# Piecewise Relative Observational Purity

## A New Paradigm for Distributed Systems Programming

Seyed Hossein Haeri
Université catholique de Louvain
Belgium
hossein.haeri@ucl.ac.be

Peter Van Roy
Université catholique de Louvain
Belgium
peter.vanroy@ucl.ac.be

## Abstract

We propose a new paradigm for distributed systems programming: Piecewise Relative Observational Purity (PROP). In this paper, we present the idea behind PROP using a schematic syntax we developed earlier. We also talk about real-world examples in which PROP programming of distributed systems is beneficial.

## 1 Introduction

A distributed system (DS) is a collection of independent nodes that perform computations as if they were performed by a single coherent system. That coherence and singleness is achieved by the nodes collaborating on the overarching computation. Nodes can manage such collaborations by passing messages to one another via network.

In our recent work, the $\lambda$(port) family [4], distributed systems programming (DSP) is by provision of *configuration*s composed by " $\|$ ". A configuration is composed of computations running concurrently on the nodes of a DS.

Experience in our two recent EU projects suggest that, like Java code [2], DS' programs exhibit a substantial degree of mostly functional behaviour. Consider the evaluation of an expression $e_1$ as a part of a computation running in a DS: $e_1 \rightarrow_\lambda \cdots \rightarrow_\lambda e_k \rightsquigarrow e_{k+1} \rightarrow_\lambda \cdots \rightarrow_\lambda e_l \rightsquigarrow e_{l+1} \rightarrow_\lambda \ldots$, where "$\rightarrow_\lambda$" is a reduction step in the pure susbset of $\lambda$(fut) [10] or any other pure dialect of $\lambda$-Calculus.

Two points are noteworthy about $e_1$'s evaluation. Let "$\twoheadrightarrow_\lambda$" be the reflexive and transitive closure of "$\rightarrow_\lambda$". (1) $e_1 \twoheadrightarrow_\lambda e_k$ but $e_1 \not\twoheadrightarrow_\lambda e_{k+1}$. That is because $e_k \rightsquigarrow e_{k+1}$ is not a step in any pure dialect of $\lambda$-Calculus that models functional programming. Colloquially, we call that step a side-effect and the computation effectful. (2) $e_1$'s evaluation is mostly functional: Reductions are mostly "$\rightarrow_\lambda$" with only few "$\rightsquigarrow$".

Our understanding is that code written for DSs executes like $e_1$'s evaluation with its ratio of "$\rightarrow_\lambda$" to "$\rightsquigarrow$". On the contrary, DSP is practised as if every reduction is a "$\rightsquigarrow$". That is, DS 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. Whereas, in reality, side-effects are rare in DSP. User interaction, handling failure, timeout, or actions that depend on timing form a small fraction of the DSs code. That suggests a need for a paradigm shift to PROP.

An important service of $\lambda$(port) for PROP is its linguistic support for **pure block**s: constructs to discipline DSP 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 the programmer's promise of not having any side effects (in the sense of algebraic effects for the nominated nodes. For example, consider $\text{pure}^{\overline{a}}\ \{e_1; e_2; \ldots; \text{pure}^{\overline{b}}\ \{e'_1; e'_2; \ldots e'_k\}; \ldots; e_n\}$, where $\overline{a}$ and $\overline{b}$ are disjoint lists of nodes. The expressions $e_1, e_2, \ldots, e_n$ promise not to have side effects for the nodes $\overline{a}$. 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 effectful for $\overline{a}$. Currently, $\lambda$(port) checks the promise of a pure block at the runtime.

PROP stands on three pillars:

1. Piecewise Purity Specification: Purity specification – rather than just puriy annotation [12] – in PROP is in the form of blocks, so it can be piecewise. (See the previous paragraph.) That is based on the observation that different pieces of a DS's program might have different purity properties.
2. Relativity of Side-Effects: PROP programming acknowledges that different nodes in a DS might observe different side-effects from a computation. It also acknowledges that, whilst computations might not be *universally* pure, they might be pure to certain nodes. We write $e_1 \sim^a e_2$ (read $e_1$ and $e_2$ are $a$-bisimilar) when $e_1$ and $e_2$ have similar side-effects for a node $a$. Note that $e_1 \sim^a e_2$ is deliberately not comparing the effectfulness of $e_1$ and $e_2$ for other nodes than $a$.
3. Observational Purity: PROP has its own take of observational purity [9]. 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 $(\text{port}, \text{heap}) \Vdash e_1 \sim^a e_2$.

Our conjecture is that code for most DSs can be designed so it is mostly pure with very few effectful parts. That would be sheer benefit. The pure parts win the code typical purity gifts such as idempotence, referential transparency, and

*ProWeb20, 23–26 Mar, Porto, Portugal*
2020.

equational reasoning. Hence, PROP urges knowledgeable decisions about the purity of different pieces of a DS's program. Schematically, that is code like $(\text{pure}^{\overline{a}}\ \{e_1; e_2\}; e_3; e_4; e_5)^{a_1} \parallel (\text{pure}^{\overline{b}}\ \{e_6\}; e_7; \text{pure}^{\overline{c}}\ \{e_8; \text{pure}^{\overline{d}}\ \{e_9; e_{10}\}\}; e_{11})^{a_2} \parallel \ldots$, where $e^a$ denotes expression $e$ running on the node $a$.

Consider $(\text{pure}^{\overline{a}}\ \{e_1; e_2\}; e_3; e_4; e_5)^{a_1}$ above. The scheduler does know when the execution of that computation is in the $\text{pure}^{\overline{a}}\ \{e_1; e_2\}$ part. That knowledge can be helpful. For the entire time that every computation on a node $a$ is pure for a node $b$, there is no risk of race conditions for the computations running on $b$ that mutate the data it exclusively shares with $a$. For example, those computations require no mutex locks for performing their desirable mutations. Hence, increased efficiency and correctness. More generally, such information is valuable in very much the same way piecewise determinism [7, 13] helps log-based rollback recovery in a DS [6, 14]: The execution is modelled as a series of nondeterministic events interleaved by high volume of determinism instructions; piecewise determinism legislates **full** logging of the nondeterministic events, exclusively.

# 2 PROP Benefits

## 2.1 Flaky Tests

When tests nondeterministically produce different results for the same input, they are said to be *flaky*. A very recent study [3] reports that the 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 [8] 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 [5] 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 respond to that call.

## 2.2 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 DSs where messages are likely to be delayed due to network reasons. Evaluation of a condition can very well depend on the arrival of messages.

Tapus and Hicky [15] give an operational semantics for speculative execution in distributed systems. In their work, a great proportion is dedicated to bookkeeping shared objects. Purity makes shared objects conceptually immutable, and, understanding speculative execution easier for DSs. That conjecture is backed up by an earlier empirical study on speculative execution of distributed file systems [11, §5.3]: correctness of speculation is reported for functions $f$ running on a node $a$ when $f$ only modifies $a$'s local state. Namely, state $\Vdash f^a \sim^a \text{pure}^{\mathfrak{N}\setminus a}\ \{f^a\}$, where $\mathfrak{N}$ is the set of all nodes.

## 2.3 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 [1] 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). For a given node $a$, that is like replacing $f^a$ by $\text{pure}^{\mathfrak{N}\setminus a}\ \{f\}$, 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.

## 2.4 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 is *stop-the-world*. That technique is, however, suboptimal because it increases downtime. The reason why stop-the-world is nevertheless used is a conflict between mutability and access control: Over the process of garbage-collection, a variable mutation may (re)create a reference to a location previously marked for deallocation. One may argue that a location a variable has access to which should not be marked. But, that argument implies a drop in the number of marked locations that cannot be tolerated. So, instead of denying access eternally, stop-the-world chooses to deny it only during garbage collection. Observational purity goes beyond that and legislates stopping only the part of the world that does observe local mutations. For a node $a$, enclosing $\text{pure}^a\ \{.\}$ blocks help automatic detection of the nodes that need not to be stopped over garbage collection at $a$. If all the computations on a node $b$ are of the form $(\text{pure}^{\overline{a}}\ \{e\})^b$, where $a \in \{\overline{a}\}$, then it is correct for $b$ not to be stopped over $a$'s garbage collection.

## 2.5 Model Checking

Partial order reduction is a classical technique used in model checking for reducing the search space. 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 [16, §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, $\forall a \in \{\overline{a}\}$. 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.

## References


[1] J. Bell and G. E. Kaiser. 2014. Unit Test Virtualization with VMVM. In $36^{th}$ *ICSE*, P. Jalote, L. C. Briand, and A. van der Hoek (Eds.). ACM, 550–561. https://doi.org/10.1145/2568225.2568248

[2] W. C. Benton and C. N. Fischer. 2009. Mostly-Functional Behavior in Java Programs. In $10^{th}$ *VMCAI (LNCS)*, N. D. Jones and M. Müller-Olm (Eds.), Vol. 5403. Springer, 29–43. https://doi.org/10.1007/978-3-540-93900-9_7

[3] M. Eck, F. Palomba, M. Castelluccio, and A. Bacchelli. 2019. Understanding Flaky Tests: The Developer's Perspective. In $27^{th}$ *SIGSOFT FSE* / $18^{th}$ *ESEC*, M. Dumas, D. Pfahl, S. Apel, and A. Russo (Eds.). ACM, 830–840. https://doi.org/10.1145/3338906.3338945

[4] S. H. Haeri and P. Van Roy. 2020. A Family of $\lambda$-Calculi with Ports. In $21^{st}$ *TFP*. LNCS. Accepted.

[5] W. Lam, P. Godefroid, S. Nath, A. Santhiar, and S. Thummalapenta. 2019. Root Causing Flaky Tests in a Large-Scale Industrial Setting. In $28^{th}$ *ISSTA*, D. Zhang and A. Møller (Eds.). ACM, 101–111. https://doi.org/10.1145/3293882.3330570

[6] H. M. Lee, D.-S. Park, H. C. Yu, and G. Lee. 2011. FRASystem: Fault Tolerant System using Agents in Distributed Computing Systems. *Cluster Comp.* 14, 1 (2011), 15–25. https://doi.org/10.1007/s10586-009-0095-x

[7] B. Liskov. 1982. On Linguistic Support for Distributed Programs. *IEEE Trans. Soft. Eng.* 8, 3 (1982), 203–210. https://doi.org/10.1109/TSE.1982.235250

[8] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov. 2014. An Empirical Analysis of Flaky Tests. In 22*nd FSE*, S.-C. Cheung, A. Orso, and M.-A. D. Storey (Eds.). ACM, 643–653. https://doi.org/10.1145/2635868.2635920

[9] D. A. Naumann. 2007. Observational Purity & Encapsulation. *TCS* 376, 3 (2007), 205–224. https://doi.org/10.1016/j.tcs.2007.02.004

[10] J. Niehren, J. Schwinghammer, and G. Smolka. 2006. A Concurrent Lambda Calculus with Futures. *TCS* 364, 3 (2006), 338–356.

[11] E. B. Nightingale, P. M. Chen, and J. Flinn. 2006. Speculative Execution in a Distributed File System. *ACM Trans. Comp. Sys.* 24, 4 (2006), 361–392. https://doi.org/10.1145/1189256.1189258

[12] D. J. Pearce. 2011. JPure: A Modular Purity System for Java. In $20^{th}$ *CC (LNCS)*, J. Knoop (Ed.), Vol. 6601. Springer, 104–123. https://doi.org/10.1007/978-3-642-19861-8_7

[13] R. E. Strom, D. F. Bacon, and S. Yemini. 1988. Volatile Logging in n-Fault-Tolerant Distributed Systems. In $18^{th}$ *FTCS*. IEEE, 44–49. https://doi.org/10.1109/FTCS.1988.5295

[14] R. E. Strom and S. Yemini. 1985. Optimistic Recovery in Distributed Systems. *TCS* 3, 3 (1985), 204–226. https://doi.org/10.1145/3959.3962

[15] C. Tapus and J. Hickey. 2009. Distributed Speculative Execution for Reliability and Fault Tolerance: An Operational Semantics. *Dist. Comp.* 21, 6 (2009), 433–455. https://doi.org/10.1007/s00446-008-0073-1

[16] M. Walker. 2016. *Déjà Fu: A Concurrency Testing Library for Haskell.* Technical Report. Dept. Comp. Sci., U. York.