actor(A, X, Y, Flag, [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, [Payload | Payloads]);
                  X ->
                    spawn(fun() -> P(Payload, H) end),
                    Actor(A, X, Y, Flag, T)
                end
            end
        end
    end
  end.
```

Figure 4.2.: behavor definition of B_a^P

```
join:def(fun(X, Y) -> {
  fun(U, V) ->
    % process P
  end,
  fun() ->
    % process Q
  end
} end)
```

Figure 4.3.: creating a join definition

```
def(PQ) ->
  % new names
  A = join:create_id(act),
  X = join:create_id(fwX),
  Y = join:create_id(fwY),
  Payloads = [],
  % get user-supplied processes
  {P, Q} = PQ(X, Y),
  % spawn in parallel (no need to spawn lists)
  join:spawn_actor(join_actor:definition(P), A, [X, Y, X, Payloads]),
  join:spawn_actor(fun join_actor:forward/2, X, [A]),
  join:spawn_actor(fun join_actor:forward/2, Y, [A]),
  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 **def** $x\langle u\rangle|y\langle v\rangle \triangleright P$ **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 **def** $x\langle u\rangle|y\langle v\rangle \triangleright$ `print(`u, v`)` **in** $x\langle$`"a"`$\rangle|y\langle 1\rangle|y\langle 2\rangle$ can now be expressed in Erlang (figure 4.6).

```
spawn_actor({Actor, Channel, Args}) ->
  spawn(fun() ->
            join:register_self(Channel),
            apply(Actor, [Channel | Args])
        end).
```

Figure 4.5.: registering and spawning an actor

```
join:def(fun(X, Y) -> {
  fun(U, V) ->
    io:format("joined ~p and ~p~n", [U, V])
  end,
  fun() ->
    join:send(X, "a"),
    join:send(Y, 1),
    join:send(Y, 2),
  end
} 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

```
def_single(PQ) ->
  join:def(fun(X, Token) ->
    % only give it one name
    {P, Q} = PQ(X),
    { fun(U, _) ->
        % after consuming the token, emit a new one
        join:send(Token, token), P(U)
      end,
      fun() ->
        % initially emit one token
        join:send(Token, token), Q()
      end
    } end).

% usage
def_single(fun(X) -> {
  fun(U) ->
    % P
  end,
  fun() ->
    % Q
  end
} end).
```

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

```
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);

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

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

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

```
% takes an additional location identifier (just for debugging).
% PQ now also receives a location name (as first parameter).
def_location(Location, NewLocName, 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
  {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).
```

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

$$\text{go}\langle a, \kappa \rangle$$

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.

```
join:def_location(Location, a, fun(A, X, Y) -> {
  fun(U, V) ->
      % P
  end,
  fun() ->
      % R
  end,
  fun() ->
      % Q
  end
} 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).

```
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)
      end,
      fun() ->
        io:format("sending get, put refs to ~p~n", [ContCell]),
        join:send(ContCell, {Get, Put})
      end
    } end)
  end,
  fun() ->
    ok
  end
} end).
```

Figure 4.12.: the applet server

```
def_single(Location, fun(Cont) -> {
  fun({Get, Put}) ->
    def_single(Location, fun(Print) -> {
      fun(X) ->
        io:format("~p~n", [X])
      end,
      fun() ->
        join:send(Put, hello),
        join:send(Put, world),
        join:send(Get, Print),
        join:send(Get, Print)
      end
    } end)
  end,
  fun() ->
    join:send(cell, {Location, Cont})
  end
} 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 Aπ 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 Aπ 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-Orientation 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. DVD

The attached DVD contains the full source code of the implementation and a PDF of this thesis. Alternatively, both are provided on GitLab [1] [2].

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

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