A Family of λ-Calculi with Ports

Seyed Hossein Haeri[1] and Peter Van Roy[1]

UCLouvain, Belgium {hossein.haeri,peter.vanroy}@uclouvain.be

Abstract. Distributed systems programming exhibits a considerable degree of mostly-functional behaviour. On the other hand, programming every distributed system with no side-effect whatsoever is not realistic. Observational purity lends itself as a plausible alternative.
To exercise that choice for programming distributed systems, we present the λ(port) family of λ-Calculi. Ports and pure blocks are two charateristics of this family. Ports are the only single source of side-effects in the λ(port) family. Pure blocks are the linguistic support for declaring observational purity. Notably, pure blocks specify which nodes in the distributed system they are pure for. We promote a programming paradigm, in which, the programmer strives to maximise pure blocks and only leave the unavoidably effectful computations outside. Our paradigm brings added value to speculative execution, mock testing, distributed garbage collection, partial order reduction, and treatments of flaky tests.
We start the λ(port) family from its most basic member, in which messages are delivered instantly. We prove that observational equivalences of the more basic members of the family are retained upon addition of message delay, message loss, node failure, and network partitions. As such, one can freely prove observational equivalences in the most basic member without worrying about any of the successor features.

Paper Category: Research

1 Introduction

Pure functional programming can be concurrent without monadic treatments. Due to the confluence of λ-Calculus, any such piece of code can run in multiple threads, execution of which is nondeterministically interleaved by a scheduler, and still remain pure. More generally, any code consisting of concurrent entities passing asynchronous messages can be pure, provided that each entity knows from which entity its next message will be received. A classical example of such a system is a Kahn network [12]. Many more such examples are given in [28, §4].

The trivia that distributed systems are by their very nature effectful is wrong. In § A, we present the Distributed λ-Calculus [27] and show its equivalence with ordinary λ-Calculus (Theorem 7). Hence, purity of the Distributed λ-Calculus.

Important classes of distributed systems, e.g. pipelines, can be modelled using the Distributed λ-Calculus, and, are thus pure. In a pipeline of processes, each process reads messages from an asynchronous channel, does a computation with

internal state, and sends an asynchronous message to the next process. Pipelines are in wide use, to the extent that they are recognised as one of the six patterns of Microsoft Durable Functions: Function Chaining.[1]

General distributed systems are not always pure. They are not pure when their execution cannot be modelled using reductions in λ-Calculus. This is the case when information is added to the execution *during the reduction* that is not known in advance (i.e., it cannot be put in the initial λ-expression). A general distributed system interacts with the external world (meaning any system outside of it). In other words, accepting information from the external world *during its execution* is an essential property of a general distributed system. For such systems, the Distributed λ-Calculus does not help.

On the other hand, experience shows that codes written for distributed systems exhibit a substantial degree of *mostly-functional behaviour* [13]. That leads us to promote a programming paradigm for distributed systems, in which the cut between pure and effectful code is clear. In our paradigm, the programmer strives to push all the pure code to one side and all the effectful part to the other. The aim is to maximise the pure side and rather isolate the side-effects.

To that end, we advise pure blocks: a linguistic support for marking portions of code that are pure. Pure blocks are similar to pure annotations of Pearce [21] but with finer granularity. In addition, a piece of distributed code might not be universally pure; but, so to a selection of nodes in the distributed system. As such, in line with *observational purity* [1], pure blocks nominate the nodes for which they are not to have any side-effects.

Our family of $\lambda(\mathsf{port})$ calculi are inspired by $\lambda(\mathsf{fut})$ [19]. However, whereas $\lambda(\mathsf{fut})$ adds futures to λ-Calculus, the $\lambda(\mathsf{port})$ calculi add ports. Futures, per se, do not imply impurity. In $\lambda(\mathsf{fut})$, cells are used for modelling side-effects. We choose ports over cells because we find them more natural to distributed systems programming. By design, ports are our solo source of side effects. A port is an asynchronous medium for unidirectional message passing. See [28, §5] for various example on the effectiveness of ports in programming distributed systems.

We chose the $\lambda(\mathsf{port})$ ports to be multiple-read/multiple-write. Any part of the program can send messages to a port; any part of a program can begin to read the contents of a port. However, suppose a piece of code begins reading a port's contents at the port's time $t = t_0$. That reader piece of code will only have access to the messages delivered at $t \geq t_0$.

Contributions Our $\lambda(\mathsf{port})$ developments start with $\lambda(\mathsf{port})_\circ$ (§ 3). $\lambda(\mathsf{port})_\circ$ is essentially λ-Calculus with ports and pure blocks. However, $\lambda(\mathsf{port})_\circ$ messages are delivered instantly. We add message delay to $\lambda(\mathsf{port})_\circ$ to get $\lambda(\mathsf{port})_1$ (§ 4); then, message loss to get $\lambda(\mathsf{port})_2$ (§ 5); then, node failure to get $\lambda(\mathsf{port})_3$ (§ 6); and, then, network partitioning to get $\lambda(\mathsf{port})_4$ (§ 7).

We have a special purpose from this stepwise addition of distributed system features. At each step, we prove that the observational equivalences of the pre-

[1] https://docs.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-overview?tabs=csharp#chaining

decessor hold in the successor too. Those results are Theorems 3, 4, 5, and 6. Our motivation is that proving observational equivalences in $\lambda(\mathsf{port})_\circ$ are much easier and particularly less involved.

We showcase the usefulness of chaining those theorems using two results obtained for $\lambda(\mathsf{port})_\circ$: First, Theorem 1 legislates pure block expansion by moving neighbour expressions inside. Second, Theorem 2 legislates load-balancing.

We also provide two example programs written in $\lambda(\mathsf{port})_\circ$: Example 1 is a simple client-server architecture. Example 2 shows (impure) pipelining with and without load balancing.

Last but not least, we formally refute the trivia on distributed systems programming being effectful by nature (Theorem 7).

The proofs we omit here can be found in the accompanying technical reports available from the authors.

Organisation We start in § 2 where we provide concrete evidence to elucidate the our design choices for the $\lambda(\mathsf{port})$ family. $\lambda(\mathsf{port})_\circ$ to $\lambda(\mathsf{port})_4$ are presented in § 3–§ 7. Literature review comes in § 8. Future work and conclusions are in § 9. Finally, we present the Distributed λ-Calculus and its results in § A.

2 Design Philosophy

A distributed system is a collection of nodes, each of which aiming at their own computations. Such computations are sometimes independent of one another and sometimes different parts of a larger computation performed collaboratively. Nodes manage such collaborations by passing messages to one another.

With that picture in mind, a distributed system's program is a collection of computations run concurrently. Such computations might have interdependencies, i.e., the result of one might depend on that of another. More than one computation might run on a given node. In addition, one tends to reuse a computation's code when required. For instances, it is possible for a single computation to be repeated on different nodes. Similarly, different instances of a computation might run on a single node – concurrently or on different occasions. As such, in our model, the *configuration* of a distributed system's program (cf. Definition 1) consists of concurrently run computations and computation definitions (to be reused). We use "$\|$" for composition of a configuration's constituents.

Pure blocks are constructs to discipline distributed systems programming with explicit purity specifications. Pure blocks operate at the level of the constituents of a configuration or lower, down to the level of individual expressions. A pure block is a promise for not having any side effects for the nominated nodes. For example, consider $\mathsf{pure}^{\overline{a}}\ \{e_1; e_2; \ldots; \mathsf{pure}^{\overline{b}}\ \{e'_1; e'_2; \ldots e'_k\}; \ldots; e_n\}$, where $\overline{a}$ and $\overline{b}$ are disjoint. The expressions $e_1, e_2, \ldots, e_n$ promise not to have side effects for the nodes $\overline{a}$.[2] Whilst those expressions are fine to be effectful for $\overline{b}$, the expressions $e'_1, e'_2, \ldots, e'_k$ are not. Neither are the latter expressions allowed to be

[2] $\overline{a}$ is $a_1, a_2, \ldots, a_m$, for some given m that is not important in this context. We use similar list comprehensions throughout this paper.

effectful for $\overline{a}$. The promise of a pure block is currently checked at the runtime. (Cf. the (PURE) rule in Definition 2.)

We have our own take of observational purity in the context of distributed systems: We write $e_1 \sim^a e_2$ when e_1 and e_2 have similar side-effects for the node a. (Cf. Definition 4.) Note that $e_1 \sim^a e_2$ is deliberately silent about comparing the effectfulness of e_1 and e_2 for other nodes than a. Additionally, examining effectfulness might be of interest for only a selection of resources. For example, only the ports and the local heap of a node might be of interest. Our notation for that would be $(\mathsf{port}\ , \mathsf{heap}) \Vdash e_1 \sim^a e_2$. From § 3 onwards, our attention would be on ports exclusively. Thus, thereafter, we write $e_1 \sim^a e_2$ for $\mathsf{port} \Vdash e_1 \sim^a e_2$.

Here is how the rest of this section is organised: § 2.1 explains what took us this particular account of distributed systems programming. § 2.2 gives real-world examples where distributed systems programming in our fashion is beneficial. Finally, we present the $\lambda(\mathsf{port})$ family as a formal model for our fashion of distributed systems programming. § 2.3 discusses alternative formalisms for specifying the $\lambda(\mathsf{port})$ semantics.

2.1 Mostly Functional

Former study shows that $24 - 78\%$ of fields in Java are never written after an object is constructed [3]. The authors of that study conclude that "realistic Java programs exhibit a substantial degree of mostly-functional behaviour."

Our experience in the FP7 SyncFree[3] and H2020 LightKone[4] EU projects suggest that distributed systems' programs too exhibit a substantial degree of mostly functional behaviour. Nevertheless, such programs typically contain far too many parts coded under the false assumptions about their effectful nature. Such assumptions lead to the use of effectful tools. That is whilst those programs very much can be coded purely. This suggests a need for a paradigm shift in programming distributed systems.

Our conjecture is that code for most distributed systems can be designed so it is mostly pure with very few effectful parts. (Example 1 illustrates that in a client-server scenario.) That would be sheer benefit. The pure parts win the code typical purity gifts such as idempotence, referential transparency, and equational reasoning. That motivates the programming paradigm we promote: a paradigm urging knowledgeable decisions about the purity of different pieces of a distributed systems' program. Schematically, that is code like the following:

$$\begin{aligned}&\mathsf{port}\ \ p_1, p_2, \ldots \parallel f_1(\overline{x}_1) = \overline{e}_1 \parallel f_2(\overline{x}_2) = \overline{e}_2 \parallel \ldots \parallel (\mathsf{pure}^{\overline{a}}\ \{e_1; e_2\}; e_3; e_4; e_5)^{a_1} \\ &\quad \parallel (\mathsf{pure}^{\overline{b}}\ \{e_6\}; e_7; \mathsf{pure}^{\overline{c}}\ \{e_8; \mathsf{pure}^{\overline{d}}\ \{e_9; e_{10}\}\}; e_{11})^{a_2} \parallel \ldots\end{aligned}$$

2.2 Observational Purity

Flaky Tests When tests nondeterministically produce different results for the same input, they are said to be *flaky*. A very recent study [7] reports that the

[3] https://pages.lip6.fr/syncfree
[4] https://www.lightkone.eu

vast majority of developers recognise flakiness as a significant impediment. That study identifies the following as a source of flakiness: not employing wait mechanisms when making asynchronous calls. A former study [15] too reports a need in 25% of the cases for fixing the order of events to tackle flakiness.

Such solutions, unfortunately, do not work for distributed systems. Fixing the order of events defeats the purpose of such systems. Observational purity, on the other hand, can make the order of interleaving irrelevant for the test results. Two test results might be different but still observationally equivalent. A test that produces results $\{\overline{e}\}$ on a node a is no longer flaky, with that insight, when $\forall e_1, e_2 \in \{\overline{e}\}.\ e_1 \sim^a e_2$.

Another recent study [14] reports 86% failure in reproduction of flaky tests, even after 100 trials. That is a call for idempotence, which (observational) purity can guarantee. Again, our a-bisimulations (Definition 4) respond to that call.

Speculative Execution At a branching point in the execution, instead of evaluating the condition first to determine the right branch, one can execute all the possible branches greedily so the condition evaluation can proceed in parallel. Such a *speculative execution* specially appeals to distributed systems where messages are likely to be delayed due to network reasons. After all, evaluation of a condition can very well depend on the arrival of messages.

Tapus and Hicky [26] give an operational semantics for speculative execution in distributed systems. Armed with their semantics, they formally prove gain in reliability and fault tolerance due to speculative execution. In their work, a great proportion is dedicated to bookkeeping shared objects. Purity makes that redundant, and, hence, understanding speculative execution easier for distributed systems. That conjecture is backed up by an earlier empirical study on speculative execution of distributed file systems [20]. Authors of the latter work observe correctness of speculation for functions with observational purity. (Although they do not formulate that observation of theirs like us [20, §5.3].)

Mock Testing Often, in testing, one needs to test a subsystem in the absence of other collaborating subsystems. To that end, one *mock*s the behaviour of the other subsystems and composes the under-test subsystem with the mocks of the other ones. Bell and Kaiser [2] report that, in practice, mocking gives false positives/negatives because of side-effects. To tackle that, their approach prescribes a sort of observational purity: They isolate each subsystem's side-effects to themselves (but also allow other side-effect that can be reverted). Suppose that the set of all nodes is $\mathfrak{N}$. Then, in our terms, for a given node a, that is like an enclosing $\mathsf{pure}^{\mathfrak{N}\setminus a}\ \{.\}$ for every computation running on a. As such, the subsystem under test is acknowledged for its effectfulness, mock testing nevertheless succeeds because that subsystem is observationally pure to other subsystems.

Distributed Garbage Collection Garbage collection in distributed databases is known to be highly non-trivial. In the absence of a clear solution, a technique

resorted to by distributed DBMSes such as Cassandra[5] is *stop-the-world*. That technique is, however, suboptimal because it increases downtime. The reason why stop-the-world has nevertheless to be used is mutability of data: Dead data might come back to life after mutation. Observational purity dismisses that need because local mutations will not be observable to the wrong parties. For a node a, enclosing $\mathsf{pure}^{\mathfrak{N}\setminus a}\ \{.\}$ blocks help automatic inference of that dismissal.

Model Checking Partial order reduction is a classical technique used in model checking for reducing the search space. For example, the technique is widely used in SPIN [11], which targets distributed systems specifically. Intuitively, one might expect purity to help partial order reduction. That is because, in the absence of side-effects, altering the interleaving makes no observable difference. Contrary to that intuition, Déjà Fu – a stateless model checker in the pure programming language HASKELL – fails to realise that expectation [29, §8]: In the absence of **any** side-effect, the execution leaves no trace either. That is another call for observational purity, where side-effects are acknowledged but under good control. For example, suppose that $\{\overline{a}\}$ is the set of nodes in the search space. Suppose also that one is interested in examining local memory traces, exclusively. Then, $\forall a \in \{\overline{a}\}.\ \mathsf{memory} \Vdash e_1 \sim^a e_2$ legislates pruning the e_1 path once that of e_2 is already searched, and vice versa.

2.3 Obvious Alternatives

Monadic Treatments Mind the subtle difference between our work and monadic programming. Meritoriously, monads pretend there are no side-effects. They interpret an effectful computation as a state transition of a superficial universe. That is, monads are to deny the existence of side-effects (despite the track they leave in type signatures and the programming paradigm). In contrast, we acknowledge side-effects, yet, help the right nodes benefit from the purity of the correctly marked code proportions.

Algebraic Effects Monads aside, algebraic effects look hopeful. After all, using algebraic effects, one typically specifies the effects of a function alongside its type signature. We are, at the moment, investigating reasonableness of that hope. Two matters are not clear to us: First, how does one use algebraic effects for purity specification at a finer granularity than functions? That is at the level of individual statements or blocks of code. Control blocks of the language Koka[6] are the first that comes to mind. Secondly, how beneficial can algebraic effects be for promoting the notion of observational purity? Using algebraic effects in conjunction with an observational semantics [18,25] or an equational semantics [5] is worth investigating for that.

[5] http://cassandra.apache.org
[6] https://koka-lang.github.io/koka/

π-Calculus One may wonder why we build on top of λ-Calculus as opposed to π-Calculus. Those two calculi were designed for modelling computations and agent systems, respectively. As such, exceptions [22,4] aside, most programming languages results are established on top of the former. In this work, we are particularly interested in those of the pure functional programming languages. Our aim is to reuse those results. So, like more conventional λ-Calculi with futures [8,19], we build on top of λ-Calculus.

3 λ(port)$_\circ$: Ports with Instant Delivery

The λ(port)$_\circ$ syntax is tailored for our minimal working example (Example 1), in particular.

Definition 1. *The expressions (E) and configurations (G) of the* λ(port) *family are defined below, where* this *font is for keywords:*

$$\begin{aligned} e &::= x \mid c \mid \lambda x.e \mid e_1\ e_2 \mid f\ \overline{e} \mid e_1; e_2 \mid e :: s \mid \mathsf{match}\ s\ \mathsf{for}\ \{x :: s' \Rightarrow e\} \\ &\quad \mid \mathsf{send}\ e\ \mathsf{to}\ p^b \mid \mathsf{pure}^{\overline{a}}\ \{e\}, \\ g &::= e^a \mid \mathsf{port}\ p^a \mid f(\overline{x}) = \overline{e} \mid g_1 \parallel g_2. \end{aligned}$$

Assume stream names $s, s', \ldots, s_1, s_2, \cdots \in S$, where E, S, and $\mathfrak{N}$ are disjoint. The syntax for an expression e is mostly routine: variables, constants, λ-abstractions, applications, applications of named functions to a list of expressions ($f\ \overline{e}$), sequential composition, and, *cons* expressions. Our pattern matching is less routine by only allowing a single match and only against cons expressions. Then, there are sending to ports, and pure blocks. Marking e's lack of side-effects for a list of nodes $\overline{a}$, if at all, is $\mathsf{pure}^{\overline{a}}\ \{e\}$. Write $\mathsf{pure}\ \{e\}$ to indicate universal purity of e. Purity markings are verified at runtime. The final piece of the expression syntax is send operations. Configurations are node-annotated expressions, port declarations, named functions (i.e., $f(\overline{x}) = e$, with parameter list $\overline{x}$), and concurrent compositions. One annotates an expression e with a node a as e^a. Such an annotation only applies to the outermost layer. The intuition is the expression e being run on the node a. Assuming port names $p, p', \ldots, p_1, p_2, \cdots \in P$, our ports $p^a, p'^a, \ldots, p^b, p'^b, \cdots \in P \times \mathfrak{N} = P\mathfrak{N}$ consist of their name and the node they belong to. We use structural congruence for concurrent compositions: $g_1 \parallel g_2$ and $g_2 \parallel g_1$ are the same. In the λ(port) family, the number of concurrent compositions is known statically.

Definition 2 presents the small-step operational semantics that λ(port)$_\circ$ gives the syntax of Definition 1. Judgements $\jmath, \jmath', \cdots \in J$ of the operational semantics take the form $\tau : g \to_\circ \tau' : g'$. A *system setting* τ is a tuple (ρ, σ), where $\rho : S \rightharpoonup \{\bot\} \cup (E \times S)$ is an environment and $\sigma : P\mathfrak{N} \rightharpoonup S$ is a store. $\rho(s) = \bot$ denotes that s is not fresh in ρ but still unbound in the current state of the program. The clause $\jmath = \tau : g \to_\circ \tau' : g'$ sets $\jmath$ as an alias for $\tau : g \to_\circ \tau' : g'$. Define $\Delta_{\mathsf{node}}(\sigma, \sigma') = \{a \mid \exists p^a.\ \sigma(p^a) \neq \sigma'(p^a)\}$ for nodes bound differently by σ and σ'. Likewise, for $\jmath = (\sigma, \_) :\_\to_\circ (\sigma', \_) :\_$, define $\Delta_{\mathsf{node}}(\jmath) = \Delta_{\mathsf{node}}(\sigma, \sigma')$, where "_" is our wildcard. Write $g \to_\circ g'$ as a shorthand for $\tau : g \to_\circ \tau : g'$.

Write $\to_\circ^*$ for the transitive and reflexive closure of $\to_\circ$. Finally, fix a set of values ranged over by $v_1, v_2, \ldots$, including unit and all cs.

Definition 2. *Rules of the* $\lambda(\mathsf{port})_\circ$ *operational semantics follow.*

$$(\lambda x.e_1)\ e_2 \to_\circ e_1[e_2/x] \quad \text{(App-E)}$$

$$f\ \overline{e} \parallel f(\overline{x}) = e' \to_\circ e'[\overline{e}/\overline{x}] \parallel f(x) = e' \quad \text{(App-F)}$$

$$\mathsf{match}\ \bot\ \mathsf{for}\ \{x :: s \Rightarrow e\} \to_\circ \mathsf{match}\ \bot\ \mathsf{for}\ \{x :: s \Rightarrow e\} \quad \text{(Mat-1)}$$

$$\mathsf{match}\ (e :: s)\ \mathsf{for}\ \{x :: s' \Rightarrow e'\} \to_\circ e'[e/x, s/s'] \quad \text{(Mat-2)}$$

$$\frac{\tau : e_1 \to_\circ \tau' : e_1'}{\tau : e_1; e_2 \to_\circ \tau' : e_1'; e_2}\ \text{(Seq-1)} \qquad v; e \to_\circ e \quad \text{(Seq-2)}$$

$$\frac{e \to_\circ e'}{e^a \to_\circ e'^a}\ \text{(Annot)} \qquad \frac{\tau : g_1 \to_\circ \tau' : g_1'}{\tau : g_1 \parallel g_2 \to_\circ \tau' : g_1' \parallel g_2}\ \text{(Cncr)}$$

$$\frac{\jmath = \tau : e \to_\circ \tau' : e' \quad \overline{a} \notin \Delta_{\mathsf{node}}(\jmath)}{\tau : \mathsf{pure}^{\overline{a}}\ \{e\} \to_\circ \tau' : \mathsf{pure}^{\overline{a}}\ \{e'\}}\ \text{(Pure)}$$

$$\frac{p, s\ \textit{fresh}}{(\rho, \sigma) : \mathsf{port}\ p^a \to_\circ (\rho[s \mapsto \bot], \sigma[p^a \mapsto s]) : \mathsf{unit}}\ \text{(Port)}$$

$$\frac{\sigma(p^a) = s \quad \rho(s) = \bot \quad s'\ \textit{fresh}}{(\rho, \sigma) : \mathsf{send}\ e\ \mathsf{to}\ p^a \to_\circ (\rho[s' \mapsto \bot, s \mapsto e :: s'], \sigma[p^a \mapsto s']) : \mathsf{unit}}\ \text{(Send)}$$

The rules in the first six lines are standard. Inside a block that is to be pure from the viewpoint of a, the rule (Pure) allows reductions that do not alter the parts of store that pertain to a. That is, they do not declare new ports for a; neither do they send to any of a's ports. According to (Send), when an expression e is sent to p^a, the respective stream s needs to be unbound.[7] In such a case, we reduce by allocating a fresh and unbound stream s' and resetting the environment binding of s to $e :: s'$, hence, putting e at the front of the trailing stream. Note that, consequently, sends are delivered instantly in $\lambda(\mathsf{port})_\circ$, implying causal order by construction.

We also define the following utility meta-function for the programmer: When $\sigma(p^a) = s$, define $\mathsf{stream}(p^a) = \rho(s)$. Moreover, at each step, we call the part of the configuration that is being reduced the *active* expression/configuration.

The following auxiliary function (for the semantics) determines the nodes that will engaged in the evaluation of an expression or configuration. That piece of information will help us in Sections 6 and 7.

[7] One should pay special attention here not to confuse port streams with those of the lazy functional languages. Contents of the latter streams can be obtained on demand. Whereas, the tail of a port stream remains **unbound** until new items are sent to it. In other words, a port is an asynchronous FIFO communication channel. Note that it is due to their asynchronicity that ports can serve anti-causal programming.

Definition 3. *Define a function* $engaged : E \cup G \to \wp(\mathfrak{N})$ *as follows:*

$$engaged(x) = engaged(c) = \varnothing \quad engaged(\lambda x.e) = engaged(e ::\_) = engaged(e)$$
$$engaged(e_1\ e_2) = engaged(e_1; e_2) = engaged(e_1) \cup engaged(e_2)$$
$$engaged(f\ \overline{e}) = engaged(f) \cup \bigcup_{e \in \overline{e}} engaged(e)$$
$$engaged(\mathsf{match}\ \_\ \mathsf{for}\ \{\_ \Rightarrow e\}) = engaged(e)$$
$$engaged(\mathsf{send}\ e\ \mathsf{to}\ p^a) = engaged(e) \cup \{a\}$$
$$engaged(\mathsf{pure}^{\overline{a}}\ \{e\}) = engaged(e) \cup \{\overline{a}\} \quad engaged(\mathsf{port}\ p^a) = \{a\}$$
$$engaged(e^a) = engaged(e) \cup \{a\} \quad engaged(g_1 \parallel g_2) = engaged(g_1) \cup engaged(g_2).\square$$

Example 1. Let $\mathfrak{N} = \{srv, c_1, c_2\}$, where srv is a server and c_1 and c_2 are clients. The internal states of the nodes are st_s, st_1, and st_2, respectively. Based on their own state, clients form a query and send it to the server's single port (p^{srv}). The server processes queries locally. Let f_c and f_s be **pure** client-side and server-side functions. Then, the client-server program below is pure everywhere, except upon: (1) declaring the port, and, (2) sending queries to the server – **exclusively** from the viewpoint of the server.

$$\begin{aligned}&\mathsf{port}\ p^{srv} \parallel (srv\ st_s\ \mathsf{stream}(p^{srv})) \parallel (client\ st_1)^{c_1} \parallel (client\ st_2)^{c_2}\\ &\quad \parallel client(st) = \mathsf{pure}^{\overline{c}}\ \{\mathsf{send}\ (query\ st)\ \mathsf{to}\ p^{srv};\ client\ (f_c\ st)\}\\ &\quad \parallel srv\ (st, s) = \mathsf{pure}\ \{\mathsf{match}\ s\ \mathsf{for}\ \{q :: s' \Rightarrow srv\ (f_s\ (q, st), s')\}\}.\end{aligned} \quad \square$$

Let $\sigma|_a = \{p^b \mapsto s \in \sigma \mid a = b\}$ and $\rho|_a = \{s \mapsto \rho(s) \mid s \in range(\sigma|_a)\}$. Likewise, for $\tau = (\sigma, \rho)$, define $\tau|_a = (\sigma|_a, \rho|_a)$. Write $\sigma \overset{\alpha}{\equiv} \sigma'$ when σ and σ' bind ports similarly modulo the familiar α-renaming. Write $\rho \overset{\alpha}{\equiv} \rho'$ when ρ and ρ' bind streams similarly, again modulo α-renaming. Next, extend $\overset{\alpha}{\equiv}$ elementwise to τs. Finally, define $\Delta^*_{\mathsf{node}}(\tau, g) = \bigcup\{\Delta_{\mathsf{node}}(\jmath) \mid \jmath \in \tau : g \to^*_\circ \_ :\_\}$. In words, $\Delta^*_{\mathsf{node}}(\tau, g)$ is the nodes for which new ports were declared or the existing ports of which were sent to over $\tau : g \to^*_\circ \_ :\_$, where "_" is our wildcard notation.

Throughout this paper, we maintain the notation $\to_n$ for a single reduction step of $\lambda(\mathsf{port})_n$. Likewise, we write $\to^*_n$ for the reflexive and transitive closure of $\to_n$. Thus far, we have only defined $\to_\circ$ (Definition 2). In the subsequent sections, we will also define $\to_n$ for $n \in \{1, 2, 3, 4\}$.

Intuitively, according to Definition 4 below, a node observes two configurations equivalently when, for reduction steps of one configuration, the other configuration can take steps with α-equivalent impacts on the parts of the environment and store that pertain to the node.

Definition 4. *For a given node* a*, call a relation* $\mathcal{R}^a_n$ *an "*a*"-bisimulation for the reduction* $\to_n$ *when* $g_1 \mathcal{R}^a_n g_2$ *implies*

$$\forall \tau.\ \begin{cases} \tau : g_1 \to^*_n \tau_1 : g_1' \Rightarrow \exists \tau_2.\ (\tau : g_2 \to^*_n \tau_2 : g_2') \wedge (\tau_1|_a \overset{\alpha}{\equiv} \tau_2|_a) \wedge (g_1' \mathcal{R}^a_n g_2') \\ \tau : g_2 \to^*_n \tau_2 : g_2' \Rightarrow \exists \tau_1.\ (\tau : g_1 \to^*_n \tau_1 : g_1') \wedge (\tau_1|_a \overset{\alpha}{\equiv} \tau_2|_a) \wedge (g_1' \mathcal{R}^a_n g_2') \end{cases}.$$

Write $\sim^a_n$ *for the largest* a*-bisimulation for* $\to_n$*.*

Proposition 1. *$g \sim_n^a g$ for all nodes a and $n \in \{0,1,2,3,4\}$.*

In words, the following theorem states that, when an expression is proceeded by a pure block, if the expression has no side-effect for the nodes the proceeding block is pure for, one can move the expression into the pure block; the result will be the same for those nodes.

Theorem 1. *Let $e_1, e_2 \in E$ and $\overline{a} \in \mathfrak{N}$. Then, $\forall \tau.\ \overline{a} \notin \Delta^*_{\mathsf{node}}(\tau, e_1)$ implies $e_1; \mathsf{pure}^{\overline{a}}\ \{e_2\} \sim_\circ^a \mathsf{pure}^{\overline{a}}\ \{e_1; e_2\}$. Likewise, $\forall \tau.\ \overline{a} \notin \Delta^*_{\mathsf{node}}(\tau, e_2)$ implies $\mathsf{pure}^{\overline{a}}\ \{e_1\}; e_2 \sim_\circ^a \mathsf{pure}^{\overline{a}}\ \{e_1; e_2\}$.*

Proof. We prove the first implication. The second one is similar. Fix a tuple τ. There are only two possible reductions for $\tau : \mathsf{pure}^{\overline{a}}\ \{e_1; e_2\}$. First,

$$\dfrac{\dfrac{\jmath_1 = \tau : e_1 \to_\circ \tau' : e_1'}{\tau : e_1; e_2 \to_\circ \tau' : e_1'; e_2}\ (\textsc{Seq-1})}{\tau : \mathsf{pure}^{\overline{a}}\ \{e_1; e_2\} \to_\circ \tau' : \mathsf{pure}^{\overline{a}}\ \{e_1'; e_2\}}\ (\textsc{Pure}).$$

In this case, one gets

$$\dfrac{\jmath_1}{\tau : e_1; \mathsf{pure}^{\overline{a}}\ \{e_2\} \to_\circ \tau' : e_1'; \mathsf{pure}^{\overline{a}}\ \{e_2\}}\ (\textsc{Seq-1}).$$

Note that it is the condition $\forall \tau.\ \overline{a} \notin \Delta^*_{\mathsf{node}}(\tau, e_1)$ that legislates conclusion in the other direction, i.e., concluding the former derivation from the latter. Without that condition, $e_1; \mathsf{pure}^{\overline{a}}\ \{e_2\}$ might reduce in situations where $\mathsf{pure}^{\overline{a}}\ \{e_1; e_2\}$ fails at runtime. That is because (Pure) requires lack of side-effects for $\overline{a}$ for $\jmath$; or, it will fail.

The second possible reduction for $\tau : \mathsf{pure}^{\overline{a}}\ \{e_1; e_2\}$ is when $e_1 = v$ for some value v:

$$\dfrac{\dfrac{}{\tau : v; e_2 \to_\circ \tau : e_2}\ (\textsc{Seq-2})}{\tau : \mathsf{pure}^{\overline{a}}\ \{v; e_2\} \to_\circ \tau : \mathsf{pure}^{\overline{a}}\ \{e_2\}}\ (\textsc{Pure}).$$

In this case,

$$\dfrac{}{\tau : v; \mathsf{pure}^{\overline{a}}\ \{e_2\} \to_\circ \tau : \mathsf{pure}^{\overline{a}}\ \{e_2\}}\ (\textsc{Seq-2}).$$

The result follows by symmetry. ∎

The following example shows $\lambda(\mathsf{port})_\circ$ stream pipelining with and without load-balancing between ports. Many aspects of this example are designed to be rudimentary. The idea is to demonstrate the usefulness of our developments in this section despite their simplicity. In particular, one can prove observational equivalence of the above two pipelining scenarios for the output node.

Example 2. Let $\mathfrak{N} = \{bes, fes, in, out\}$, where *bes* is a back-end and *fes* is a front-end server, *in* is the input stream provider, and, *out* is our output stream node. Fix a constant c and a pure function f. In the configuration g_{lb^-} below,

in sends an infinite stream of cs to the port p^{fes} of *fes*. In return, *fes* applies f to every element in p^{fes} and sends the result to p^{out} of *out*. The configuration g_{lb^+} is similar to g_{lb^-} but with load-balancing. In g_{lb^+}, the same input stream of *in* is sent to p^{fes}. However, based on the output of a function *choose-p#*, the result of applying f to c is sent to either p_1^{bes} or p_2^{bes}. Then, *bes* sends the contents of both those ports of its, intact, to p^{out}. The configuration g_{cm} below captures the commonality between g_{lb^-} and g_{lb^+}. Note that, in g_{lb^+}, we assume let-expressions even though the $\lambda(\mathsf{port})_\circ$ syntax (Definition 1) does not include that. That is only for clarity. The example works without that too.

$$
\begin{aligned}
g_{cm} &= \mathsf{port}\ p^{fes} \parallel \mathsf{port}\ p^{out} \parallel \mathit{traffic}(x) = \{\mathsf{send}\ x\ \mathsf{to}\ p^{fes}; \mathit{traffic}\ x\} \parallel (\mathit{traffic}(c))^{in} \\
&\quad \parallel \mathit{process}(s) = \{\mathsf{match}\ s\ \mathsf{for}\ \{x :: s' \Rightarrow \\
&\qquad\qquad \mathsf{send}\ (f\ x)\ \mathsf{to}\ p^{out}; \mathit{process}\ s'\}\} \\
g_{lb^-} &= g_{cm} \parallel (\mathit{process}\ \mathsf{stream}(p^{fes}))^{fes} \\
g_{lb^+} &= g_{cr} \parallel \mathit{balance}(s) = \{\mathsf{match}\ s\ \mathsf{for}\ \{x :: s' \Rightarrow \mathsf{let}\ i = \mathit{choose\text{-}p\#}()\ \mathsf{in} \\
&\qquad\qquad\qquad\qquad \mathsf{send}\ x\ \mathsf{to}\ p_i^{bes}; \\
&\qquad\qquad\qquad\qquad \mathit{balance}\ s'\}\} \\
&\quad \parallel \mathsf{port}\ p_1^{bes} \parallel \mathsf{port}\ p_2^{bes} \parallel (\mathit{balance}\ \mathsf{stream}(p^{fes}))^{fes} \\
&\quad \parallel (\mathit{process}\ \mathsf{stream}(p_1^{bes}))^{bes} \parallel (\mathit{process}\ \mathsf{stream}(p_2^{bes}))^{bes}
\end{aligned}
$$

□

We now present a couple of utility lemmata (Lemma 1 and Lemma 2) to set the stage for proving that the output node of Example 2 observes the stream pipelining equivalently with and without the load balancing (Theorem 2).

Lemma 1. *$g \sim_\circ^a g \parallel \mathsf{port}\ p^b$, when b is fresh in g and $a \neq b$.*

Lemma 2. *$g \sim_\circ^a g \parallel f(\overline{x}) = e$, when f is fresh in g.*

Theorem 2. *In Example 2, $g_{lb^-} \sim_\circ^{out} g_{lb^+}$.*

Proof. g_{lb^-} and g_{lb^+} have configurations in common (which constitute g_{cm}) in addition to uncommon ones. According to Proposition 1, for establishing the desirable *out*-bisimulation, we can disregard those configurations in g_{cm}. Furthermore, according to Lemmata 1 and 2, we can disregard the declarations of p_1^{bes} and p_2^{bes} as well as declaration of *balance*.

In order to proceed with the remaining parts of g_{lb^-} and g_{lb^+}, take n to be the number of expressions placed thus far on p^{out}. We proceed by induction on n. The goal is to prove that, for every n, the contents of p^{out} will be exactly n repetitions of $f(c)$ for both g_{lb^-} and g_{lb^+}.

The result is trivial for $n = 0$. Suppose, that it is correct for $n = k$ as well. Suppose also that, whilst p^{out} contains k copies of $f(c)$, the configuration g_{lb^-} takes an $\rightarrow_\circ$ step that is due a reduction step in $(\mathit{process}\ \mathsf{stream}(p^{fes}))^{fes}$. The goal is clearly correct when that step is due to a (App-F), (Mat-1), or (Mat-2).

When that step is due to a (Send), the contents of p^{out} becomes $k + 1$ copies of $f(c)$. But, then, by definition of g_{lb^+}, there exists $i \in \{1, 2\}$ such that the configuration $(\mathit{process}\ \mathsf{stream}(p_i^{bes}))^{bes}$ is also $\rightarrow_\circ$-reducible by (Send), thus, placing the $(k+1)^{st}$ copy of $f(c)$ on p^{out}. The result follows by symmetry. ■

Remark 1. One may suspect Theorem 2 would fail if p^{out} is not exclusively filled with $f(c)$. That is not the case. Provided the same input stream, the stream of p^{out} can still be the same. And, that is all that is required for the result to follow (Definition 4). (Note that the choice of the active configuration gives the same non-determinism to both g_{lb^-} and g_{lb^+}.) Our choice of the input stream is only to facilitate the unity of input stream for g_{lb^-} and g_{lb^+}. □

4 $\lambda(\mathsf{port})_1$: Addition of Message Delay

Sent expressions in $\lambda(\mathsf{port})_\circ$ are delivered instantly: At the exact same step that a send operation is reduced, the sent expression is delivered to the port. (Causal order of send operations is, hence, guaranteed in $\lambda(\mathsf{port})_\circ$ by construction.) In real distributed systems, however, messages commonly experience delays.

In order to model message delay, we present a successor for $\lambda(\mathsf{port})_\circ$: In $\lambda(\mathsf{port})_1$, the syntax is the same $\lambda(\mathsf{port})_\circ$. But, the semantics (Definition 5) introduces a *message soup* $\mathcal{M} : P\mathfrak{N} \rightharpoonup \wp(E)$, in which, for every port, all the sent expressions that are still undelivered are stored. $\lambda(\mathsf{port})_1$ reinforces $\lambda(\mathsf{port})_\circ$'s notion of a system setting for τ to be a tuple $(\rho, \sigma, \mathcal{M})$.

The idea is that the sent expressions reside in the message soup, says (D-Send), until they are picked up at some later reduction by the port, says (PkUp). For a port p^a, the entry $\mathcal{M}(p^a)$ is a set of undelivered expressions thus far. Out of all those expressions, one will be picked up without paying attention to the send timestamps, thus ignoring causal order and simulating message delay. Of course, reducing port declarations is updated accordingly, cf. (D-Port).

Definition 5. *The $\lambda(\mathsf{port})_1$ operational semantics copies all the rules of $\lambda(\mathsf{port})_\circ$ except (Port) and (Send), which are replaced by the following two, respectively:*

$$\frac{p, s \text{ fresh}}{(\rho, \sigma, \mathcal{M}) : \mathsf{port}\ p^a \to_1 (\rho[s \mapsto \bot], \sigma[p^a \mapsto s], \mathcal{M}[p^a \mapsto \varnothing]) : \mathsf{unit}}\ \text{(D-Port)}$$

$$\frac{\mathcal{M}(p^a) = M}{\mathcal{M} : \mathsf{send}\ e\ \mathsf{to}\ p^a \to_1 \mathcal{M}[p^a \mapsto M \cup \{e\}] : \mathsf{unit}}\ \text{(D-Send)},$$

where $\mathcal{M} : g \to_1 \mathcal{M}' : g'$ is a shorthand for $(\sigma, \rho, \mathcal{M}) : g \to_1 (\sigma, \rho, \mathcal{M}') : g'$. Besides, the $\lambda(\mathsf{port})_1$ operational semantics adds the following pick-up rule:

$$\frac{\mathcal{M}(p^a) = M \quad e \in M \quad \sigma(p^a) = s \quad \rho(s) = \bot \quad s' \text{ fresh}}{(\rho, \sigma, \mathcal{M}) : g \to_1 (\rho[s' \mapsto \bot, s \mapsto e :: s'], \sigma[p^a \mapsto s'], \mathcal{M}[p^a \mapsto M \setminus e]) : g}\ \text{(PkUp)}.$$

In order to reuse Definition 4, we write $\mathcal{M}_1 \overset{\alpha}{\equiv} \mathcal{M}_2$ when $\mathcal{M}_1$ and $\mathcal{M}_2$ bind ports to sets of expressions that are equal up to α-renaming. Then, we update $\tau_1 \overset{\alpha}{\equiv} \tau_2$ and $\tau|_a$ to take message soups too into consideration. Our objective is to prove the validity of $\lambda(\mathsf{port})_\circ$ bisimulations in $\lambda(\mathsf{port})_1$. To that end, we first prove an *imitation lemma*:

Lemma 3. *Let* $(\rho,\sigma) : g \to_\circ (\rho',\sigma') : g'$ *and* $\mathcal{M}$ *be a message soup. Then,* $(\rho,\sigma,\mathcal{M}) : g \to_1^* (\rho',\sigma',\mathcal{M}') : g'$*, where either* $\mathcal{M}' = \mathcal{M}$ *or* $\mathcal{M}' = \mathcal{M}[p^a \mapsto \varnothing]$ *for some fresh* p*.*

Proof. Let $\jmath = (\rho,\sigma) : g \to_\circ (\rho',\sigma') : g'$. The proof is by case distinction on the last rule (R) applied in $\jmath$. When (R) is amongst those rules of $\lambda(\mathsf{port})_\circ$ that $\lambda(\mathsf{port})_1$ copies over, $\mathcal{M} = \mathcal{M}'$, trivially. For (R) = (Port), $\jmath$ takes the form

$$\frac{p, s \textit{ fresh}}{(\rho,\sigma) : \mathsf{port}\ p^a \to_\circ (\rho[s \mapsto \bot], \sigma[p^a \mapsto s]) : \mathsf{unit}}\ (\textsc{Port}),$$

in which case $\mathcal{M}' = \mathcal{M}[p^a \mapsto \varnothing]$ because, by (D-Port),

$$(\rho,\sigma,\mathcal{M}) : \mathsf{port}\ p^a \to_1 (\rho[s \mapsto \bot], \sigma[p^a \mapsto s], \mathcal{M}[p^a \mapsto \varnothing]) : \mathsf{unit}.$$

For (R) = (Send), $\jmath$ takes the form

$$\frac{\sigma(p^a) = s \quad \rho(s) = \bot \quad s' \textit{ fresh}}{(\rho,\sigma) : \mathsf{send}\ e\ \mathsf{to}\ p^a \to_\circ (\rho[s' \mapsto \bot, s \mapsto e :: s'], \sigma[p^a \mapsto s']) : \mathsf{unit}}\ (\textsc{Send}),$$

in which case, $\mathcal{M}' = \mathcal{M}$ because

$$\begin{aligned}(\rho,\sigma,\mathcal{M}) : \mathsf{send}\ e\ \mathsf{to}\ p^a &\to_1 (\rho,\sigma,\mathcal{M}[p^a \mapsto M \cup \{e\}]) : \mathsf{unit} && \text{(D-Send)}\\ &\to_1 (\rho[s' \mapsto \bot, s \mapsto e :: s'], \sigma[p^a \mapsto s'], \mathcal{M}) : \mathsf{unit} && \text{(PkUp).}\end{aligned}$$ ■

In words, Theorem 3 states that observational equivalences established in $\lambda(\mathsf{port})_\circ$ remain valid upon addition of message delays.

Theorem 3. $\sim_\circ^a \subseteq \sim_1^a$*. That is, let* g_1 *and* g_2 *be two configurations. Then,* $g_1 \sim_\circ^a g_2$ *implies* $g_1 \sim_1^a g_2$*.*

Proof. Let $g_1 \sim_\circ^a g_2$. Fix a ρ, a σ, and an $\mathcal{M}$ and suppose that

$$(\rho,\sigma) : g_1 \to_\circ^* (\rho_1',\sigma_1') : g_1'. \tag{4.1}$$

Then, by Definition 4, there exist ρ_2' and σ_2' such that

$$(\rho,\sigma) : g_2 \to_\circ^* (\rho_2',\sigma_2') : g_2' \tag{4.2}$$

where $(\rho_1',\sigma_1')|_a \overset{\alpha}{\equiv} (\rho_2',\sigma_2')|_a$ and $g_1' \sim_\circ^a g_2'$. On the other hand, by Lemma 3 and Equations (4.1) and (4.2), one gets

$$(\rho,\sigma,\mathcal{M}) : g_1 \to_1^* (\rho_1',\sigma_1',\mathcal{M}_1') : g_1' \tag{4.3}$$
$$(\rho,\sigma,\mathcal{M}) : g_2 \to_1^* (\rho_2',\sigma_2',\mathcal{M}_2') : g_2' \tag{4.4}$$

where it $\mathcal{M}_1' = \mathcal{M}_2'$, and, thus, $(\rho_1',\sigma_1',\mathcal{M}_1')|_a \overset{\alpha}{\equiv} (\rho_2',\sigma_2',\mathcal{M}_2')|_a$. But, then, given that $g_1' \sim_\circ^a g_2'$, by symmetry and Definition 4, it follows from Equations (4.3) and (4.4) that $\sim_\circ^a$ is an a-bisimulation for $\to_1$. By Definition 4, however, $\sim_1^a$ is the largest a-bisimulation for $\to_1$. Hence, $\sim_\circ^a \subseteq \sim_1^a$, as desired. ■

5 $\lambda(\mathsf{port})_2$: Allowing Message Loss

In real distributed systems, messages occasionally get lost. $\lambda(\mathsf{port})_1$'s successor $\lambda(\mathsf{port})_2$ models that by keeping the same syntax and semantics of its predecessor but also adding a single rule for message loss.

Definition 6. *The $\lambda(\mathsf{port})_2$ syntax is the same as its predecessors. The $\lambda(\mathsf{port})_2$ semantics keeps all the $\lambda(\mathsf{port})_1$ rules and adds the following rule:*

$$\frac{M = \mathcal{M}(p^a) \quad e \in M}{\mathcal{M} : g \to_2 \mathcal{M}[p^a \mapsto M \setminus e] : g} \text{ (Loss).}$$

Note that, theoretically, $\lambda(\mathsf{port})_2$ opens door to a new sort of nondeterminism. The choice of the active configuration was already nondeterministic in $\lambda(\mathsf{port})_\circ$ and $\lambda(\mathsf{port})_1$. On top, at each step, $\lambda(\mathsf{port})_2$ allows nondeterministically continuing reduction according to $\lambda(\mathsf{port})_1$ or losing information.

The imitation lemma for $\lambda(\mathsf{port})_2$ is rather trivial:

Lemma 4. *$\tau : g \to_1 \tau' : g'$ implies $\tau : g \to_2 \tau' : g'$.*

In words, Theorem 4 states that observational equivalences established in $\lambda(\mathsf{port})_1$ remain valid upon addition of message loss. Together with Theorem 3, then, they establish validity of $\lambda(\mathsf{port})_\circ$ observational equivalences upon addition of message delay and loss.

Theorem 4. *$\sim_1^a \subseteq \sim_2^a$. That is, let g_1 and g_2 be two configurations. Then, $g_1 \sim_1^a g_2$ implies $g_1 \sim_2^a g_2$.*

Proof. Similar to Theorem 3 and using Lemma 4. ■

6 $\lambda(\mathsf{port})_3$: Allowing Node Failure

In reality, at a given point in time, nodes can fail, and, depending on the failure model, come back to life later. To model node failure, $\lambda(\mathsf{port})_3$ takes a failure detector for granted. For $g \in G$ and $k \in \mathbb{N}$, write $t_n(g) = k$ when at the k^{th} step of the interleaving model for g's reduction due to $\to_n$. We drop explicit inclusion of g and n in the notation when clear. $\lambda(\mathsf{port})_3$ assumes a temporal predicate $\mathit{fail}_k(a)$ that, at $t = k$, detects failure of the node a.

Write $\mathit{fail}_n^a(g) = \{k \in \mathbb{N} \mid \mathit{fail}_k(a) \text{ when } t_n(g) = k\}$. Again, simply write $\mathit{fail}^a(g)$ when n is clear from the context.

Definition 7. *The $\lambda(\mathsf{port})_3$ syntax is that of its predecessors. The $\lambda(\mathsf{port})_3$ semantics consists of the $\lambda(\mathsf{port})_2$ rules, apart from (D-Port), (D-Send), and*

(PkUp) *that are replaced by*

$$\frac{t = k \quad \neg \mathit{fail}_k(a) \quad p, s \text{ fresh}}{(\rho, \sigma, \mathcal{M}) : \mathsf{port}\ p^a \to_3 (\rho[s \mapsto \bot], \sigma[p^a \mapsto s], \mathcal{M}[p^a \mapsto \varnothing]) : \mathsf{unit}} \text{(D-Port-F)}$$

$$\frac{t = k \quad \neg \mathit{fail}_k(a) \quad \mathcal{M}(p^a) = M}{\mathcal{M} : \mathsf{send}\ e\ \mathsf{to}\ p^a \to_3 \mathcal{M}[p^a \mapsto M \cup \{e\}] : \mathsf{unit}} \text{(D-Send-F)}$$

$$\frac{\begin{array}{c} t = k \quad \neg \mathit{fail}_k(a) \quad M = \mathcal{M}(p^a) \\ e \in M \quad \sigma(p^a) = s \quad \rho(s) = \bot \quad s' \text{ fresh} \end{array}}{\begin{array}{l} (\rho, \sigma, \mathcal{M}) : g \to_3 \\ \qquad (\rho[s' \mapsto \bot, s \mapsto e :: s'], \sigma[p^a \mapsto s'], \mathcal{M}[p^a \mapsto M \setminus e]) : g \end{array}} \text{(PkUp-F).}$$

The imitation lemma of $\lambda(\mathsf{port})_3$ is less trivial than that of $\lambda(\mathsf{port})_2$. It guarantees the imitation when the important nodes do not fail at the imitation step.

Lemma 5. *Let* $t_3(g) = k$ *and* $\forall a \in \mathit{engaged}(g).\ \neg \mathit{fail}_k(a)$. *Then,* $\tau : g \to_2 \tau' : g'$ *implies* $\tau : g \to_3 \tau' : g'$.

Definition 8. *Write* $\mathit{fail}(g_1) = \mathit{fail}(g_2)$ *when* $\mathit{fail}^a(g_1) = \mathit{fail}^a(g_2)$ *for all* $a \in \mathit{engaged}(g_1) \cup \mathit{engaged}(g_2)$. *Read* $\mathit{fail}(g_1) = \mathit{fail}(g_2)$ *as* g_1 *and* g_2 *fail likewise.*

In words, Theorem 5 states that observational equivalences established in $\lambda(\mathsf{port})_2$ remain valid even when nodes are allowed to fail, provided that fail likewise. Together with Theorems 3 and 4, then, they establish validity of $\lambda(\mathsf{port})_\circ$ observational equivalences upon addition of message delay and loss and (similar) node failure.

Theorem 5. *Let* g_1 *and* g_2 *be two configurations that fail likewise. Then,* $g_1 \sim_2^a g_2$ *implies* $g_1 \sim_3^a g_2$.

Proof. Similar to Theorem 3 and using Lemma 5. ■

Remark 2. We understand that, in practice, the condition of failing likewise is too restrictive. Definition 4 allows two configurations to be in an a-bisimulation even when they do not fail likewise. Nonetheless, that is the tightest sufficient condition that we have thus far come up with for legislating bisimulations of $\to_2$ for $\to_3$. Finding a less restrictive condition is subject for future research. □

7 $\lambda(\mathsf{port})_4$: Network Partitions

It is common for a distributed system to experience network partitions, at least temporarily. We present $\lambda(\mathsf{port})_4$ – a successor of $\lambda(\mathsf{port})_3$ – to model that. $\lambda(\mathsf{port})_4$ assumes a temporal predicate $[a] =_k [b]$, which, when $t = k$, determines whether nodes a and b belong to the same partition.

The $\lambda(\mathsf{port})_4$ semantics uses *thickened* message soup, over the set of which $\mathcal{T}$ ranges. It manipulates the message soup so it also records the node from which

a given send operation is performed. Thus, $\mathcal{T} : P\mathfrak{N} \rightharpoonup \wp(E \times \mathfrak{N})$. A pick-up in $\lambda(\mathsf{port})_4$ is allowed only when, at that very pick-up step, the source node and the destination node of a send operation are at the same partition. A system setting in $\lambda(\mathsf{port})_4$ is a tuple $(\sigma, \rho, \mathcal{T})$.

Definition 9. *The $\lambda(\mathsf{port})_4$ syntax is that of $\lambda(\mathsf{port})_\circ$. The $\lambda(\mathsf{port})_4$ semantics retains the rules of $\lambda(\mathsf{port})_3$ except (D-Port-F), (D-Send-F), and (PkUp-F), which it replaces by the following rules, respectively:*

$$\frac{t = k \quad \neg \mathit{fail}_k(a) \quad p, s \text{ fresh}}{(\rho, \sigma, \mathcal{T}) : \mathsf{port}\ p^a \to_4 (\rho[s \mapsto \bot], \sigma[p^a \mapsto s], \mathcal{T}[p^a \mapsto \varnothing]) : \mathsf{unit}} \text{(P-Port-F)}$$

$$\frac{t = k \quad \neg \mathit{fail}_k(b) \quad \mathcal{T}(p^a) = T}{\mathcal{T} : (\mathsf{send}\ e\ \mathsf{to}\ p^a)^b \to_4 \mathcal{T}[p^a \mapsto T \cup \{e^b\}] : \mathsf{unit}} \text{(P-Send-F)}$$

$$\frac{\begin{array}{c} t = k \quad \neg \mathit{fail}_k(a) \quad T = \mathcal{T}(p^a) \quad e^b \in T \\ [a] =_k [b] \quad \sigma(p^a) = s \quad \rho(s) = \bot \quad s' \text{ fresh} \end{array}}{\begin{array}{l}(\rho, \sigma, \mathcal{T}) : g \to_4 \\ \qquad (\rho[s' \mapsto \bot, s \mapsto e :: s'], \sigma[p^a \mapsto s'], \mathcal{T}[p^a \mapsto T \setminus e^b]) : g\end{array}} \text{(P-PkUp-F).}$$

When in conflict, (P-Send-F) has priority over (Annot).

Here is the setup required for reusing Definition 4: Write $e^a \overset{\alpha}{\equiv} e'^{a'}$ when $a = a'$ and $e \overset{\alpha}{\equiv} e'$. Let $EA = \{e_1^a, e_2^a, \ldots, e_n^a\}$ and $EA' = \{e_1'^{a'}, e_2'^{a'}, \ldots, e_m'^{a'}\}$. Write $EA \overset{\alpha}{\equiv} EA'$ when $n = m$ and for each e_i^a, there is one and only one $e_j'^{a'}$ such that $e_i^a \overset{\alpha}{\equiv} e_j'^{a'}$. Next, write $\mathcal{T} \leqq_\alpha \mathcal{T}'$ when, for $\mathcal{T}(p^a) = EA$, there exists EA' such that $\mathcal{T}'(p^a) = EA'$ and $EA \overset{\alpha}{\equiv} EA'$. Then, write $\mathcal{T} \overset{\alpha}{\equiv} \mathcal{T}'$ when $\mathcal{T} \leqq_\alpha \mathcal{T}'$ and $\mathcal{T}' \leqq_\alpha \mathcal{T}$. Finally, define $\mathcal{T}|_a = \{p^b \mapsto T \in \mathcal{T} \mid a = b\}$. We now update $\tau_1 \overset{\alpha}{\equiv} \tau_2$ and $\tau|_a$ to take the thickened message soups into consideration as well.

Let $\mathbb{M}$ and $\mathbb{T}$ be the set of all message soups and thickened message soups. Define a function $[\![.]\!]^- : \mathbb{T} \to \mathbb{M}$ for stripping the node information off a thickened soup: $[\![\mathcal{T}]\!]^- = \{p^a \mapsto \{\overline{e}\} \mid \exists \overline{b} \in \mathfrak{N}.\ \mathcal{T}(p^a) = \{(\overline{e}, \overline{b})\}\}$. We also overload $[\![.]\!]^-$ so that $[\![(\sigma, \rho, \mathcal{T})]\!]^- = (\sigma, \rho, [\![\mathcal{T}]\!]^-)$.

Definition 10. *Define the partition time of a configuration as: partition(g) = $\{k \in \mathbb{N} \mid \exists a_1, a_2 \in \mathit{engaged}(g).\ [a_1] \neq_k [a_2]\}$.*

$\lambda(\mathsf{port})_4$'s imitation lemma depends on the lack of partitions amongst the relevant nodes of a $\to_4$ reduction.

Lemma 6. *Suppose that $t_4(g) = k \notin \mathit{partition}(g)$. Then, $\tau : g \to_3 \tau' : g'$ implies $\tau^+ : g \to_4 \tau'^+ : g'$, where $\tau = [\![\tau^+]\!]^-$ and $\tau' = [\![\tau'^+]\!]^-$.*

In words, Theorem 6 states that observational equivalences established in $\lambda(\mathsf{port})_3$ remain valid when, in our system, networks can be partitioned – subject to conditions. Together with Theorems 3, 4, and 5, then, they establish validity of $\lambda(\mathsf{port})_\circ$ observational equivalences upon addition of message delay and loss, node failure, and network partitioning.

Theorem 6. *$g_1 \sim_3^a g_2$ implies $g_1 \sim_4^a g_2$, when $partition(g_1) = partition(g_2)$.*

Proof. Similar to Theorem 3 and using Lemma 6. ■

Remark 3. Again, we understand that the condition of similar partitioning over the reductions g_1 and g_2 might be impractically restrictive. After all, both g_1 and g_2 might still be $\rightarrow_4$-reducible even with partitions amongst nodes engaged in their reduction. Nonetheless, that is the tightest sufficient condition that we have thus far come up with for legislating bisimulations of $\rightarrow_3$ for $\rightarrow_4$. Finding a less restrictive condition is subject for future research. □

Corollary 1. *Let $e_1, e_2 \in E$ and $\overline{a} \in \mathfrak{N}$. Then, $\forall \tau.\ \overline{a} \notin \Delta^*_{\mathsf{node}}(\tau, e_1)$ implies $e_1; \mathsf{pure}^{\overline{a}}\ \{e_2\} \sim_4^a \mathsf{pure}^{\overline{a}}\ \{e_1; e_2\}$. Likewise, $\forall \tau.\ \overline{a} \notin \Delta^*_{\mathsf{node}}(\tau, e_2)$ implies $\mathsf{pure}^{\overline{a}}\ \{e_1\}; e_2 \sim_4^a \mathsf{pure}^{\overline{a}}\ \{e_1; e_2\}$.*

Corollary 2. *$g \sim_4^a g \parallel \mathsf{port}\ p^b$, when b is fresh in g and $a \neq b$.*

Corollary 3. *$g \sim_4^a g \parallel f(\overline{x}) = e$, when f is fresh in g.*

Corollary 4. *In Example 2, $g_{lb^-} \sim_4^{out} g_{lb^+}$.*

Of the above corollaries, it is only the latter for which the value of Theorems 3, 4, 5, and 6 will be noticed. We invite the reader to try proving Corollary 4 directly to witness the unnecessary noise involved.

8 Literature Review

Barnett et al. [1] were the first to consider universal insufficient in practice and recommends observational purity. Naumann [16] builds on that and gives a formal semantics to different sorts of purity: weak vs strong. Salcianu and Rinard [24] define a function pure when it does not mutate locations existing in the program state right before the invocation of the method. According to their notion of observational purity, mutation upon creation is permissible. A taxonomy for the sorts of purity is given by Helm et al. [10]. Garrigue's seminal work [9] seems to be the origin of purity analysis. Nicolay et. al [17] use abstract interpretation for that purpose.

Pearce [21] is the first to suggest linguistic support for marking purity of code fragments. His pure annotations only decorate a function as-a-whole. Pearce does not consider distributed systems. Pitidis and Sagonas [23] explore extending Erlang with guard expressions for purity specification for functions. They do not support nomination of the observer.

Knight [13] is the first to recognise mostly-functional behaviour. He suggests a successor paradigm of ours but with no focus on distributed systems.

Desai et al. [6] tackle compositional testing of distributed systems. Wilcox, Sergey, and Tatlock [30] enable language-based verification of such systems.

9 Conclusions and Future Work

We present the $\lambda(\mathsf{port})$ family: λ-calculi with ports and pure blocks, message delay, message loss, node failure, and network partitioning. $\lambda(\mathsf{port})$ models the mostly-functional nature of distributed systems programming. We promote a programming paradigm that is found in that nature. Pure blocks give fine-grain linguistic support for observational purity per node. $\lambda(\mathsf{port})$ developments seem promising for speculative execution, mock testing, distributed garbage collection, partial order reduction, and treatments of flaky tests.

Exploring the benefits of the $\lambda(\mathsf{port})$ family for each of the above areas is a complete new line of research. On a smaller scale, extending the $\lambda(\mathsf{port})$ family to model elastic distributed systems with more realistic features is our immediate next goal. We also aim at providing more linguistic support for pure distributed systems programming. Finally, we would like to seek better sufficient conditions for porting $\lambda(\mathsf{port})_2$'s observational equivalences to its successors.

A The Distributed λ-Calculus

The purpose of this section is to present a pure calculus for distributed programming (Definition 12) that is computationally equivalent to the ordinary λ-calculus (Theorem 7). We begin by fixing our definition of the ordinary λ-calculus:

Definition 11. *Let $x, y, z, \ldots, x', y', z', \ldots, x_1, y_1, z_1, \ldots$ range over a countably infinite set of variable X. Ordinary λ-terms (ranged over by $t_1, t_2, t_3, \ldots, t', t'', \ldots$) are defined as: $\lambda_o \ni t ::= x \mid (\lambda x.t) \mid (t\ t)$. Reductions on the ordinary λ-terms are: $\lambda x.t \xrightarrow{o}_{\alpha} \lambda y.t[y/x]$ (α) $\quad (\lambda x.t_1)\ t_2 \xrightarrow{o}_{\beta} t_1[t_2/x]$ (β) $\quad (\lambda x.(t\ x)) \xrightarrow{o}_{\eta} t$ (η), where the usual capture-avoiding measures apply.*

Then, we give the formal definition of the distributed λ-calculus:

Definition 12. *Let $a, b, c, \ldots, a', b', c', \ldots$ range over a set of nodes ($\mathfrak{N}$). The distributed λ-terms are defined as: $\lambda_d \ni t^a = x^a \mid (\lambda x.t^a)^b \mid (t^a\ t^b)^c$. Reductions on the distributed λ-terms follow, where the usual capture-avoiding measures apply:*

$$(\lambda x.t^a)^a \xrightarrow{d}_{\alpha} (\lambda y.t^a[y^a/x])^a \ (\alpha_{\mathbf{d}}) \qquad ((\lambda x.t_1^a)^a\ t_2^a)^a \xrightarrow{d}_{\beta} t_1^a[t_2^a/x] \ (\beta_{\mathbf{d}})$$
$$(\lambda x.(t^a\ x^a)^a)^a \xrightarrow{d}_{\eta} t^a \qquad (\eta_{\mathbf{d}}) \qquad t^a \xrightarrow{d}_{\mu} t^b \qquad (\mu_{\mathbf{d}}).$$

Intuitively, t^a is the term t running on the node a. The $(\mu_{\mathbf{d}})$ rule below captures communication (or, mobility, hence "μ") **without** side-effects. Upon $t^a \xrightarrow{d}_{\mu} t^b$, the term t that was running on a gets to run on b. In other words, t is mobilised from a to b.

It now is time to define what it means to transform terms back and forth between the ordinary and distributed λ-calculi.

Definition 13. $[\![.]\!]^- : \lambda_d \to \lambda_o$ *strips the node information from a distributed* λ*-term:* $[\![x^a]\!]^- = x, \quad [\![(\lambda x.t^a)^b]\!]^- = \lambda x.[\![t^a]\!]^-, \quad [\![(t^a\ t^b)^c]\!]^- = [\![t^a]\!]^-\ [\![t^b]\!]^-$*. In the reverse direction,* $[\![.]\!]^{+\cdot} : \lambda_o \times \mathfrak{N} \to \lambda_d$ *annotates an ordinary* λ*-term with a fixed node:* $[\![x]\!]^{+a} = x^a, \quad [\![\lambda x.t]\!]^{+a} = (\lambda x.[\![t]\!]^{+a})^a, \quad [\![t_1\ t_2]\!]^{+a} = ([\![t_1]\!]^{+a}\ [\![t_2]\!]^{+a})^a$.

The main result of this section is the following theorem. In words, it states that the ordinary and distributed λ-calculi are computationally equivalent. That is, every computation in one has a corresponding in the other.

Theorem 7. $t \overset{o}{\twoheadrightarrow} t'$ *implies* $[\![t]\!]^{+a} \overset{d}{\twoheadrightarrow} [\![t']\!]^{+a}$*, for every node a. Likewise,* $t^a \overset{d}{\twoheadrightarrow} t'^b$ *implies* $[\![t^a]\!]^- \overset{o}{\twoheadrightarrow} [\![t'^b]\!]^-$*, for every node a and b.*

References

1. M. Barnett, D. A. Naumann, W. Schulte, and Q. Sun. 99.44% Pure: Useful Abstractions in Specifications. In J. Malenfant and B. M. Østvold, editors, 6^{th} *FTfJP*. Springer Berlin Heidelberg, June 2004.
2. J. Bell and G. E. Kaiser. Unit Test Virtualization with VMVM. In P. Jalote, L. C. Briand, and A. van der Hoek, editors, 36^{th} *ICSE*, pages 550–561. ACM, May 2014.
3. W. C. Benton and C. N. Fischer. Mostly-Functional Behavior in Java Programs. In N. D. Jones and M. Müller-Olm, editors, 10^{th} *VMCAI*, volume 5403 of *LNCS*, pages 29–43. Springer, January 2009.
4. S. Conchon and F. Le Fessant. Jocaml: Mobile Agents for Objective-Caml. In $1^{st}/3^{rd}$ *ASA/MA*, pages 22–29. IEEE Computer Society, October 1999.
5. L. Correnson. Equational Semantics. *Inf. Slovenia*, 24(3), 2000.
6. A. Desai, A. Phanishayee, S. Qadeer, and S. A. Seshia. Compositional Programming and Testing of Dynamic Distributed Systems. *PACMPL*, 2(OOPSLA):159:1–159:30, 2018.
7. M. Eck, F. Palomba, M. Castelluccio, and A. Bacchelli. Understanding Flaky Tests: The Developer's Perspective. In M. Dumas, D. Pfahl, S. Apel, and A. Russo, editors, 27^{th} *SIGSOFT FSE* / 18^{th} *ESEC*, pages 830–840. ACM, August 2019.
8. C. Flanagan and M. Felleisen. The Semantics of Future and an Application. *JFP*, 9(1):1–31, 1999.
9. J. Garrigue. Relaxing the Value Restriction. In 3^{rd} *APLAS*, pages 31–45, November 2002.
10. D. Helm, F. Kübler, M. Eichberg, M. Reif, and M. Mezini. A Unified Lattice Model and Framework for Purity Analyses. In S. Becker, I. Bogicevic, G. Herzwurm, and S. Wagner, editors, *SE/SWM*, volume P-292 of *LNI*, pages 51–52. GI, February 2019.
11. G. J. Holzmann. The Model Checker Spin. *IEEE Trans. Soft. Eng.*, 23(5):279–295, 1997.
12. G. Khan. The Semantics of a Simple Language for Parallel Programming. *Inf. Proc.*, 74:471–475, 1974.
13. T. F. Knight. An Architecture for Mostly Functional Languages. In 4^{th} *LFP*, pages 105–112. ACM, August 1986.
14. W. Lam, P. Godefroid, S. Nath, A. Santhiar, and S. Thummalapenta. Root Causing Flaky Tests in a Large-Scale Industrial Setting. In D. Zhang and A. Møller, editors, 28^{th} *ISSTA*, pages 101–111. ACM, July 2019.

15. Q. Luo, F. Hariri, L. Eloussi, and D. Marinov. An Empirical Analysis of Flaky Tests. In S.-C. Cheung, A. Orso, and M.-A. D. Storey, editors, 22*nd FSE*, pages 643–653. ACM, November 2014.
16. D. A. Naumann. Observational Purity & Encapsulation. *TCS*, 376(3):205–224, 2007.
17. J. Nicolay, Q. Stiévenart, W. De Meuter, and C. De Roover. Purity Analysis for JavaScript through Abstract Interpretation. *J. Soft. Evol. & Proc.*, 29(12), 2017.
18. J. Niehren, D. Sabel, M. Schmidt-Schauß, and J. Schwinghammer. Observational Semantics for a Concurrent Lambda Calculus with Reference Cells and Futures. *ENTCS*, 173:313–337, 2007.
19. J. Niehren, J. Schwinghammer, and G. Smolka. A Concurrent Lambda Calculus with Futures. *TCS*, 364(3):338–356, 2006.
20. E. B. Nightingale, P. M. Chen, and J. Flinn. Speculative Execution in a Distributed File System. *ACM Trans. Comp. Sys.*, 24(4):361–392, 2006.
21. D. J. Pearce. JPure: A Modular Purity System for Java. In J. Knoop, editor, 20^{th} *CC*, volume 6601 of *LNCS*, pages 104–123. Springer, April 2011.
22. B. C. Pierce and D. N. Turner. Pict: A Programming Language Based on the π-Calculus. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*, pages 455–494. MIT Press, 2000.
23. M. Pitidis and K. Sagonas. Purity in Erlang. In J. Hage and M. T. Morazán, editors, 22^{nd} *IFL (Revised Selected Papers)*, volume 6647 of *LNCS*, pages 137–152. Springer, September 2010.
24. A. Salcianu and M. C. Rinard. Purity and Side Effect Analysis for Java Programs. In R. Cousot, editor, 6^{th} *VMCAI*, volume 3385 of *LNCS*, pages 199–215. Springer, January 2005.
25. M. Schmidt-Schauß, D. Sabel, J. Niehren, and J. Schwinghammer. Observational Program Calculi and the Correctness of Translations. *TCS*, 577:98–124, 2015.
26. C. Tapus and J. Hickey. Distributed Speculative Execution for Reliability and Fault Tolerance: An Operational Semantics. *Dist. Comp.*, 21(6):433–455, 2009.
27. P. Van Roy. Why Time is Evil in Distributed Systems and What To Do About It. CodeBEAM Keynote Talk. Available online at: https://www.youtube.com/watch?v=NBJ5SiwCNmU, May 2019.
28. P. Van Roy and S. Haridi. *Concepts, Techniques, and Models of Computer Programming*. MIT Press, 2004.
29. M. Walker. Déjà Fu: A Concurrency Testing Library for Haskell. Technical report, Dept. Comp. Sci., U. York, 2016.
30. J. R. Wilcox, I. Sergey, and Z. Tatlock. Programming Language Abstractions for Modularly Verified Distributed Systems. In B. S. Lerner, R. Bodík, and S. Krishnamurthi, editors, 2^{nd} *SNAPL*, volume 71 of *LIPIcs*, pages 19:1–19:12. Schloss Dagstuhl, May 2017.