Appendix - Verifying Safety and Accuracy of Approximate Parallel Programs via Canonical Sequentialization

VIMUTH FERNANDO, University of Illinois at Urbana-Champaign, USA
KEYUR JOSHI, University of Illinois at Urbana-Champaign, USA
SASA MISAILOVIC, University of Illinois at Urbana-Champaign, USA

We present Parallely, a programming language and a system for verification of approximations in parallel message-passing programs. Parallely's language can express various software and hardware level approximations that reduce the computation and communication overheads at the cost of result accuracy.

Parallely's safety analysis can prove the absence of deadlocks in approximate computations and its type system can ensure that approximate values do not interfere with precise values. Parallely's quantitative accuracy analysis can reason about the frequency and magnitude of error. To support such analyses, Parallely presents an approximation-aware version of canonical sequentialization, a recently proposed verification technique that generates sequential programs that capture the semantics of well-structured parallel programs (i.e., ones that satisfy a symmetric nondeterminism property). To the best of our knowledge, Parallely is the first system designed to analyze parallel approximate programs.

We demonstrate the effectiveness of Parallely on eight benchmark applications from the domains of graph analytics, image processing, and numerical analysis. We also encode and study five approximation mechanisms from literature. Our implementation of Parallely automatically and efficiently proves type safety, reliability, and accuracy properties of the approximate benchmarks.

CCS Concepts: • **Computing methodologies → Parallel programming languages**; • **Software and its engineering → Formal software verification**.

Additional Key Words and Phrases: Approximate Computing, Reliability, Accuracy, Safety

ACM Reference Format:
Vimuth Fernando, Keyur Joshi, and Sasa Misailovic. 2019. Appendix - Verifying Safety and Accuracy of Approximate Parallel Programs via Canonical Sequentialization. *Proc. ACM Program. Lang.* 3, OOPSLA (October 2019), 26 pages. https://doi.org/10.1145/3360545

2475-1421/2019/10-ART
https://doi.org/10.1145/3360545

APPENDIX A

In this appendix we present the full syntax semantics, and type rules for our language. In addition we provide the proofs for the properties stated in the paper.

1 DEFINITIONS

Figure 1 shows the syntax of the language. In this section we define key terms and the key definitions.

References. A *reference* is a pair $\langle n_b, \langle n_1, ..., n_k \rangle \rangle \in \texttt{Ref}$ that consists of a base address $n_b \in Loc$ and a dimension descriptor $\langle n_1, ..., n_k \rangle$. References describe the location and the dimension of variables in the heap.

Frames, Stacks, and Heaps. A *frame* σ is an element of the domain $\mathrm{E} = \mathrm{Var} \to \mathrm{Ref}$ which is the set of finite maps from program variables to references. A *heap* $h \in H = \mathbb{N} \to \mathbb{N} \cup \mathbb{F} \cup \{\varnothing\}$ is a finite map from addresses (integers) to values. Values can be an Integer, Float or the special *empty message* ($\varnothing$).

Processes. Individual processes execute their statements in sequential order. Each process has a unique process identifier (Pid). Processes can refer to each other using the process identifier. We do not discuss process creation and removal. We assume that the processes have disjoint variable sets of variable names. We write <pid>.<var> to refer to variable <var> of process <pid>. When unambiguous, we will omit <pid> and just write <var>.

Types. Types in Parallely are either precise (meaning that no approximation can be applied to them) and approximate. Parallely supports integer and floating-point scalars and arrays with different levels of precision.

Typed Channels and Message Orders. Processes communicate by sending and receiving messages over a typed *channel.* There is a separate subchannel for each pair of processes further split by the *type* of message. $\mu \in Channel = Pid \times Pid \times Type \longrightarrow Val^*$. Messages on the same subchannel are delivered in order but there are no guarantees for messages sent on separate (sub)channels.

Programs. We define a program as a parallel composition of processes. We denote a program as $P = [P]_1 \parallel \cdots \parallel [P]_i \parallel \cdots \parallel [P]_n$. Where $1 \ldots n$ are process identifiers. An approximated program executes within *approximation model*, ψ, which in general may contain the parameters for approximation (e.g., probability of selecting original or approximate expression). We define special reliable model 1_ψ, which evaluates the program without approximations.

Global and Local Environments. Each process works on its private environment consisting of a frame and a heap, $\langle \sigma^i, h^i \rangle \in \Lambda = H \times \mathrm{E}$. We define a global configuration as a triple $\langle P, \epsilon, \mu \rangle$ of a program, global environment, and a channel. The global environment is a map from the process identifiers to the local environment $\epsilon \in Env = Pid \mapsto \Lambda$.

Scheduler Distributions. $P_s(i \mid \langle P, \epsilon, \mu \rangle)$ models the probability that the thread with id i is scheduled next. We define it history-less and independent of ϵ contents. For reliability analysis we assume a fair scheduler that in each step has a positive probability for all threads that can take a step in the program.

We make the following assumptions for the reliability analysis to ensure that the scheduler is fair. (The remaining analyses do not take into account this distribution).

(1) $\forall \epsilon, \mu.\ \sum_{\alpha \in Tid} P(\alpha | (P, \epsilon, \mu)) = 1$
(2) $\forall P, \epsilon, \mu.\ \forall \alpha.\ P(\alpha | (P, \epsilon, \mu)) > 0$ iff $\exists\, P', \epsilon' \mu'$ s.t. $(\epsilon, \mu, P) \xrightarrow{\alpha, p}_{\psi} (\epsilon', \mu', P')$

n	$\in$	$\mathbb{N}$	*quantities*
$\mathtt{m}$	$\in$	$\mathbb{N} \cup \mathbb{F} \cup \{\varnothing\}$	*values*
x, b, X	$\in$	Var	*variables*
a	$\in$	ArrVar	*array variables*
α, β	$\in$	Pid	*process ids*
Exp	$\rightarrow$	$m \mid x \mid f(Exp^*) \mid$ $(Exp) \mid Exp\ op\ Exp$	*expressions*
q	$\rightarrow$	$\mathtt{precise} \mid \mathtt{approx}$	*type qualifiers*
t	$\rightarrow$	$\mathtt{int<n>} \mid \mathtt{float<n>}$	*basic types*
T	$\rightarrow$	$q\ t \mid q\ t\ []$	*types*
D	$\rightarrow$	$\mathrm{T\ x} \mid \mathrm{T\ a}[n^+] \mid$ $D; D$	*variable* *declarations*
P	$\rightarrow$	$[D; S]_\alpha \mid$ $\Pi.\alpha : X\ [D; S]_\alpha \mid$ $P \parallel P$	*process* *process group* *process composition*

$S \rightarrow$

$\mathtt{skip}$	*empty program*
$\mid x = Exp$	*assignment*
$\mid x = Exp\ [r]\ Exp$	*probabilistic choice*
$\mid x = b?\ Exp : Exp$	*conditional choice*
$\mid S; S$	*sequence*
$\mid x = a[Exp^+]$	*array load*
$\mid a[Exp^+] = Exp$	*array store*
$\mid \mathtt{if}\ x\ S\ S$	*branching*
$\mid$ repeat n {S}	*repeat n times*
$\mid x = (\mathrm{T})Exp$	*cast*
$\mid \mathtt{for}\ i : [Pid^+]\{S\}$	*iterate over processes*
$\mid \mathrm{send}(\alpha, T, x)$	*send message*
$\mid x = \mathrm{receive}(\alpha, T)$	*receive a message*
$\mid \text{cond-send}(b, \alpha, T, x)$	*conditionally send*
$\mid b, x = \text{cond-receive}(\alpha, T)$	*receive from a cond-send*

Fig. 1. Parallely syntax

E-VAR-C
$$\frac{\langle n_b, \langle 1 \rangle \rangle = \sigma(x)}{\langle x, \sigma, h \rangle \rightarrow_{\psi} \langle h(n_b), \sigma, h \rangle}$$

E-VAR-F
$$\frac{\langle n_b, \langle 1 \rangle \rangle = \sigma(x)}{\langle x, \sigma, h \rangle \xrightarrow{1}_{\psi} \langle n_f, \sigma, h \rangle}$$

E-IOP-R1
$$\frac{\langle e_1, \sigma, h \rangle \xrightarrow{p}_{\psi} \langle e_1', \sigma, h \rangle}{\langle e_1\ op\ \ e_2, \sigma, h \rangle \xrightarrow{p}_{\psi} \langle e_1'\ op\ \ e_2, \sigma, h \rangle}$$

E-IOP-R2
$$\frac{\langle e_2, \sigma, h \rangle \xrightarrow{p}_{\psi} \langle e_2', \sigma, h \rangle}{\langle n\ op\ \ e_2, \sigma, h \rangle \xrightarrow{p}_{\psi} \langle n\ op\ \ e_2', \sigma, h \rangle}$$

E-IOP-C
$$\frac{}{\langle n_1\ op\ \ n_2, \sigma, h \rangle \xrightarrow{1}_{\psi} \langle op(n_1, n_2), \sigma, h \rangle}$$

Fig. 2. Dynamic Semantics of Expressions

DEC-VAR
$$\frac{\langle n_b, h' \rangle = \mathsf{new}(h, \langle 1 \rangle)}{\langle \mathrm{T\ x}, \sigma :: \sigma, h, \mu \rangle \xrightarrow{1}_{\psi} \langle \mathtt{skip}, \sigma[x \mapsto \langle n_b, \langle 1 \rangle m \rangle] :: \sigma, h', \mu \rangle}$$

DEC-ARRAY
$$\frac{\forall i.0 < n_i \qquad \langle n_b, h' \rangle = \mathsf{new}(h, m, \langle n_1 \ldots n_k \rangle) \qquad \sigma' = \sigma[x \mapsto \langle n_b, \langle n_1 .. n_k \rangle m \rangle]}{\langle \mathrm{T\ x}[n_1 \ldots n_k], \sigma :: \sigma, h, \mu \rangle \xrightarrow{1}_{\psi} \langle \mathtt{skip}, \sigma' :: \sigma, h', \mu \rangle}$$

Fig. 3. Semantics of Declarations

2 LANGUAGE SEMANTICS

Figure 2 defines the semantics for expressions. Figures 3, 5, and 4 define the semantics for a single process running sequentially. Figure 6 presents the global semantics of a parallel program.

E-Assign-R

$$\frac{\langle e, \sigma, h\rangle \xrightarrow{p}_{\psi} \langle e', \sigma, h\rangle}{\langle x = e, \sigma, h, \mu\rangle \xrightarrow{p}_{\psi} \langle x = e', \sigma, h, \mu\rangle}$$

E-Assign-C

$$\frac{\langle n_b, \langle 1\rangle\rangle = \sigma(x)}{\langle x = n, \sigma, h, \mu\rangle \xrightarrow{1}_{\psi} \langle \mathtt{skip}, \sigma, h[n_b \mapsto n], \mu\rangle}$$

E-Assign-Prob-True

$$\frac{}{\langle x = e_1\ [r]\ e_2, \sigma, h, \mu\rangle \xrightarrow{r}_{\psi} \langle x = e_1, \sigma, h, \mu\rangle}$$

E-Assign-Prob-False

$$\frac{}{\langle x = e_1\ [r]\ e_2, \sigma, h, \mu\rangle \xrightarrow{1-r}_{\psi} \langle x = e_2, \sigma, h, \mu\rangle}$$

E-Assign-Approx-True

$$\frac{\langle l, \langle 1\rangle\rangle = \sigma(b) \qquad h[l] \neq 0}{\langle x = e_1\ [b]\ e_2, \sigma, h, \mu\rangle \xrightarrow{1}_{\psi} \langle x = e_1, \sigma, h, \mu\rangle}$$

E-Assign-Approx-True

$$\frac{\langle l, \langle 1\rangle\rangle = \sigma(b) \qquad h[l] = 0}{\langle x = e_1\ [b]\ e_2, \sigma, h, \mu\rangle \xrightarrow{1}_{\psi} \langle x = e_2, \sigma, h, \mu\rangle}$$

E-Seq-R1

$$\frac{\langle s_1, \sigma, h, \mu\rangle \xrightarrow{p}_{\psi} \langle s_1', \sigma', h', \mu'\rangle}{\langle s_1; s_2, \sigma, h, \mu\rangle \xrightarrow{p}_{\psi} \langle s_1'; s_2, \sigma', h', \mu'\rangle}$$

E-Seq-R2

$$\frac{}{\langle \mathtt{skip}; s_2, \sigma, h, \mu\rangle \xrightarrow{1}_{\psi} \langle s_2, \sigma, h, \mu\rangle}$$

E-If-True

$$\frac{\langle n_b, \langle 1\rangle\rangle = \sigma(x) \qquad h[n_b] \neq 0}{\langle \mathtt{if}\ x\ s_1\ s_2, \sigma, h, \mu\rangle \xrightarrow{1}_{\psi} \langle s_1, \sigma, h, \mu\rangle}$$

E-If-False

$$\frac{\langle n_b, \langle 1\rangle\rangle = \sigma(x) \qquad h[n_b] = 0}{\langle \mathtt{if}\ x\ s_1\ s_2, \sigma, h, \mu\rangle \xrightarrow{1}_{\psi} \langle s_2, \sigma, h, \mu\rangle}$$

E-Send

$$\frac{isPid(\beta) \qquad \langle n_b, \langle 1\rangle\rangle = \sigma(y) \qquad h[n_b] = n \qquad \mu[\langle \alpha, \beta, t\rangle] = m}{\langle [\mathrm{send}(\beta, t, y)]_{\alpha}, \sigma, h, \mu\rangle \xrightarrow{1}_{\psi} \langle \mathtt{skip}, \sigma, h, \mu[\langle \alpha, \beta, t\rangle \mapsto m + +n]\rangle}$$

E-Receive

$$\frac{\mu[(\beta = w) \vee (\beta \in w) \qquad \langle \beta, \alpha, t\rangle] = m :: n \qquad \langle n_b, \langle 1\rangle\rangle = \sigma(x)}{\langle [x = \mathrm{receive}(w, t)]_{\alpha}, \sigma, h, \mu\rangle \xrightarrow{p}_{\psi} \langle \mathtt{skip}, \sigma, h[n_b \mapsto v], \mu[\langle \beta, \alpha, t\rangle \mapsto n]\rangle}$$

E-CondSend-True

$$\frac{\begin{array}{c}\langle l, \langle 1\rangle\rangle = \sigma(b) \qquad h[l] \neq 0 \qquad isPid(\beta) \\ \langle n_b, \langle 1\rangle\rangle = \sigma(y) \qquad h[n_b] = v \qquad \mu[\langle \alpha, \beta, t\rangle] = m\end{array}}{\langle [\mathrm{cond\text{-}send}(b, \beta, t, y)]_{\alpha}, \sigma, h, \mu\rangle \xrightarrow{1}_{\psi} \langle \mathtt{skip}, \sigma, h, \mu[\langle \alpha, \beta, t\rangle \mapsto m + +n]\rangle}$$

E-CondSend-False

$$\frac{\langle l, \langle 1\rangle\rangle = \sigma(b) \qquad h[l] = 0 \qquad isPid(\beta) \qquad \mu[\langle \alpha, \beta, t\rangle] = m}{\langle [\mathrm{cond\text{-}send}(b, \beta, t, y)]_{\alpha}, \sigma, h, \mu\rangle \xrightarrow{1}_{\psi} \langle \mathtt{skip}, \sigma, h, \mu[\langle \alpha, \beta, t\rangle \mapsto m + +\varnothing]\rangle}$$

E-CondReceive-True

$$\frac{\mu[\langle \beta, \alpha, t\rangle] = m :: n \qquad (\beta = w) \vee (\beta \in w) \qquad \langle n_1, \langle 1\rangle\rangle = \sigma(x) \qquad \langle n_2, \langle 1\rangle\rangle = \sigma(b)}{\langle [b, x = \mathrm{cond\text{-}receive}(\beta, t)]_{\alpha}, \sigma, h, \mu\rangle \xrightarrow{1}_{\psi} \langle \mathtt{skip}, \sigma, h[n_1 \mapsto v][n_2 \mapsto 1], \mu[\langle \beta, \alpha, t\rangle \mapsto n]\rangle}$$

E-CondReceive-False

$$\frac{\mu[\langle \beta, \alpha, t\rangle] = \varnothing :: m \qquad (\beta = w) \vee (\beta \in w) \qquad \langle n_b, \langle 1\rangle\rangle = \sigma(b)}{\langle [b, x = \mathrm{cond\text{-}receive}(\beta, t)]_{\alpha}, \sigma, h, \mu\rangle \xrightarrow{1}_{\psi} \langle \mathtt{skip}, \sigma, h[n_b \mapsto 0], \mu[\langle \beta, \alpha, t\rangle \mapsto m]\rangle}$$

E-Cast-R

$$\frac{\langle e, \sigma, h\rangle \xrightarrow{p}_{\psi} \langle e', \sigma, h\rangle}{\langle x = (\mathrm{T})e, \sigma, h, \mu\rangle \xrightarrow{p}_{\psi} \langle x = (\mathrm{T})e', \sigma, h, \mu\rangle}$$

E-Cast-C

$$\frac{n' = \mathtt{convert}(T, n) \qquad \langle n_b, \langle 1\rangle\rangle = \sigma(x)}{\langle \mathrm{x} = (\mathrm{T})n, \sigma, h, \mu\rangle \xrightarrow{1}_{\psi} \langle \mathtt{skip}, \sigma, h[n_b \mapsto n'], \mu\rangle}$$

E-Par-Iter

$$\frac{}{\begin{array}{c}\langle \mathtt{for}\ i : [\alpha_1 \ldots \alpha_k]\{S\}, \sigma, h, \mu\rangle \xrightarrow{1}_{\psi} \\ \langle S[\alpha_1/i]; \ldots; S[\alpha_k/i], \sigma, h, \mu\rangle\end{array}}$$

Fig. 4. Sequential Semantics of Statements

E-Array-Load-Idx

$$\frac{\langle e_i, \sigma, h\rangle \xrightarrow{p}_{\psi} \langle e_i', \sigma, h\rangle}{\langle x = a[n_1, \ldots, e_i, \ldots, e_k], \sigma, h, \mu\rangle \xrightarrow{p}_{\psi} \langle x = a[n_1, \ldots, e_i', \ldots, e_k], \sigma, h, \mu\rangle}$$

E-Array-Load-C

$$\frac{\langle n_b, \langle l_1, \ldots, l_k\rangle\rangle = \sigma(x) \qquad n_o = l_k + \Sigma_{i=0}^{k-1} n_i \cdot l_i \qquad n = h(n_b + n_o)}{\langle x = a[n_1, \ldots, n_k], \sigma, h, \mu\rangle \xrightarrow{p}_{\psi} \langle x = n, \sigma, h, \mu\rangle}$$

E-Array-Store-Idx

$$\frac{\langle e_i, \sigma, h\rangle \xrightarrow{p}_{\psi} \langle e_i', \sigma, h\rangle}{\langle a[n_1, \ldots, e_i, \ldots, e_k] = x, \sigma, h, \mu\rangle \xrightarrow{p}_{\psi} \langle a[n_1, \ldots, e_i', \ldots, e_k] = x, \sigma, h, \mu\rangle}$$

E-Array-Store-C

$$\frac{\begin{array}{c}\langle n_b, \langle l_1, \ldots, l_k\rangle\rangle = \sigma(x) \qquad n_o = l_k + \Sigma_{i=0}^{k-1} n_i \cdot l_i \\ \langle n_b', \langle 1\rangle\rangle = \sigma(x) \qquad h[n_b'] = v \qquad \psi(wr(m)) = 1\end{array}}{\langle a[n_1, \ldots, n_k] = x, \sigma, h, \mu\rangle \xrightarrow{1}_{\psi} \langle skip, \sigma, h[(n_b + n_o) \mapsto v], \mu\rangle}$$

Fig. 5. Sequential Semantics of Arrays

GLOBAL-STEP

$$\frac{p_s = P_s[\alpha \mid (\epsilon, \mu, P_i \parallel P_j)] \qquad E[\alpha] = \langle \sigma, h\rangle \qquad \langle P_\alpha, \sigma, h, \mu\rangle \xrightarrow{p}_{\psi} \langle P_\alpha', \sigma', h', \mu'\rangle \qquad p' = p \cdot p_s}{(\epsilon, \mu, P_\alpha \parallel P_\beta) \xrightarrow{\alpha, p'}_{\psi} (\epsilon[i \mapsto \langle \sigma', h'\rangle], \mu', P_\alpha' \parallel P_\beta)}$$

Fig. 6. Global Semantics

E-Assign-Prob-Exact

$$\frac{}{\langle x = e_1\ [r]\ e_2, \sigma, h, \mu\rangle \xrightarrow{1}_{1_{\psi}} \langle x = e_1, \sigma, h, \mu\rangle}$$

E-Cast-Exact

$$\frac{}{\langle x = (\mathrm{T})e, \sigma, h, \mu\rangle \xrightarrow{1}_{1_{\psi}} \langle x = e, \sigma, h, \mu\rangle}$$

Fig. 7. Exact Execution Semantics of Statements (Selection)

$$\textsc{TR-Val}\ \frac{\mathtt{type(n)} = t}{\Theta \vdash n : \text{precise } t} \qquad \textsc{TR-Var}\ \frac{\Theta(x) \ = \ T}{\Theta \vdash x : T} \qquad \textsc{TR-Iop}\ \frac{\Theta \vdash e_1 : T \qquad \Theta \vdash e_2 : T}{\Theta \vdash e_1 \ op \ e_2 : T} \qquad \textsc{TR-Iop-approx}\ \frac{\Theta \vdash e_1 : q\ t \qquad \Theta \vdash e_2 : q'\ t}{\Theta \vdash e_1 \ op \ e_2 : \text{approx } t}$$

Fig. 8. Types for Integer Expressions

$$\textsc{TR-Skip}\ \ \Theta \vdash \mathtt{skip} : \Theta \qquad \textsc{TR-Var}\ \frac{\Theta \vdash x : T \qquad \Theta \vdash e : \Theta}{\Theta \vdash \mathrm{x = e} : \Theta} \qquad \textsc{TR-Var2}\ \frac{\Theta \vdash x : \text{approx } t \qquad \Theta \vdash e : q\ t}{\Theta \vdash \mathrm{x = e} : \Theta}$$

$$\textsc{TR-Prob}\ \frac{\Theta \vdash e_1 : q\ t \qquad \Theta \vdash e_2 : q'\ t \qquad \Theta \vdash x : \text{approx } t}{\Theta \vdash \mathrm{x} = e_1\ [p]\ e_2 : \Theta}$$

$$\textsc{TR-ApproxAssign}\ \frac{\Theta \vdash e_1 : q\ t \qquad \Theta \vdash e_2 : q'\ t \qquad \Theta \vdash x : \text{approx } t \qquad \Theta \vdash b : q''\ \mathtt{int}}{\Theta \vdash \mathrm{x} = e_1\ [b]\ e_2 : \Theta} \qquad \text{TR-SEQ}\ \frac{\Theta \vdash s_1 : \Theta \qquad \Theta \vdash s_2 : \Theta}{\Theta \vdash s_1; s_2 : \Theta}$$

$$\textsc{TR-If}\ \frac{\Theta \vdash b : \text{precise } \mathtt{int}}{\Theta \vdash \mathtt{if}\ b\ s_1\ s_2 : \Theta} \qquad \textsc{TR-Array-load}\ \frac{\begin{array}{c}\Theta \vdash e_1 : \text{precise } \mathtt{int} \ldots \Theta \vdash e_k : \text{precise } \mathtt{int}\\ \Theta \vdash a : q\ t[] \qquad \Theta \vdash x : q\ t\end{array}}{\Theta \vdash x = a[e_1 \ldots e_k] : \Theta} \qquad \textsc{TR-Array-load2}\ \frac{\begin{array}{c}\Theta \vdash e_1 : \text{precise } \mathtt{int} \ldots \Theta \vdash e_k : \text{precise } \mathtt{int}\\ \Theta \vdash a : q\ t[] \qquad \Theta \vdash x : \text{approx } t\end{array}}{\Theta \vdash x = a[e_1 \ldots e_k] : \Theta}$$

$$\textsc{TR-Array-store}\ \frac{\begin{array}{c}\Theta \vdash e_1 : \text{precise } \mathtt{int} \ldots \Theta \vdash e_k : \text{precise } \mathtt{int}\\ \Theta \vdash a : q\ t[] \qquad \Theta \vdash e : q\ t\end{array}}{\Theta \vdash a[e_1 \ldots e_k] = \mathrm{e} : \Theta} \qquad \textsc{TR-Array-store2}\ \frac{\begin{array}{c}\Theta \vdash e_1 : \text{precise } \mathtt{int} \ldots \Theta \vdash e_k : \text{precise } \mathtt{int}\\ \Theta \vdash a : \text{approx } t[] \qquad \Theta \vdash e : q\ t\end{array}}{\Theta \vdash a[e_1 \ldots e_k] = \mathrm{e} : \Theta}$$

$$\textsc{TR-Send}\ \frac{\Theta \vdash y : T}{\Theta \vdash \text{send}(q, T, y) : \Theta} \qquad \textsc{TR-Receive}\ \frac{\Theta \vdash x : T}{\Theta \vdash x = \text{receive}(q, T) : \Theta} \qquad \textsc{TR-CondSend}\ \frac{\Theta \vdash b : \text{precise } \mathtt{int} \qquad \Theta \vdash y : \text{approx } t \qquad T = \text{approx } t}{\Theta \vdash \text{cond-send}(b, q, T, y) : \Theta}$$

$$\textsc{TR-CondReceive}\ \frac{\Theta \vdash x : \text{approx } t \qquad T = \text{approx } t \qquad \Theta \vdash b : \text{approx } \mathtt{int}}{\Theta \vdash b, x = \text{cond-receive}(q, T) : \Theta} \qquad \textsc{TR-Cast}\ \frac{\Theta \vdash x : \text{approx } t \qquad \Theta \vdash e : q'\ t}{\Theta \vdash \mathrm{x = (q\ t)\ e} : \Theta}$$

Fig. 9. Types for Statements

APPENDIX B

3 NON-INTERFERENCE

Equality. We use similar definitions as in EnerJ [3].
We use ≅ to denote equality disregarding approximate values for values, environments and heaps. For primitive values, $v \cong v'$ iff they have the same type $q\ T$ and either q is approx or $v = v'$. For heaps $h \cong h'$ iff they have the same set of addresses M and $\forall m \in M.\ h(m) \cong h'(m)$ (Similarly for the frames, stacks).
We use the same definition for channels, $\mu \cong \mu'$ if $domain(\mu) = domain(\mu')$ and $\forall (p, q, \text{precise } t) \in domain(\mu)\ \mu[(p, q, \text{precise } t)] = \mu'(p, q, \text{precise } t)$

3.1 Sequential Non-Interference

If $\Theta \vdash s : \Theta$, $\langle s,\ \sigma_s, h_s, \mu_s \rangle \rightharpoonup_{\psi} \langle s',\ \sigma_f, h_f, \mu_f \rangle$, $\sigma_s \cong \sigma'_s$, $h_s \cong h'_s$, and $\mu_s \cong \mu'_s$ Then there exists, $(\sigma'_f, h'_f, \mu'_f)$ s.t. $\langle s,\ \sigma'_s, h'_s, \mu'_s \rangle \rightharpoonup_{\psi} \langle s',\ \sigma'_f, h'_f, \mu'_f \rangle$ and $\sigma_f \cong \sigma'_f$, $h_f \cong h'_f$, and $\mu_s \cong \mu'_s$

Proof. We will use rule induction on the semantics.

Case 1: E-ASSIGN-R
The environment is not modified. So, $h_s = h_f$ and $h'_s = h'_f$. As, $h_s \cong h'_s$ we can trivially say $h_f \cong h'_f$. Same argument holds for the stack and mail box.

Case 2: E-ASSIGN-C
The stack and the channel does not change.
From assumption $\Theta \vdash x = n : \Theta$, therefore either both x and n both have the same type, or $\Theta \vdash x :$ approx t and $\Theta \vdash n : q\ t$. In the first case, If both x and n are approx then, $h_s \cong h_f$ by definition as only the approximate value changes and the property holds. If both values are precise they will be the same in h_s and h'_s and we can take the same step. In the second case, x is approx and n is precise. Again in this case only approx values in the heap will change.

Case 3: E-Assign-Prob-True and E-Assign-Prob-False
The type rule `TR-Prob` ensures that the assigned variable is approximate. Therefore only the approximate regions of the environment changes and the property holds.

Case 4: E-SEQ-R1, E-SEQ-R2
Follows directly from the inductive hypothesis.

Case 5: E-If, E-If-True, E-If-False
In the case of rule `E-If`, The environment does not change and the property is satisfied trivially and since $\Theta \vdash$ `if` $b\ s_1\ s_2 : \Theta$ we know $\Theta \vdash b :$ precise `int` therefore, the guard evaluates to the same value in both $\langle \sigma_s, h_s, \mu_s \rangle$ and $\langle \sigma'_s, h'_s, \mu'_s \rangle$ and takes the same branch. In the case of rules `E-If-True` and `E-If-False` the property follows from the inductive hypothesis.

Case 6: E-ARRAY-LOAD-IDX, E-ARRAY-LOAD-C
In E-ARRAY-LOAD-IDX the environment does not change. As $\Theta \vdash$ x = a[$e_1 ... e_k$] : Θ. All e_i has precise type. Therefore all array indices will evaluate to the same value in $\langle \sigma'_s, h'_s, \mu'_s \rangle$. In addition if $\Theta \vdash x :$ approx t then $h_f \cong h_s$ and the property holds.
Similarly, if $\Theta \vdash x :$ precise t then $\Theta \vdash a :$ precise t and the resultant value n will be the same in both h_s and h'_s resulting in the same update to heap.

Case 7: E-ARRAY-STORE-IDX, E-Array-Store-C
As $\Theta \vdash$ a[$e_1 ... e_k$] = x : Θ. As in the previous case, all e_i has precise type. Therefore all array indices will evaluate to the same value in $\langle \sigma'_s, h'_s, \mu'_s \rangle$. In addition if $\Theta \vdash x :$ approx t, then $\Theta \vdash a :$ approx t and $h_f \cong h_s$ and the property holds.
Similarly, if $\Theta \vdash x :$ precise t then $\Theta \vdash x :$ precise t and the resultant value n will be the same in both h_s and h'_s resulting in the same update to heap.

Case 8: E-SEND
As $\Theta \vdash$ send(Pid, T, v) : Θ, v has the type T. If T is a approx type only the approximate part of the mailbox changes and the property holds. If T is precise, type safety also ensures that v has precise type and will evaluate to the same value under $\langle \sigma'_s, h'_s, \mu'_s \rangle$ resulting in a equivalent environment.

Case 9: E-Receive
As $\Theta \vdash$ receive(Pid, T) : Θ, x has some type T. If x is an approx type only the approximate part of

the mailbox and environment changes and the property holds. If T is precise, the precise part of the mailbox is accessed, and the value in $\langle \sigma_s', h_s', \mu_s' \rangle$ will be the same, therefore the final environment will be the same for any equivalent start environment.

Case 10: E-CONDRECEIVE-TRUE, E-CONDRECEIVE-FALSE
In the case of E-CONDRECEIVE-TRUE the behavior is similar to E-Receive. But since $\Theta \vdash x = \text{cond-receive}(\mathit{Pid}, T) : \Theta, T = \text{ approx } t$. Therefore the only change in the channel is to $\mu(\alpha, \beta, \text{approx } t)$ therefore $\mu_s \cong \mu_f$. Similarly, since $\Theta \vdash x : \text{approx} t$, $h_s \cong h_f$.
Same argument for E-CONDRECEIVE-FALSE. In this case the heap and stack does not change.

Case 11: E-Cast-R and E-Cast-E
The type rule `TR-Cast` ensures that the assigned variable is approximate. Therefore only the approximate regions of the environment changes and the property holds.

3.2 Distributed Noninterference

Concurrent Non-interference says that if each individual process in the program was well typed then the parallel program has similar noninterference property guaranteed.

Theorem 1 (Parallel Non-Interference). *Suppose $P_i \parallel P_j$ is well typed under Θ and $\forall \epsilon, \epsilon', \epsilon_f \in$ Env and $\mu, \mu, \mu_f \in$ Channel, such that, $\langle \epsilon, \mu \rangle \cong \langle \epsilon', \mu' \rangle$, if $(\epsilon, \mu, P_i \parallel P_j) \longrightarrow_{\psi} (\epsilon_f, \mu_f, P_i' \parallel P_j)$, then there exists $\epsilon_f' \in$ Env and $\mu_f' \in$ Channel such that $(\epsilon', \mu', P_i \parallel P_j) \longrightarrow_{\psi} (\epsilon_f', \mu_f', P_i' \parallel P_j)$ and $\langle \epsilon_f, \mu_f \rangle \cong \langle \epsilon_f', \mu_f' \rangle$*

proof. If $(\epsilon, \mu, P_i \parallel P_j) \longrightarrow_{\psi} (\epsilon_f, \mu_f, P_i' \parallel P_j)$ then from the semantics of global execution we can see that there exist i such that $\epsilon[i] = \langle \sigma, h \rangle$ and $\langle P_i,\ \sigma, h, \mu \rangle \rightharpoonup \langle P_i',\ \sigma_f, h_f, \mu_f \rangle$.
From sequential non-interference we know that For any $\sigma' \cong \sigma$, $h' \cong h$, and $\mu' \cong \mu$ there exists $(\sigma_f', h_f', \mu_f')$ s.t. $\langle s,\ \sigma', h', \mu' \rangle \rightharpoonup \langle s',\ \sigma_f', h_f', \mu_f' \rangle$ and $\sigma_f \cong \sigma_f'$, $h_f \cong h_f'$, and $\mu \cong \mu_f'$
So we can consider ϵ', where $\epsilon' = \epsilon[i \mapsto \langle \sigma', h' \rangle]$ Consider the same transition as before, We will end up at $\epsilon_f' = \epsilon_f[i \mapsto \langle \sigma_f', h_f' \rangle]$.
$\sigma_f' \cong \sigma_f$, $h_f' \cong h_f$, therefore $\epsilon_f' \cong \epsilon_f$

3.3 Type safety

Lemma (Subject reduction for expressions). *If $\Theta \vdash e : T$ and $\langle e, \cdot, \cdot \rangle \rightharpoonup_{\psi} \langle e', \cdot, \cdot \rangle$ then $\Theta \vdash e' : T$*

Proof. proof is by rule induction on the typing rules for expressions.

Lemma 2 (For individual processes the type system is sound.). *For a single process, assuming there are no deadlocks, if $\Theta \vdash s : \Theta$, then either $\langle s, \sigma, h, \mu \rangle \rightharpoonup \langle \texttt{skip}, \sigma, h, \mu \rangle$ or $\langle s, \sigma, h, \mu \rangle \rightharpoonup \langle s', \sigma, h, \mu \rangle$ and $\Theta \vdash s' : \Theta$*

Proof Sketch. We prove this property by induction on the typing rules. Since we assume there are no deadlocks all processes would be eventually scheduled and the statement will be executed. Most statements evaluate to skip in a single step and the proof is straightforward. We will present several cases the proof of the remaining cases are similar.
Case: $x = e$ If $\langle x = e, \sigma, h, \mu \rangle \rightharpoonup \langle x = e', \sigma, h, \mu \rangle$, we know from the subject reduction lemma for expressions that $\Theta \vdash e' : T$. Therefore if $\Theta \vdash s : \Theta$, then either $\Theta \vdash x : T$ and $\Theta \vdash e : T$ in which case the property holds as $\Theta \vdash e' : T$ or $\Theta \vdash x : \text{approx } t$ and $\Theta \vdash e : q\ t$ which again will be satisfied as $\Theta \vdash e' : q\ t$.
Case: send(q, T, x) Consider send statements send(q, T, x), send statements are always enabled and will evaluate to `skip`.

Case: x=receive(q, T) Receive statements will eventually get enabled as we assume there are no deadlocks and evaluate to `skip`.
Case: x=e_1 $[r]$ e_2 Probabilistic choice statements are always enabled as we assume there are no deadlocks and evaluate to either $x = e_1$ or $x = e_2$. From the assumption we know that $\Theta \vdash$ x = e_1 $[p]$ e_2 : approx t based on type rule `TR-Prob`. Therefore, $\Theta \vdash x$: approx t and we can say that $\Theta \vdash$ x = e_1 : approx t and $\Theta \vdash$ x = e_2 : approx t based on `TR-Var2`
The remaining cases are similar.

Theorem 2 (The type system is sound.). *If* $\emptyset, \emptyset, P \rightsquigarrow^* \emptyset, \Delta,$ `skip` *and* $\Theta \vdash P : \Theta$*, then either* $(\cdot, \cdot, P) \longrightarrow_{\psi} (\cdot, \cdot,$ `skip`$)$
or $(\cdot, \cdot, P) \longrightarrow_{\psi} (\cdot, \cdot, P')$ *and* $\Theta \vdash P' : \Theta$

Proof sketch. As the program P can be sequentialized there are no deadlocks. Therefore there exist at least one individual process that can take a step. From lemma 2 we know that this step will preserve the type of the statement and therefore the entire program will remain well typed.

APPENDIX C

4 REWRITE RULES

We define rewrite rules of the form, $\Gamma, \Delta, P \rightsquigarrow \Gamma', \Delta', P'$. The key rules are available in Figure 10. In addition we define *guarded expressions* to support our rewriting steps. We redefine the interpretation of contexts as follows for guarded expressions,

$$\text{If } \Gamma(\alpha, \beta, t) = (b : n),\ [\![\Gamma(\alpha, \beta, t)]\!]_\sigma = \begin{cases} \sigma[n], & \text{if } \sigma[b] \neq 0 \\ \varnothing, & \text{else} \end{cases}$$

R-Send

$$\frac{\begin{array}{c}\Delta \models x = \beta \qquad \beta \textit{ is a Pid} \\ \Gamma[\alpha, \beta, t] = m \qquad \Gamma' = \Gamma[\alpha, \beta, t \mapsto m + +y]\end{array}}{\Gamma, \Delta, [\text{send}(x, t, y)]_\alpha \rightsquigarrow \Gamma', \Delta, \texttt{skip}}$$

R-Receive

$$\frac{\begin{array}{c}\Delta \models x = \beta \qquad \beta \textit{ is a Pid} \\ \Gamma[\beta, \alpha, t] = m :: n \qquad \Gamma' = \Gamma[\beta, \alpha, t \mapsto n] \qquad \Delta' = \Delta; \mathrm{y} = \mathrm{m}\end{array}}{\Gamma, \Delta, [y = \text{receive}(x, t)]_\alpha \rightsquigarrow \Gamma', \Delta', \texttt{skip}}$$

R-CondSend

$$\frac{\begin{array}{c}\Delta \models x = \beta \qquad \beta \textit{ is a Pid} \\ \Gamma[\alpha, \beta, t] = m \qquad \Gamma' = \Gamma[\alpha, \beta, t \mapsto m + +(b : y)]\end{array}}{\Gamma, \Delta, [\text{cond-send}(b, x, t, y)]_\alpha \rightsquigarrow \Gamma', \Delta, \texttt{skip}}$$

R-CondReceive

$$\frac{\begin{array}{c}\Delta \models x = \beta \qquad \beta \textit{ is a Pid} \qquad \Gamma[\beta, \alpha, t] = (b' : m) :: n \\ \Gamma' = \Gamma[\beta, \alpha, t \mapsto n] \qquad \Delta' = \Delta; \mathrm{b} = b'?\ 1 : 0; \mathrm{y} = b'?\ m : x\end{array}}{\Gamma, \Delta, [b, y = \text{cond-receive}(x, t)]_\alpha \rightsquigarrow \Gamma', \Delta', \texttt{skip}}$$

R-Context

$$\frac{\Gamma, \Delta, A \rightsquigarrow \Gamma', \Delta', A'}{\Gamma, \Delta, \mathrm{A}; \mathrm{B} \rightsquigarrow \Gamma', \Delta', \mathrm{A'}; \mathrm{B}}$$

Fig. 10. Rewrite Rules

5 REWRITE RULE SOUNDNESS

5.1 Definitions

Definition (Transitive closure of global semantics). $\rightarrow^*$ *is defined as the transitive closure over global semantics rules.*

Definition (Process-wise composition). $A \ltimes B$ *is the process-wise composition of A and B. For each process α in B, $A \ltimes B$ sequences the statements belonging to α in A before those in B. This definition is from [1].*

Definition (Interpretation of stores). $\epsilon \in [\![\Delta]\!]_{\epsilon_0}$ *if and only if* $(\epsilon_0, \varnothing, \Delta) \rightarrow^* (\epsilon, \varnothing, \texttt{skip})$. *This definition is from [1].*

Definition (Interpretation of contexts). $\mu \in [\![\Gamma]\!]_\epsilon$ *if and only if*$\forall(\alpha, \beta, t) \in dom(\Gamma).\mu(\alpha, \beta, t) = [\![\Gamma(\alpha, \beta, t)]\!]_\epsilon$. *This definition is from [1].*

Definition (Interpretation of stores and contexts). $(\epsilon, \mu) \in [\![\Delta, \Gamma]\!]_{\epsilon_0}$ *if and only if*

- $\epsilon \in [\![\Delta]\!]_{\epsilon_0}$

- *$\mu \in [\![\Gamma]\!]_\epsilon$*
- *for all constraints $\{x \in X\}$ in Γ,$\epsilon(x) \in \epsilon(X)$*
- *for all constraints $\{\varnothing \subset X \subseteq Y\}$ in Γ,$\varnothing \subset \epsilon(X) \subseteq \epsilon(Y)$*

This definition is from [1].

Definition (Preorder on stores and buffers).

$$\epsilon \preceq \epsilon' \leftrightarrow dom(\epsilon) \subseteq dom(\epsilon') \wedge \forall x \in dom(\epsilon).\epsilon'(x) = \epsilon(x)$$
$$\mu \preceq \mu' \leftrightarrow dom(\mu) \subseteq dom(\mu') \wedge \forall x \in dom(\mu).\exists m.\mu'(x) = \mu(x) + +m$$

This definition is from [1].

Definition (Halted processes). *α is a halted process, i.e. $\alpha \in hprocs(\epsilon, \mu, P)$ if any of the following hold:*

- *α's remaining program is* `skip` *or an error.*
- *α's next statement is a receive or cond-receive, but there is no matching send or cond-send in the rest of the program.*

This definition is from [1].

Definition (Restriction of program stores and buffers). *$\epsilon|_X$ is the projection of ϵ to the set of variables local to the processes in X. $\mu|_X$ is the projection of μ to the subchannels whose destination process is a process in X. This definition is from [1].*

Definition (Preorder on halted environments).

$$(\epsilon, \mu, P) \preceq (\epsilon', \mu', P') \leftrightarrow \epsilon|_H \preceq \epsilon'|_H \wedge \mu|_H \preceq \mu'|_H$$

Where $H = hprocs(\epsilon, \mu, P)$. This definition is from [1].

Definition (Simulation on environments). *$(\epsilon, \mu, P) \sqsubseteq (\epsilon', \mu', P')$ if and only if, for all $(\epsilon, \mu, P) \longrightarrow^* (\epsilon_f, \mu_f, P_f)$, there exists $(\epsilon'_f, \mu'_f, P'_f)$, such that $(\epsilon', \mu', P') \longrightarrow^* (\epsilon'_f, \mu'_f, P'_f)$ and $(\epsilon_f, \mu_f, P_f) \preceq (\epsilon'_f, \mu'_f, P'_f)$. This definition is from [1].*

Definition (Simulation on rewrite rules). *$\Gamma, \Delta, P \sqsubseteq \Gamma', \Delta; \Delta', P'$ if and only if, for all P_x such that $P \ltimes P_x$ is symmetrically nondeterministic,*

$$\forall(\epsilon, \mu) \in [\![\Delta, \Gamma]\!]_\varnothing.\exists(\epsilon', \mu') \in [\![\Delta; \Delta', \Gamma']\!]_\varnothing.(\epsilon, \mu, P \ltimes P_x) \sqsubseteq (\epsilon', \mu', P' \ltimes P_x)$$

This definition is from [1].

Definition (Left movers). *s_1 is a left mover in $(\epsilon, \mu, P||[s_1; s]_\alpha)$ if and only if*

- *If s_1 is enabled in $(\epsilon, \mu, P||[s_1; s]_\alpha)$, and $(\epsilon, \mu, P||[s_1; s]_\alpha) \longrightarrow^* (\epsilon', \mu', P'||[s_1; s]_\alpha)$, then s_1 is still enabled in $(\epsilon', \mu', P'||[s_1; s]_\alpha)$.*
- *If $(\epsilon, \mu, P||[s_1; s]_\alpha) \xrightarrow{\beta} (\epsilon_0, \mu_0, P'||[s_1; s]_\alpha) \xrightarrow{\alpha} (\epsilon', \mu', P'||[s]_\alpha)$ then there exists ϵ_1 and μ_1 such that $(\epsilon, \mu, P||[s_1; s]_\alpha) \xrightarrow{\alpha} (\epsilon_1, \mu_1, P||[s]_\alpha) \xrightarrow{\beta} (\epsilon', \mu', P'||[s]_\alpha)$. That is, s_1 commutes to the left.*

This definition is from [1].

5.2 Left Movers

Lemma. *For all $\epsilon, \mu, P, \alpha, s$, cond-send$(b, x, t, m)$ is a left mover in $(\epsilon, \mu, P||[$cond-send$(b, x, t, m); s]_\alpha)$.*

Proof: The proof is by definition of left movers. cond-send is always enabled when it is the first statement in a process. This satisfies the first condition in the definition of left movers.

Suppose $(\epsilon, \mu, P||[\text{cond-send}(b, x, t, m); s]_\alpha) \xrightarrow{\beta} (\epsilon_0, \mu_0, P_0||[\text{cond-send}(b, x, t, m); s]_\alpha) \xrightarrow{\alpha} (\epsilon_1, \mu_1, P_0||[s]_\alpha)$ and $[s_1]_\beta$ is the first statement in β. Statement $[s_1]_\beta$ is enabled at the start. Since cond-send can only push to message queues where the source is α and does not affect any variables, $[s_1]_\beta$ cannot be disabled if cond-send is run first instead.

Let $(\epsilon, \mu, P||[\text{cond-send}(b, x, t, m); s]_\alpha) \xrightarrow{\alpha} (\epsilon_2, \mu_2, P||[s]_\alpha) \xrightarrow{\beta} (\epsilon_3, \mu_3, P_0||[s]_\alpha)$. Now we need to prove that $\epsilon_1 = \epsilon_3$ and $\mu_1 = \mu_3$.

Statement $[s_1]_\beta$ can only access and modify variables that are not local to α. cond-send does not modify any variables. $[s_1]_\beta$ may push messages into message queues whose source is not α and may pop messages from queues whose destination is not α. cond-send may only push messages to queues whose source is α. In short, the actions performed by $[s_1]_\beta$ and cond-send do not interfere with each other. Therefore, the changes made by $[s_1]_\beta$ to convert ϵ to ϵ_0 and μ to μ_0 are the same changes as those made by $[s_1]_\beta$ to convert ϵ_2 to ϵ_3 and μ_2 to μ_3. Also, the changes made by cond-send to convert ϵ_0 to ϵ_1 and μ_0 to μ_1 are the same changes as those made by cond-send to convert ϵ to ϵ_2 and μ to μ_2. Therefore, $\epsilon_1 = \epsilon_3$ and $\mu_1 = \mu_3$.

Lemma. *For all $\epsilon, \mu, P, \alpha, s, b, y = \textit{cond-receive}(x, t)$ is a left mover in $(\epsilon, \mu, P||[b, y = \textit{cond-receive}(x, t); s]_\alpha)$ if the subchannel $\mu(\epsilon(x), \alpha, t)$ is not empty.*

Proof: The proof is by definition of left movers. Only a statement in process α can pop from the message queue $\mu(\epsilon(x), \alpha, t)$. Running statements from other processes does not affect the message currently at the head of this queue. Therefore cond-receive is enabled even if P is run first. This satisfies the first condition in the definition of left movers.

Suppose $(\epsilon, \mu, P||[b, y = \text{cond-receive}(x, t); s]_\alpha) \xrightarrow{\beta} (\epsilon_0, \mu_0, P_0||[b, y = \text{cond-receive}(x, t); s]_\alpha) \xrightarrow{\alpha} (\epsilon_1, \mu_1, P_0||[s]_\alpha)$ and $[s_1]_\beta$ is the first statement in β. Statement $[s_1]_\beta$ is enabled at the start. Since cond-receive can only affect variables local to α and can only pop from message queues where the destination is α, $[s_1]_\beta$ cannot be disabled if cond-receive is run first instead.

Let $(\epsilon, \mu, P||[b, y = \text{cond-receive}(x, t); s]_\alpha) \xrightarrow{\alpha} (\epsilon_2, \mu_2, P||[s]_\alpha) \xrightarrow{\beta} (\epsilon_3, \mu_3, P_0||[s]_\alpha)$. Now we need to prove that $\epsilon_1 = \epsilon_3$ and $\mu_1 = \mu_3$.

Statement $[s_1]_\beta$ can only access and modify variables that are not local to α. cond-receive may only modify variables local to α. $[s_1]_\beta$ may push messages into message queues whose source is not α and may pop messages from queues whose destination is not α. cond-receive may only pop messages from queues whose destination is α. In short, the actions performed by $[s_1]_\beta$ and cond-receive do not interfere with each other. Therefore, the changes made by $[s_1]_\beta$ to convert ϵ to ϵ_0 and μ to μ_0 are the same changes as those made by $[s_1]_\beta$ to convert ϵ_2 to ϵ_3 and μ_2 to μ_3. Also, the changes made by cond-receive to convert ϵ_0 to ϵ_1 and μ_0 to μ_1 are the same changes as those made by cond-receive to convert ϵ to ϵ_2 and μ to μ_2. Therefore, $\epsilon_1 = \epsilon_3$ and $\mu_1 = \mu_3$.

Lemma. *If s_1 is a left mover in $(\epsilon, \mu, P||[s_1; s]_\alpha)$ then $(\epsilon, \mu, P||[s_1; s]_\alpha) \sqsubseteq (\epsilon, \mu, s_1; P||[s]_\alpha)$*

Proof: The proof analogous to the proof of the same in [1].

5.3 Rewrite Rule Soundness

Lemma 1. *If $\Gamma, \Delta, P \rightsquigarrow \Gamma', \Delta; \Delta', P'$ then $\Gamma, \Delta, P \sqsubseteq \Gamma', \Delta; \Delta', P'$*

Proof: The proof is by induction on the derivation of $\Gamma, \Delta, P \rightsquigarrow \Gamma', \Delta; \Delta', P'$. Each rewrite rule has a separate case. The proof for all rewrite rules except R-Cond-Send and R-Cond-Receive is analogous to the proof of the same in [1]. The remaining proof is given here.

Case R-Cond-Send:

Let $(\epsilon, \mu) \in [\![\Delta, \Gamma]\!]_\varnothing$ and assume

$$(\epsilon, \mu, [\text{cond-send}(b, x, t, n)]_\alpha \ltimes P_x) \longrightarrow^* (\epsilon_f, \mu_f, H)$$

since cond-send is a left mover,

$$(\epsilon, \mu, [\text{cond-send}(b, x, t, n)]_\alpha; P_x) \longrightarrow^* (\epsilon_f, \mu_f, H)$$

Suppose $\epsilon(x) = \beta$. By the R-Cond-Send rewrite step, $\Gamma' = \Gamma[(\alpha, \beta, t) \mapsto \Gamma(\alpha, \beta, t) + +(n : b)]$ and $\Delta' = \texttt{skip}$. Suppose $(\epsilon', \mu') \in \llbracket \Delta', \Gamma' \rrbracket_\epsilon$. Then $\epsilon' = \epsilon$ and $\mu' = \mu[(\alpha, \beta, t) \mapsto \mu(\alpha, \beta, t) + +m]$ where m is $\epsilon(n)$ when $\epsilon(b) \neq 0$ or $\varnothing$ when $\epsilon(b) = 0$.

Suppose $\epsilon(b) \neq 0$. Then by semantic rule E-Cond-Send-True,

$$(\epsilon, \mu[(\alpha, \beta, t) \mapsto \mu(\alpha, \beta, t) + +\epsilon(n)], P_x) \longrightarrow^* (\epsilon_f, \mu_f, H)$$

Suppose $\epsilon(b) = 0$. Then by semantic rule E-Cond-Send-False,

$$(\epsilon, \mu[(\alpha, \beta, t) \mapsto \mu(\alpha, \beta, t) + +\varnothing], P_x) \longrightarrow^* (\epsilon_f, \mu_f, H)$$

that is,

$$(\epsilon', \mu', P_x) \longrightarrow^* (\epsilon_f, \mu_f, H)$$

Therefore, $(\epsilon, \mu, [\text{cond-send}(b, x, t, n)]_\alpha \ltimes P_x) \sqsubseteq (\epsilon', \mu', P_x)$.

Case R-Cond-Receive:

Let $(\epsilon, \mu) \in \llbracket \Delta, \Gamma \rrbracket_\varnothing$ and assume

$$(\epsilon, \mu, [b, y = \text{cond-receive}(x, t)]_\beta \ltimes P_x) \longrightarrow^* (\epsilon_f, \mu_f, H)$$

since cond-receive is a left mover,

$$(\epsilon, \mu, [b, y = \text{cond-receive}(x, t)]_\beta; P_x) \longrightarrow^* (\epsilon_f, \mu_f, H)$$

Suppose $\epsilon(x) = \alpha$. By the R-Cond-Receive rewrite step, $\Gamma' = \Gamma[(\alpha, \beta, t) \mapsto \text{pop}(\Gamma(\alpha, \beta, t))]$ and $\Delta' = [\beta.b = \alpha.b'?\ 1 : 0; \beta.y = \alpha.b'?\ \alpha.n : \beta.y]_\beta$ when $\text{head}(\Gamma(\alpha, \beta, t)) = (n : b')$. Suppose $(\epsilon', \mu') \in \llbracket \Delta', \Gamma' \rrbracket_\epsilon$. Then $\mu' = \mu[(\alpha, \beta, t) \mapsto \text{pop}(\mu(\alpha, \beta, t))]$. Further, either $\epsilon' = \epsilon[\beta.b \mapsto 1][\beta.y \mapsto \alpha.n]$ when $\text{head}(\mu(\alpha, \beta, t)) = \alpha.n$ or $\epsilon' = \epsilon[\beta.b \mapsto 0]$ when $\text{head}(\mu(\alpha, \beta, t)) = \varnothing$.

Suppose $\text{head}(\mu(\alpha, \beta, t)) = \alpha.n$. Then by semantic rule E-Cond-Receive-True,

$$(\epsilon[\beta.b \mapsto 1][\beta.y \mapsto \alpha.n], \mu[(\alpha, \beta, t) \mapsto \text{pop}(\mu(\alpha, \beta, t))], P_x) \longrightarrow^* (\epsilon_f, \mu_f, H)$$

Suppose $\text{head}(\mu(\alpha, \beta, t)) = \varnothing$. Then by semantic rule E-Cond-Receive-False,

$$(\epsilon[\beta.b \mapsto 0], \mu[(\alpha, \beta, t) \mapsto \text{pop}(\mu(\alpha, \beta, t))], P_x) \longrightarrow^* (\epsilon_f, \mu_f, H)$$

that is,

$$(\epsilon', \mu', P_x) \longrightarrow^* (\epsilon_f, \mu_f, H)$$

Therefore, $(\epsilon, \mu, [b, y = \text{cond-receive}(x, t)]_\alpha \ltimes P_x) \sqsubseteq (\epsilon', \mu', P_x)$.

Lemma. *If s_1 is a left mover in $(\epsilon, \mu, P||[s_1; s]_\alpha)$ then $(\epsilon, \mu, P||[s_1; s]_\alpha) \sqsupseteq (\epsilon, \mu, s_1; P||[s]_\alpha)$*

Proof: Suppose $(\epsilon, \mu, s_1; P||[s]_\alpha) \longrightarrow^* (\epsilon', \mu', P')$. We need to show that there exists (ϵ'', μ'', P'') such that $(\epsilon, \mu, P||[s_1; s]_\alpha) \longrightarrow^* (\epsilon'', \mu'', P'')$ and $(\epsilon', \mu', P') \leq (\epsilon'', \mu'', P'')$. The first statement that must be executed from $(s_1; P||[s]_\alpha)$ is s_1. Let $(\epsilon, \mu, s_1; P||[s]_\alpha) \xrightarrow{\alpha} (\epsilon_{s_1}, \mu_{s_1}, P||[s]_\alpha)$. If s_1 is also the first statement executed from $(P||[s_1; s]_\alpha)$, then we get $(\epsilon, \mu, P||[s_1; s]_\alpha) \xrightarrow{\alpha} (\epsilon_{s_1}, \mu_{s_1}, P||[s]_\alpha)$. From this point, both programs have the exact same behavior, hence the lemma is proved.

Lemma 3. *If* $\Gamma, \Delta, P \rightsquigarrow \Gamma', \Delta; \Delta', P'$ *then* $\Gamma, \Delta, P \sqsupseteq \Gamma', \Delta; \Delta', P'$

Proof: Proof is split into multiple cases depending on the rewrite rule.
Case R-Send:
Let $(\epsilon', \mu') \in [\![\Delta; \Delta', \Gamma']\!]_\varnothing$ and assume

$$(\epsilon', \mu', P_x) \longrightarrow^* (\epsilon_f, \mu_f, H)$$

By the R-Send rewrite step, $\Gamma' = \Gamma[(\alpha, \beta, t) \mapsto \Gamma(\alpha, \beta, t) + +n]$ and $\Delta' = \texttt{skip}$. Suppose $(\epsilon, \mu) \in [\![\Delta, \Gamma]\!]_\varnothing$. Then $\epsilon' = \epsilon$ and $\mu' = \mu[(\alpha, \beta, t) \mapsto \mu(\alpha, \beta, t) + +n]$. Therefore,

$$(\epsilon, \mu[(\alpha, \beta, t) \mapsto \mu(\alpha, \beta, t) + +n], P_x) \longrightarrow^* (\epsilon_f, \mu_f, H)$$

by semantic rule E-Send,

$$(\epsilon, \mu, [\text{send}(\beta, t, n)]_\alpha; P_x) \xrightarrow{\alpha} (\epsilon, \mu[(\alpha, \beta, t) \mapsto \mu(\alpha, \beta, t) + +n], P_x)$$

therefore,

$$(\epsilon, \mu, [\text{send}(\beta, t, n)]_\alpha; P_x) \longrightarrow^* (\epsilon_f, \mu_f, H)$$

since send is a left mover,

$$(\epsilon, \mu, [\text{send}(\beta, t, n)]_\alpha \ltimes P_x) \longrightarrow^* (\epsilon_f, \mu_f, H)$$

Case R-Receive:
Let $(\epsilon', \mu') \in [\![\Delta; \Delta', \Gamma']\!]_\varnothing$ and assume

$$(\epsilon', \mu', P_x) \longrightarrow^* (\epsilon_f, \mu_f, H)$$

By the R-Receive rewrite step, $\Gamma' = \Gamma[(\alpha, \beta, t) \mapsto \text{pop}(\Gamma(\alpha, \beta, t))]$ and $\Delta' = [\beta.y = \alpha.n]_\beta$ when $\text{head}(\Gamma(\alpha, \beta, t)) = n$. Suppose $(\epsilon, \mu) \in [\![\Delta, \Gamma]\!]_\varnothing$. Then $\epsilon' = \epsilon[\beta.y \mapsto \alpha.n]$ and $\mu' = \mu[(\alpha, \beta, t) \mapsto \text{pop}(\mu(\alpha, \beta, t))]$. Therefore,

$$(\epsilon[\beta.y \mapsto \alpha.n], \mu[(\alpha, \beta, t) \mapsto \text{pop}(\mu(\alpha, \beta, t))], P_x) \longrightarrow^* (\epsilon_f, \mu_f, H)$$

by semantic rule E-Receive,

$$(\epsilon, \mu, [\text{receive}(\alpha, t)]_\beta; P_x) \xrightarrow{\beta} (\epsilon[\beta.y \mapsto \alpha.n], \mu[(\alpha, \beta, t) \mapsto \text{pop}(\mu(\alpha, \beta, t))], P_x)$$

therefore,

$$(\epsilon, \mu, [\text{receive}(\alpha, t)]_\beta; P_x) \longrightarrow^* (\epsilon_f, \mu_f, H)$$

since receive is a left mover,

$$(\epsilon, \mu, [\text{receive}(\alpha, t)]_\beta \ltimes P_x) \longrightarrow^* (\epsilon_f, \mu_f, H)$$

Case R-Cond-Send:
Let $(\epsilon', \mu') \in [\![\Delta; \Delta', \Gamma']\!]_\varnothing$ and assume

$$(\epsilon', \mu', P_x) \longrightarrow^* (\epsilon_f, \mu_f, H)$$

By the R-Cond-Send rewrite step, $\Gamma' = \Gamma[(\alpha, \beta, t) \mapsto \Gamma(\alpha, \beta, t) + +(n : b)]$ and $\Delta' = \texttt{skip}$. Suppose $(\epsilon, \mu) \in [\![\Delta, \Gamma]\!]_\varnothing$. Then $\epsilon' = \epsilon$ and $\mu' = \mu[(\alpha, \beta, t) \mapsto \mu(\alpha, \beta, t) + +m]$ where m is $\epsilon(n)$ when $\epsilon'(b) \neq 0$ or $\varnothing$ when $\epsilon'(b) = 0$. Therefore,

$$(\epsilon, \mu[(\alpha, \beta, t) \mapsto \mu(\alpha, \beta, t) + +m], P_x) \longrightarrow^* (\epsilon_f, \mu_f, H)$$

by semantic rule E-Cond-Send-True or E-Cond-Send-False (depending on m),

$$(\epsilon, \mu, [\text{cond-send}(b, \beta, t, n)]_\alpha; P_x) \xrightarrow{\alpha} (\epsilon, \mu[(\alpha, \beta, t) \mapsto \mu(\alpha, \beta, t) + +m], P_x)$$

therefore,

$$(\epsilon, \mu, [\text{cond-send}(b, \beta, t, n)]_\alpha; P_x) \longrightarrow^* (\epsilon_f, \mu_f, H)$$

since cond-send is a left mover,

$$(\epsilon, \mu, [\text{cond-send}(b, \beta, t, n)]_\alpha \ltimes P_x) \longrightarrow^* (\epsilon_f, \mu_f, H)$$

Case R-Cond-Receive:

Let $(\epsilon', \mu') \in [\![\Delta; \Delta', \Gamma']\!]_\varnothing$ and assume

$$(\epsilon', \mu', P_x) \longrightarrow^* (\epsilon_f, \mu_f, H)$$

By the R-Cond-Receive rewrite step, $\Gamma' = \Gamma[(\alpha, \beta, t) \mapsto \text{pop}(\Gamma(\alpha, \beta, t))]$ and $\Delta' = [\beta.b = \alpha.b'?\ 1 : 0; \beta.y = \alpha.b'?\ \alpha.n : \beta.y]_\beta$ when $\text{head}(\Gamma(\alpha, \beta, t)) = (n : b')$. Suppose $(\epsilon, \mu) \in [\![\Delta, \Gamma]\!]_\varnothing$. Then $\mu' = \mu[(\alpha, \beta, t) \mapsto \text{pop}(\mu(\alpha, \beta, t))]$. Further, either $\epsilon' = \epsilon[\beta.b \mapsto 1][\beta.y \mapsto \alpha.n]$ when $\text{head}(\mu(\alpha, \beta, t)) = \alpha.n$ or $\epsilon' = \epsilon[\beta.b \mapsto 0]$ when $\text{head}(\mu(\alpha, \beta, t)) = \varnothing$.

Suppose $\text{head}(\mu(\alpha, \beta, t)) = \alpha.n$. Then,

$$(\epsilon[\beta.b \mapsto 1][\beta.y \mapsto \alpha.n], \mu[(\alpha, \beta, t) \mapsto \text{pop}(\mu(\alpha, \beta, t))], P_x) \longrightarrow^* (\epsilon_f, \mu_f, H)$$

Suppose $\text{head}(\mu(\alpha, \beta, t)) = \varnothing$. Then,

$$(\epsilon[\beta.b \mapsto 0], \mu[(\alpha, \beta, t) \mapsto \text{pop}(\mu(\alpha, \beta, t))], P_x) \longrightarrow^* (\epsilon_f, \mu_f, H)$$

by semantic rule E-Cond-Receive-True or E-Cond-Receive-False,

$$(\epsilon, \mu, [b, y = \text{cond-receive}(\alpha, t)]_\beta; P_x) \xrightarrow{\beta} (\epsilon[\beta.b \mapsto 1][\beta.y \mapsto \alpha.n], \mu[(\alpha, \beta, t) \mapsto \text{pop}(\mu(\alpha, \beta, t))], P_x)$$

therefore,

$$(\epsilon, \mu, [b, y = \text{cond-receive}(\alpha, t)]_\beta; P_x) \longrightarrow^* (\epsilon_f, \mu_f, H)$$

since cond-receive is a left mover,

$$(\epsilon, \mu, [b, y = \text{cond-receive}(\alpha, t)]_\beta \ltimes P_x) \longrightarrow^* (\epsilon_f, \mu_f, H)$$

Case R-Context:

Proof is by application of the inductive hypothesis over rewrite rules.

Case R-Congruence:

By definition, $A \equiv B$ if and only if the set of program traces in A is equivalent to the program traces in B.

Case R-If-Then and R-If-Else

Since we do not allow communication inside conditionals, the rewritten program is congruent to the original program.

5.4 Equivalence Lemma

Lemma 4. *Suppose* $\varnothing, \varnothing, P \rightsquigarrow \varnothing, \Delta, \texttt{skip}$*. Then* $(\varnothing, \varnothing, P) \longrightarrow^* (\epsilon, \varnothing, P_0)$ *if and only if* $(\varnothing, \varnothing, \Delta) \longrightarrow^* (\epsilon', \varnothing, P_0')$ *such that* $\epsilon|_H = \epsilon'|_H$ *where* $H = \mathit{halted}(\epsilon, \varnothing, P_0)$*.*

Proof: By applying Lemma 1, we know that $\forall(\epsilon_0, \mu_0) \in [\![\varnothing, \varnothing]\!]_\varnothing, \exists(\epsilon_1, \mu_1) \in [\![\Delta, \varnothing]\!]_\varnothing$ such that whenever $(\epsilon_0, \mu_0, P) \longrightarrow^* (\epsilon, \mu_f, P_0)$ then there exists $(\epsilon', \mu_f', P_0')$ such that $(\epsilon_1, \mu_1, \texttt{skip}) \longrightarrow^* (\epsilon', \mu_f', P_0')$ and $\epsilon|_H = \epsilon'|_H$ where $H = \text{halted}(\epsilon, \mu_f, P_0)$. By unifying variables, we get that if $(\varnothing, \varnothing, P) \longrightarrow^* (\epsilon, \varnothing, \texttt{skip})$ then $(\varnothing, \varnothing, \Delta) \longrightarrow^* (\epsilon, \varnothing, \texttt{skip})$.

Similarly, By applying Lemma 3, we know that the inverse holds true.

APPENDIX D

6 VERIFYING SAFETY AND ACCURACY OF TRANSFORMATIONS

6.1 Common Safety Properties

Transformed programs generated using the transformations in this section retain the following safety properties of the original programs:

Sequentializability. If the original program can be sequentialized, then the transformed program can also be sequentialized. As a result, if the original program is deadlock free, then the transformed program is also deadlock free. This is because the transformations do not remove the sends and receives from the programs, nor do they place the sends and receives inside a conditional statement. Further, when a send is converted to a cond-send, the corresponding receive is always converted to a cond-receive.

Type Safety. If the original program is type safe, then the transformed program is also type safe. In particular, if approximate variables do not affect the values of exact variables in the original program, then the same applies for the transformed program. This is because we only apply these transformations, which introduce approximation, when all the variables affected by the transformation are approximate.

6.2 Precision Reduction

This transformation reduces the precision of approximate data being transferred between processes in order to reduce transmission time and energy usage. The type of the data must be changed in both the sending and receiving processes to the same less precise type.

$$\left[\begin{array}{l}\texttt{precise t1 n;}\\ \texttt{send(}\beta\texttt{, precise t1, n);}\end{array}\right]_\alpha \parallel \left[\begin{array}{l}\texttt{precise t1 x;}\\ \texttt{x = receive(}\alpha\texttt{, precise t1);}\end{array}\right]_\beta$$

$$\Downarrow$$

$$\left[\begin{array}{l}\texttt{approx t1 n;}\\ \texttt{approx t2 n' = (approx t2) n;}\\ \texttt{send(}\beta\texttt{, approx t2, n');}\end{array}\right]_\alpha \parallel \left[\begin{array}{l}\texttt{approx t1 x;}\\ \texttt{approx t2 x';}\\ \texttt{x' = receive(}\alpha\texttt{, approx t2);}\\ \texttt{x = (approx t1) x';}\end{array}\right]_\beta$$

$$\left[\begin{array}{l}\texttt{precise t1 }\beta\texttt{.x, }\alpha\texttt{.n;}\\ \beta\texttt{.x = }\alpha\texttt{.n;}\end{array}\right]_{seq}$$

$$\Downarrow$$

$$\left[\begin{array}{l}\texttt{approx t1 }\beta\texttt{.x, }\alpha\texttt{.n;}\\ \texttt{approx t2 }\beta\texttt{.x', }\alpha\texttt{.n';}\\ \alpha\texttt{.n' = (approx t2) }\alpha\texttt{.n;}\\ \beta\texttt{.x' = }\alpha\texttt{.n';}\\ \beta\texttt{.x = (approx t1) }\beta\texttt{.x';}\end{array}\right]_{seq}$$

To use this transformation, it should be possible to convert the original data type t_1 to a less precise data type t_2 and back, such as converting doubles to floats or 32 bit integers to 16 bit integers. Further, there must not be already messages of type t_2 being sent from p to q, else the converted code may affect the order of the messages.

6.3 Data Transfers over Noisy Channels

This transformation simulates the transfer of approximate data over an unreliable channel. The channel may corrupt the data being sent over it with probability r and the receiver may receive a garbage value. We simulate this by choosing to corrupt the data being sent at the sender with probability r. If corrupted, the value being sent is replaced with a randomly chosen value.

```
      [ precise t n;               ]     ||  [ precise t x;                 ]
      [ send(β, precise t, n);     ]α        [ x = receive(α, precise t);   ]β
                                         ⇓
[ approx t n;                      ]     ||  [ approx t x;                  ]
[ n = n [r] randVal(approx t);     ]         [ x = receive(α, approx t);    ]β
[ send(β, approx t, n);            ]α
```

```
            [ precise t β.x, α.n;              ]
            [ β.x = α.n;                       ]seq
                          ⇓
        [ approx t β.x, α.n;                   ]
        [ α.n = α.n [r] randVal(approx t);     ]
        [ β.x = α.n;                           ]seq
```

Precise data must be sent over a perfectly reliable channel to avoid corruption.

6.4 Failing Tasks

This transformation simulates the execution of tasks that can fail with some probability r due to unreliable hardware. We simulate this by converting the send of the approximate result to a cond-send and the corresponding receive to a cond-receive. The condition of cond-send is 1 with probability $1 - r$ and 0 with probability r.

```
      [ precise t n;                 ]     ||  [ precise t x;                        ]
      [ send(β, precise t, n);       ]α        [ x = receive(α, precise t);          ]β
                                           ⇓
[ approx t n;                        ]     ||  [ approx t x;                         ]
[ approx int b = 1 [r] 0;            ]         [ approx int b;                       ]
[ cond-send(b, β, approx t, n);      ]α        [ b, x = cond-receive(α, approx t);   ]β
```

```
            [ precise t β.x, α.n;        ]
            [ β.x = α.n;                 ]seq
                          ⇓
            [ approx t β.x, α.n;         ]
            [ approx int α.b, β.b;       ]
            [ α.b = 1 [r] 0;             ]
            [ β.b = α.b ? 1 : 0;         ]
            [ β.x = α.b ? α.n : β.x;     ]seq
```

6.5 Approximate Map

This transformation uses approximate memoization [2] to reduce the number of tasks sent to worker threads. This results in decreased communication and improves energy efficiency. If the master thread decides not to send a task to a worker thread, then that worker thread will return an empty result. Upon receiving an empty result, the master thread uses the most recently received result in its place.

```
precise t[] work[size(Q)];
precise t'[] results[size(Q)];
precise t' y;
precise int index = 0;
for(β:Q){
  send(β, precise t, work[index]);
  index = index + 1;
};
index = 0;
for(β:Q){
  y = receive(β, precise t');
  results[index] = y;
  index = index + 1;
};
```

$$]_\alpha \quad \| \quad \Pi.\beta : Q$$

```
precise t x;
precise t' y;
x = receive(α, precise t);
y = dowork(x);
send(α, approx t', y);
```

$$]_\beta$$

$$\Downarrow$$

```
approx t[] work[size(Q)];
approx t'[] results[size(Q)];
approx t' y;
approx int b, c;
precise int index = 0;
for(β:Q){
  b = 1 [r] 0;
  cond-send(b, β, approx t, work[index]);
  index = index + 1;
};
index = 0;
for(β:Q){
  c, y = cond-receive(β, approx t');
  results[index] = y;
  index = index + 1;
};
```

$$]_\alpha \quad \| \quad \Pi.\beta : Q$$

```
approx t x;
approx t' y;
approx int b;
b, x = cond-receive(α, approx t);
y = b ? dowork(x) : 0;
cond-send(b, α, approx t', y);
```

$$]_\beta$$

```
precise t[] α.work[size(Q)];
precise t'[] α.results[size(Q)];
precise t β.x;
precise t' α.y,β.y;
precise int α.index = 0;
for(β:Q){
  β.x = work[α.index];
  β.y = dowork(β.x);
  α.index = α.index + 1;
};
α.index = 0;
for(β:Q){
  α.y = β.y;
  α.results[α.index] = α.y;
  α.index = α.index + 1;
};
```

$$]_{seq} \quad \Rightarrow$$

```
approx t[] α.work[size(Q)];
approx t'[] α.results[size(Q)];
approx t β.x;
approx t' α.y,β.y;
approx int α.b,α.c,β.b;
precise int α.index = 0;
for(β:Q){
  α.b = 1 [r] 0;
  β.b = α.b ? 1 : 0;
  β.x = α.b ? α.work[α.index] : β.x;
  β.y = β.b ? dowork(β.x) : β.y;
  α.index = α.index + 1;
};
α.index = 0;
for(β:Q){
  α.c = β.b ? 1 : 0;
  α.y = β.b ? β.y : α.y;
  α.results[α.index] = α.y;
  α.index = α.index + 1;
};
```

$$]_{seq}$$

This transformation requires the program to match a pattern where a single thread (the master) sends messages to multiple symmetric threads (the workers) and then gathers results from all the workers.

6.6 Approximate Reduce

This transformation approximates an aggregation operation such as finding the minimum, maximum, or sum of multiple elements. The master thread does not send all the work items to the worker threads. The worker threads that do not receive work reply with an empty result. This empty result is not aggregated. At the end, the master threads adjusts the aggregate based on the number of workers that responded with a result.

```
[precise int s = 0, y;             ]
[for (β:Q){                        ]
[  y = receive(β, precise int);    ]      ||  Π.β : Q [precise int y = dowork();]
[  s = s + y;                      ]                  [send(α, precise int, y)  ]β
[};                                ]
[s = s / size(Q)                   ]α
                                          ⇓
[approx int s = 0, y, c, ctr = 0, skip;]
[for (β:Q){                            ]
[  c, y = cond-receive(β, approx int); ]          [approx int y,b                 ]
[  s = s + (c ? y : 0);                ]  ||  Π.β : Q [b = 1 [r] 0;           ]
[  ctr = ctr + (c ? 1 : 0);            ]          [y = b ? dowork() : 0;          ]
[};                                    ]          [cond-send(b, α, approx int, y) ]β
[skip = ctr > 0;                       ]
[s = skip ? (s / ctr) : 0              ]α
```

```
[precise int α.y, β.y, α.s=0;]
[for(β:Q){                   ]
[  β.y = dowork();           ]
[};                          ]
[for(β:Q){                   ]
[  α.y = β.y;                ]
[  α.s = α.s + α.y;          ]
[};                          ]
[α.s = α.s / size(Q);        ]seq
              ⇓
[approx int α.y, α.c, β.y, β.b,        ]
[          α.ctr=0, α.s=0, α.skip;     ]
[for(β:Q){                             ]
[  β.b = 1 [r] 0;                      ]
[  β.y = β.b ? dowork() : 0;           ]
[};                                    ]
[for(q:Q){                             ]
[  α.c = β.b ? 1 : 0;                  ]
[  α.y = β.b ? β.y : α.y;              ]
[  α.s = α.s + (α.c ? α.y : 0);        ]
[  α.ctr = α.ctr + (α.c ? 1 : 0);      ]
[};                                    ]
[α.skip = α.ctr > 0;                   ]
[α.s = α.skip ? (α.s / ctr) : 0;       ]seq
```

This transformation requires the program to match a pattern where a single thread (the master) sends messages to multiple symmetric threads (the workers) and then gathers results from all the workers. The result from the workers must be aggregated via an operation such as sum, min, or max. Care must be taken when performing the aggregate adjustment, as the number of workers that respond with a usable result may be zero.

In the example, it is necessary to add a check to ensure that the *ctr* variable is nonzero when adjusting the aggregate, otherwise a divide by zero error can occur. By performing this check, this transformation does not introduce the possibility of a new divide by zero error.

6.7 Skipping Negligible Updates

This transformation drops packets if the value being sent is a scalar below a certain threshold. The receiver must be adding the received value to a sum variable. This transformation saves energy by not sending small updates to the receiver, which will cause an insignificant change in the sum.

```
precise int y,s=0;
for(β:Q){
  y = receive(β, precise int);
  s = s + y;
};
```

$]_{\alpha} \parallel \Pi.\beta : Q$

```
precise int y;
y = dowork();
send(α, precise int, y);
```

$]_{\beta}$

$\Downarrow$

```
approx int y,s=0,b;
for(β:Q){
  b, y = cond-receive(β, approx int);
  s = s + (b ? y : 0);
};
```

$]_{\alpha} \parallel \Pi.\beta : Q$

```
approx int y,b;
y = dowork();
b = (y >= threshold);
cond-send(b, α, precise int, y);
```

$]_{\beta}$

```
precise int α.y,β.y,α.s=0;
for(β:Q){
  β.y = dowork();
};
for(β:Q){
  α.y = β.y;
  α.s = α.s + α.y;
};
```

$]_{seq}$

$\Downarrow$

```
approx int α.y,β.y,α.s=0,α.b,β.b;
for(β:Q){
  β.y = dowork();
};
for(β:Q){
  β.b = (β.y >= threshold);
  α.b = β.b ? 1 : 0;
  α.y = β.b ? β.y : α.y;
  α.s = α.s + (α.b ? α.y : 0);
};
```

$]_{seq}$

This transformation requires that the received value is used to update some other variable via addition. The result message type must be scalar to allow thresholding.

6.8 Scatter-Gather

The scatter-gather pattern is similar to the map pattern. However, instead of sending a worker one work item and receiving one result, the worker is sent an entire array. The worker may randomly access parts of the array and returns multiple results. In the code below, for compactness, we also use the task id β as an index variable.

```
precise t[] data[N];
precise t'[] results[size(Q)*2];
for(β:Q){
  send(β, precise t[], data);
  send(β, precise int, slice(β, N));
  send(β, precise int, slice(β+1, N));
};
for(β:Q){
  results[β*2] = receive(β, precise t');
  results[β*2+1] = receive(β, precise t');
};
```

$]_{\alpha} \parallel \Pi.\beta : Q$

```
precise t[] data[N];
precise int start,end;
precise t' result;
data = receive(α, precise t[]);
start = receive(α, precise int);
end = receive(α, precise int);
result = job1(data, start, end);
send(α, precise t', result);
result = job2(data, start, end);
send(α, precise t', result);
```

$]_{\beta}$

$\Downarrow$

```
approx t[] data[N];
approx t'[] results[size(Q)*2];
approx int fail;
for(β:Q){
  send(β, approx t[], data);
  send(β, approx int, slice(β, N));
  send(β, approx int, slice(β+1, N));
};
for(β:Q){
  fail, results[β*2] = cond-receive(β, approx t');
  fail, results[β*2+1] = cond-receive(β, approx t');
};
```

$]_{\alpha} \parallel \Pi.\beta : Q$

```
approx t[] data[N];
approx int start,end;
approx t' result;
approx int fail;
data = receive(α, approx t[]);
start = receive(α, approx int);
end = receive(α, approx int);
fail = 1 [r] 0;
result = fail ? job1(data, start, end) : result;
cond-send(fail, α, approx t', result);
result = fail ? job2(data, start, end) : result;
cond-send(fail, α, approx t', result);
```

$]_{\beta}$

```
precise t[] α.data[N],β.data[N];
precise t'[] α.results[size(β)*2];
precise int β.start,β.end;
precise t' β.result;
for(β:Q){
  β.data = α.data;
  β.start = slice(β,N);
  β.end = slice(β+1,N);
};
for(β:Q){
  β.result = job1(β.data,β.start,β.end);
  α.results[β*2] = β.result;
  β.result = job2(β.data,β.start,β.end);
  α.results[β*2+1] = β.result;
};
```

$_{seq}$

$$\Downarrow$$

```
approx t[] α.data[N],β.data[N];
approx t'[] α.results[size(β)*2];
approx int β.start,β.end;
approx t' β.result;
approx int α.fail,β.fail;
for(β:Q){
  β.data = α.data;
  β.start = slice(β,N);
  β.end = slice(β+1,N);
};
for(β:Q){
  β.fail = 1 [r] 0;
  β.result = β.fail ? job1(β.data,β.start,β.end) : β.result;
  α.fail = β.fail ? 1 : 0;
  α.results[β*2] = β.fail ? β.result : α.results[β*2];
  β.result = β.fail ? job2(β.data,β.start,β.end) : β.result;
  α.fail = β.fail ? 1 : 0;
  α.results[β*2+1] = β.fail ? β.result : α.results[β*2+1];
};
```

$_{seq}$

Several previous transformations, such as precision reduction, approximate map, failing tasks, etc. can also be applied to this pattern.

6.9 Scan

The scan pattern takes an input array and generates an output array. The n^{th} element of the output depends on the first n elements of the input and is calculated by an associative function (such as summation, average, etc.) In the code below, for compactness, we also use the task id β as an index variable.

```
precise int[] input[N];
precise int[] output[N];
precise int index;
index = 0;
for(β:Q){
  send(β, precise int, index);
  send(β, precise int[], input);
  index = index+1;
};
for(β:Q){
  output[β] = receive(β, precise int);
};
```

α

$\parallel \quad \Pi.\beta : Q$

```
precise int[] input[N];
precise int output;
precise int index, i;
index = receive(α, precise int);
input = receive(α, precise int[]);
output = 0;
i = 0;
repeat N{
  if(index <= i){
    output = output + input[i];
  };
  i = i + 1;
};
send(α, precise int, output);
```

β

$\Downarrow$

```
approx int[] input[N];
approx int[] output[N];
precise int index;
index = 0;
for(β:Q){
  send(β, precise int, index);
  send(β, approx int[], input);
  index = index+1;
};
for(β:Q){
  output[β] = receive(β, approx int);
};
```

α

$\parallel \quad \Pi.\beta : Q$

```
approx int[] input[N];
approx int output;
precise int index, i;
index = receive(α, precise int);
input = receive(α, approx int[]);
output = 0;
i = 0;
repeat N{
  if(index <= i){
    output = output + input[i];
  };
  i = i + 1;
};
//simulate noisy channel
output = output [r] randInt();
send(α, approx int, output);
```

β

```
precise int[] α.input[N];
precise int[] α.output[N];
precise int[] β.input[N];
precise int α.index,β.output,β.index,β.i;
α.index = 0;
for(β:Q){
  β.index = α.index;
  β.input = α.input;
  α.index = α.index+1;
  β.output = 0;
  β.i = 0;
  repeat N{
    if(β.index <= β.i){
      β.output = β.output + β.input[β.i];
    };
    β.i = β.i + 1;
  };
};
for(β:Q){
  α.output[β] = β.output;
};
```

seq

$\Rightarrow$

```
approx int[] α.input[N];
approx int[] α.output[N];
approx int[] β.input[N];
precise int α.index,β.index,β.i;
approx int β.output;
α.index = 0;
for(β:Q){
  β.index = α.index;
  β.input = α.input;
  α.index = α.index+1;
  β.output = 0;
  β.i = 0;
  repeat N{
    if(β.index <= β.i){
      β.output = β.output + β.input[β.i];
    };
    β.i = β.i + 1;
  };
};
for(β:Q){
  //simulate noisy channel
  β.output = β.output [r] randInt();
  α.output[β] = β.output;
};
```

seq

For this pattern we simulate a noisy channel. Other approximations can also be applied.

6.10 Stencil

The stencil pattern calculates each element of the output array by applying some function to the corresponding element in the input array along with its neighbors. It is used in many image-processing and scientific applications. In the code below, for compactness, we also use the task id β as an index variable.

```
precise float64[] input[N];
precise float64[] output[N];
precise int index;
index = 0;
for(β:Q){
  send(β, precise int, index);
  send(β, precise float64[], input);
  index = index+1;
};
for(β:Q){
  output[β] = receive(β, precise float64);
};
```

$_{\alpha}$

$\parallel \quad \Pi.\beta : Q$

```
precise float64[] input[N];
precise float64 output;
precise int index;
index = receive(α, precise int);
input = receive(α, precise float64[]);
output = (input[index-1]+input[index]+input[index+1])/3;
send(α, precise float64, output);
```

$_{\beta}$

$\Downarrow$

```
approx float64[] input[N];
approx float32[] input32[N];
approx float64[] output[N];
approx float32 output32;
precise int index;
index = 0;
input32 = (approx float32[])input;
for(β:Q){
  send(β, precise int, index);
  send(β, approx float32[], input32);
  index = index+1;
};
for(β:Q){
  output32 = receive(β, approx float32);
  output[β] = (approx float64)output32;
};
```

$_{\alpha}$

$\parallel \quad \Pi.\beta : Q$

```
approx float32[] input[N];
approx float32 output;
precise int index;
index = receive(α, precise int);
input = receive(α, approx float32[]);
output = (input[index-1]+input[index]+input[index+1])/3;
send(α, approx float32, output);
```

$_{\beta}$

```
precise float64[] α.input[N];
precise float64[] α.output[N];
precise float64[] β.input[N];
precise float64 β.output;
precise int α.index,β.index;
α.index = 0;
for(β:Q){
  β.index = α.index;
  β.input = α.input;
  α.index = α.index+1;
  β.output = (β.input[β.index-1]+β.input[β.index]+β.input[β.index+1])/3;
};
for(β:Q){
  α.output[β] = β.output;
};
```

$_{seq}$

$\Downarrow$

```
approx float64[] α.input[N];
approx float32[] α.input32[N];
approx float64[] α.output[N];
approx float32[] β.input[N];
approx float32 α.output32,β.output;
precise int α.index,β.index;
α.input32 = (approx float32[])α.input;
α.index = 0;
for(β:Q){
  β.index = α.index;
  β.input = α.input32;
  α.index = α.index+1;
  β.output = (β.input[β.index-1]+β.input[β.index]+β.input[β.index+1])/3;
};
for(β:Q){
  α.output32 = β.output;
  α.output[β] = (approx float64)α.output32;
};
```

$_{seq}$

This code simulates precision reduction.

6.11 Partition

This pattern is similar to the stencil pattern, but the calculations are performed on disjoint partitions of the input array to obtain the output array. In the code below, for compactness, we also use the task id β as an index variable.

```
precise float64[] input[N];
precise float64[] output[N];
precise int index;
index = 0;
for(β:Q){
  send(β, precise int, index);
  send(β, precise float64[], input);
  index = index+2;
};
for(β:Q){
  output[β] = receive(β, precise float64);
};
```
α

$\parallel \quad \Pi.\beta : Q$

```
precise float64[] input[N];
precise float64 output;
precise int index;
index = receive(α, precise int);
input = receive(α, precise float64[]);
output = (input[index]+input[index+1])/2;
send(α, precise float64, output);
```
β

$\Downarrow$

```
approx float64[] input[N];
approx float64[] output[N];
precise int index;
approx int fail;
index = 0;
for(β:Q){
  send(β, precise int, index);
  send(β, approx float64[], input);
  index = index+2;
};
for(β:Q){
  fail, output[β] = cond-receive(β, approx float64);
};
```
α

$\parallel \quad \Pi.\beta : Q$

```
approx float64[] input[N];
approx float64 output;
precise int index;
approx int fail;
index = receive(α, precise int);
input = receive(α, approx float64[]);
output = (input[index]+input[index+1])/2;
//simulate failing tasks
fail = 1 [0.99] 0;
cond-send(fail, α, approx float64, output);
```
β

```
precise float64[] α.input[N];
precise float64[] α.output[N];
precise float64[] β.input[N];
precise int α.index,β.index;
precise float64 β.output;
α.index = 0;
for(β:Q){
  β.index = α.index;
  β.input = α.input;
  α.index = α.index+2;
  β.output = (β.input[β.index]+β.input[β.index+1])/2;
};
for(β:Q){
  α.output[β] = β.output;
};
```
seq

$\Rightarrow$

```
approx float64[] α.input[N];
approx float64[] α.output[N];
approx float64[] β.input[N];
precise int α.index,β.index;
approx float64 β.output;
approx int α.fail,β.fail;
α.index = 0;
for(β:Q){
  β.index = α.index;
  β.input = α.input;
  α.index = α.index+2;
  β.output = (β.input[β.index]+β.input[β.index+1])/2;
};
for(β:Q){
  //simulate failing tasks
  β.fail = 1 [0.99] 0;
  α.fail = β.fail ? 1 ; 0;
  α.output[β] = β.fail ? β.output : α.output[β];
};
```
seq

This code simulates task failing

APPENDIX E

7 EVALUATION

We evaluated the benefits of some approximations by cross-compiling our programs from Parallely to Go language. How each approximation was simulated is described in the paper. Table 1 shows the parameters used in each of the benchmark's approximation (Column 2), number of processes we used (Column 3), and the size of the inputs (Column 4).

Table 1. Experimental Setup for Evaluation

Benchmark	Approximation	# procs	Input
PageRank	Failing Tasks with 1×10^{-6} probability	16	10 Iterations, randomly generated graph with 1000 nodes
Scale	Failing Tasks with 0.0001 probability	16	512 × 512 pixel image (baboon.ppm)
SOR	Precision Reduction (Float64 to Float32)	10	10 iteration on a 1000 × 1000 array
Sobel	Precision Reduction (Float64 to Float32)	10	1000 × 1000 array in the range [0,1]
Motion	Approximate Reduce (Skipping 90% of tasks)	10	10 blocks with 1600 pixels each

For each benchmark we collected the results over 100 runs and present the averaged performance and accuracy numbers. For each benchmark, we verified the accuracy or reliability property specified in Table 1 of the paper.

Table 2. Generated Constraints from reliability/accuracy analysis

Benchmark	Analysis	Calculated bound
PageRank	Reliability	$0.99 \geq \mathcal{R}(\text{pagerank})$
Scale	Reliability	$0.99 \geq \mathcal{R}(\text{output})$
SOR	Accuracy	$2^{-18} \geq \mathcal{D}(\text{result})$
Sobel	Accuracy	$2^{-15} \geq \mathcal{D}(\text{result})$

Table 2 presents the final outcome of the reliability/accuracy analysis for the benchmarks we evaluated for performance. Column 3 shows the final constraint we calculated for each benchmark. These bounds are more tighter than the required bounds from the specification.

We omitted the accuracy analysis for *Motion* since it returns an index and the accuracy requirements cannot be specified using our specification language. Therefore, we only verified type safety and deadlock-freeness.

REFERENCES

[1] Alexander Goldberg Bakst. *Sequentialization and Synchronization for Distributed Programs*. PhD thesis, UC San Diego, 2017.

[2] Mehrzad Samadi, Davoud Anoushe Jamshidi, Janghaeng Lee, and Scott Mahlke. Paraprox: Pattern-based approximation for data parallel applications. In *ASPLOS*, 2014.

[3] Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. Enerj: Approximate data types for safe and general low-power computation. In *PLDI*, 2011.