PURELY FUNCTIONAL DISTRIBUTED SYSTEMS PROGRAMMING

SEYED H. HAERI PETER VAN ROY

Abstract. Code written for distributed systems requires to handle communication between nodes in the system. Handling that communication involves mutating the state of its medium – or, at least that is the common impression. Hence, distributed systems programming is construed as effectful (aka impure) ipso facto. This work refutes that construal. We show that the above effectfulness (aka impurity) is accidental complexity.

Our $\lambda(\mathsf{refut})$ is a pure functional model for distributed systems programming. $\lambda(\mathsf{refut})$ ships *remote futures*: a built-in facility for node communication. To use remote futures, the $\lambda(\mathsf{refut})$ programmer does not mutate variables. Hence, purity of the programming. The payback is that $\lambda(\mathsf{refut})$'s term rewriting is impure. But, so is also the semantics of the famous purely functional languages.

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 (DSs).

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. In short, equational reasoning is the process of interpreting code by substituting expressions for their equivalents; referential transparency is correctness of replacing an expression by its value (or any other expression having the same value); and, idempotence is when a function is invariant with respect to multiple calls. Every single one of those nice properties can boost code efficiency considerably.

The traditional conception about DSs 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.

In earlier work [3, 4], we raised the point that effectfulness can be relative. What a node might find effectful is not necessarily effectful to other nodes too. In other words, side-effects might not be observable to every node in the DS.

This work takes that relativism and observability one step further. Side-effects observable to the PL's compiler (or, interpreter, evaluator, etc.) need not to be observable to the nodes in the DS. This idea, on its own, is not our invention. However, to the best of our knowledge, we are the first to notice that one can leverage that idea for a pure functional model for DSs programming.

In a simple two level world, a PL can have an object level as well as a meta-level. The object level is the one at which the programmer codes. The meta-level is the one that processes the object level. Let us set those PLs that supply metaprogramming aside for a moment. A pure

PL does not offer means for state manipulation at the object level. Nevertheless, it is perfectly fine for a pure code to incur change of state at the meta-level, and, hence, be effectful to that level.

Consider the common let-expressions of functional PLs (FPLs). In a pure PL, the programmer cannot rebind a variable. Memory allocation is, however, inevitable for the variable. In addition, reductions might update the runtime environment's bookkeeping of the variable. The Haskell let-bindings, in particular, are that way [6, 1]. Despite that, Haskell is today's most famous pure FPL because that effectfulness to memory is only at the meta-level.

In $\lambda(\mathsf{refut})$ too, for node communication, the programmer does not manipulate a communication medium at the object level. Instead, communication is by a node launching a remote task at the other node, whilst having a single-assignment handle to it. That is similar to dataflow futures of Niehren et al. [7], but, $\lambda(\mathsf{refut})$ provides a let notation for that. Thus, the set of remotely bound variables varies during runtime. Furthermore, a remote task may reduce over time too – as if the variable bound to it is being reassigned. Nonetheless, $\lambda(\mathsf{refut})$ programming remains pure. Because, at $\lambda(\mathsf{refut})$'s object level, the set of remotely bound variables is not observable. That set is only for $\lambda(\mathsf{refut})$'s meta-level bookkeeping. In other words, the impurity of message passing does not disappear in $\lambda(\mathsf{refut})$; it only is not observable at the object level.

$\lambda(\mathsf{refut})$ is an important improvement to the Distributed λ-Calculus [2] because, contrary to $\lambda(\mathsf{refut})$, the latter system cannot handle user input. In addition, $\lambda(\mathsf{refut})$ is an improvement over the classical model of Kahn networks [5] because it can handle non-deterministic ordering of inputs from other nodes in the DS. (Cf. Examples 3.1 and 3.3, respectively.)

An important shortcoming of $\lambda(\mathsf{refut})$ is that it cannot model distributed programs such as client-server, in which the **order** of messages is significant. The server is expected to be FIFO: earlier requests should be handled before the later ones. Of course, clients might have different priorities. Yet, the FIFO property is expected of the server for clients of the same priority.

Here is how this paper is organised: In § 2, we present the syntax and semantics of $\lambda(\mathsf{refut})$. In § 3, we provide three case-studies for $\lambda(\mathsf{refut})$: Example 3.1, 3.3, and 3.6 describe a master-slave architecture, a sensor network, and a replicated key-value store, respectively.

The study of how to extend $\lambda(\mathsf{refut})$ to handle significance in the order of messages is future work. Ports are our first candidate for that future work. Ports are a standard medium for message passing in DSs. Each port has an associated queue to guarantee the FIFO property required, say, for a client-server scenario.

2. Formalism

Definition 2.1. *The* $\lambda(\mathsf{refut})$ *syntax follows.*

$$\begin{array}{llll@{\qquad}llll}
p & ::= & g \text{ where } d & \text{programs} & g & ::= & t \mid x \mapsto t \mid g \parallel g & \text{configurations} \\
e & ::= & x \mid c \mid \lambda x.e \mid e_1\ e_2 \mid f\ e & & t & ::= & e^a & \text{tasks} \\
 & \mid & \text{let } \{b\} \text{ in } e \mid e; e & \text{expressions} & b & ::= & x = e \mid x = t \mid b, b & \text{bindings} \\
d & ::= & f(\overline{x}) = \{e\} \mid d, d & \text{definitions} & & & & \square
\end{array}$$

A program p in $\lambda(\mathsf{refut})$ is a configuration g of tasks, given some definitions d. A configuration is a set of tasks (with or without a handle to) that run concurrently, denoted by "$\parallel$" in the syntax. A task e^a is an expression e to run on a node a. Remote futures have handles to a task as in $x \mapsto e^a$. What is new in a remote future $x \mapsto e^a$ in comparison with dataflow futures of Niehren et al [7] is e running remotely on the node a.

The syntax for expressions is routine: variables (x), constants (c), λ-abstractions ($\lambda x.e$), function applications ($e_1\ e_2$), application of named functions on expressions ($f\ e$), let-expressions (let $\{b\}$ in e), and expression sequencing ($e_1; e_2$). Bindings of a let-expression can be of two kind: local bindings ($x = e$) and remote bindings ($x = e^a$). The latter kind gives the task of evaluating e to the node a and keeps a handle x to it so e's value can be used via x when ready, if at all. When a in $x = e^a$ is the current node, the binding is local and the node annotation will be disregarded. The notation $f(\overline{x}) = \{e\}$ is list comprehension for $f(x_1, \ldots, x_n) = \{e\}$, for a given n. We may drop the curly braces around e in $f(x_1, \ldots, x_n) = \{e\}$ when no confusion. So may we also do for the separator commas between definitions and between bindings of a let-expression.

We assume a set of values ranged over by $v, v', \ldots, v_1, v_2, \ldots$. Subscripts and priming do not change the syntactic category. For example, $x_1, x_2, \ldots, x', x'', \ldots$ are all variables.

We take the order of functions introduced after the where clause to be irrelevant. Such we also take the order of bindings in a let expression and that of the constituents of a configuration. Moreover, we take let $\{b_1, b_2\}$ in e to be a shorthand for let $\{b_1\}$ in (let $\{b_2\}$ in e). The above dismissals of ordering is an informal account of our structural congruence below.

$$g \ \mathsf{where}\ \ d_1, d_2 \equiv g \ \mathsf{where}\ d_2, d_1 \qquad \mathsf{let}\ \{b_1, b_2\}\ \mathsf{in}\ e \equiv \mathsf{let}\ \{b_2, b_1\}\ \mathsf{in}\ e$$
$$g_1 \parallel g_2 \equiv g_2 \parallel g_1 \qquad \mathsf{let}\ \{b_1, b_2\}\ \mathsf{in}\ e \equiv \mathsf{let}\ \{b_1\}\ \mathsf{in}\ \mathsf{let}\ \{b_2\}\ \mathsf{in}\ e$$

Definition 2.2. *Evaluation contexts C of $\lambda(\mathsf{refut})$ are of three sorts E, G, P (corresponding to e, g, and p of the $\lambda(\mathsf{refut})$ syntax, respectively):*

$$\begin{aligned}
C &::= (E, \delta) \mid (G, \delta) \mid (P, \delta) \\
E &::= \square \mid E\ e \mid (\lambda x.e)\ E \mid f\ E \mid E; e \mid \mathsf{let}\ \{b\}\ \mathsf{in}\ E \mid \mathsf{let}\ \{x = E\}\ \mathsf{in}\ e \\
G &::= E^a \mid x \Mapsto E^a \mid G \parallel g \\
P &::= G\ \ \mathsf{where}\ \ d
\end{aligned}$$

where δ is a mapping from variables to expressions. □

Given that $\lambda(\mathsf{refut})$ programs contain `where` clauses, $\lambda(\mathsf{refut})$ contexts carry δ around to recall the functions defined in the `where` clause. The context E is standard call-by-value with sequencing and let-expressions. Notice that E has no representative for remote futures. That is because a remote future is to only reduce at its respective remote node.

We are now prepared for the $\lambda(\mathsf{refut})$ semantics.

Definition 2.3. *The reduction semantics of $\lambda(\mathsf{refut})$ is of the form $C[e] \to C[e']$, where we will leave the δ out when invariant upon the reduction. The reduction rules follow:*

$$(P, \delta)[g\ \ \mathsf{where}\ \ d_1, \ldots, d_n] \to (G, \delta[f_i \mapsto \lambda\overline{x}_i.e_i]_{i=1}^n)[g] \qquad \text{where } d_i \text{ is } f_i(\overline{x}_i) = e_i \text{ and } f_i \notin \mathsf{domain}(\delta) \qquad \text{(PRG)}$$

$$E[(\lambda x.e)\ v] \to E[e[v/x]] \qquad \text{(AP-E)}$$

$$E[\mathsf{let}\ \{x = v\}\ \mathsf{in}\ e] \to E[e[v/x]] \qquad \text{(LT-E)}$$

$$E[v; e] \to E[e] \qquad \text{(SQ)}$$

$$E[f] \to E[\lambda\overline{x}.e] \qquad \text{where } \delta(f) = \lambda\overline{x}.e \qquad \text{(LD-F)}$$

$$G[\mathsf{let}\ \{x = t\}\ \mathsf{in}\ e] \to G[e[y_a/x]] \parallel y_a \Mapsto t \qquad \text{where } y \text{ fresh and } \mathsf{node} = a \qquad \text{(LT-T)}$$

$$G[x] \parallel x \Mapsto v^a \to G[v] \parallel x \Mapsto v^a \qquad \text{(DF)}$$

Technically, the following rule is not required. Yet, we adopt it as a shorthand for an application of (LD-F) *followed by a long-enough sequence of* (AP-E)*s. Suppose that $\delta(f) = \lambda\overline{x}.e$. Then,*

$$E[f\ \overline{v}] \to E[e[\overline{v}/\overline{x}]]. \qquad \text{(AP-F)}\ \square$$

(PRG) simply dumps the functions defined at the `where` clause into δ. The function application rule (AP-E) is routine call-by-value. The named function application rule (AP-F) is not much more complicated. The main difference lies in the comprehension notation: $f\ \overline{v}$ abbreviates $f\ v_1\ \ldots\ v_n$, for some given n; and, $e'[\overline{v}/\overline{x}]$ abbreviates $e'[v_1/x_1, \ldots, v_n/x_n]$, again, for some given n. According to (LT-E), a let-bound variable can only be substituted when it is bound to a value (possibly after reduction). (Sq) is routine for sequencing. (Ld-F) uses δ to substitute the body of a named function for its name.

(Lt-T) is less trivial. λ(refut) maintains a metavariable node to represent the node in which the reduction is taking place. The notation node $= a$ in (Lt-T) is for "let us assume that the node at which this let-expression is running is a." In such a case, we launch a new concurrent task t at the right node and give it a fresh handle. (That is at a node b when $t = e'^b$.) The subscript a in y_a is to prevent variable shadowing whilst evaluating the same let-expression at different nodes. Furthermore, we choose y to be fresh to prevent shadowing upon subsequent evaluations of the let-expression. Example 3.3 makes use of both above freshness tricks. Of course, due to the refreshing of x to y_a, occurrences of the latter need to be renamed in e accordingly.

(Df) describes the situation when a dataflow variable x has a handle to a value (possibly the result of a formerly more complicated expression). In such a situation, according to (Df), the value x is bound to can be substituted for its occurrences.

Here is an important note in relation to (Df) and (Lt-T). One might have come to ask whether λ(refut) advises for too much concurrency. After all, according to (Lt-T), every remote let-binding spawns a new concurrent evaluation. We find that alright because if one modifies (Df) to

$$G[x] \parallel x \Mapsto v^a \to G[v] \parallel x \Mapsto v^a \qquad \text{when } \mathsf{node} = b \text{ and } a \neq b$$

one can help the scheduler with putting threads of b that have references to x to sleep until the above rule is applied to them. Furthermore, the scheduler can always perform garbage collection on a handled task $x \Mapsto v^a$ when there no longer is a reference to x in the configuration. We drop those in Definition 2.3 for simplicity.

3. Using λ(refut)

Example 3.1. Consider a master-slave scenario with a master node m and a sufficiently large set of slave nodes $S = \{s_1, s_2, \dots\}$. Let us assume the availability of an input function that inputs a single item from the user. Let us also assume the any metafunction defined below

$$\frac{x_i \Mapsto v^{a_j} \text{ for } i, j \in \{1, \dots, n\}}{\mathsf{any}(x_1, x_2, \dots, x_n) \to v}.$$

The following λ(refut) code gives the task of computing a function f to as many slaves as requested from m. The code is done as soon as any of those slaves is done with the computation of f, if ever.

```
 (let {k = input} in distribute f k)^m
where
  distribute(g, n) = let
    {x_i = (g ())^{s_i}}_{i=1}^n
  in
    any(x_1, ..., x_n)
  f() = {...}
```

Let us call the above λ(refut) program p. Let us assume that the user enters k₀ for k. Then, dropping the where clauses in the presentation

p $\to^*$ (Prg),(Ap-E)

(let { k = k₀ } in distribute f k)^m $\to$ (Lt-E)

(distribute f k₀)^m $\to$ (Ap-F)

(let $\{x_i$ = (f ())$^{s_i}\}_{i=1}^{k_0}$ in any(x₁, ..., x_{k_0}))^m $\to^*$ (Lt-T)

(any(x₁, ..., x_{k_0}))^m|| x₁ |=> (f ())^{s_1}|| ... || x_{k_0} |=> (f ())$^{s_{k_0}}$ □

Example 3.1 demonstrates an important step forward after the Distributed λ-Calculus [2]: λ(refut) can also handle user input.

Remark 3.2. One may argue that `input` is not pure and so is not `p` in Example 3.1. We would like to draw the reader's attention to the common monadic treatments of I/O, for example, in order to give that a pure interface on a single node. This paper takes such (or similar) treatments for granted to focus on purity of distribution. □

Example 3.3. Consider a control unit c receiving updates from sensors $s_1, \dots, s_n$. Initially, the sensors are configured to take samples every 1 second. Based on their updates, the duration between two sampling may change, as instructed by the control unit in response to the update. The order in which the sensors send their update is unknown. So is also the order in which they are handled. The $\lambda(\textsf{refut})$ code below describes that scenario.

```
1  (sensor 1)^{s1} || ... || (sensor 1)^{sn}
2  where
3    sensor(k) = let
4      s = sample k
5      u = (update s)^c
6    in
7      sensor u
8    update(s) = {...}, sample(k) = {...}
```

To demonstrate a sample run, we take $n = 2$ and call the above code `p`. Again, in the demonstration, we will drop the `where` clause.

$$\mathtt{p} \longrightarrow^* \qquad \text{(Prg), (Ap-F)}$$

$$\mathtt{(\textbf{let}\ s = sample\ 1,\ u = (update\ s)^{c}\ \textbf{in}\ sensor\ u)^{s_1} ||\ (sensor\ 1)^{s_2}} \longrightarrow^* \qquad \text{(Ap-F), (Lt-E)}$$

$$\mathtt{(\textbf{let}\ u = (update\ k_1^1)^{c}\ \textbf{in}\ sensor\ u)^{s_1} ||\ (sensor\ 1)^{s_2}} \longrightarrow \qquad \text{(Lt-T)}$$

$$\mathtt{(sensor\ u_1^1)^{s_1} ||\ (sensor\ 1)^{s_2} ||\ u_1^1\ |{=}{>}\ (update\ k_1^1)^{c}} \longrightarrow \qquad \text{(Ap-F)}$$

$$\mathtt{(sensor\ u_1^1)^{s_1} ||\ (sensor\ 1)^{s_2} ||\ u_1^1\ |{=}{>}\ (v_1^1)^{c}} \longrightarrow \qquad \text{(Df)}$$

$$\mathtt{(sensor\ v_1^1)^{s_1} ||\ (sensor\ 1)^{s_2} ||\ u_1^1\ |{=}{>}\ (v_1^1)^{c}} \longrightarrow \qquad \text{(Ap-F)}$$

$$\mathtt{(sensor\ v_1^1)^{s_1} ||\ (\textbf{let}\ s = sample\ 1,\ u = (update\ s)^{c}\ \textbf{in}\ sensor\ u)^{s_2}}$$

$$\mathtt{||\ u_1^1\ |{=}{>}\ (v_1^1)^{c}} \longrightarrow^* \qquad \text{(Ap-F), (Lt-E), (Lt-T), (Ap-F), (Df)}$$

$$\mathtt{(sensor\ v_1^1)^{s_1} ||\ (sensor\ v_1^2)^{s_2} ||\ u_1^1\ |{=}{>}\ (v_1^1)^{c} ||\ u_1^2\ |{=}{>}\ (v_1^2)^{c}} \longrightarrow \dots$$

Of course, the scheduler might choose a very different interleaving. □

Remark 3.4. In Example 3.3, according to the wiring of the (Lt-T) rule, each call to `sensor` will place a new task in the configuration. The handle to that task is different from those of the previous calls and those of the other nodes. (In the sample run, we write $\mathtt{u}_j^i$ for the j^{th} handle of $\mathtt{s}_i$.) Note also that the application `sensor u` in line 7 needs not to be $\mathtt{(sensor\ u)}^{s_i}$, where `node` = $\mathtt{s}_i$. Thanks to G in (Lt-T), the node information is invariant upon the reduction. (In fact, according to Definition 2.1, replacing `sensor u` by $\mathtt{(sensor\ u)}^{s_i}$ would be ungrammatical.) □

Remark 3.5. Notice how, in Example 3.3, each message sent from a sensor to the control unit places a new remote task on the configuration. That is, however, not really a problem since each such task can be garbage-collected as soon as the respective (Df) is instantiated. It is easy to get the scheduler trigger that garbage collection. □

Example 3.3 is important because it shows how $\lambda(\textsf{refut})$ relaxes a stringent restriction of the Kahn networks [5], whilst still remaining pure. Putting Kahn networks into the context of distribute systems, the restriction would be that each node knows which node will provide its next input. That knowledge is not provided to the node c in Example 3.3, for instance.

Example 3.3 also develops over Example 3.1 by showing how, in $\lambda(\textsf{refut})$ nodes can pass messages back and forth. Each sensor sends its s to the control unit and receives u back as the updated k. All that message passing is purely functional.

Example 3.6. Consider a CRDT[1] key-value store with replicas $r_1, \ldots, r_n$ and a front-end node f. Assume that the value type here is a semilattice. The $\lambda(\mathsf{refut})$ code below provides a possible run of such a system along with the functions required.

```
(write k_1 v_1; write k_1 v_2; write k_2 v_2; write k_1 v_3; read k_1)^f
where
  write(k, v) = let
    i = replica-no ()
    ret = (local-write k v)^{r_i}
  in
    ret

  read(k) = let
    {rv_i = (local-read k)^{r_i}}_{i=1}^{n}
  in
    rv_1 ⊓···⊓ rv_n

  local-write(k, v) = {...}, local-read(k) = {...}, replica-no() = {...}
```

Given that the value type is a semilattice, it suffices for correctness to only write to one of the replicas (*local-write*). The function *write* above does that after choosing the replica, say based on the proximity to the user or traffic load. (We assume that *local-write* returns a Boolean flagging its success.) A crucial factor in the correctness of this implementation is a propagation policy enforcing eventual consistency. It is only under that condition that the ordering of writes does not matter. Reading is different: It requires reading all the replicas first (*local-read*) and then performing a join operation ($\sqcap$) on the results.

For a sample run, we again take $n = 2$ and call the $\lambda(\mathsf{refut})$ code above `p`. Furthermore, let $\mathtt{e}_1 = \mathtt{write\ k}_1\ \mathtt{v}_1$, $\mathtt{e}_2 = \mathtt{write\ k}_1\ \mathtt{v}_2$, $\mathtt{e}_3 = \mathtt{write\ k}_2\ \mathtt{v}_2$, $\mathtt{e}_4 = \mathtt{write\ k}_1\ \mathtt{v}_3$, and $\mathtt{e}_5 = \mathtt{read\ k}_1$. Let also $\mathtt{v}_2 \sqsubset \mathtt{v}_3 \sqsubset \mathtt{v}_1$. Dropping the **where** clauses like the two previous examples, one gets

```
p  →*                                                                        (Prg),(Ap-F)
((let i = replica-no () , ret = (...)^{r_i} in ret);e_2; ...; e_5)^f →              (Ap-F)
((let i = 2, ret = (...)^{r_i} in ret); e_2; ...; e_5)^f →                          (Lt-E)
(let ret = (local-write k_1 v_1)^{r2}); e_2; ...; e_5)^f →                          (Lt-T)
(ret_1^2; e_2; ...; e_5)^f || ret_1^2 |=> ( local-write k_1 v_1 )^{r2} →            (Ap-F)
(ret_1^2; e_2; ...; e_5)^f || ret_1^2 |=> ( false )^{r2} →                          (Df)
( false ; e_2...; e_5)^f || ret_1^2 |=> (false)^{r2} →                              (Sq)
( write k_1 v_2 ; ...; e_5)^f || ret_1^2 |=> (false)^{r2} →*                        (...)
(ret_2^2; ...; e_5)^f || ret_1^2 |=> (false)^{r2}|| ret_2^2 |=> ( true )^{r2} →*    (...)
( write k_2 v_2 ; ...; e_5)^f || ret_1^2 |=> (false)^{r2}|| ret_2^2 |=> (true)^{r2} →*   (...)
( read k_1 )^f || ret_1^2 |=> (false)^{r2}|| ret_2^2 |=> (true)^{r2}
          || ret_1^1 |=> (true)^{r2}|| ret_3^2 |=> (true)^{r2} →                    (Ap-F)
(let {rv_i = (local-read k_1)^{r_i}}_{i=1}^2 in rv_1 ⊓ rv_2)^f || ret_1^2 |=> (false)^{r2}|| ret_2^2 |=> (true)^{r2}
    || ret_1^1 |=> (true)^{r2}|| ret_2^1 |=> (true)^{r2} →*                        (Lt-T)
(rv_1⊓rv_2)^f || rv_1 |=> (local-read k_1)^{r1}|| rv_2 |=> ( local-read k_1 )^{r2}|| ret_1^2 |=> (false)^{r2}
   || ret_2^2 |=> (true)^{r2}|| ret_1^1 |=> (true)^{r2}|| ret_2^1 |=> (true)^{r2} →   (Ap-F)
(rv_1⊓rv_2)^f || rv_1 |=> (local-read k_1)^{r1}|| rv_2 |=> ( v_3 )^{r2}|| ret_1^2 |=> (false)^{r2}
    || ret_2^2 |=> (true)^{r2}|| ret_1^1 |=> (true)^{r2}|| ret_2^1 |=> (true)^{r2} →  (Df)
(rv_1⊓v_3)^f || rv_1 |=> ( local-read k_1 )^{r1}|| rv_2 |=> (v_3)^{r2}|| ret_1^2 |=> (false)^{r2}
   || ret_2^2 |=> (true)^{r2}|| ret_1^1 |=> (true)^{r2}|| ret_2^1 |=> (true)^{r2} →   (Ap-F)
(rv_1⊓v_3)^f || rv_1 |=> ( v_2 )^{r1}|| rv_2 |=> (v_3)^{r2}|| ret_1^2 |=> (false)^{r2}
    || ret_2^2 |=> (true)^{r2}|| ret_1^1 |=> (true)^{r2}|| ret_2^1 |=> (true)^{r2} →  (Df)
( v_2⊓v_3 )^f || rv_1 |=> (v_2)^{r1}|| rv_2 |=> (v_3)^{r2}|| ret_1^2 |=> (false)^{r2}
```

[1]Conflict-free Replicated Datatype [8]

$$\texttt{||}\ \texttt{ret}_2^2\ \texttt{|=>}\ (\texttt{true})^{\texttt{r}_2}\texttt{||}\ \texttt{ret}_1^1\ \texttt{|=>}\ (\texttt{true})^{\texttt{r}_2}\texttt{||}\ \texttt{ret}_2^1\ \texttt{|=>}\ (\texttt{true})^{\texttt{r}_2}\ \longrightarrow \qquad \text{(AP-F)}$$
$$(\texttt{v}_3)^{\texttt{f}}\ \texttt{||}\ \texttt{rv}_1\ \texttt{|=>}\ (\texttt{v}_2)^{\texttt{r}_1}\texttt{||}\ \texttt{rv}_2\ \texttt{|=>}\ (\texttt{v}_3)^{\texttt{r}_2}\texttt{||}\ \texttt{ret}_1^2\ \texttt{|=>}\ (\texttt{false})^{\texttt{r}_2}$$
$$\texttt{||}\ \texttt{ret}_2^2\ \texttt{|=>}\ (\texttt{true})^{\texttt{r}_2}\texttt{||}\ \texttt{ret}_1^1\ \texttt{|=>}\ (\texttt{true})^{\texttt{r}_2}\texttt{||}\ \texttt{ret}_2^1\ \texttt{|=>}\ (\texttt{true})^{\texttt{r}_2} \qquad \square$$

Remark 3.7. Pure functional programming of a single node database is not the focus of this article. Example 3.6 assumed that writing to a database is done in a pure fashion, namely, *local-write* is pure. For more on that, see the recent developments of Rob Norris[2]. Our focus in Example 3.6 is on showing pure message passing between nodes of a replicated key-value store. $\square$

References

[1] S. H. Haeri. Observational Equivalence and a New Operational Semantics for Lazy Evaluation with Selective Strictness. In Z. Majkic, S.-Y. Hsieh, J. Ma, I. M. M. El Emary, and K. S. Husain, editors, *TMFCS*, pages 143–150. ISRST, July 2010.

[2] S. H. Haeri and P. Van Roy. Distributed λ-Calculus. Technical report, ICTEAM, UCLouvain, Belgium, October 2019. Available online at: http://hdl.handle.net/2078.1/228131.

[3] S. H. Haeri and P. Van Roy. A Family of λ-Calculi with Ports. In 21st *TFP*. LNCS, February 2020. Accepted, Available online at: http://hdl.handle.net/2078.1/228133.

[4] S. H. Haeri and P. Van Roy. Piecewise Relative Observational Purity. In 4th *ProWeb*. ACM, March 2020. Accepted.

[5] G. Khan. The Semantics of a Simple Language for Parallel Programming. *Inf. Proc.*, 74:471–475, 1974.

[6] J. Launchbury. A Natural Semantics for Lazy Evaluation. In 20th *POPL*, pages 144–154. ACM, 1993.

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

[8] M. Shapiro, N. M. Preguiça, C. Baquero, and M. Zawirski. Conflict-Free Replicated Data Types. In X. Défago, F. Petit, and V. Villain, editors, 13th *SSS*, volume 6976 of *LNCS*, pages 386–400. Springer, October 2011.

[2]https://www.youtube.com/watch?v=NJrgj1vQeAI