Marking Piecewise Observable Purity

Seyed Hossein Haeri* and Peter Van Roy

INGI, Université catholique de Louvain, Belgium; {hossein.haeri,peter.vanroy}@uclouvain.be

Abstract

We present a calculus for distributed programming with explicit node annotation, built-in ports, and pure blocks. To the best of our knowledge, this is the first calculus for distributed programming, in which the effects of certain code blocks can go unobserved, making those blocks pure. This is to promote a new paradigm for distributed programming, in which the programmer strives to move as much code as possible into pure blocks. The purity of such blocks wins familiar benefits of pure functional programming.

1 Introduction

Reasoning about distributed programs is difficult. That is a well-known fact. That difficulty is partly because of the relatively little programming languages (PL) research on the topic. But, it also is partly due to the unnecessary complexity conveyed by the traditional conception of distributed systems.

Reasoning about sequential programs written for a single machine, in contrast, is the bread and butter of the PL research. Many models of functional programming, for example, have received considerable and successful attention from the PL community. The mathematically rich nature of those models is the main motivation.

In particular, pure functional programming, i.e., programming with no side-effects, has been a major source of attraction to the PL community. In that setting, many interesting properties are exhibited elegantly, e.g., equational reasoning, referential transparency, and idempotence.

The traditional conception about distributed programming, however, lacks those nice properties. In that conception, side-effects are inherent in distributed programming. The argument is: No sensible distributed program can run without communication between nodes; and, such communications **always** alter the state of the communication medium; hence, the effectfulness.

The *distributed* λ-calculus [6] below refutes that argument. Let $a, b, c, \ldots, a', b', c', \ldots$ range over a set of nodes ($\mathfrak{N}$). 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.

Definition 1. *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* $. \xrightarrow{d} .$ *on* λ_d *follow, where common capture-avoiding measure apply:*

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

Take λ_o for the set of ordinary λ-calculus terms and $\xrightarrow{o}$ for its reductions. Distributed λ-calculus is equivalent to the ordinary λ-calculus, and, is pure.

Theorem 2. *For every node a and b, the reduction $t \xrightarrow{o}\!\!\!\!\rightarrow t'$ implies $[\![t]\!]^{+a} \xrightarrow{d}\!\!\!\!\rightarrow [\![t']\!]^{+a}$, and, the reduction $t^a \xrightarrow{d}\!\!\!\!\rightarrow t'^b$ implies $[\![t^a]\!]^{-} \xrightarrow{o}\!\!\!\!\rightarrow [\![t'^b]\!]^{-}$.*

*This work is funded by the LightKone European H2020 project under grant agreement no. 732505.

The distributed λ-calculus is not general enough to model all distributed programming. In particular, it misses interaction with the outside world, e.g., human-beings. Such interactions are the **only** source of effectfulness. Other node communications can be programmed purely.

But, what if the effectfulness of communication can still go unobserved and be *viewed* as pure functional? That is certainly not possible universally, i.e., to every observer. Some observers might, nonetheless, not be concerned about the given communication and its side-effects.

Noticing that possible lack of concern drives this paper. We formalise what it means for a node not to be concerned about the side-effects of a given distributed program. Armed with that formalism, we present a new programming paradigm, in which the programmer is urged to mark the proportions of their code that are pure for a given node. The larger such proportions are, the more it is possible for the given node to benefit from the purity properties of the code. Typical benefits include the ones enumerated earlier for pure functional programming. Of course, it remains for the language to verify the correctness of purity markings.

Mind the subtle difference with 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 (even though their type indicates the type of side-effects they have). In contrast, we acknowledge side-effects, yet, help the right nodes benefit from the purity of the correctly marked code proportions.

We are not the first to notice that effectfulness depends on the observer. Even though effectful, the Accelerate library [3] of Haskell has a purely functional interface, and, refrains from monadic treatments. That is becaus the only side effect of evaluating an Accelerate expression with 'run' is compilation and execution of the program. However, that side effect is not observable to Accelerate's host PL (i.e., Haskell).

In this paper, we present $\lambda(\mathsf{port})_\circ$: a variation of $\lambda(\mathsf{fut})$ [4] with built-in ports and pure blocks (i.e., blocks of code marked for purity). Ports are the only means for side-effects in $\lambda(\mathsf{port})_\circ$. In essence, the $\lambda(\mathsf{port})_\circ$ ports are asynchronous communication media. They are designed for interaction with our otherwise pure calculus. Sending to a port will add a message to the port's stream, which can be read by a pure expression (c.f. *srv* at Example 7).

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 [5, 1] aside, most PL results are established on top of the former. In this work we are particularly interested in the pure functional programming results. Our aim is to reuse those results. So, like more conventional λ-Calculi with futures [2, 4], we build on top of λ-Calculus.

We present the syntax and semantics of $\lambda(\mathsf{port})_\circ$ (Definitions 3 and 4). We formally define a notion of observational equivalence for a given node (Definition 5). Most importantly, we give an example where expanding a pure block goes unobserved by a node (Theorem 6). Finally, we showcase $\lambda(\mathsf{port})_\circ$ for a client-server application with only a couple of clients (Example 7).

2 Syntax and Semantics

The $\lambda(\mathsf{port})_\circ$ syntax is tailored for our minimal working example (Example 7), in particular.

Definition 3. *The* $\lambda(\mathsf{port})_\circ$ *expressions (E) and configurations (G) are defined below, where* this *font is for keywords:*

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

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, function applications, named functions ($f(x) = e$), sequential composition, and, *cons* expressions. Our pattern matching is less routine by only allowing a single match and only against cons expressions. 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. Then, there are sending to ports, and pure blocks. Configurations are port declarations, (node-annotated) expressions, and concurrent compositions. One annotates an expression e with a node a as e^a. Such an annotation only applies to the outermost layer. 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 $\lambda(\mathsf{port})_\circ$, the number of concurrent compositions is known statically.

We present a small-step operational semantics for the syntax given in § 2. Judgements $\jmath, \jmath', \cdots \in J$ of the operational semantics take the form $\tau : g \to \tau' : g'$. Each 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 known to ρ but unbound in the current state of the program. The clause $\jmath = \tau : g \to \tau' : g'$ sets $\jmath$ as an alias for $\tau : g \to \tau' : g'$. Write $nc_j(\jmath)$ for the set of nodes (environment or store) bindings of which were changed over $\jmath$. Write $g \to g'$ as a shorthand for $\tau : g \to \tau : g'$. Write $\to^*$ for the transitive and reflexive closure of $\to$. Finally, fix a set of values ranged over by $v_1, v_2, \ldots$, including unit and all cs.

Definition 4. *Rules of the $\lambda(\mathsf{port})_\circ$ operational semantics follow.*

$$(\lambda x.e_1)\ e_2 \to e_1[e_2/x] \quad \text{(App-E)} \qquad f\ e \parallel f(x) = e' \to e'[e/x] \parallel f(x) = e' \quad \text{(App-F)}$$

$$v; e \to e \quad \text{(Seq-1)} \qquad \mathsf{match}\ (e :: s)\ \mathsf{for}\ \{x :: s' \Rightarrow e'\} \to e'[e/x, s/s'] \quad \text{(Mat-1)}$$

$$\mathsf{match}\ \bot\ \mathsf{for}\ \{x :: s \Rightarrow e\} \to \mathsf{match}\ \bot\ \mathsf{for}\ \{x :: s \Rightarrow e\} \quad \text{(Mat-2)}$$

$$\frac{\tau : e_1 \to \tau' : e_1'}{\tau : e_1; e_2 \to \tau' : e_1'; e_2}\ \text{(Seq-2)} \qquad \frac{e \to e'}{e^a \to e'^a}\ \text{(Annot)} \qquad \frac{\tau : g_1 \to \tau' : g_1'}{\tau : g_1 \parallel g_2 \to \tau' : g_1' \parallel g_2}\ \text{(Cncr)}$$

$$\frac{\jmath = \tau : e \to \tau' : e' \quad \overline{a} \notin nc_j(\jmath)}{\tau : \mathsf{pure}^{\overline{a}}\ \{e\} \to \tau' : \mathsf{pure}^{\overline{a}}\ \{e'\}}\ \text{(Pure)}$$

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

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

The rules in the first four 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. 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.

3 Results

Write $\Delta^*_{\mathsf{port}}(\tau, g)$ for the nodes for which new ports were declared or the existing ports of which were sent to over $\tau : g \to^* \_ : \_$, where "_" is our wildcard notation. Intuitively, according to

Definition 5 below, a node observes two configurations equivalently when, for a single reduction step of one, the other can take one step (or more) with α-equivalent impacts on the parts of the environment and store that pertain to the node.

Definition 5. *Call g_1 and g_2 observationally equivalent **for a node** a (write $g_1 \sim_a g_2$) when*

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

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

Example 7. *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. Let also $\mathsf{stream}(p^a)$ denote the stream of p^a. 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.*

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

□

References

[1] S. Conchon and F. Le Fessant. Jocaml: Mobile agents for Objective-Caml. In *Proc. of 1st Int. Symp. on Agent Systems and Applications and 3rd Int. Symp. on Mobile Agents, ASA/MA '99*, pages 22–29. IEEE, 1999.

[2] C. Flanagan and M. Felleisen. The semantics of future and an application. *J. Funct. Program.*, 9(1):1–31, 1999.

[3] T. L. McDonell. *Optimising Purely Functional GPU Programs*. PhD thesis, Univ. of New South Wales, Sydney, 2015.

[4] J. Niehren, J. Schwinghammer, and G. Smolka. A concurrent lambda calculus with futures. *Theor. Comput. Sci.*, 364(3):338–356, 2006.

[5] 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.

[6] P. Van Roy. Why time is evil in distributed systems and what to do about it. CodeBEAM keynote talk, 2019. Available at https://www.youtube.com/watch?v=NBJ5SiwCNmU