# Distributed Systems As Programs With Placement Types

Luna Phipps-Costin, Arjun Guha

ROBLOX Research

## Background

Distributed systems like Roblox experiences are often written as a number of programs running on different nodes. These programs interact by passing messages or interacting with a replicated database. However, systems like these are difficult to program and prone to error. Programs with erroneous message passing might result in deadlock, inconsistent state, or race conditions, and all interaction with a replicated database must be correct with respect to the consistency guarantees of the database.

## Introduction

Programs in our language instead describe an entire (distributed) computation. Programmers use type annotations to describe on which nodes computation and effects are performed. Using these annotations, the implementation generates the message-passing necessary to run the program across multiple nodes, and the type system ensures that the protocol is correct. Further, the type system is amenable to best-effort constraint-based type inference, which we plan to leverage to allow programs to elide type annotations while maintaining performance.

## A Note

This poster represents in-progress research. Results may be invalidated and the language presented has multiple problems yet to be worked out.

## Type Inference

If we wrap all expressions in send, every simply well-typed program has a set of placement type assignments that type-checks. Further, this type inference task is undecidable (if only due to the use of System F). These two facts suggest a best-effort constraint-based type inference using (possibly programmer-provided) heuristics to guide typing.

## Syntax

$$
\begin{aligned}
e ::=\ & \text{true } p \mid \text{false } p \mid \lambda_p x : t.\, e \mid e\ e \mid \text{ref}_p e \mid\ !e \mid e := e \mid \{l = e, ...\}_p \mid e.\, l \\
& \mid \text{send}_p e \mid \Lambda P.\, e \mid e\ p \mid \text{fold}_{\mu\alpha.t} e \mid \text{unfold}_{\mu\alpha.t} e \mid \text{pack } \{p, e\} \text{ as } \exists P.\, t \mid \text{unpack } \{P, x\} = v \text{ in } e
\end{aligned}
$$

$$
\begin{aligned}
u &::= \text{bool} \mid t \to t \mid \mu\alpha.\, u \mid \alpha \mid \text{ref } t \mid \{l_1 : u_1, ...\} \\
p &::= \text{node } n \mid P \\
t &::= p\ u \mid \forall P.\, t \mid \exists P.\, t
\end{aligned}
$$

The standard base of our language is System F plus records, mutation, and recursive types.

## Semantics

$$
\begin{aligned}
&s; \lambda_p x : t.\, e \to_p s; p\lambda x : t.\, e \\
&s; (p\ \lambda x.\, e)\ v \to_p s; e[x \,/\, v] \\
&s; \Lambda P.\, e \to_? s; e \\
&s; \{l = p\ c, ...\}_p \to_p s; p\ \{l = c, ...\} \\
&s; (p\ \{l = c, ...\}).\, l \to_p s; p\ c \\
&s; \text{ref}_q(p\ c) \to_p s, l \mapsto p\ c; \text{send}_q(p\ l)\ (l \text{ fresh}) \\
&s, l \mapsto v; !(p\ l) \to_p s, l \mapsto v; v \\
&s; (p\ l) := v \to_p s, l \mapsto v; p\ l \\
&s; \text{send}_q(p\ c) \xrightarrow{\text{msg}(p,q,c)}_p s; q\ c \\
&s; \text{fold}(p\ c) \to_p s; p\ c \\
&s; \text{unfold}(p\ c) \to_p s; p\ c
\end{aligned}
$$

$$
\frac{s; H[r] \to_q s'; H[r]; p \neq q}{p; s; H[r] \xrightarrow{\text{msg}(p,q,H[r])} q; s; H[r]}
$$

Using the steps, the above describes message passing.

It is obvious from our formalization that each step occurs on exactly one node.

We also intend to prove that a node only ever dereferences locations it allocated.

## Types

$$
\frac{\Delta, p \vdash u \text{ ok}}{\Delta, p \vdash p\ u \text{ ok}} \qquad
\frac{\Delta, P \vdash t \text{ ok}}{\Delta \vdash \forall P.\, t \text{ ok}} \qquad
\frac{\Delta, P \vdash t \text{ ok}}{\Delta \vdash \exists P.\, t \text{ ok}} \qquad
\Delta \vdash \text{bool ok} \qquad
\frac{\Delta \vdash t_1 \text{ ok} \quad \Delta \vdash t_2 \text{ ok}}{\Delta \vdash t_1 \to t_2 \text{ ok}} \qquad
\frac{\Delta \vdash t \text{ ok}}{\Delta \vdash \text{ref } t \text{ ok}}
$$

Valid types

$$
\Gamma \vdash \text{true } p : p \text{ bool}
$$

$$
\frac{\Gamma, x : t \vdash e : s \quad \{p\} = \{p | p\ u \in \text{fv}(e)\}}{\lambda_p x : t.\, e : p\ (t \to s)}
\qquad
\frac{\Gamma \vdash e_1 : q\ (p\ u \to t) \quad \Gamma \vdash e_2 : p\ u}{\Gamma \vdash e_1\ e_2 : t}
$$

$$
\frac{\Gamma \vdash e : p\ u}{\Gamma \vdash \text{ref}_p e : p \text{ ref } u}
\qquad
\frac{\Gamma \vdash e : t}{\Gamma \vdash \text{ref}_p e : p \text{ ref } t}
\qquad
\frac{\Gamma \vdash e : p \text{ ref } t}{\Gamma \vdash !e : t}
\qquad
\frac{\Gamma \vdash e_1 : p \text{ ref } t \quad \Gamma \vdash e_2 : t}{\Gamma \vdash e_1 := e_2 : p \text{ ref } t}
$$

$$
\frac{\Gamma \vdash e : p\ u, ...}{\Gamma \vdash \{l = e, ...\}_p : p\ \{l : u, ...\}}
\qquad
\frac{\Gamma \vdash e : p\ \{l : u, ...\}}{\Gamma \vdash e.\, l : p\ u}
$$

$$
\frac{\Gamma \vdash e : q\ u}{\Gamma \vdash \text{send}_p e : p\ u}
\qquad
\frac{\Delta \cup \{P\}, \Gamma \vdash e : t}{\Delta, \Gamma \vdash \Lambda P.\, e : \forall P.\, t}
\qquad
\frac{\Delta \vdash p \text{ ok} \quad \Delta, \Gamma \vdash e : \forall P.\, t}{\Delta, \Gamma \vdash e\ p : t[P \,/\, p]}
$$

$$
\frac{\Gamma \vdash e : \mu\alpha.\, t}{\Gamma \vdash \text{unfold } e : t[\alpha \,/\, \mu\alpha.\, t]}
\qquad
\frac{\Gamma \vdash e : t[\alpha \,/\, \mu\alpha.\, t]}{\Gamma \vdash \text{fold } e : \mu\alpha.\, t}
$$

$$
\frac{\Delta, \Gamma \vdash e : t[P \,/\, p] \quad \Delta \vdash \exists P.\, t \text{ ok}}{\Delta, \Gamma \vdash \text{pack } \{P, e\} \text{ as } \exists P.\, t : \exists P.\, t}
$$

$$
\frac{\Delta, \Gamma \vdash e_1 : \exists P.\, t_1 \quad \Delta \cup \{P\}, \Gamma, x : t \vdash e_2 : t_2 \quad \Delta \vdash t_2 \text{ ok}}{\Delta, \Gamma \vdash \text{unpack } \{P, x\} = e_1 \text{ in } e_2 : t_2}
$$

We elide type well-formedness checking in most typing rules; they are standard.

We extend the conventional type system by:

- Pairing each type with a placement annotation,
- supporting placement polymorphism à la System F, and
- supporting placement existentials à la ML modules.

Note that records are homogenously placed, that referenced values can never be sent, and that lambda must live where its free variables live.

## Next Steps

Our next task is to implement the dynamic semantics in our prototype. Using our implementation, we will be able to write example programs to gain confidence in the usefulness of the current design. Our first benchmarks will be an arbitrary-node producer-consumer queue and a simple client-server benchmark from the literature. Once we are confident in the explicitly typed language, we will begin work on our type inference system. In the longer term, much further work could be done on the basis of Mycelium, such as support for concurrency, automatic parallelization, and client speculation.

## Related Work

The PLT Scheme Web Server (Krishnamurthi et al 2007) supports sending continuations from the server to the client and vice versa, modeling Web navigation as invoking continuations.

Swift (Chong et al 2007) usse Information-Flow Control (based on JIF) to describe programs, which then are split between client and server using a heuristic based on the min-flow problem.

Pyxis (Cheung et al 2012) allows parts of programs to be offloaded to the database server with no annotations to improve performance. It also uses a program graph to optimize the split.

Links (Hutchison et al 2007) is a language with server and client annotations on functions, allowing programs to be written that span both.

Fission (Guha et al 2017) distributes the program dynamically.

Our language is type-based, distributes on the expression level, and supports functions which capture their environment, while the above work supports at most one of these three features.

Fabric (Liu et al 2009) represents a scalable, multi-node, mutually distrustful programming system for distributed systems. While Fabric emphasizes concurrency and state synchronization, our language prioritizes linear, deterministic protocols proven by the type system.