Thoughts on Tierless Programming

June 6, 2023

Luna Phipps-Costin

1 Related Work

1.1 Tierless Programming

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. This facilitates programming Web applications in a single context by describing all interaction between client and server with this paradigm. It does not compile any code to JavaScript to be run on the client. Subsequent work builds on PLT Web's model of web programming by using and sending continuations.

Chong et al. (2007) seek to make web applications secure by construction. Building off of JIF [3], it uses privacy and integrity annotations along with information flow control typing to annotate programs. After type-checking to enforce security, Swift programs are compiled to another language called WebIL, which I find particularly interesting. WebIL annotates each statement with a placement annotation: one of C, S, Sh, C?Sh, C? S?, CS, CS?, C?S, and CSh. These types respectively annotate where statements must execute, where they may optionally execute, and whether runtime integrity constraints must be enforced at that site. Using a syntactic reduction to the min-flow problem, using weights that assume for-loops run 10 times, WebIL is compiled to "well-placed WebIL," in which each statement is placed on the server, the client, or both only. I think something like this technique (or Pyxis') is absolutely necessary to do static tierless programming without excessive annotation burden. WebIL is constrained by A-Normal Form; for example a.field1.field2 is not valid WebIL; this allows placement annotations to make sense per-statement. Well-placed WebIL is then compiled to continuation-passing style; client-placed programs are compiled to JavaScript, and the runtime communicates by serializing and sending continuations (which is simplified by, ie, the lack of λ in Java-as-of-2007).

Pyxis (Cheung et al. 2012) allows parts of programs to be offloaded to the database server using no annotations with the goal of performance optimization. Pyxis forms a control-flow-and-dataflow graph, similarly to Swift but with more edges. Programs are compiled to PixIL which annotates statements with their presence on the application server or database server (only). This is analogous to well-placed WebIL. Pyxis statically generates different partitions and dynamically switches between them to improve performance; it uses a dynamic profile for its initial partition. Pyxis references a number of other partitioning languages which I'd love to look into: Hilda, Wishbone, Coign, Odessa, MAUI, CloneCloud, Chroma, and Spectra. I want to read this longer paper in more depth. Other interesting things Pyxis does include state caching, CFG performance optimizations, and heterogenous objects (but not heterogenous arrays).

Links (Cooper et al. 2007) described in 2007 is not to be confused with Links-as-of-2023 which now has an emphasis on algrebraic effect handlers. Links (2007) is a functional programming language for tierless programming of Web applications. Each function is annotated with its residence on the client or server. Server functions can also tierlessly access the database. Links supports client-server polymorphism by omitting an annotation. Like (all?) other static splitting languages, Links is compiled by serializing continuations and sending them over the wire.

Ur/Web (Chlipala 2015) follows Links closely, plus a module system and affordances for reactive programming. However, unlike Links, it does not support transfer of λ, and uses explicit RPCs rather than implicit control transfer between nodes.

Fission (Guha et al. 2017) stands out from other languages by tiering the program dynamically. Arjun says this makes the protocol "way too complicated" but I don't fully understand how so (?), given that even the static splits inevitably have to send continuations over the wire due to the halting problem. Somewhat like Termite Scheme, Fission uses faceted values to send secret or unserializable values (such as open files).

TODO: Write about Mogk 2019. Also read the related work.

TODO: Write about Fiber. Also read the related work.

Language	λ	Multi-placement	Implicit control transfer	Granularity
PLT Web	Yes	No	No	Explicit
Swift	No	Yes	Yes	ANF-Statement
Links	Yes	Yes	Yes	Top-level Function
Ur/Web	No	Explicit	No	Function
Pyxis	No?	Yes?	Yes	ANF-Statement
Fission	Yes	Dynamic	Yes	Expression (Dynamically)
Our language	Yes	Yes	Yes	Expression

Table 1: Relevant feature support by various related languages

1.1.1 The secret Databases papers

Don't ask about them. They exist but they are not for your mortal eyes. Apparently there is some stuff about effects in here. Apparently most of this sidesteps turing completeness. Ask Mae Milano about it if you dare.

1.2 Other Related Work

Milano and Myers (2018) allow programmers to encode (transactional) consistency constraints in their programs, and to annotate the consistency guaranteed by their various underlying databases. MixT checks that these contraints are safe (ie low consistency data does not flow into high-consistency) and then provides a runtime to guarantee stronger consistency than the underlying database when necessary.

Miller, Haller, and Odersky (n.d.) provide a type-based model for making guarantees about the free variables of functions. In particular, Spores can constrain functions to be serializable. Termite scheme (Germain 2006) (akin to Erlang-for-scheme), uses an alternative approach which simply builds the language around the guarantee that all values, including λ, are serializable.

1.3 Roblox "Next Generation Programming Model": What The Heck Is It

Overall, this is a highly specific adjustment to Roblox programs. In particular, it very slightly adjusts the way programmers interact with not-present state (not all state is replicated all of the time, apparently), and it proposes a fairly ad-hoc way of running the same program on multiple nodes by using the replicated database.

1.3.1 "Boundaries between instances"

This is quite simple and very specific. Only part of the environment / database (DOM) is replicated to the client at a time. However, Roblox allows you to annotate parts of the DOM tree as "atomic," meaning they are either entirely replicated or entirely absent. Roblox currently allows programmers to attempt to access any state (the DOM is connected, in the graph sense). If the state is not currently replicated, access simply fails. The proposal is to **disallow following pointers from one atomic subtree to another**.

1.3.2 "Proxy instances"

Currently, on a disallowed field access (see above), the return value is `null`. The proposal is to return a promise that can be awaited instead.

1.3.3 "Replicated scripts"

The proposal is to support running the same program on the client and server. (This does not include inferring values, so is more like Swift than true client prediction.) The proposal anticipates a few problems: shared mutable state, unserializable values, and side effects.

Shared mutable state: The replicated database holds all "long-lived" mutable state. There is some discussion of disallowing certain mutation in the respective environments, but it's highly confused and unclear.

Unserializable values: No free variables may reference mutable state, including module imports.

Side effects: Speculated programs are executed in a special side-effect free environment (besides database effects).

1.3.4 Glossary

Terms that are needed to understand any of this document: (All terms refer to existing behaviour unless otherwise noted)

1. **Streaming**: Only part of the DOM is persisted to a client at a time.
2. **Atomic streaming**: You can annotate a node of the DOM with "atomic" in which case it is either all persisted or not.
3. **Navigation**: Field access within the DOM (which holds parent pointers).
4. **Streaming set**: ??? This may refer to a subtree that is always either present or not present.
5. **RunContext**: Annotates a program as running on the client or server.
6. **Data model**: Eventually-consistent replicated in-memory database. I believe the DOM is a subset or equal to this, but I'm not sure.
7. **Callback**: I believe this is more specific than just a function passed to a scheduling environment.
8. **Replica**: ??? This may refer to a subtree which is being simulated by another node.
9. **Authoritative**: Some physics simulation is performed by the client, on a subtree-by-subtree basis. This is referred to as the client having authority over that subtree.

10. **Instance / Part / ???**: These are all different words for different subsets of "subtree."

2 Our approach

Describing Computation Over Multiple Nodes

Or, **Distributed Systems As Programs With Placement Types**

Three-sentence pitch: Computations like Roblox experiences are often written as a number of programs running on multiple nodes, interacting using message-passing or a replicated database. To simplify this and reduce errors, programs in our language instead describe an entire (distributed) computation. Programmers use type annotations to describe on which node each part of the computation is performed, and the system runs the program by sending continuations between nodes.

More-sentence pitch:

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.

2.1 Placement Types

Placement annotations on all the related work I've read so far don't fit into a standard model of types: either they are an annotation on top-level functions, or an annotation on statements, with some restriction on statements (ie ANF). I'd like to include a placement annotation on every single type, thereby placing each expression (modulo polymorphism). This feels more "principled" than annotating statements or functions. It should also enable more traditional or constraint-based type inference techniques to address automatic partitioning.

2.2 Syntax

We base our language on System F, plus records, mutation, and recursive types.

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

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

2.3 Type system

We extend the traditional type system by:

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

We define valid types:

$$\frac{\Delta, p \vdash u \text{ ok}}{\Delta, p \vdash p\ u \text{ ok}}$$

$$\frac{\Delta, P \vdash t \text{ ok}}{\Delta \vdash \forall P.t \text{ ok}}$$

$$\frac{\Delta, P \vdash t \text{ ok}}{\Delta \vdash \exists P.t \text{ ok}}$$

$$\Delta \vdash \text{bool ok}$$

$$\frac{\Delta \vdash t_1 \text{ ok} \quad \Delta \vdash t_2 \text{ ok}}{\Delta \vdash t_1 \to t_2 \text{ ok}}$$

$$\frac{\Delta \vdash t \text{ ok}}{\Delta \vdash \text{ref } t \text{ ok}}$$

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

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

$$\frac{\Gamma, x : t_1 \vdash e : t_2 \quad \{p\} = \{p | p\ u \in \text{fv}(e)\}}{\Gamma \vdash \lambda_p x : t_1.e : p\ (t_1 \to t_2)}$$

$$\frac{\Gamma \vdash e_1 : q\ (p\ u \to t) \quad \Gamma \vdash e_2 : p\ u}{\Gamma \vdash e_1\ e_2 : t}$$

$$\frac{\Gamma \vdash e : t}{\Gamma \vdash \text{ref}_p e : p \text{ ref } t}$$

$$\frac{\Gamma \vdash e : p \text{ ref } t}{\Gamma \vdash !e : t}$$

$$\frac{\Gamma \vdash e_1 : p \text{ ref } t \quad \Gamma \vdash e_2 : t}{\Gamma \vdash e_1 := e_2 : p \text{ ref } t}$$

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

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

$$\frac{\Gamma \vdash e : q\ u}{\Gamma \vdash \text{send}_p e : p\ u}$$

$$\frac{\Delta \cup \{P\}, \Gamma \vdash e : t}{\Delta, \Gamma \vdash \Lambda P.e : \forall P.t}$$

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

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

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

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

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

2.4 Semantics

Some reasoning could be done here to show that the reduction placement annotation respects the type rules, and that labels in s are never accessed by the wrong node.

A single step always steps on exactly one node:

$$\sigma; \lambda_p x : t.e \underset{p}{\to} \sigma; p\lambda x : t.e$$

$$\sigma; (p\ \lambda x.e)\ v \underset{p}{\to} \sigma; e[x/v]$$

$$\sigma; \Lambda P.e \underset{?}{\to} \sigma; e$$

$$\sigma; \{l = p\ c, ...\}_p \underset{p}{\to} \sigma; p\ \{l = c, ...\}$$

$$\sigma; (p\ \{l = c, ...\}).l \underset{p}{\to} \sigma; p\ c$$

$$\sigma; \text{ref}_q(p\ c) \underset{p}{\to} \sigma, l \mapsto p\ c; \text{send}_q(p\ l)\ (l \text{ fresh})$$

$$\sigma, l \mapsto v; !(p\ l) \underset{p}{\to} \sigma, l \mapsto v; v$$

$$\sigma; (p\ l) := v \underset{p}{\to} \sigma, l \mapsto v; p\ l$$

$$\sigma; \text{send}_q(p\ c) \xrightarrow[p]{\text{msg}(p,q,c)} \sigma; q\ c$$

$$\sigma; \text{fold}(p\ c) \underset{p}{\to} \sigma; p\ c$$

$$\sigma; \text{unfold}(p\ c) \underset{p}{\to} \sigma; p\ c$$

Using the steps, we define our placed semantics:

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

I would like to state that:

1. At each step, exactly one node steps
2. At each step, a node reads/writes state only belonging to itself

2.5 Multiple nodes

A key insight is that we rarely want to talk about *nodes themselves* or *all nodes of a certain type*. Instead, we may want to talk about *a list of values* that live in *heterogenous places*. For example, a list of players. Thus it makes sense to model arbitrary numbers of nodes with existential types:

$$
\begin{aligned}
t &::= \ldots \mid \exists P.t \\
e &::= \ldots \mid \text{pack } \{t_1, e\} \text{ as } \exists P.t_2 \\
&\quad\ \mid \text{unpack } \{P, x\} = e_1 \text{ in } e_2
\end{aligned}
$$

Let's assume we also have lists (how?).

This allows us to say things like (modulo proper ADTs):

$$
\begin{gathered}
\text{type player} = \{\text{keyA} : P \text{ bool}, \text{keyB} : P \text{ bool}\} \text{ in} \\
\text{type list} = \forall X.\mu\alpha.\{\text{elem} : X, \text{next} : \alpha\} \\
\text{allPlayers} : \text{server list}[X/\text{server ref } \exists P.P \text{ player}]
\end{gathered}
$$

A system (not the language) can model a player joining as, for example, a callback of type $\exists P.P$ player $\to$ (), or a concurrent stateful update of "allPlayers." Note that it would still be the programmer's responsibility to ensure any desired effects are realized on the new node.

My guess is that nearly all interesting expressions would end up either place-polymorphic or inside an unpack expression (and calling place-polymorphic functions). Only functions given to a particular known node (ie a central server) would escape this treatment.

I think under this model we should have (placement) subtyping, unfortunately. We want to be able to discuss "an arbitrary node that has server capabilities." We don't want it to be ad-hoc; we might also have a subtype for "an arbitrary node that has database capabilities:" this would be based only on the types we assign to the functions that operate over these types.

2.6 Benchmarks

2.6.1 Multi-node benchmarks

Where's the multi-node benchmarks? Obviously none of the prior papers support this, so I must look elsewhere. Andreas may have some good examples here, says Arjun, though I'm not sure exactly where to find them. There's an ndlog program in engine/include/program.h of megascale50k that that may be worth imitating, although decoding its exact meaning may take some time.

2.6.2 2-Node Tierless programming

A list of tierless programming benchmarks from each paper:

2.6.2.1 PLT Web

The Continue system is designed to support paper submission and review for academic conferences.

2.6.2.2 Swift

Swift is benchmarked on 6 different web applications: Guess-A-Number, Shop, Poll, Secret Keeper, Treasure Hunt, and Auction.

2.6.2.3 Ur/Web

Many real Ur/Web programs exist. Called out in the cited paper are a feed reader, a bitcoin exchange, a mapping application, a firmware hosting site, and an interface for a proof assistant.

2.6.2.4 Mogk 2019

TodoMVC

2.6.2.5 Pyxis

Pyxis is benchmarked on standard databases benchmarks(?) TPC-C and TPC-W.

2.7 Placement type inference

We could do constraint-based type inference on these placement types. We could have a language for specifying constraints, ie specifying the weight of communication between two nodes (those in the same network might have ~zero) or the bandwidth cost of a given expression. The reduction would probably look a lot like Swift's or Pyxis' min-flow reductions.

2.8 Problems / Next Steps

1. Choose some (multi-node) benchmark programs
2. Further formalize existential types
3. Think about placement subtyping
4. What to write down in a semantics. We could write this as essentially a lambda calculus semantics. But I want to be able to eyeball / prove properties about placement and messages.
5. Write down the semantics given the above.
6. Type inference:

    1. Figure out partition constraint language
    2. Figure out what problem to reduce to (SMT?)

Wontfix?

1. State (ref) lives on the node that created it and *cannot* be transferred.

2.9 Names

- Psystem eFedrine (PSF)
- Mycelium

3 Going further

More potentially novel things to do in this space that I might be interested in (WARNING: DECREASINGLY IN TOUCH WITH REALITY. TAKE SERIOUSLY AT YOUR OWN PERIL):

3.1 Tierless programming languages

1. An effect-based approach (standard effects with placement annotations) to describing tierless programs
2. A compiler from the effects formulation to the types formulation from Section 2.

3.2 Concurrency and consistency

1. Generalize automatic database partitioning (TODO: what is this called? Look at more related work) to different consistency guarantees (using a MixT-like method?). ie, I can annotate state / expressions? with consistency guarantees and the database / concurrency stories are handled for me.
2. A generalization of tierless programs to concurrent programs (TODO: i'm inconsistent above in how i talk about concurrency. Write explicitly about it and make it more orthogonal). Note that this requires thinking about consistency.
3. In standard straight-line programs (!), measure what consistency expectations programs have. Perform analyses to determine what consistency level is necessary for different parts of programs to be sound.

3.3 Optimizing tierless programs

Using an already-tierless program, certain optimizations that would otherwise be difficult to describe may be easier:

1. Using the information available in a tierless program, and potentially some additional annotations, provide more eager client speculation than done in Chong et al. (2007)
2. Using the information available in a tierless program, and potentially some additional annotations, provide automatic parallelization for performance.

4 More tierless programming

4.1 Placeful Effects

At a baseline, any program over multiple nodes could be modeled as a single program where effects (including state) are realized by separate nodes (intuitively). As a consequence, I think it might be neat to think about placement of computation as a performance optimization only. The result might be that a high-level language would look like an effectful functional programming language where effects have node placements. Maybe this language would be compiled to a language with placement types (like that implied above) as an IR in which to perform optimizations.

I think it should be possible to compile this language into the place-types language.

5 Concurrency and consistency

A program that is written with no concurrency may make assumptions about linearity of state updates, however the speed of light dictates that this may be high-latency. Allowing for concurrency allows us to gain back performance, but may break these guarantees.

5.1 Automatic database partitioning

(TODO: What is this called?) Some programming languages automatically determine placement of program state in memory or on a database. Some tierless programming languages do something similar either with an embedded query language (Ur/Web) or a supported subset of the language (Links). This is also referenced in Related Work for Pyxis. Some languages for serverless computing also do this [11], [12].

5.2 Concurrency

5.2.1 Concurrent program optimization

Pyxis does statement reordering, but as far as I know it doesn't do *concurrent* statement reordering (why not?). I'm thinking of something a bit like async-await:

```
L0 let x: server = ...
L1 let y: client = f_client1 x in
L2 f_server1_very_slow x;
L3 let z = f_client2 y
L4 f_server2 z
```

Here we can infer that L2 doesn't depend on values from L1 and can thus be run concurrently while awaiting the response (`y`) from L2 in order to run L3/L4. My understanding is that Pyxis would be able to reorder L2 and L3 to avoid two roundtrips, but still run the entire program linearly (but I'm not sure).

It's worth noting that client speculation is also a form of concurrency, and so those two concepts are related. Some abstractions that would work for this form of concurrency might work for speculation (and vice versa?).

5.2.2 Explicit concurrency

Something dead simple like having multiple program threads or coroutines. Is this easy besides the consistency / shared memory model? This comes from the Roblox thought of having "multiple scripts" or something.

5.3 Consistency

One approach to allowing concurrency is relaxing guarantees about consistency.

Roblox programs already have weak consistency guarantees, so one solution is to simply say that no state has linear consistency guarantees.

By the way Mae Milano says distributed garbage collection is fine and the state of the art is reasonable. It's just the correct combination of reference counting (between nodes) and mark/sweep (within nodes).

Another idea is to ask programmers to annotate their consistency expectations. Additionally, with some additional annotations or type inference, it may be possible to optimize out shared state when values are immutable or owned by a single node (TODO: break this sentence and part of these rules out elsewhere (where?)).

$$l ::= \text{server} \mid \text{client} \mid ...$$

$$p ::= \text{repl } \{l_1, ...\} \mid \text{owned } l \mid \text{imm } \{l_1, ...\} \mid \alpha$$

In terms of actual typing rules, this requires some affine type nonsense (to-be-understood). It also has a potentially neat rule for the type of λ (modulo affine nonsense): (TODO: what the heck. what. this is cute but think about this before pasting it in a random spot)

$$\frac{\Gamma; x : t \vdash e : s \qquad \bigcap_{\text{fps}} = \{l_1, ...\} \qquad \text{fps} = \{p \mid p\ u \in \text{fv}(e)\}}{\lambda x : t.e : \text{imm } \{l_1, ...\}\ t \to s}$$

5.4 Unannotated Consistency

Ignore tierless programming for a second here (even though it's potentially related). We want to be able to take programs that were written targetting a particular consistency guarantee ("Roblox status-quo") and change the consistency guarantees. Obviously upgrading consistency means these programs are still correct, but may take a performance penalty. Meanwhile, downgrading consistency of the backing database may improve performance, but a MixT-style runtime may restore semantic guarantees in some areas (at some cost). Could we do some dynamic or static analysis that indicates that such a downgrade would be possible? Could we make a modification to the language or runtime such that parts of programs that could be downgraded without creating bugs could be optimized, while others are backed by a MixT shim?

6 Optimizing tierless programs

6.1 Speculation

Swift showed that speculation can come as a side effect of a good tierless programming model. I think with some additional annotations, even proper real-time speculation could be done. For example, Pyxis does statement reordering based on the control-flow graph. This alone can improve Swift's speculation. We could also add an annotation on some types that marks them "rewindable." Type inference would sloppily annotate rewindable values which must live on one node as living on multiple nodes when it greatly reduces messages passed, and insert rewinding code that re-synchronizes those values as soon as it's no longer beneficial.

6.2 Automatic parallelization

I know there's a wealth of work on automatic parallelization. Even Haskell has this built in. Surely if automatic parallelization was applied to a tierless program, where the parallelized components live on different nodes, programs could scale to multiple servers for performance without (any?) annotations. Rather than Pyxis, which improves performance by reducing round-trips, this would allow multiple nodes to do computation at the same time. This may assume there exists a concurrency story.

References

[1] S. Krishnamurthi, P. W. Hopkins, J. McCarthy, P. T. Graunke, G. Pettyjohn, and M. Felleisen, "Implementation and use of the PLT scheme Web server," *Higher-Order and Symbolic Computation*, vol. 20, no. 4, pp. 431–460, Nov. 2007, doi: 10.1007/s10990-007-9008-y.

[2] S. Chong *et al.*, "Secure web applications via automatic partitioning," *ACM SIGOPS Operating Systems Review*, vol. 41, no. 6, pp. 31–44, Oct. 2007, doi: 10.1145/1323293.1294265.

[3] K. Pullicino, "Jif: Language-based Information-flow Security in Java." Accessed: Jun. 06, 2023. [Online]. Available: http://arxiv.org/abs/1412.8639

[4] A. Cheung, S. Madden, O. Arden, and A. C. Myers, "Automatic partitioning of database applications," *Proceedings of the VLDB Endowment*, vol. 5, no. 11, pp. 1471–1482, Jul. 2012, doi: 10.14778/2350229.2350262.

[5] E. Cooper, S. Lindley, P. Wadler, and J. Yallop, "Links: Web Programming Without Tiers," *Formal Methods for Components and Objects*, vol. 4709. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 266–296, 2007. doi: 10.1007/978-3-540-74792-5_12.

[6] A. Chlipala, "Ur/Web: A Simple Model for Programming the Web," in *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Mumbai India: ACM, Jan. 2015, pp. 153–165. doi: 10.1145/2676726.2677004.

[7] A. Guha, J.-B. Jeannin, R. Nigam, J. Tangen, and R. Shambaugh, "Fission: Secure Dynamic Code-Splitting for JavaScript," p. 13, 2017, doi: 10.4230/LIPICS.SNAPL.2017.5.

[8] M. Milano and A. C. Myers, "MixT: a language for mixing consistency in geodistributed transactions," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, Philadelphia PA USA:

ACM, Jun. 2018, pp. 226–241. doi: 10.1145/3192366.3192375.

[9] H. Miller, P. Haller, and M. Odersky, “Spores: A Type-Based Foundation for Closures in the Age of Concurrency and Distribution,”

[10] G. Germain, “Concurrency oriented programming in termite scheme,” in *Proceedings of the 2006 ACM SIGPLAN workshop on Erlang*, Portland Oregon USA: ACM, Sep. 2006, p. 20. doi: 10.1145/1159789.1159795.

[11] A. Jangda, D. Pinckney, Y. Brun, and A. Guha, “Formal Foundations of Serverless Computing,” *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 1–26, Oct. 2019, doi: 10.1145/3360575.

[12] K. Kallas, H. Zhang, R. Alur, S. Angel, and V. Liu, “Executing Microservice Applications on Serverless, Correctly,” *Proceedings of the ACM on Programming Languages*, vol. 7, no. POPL, pp. 367–395, Jan. 2023, doi: 10.1145/3571206.