Type-Safe Dynamic Placement with First-Class Placed Values

GEORGE ZAKHOUR, PASCAL WEISENBURGER, and GUIDO SALVANESCHI,
University of St. Gallen, Switzerland

Several distributed programming language solutions have been proposed to reason about the placement of data, computations, and peers interaction. Such solutions include, among the others, multitier programming, choreographic programming and various approaches based on behavioral types. These methods statically ensure safety properties thanks to a complete knowledge about placement of data and computation at compile time. In distributed systems, however, dynamic placement of computation and data is crucial to enable performance optimizations, e.g., driven by data locality or in presence of a number of other constraints such as security and compliance regarding data storage location. Unfortunately, in existing programming languages, dynamic placement conflicts with static reasoning about distributed programs: the flexibility required by dynamic placement hinders statically tracking the location of data and computation.

In this paper we present Dyno, a programming language that enables static reasoning about dynamic placement. Dyno features a type system where values are explicitly placed, but in contrast to existing approaches, placed values are also first class, ensuring that they can be passed around and referred to from other locations. Building on top of this mechanism, we provide a novel interpretation of dynamic placement as unions of placement types. We formalize type soundness, placement correctness (as part of type soundness) and architecture conformance. In case studies and benchmarks, our evaluation shows that Dyno enables static reasoning about programs even in presence of dynamic placement, ensuring type safety and placement correctness of programs at negligible performance cost. We reimplement an Android app with ~7 K LOC in Dyno, find a bug in the existing implementation, and show that the app's approach is representative of a common way to implement dynamic placement found in over 100 apps in a large open-source app store.

CCS Concepts: • **Software and its engineering** → **Distributed programming languages**; Domain specific languages; • **Theory of computation** → *Distributed computing models*.

Additional Key Words and Phrases: Distributed Programming, Multitier Programming, Placement Types, Scala, Dynamic Placement, Union Types

ACM Reference Format:
George Zakhour, Pascal Weisenburger, and Guido Salvaneschi. 2023. Type-Safe Dynamic Placement with First-Class Placed Values. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 297 (October 2023), 32 pages. https://doi.org/10.1145/3622873

1 INTRODUCTION

When developing a distributed system, programmers implement the different components as separate modules that are later executed on different machines. The information about *placement* of data and computation in the distributed system can be exploited to reason about software correctness, fault tolerance and optimization. Yet, most programming languages do not directly support abstractions for placement. Thus, recent research investigated dedicated language constructs that support expressing *explicit* and *static* placement information. Serrano et al. [2006] and Cooper et al. [2006] introduce a language with two function annotations: client and server, that indicate where a

Authors' address: George Zakhour, george.zakhour@unisg.ch; Pascal Weisenburger, pascal.weisenburger@unisg.ch; Guido Salvaneschi, guido.salvaneschi@unisg.ch, University of St. Gallen, Torstrasse 25, St. Gallen, SG, 9000, Switzerland.

2475-1421/2023/10-ART297
https://doi.org/10.1145/3622873

function is executed. Later, Murphy VII et al. [2008] developed type systems for placements based on modal logic where *possible worlds* represent different places. Type-based approaches have not only been successful in modeling places but also their interaction. In particular, multiparty session types [Honda et al. 2008] statically specify the communication protocol for passing messages over channels between different locations. Similarly, choreographic programming [Giallorenzo et al. 2020] guarantees safe communication protocols across locations encoded by different role type parameters. Finally, information flow type systems have been applied to define the placement of data and computation such that private data does not leak to untrusted parties [Zdancewic et al. 2002]. The approaches above ensure compile time guarantees of important software properties, such as architectural conformance, confidentiality of private data, and protocol correctness but require all placement to be determined statically.

Dynamic Placement. In a number of computing domains, dynamic placement decisions are essential. First, the placement of a computation and of the data it generates can be dynamically selected. For example, in a controller–worker system, the controller decides – based on the execution environment and the job's parameters – which worker is best suitable [Yang et al. 2016]. In such case, the result of the computation is placed dynamically on the location where it has been generated.

Second, computation can depend on data that is moving among places. For example, frequently-accessed data in a remote database is stored in a cache for faster access [Arani et al. 2020; Zulfa et al. 2020]. An application that manipulates data should be able to do so for both data in the database or in the cache. In contrast, data not in the database nor in the cache might need to be handled differently. For example, data from a client must be sanitized before storage.

The principles above have been applied extensively. Data-intensive computer clusters dynamically choose function placement to co-locate data and computation to handle system resources such as file and memory reads/writes more efficiently [Amiri et al. 2000; Karve et al. 2006]. Network function virtualization separates services such as routers and firewalls from dedicated hardware, allowing them to be moved among network devices, improving scalability [Clayman et al. 2014]. Query processing dynamically places operators to optimize cost metrics such as latency or throughput, e.g., by pushing selections near the sources [Zhou et al. 2005]. Geo-distributed data centers dynamically place frequently used data geographically closer to the consumer [Teyeb et al. 2017].

What these kinds of dynamic placement decisions have in common is that the placement, once chosen, is fixed. Whereas the peer on which data is dynamically placed is picked at run time – and this decision itself can be based on dynamically available data such as latency or environment variables – the peer that has been selected for the dynamic placement cannot change. Thus, we consider a model where the placement decision is dynamic but dynamically placed data cannot change its placement at run time.

Dynamic Placement without Language Support. The lack of programming abstractions for dynamic placement is addressed by developers via encoding placement information in the program.

The first option is to extract the common interface. For example, a client–server system or a controller–worker system would be designed simply as a system with a single placement implementing the request–response protocol, and a database–cache system would be designed as a system where all places can store data. Extracting the common interface means treating the system as an homogeneous one; made of a single type of place, effectively losing the precision required to distinguish different kinds of nodes. This approach may result in run time errors because differences among nodes are abstracted away, e.g., producing side effects on the wrong node or an attempt to retrieve data from a node that is not reachable. For example, in a client–server–database system, developers must ensure that data is always in the database, sensitive computations are run on the server, and the client is not able to communicate with the database except through the server.

The second option is that no interface is explicitly extracted and only the programmer is aware of the equivalent functionalities among places. Thus, places exhibit different interfaces and it is possible to statically reason about where each functionality is located. Yet, the fact that the location of some functionalities can be decided dynamically is not reflected in the program and the same functionality has to be implemented multiple times for different places, leading to code repetitions. Correct dynamic placement can still be guaranteed but at the high cost of reorganizing code and sacrificing composability and modularity (see Section 5).

Safe Dynamic Placement. In this paper, we present Dyno, a language with a novel type system that is able to capture uncertainty of placement. Dyno follows the tradition of successfully applying language and type system features to increase the safety of distributed systems and fills the gap in current approaches to support dynamic placement, which is common in distributed systems but not directly tackled by languages so far.

Dyno's type system enables static reasoning about the placement of values not bound to a specific place before execution – only to a known subset of places. In Dyno, placements are not just annotations but rather a fundamental part of the type system. Placements, however, must be treated differently from data types. Whereas all peers of a distributed system can create a value of some data type, peers should not be able to create a value placed on another peer, e.g., the cache cannot create a value and type it as a value on the database. Dyno's type safety for dynamically placed values stems from the fact that (i) the introduction form for remote references guarantees correct placement and (ii) the type systems tracks the (dynamic) placement of values.

We interpret placed values as remote references: For a function that takes a placed value as input, any interpretation that copies the value to the function's execution location would effectively mingle placement and origin; two concepts which are not generally identical. Remote references fit our model more naturally since a remote reference is placed where the function is evaluated while pointing to a value on the location indicated in the placed value's type. Hence, the interpretation of dynamic placement is derived from the type-union of the previously mentioned remote references, each placed on a different place.

Example. The following example shows a database–cache retrieval interface in Dyno:

```
1 def retrieve(key: Key): Data at (Database | Cache) on Server =
2   if (cached(key)) on[Cache].run.capture(key){ remote ref readCache(key) }.asLocal
3   else on[Database].run.capture(key){ remote ref readDB(key) }.asLocal
```

Depending on whether a key is in the cache or in the database (checked by `cached(key)`, Line 2), the code either creates a reference (via `remote ref`) to the value in the cache (Line 2) or to the value in the database (Line 3). Since both branches create placed values with different placements, the type of `retrieve` is `Data at (Database | Cache) on Server` (Line 1), i.e., a reference on the `Server` to a `Data` value that lives on the `Cache` or the `Database`. The union type `Database | Cache` expresses placement uncertainty.

Contribution. We present a type system that allows representing dynamic placement at the type level, providing static type safety for dynamically placed remote values.[1] In summary, this paper makes the following contributions:

- We introduce the design of Dyno, a distributed language where data and computation locations are explicit and have first-class status. We propose a novel type-level interpretation of dynamic placement as a combination of placement types and union types (Section 3 and Section 4).

[1]All artifacts presented in this paper are publicly available at https://doi.org/10.5281/zenodo.8148841

- We show how existing programming frameworks for distributed systems fall short to support safe dynamic placement of computation and data and demonstrate how these issues can be addressed with Dyno (Section 5).
- We define a formalization of Dyno with a type system where placed values are first-class citizens, prove that it is sound, and that it preserves architectural conformance even in the case of dynamic placement (Section 6).
- We describe Dyno's implementation (Section 7) and present an evaluation that illustrates several applications where our type system enables simpler and more robust code, increasing safety (Section 8.1) at a negligible performance overhead (Section 8.2). In a larger case study with ~ 7 K LOC, we reimplement an open-source Android application (Section 8.3) in Dyno and discover a placement bug in the existing implementation. We show that dynamic placement – encoded as URL schemes – occurs in over a hundred open-source Android applications in the F-Droid app store. (Section 8.4).

2 BACKGROUND

Our approach builds on languages where placement is explicit. We show an encoding of placement based on Scala since we adopt it throughout the paper. However, similar ideas apply to other languages with explicit placement [Murphy VII et al. 2008; Reynders et al. 2020].

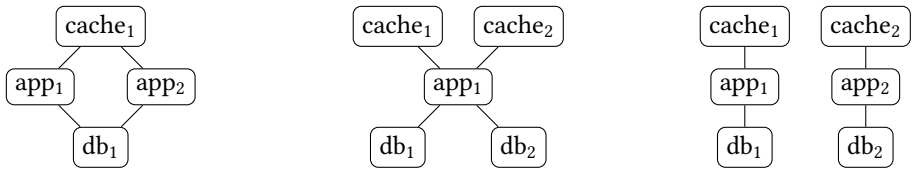
In the flavor of explicit placement that we use to illustrate our approach in the rest of the paper, the components of the distributed system (e.g., database, cache, and app) are modeled as *peer types* and their architectural relation as *ties*, i.e., a description of the system's architecture. Hence, peer types represent logical places (i.e., two components could be deployed to the same physical machine). Ties are of three kinds: single ties between two peer types guarantee that every instance of the first peer type is connected to exactly one instance of the other; optional ties between two peers types guarantee that every instance of the first is connected to at most one instance of the other while enabling an instance of the other peer to join and leave; and multiple ties allow every instance of the first to be connected to zero or more instance of the other, and like optional ties, enables peers to join and leave. Thus, optional and multiple ties allow for dynamic topologies within the architectural constraints. We have chosen to allow these multiplicities in Dyno to stay in line with previous work on expressing architectural relations [Balzer 2011; Harkes and Visser 2014; Steimann 2013, 2015], which statically models only a subset of zero, one, zero-or-one, zero-or-more, and one-or-more relations. We find this is a sweet spot in the design space between providing useful static guarantees and keeping the type system simple. Moreover we have found that more granular multiplicities are rarely needed in practice.

Throughout this paper we use an encoding that embeds peer types and their constraints into Scala types. The encoding uses a combination of subtyping, type refinements, and compound types.

```
1 @peer type App      <: { type Tie <: Single[Database] with Single[Cache] }
2 @peer type Database <: { type Tie <: Multiple[App] }
3 @peer type Cache    <: { type Tie <: Multiple[App] }
```

The first line specifies that every application is *tied* to a single database and a single cache. The remaning lines specify that every database and cache must be tied to any number of applications but never to each other.

Figure 1 illustrates three possible configuration of multiple databases, caches, and applications connected in different ways. Boxes with rounded corners represent peers that belong to the configuration, and edges represent an established network connection between the two peers. As-is, the architecture in the previous code snippet describes the configurations in Figures 1a and 1c but it does not describe the configuration in Figure 1b.



(a) Diamond. (b) Star. (c) Disconnected linear.

Fig. 1. Configurations of a database–cache–application architecture

We can now place values and methods on the peers defined in the architectural specification. The following code snippet follows the database–cache retrieval example from Section 1:

```
4  def readCache(key: Key): Data on Cache = /* ... */
5
6  def readDB(key: Key): Data on Database = /* ... */
7
8  def retrieve(key: Key): Data on App =
9    if (cached(key)) on[Cache].run.capture(key){ readCache(get(key)) }.asLocal
10   else on[Database].run.capture(key){ readDB(query(key)) }.asLocal
```

The *placement types* on methods or values using the `on` keyword indicate on which peers the methods are executed and the values are located. For example, the `readCache` method is placed on the cache, `readDB` on the database and `retrieve` on the application. Placing a method or a value on a peer means that it is evaluated on the given peer. For example, when calling `retrieve`, the method is executed on the `App`, i.e., Lines 9 and 10 are evaluated in the `App` context. Any attempt to call `retrieve` directly on the cache or on the database is rejected by the type system.

Instead, diverting the control flow to a remote peer is explicit using the `on[...].run.capture(...){ ... }` construct, which specifies the peer on which the execution takes place, the values transferred over the network, and the expression to execute in the context of the remote peer. The captured expressions are first evaluated to a value then serialized and transfered over the network, i.e., the captured values are copied and peers do not share mutable state. The `asLocal` marker accesses a remote value transferring the value over the network and constructing a local copy. For example, `retrieve` either reads the data from the cache (Line 9) or the database (Line 10), depending on whether a given key is cached or not.

Placement ensures some safety properties: For example, directing the control flow from the database to the cache and vice versa would violate the specified architecture and thus is rejected by the compiler. Hence, encoding placement at the type level allows the compiler to statically enforce architectural constraints and interleaving local and remote code allows type-checking across the complete distributed system.

3 DYNO PROGRAMMING ABSTRACTIONS

In this section, we introduce Dyno's abstractions and we show how they can be combined to support type-safe dynamic placement.

3.1 First-class Placed Values

In Dyno, the type `T at P` expresses the placement of remote values. A value of type `T at P` is a first-class placed value that can be passed as argument, returned by a function, and bound to an identifier. For example, in a controller–worker application driven by a client, the type of a function that receives a job from a client and returns the job's result after it is executed on a worker is `(Job at Client) => (Result at Worker)`.

We interpret values of type `T at P` as remote references that point to values of type `T` at place `P`. Unlike `T on P` (see Section 2), `T at P` does not represent where expressions are evaluated but rather where evaluated values live. Hence, the type `T at P` indicates that a value of type `T` exists at a peer `P` and can be retrieved remotely while a value of type `T on P` indicates that a value of type `T` can be used only in the context of `P`. Whereas `T at P` can occur anywhere in the type, `T on P` cannot occur as part of another type, i.e., `T on P` cannot be a function's argument type, nor be its return type, nor be an argument to a type-level operator. Thus, the type of the aforementioned job-execution function, if it exists on the controller, is `(Job at Client) => (Result at Worker) on Main`.

Creating and using first-class placed values. Using first-class placed values in the running database–cache example of the introduction, the following code places a method on the server which takes a key as argument and returns a value of type `Data` that lives on the database:

```
def retrieveFromDatabase(key: Key): Data at Database on App = /* ... */
```

For an expression `e` of type `T`, executing `remote ref e` at peer `P` creates a remote reference to a local value that has type `T at P`. A remote reference retains its type when it is copied to another peer. We now show an implementation of `retrieveFromDatabase`:

```
def retrieveFromDatabase(key: Key): Data at Database on App =
  on[Database].run.capture(key){ remote ref readDB(query(key)) }.asLocal
```

The call to `on[Database].run` moves the program flow from the `App` to the `Database`. The cache retrieves the cached value and creates a reference to the value via `remote ref`, which has type `Data at Database`. The remote reference is returned to the server through `asLocal`.

Analogously, we place `retrieveFromCache` on the `App` to get a `Data at Cache` from the `Cache`:

```
def retrieveFromCache(key: Key): Data at Cache on App =
  on[Cache].run.capture(key){ remote ref readCache(get(key)) }.asLocal
```

To retrieve the value to which a remote reference points, i.e. to dereference it, we call `deref` on the remote reference (i.e., on a value of type `T at P`). Thus, both `asLocal` and `deref` signal a remote access and the creation of a local copy of remote values. The following code snippet dereferences the result of `retrieveFromDatabase`:

```
on[App] { retrieveFromDatabase(/* some key */).deref }
```

This line of code type-checks if the server is tied to both the database and the cache. Thanks to the type-level encoding of the architecture, we can statically reject remote dereferences that would fail at run time, providing safety through compiler checks.

3.2 Dynamically Placed Values

Dyno's core feature to track dynamically placed values statically are *placement union types*. For example, the type `Data at (Database | Cache)` is a remote reference to a value of type `Data` that lives either at a `Database` or at a `Cache`. The actual placement is chosen at run time. A modified version of the `retrieveFromDatabase` method returns a value of type `Data at (Database | Cache)` on the `App`:

```
def retrieveFromDatabase(key: Key): Data at (Database | Cache) on App =
  on[Database].run.capture(key){ remote ref readDB(query(key)) }.asLocal
```

It is valid to assign to a variable of type `T at (P | Q)` a remote reference with type `T at P` or with type `T at Q`. Similarly, it is valid to call a function expecting a `T at (P | Q)` with a remote reference of type `T at Q`. Hence, the above snippet type-checks. Dereferencing a remote reference of type `T at (P | Q)` using the `deref` method type-checks if it is evaluated on a peer tied to both `P` and `Q`.

Using first-class dynamically placed values, we combine both methods `retrieveFromDatabase` and `retrieveFromCache` into a single `retrieve` method:

```
def retrieve(key: Key): Data at (Database | Cache) on App =
  if (cached(key)) on[Cache].run.capture(key){ remote ref readCache(get(key)) }.asLocal
  else on[Database].run.capture(key){ remote ref readDB(query(key)) }.asLocal
```

The method provides a uniform way to query both the database and the cache while retaining the correctness of the placement information.

3.3 Placement Inspection

Dyno's core features aim at making it possible to treat placed values safely and independently of their dynamically chosen place. In certain situations, however, user code may be required to make decisions based on the run time placement of a value. Thus, placed values provide an interface to access the concrete placement associated to them at run time.

First, calling `v.peer` on a placed value `v` retrieves the peer on which the referenced data lives.

Second, if `v` was dynamically placed on `P | Q`, then `v.toEither[P, Q]` retrieves a value of type `Either[T at P, T at Q]` which enables an exhaustive pattern match on the peer type a reference points to. The `fromEither` performs the reverse operation.

In the running database–cache example, we can use this feature to add the fetched data to the cache if it originated from the database. The following code assumes an `add` method on the `Cache`:

```
1 on[App] {
2   val cursor: Data at (Database | Cache) = retrieve(/* some key */)
3
4   cursor.toEither[Database, Cache] match {
5     case Left(dbCursor) =>
6       val data = dbCursor.deref
7       on[Cache].run.capture(data){ add(data) }
8       data
9     case Right(cacheCursor) => cacheCursor.deref } }
```

From the `cursor` reference to some data on the database or the cache, we pattern match on the placement (Line 4). If the retrieved value originated from the database (Line 5), we add it to the cache (Line 7) and return it (Line 8). Otherwise, we return the already cached value (Line 9).

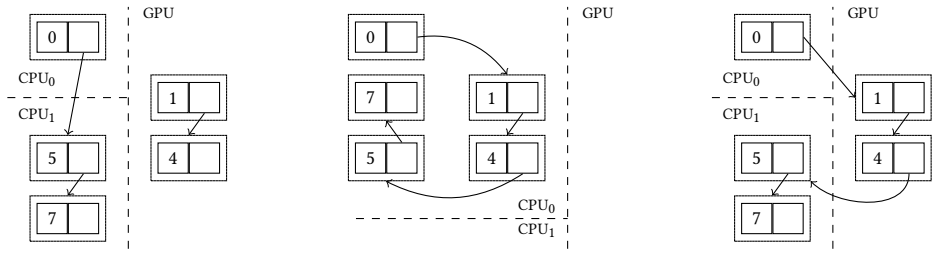
4 DYNO IN ACTION

This section demonstrates how Dyno's abstractions for first-class dynamically placed values enable accounting for uncertainty in dynamic placement decisions.

4.1 Dynamically Distributed Data Structures

First-class placed values enable expressing distributed data structures that span multiple peers. Without them, some tasks can be tedious and sometimes impossible [Giallorenzo et al. 2021].

Let us consider an application that merges two sorted remote linked lists of integers into one. Figure 2 shows the three conceptual steps involved in the merge. The lists are comprised of cells, each of which is a pair of an integer and a reference to the next cell. We represent each cell as the outer solid box, the pair as the two adjacent inner boxes, references as arrows, and the network boundary between peers as a dashed line. Horizontal dashed lines represent instances of the same peer type while vertical ones represent different peer types. In Figure 2a, the CPU peers hold each a linked list, and the GPU peer holds one. Without dynamic placement references are only allowed to cross horizontal separations between peers of the same type. In Figure 2b, all peers move their linked list to the first CPU which proceeds to merge the two lists. At this stage, the merged list is a local one and not a distributed one as all its members are on a single peer. In Figure 2c, we move



(a) Separate lists before merging. (b) Moving lists and local merging. (c) Keeping data at its origin.

Fig. 2. First-class dynamically placed values as remote references are needed for merging remote lists.

the cells back to where they originated. This forces some references to become remote ones. So dynamic placement allows crossing vertical separations as well as horizontal ones.

The place to which the references point depends on the run time values in the list. Hence, the references are dynamically placed values. If we represent the left peer by the peer type `CPU` and the right peer by `GPU`, we can type the next cell reference as `Cell at (CPU | GPU)`.

The following code implements the merge of Figure 2 (but without actually ever constructing the intermediate state Figure 2b with the complete list transferred from one peer to the other):

```
1  @peer type CPU <: { type Tie <: Multiple[GPU] }
2  @peer type GPU <: { type Tie <: Multiple[CPU] }
3
4  type RemoteList = List[Int] at (CPU | GPU)
5
6  sealed trait DList
7  case class DCell(head: Int, tail: DList at (CPU | GPU)) extends DList
8  case object DNil extends DList
9
10 def sortByHead(l1: RemoteList, l2: RemoteList): Option[(RemoteList, RemoteList)] on (CPU | GPU) = {
11   def head(l: RemoteList): Option[Int] = on(l.peer).run.capture(l){ l.deref.headOption }.asLocal
12   (head(l1), head(l2)) match {
13     case (Some(h1), Some(h2)) => if (h1 <= h2) Some((l1, l2)) else Some((l2, l1))
14     case (Some(h1), None)     => Some((l1, l2))
15     case (None, Some(h2))     => Some((l2, l1))
16     case (None, None)         => None } }
17
18 def merge(l1: RemoteList, l2: RemoteList): DList at (CPU | GPU) on (CPU | GPU) = sortByHead(l1, l2) match {
19   case None => remote ref DNil
20   case Some((small, big)) => on(small.peer).run.capture(small, big){
21     small.deref match {
22       case smallHead :: smallTail => remote ref DCell(smallHead, merge(remote ref smallTail, big))
23       case Nil => merge(big, remote ref Nil) } } }
```

Lines 1 to 2 declare the peer types CPU and GPU – both with ties to each other. We omit the code on the CPU and GPU to generate the lists. Line 4 defines a `RemoteList` as a reference to an ordinary list placed on the CPU or on the GPU.[2] Lines 6 to 8 define a dynamically placed distributed list. The `sortByHead` function (Lines 10 to 16) takes two references to remote lists, and returns a pair with the list with the smallest head first. The `merge` function (Lines 18 to 23) takes two references to

[2]Dyno's implementation allows generalizing `RemoteList` to be polymorphic over placements. A polymorphic `RemoteList` can be defined as follows:

```
sealed trait RemoteList[P, T]
case class Nil[P, T]() extends RemoteList[P, T]
case class Cons[P, T](head: T, tail: RemoteList[P, T] at P) extends RemoteList[P, T]
```

Multitier modules that are polymorphic over placements were described by Weisenburger and Salvaneschi [2019].

remote lists, and produces a reference to a dynamically placed distributed list on the CPU or on the GPU. Lines 20 to 23 access the peer holding the smallest head and uses it to construct a `DCell` list whose tail is the result of the recursive call. Crucially, the recursive call can jump back and forth between the places where the smallest head is.

This implementation avoids copying the complete list onto the merging peer. Instead, we always copy only the head of the list currently needed for comparison, which allows us to merge the two lists in bounded memory. Hence, merging lists beyond a single machine's capacity becomes possible. After merging, every consumer who may partially traverse the list can be sent the head of the list and a reference to its tail as opposed to the entire list.

4.2 Dynamically Placed Computations

In Dyno, it is possible to reason about dynamically placed closures. References to placed closures can be moved to other peers who can only use them to remotely evaluate the closure at the peer on which the closure was created.[3]

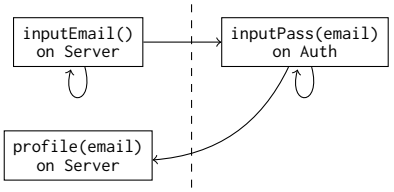


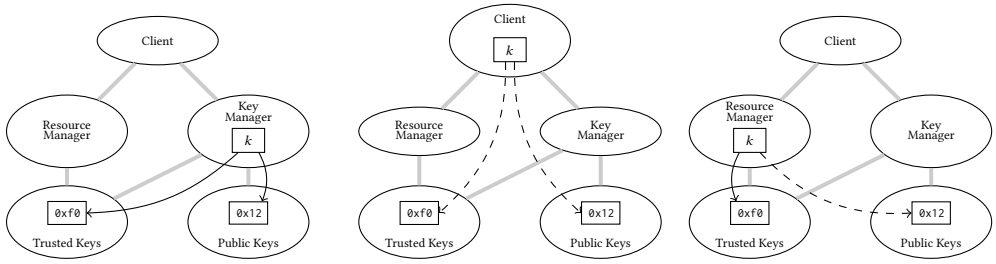
Fig. 3. Web session state machine.

Web applications are often stateful and traditionally implemented with session tokens, i.e., keys of a (persistent) map on the server, which contains the information required by the next step to progress through the Web application. Instead of managing such a map, we replace session tokens with placed closures that capture the information needed by the next step.

Consider an example where a user logs into their profile page. First, the user is asked their email, then the computation moves to an authentication service that asks the user's password. The request is repeated if the credentials are incorrect. Otherwise, the computation moves back to the server and provides the contents of the user's profile. The state machine of this application is in Figure 3. The boxes are the user's states, the arrows are the transitions between states, and the dashed line is the network boundary. The following code implements the example:

```
1  @peer type Client <: { type Tie <: Single[Server] with Single[Auth] }
2  @peer type Server <: { type Tie <: Single[Client] with Single[Auth] }
3  @peer type Auth   <: { type Tie <: Single[Client] with Single[Server] }
4
5  case class Response(body: String, next: Option[String => Response at (Server | Auth)])
6
7  def inputEmail() = on[Server] { Response("E-Mail: ", Some(remote ref { email =>
8    if (validate(email))
9      on[Auth].run.capture(email){ inputPass(email) }.asLocal else inputEmail() })) }
10
11 def inputPass(email: String) = on[Auth] { Response("Password: ", Some(remote ref { pass =>
12   if (verify(email, pass))
13     on[Server].run.capture(email){ profile(email) }.asLocal else inputPass(email) })) }
14
15 def profile(email: String) = on[Server] { Response(s"Welcome $email", None) }
16
17 def callNext(next: String => Response at (Server | Auth), input: String): Response on Client =
18   on(next.peer).run.capture(next, input){ (next.deref)(input) }.asLocal
19
20 def main() = on[Client] {
21   def run(response: Response): Option[Response] = {
22     println(response.body)
23     response.next match { case Some(next) => run(callNext(next, io.StdIn.readLine())) case _ => None  } }
24   run(on[Server].run{ inputEmail() }.asLocal) }
```

[3]We do not allow dereferencing a placed closure remotely as this would entail copying all values that are implicitly captured in the closure. In contrast, Spores [Miller et al. 2014] make captured values explicit and allow to transfer the closure.



(a) Key request to key manager. (b) Key moved to the client. (c) Key moved to resource manager.

Fig. 4. A remote reference can move to a peer that has no ties to the peer the referenced value is at.

The architecture features three peers: a server, a client, and an authentication service (Lines 1 to 3). Line 5 defines a response which has a body and possibly a reference to the next stage of the application. Stages are represented by dynamically placed closures that refer to closures on the server or on the authentication service since the application's execution moves between both peers.

The user can be at three different stages (Lines 7 to 15). The `inputEmail` method (Line 7) corresponds to the stage of the user inputing their email. If the email is not valid, the next stage is the same stage again. Otherwise, the next stage is represented by the result of `inputPass(email)`. The `inputPass` method (Line 11) lives on the authentication service. Crucially, the email is captured inside the closure created at Line 9, i.e., when the reference is created, the email is stored in the closure. If the email–password pair is correct, the computation is transferred back to the server when the closure wrapping `profile(email)` is called. The `profile` method (Line 15), which does not have any continuation, defines the end of the application. The main method (Line 20) starts with the response of `inputEmail()` on the server and keeps calling each response's continuation with an input until there are no continuations to execute. `callNext` (Line 17) calls its placed closure argument on the peer on which it is located (`next.peer`).

The example uses union placement types to enable static reasoning about the execution location of remote continuations. In particular, the type system guarantees that every remote continuation is invokable by the client (i.e., the architecture specifies a connection from the client to the locations that can inhabit possible continuations), ensuring that every state is reachable from the client.

4.3 Architecture-Conformant Dynamic Placement

The Dyno type system forbids accessing a value that is dynamically placed on a peer that the accessor has no ties to. This restriction applies to the data referenced by the placed value and not the reference itself that encodes the dynamic placement. For example, a client with no tie to a database may get hold of a reference to data in the database but cannot directly access the referenced data.

Figure 4 shows the movement of a value k along an architecture with five peers: a database of trusted keys, a database of public keys, a key manager, a resource manager, and a client. Only the peers connected by a thick grey edge are tied, and thus can communicate. Solid arrows represent remote references that can be dereferenced while dashed arrows represent remote references that cannot. Values are represented in a box. First, in Figure 4a, the client has requested the key manager to create a key for it. If the user has a trusted key, the resource manager will return a reference pointing to the trusted key. If the user is not trusted, the reference will point to a public key. Thus, the key is dynamically placed. Second, in Figure 4b, the client has received the key from the key manager. The key still points to the correct key on either database. However, since the client does not have a tie to either of them, the client is unable to dereference the reference and obtain the

referenced value of the key, trusted or not. Finally, in Figure 4c, the client has sent the key to the resource manager. In contrast to the situation with the client, the resource manager is allowed to dereference the key reference only if it comes from the trusted key database since a tie to it exists.

The following code implements the interaction of Figure 4:

```
1  @peer type Client        <: { type Tie <: Single[KeyManager] with Single[ResourceManager] }
2  @peer type KeyManager    <: { type Tie <: Single[Client] with Single[TrustedKeyDB] with Single[PublicKeyDB] }
3  @peer type ResourceManager <: { type Tie <: Single[Client] with Single[TrustedKeyDB] }
4  @peer type TrustedKeyDB  <: { type Tie <: Single[KeyManager] with Single[ResourceManager] }
5  @peer type PublicKeyDB   <: { type Tie <: Single[KeyManager] }
6
7  def getKey(userId: ID): Key at (TrustedKeyDB | PublicKeyDB) on KeyManager =
8    if (userId == adminId) on[TrustedKeyDB].run.capture(userId){ remote ref key.get(userId) }.asLocal
9    else on[PublicKeyDB].run.capture(userId){ remote ref key.get(userId) }.asLocal
10
11 def getResource(key: Key at (TrustedKeyDB | PublicKeyDB)) : Option[Resource] on ResourceManager = {
12   key.toEither[PublicKeyDB, TrustedKeyDB].toOption map { trustKey => unlockResource(trustKey.deref) } }
13
14 def main() = on[Client] {
15   val key: Key at (TrustedKeyDB | PublicKeyDB) = on[KeyManager].run.capture(myId){ getKey(myId) }.asLocal
16   val resource: Option[Resource] = on[ResourceManager].run.capture(key){ getResource(key) }.asLocal
17   /* use the resource here */ }
```

Lines 1 to 5 define the peers and their ties as in the architecture of Figure 4. The `getKey` function (Lines 7 to 9) lives on the key manager. If the user asking for the key is the admin, a reference to a trusted key is returned, otherwise a reference to a public key. The resource manager's `getResource` function (Lines 11 to 12) examines the placement of the reference. If the key is from the trusted key database, `trustKey.deref` fetches the key. Otherwise, `toOption` yields `None`. Fetching type-checks thanks to the tie between the resource manager and the trusted key database. Fetching from the public key database would lead to a compilation error because such access would violate the architecture. Finally, Lines 14 to 17 define the client's main function, which obtains a key reference from the key manager using `getKey` and passes it to the resource manager's `getResource` function.

5 PLACEMENT CORRECTNESS

Thanks to the first-class treatment of placed values which allows us to retain the referenced value's placement in the type, accessing placed values is safe and conforms to the architectural constraints. For every dereference, the type system checks if a tie between the dereferencing peer and the referenced peer exists in the architecture. Thus, in Dyno, if a remote reference has the type `T at (P | Q)`, it is guaranteed to point to data of type `T` either at `P` or at `Q`. We call this property *placement correctness* (which will formalize later). Informally: placements do not lie.

In lack of a type system with support for dynamic placement, developers can add a tag to data that represents the placement, but without enforcing the correctness of the tag. Enforcing correctness without support from the type system hinders modularity.

Tagging does not guarantee placement correctness. In languages that lack support for first-class placed values, one can alternatively tag values with placement information, e.g., by always transmitting a pair `(Data, Tag)` or through an ad-hoc insertion of the place in the type. An example is manually encoding the placed value `Data at Cache` through a `CacheCursor` class, which is a simple wrapper of a `Cursor` pointer to a `Data` value (similarly, a `DBCursor` encodes a `Data at Database`):

```
1  case class CacheCursor(cursor: Cursor)
2  case class DBCursor(cursor: Cursor)
3
4  def retrieve(key: Key): Either[CacheCursor, DBCursor] on App =
5    if (cached(key)) Left(CacheCursor(on[Cache].run.capture(key){ get(key) }.asLocal))
6    else Right(DBCursor(on[Database].run.capture(key){ query(key) }.asLocal))
```

However, in such solution, the data being wrapped could have been computed elsewhere. The placement tag is not statically checked, so the programmer could accidentally wrap cache data with `DBCursor` or wrap database data with `CacheCursor`. The following code is the same as above but with the the `CacheCursor` and `DBCursor` creation swapped in both branches by accident. The code type-checks since correct placement cannot be statically enforced:

```
1 def retrieve(key: Key): Either[CacheCursor, DBCursor] on App =
2   if (cached(key)) Right(DBCursor(on[Cache].run.capture(key){ get(key) }.asLocal))
3   else Left(CacheCursor(on[Cache].run.capture(key){ get(key) }.asLocal))
```

Guaranteeing placement correctness sacrifices modularity. The placement tag can be statically checked if an encoding similar to the one in the following listing is chosen. For example, calling `Cache.remoteRef[Cursor]` anywhere outside Lines 5 and 7 would result in a compilation error. Because `Cache` extends `Place`, `Cache` inherits `remoteRef`, such that `Cache.remoteRef` is only in scope in Lines 5 and 7 since it is protected. Moreover, because the constructor of `Ref` is private to `Place`, constructing a `Ref` anywhere outside the scope of `Place` (Lines 1 and 3) results in a compilation error, including inside `Cache` and `DB`. Therefore, the only way to obtain a value of type `Ref` is by calling either `DB.retrieveFromDatabase` or `Cache.retrieveFromCache`. However this code sacrifices modularity: the `Cache` and `DB` objects cannot be separated from any function that places values on them. Since these functions are arbitrary, embedding them in the `Cache` and `DB` context violates the separation of concerns.

```
1  trait Place {
2    case class Ref[T] private[Place] (val data: T)
3    protected def remoteRef[T](data: T): Ref[T] = new Ref(data) }
4
5  object Cache extends Place {
6    def retrieveFromCache(k: Key): Ref[Int] =
7      remoteRef(on[Cursor].capture(k){ get(k) }.asLocal) }
8
9  object DB extends Place {
10   def retrieveFromDatabase(k: Key): Ref[Int] =
11     remoteRef(on[DB].capture(k){ query(k) }.asLocal) }
12
13 object App extends Place {
14   def retrieve(k: Key): Either[Cache.Ref[Cursor], DB.Ref[Cursor]] =
15     if (cached(k)) Left(Cache.get(k)) else Right(DB.query(k)) }
```

In Dyno, such placement issues cannot occur as the introduction form `remote ref` for references guarantees correct placement without sacrificing modularity. Correctness for remote access is ensured by lifting placement to the type-level and tracking the placement in the type. Hence, placement correctness is part of type soundness – ensured statically by the type checker.

6 FORMALIZATION

In this section we formalize a core calculus for Dyno that models first-class placed values with union placement types for dynamic placement that respect the architectural properties of peers and ties. We formally state and prove the placement correctness property defined in Section 5. Section 6.1 presents the syntax of the core calculus, Section 6.2 presents the evaluation rules, Section 6.3 presents the typing rules, and in Sections 6.4 and 6.5 we prove properties about the calculus such as type soundness and architectural soundness.

6.1 Syntax

The syntax for the Dyno core calculus is in Definition 1. The placement types have one of two general forms: concrete peer types – represented as a lowercase p – or a union of at least one peer

type – represented as an uppercase P. We use the over-bar notation to represent a sequence of zero, one, or more repetitions of the syntactic form under the bar. The types in the calculus are – in order of presentation: basic types, function types, reference types, and peer instance types. The expressions in the calculus are the usual variables, applications, and functions, extended with: explicit peer instances $\pi_{\mathsf{P}}^{\mathsf{p}}$ whose *actual* run time peer type is p (superscript) and its static peer type *annotation* is the P peer type union (subscript); references ℓ_{T} at $\pi_{\mathsf{P}}^{\mathsf{p}}$ to values of type T on a peer instance $\pi_{\mathsf{P}}^{\mathsf{p}}$; peerof e extracts the peer instance from a reference; $\mathsf{broad}_{\mathsf{P}}$ e broadens the placement of a reference e to the union placement type P; the $\mathsf{on}(\mathsf{e}_1).\mathsf{capture}(\overline{\mathsf{x} = \mathsf{e}_2})\ \{\ \mathsf{ref}_{\mathsf{T}}\ \mathsf{e}_3\ \}$ construct executes e_3 of type T on the peer e_1 after explicitly moving all expressions e_2 used by e_3 and returning a remote reference to the remote value; the dereference operator e_1 deref $\{\mathsf{x} \Rightarrow \mathsf{e}_2;\ \Rightarrow \mathsf{e}_3\}$ executes e_2 with the dereferenced value bound to x if the value exists[4] or e_3 otherwise; the placement-match construct e_1 match $\{\overline{\mathsf{x}\ \mathsf{at}\ \mathsf{p} \Rightarrow \mathsf{e}_2}\}$ executes a match's branch based on the actual placement of a remote reference; and similarly e_1 match $\{\overline{\mathsf{x}\ \mathsf{is}\ \mathsf{p} \Rightarrow \mathsf{e}_2}\}$ is a pattern match over peer instances. The values of the calculus are abstractions, peer instances, and references to remote values.

Definition 1 (Syntax).

Placement Types	$\mathsf{P} ::= \mathsf{p}_1, \overline{\mathsf{p}_2}$
Types	$\mathsf{T} ::= \mathsf{B} \mid \mathsf{T}_1 \to \mathsf{T}_2 \mid \mathsf{T}\ \mathsf{at}\ \mathsf{P} \mid \mathsf{Peer}\ \mathsf{P}$
Expressions	$\mathsf{e} ::= \mathsf{x} \mid \mathsf{e}_1\ \mathsf{e}_2 \mid \lambda \mathsf{x} : \mathsf{T}.\ \mathsf{e} \mid \pi_{\mathsf{P}}^{\mathsf{p}} \mid \ell_{\mathsf{T}}\ \mathsf{at}\ \pi_{\mathsf{P}}^{\mathsf{p}} \mid \mathsf{peerof}\ \mathsf{e} \mid \mathsf{broad}_{\mathsf{P}}\ \mathsf{e}$ $\mid \mathsf{on}(\mathsf{e}_1).\mathsf{capture}(\overline{\mathsf{x} = \mathsf{e}_2})\ \{\ \mathsf{ref}_{\mathsf{T}}\ \mathsf{e}_3\ \} \mid \mathsf{e}_1\ \mathsf{deref}\ \{\mathsf{x} \Rightarrow \mathsf{e}_2;\ \Rightarrow \mathsf{e}_3\}$ $\mid \mathsf{e}_1\ \mathsf{match}\ \{\overline{\mathsf{x}\ \mathsf{at}\ \mathsf{p} \Rightarrow \mathsf{e}_2}\} \mid \mathsf{e}_1\ \mathsf{match}\ \{\overline{\mathsf{x}\ \mathsf{is}\ \mathsf{p} \Rightarrow \mathsf{e}_2}\}$
Values	$\mathsf{v} ::= \lambda \mathsf{x} : \mathsf{T}.\ \mathsf{e} \mid \pi_{\mathsf{P}}^{\mathsf{p}} \mid \ell_{\mathsf{T}}\ \mathsf{at}\ \pi_{\mathsf{P}}^{\mathsf{p}}$

6.2 Evaluation Rules

The evaluation rule takes the form $\Omega; \sigma \triangleright \mathsf{e} \to^{\pi} \Omega'; \sigma' \triangleright \mathsf{e}'$ and expresses that e evaluates to e′ on the peer instance π under the peer configuration Ω and the store σ, updating them to Ω' and σ' respectively. Peer configurations, stores and the evaluation context are defined in Definition 2.

Definition 2 (Evaluation Context).

Configurations	$\Omega ::= \overline{\pi_1 : \mathsf{p}}; \overline{\pi_2 \rightarrowtail \pi_3}$
Stores	$\sigma ::= \cdot \mid \sigma, \ell_{\mathsf{T}}\ \mathsf{at}\ \pi^{\mathsf{p}} = \mathsf{e} \mid \sigma, \ell_{\mathsf{T}}\ \mathsf{at}\ \pi^{\mathsf{p}} = \bot$
Eval. Contexts	$\mathsf{E} ::= [] \mid \mathsf{E}\ \mathsf{e} \mid \mathsf{v}\ \mathsf{E} \mid \mathsf{peerof}\ \mathsf{E} \mid \mathsf{broad}_{\mathsf{P}}\ \mathsf{E} \mid \mathsf{E}\ \mathsf{match}\ \{\overline{\mathsf{x}\ \mathsf{at}\ \mathsf{p} \Rightarrow \mathsf{e}}\}$ $\mid \mathsf{E}\ \mathsf{match}\ \{\overline{\mathsf{x}\ \mathsf{is}\ \mathsf{p} \Rightarrow \mathsf{e}}\} \mid \mathsf{E}\ \mathsf{deref}\ \{\mathsf{x} \Rightarrow \mathsf{e}_1;\ \Rightarrow \mathsf{e}_2\}$ $\mid \mathsf{on}(\mathsf{E}).\mathsf{capture}(\overline{\mathsf{x} = \mathsf{e}_1})\ \{\ \mathsf{ref}_{\mathsf{T}}\ \mathsf{e}_2\ \}$ $\mid \mathsf{on}(\mathsf{v}_1).\mathsf{capture}(\overline{\mathsf{x}_1 = \mathsf{v}_2}, \mathsf{x}_2 = \mathsf{E}, \overline{\mathsf{x}_3 = \mathsf{e}_1})\ \{\ \mathsf{ref}_{\mathsf{T}}\ \mathsf{e}_2\ \}$

A peer configuration Ω is a collection of peer instances with their type and a list connected peer instances. A store σ is a partial function from placed labels to expressions. The evaluation context dictates a deterministic, left-to-right, eager evaluation strategy.

We present the evaluation rules in three parts: Definition 3 presents a deterministic and sequential execution model; Definition 4 adds two rules that introduce non-determinism to model concurrent execution; Definition 5 adds a rule that allows the configuration Ω to change.

Definition 3 (Evaluation Rules).

(E-App) $\Omega; \sigma \triangleright (\lambda \mathsf{x} : \mathsf{T}.\ \mathsf{e})\ \mathsf{v} \to^{\pi_1} \Omega; \sigma \triangleright \mathsf{e}[\mathsf{x}/\mathsf{e}]$ (E-Peerof) $\Omega; \sigma \triangleright \mathsf{peerof}\ \ell_{\mathsf{T}}\ \mathsf{at}\ \pi_{\mathsf{P}}^{\mathsf{p}} \to^{\pi'} \Omega; \sigma \triangleright \pi_{\mathsf{P}}^{\mathsf{p}}$

[4] A value exists if the remote reference points to a value and not to nowhere. A reference that points nowhere will point to the bottom value as described in Definition 2.

$$\text{(E-Broad)}\ \Omega;\sigma \triangleright \mathsf{broad}_{\mathsf{P}_2}\ \ell_\mathsf{T}\ \text{at}\ \pi^\mathsf{p}_{\mathsf{P}_1} \rightarrow^{\pi'} \Omega;\sigma \triangleright \ell_\mathsf{T}\ \text{at}\ \pi^\mathsf{p}_{\mathsf{P}_2} \qquad \text{(E-Context)}\ \frac{\Omega;\sigma \triangleright e \rightarrow^{\pi} \Omega';\sigma' \triangleright e'}{\Omega;\sigma \triangleright E[e] \rightarrow^{\pi} \Omega';\sigma' \triangleright E[e']}$$

$$\text{(E-Match)}\ \frac{p_1 = p_{2,i}}{\Omega;\sigma \triangleright \ell_\mathsf{T}\ \text{at}\ \pi^{p_1}_\mathsf{P}\ \mathsf{match}\ \{\overline{x\ \text{at}\ p_2 \Rightarrow e}\} \rightarrow^{\pi'} \Omega;\sigma \triangleright e_i[\ell_\mathsf{T}\ \text{at}\ \pi^{p_1}_{p_1}/x_i]}$$

$$\text{(E-PMatch)}\ \frac{p_1 = p_{2,i}}{\Omega;\sigma \triangleright \pi^{p_1}_\mathsf{P}\ \mathsf{match}\ \{\overline{x\ \text{is}\ p_2 \Rightarrow e}\} \rightarrow^{\pi'} \Omega;\sigma \triangleright e_i[\pi^{p_1}_{p_1}/x_i]}$$

$$\text{(E-Deref-OK)}\ \frac{(\pi = \pi' \ \vee\ \pi' \rightarrowtail \pi \in \Omega) \qquad \sigma(\ell_\mathsf{T}\ \text{at}\ \pi^\mathsf{p}) = v}{\Omega;\sigma \triangleright \ell_\mathsf{T}\ \text{at}\ \pi^\mathsf{p}_\mathsf{P}\ \mathsf{deref}\ \{x \Rightarrow e_1;\ \Rightarrow e_2\} \rightarrow^{\pi'} \Omega;\sigma \triangleright e_1[v/x]}$$

$$\text{(E-Deref-Err)}\ \frac{(\pi' \rightarrowtail \pi \notin \Omega \quad \vee \quad \sigma(\ell_\mathsf{T}\ \text{at}\ \pi^\mathsf{p}) = \bot \quad \vee \quad \ell_\mathsf{T}\ \text{at}\ \pi^\mathsf{p} \notin \mathrm{dom}\ \sigma)}{\Omega;\sigma \triangleright \ell_\mathsf{T}\ \text{at}\ \pi^\mathsf{p}_\mathsf{P}\ \mathsf{deref}\ \{x \Rightarrow e_1;\ \Rightarrow e_2\} \rightarrow^{\pi'} \Omega;\sigma \triangleright e_2}$$

$$\text{(E-Rem-Move)}\ \frac{\pi' \rightarrowtail \pi \in \Omega}{\Omega;\sigma \triangleright \mathsf{on}(\pi^\mathsf{p}_\mathsf{P}).\mathsf{capture}(\overline{x = v})\ \{\ \mathsf{ref}_\mathsf{T}\ e\ \} \rightarrow^{\pi'} \Omega;\sigma \triangleright \mathsf{on}(\pi^\mathsf{p}_\mathsf{P}).\mathsf{capture}()\ \{\ \mathsf{ref}_\mathsf{T}\ e[\overline{v/x}]\ \}}$$

$$\text{(E-Rem-Eval)}\ \frac{\pi' \rightarrowtail \pi \in \Omega \qquad \Omega;\sigma \triangleright e \rightarrow^{\pi} \Omega';\sigma' \triangleright e'}{\Omega;\sigma \triangleright \mathsf{on}(\pi^\mathsf{p}_\mathsf{P}).\mathsf{capture}()\ \{\ \mathsf{ref}_\mathsf{T}\ e\ \} \rightarrow^{\pi'} \Omega;\sigma \triangleright \mathsf{on}(\pi^\mathsf{p}_\mathsf{P}).\mathsf{capture}()\ \{\ \mathsf{ref}_\mathsf{T}\ e'\ \}}$$

$$\text{(E-Rem-OK)}\ \frac{\pi' \rightarrowtail \pi \in \Omega \qquad \ell_\mathsf{T}\ \text{at}\ \pi^\mathsf{p} \notin \sigma}{\Omega;\sigma \triangleright \mathsf{on}(\pi^\mathsf{p}_\mathsf{P}).\mathsf{capture}()\ \{\ \mathsf{ref}_\mathsf{T}\ v\ \} \rightarrow^{\pi'} \Omega;\sigma, \ell_\mathsf{T}\ \text{at}\ \pi^\mathsf{p} = v \triangleright \ell_\mathsf{T}\ \text{at}\ \pi^\mathsf{p}_\mathsf{P}}$$

$$\text{(E-Rem-Err)}\ \frac{\pi' \rightarrowtail \pi \notin \Omega \qquad \ell_\mathsf{T}\ \text{at}\ \pi^\mathsf{p} \notin \sigma}{\Omega;\sigma \triangleright \mathsf{on}(\pi^\mathsf{p}_\mathsf{P}).\mathsf{capture}(\overline{x = v})\ \{\ \mathsf{ref}_\mathsf{T}\ e\ \} \rightarrow^{\pi'} \Omega;\sigma, \ell_\mathsf{T}\ \text{at}\ \pi^\mathsf{p} = \bot \triangleright \ell_\mathsf{T}\ \text{at}\ \pi^\mathsf{p}_\mathsf{P}}$$

Definition 4 (Concurrent Evaluation Rules).

$$\text{(E-Conc-Rem)}\ \frac{\pi' \rightarrowtail \pi \in \Omega \qquad \ell_\mathsf{T}\ \text{at}\ \pi^\mathsf{p} \notin \sigma}{\Omega;\sigma \triangleright \mathsf{on}(\pi^\mathsf{p}_\mathsf{P}).\mathsf{capture}()\ \{\ \mathsf{ref}_\mathsf{T}\ e\ \} \rightarrow^{\pi'} \Omega;\sigma, \ell_\mathsf{T}\ \text{at}\ \pi^\mathsf{p} = e \triangleright \ell_\mathsf{T}\ \text{at}\ \pi^\mathsf{p}_\mathsf{P}}$$

$$\text{(E-Conc-Lab)}\ \frac{\Omega;\sigma \triangleright e_1 \rightarrow^{\pi} \Omega';\sigma' \triangleright e_1' \qquad \ell_\mathsf{T}\ \text{at}\ \pi^\mathsf{p} \notin e_1'}{\Omega;\sigma, \ell_\mathsf{T}\ \text{at}\ \pi^\mathsf{p} = e_1 \triangleright e_2 \rightarrow^{\pi'} \Omega';\sigma', \ell_\mathsf{T}\ \text{at}\ \pi^\mathsf{p} = e_1' \triangleright e_2}$$

Definition 5 (Distributed Evaluation Rules).

$$\text{(E-Dist)}\ \Omega_t;\sigma \triangleright e \rightarrow^{\pi} \Omega_{t+1};\sigma \triangleright e$$

We describe each rule briefly: E-App and E-Context are standard; E-Peerof extracts the peer instance from a remote reference; E-Broad broadens a peer's placement annotation; E-Match and E-PMatch picks the branch of a match expression whose placement or peer (respectively) is identical to the one in the remote reference or peer (respectively); E-Deref-OK dereferences a reference when a connection exists; E-Deref-Err executes the fail branch of the dereference when a connection does not exist; E-Rem-Move moves all the captured values, by copying them, to the remote peer instance; E-Rem-Eval reduces a remote expression after moving all captured expressions, this rule is expected to be applied over and over until the remote expression evaluates to a value unless E-Conc-Rem is applied to break this chain, in which case it will resume with a series of applications of E-Conc-Lab; E-Rem-OK creates a new entry in the store to the remote expression when it is a value and reduces to a remote reference to the store entry only if a connection to the remote peer instance exists; E-Rem-Err applies whenever the connection to a remote peer does not exist and returns a reference that points to $\bot$; E-Conc-Rem allows to bypass E-Rem-Eval to use E-Rem-OK by allowing the remote reference to point to an expression instead of a value, this

does not mean that it is possible to dereference an unevaluated expression as E-Deref-OK and E-Deref-Err still require the reference to point to a value, or bottom, for a dereference to be possible; E-Conc-Lab allows the reduction of an expression in the codomain of a store; E-Dist allows to step the evaluation configuration through an arbitrary specified sequence of configurations.

6.3 Typing Rules

The typing rules take the form $\mathcal{A}; \Gamma \vdash e : T \text{ on } p$ and express that e has type T and is placed on p under the architecture $\mathcal{A}$ and the typing context Γ. The architecture $\mathcal{A}$ and the typing context Γ are defined in Definition 6. An architecture is a collection of peer types and tie declarations between them. We assume that all peer types used in tie declarations belong to the peer type collection. Ties are of two kinds (1) singles: indicated by $\mapsto^1$ and (2) multiples: indicated by $\mapsto^*$. When the multiplicity of a tie is irrelevant we use $\mapsto$. A typing context is a binding of variables to their types and the peer type on which they are placed.

Definition 6 (Typing Context).

$$\text{Architectures } \mathcal{A} ::= \overline{p_1}; \overline{p_2 \mapsto^1 p_3}; \overline{p_4 \mapsto^* p_5} \qquad \text{Typing Contexts } \Gamma ::= \overline{x : T \text{ on } p}$$

Definition 7 (Typing Rules).

$$\text{(T-Var)}\ \frac{x : T \text{ on } p \in \Gamma}{\mathcal{A}; \Gamma \vdash x : T \text{ on } p} \qquad \text{(T-Abs)}\ \frac{\mathcal{A}; \Gamma, x : T_1 \text{ on } p \vdash e : T_2 \text{ on } p}{\mathcal{A}; \Gamma \vdash \lambda x : T_1.\ e : T_1 \to T_2 \text{ on } p} \qquad \text{(T-App)}\ \frac{\mathcal{A}; \Gamma \vdash e_1 : T_1 \to T_2 \text{ on } p \quad \mathcal{A}; \Gamma \vdash e_2 : T_1 \text{ on } p}{\mathcal{A}; \Gamma \vdash e_1\ e_2 : T_2 \text{ on } p}$$

$$\text{(T-Peer)}\ \frac{p_1 \in P}{\mathcal{A}; \Gamma \vdash \pi_P^{p_1} : \text{Peer } P \text{ on } p_2} \qquad \text{(T-Label)}\ \frac{p_1 \in P}{\mathcal{A}; \Gamma \vdash \ell_T \text{ at } \pi_P^{p_1} : T \text{ at } P \text{ on } p_2}$$

$$\text{(T-Peerof)}\ \frac{\mathcal{A}; \Gamma \vdash e : T \text{ at } P \text{ on } p}{\mathcal{A}; \Gamma \vdash \text{peerof } e : \text{Peer } P \text{ on } p} \qquad \text{(T-Broad)}\ \frac{\mathcal{A}; \Gamma \vdash e : T \text{ at } P_1 \text{ on } p \quad P_1 \subseteq P_2}{\mathcal{A}; \Gamma \vdash \text{broad}_{P_2}\ e : T \text{ at } P_2 \text{ on } p}$$

$$\text{(T-Match)}\ \frac{\overline{\mathcal{A}; \Gamma, x : T_1 \text{ at } p_2 \text{ on } p_1 \vdash e_2 : T_2 \text{ on } p_1} \quad \mathcal{A}; \Gamma \vdash e_1 : T_1 \text{ at } P \text{ on } p_1 \quad P = \overline{p_2} \quad \forall i \neq j.\ p_{2,i} \neq p_{2,j}}{\mathcal{A}; \Gamma \vdash e_1 \text{ match } \{\overline{x \text{ at } p_2 \Rightarrow e_2}\} : T_2 \text{ on } p_1}$$

$$\text{(T-PMatch)}\ \frac{\overline{\mathcal{A}; \Gamma, x : \text{Peer } p_2 \text{ on } p_1 \vdash e_2 : T_2 \text{ on } p_1} \quad \mathcal{A}; \Gamma \vdash e_1 : \text{Peer } \overline{p_2} \text{ on } p_1 \quad \forall i \neq j.\ p_{2,i} \neq p_{2,j}}{\mathcal{A}; \Gamma \vdash e_1 \text{ match } \{\overline{x \text{ is } p_2 \Rightarrow e_2}\} : T_2 \text{ on } p_1}$$

$$\text{(T-Remote)}\ \frac{\mathcal{A}; \Gamma \vdash e_1 : \text{Peer } \overline{p_2} \text{ on } p_1 \quad \overline{\mathcal{A}; \Gamma \vdash e_2 : T_2 \text{ on } p_1} \quad \overline{\text{mobile } T_2} \quad \overline{p_1 \mapsto p_2 \in \mathcal{A}} \quad \mathcal{A}; \overline{x : T_2 \text{ on } p_2} \vdash e_3 : T_1 \text{ on } p_2}{\mathcal{A}; \Gamma \vdash \text{on}(e_1).\text{capture}(\overline{x = e_2})\ \{\ \text{ref}_{T_1}\ e_3\ \} : T_1 \text{ at } \overline{p_2} \text{ on } p_1}$$

$$\text{(T-Deref)}\ \frac{\mathcal{A}; \Gamma \vdash e_1 : T_1 \text{ at } \overline{p_2} \text{ on } p_1 \quad \overline{p_1 \mapsto p_2 \in \mathcal{A}} \quad \text{mobile } T_1 \quad \mathcal{A}; \Gamma, x : T_1 \text{ on } p_1 \vdash e_2 : T_2 \text{ on } p_1 \quad \mathcal{A}; \Gamma \vdash e_3 : T_2 \text{ on } p_1}{\mathcal{A}; \Gamma \vdash e_1 \text{ deref } \{x \Rightarrow e_2;\ \Rightarrow e_3\} : T_2 \text{ on } p_1}$$

We describe briefly the rules: T-Var, T-Abs, and T-App are standard rules extended with peer placement – notice that a function call type checks when both the function and its argument are placed on the same peer type; T-Peer and T-Label enforce that a peer instance's actual type belongs to its peer type union annotation; T-Peerof extracts the peer type from a remote reference type; T-Broad broadens the placement of a remote reference if and only if no previous placement is lost; T-Match enforce that the branches are exhaustive and each handles a unique placement type and all branches have the same type; T-PMatch similarly allows pattern matching on peer instances based on their type; T-Remote allows a remote execution when a tie, regardless of its multiplicity, to all the peers in a union placement type exists, when the remote expression typechecks under the captured expressions, and when the type of those expressions is mobile; T-Deref enforce that dereferencing only applies when a tie exists and the data type is mobile. A mobile type is defined to be any type that is not a function.

Definition 8 (Mobile Types). (M-Base) mobile B (M-Ref) mobile T at P (M-Peer) mobile Peer P

6.4 Type Soundness

This section lists the properties that the core calculus enjoys, which we prove for the complete concurrent and distributed extension of the calculus, i.e., for the evaluation rules in Definitions 3 to 5.

The first major result is type soundness. The calculus is type-sound given the following well-formedness conditions for configurations and the (usual) well typedness condition for stores. Definition 9 states that an evaluation configuration satisfies an architecture when all peer instances have a known peer type and that single ties are respected. Definition 10 defines a consistent sequence of evaluation configurations to be one where peer instances never change types. Definition 11 defines a configuration to be coherent with any expression, including those in the domain and codomain of a store, when all the literal peer instances in those expressions are annotated with the known peer type in the configuration. Definition 12 defines a store to be well typed when all the expressions in its codomain are well typed.

Definition 9 (Architecture Satisfaction). A configuration Ω satisfies an architecture $\mathcal{A}$, written $\Omega \vDash \mathcal{A}$, when $\mathsf{p} \in \mathcal{A}$ for every $\pi : \mathsf{p} \in \Omega$. And if $\mathsf{p}_1 \mapsto^1 \mathsf{p}_2 \in \mathcal{A}$ then for every $\pi : \mathsf{p}_1 \in \Omega$ there exists a *unique* $\pi' : \mathsf{p}_2 \in \Omega$ such that $\pi \rightarrowtail \pi' \in \Omega$.

Definition 10 (Consistent Configurations). A sequence $\{\Omega_t\}_{t \in \mathbb{N}}$ is consistent when for every $\pi : \mathsf{p} \in \Omega_{t_0}$ and $\mathsf{t} > \mathsf{t}_0$, if $\pi : \mathsf{p}' \in \Omega_t$ then $\mathsf{p}' = \mathsf{p}$.

Definition 11 (Coherent Configuration). A sequence $\{\Omega_t\}_{t \in \mathbb{N}}$ is coherent with an expression e and a store σ when: (1) for every sub-expression of e of the form $\pi^{\mathsf{p}}_{\mathsf{p}}$ there exists a $t \in \mathbb{N}$ such that $\pi : \mathsf{p} \in \Omega_t$, (2) for every ℓ_{T} at $\pi^{\mathsf{p}} \in \mathrm{dom}\ \sigma$ then there exists a $t \in \mathbb{N}$ such that $\pi : \mathsf{p} \in \Omega_t$, and (3) for every $\mathsf{e} \in \mathrm{codom}\ \sigma$ then $\{\Omega_t\}_{t \in \mathbb{N}}$ is coherent with e.

Definition 12 (Well typed Store). A store σ is well typed under $\mathcal{A}; \Gamma$ when there exists a type T and peer type p such that $\mathcal{A}; \Gamma \vdash \mathsf{e} : \mathsf{T}$ on p for every $\mathsf{e} \in \mathrm{codom}\ \sigma$.

The usual progress and preservation theorems (Theorems 1 and 2) hold under the assumption of a coherent and consistent configuration and a well typed context as defined in Definitions 9 to 12. Theorem 1 prevents Ω from changing to rule out trivial progress by only applying E-Dist repeatedly without reducing the term.

Theorem 1 (Type Progress). *Let $\Omega \vDash \mathcal{A}$ and $\pi : \mathsf{p} \in \Omega$ such that Ω is picked from a consistent and coherent sequence of configurations with e and a well typed store σ under $\mathcal{A}; \cdot$ be given. If $\mathcal{A}; \cdot \vdash \mathsf{e} : \mathsf{T}$ on p then e is a value, or there exists e' such that $\Omega; \sigma \triangleright \mathsf{e} \rightarrow^{\pi} \Omega; \sigma' \triangleright \mathsf{e}'$.*

Theorem 2 (Type Preservation). *Let* $\Omega \vDash \mathcal{A}$ *and* $\pi : \mathsf{p} \in \Omega$ *such that* Ω *is picked from a consistent and coherent sequence of configurations with* e *and a well typed store* σ *under* $\mathcal{A}; \Gamma$ *be given. If* $\mathcal{A}; \Gamma \vdash \mathsf{e} : \mathsf{T}$ on p *and* $\Omega; \sigma \triangleright \mathsf{e} \rightarrow^{\pi} \Omega'; \sigma' \triangleright \mathsf{e}'$ *then* $\mathcal{A}; \Gamma \vdash \mathsf{e}' : \mathsf{T}$ on p.

Whereas these two theorems are usually enough to prove type safety, in our system we must prove two additional lemmas that allow the inductive proof to be sound [Wright and Felleisen 1994]. Lemma 1 states a well typed store remains well typed after one application of the evaluation rule as long as the expression being reduced is well typed. And similarly, Lemma 2 states that a coherent configuration remains coherent. We do not need a consistency preservation theorem since consistency neither depends on expressions nor on stores: in fact, from Definition 10, it is clear that if a sequence $\{\Omega_t\}_{t \geq t_0}$ is consistent then $\{\Omega_t\}_{t \geq t_1}$ is still consistent when $t_1 \geq t_0$.

Lemma 1 (Store well typeness Preservation). *Let* $\Omega \vDash \mathcal{A}$ *and* $\pi : \mathsf{p} \in \Omega$ *such that* Ω *is picked from a consistent and coherent sequence of configurations with* e *and a well typed store* σ *under* $\mathcal{A}; \Gamma$ *be given. If* $\mathcal{A}; \Gamma \vdash \mathsf{e} : \mathsf{T}$ on p *and* $\Omega; \sigma \triangleright \mathsf{e} \rightarrow^{\pi} \Omega; \sigma' \triangleright \mathsf{e}'$ *then* σ' *is well typed under* $\mathcal{A}; \Gamma$.

Lemma 2 (Configuration Coherence Preservation). *Let* $\Omega \vDash \mathcal{A}$ *and* $\pi : \mathsf{p} \in \Omega$ *such that* Ω *is picked from a consistent and coherent sequence of configurations with* e *and a well typed store* σ *under* $\mathcal{A}; \cdot$ *be given. If* $\mathcal{A}; \cdot \vdash \mathsf{e} : \mathsf{T}$ on p *and* $\Omega; \sigma \triangleright \mathsf{e} \rightarrow^{\pi} \Omega; \sigma' \triangleright \mathsf{e}'$ *then* Ω *is coherent with* e' *and* σ'.

6.5 Properties of Dynamic Placements and Architectural Soundness

In our calculus, it is not possible to modify the value a remote reference points to once it is created. Moreover it is not possible to accidentally refer to a reference that was not yet created. Thus the dependency graph of remote references is acyclical. Theorem 3 is the formal expression of these observations and Definition 13 defines an acyclical store.

Definition 13 (Non-Circular Store). The empty store $\cdot$ is non-circular. The store $\sigma, \ell_{\mathsf{T}}$ at $\pi^{\mathsf{p}} = \bot$ is non-circular when σ is non-circular. And the store $\sigma, \ell_{\mathsf{T}_1}$ at $\pi^{\mathsf{p}_1} = \mathsf{e}$ is non-circular when σ is non-circular and for every sub-expression of e of the form ℓ_{T_2} at $\pi^{\mathsf{p}_2}_{\mathsf{P}}$ then ℓ_{T_2} at $\pi^{\mathsf{p}_2} \in \mathrm{dom}\ \sigma$.

Theorem 3 (Non-Circular Store). *Let* $\Omega \vDash \mathcal{A}$ *and* $\pi : \mathsf{p} \in \Omega$ *such that* Ω *is picked from a consistent and coherent sequence of configurations with* e *and a well typed store* σ *under* $\mathcal{A}; \cdot$ *be given. If* $\mathcal{A}; \cdot \vdash \mathsf{e} : \mathsf{T}$ on p, *and* e *does not contain any sub-expression of the form* ℓ_{T} at $\pi^{\mathsf{p}}_{\mathsf{P}}$, *and* $\Omega; \cdot \triangleright \mathsf{e} \twoheadrightarrow^{\pi} \Omega; \sigma \triangleright \mathsf{e}'$ *then* σ *is non-circular.*

A design goal for the calculus is that computations must only use locally available resources. If a computation requires a remote value then it must explicitly dereference it. Theorem 4 proves that this goal was achieved. It states that, if an expression type checks under a context, then stripping away all non-local variables from that context, Definition 14, will not affect type checking.

Definition 14 (Restricted Γ*).* If Γ is a typing context and p is a peer type then $\Gamma|_{\mathsf{p}}$ is the sub-context of Γ that only contains the bindings on p.

Theorem 4 (Local Placement). *If* $\mathcal{A}; \Gamma \vdash \mathsf{e} : \mathsf{T}$ on p *then* $\mathcal{A}; \Gamma|_{\mathsf{p}} \vdash \mathsf{e} : \mathsf{T}$ on p.

Another design goal of the calculus is for the placement in the type of references to accurately reflect the placement of the actual remote reference. Under dynamic placement the placement types in the reference type must include the actual placement type of the reference. Corollary 1, which follows directly from type safety, shows that this property holds.

Corollary 1 (Correct Reference Placement). *Let* $\Omega \vDash \mathcal{A}$ *and* $\pi : \mathsf{p} \in \Omega$ *such that* Ω *is picked from a consistent and coherent sequence of configurations with* e *and a well typed store* σ *under* $\mathcal{A}; \Gamma$ *be given. If* $\mathcal{A}; \Gamma \vdash \mathsf{e} : \mathsf{T}$ at $\overline{\mathsf{p}_2}$ on p_1 *and* $\Omega; \cdot \triangleright \mathsf{e} \twoheadrightarrow^{\pi} \Omega'; \sigma \triangleright \ell_{\mathsf{T}}$ at $\pi^{\mathsf{p}_3}_{\mathsf{P}}$ *then* $\mathsf{P} = \overline{\mathsf{p}_2}$ *and there exists* i *such that* $\mathsf{p}_3 = \mathsf{p}_{2,\mathsf{i}}$.

Finally, we state a theorem that formalizes our intuition about architectural ties: if a tie is not declared in the architecture then, even if the configuration includes such a connection, it cannot be used to access values remotely. Definition 15 allows us to restrict a configuration under two peer instances by disconnecting them directly or disconnecting any proxy peer that can be used to relay values. Theorem 5 states that, under a restricted configuration extended with a connection between two peer instances whose types are not declared to be tied, references on the first instance cannot affect a computation happening on the second instance.

To formalize the intuition of this non-interaction, we define in Definition 16 "disagreeing stores" to be two stores that agree on values on the different peers but disagree on at least one value's placement on a particular peer. Then the architectural soundness theorem states that under disagreeing states the evaluation of an expression does not change.

Definition 15 (Restricted Ω). Let Ω and $\pi : \mathsf{p}, \pi' : \mathsf{p}' \in \Omega$ be given. We denote by $\Omega_{/\pi \rightarrowtail \pi'}$ any sub-configuration of Ω that satisfies: (1) for every $\pi'' : \mathsf{p}'' \in \Omega$ then $\pi'' : \mathsf{p}'' \in \Omega_{/\pi \rightarrowtail \pi'}$, and (2) for every sequence $\pi_1 \rightarrowtail \pi_2, \cdots, \pi_n \rightarrowtail \pi_{n+1} \in \Omega$ such that $\pi_1 = \pi$ and $\pi_{n+1} = \pi'$ then there exists i such that $\pi_i \rightarrowtail \pi_{i+1} \notin \Omega_{/\pi \rightarrowtail \pi'}$.

Definition 16 (Disagreeing Stores). Two stores σ_1 and σ_2 are said to disagree on a place type p_1 if (1) for every ℓ_T at $\pi^{\mathsf{p}_2} \in \mathrm{dom}\ \sigma_1 \cup \mathrm{dom}\ \sigma_2$ such that $\mathsf{p}_1 \neq \mathsf{p}_2$ then $\sigma_1(\ell_T \text{ at } \pi^{\mathsf{p}_2}) = \sigma_2(\ell_T \text{ at } \pi^{\mathsf{p}_2})$, and (2) there exists ℓ_T at π^{p_1} such that $\sigma_1(\ell_T \text{ at } \pi^{\mathsf{p}_1}) \neq \sigma_2(\ell_T \text{ at } \pi^{\mathsf{p}_1})$.

Theorem 5 (Architectural Soundness). *Let $\Omega \vDash \mathcal{A}$ and $\pi : \mathsf{p} \in \Omega$ such that Ω is picked from a consistent and coherent sequence of configurations with e and two well typed stores σ_1, σ_2 under $\mathcal{A}; \Gamma$ be given. If $\mathcal{A}; \Gamma \vdash \mathsf{e} : \mathsf{T}$ on p_1 and $\pi_1 : \mathsf{p}_1 \in \Omega$ and $\pi_2 : \mathsf{p}_2 \in \Omega$ and $\pi_1 \neq \pi_2$ and $\mathsf{p}_1 \mapsto \mathsf{p}_2 \notin \mathcal{A}$ and σ_1, σ_2 only disagree on p_2, and $\Omega_{/\pi_1 \rightarrowtail \pi_2}, \pi_1 \rightarrowtail \pi_2; \sigma_1 \triangleright \mathsf{e} \rightarrow^{\pi_1} \Omega'; \sigma_1' \triangleright \mathsf{e}_1$, and $\Omega_{/\pi_1 \rightarrowtail \pi_2}, \pi_1 \rightarrowtail \pi_2; \sigma_2 \triangleright \mathsf{e} \rightarrow^{\pi_1} \Omega'; \sigma_2' \triangleright \mathsf{e}_2$ then $\mathsf{e}_1 = \mathsf{e}_2$.*

7 IMPLEMENTATION

Dyno is implemented on top of ScalaLoci [Weisenburger et al. 2018] which itself is an extension to Scala and the Scala type system. We chose ScalaLoci since its type system already allows expressing placement within types. This placement, however, is only static and not first-class, i.e., one cannot express first-class values of type `Int on DB` or values of type `Int at (DB | Cache)`. Recall from Section 6 the difference between `on` and `at`: the former is a type annotation only used during type checking and exists only in the typing judgement, whereas `at` is a type constructor that has an introduction and elimination form.

Dyno's type-checker is a Scala macro that is executed at compile-time and transforms Scala ASTs. Dyno implements two notable extensions: First, we give every peer instance a GUID, which is a unique identifier of type `java.util.UUID` as provided by the Java standard library to distinguish different instances. This is analogous to the π_p expression with the peer type p. Second, we introduce a `Ref[T]` class which wraps two GUIDs: one for the peer instance and one for the value. The class exposes a dereferencing method that moves the value from the peer instance whose GUIDs are identical to the wrapped GUIDs to the local instance. Since this operation could fail, the method returns an option. This is equivalent to returning a nullable reference in the formalism through E-Rem-OK and E-Rem-Err (note that deref checks the validity of the reference through E-Deref-OK and E-Deref-Err, hence nullable references are safe). Thus, Dyno associates GUIDs to every remote value reference that is created by `remote ref`. These GUIDs are recorded in a map that is not exposed to the developer and only used internally by the `remote ref` and `deref` constructs. This map is thus equivalent to a slice of σ whose domain are all references created at the instance.

8 EVALUATION

The main hypothesis of Dyno's design is that union types to represent dynamic placement (1) reduce the complexity of handling dynamic placement decisions, (2) increase type safety and (3) come at negligible cost to the run time performance.

We evaluate these hypotheses with side-by-side comparisons of variants of the same applications (Section 8.1), benchmarking different variants (Section 8.2) and applying Dyno to a large open-source application (Section 8.3). Finally, we provide an intuition of to what extend developers have to deal with dynamic placement by mining the F-Droid open-source app repository [Gultnieks 2010] (Section 8.4). In such study, we check for the specific dynamic placement approach used in the case study of Section 8.3, showing that this approach is not uncommon and providing a lower bound for the number of critical run time checks related to dynamic placement in the repository.

8.1 Analysis of Variants: Dynamic Placement in Dyno vs. Java RMI vs. Akka Typed

We validate our first two hypotheses with side-by-side comparisons of alternative designs of applications from different domains. The case studies are full versions of the minimal examples in Section 4. We compare Dyno implementations against functionally equivalent variants implemented in (i) Java RMI [Downing 1998] – supported on the JVM out of the box – (ii) Akka Typed [Lightbend Inc 2020] as a state-of-the-art distributed actor system, and (iii) ScalaLoci [Weisenburger et al. 2018] which supports placement types but not dynamic placement. We first shortly describe each case study. Then we describe the issues with existing approaches and quantify their impact on the code base.

- *Unified Feed of Twitter and Mastodon Messages.* This case study merges the message feed from two sources into a single list of messages (similar to Section 4.1). One source retrieves the messages from Twitter tweets, the other from Mastodon [Mastodon 2016], a Twitter-like application. A client peer interacts with both sources for displaying the messages to the user. The client only needs to fetch the elements in the distributed list that it displays without pulling the entire message history.
- *Web Sessions.* This case study is a multi-user administrative panel, implemented as a stateful request–response application (similar to Section 4.2) that spans over the administrators' client driving the interaction, a server peer, an authentication service peer, and an administration service peer. We use placed closures for the callbacks that are executed when the user "follows" the links in the application.
- *Resource Management.* This case study controls a users' access to resources. Users can unlock resources if their keys are stored in a trusted key database (similar to Section 4.3). In this case study, the user can further ask a supervisor peer to trust a public key. If the supervisor agrees, the key is added to the trusted key database.

We categorize the issues that we faced in the case studies into three groups, described in the following paragraphs. For each, we quantify how much Dyno improves safety, identifying the code locations in Java RMI and Akka Typed that access a dynamically placed value (Table 1): creation of remote references ("Creation" column), acquisition of references that were created remotely ("Acquisition" column) and dereference operations ("Access" column). In all variants, when accessing remote data, the value's data type is statically checked. Yet, static checks in the Java RMI and Akka Typed do not cover a value's placement.

Placement correctness. In both Java RMI and Akka Typed, one can accidentally swap the correct placement with a wrong one. For instance, in the first case study, both the Twitter and Mastodon peers hold a sorted `TwitterFeed` and a `MastodonFeed` object for messages. Hence, it is easy to have

the Twitter peer construct a `MastodonFeed` and vice-versa. Similarly, in RMI, the reference to the callback in the second case study can be accidentally moved to the client, transferring a secret over the network. The type system does not prevent these errors because the local object and the remote object of another place have identical type, i.e., remote access looks like local access. In Akka, invoking a remote function and retrieving its result entails separate messages, which requires maintaining the state to keep track of their correlation. Hence, the type system does not help the developer reasoning statically on placement.

Table 1. Code metrics for the case studies.

Case Study	Version	Creation	Acquisition	Access	LOC
Feeds	RMI	3	1	1	148
	Akka	3	1	2	173
	ScalaLoci	4	5	3	178
	Dyno	3	4	2	147
Sessions	RMI	9	3	3	154
	Akka	7	9	14	132
	ScalaLoci	17	26	17	176
	Dyno	14	23	14	125
Resources	RMI	2	4	5	151
	Akka	2	6	4	207
	ScalaLoci	3	6	4	115
	Dyno	2	5	3	98

The ScalaLoci implementations do maintain placement correctness since they are implemented in the style outlined in Section 5. The disadvantage of this style is that it sacrifices modularity which required us to introduce code duplication in multiple places – for each placement type in fact – which is reflected in the inflated number of lines of code ("LOC" column).

Dyno does not suffer from such issues. First, remote references to data and the data they point to have different types. As a result, one cannot accidentally fetch data from another peer without explicitly moving the execution there. Second, creating a remote reference is guaranteed to have the correct placement. Since placement correctness crucially relies on attaching the correct placement when creating a reference, the reference creation count is a measure of critical spots in the code where a reference – or a reference-like object, such as the keys of a manually managed reference table – are created ("Creation" column in Table 1). In such spots in Akka and RMI, as placement is manually encoded, one can use the wrong placement, i.e., placement correctness could be violated. Whereas the number of spots where remote references (or encodings of the same) are created does not differ widely across case studies, the reported numbers quantify the spots were Dyno features improved safety, confirming our hypothesis.

Architectural safety. Architectures can be enforced statically to some degree with Akka Typed, but not with RMI – since the type system is completely oblivious to placement. In RMI variants of our case studies, a registry acts as a centralized component that all peers connect to and it is used to store and move references. Therefore, any reference in the registry can be read by any peer who has access. In the Akka implementation of the second case study, for example, we need to manually maintain a table with references that remote actors can access with a message that requests a referenced value. Such messages may include the actor reference to reply to, but there is no guarantee that the correct actor reference is actually used. Also, a message cannot enforce a specific place, neither for the sending actor, nor for the receiving one. The implication on dynamic placement is that there cannot be strict placement guarantees – static or dynamic. With messages that can be relayed and actor references that can be ignored, encoding placement is at best conventionally agreed upon.

In Dyno, there is no central registry for references. Instead, each peer stores its own references and access to them is subject to architectural checks. Further, Dyno relieves developers from maintaining a reference table manually – which is error-prone.

As Dyno performs architectural checks to verify the validity of accesses to referenced values, we report the number of dereference operations ("Access" column in Table 1). These places may harbor violation to the architectural safety: a peer with no tie to another may try to dereference a reference

pointing to a value on the latter. The reference acquisition count measures the places where a remote reference has been acquired by a peer without necessarily dereferencing it ("Acquisition" column). While the number of accesses to remote references does not vary much across variants, the numbers indicate the spots were Dyno provides static safety checks that RMI and Akka do not. Both Dyno and ScalaLoci contain placement types and statically-enforced architectural ties thus architectural safety is guaranteed in both.

Communication boilerplate. Handling dynamic placement manually induces overhead and increases accidental complexity that is not part of the application logic. For example, in Akka Typed, decoupled send and receive operations may require an actor to queue incoming requests for some remotely referenced data until potentially required pending responses from other actors were received. Manually maintaining this state between messages is potentially complex. In particular, the amount of code needed to handle such state increases with the number of peers that communicate. Among the case studies, the third one defines the most peers and thus the most complex cross-peer communication, making the communication overhead most evident.

For both Java RMI and Dyno, each remote method is declared with its argument and return types. RMI, however, requires an additional interface definition for each remote object.

We report the lines of code (LOC) to measure the boilerplate necessary to implement dynamic placement ("LOC" column in Table 1). Since the case studies are implemented in the same language but with different technologies, all implementations share the same core and differ only in the treatment of references and dynamic placement. Depending on the structure of the cases study (number of peers and interaction patterns), either Akka or RMI may require less management overhead as the respective other variant – for remote interface definitions in RMI or coordinating sends and corresponding receives in Akka. In all case studies, Dyno reduces the amount of potentially error-prone user code to manually handling dynamic value placement. This observation is in line with our hypothesis the Dyno reduces the complexity of handling dynamic placement.

8.2 Performance

To evaluate the potential impact of dynamic placement introduced by Dyno, we implemented a benchmark in Dyno and in RMI based on the database–cache system of Section 3. We measure the performance of 2.5 K (red circle) and 5 K (blue square) retrieve operations for various cache hit/miss rates. We use the RMI performance as a reference. The results are in Figure 5. The x-axis is the percentage of cache hit rate out of 2.5 K and 5 K retrievals, ranging from only cache misses (0 %) to all, but one, cache hits (100 %). The y-axis denotes the time in milliseconds to retrieve the 2.5 K and 5 K references from the database or the cache. We used Redis [Macedo and Oliveira 2011], an in-memory key–value store, for the cache and MariaDB [Kenler and Razzoli 2015] for the database. The benchmark was executed on a 4.8 GHz Intel Core i7 with 32 GB of RAM.

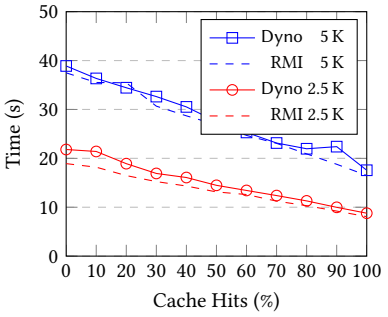


Fig. 5. Benchmark for database–cache.

The results show that when 60 % of the retrieval operations are repeated, then the database–cache system shaves off 38 % of its time when 2.5 K retrievals are made and 35 % when 5 K retrievals are made. Crucially, in contrast to existing systems, this benefit does not come at the cost of placement correctness and architectural guarantees.

8.3 Open-Source Case Study

AntennaPod [AntennaPod Developers 2011] is a widely-used open-source Android application for managing and playing podcasts. Its media playing subsystem amounts to 6.66 kLOC over five components distributed across three different nodes. We reimplemented AntennaPod's ad hoc approach to represent placement using in Dyno, increasing safety at numerous critical points in the code.

Dynamic Placement in AntennaPod. The media player in AntennaPod can be placed on the local device or on a remote device. The local media player is capable of playing both local and remote episodes, the remote one can only play remote episodes. Thus, from the point of view of a media player, the episode it is given is dynamically placed. Similarly, from the point of view of the database, the byte-stream of an episode whose metadata (i.e., URL) it stores is also dynamically placed.

Placement Representation. To dynamically place the *media player* functionality, both the local player `LocalPSMP` and the remote player `CastPsmp` extend a common abstract class. Similarly, for media whose *metadata* can be local or remote, `FeedMedia` and `RemoteMedia` implement a common interface. The separation between local and remote *byte-streams*, on the other hand, is modeled using a different mechanism: First, a boolean flag indicating whether the media was downloaded or not, and second, the scheme component of the URL string, e.g., a remote episode has `http` or `https` as the scheme, whereas a local one has `content` as its scheme or is a schemeless absolute path.

Architectural Constraints. Placing the three entities (media player, metadata, byte-stream) remotely or locally amounts to eight potential dynamic placement combinations. AntennaPod deems one of these combinations – playing an episode whose byte-stream is local on the remote player – too complex and prohibits it. `CastPsmp` eliminates such combination by checking whether a URL string starts with the `content` scheme and throws an exception if it does.

Design Issues and Placement Bug. Representing placement manually caused convoluted logic that relied on various nested run time checks to maintain non-obvious invariants. Thus, upholding such invariants correctly is prone to programming errors. In particular, we found that AntennaPod fails to perform the check on URLs correctly. In our experiment, we were able to violate the architectural constraints by creating an RSS feed for a podcast that contains media URLs with schemeless absolute paths. When a user subscribes to this podcast, the local player plays a file from the player's device if it happens to exist or it fails. We argue that these error-prone run time checks are only necessary due to the lack of proper abstractions that guarantee architectural constraints.

A Safe Version of AntennaPod. We reimplemented the components of AntennaPod's media playing subsystem in Dyno. We use different placements for the different roles (FS, PSMP, and App) and their location (local and remote). We place the locally available byte-streams on the `LocalFS` peer and remote byte-streams on `RemoteFS`. `LocalPsmp` and `CastPsmp` are the peers that manage the local and remote media playback, respectively. `App` is the peer on which the Android application is be executed, which also maintains a *downloaded-from*-relation between references to the `LocalFS` and the `RemoteFS`. The architecture connecting the five peers is visualized in Figure 6. Nodes are the peer types and edges are the architectural ties among them. Ruling out the prohibited dynamic placement scenario – remotely playing a downloaded episode – is hence clearly enforced by the

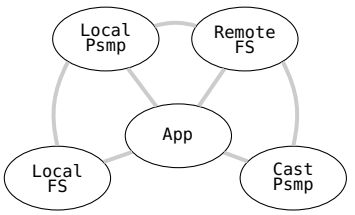


Fig. 6. Architectural constraints of the AntennaPod case study.

architecture: Any attempt to dereference a reference to a local file on the remote player will fail to type-check as their is no architectural tie between the two peers.

Summary. To quantify the advantage of the Dyno reimplementation, we first identified branchings that dynamically check properties of the URL and the boolean flags maintained by the media objects, and any run time type check related to how placement is modeled in the original implementation. We found 84 such run time checks which can result in the application taking an incorrect branch if they are erroneous or incomplete (as is the case with the bug described earlier).

By tracking the placement in the types and using Dyno's type system, we can guarantee that placement assumptions made by any function are always explicit and cannot be violated.

8.4 Dynamic Placement in the F-Droid App Store

As it is difficult to automatically determine if an application is adopting dynamic placement in general, we focus on a specific case: the encoding of dynamic references through URLs, which resulted into the bug we found in AntennaPod. We scanned F-Droid [Gultnieks 2010] – the largest app store of open-source apps with over 3 K repositories – for projects that check a URL's schema. In total we have found 133 projects that do so, showing that placement is a common issue. Among the 499 files that contain such checks, after manual inspection, we spotted 420 checks in a branch condition – meaning that a schema's URL, whether it's a file, a database entry, or a remote resource, guides the program's logic. We observed two reasons for branching based on the URL scheme: to pick the appropriate API that handles each URL correctly, and to eliminate a case by exiting early.

An example of the first category is in the Alarmio alarm clock app:

```
if (backgroundUrl.startsWith("http"))
  Glide.with(imageView.getContext()).load(backgroundUrl).into(imageView);
else if (backgroundUrl.contains("://")) {
  if (backgroundUrl.startsWith("content://")) {
    String path = Uri.parse(backgroundUrl).getLastPathSegment();
    if (path != null && path.contains(":"))
      path = "/storage/" + path.replaceFirst(":", "/");
    else
      path = Uri.parse(backgroundUrl).getPath();
    Glide.with(imageView.getContext()).load(new File(path)).into(imageView); }
  else
    Glide.with(imageView.getContext()).load(Uri.parse(backgroundUrl)).into(imageView); }
else
  Glide.with(imageView.getContext()).load(new File(backgroundUrl)).into(imageView);
```

The code takes different branches based on the location of the background image to display for a ringing alarm. This is necessary as the code to load the image differs between places and the language does not allow abstracting over placements.

The second category can be found, among others, in the F-Droid app itself, in the VLC media player app and in the Elements secure messenger. The following code from F-Droid's, executed when a repository is created, returns early for specific places (i.e., for the `content` and `file` place):

```
if (repo.getAddress().startsWith("content://") || repo.getAddress().startsWith("file://")) {
  // no need to show a QR Code, it is not shareable
  return; }
```

The following example from the the VideoLAN Client (VLC) media player is part of the implementation of a `MediaLibrary` and returns a wrapped media object given its URI. The code cannot return a valid media object for all places and returns `null` in such cases:

```
public MediaWrapper getMedia(Uri uri) {
  if ("content".equals(uri.getScheme())) return null;
  final String vlcMrl = Tools.encodeVLCMrl(uri.toString());
  return mIsInitiated && !TextUtils.isEmpty(vlcMrl) ? nativeGetMediaFromMrl(vlcMrl) : null; }
```

Next we show an example from the Element's application for secure group chatting – executed whenever a ringtone is being set. The code ignores the `file` location because not treating this location specially has already led to run time errors, as indicated by the comment:

```
value = uri.toString()
if (value.startsWith("file://")) {
  // it should never happen
  // else android.os.FileUriExposedException will be triggered.
  // see https://github.com/vector-im/riot-android/issues/1725
  return }
```

All the error-prone run time checks discussed aboves can be avoided using Dyno's approach of tracking placement in the type system as (1) the type system can rule out cases that can never happen statically and (2) dynamic case distinctions based on the placement of a value enjoy exhaustiveness checks thanks to describing potential placements as union type.

9 RELATED WORK

Multitier Web Languages. Dyno's language support to specify the placement of data and computation is similar to multitier programming, which emerged in the Web context to remove the separation between client and server code. The same multitier language is used for all tiers and the compiler takes care of the translation to the target platform (e.g., to JavaScript for the Browser tier). Hop [Serrano et al. 2006] and Hop.js [Serrano and Prunet 2016] are dynamically typed languages that follow a traditional client–server communication scheme with asynchronous callbacks. In Links [Cooper et al. 2006] and Opa [Rajchenbach-Teller and Sinot 2010], functions are annotated to specify either client- or server-side execution. Both languages also follow the client–server model and feature a static type system. Other examples for multitier languages include the ML-based Ur/Web language [Chlipala 2015] and the Eliom language [Radanne et al. 2016] and the JavaScript-based StiP.js [Philips et al. 2018] that uses slicing to separate the server and client parts of the program by detecting dependencies between both. Due to the restriction to client–server Web applications, all approaches above lack language abstractions for architectural specifications and multiple peers. Further, they do not feature language support for dynamic placement.

Languages With (Automatic) Placements. In languages such as Fabric [Liu et al. 2017, 2009] distribution and persistence is checked based on security and privacy policies and information flow among peers. Fabric performs remote calls using the `@` notation and require two checks, one at compile-time and one at run time. Hence, Fabric's remote calls can fail at run time due to security reasons. In contrast, placement is always explicit in Dyno and the type system ensures that a call never violates the architectural specification at run time. We believe that, in addition to Dyno's architectural checks, an information flow type system could check security properties similar to Fabric.

Fabric's persistence layer, inspired by the persistence abstractions in Thor [Liskov et al. 1996] and Theta [Liskov et al. 1995], allows the transparent distribution of the functions and objects to the different roles. The compiler effectively separates the program and generates the roles that each peer must perform. Similar to abstracting remote access in Java RMI (discussed in Section 8.1), persistence abstractions lack placement correctness and architectural safety which are one of our design goals.

Approaches that make distribution and persistence largely transparent to developers based on crosscutting concerns such as security or privacy (e.g., Fabric) or performance or offline availability (e.g., StiP.js [Philips et al. 2014]) apply to a domain for which the developer can provide domain-specific knowledge. Dyno lies in a point in the design space that is general and where such automation is not possible. On the other hand, Dyno offers explicit means for developers to reason

about placements. The reason is that – without further domain knowledge – the differences between local and remote invocations cannot be completely abstracted away [Waldo et al. 1994].

Multitier Languages With Placement Types. Following IS5, the intuitionistic modal logic of Simpson [1994], Murphy VII et al. [2004] developed Lambda 5, where worlds known from modal logic correspond to Dyno's peers. The authors provide an interpretation for $\Diamond A$ to mean a reference for a resource of type A in some world (the exact world is not captured in the typing). The only values that inhabit $\Diamond A$ are references of the form $w.\ell$ where w is a specific world and ℓ is a label. The $\Diamond A$ type coincides with our references if the architecture of the program only defines one peer type that is connected to itself. The interpretation of $\Box A$ means that a resource of type A can be used in any world. The intuition is that if a resource still types from the point of view of a newly-created abstract world where nothing is known, then this resource can be used in any world.

Murphy VII et al. [2008] designed the ML5 multitier language based on Lambda 5. The $\Box$ operator is replaced with the "shamrock" operator which introduces an hypothetical world that can occur in the type. Yet, developers cannot choose a world to substitute for the abstract one introduced through the operator. Instead, the current world is always picked. In contrast, peer type variables support instantiating a peer variable with a concrete peer type. In this paper, we tackle this problem using ScalaLoci's peer types [Weisenburger et al. 2018] which were inspired by ML5. Generalizing multitier languages for the web client–server setup [Weisenburger et al. 2020], ScalaLoci extended type judgments with peer types to indicate expression placement. Yet, in ScalaLoci, peer types are part of the typing judgments and not the types themselves.

Aggregate programming. Multitier programming resembles aggregate programming in that it promotes a global, high-level view to the developer of a computing system. This approach emphasizes global patterns that emerge as the result of local interactions and computations, avoiding the need for any centralized governance. The *field computing* model [Lafuente et al. 2017; Mamei and Zambonelli 2004; Viroli et al. 2019] conceptualizes the behavior of distributed setups as a "computational field" where a function links nodes in a network to data points. The Field calculus [Audrito et al. 2019] facilitates distributed computational processes across devices in a geospatial framework including networks of sensors, autonomous robots, or any system with a spatial component. Operations are visualized through the computational field, i.e., relations between space-time and data elements. Devices have the capacity to access and alter the immediate values of the fields and, by sensing data from adjacent devices, to determine new values for these fields. This enables a programming paradigm where challenges such as concurrency, asynchronous functions, network dialogues, potential loss of messages, and system disruptions do not need to be addressed directly [Audrito et al. 2022].

Choreographies. Similar to the multitier programming paradigm, in choreographic programming, a concurrent system is defined as a single compilation unit, which is a global description of the interactions and computations of the distributed system's connected components [Lanese et al. 2008; Montesi 2014]. With its formal foundations rooted in process calculi [Baeten 2005], choreographic programming has been used to investigate new techniques on information flow control [Lluch Lafuente et al. 2015], deadlock-free distributed algorithms [Cruz-Filipe and Montesi 2016], and protocols for dynamic run time code updates for components [Preda et al. 2017]. Role parameters in the choreographic language Choral [Giallorenzo et al. 2020] recall Dyno's peer types and can be freely instantiated with different arguments. To the best of our knowledge, uncertainty of placement decisions has not been explored so far in the context of choreographies.

HasChor [Shen et al. 2023] is a state-of-the-art implementation of a choreography library in Haskell. In HasChor peer instances are identified in the type and thus peer types do not exist.

This forces all applications expressed in HasChor to have a static architecture. To get around this limitation HasChor allows quantifying over peer instances, but this placement polymorphism differs greatly from the one supported by Dyno; quantified peers stand for any peer, thus roles expressed in the types do not exist. Therefore HasChor is unable to represent fine-grained dynamic placement without resorting to encoding placement and enforcing it dynamically through techniques similar to the ones in 5.

Session Types. Multiparty session types [Honda et al. 2008] model communication protocols in distributed systems by capturing the structure and behavior of the communication between multiple processes at the type level. Session types ensure that communication is well-formed and adheres to a predefined protocol. The different processes can be seen as representing different places and hence specifying their communication as placing certain parts of the communication protocol on certain processes. Yet, in contrast to Dyno, the processes that communicate are static and cannot express dynamic placement.

10 CONCLUSION

Dynamic placement of computation and data is crucial in many distributed software systems. In this paper, we present the design of Dyno, a programming language where placed values are first class. This feature enables a novel interpretation of dynamic placement as union (placement) types. We provide a formalization that shows that our type system is sound and it ensures placement correctness and architectural conformance. Our evaluation shows that Dyno can reduce the potential errors in applications using dynamic placement.

ACKNOWLEDGMENTS

We would like to thank Lukas Lehmann for the implementation of the first prototype of this work and all reviewers of this paper for their comments and suggestions. This work is supported by the Swiss National Science Foundation (SNSF), grant 200429, and the Basic Research Fund of the University of St. Gallen (GFF) through the International Postdoctoral Fellowship (IPF) 1031569.

DATA AVAILABILITY STATEMENT

The artifact is available on Zenodo [Zakhour et al. 2023]. It includes the implementation of Dyno as well as scripts which can be executed to reproduce our evaluation results.

REFERENCES

Khalil Amiri, David Petrou, Gregory R. Ganger, and Garth A. Gibson. 2000. Dynamic Function Placement for Data-Intensive Cluster Computing. In *Proceedings of the 2000 USENIX Annual Technical Conference (USENIX ATC '00)*. USENIX Association, San Diego, CA, 16 pages. https://www.usenix.org/conference/2000-usenix-annual-technical-conference/dynamic-function-placement-data-intensive-cluster

AntennaPod Developers. 2011. AntennaPod – The Open Podcast Player. https://antennapod.org, https://github.com/AntennaPod/AntennaPod. Accessed: 2023-07-30, Commit: 88289d0.

Zachary Arani, Drake Chapman, Chenxiao Wang, Le Gruenwald, Laurent d'Orazio, and Taras Basiuk. 2020. A Scored Semantic Cache Replacement Strategy for Mobile Cloud Database Systems. In *ADBIS, TPDL and EDA 2020 Common Workshops and Doctoral Consortium*, Ladjel Bellatreche, Mária Bieliková, Omar Boussaïd, Barbara Catania, Jérôme Darmont, Elena Demidova, Fabien Duchateau, Mark Hall, Tanja Merčun, Boris Novikov, Christos Papatheodorou, Thomas Risse, Oscar Romero, Lucile Sautot, Guilaine Talens, Robert Wrembel, and Maja Žumer (Eds.). Springer International Publishing, Cham, 237–248. https://doi.org/10.1007/978-3-030-55814-7_20

Giorgio Audrito, Roberto Casadei, Ferruccio Damiani, Guido Salvaneschi, and Mirko Viroli. 2022. Functional Programming for Distributed Systems with XC. In *Proceedings of the 36th European Conference on Object-Oriented Programming (ECOOP '22)* (Berlin, Germany) *(Leibniz International Proceedings in Informatics (LIPIcs), Vol. 222)*, Karim Ali and Jan Vitek (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, Article 20, 28 pages. https://doi.org/10.4230/LIPIcs.ECOOP.2022.20

Giorgio Audrito, Mirko Viroli, Ferruccio Damiani, Danilo Pianini, and Jacob Beal. 2019. A Higher-Order Calculus of Computational Fields. *ACM Transactions on Computational Logic* 20, 1, Article 5 (Jan. 2019), 55 pages. https://doi.org/10.1145/3285956

J. C. M. Baeten. 2005. A Brief History of Process Algebra. *Theoretical Computer Science* 335, 2–113 (May 2005), 131–146. https://doi.org/10.1016/j.tcs.2004.07.036

Stephanie Balzer. 2011. *Rumer: A programming language and modular verification technique based on relationships.* Ph. D. Dissertation. Zürich, Switzerland. https://doi.org/10.3929/ethz-a-007086593

Adam Chlipala. 2015. 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) *(POPL '15)*. ACM, New York, NY, USA, 153–165. https://doi.org/10.1145/2676726.2677004

Stuart Clayman, Elisa Maini, Alex Galis, Antonio Manzalini, and Nicola Mazzocca. 2014. The Dynamic Placement of Virtual Network Functions. In *2014 IEEE Network Operations and Management Symposium (NOMS '14)*. 1–9. https://doi.org/10.1109/NOMS.2014.6838412

Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. 2006. Links: Web Programming without Tiers. In *Proceedings of the 5th International Conference on Formal Methods for Components and Objects* (Amsterdam, Netherlands) *(FMCO '06)*, Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever (Eds.). Springer-Verlag, Berlin, Heidelberg, 266–296. https://doi.org/10.1007/978-3-540-74792-5_12

Luís Cruz-Filipe and Fabrizio Montesi. 2016. Choreographies in Practice. In *Proceedings of the 36th IFIP International Conference on Formal Techniques for Distributed Objects, Components, and Systems* (Heraklion, Greece) *(FORTE '16)*, Elvira Albert and Ivan Lanese (Eds.). Springer-Verlag, Berlin, Heidelberg, 114–123. https://doi.org/10.1007/978-3-319-39570-8_8

Troy Bryan Downing. 1998. *Java RMI: Remote Method Invocation* (1st ed.). IDG Books Worldwide, Inc., USA.

Saverio Giallorenzo, Fabrizio Montesi, and Marco Peressotti. 2020. Choreographies as Objects. arXiv:2005.09520

Saverio Giallorenzo, Fabrizio Montesi, Marco Peressotti, David Richter, Guido Salvaneschi, and Pascal Weisenburger. 2021. Multiparty Languages: The Choreographic and Multitier Cases. In *Proceedings of the 35th European Conference on Object-Oriented Programming (ECOOP '21)* (Aarhus, Denmark) *(Leibniz International Proceedings in Informatics (LIPIcs), Vol. 194)*, Anders Møller and Manu Sridharan (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, Article 22, 27 pages. https://doi.org/10.4230/LIPIcs.ECOOP.2021.22

Ciaran Gultnieks. 2010. F-Droid – Free and Open Source Android App Repository. https://f-droid.org. Accessed: 2023-07-30.

Daco Harkes and Eelco Visser. 2014. Unifying and Generalizing Relations in Role-Based Data Modeling and Navigation. In *Software Language Engineering*, Benoît Combemale, David J. Pearce, Olivier Barais, and Jurgen J. Vinju (Eds.). Springer International Publishing, Cham, 241–260. https://doi.org/10.1007/978-3-319-11245-9_14

Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2008. Multiparty Asynchronous Session Types. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Francisco, CA, USA) *(POPL '08)*. ACM, New York, NY, USA, 273–284. https://doi.org/10.1145/1328438.1328472

A. Karve, T. Kimbrel, G. Pacifici, M. Spreitzer, M. Steinder, M. Sviridenko, and A. Tantawi. 2006. Dynamic Placement for Clustered Web Applications. In *Proceedings of the 15th International Conference on World Wide Web* (Edinburgh, Scotland) *(WWW '06)*. ACM, New York, NY, USA, 595–604. https://doi.org/10.1145/1135777.1135865

Emilien Kenler and Federico Razzoli. 2015. *MariaDB Essentials.* Packt Publishing Ltd.

Alberto Lluch Lafuente, Michele Loreti, and Ugo Montanari. 2017. Asynchronous Distributed Execution of Fixpoint-Based Computational Fields. *Logical Methods in Computer Science* 13 (2017). Issue 1. https://doi.org/10.23638/LMCS-13(1:13)2017

Ivan Lanese, Claudio Guidi, Fabrizio Montesi, and Gianluigi Zavattaro. 2008. Bridging the Gap between Interaction- and Process-Oriented Choreographies. In *Proceedings of the 6th IEEE International Conference on Software Engineering and Formal Methods* (Cape Town, South Africa) *(SEFM '08)*. IEEE Computer Society, Washington, DC, USA, 323–332. https://doi.org/10.1109/SEFM.2008.11

Lightbend Inc. 2020. *Akka Scala Documentation, Release 2.4.20.* Accessed: 2023-07-30.

B. Liskov, A. Adya, M. Castro, S. Ghemawat, R. Gruber, U. Maheshwari, A. C. Myers, M. Day, and L. Shrira. 1996. Safe and Efficient Sharing of Persistent Objects in Thor. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data* (Montreal, Quebec, Canada) *(SIGMOD '96)*. ACM, New York, NY, USA, 318–329. https://doi.org/10.1145/233269.233346

Barbara Liskov, Dorothy Curtis, Mark Day, Sanjay Ghemawat, Robert Gruber, Paul Johnson, and Andrew C. Myers. 1995. Theta Reference Manual. http://www.cs.cornell.edu/andru/papers/thetaref.pdf. (1995). Accessed: 2023-07-30.

Jed Liu, Owen Arden, Michael D. George, Andrew C. Myers, Toby Murray, Andrei Sabelfeld, and Lujo Bauer. 2017. Fabric: Building Open Distributed Systems Securely by Construction. *Journal of Computer Security* 25, 4–5 (Jan. 2017), 367–426. https://doi.org/10.3233/JCS-15805

Jed Liu, Michael D. George, K. Vikram, Xin Qi, Lucas Waye, and Andrew C. Myers. 2009. Fabric: A Platform for Secure Distributed Computation and Storage. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles* (Big Sky, MT, USA) *(SOSP '09)*. ACM, New York, NY, USA, 321–334. https://doi.org/10.1145/1629575.1629606

Alberto Lluch Lafuente, Flemming Nielson, and Hanne Riis Nielson. 2015. *Discretionary Information Flow Control for Interaction-Oriented Specifications*. Lecture Notes in Computer Science, Vol. 9200. Springer-Verlag, Berlin, Heidelberg, 427–450. https://doi.org/10.1007/978-3-319-23165-5_20

Tiago Macedo and Fred Oliveira. 2011. *Redis Cookbook: Practical Techniques for Fast Data Manipulation*. O'Reilly Media.

Marco Mamei and Franco Zambonelli. 2004. Programming Pervasive and Mobile Computing Applications with the TOTA Middleware. In *Proceedings of the Second IEEE Annual Conference on Pervasive Computing and Communications*. IEEE Press, Piscataway, NJ, USA, 263–273. https://doi.org/10.1109/PERCOM.2004.1276864

Mastodon. 2016. *Mastodon: Your self-hosted, globally interconnected microblogging community*. https://github.com/mastodon/mastodon Accessed: 2023-07-30.

Heather Miller, Philipp Haller, and Martin Odersky. 2014. Spores: A Type-Based Foundation for Closures in the Age of Concurrency and Distribution. In *ECOOP '14*, Richard Jones (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 308–333. https://doi.org/10.1007/978-3-662-44202-9_13

Fabrizio Montesi. 2014. Kickstarting Choreographic Programming. In *Proceedings of the 13th International Workshop on Web Services and Formal Methods* (Eindhoven, Netherlands) *(WS-FM '14)*, Thomas Hildebrandt, António Ravara, Jan Martijn van der Werf, and Matthias Weidlich (Eds.). Springer-Verlag, Berlin, Heidelberg, 3–10. https://doi.org/10.1007/978-3-319-33612-1_1

Tom Murphy VII, Karl Crary, and Robert Harper. 2008. Type-Safe Distributed Programming with ML5. In *Trustworthy Global Computing* (Sophia-Antipolis, France), Gilles Barthe and Cédric Fournet (Eds.). Springer-Verlag, Berlin, Heidelberg, 108–123. https://doi.org/10.1007/978-3-540-78663-4_9

Tom Murphy VII, Karl Crary, Robert Harper, and Frank Pfenning. 2004. A Symmetric Modal Lambda Calculus for Distributed Computing. In *Proceedings of the 19th IEEE Symposium on Logic in Computer Science* (Turku, Finland) *(LICS '04)*. IEEE, Piscataway, NJ, USA, 286–295. https://doi.org/10.1109/LICS.2004.1319623

Laure Philips, Joeri De Koster, Wolfgang De Meuter, and Coen De Roover. 2018. Search-based Tier Assignment for Optimising Offline Availability in Multi-tier Web Applications. *The Art, Science, and Engineering of Programming* 2, 2, Article 3 (Dec. 2018), 29 pages. https://doi.org/10.22152/programming-journal.org/2018/2/3

Laure Philips, Coen De Roover, Tom Van Cutsem, and Wolfgang De Meuter. 2014. Towards Tierless Web Development without Tierless Languages. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software* (Portland, OR, USA) *(Onward! '14)*. ACM, New York, NY, USA, 69–81. https://doi.org/10.1145/2661136.2661146

Mila Dalla Preda, Maurizio Gabbrielli, Saverio Giallorenzo, Ivan Lanese, and Jacopo Mauro. 2017. Dynamic Choreographies: Theory And Implementation. *Logical Methods in Computer Science* 13, 2 (April 2017), 57 pages. https://doi.org/10.23638/LMCS-13(2:1)2017

Gabriel Radanne, Jérôme Vouillon, and Vincent Balat. 2016. Eliom: A core ML language for Tierless Web Programming. In *Proceedings of the 14th Asian Symposium on Programming Languages and Systems* (Hanoi, Vietnam) *(APLAS '16)*, Atsushi Igarashi (Ed.). Springer-Verlag, Berlin, Heidelberg, 377–397. https://doi.org/10.1007/978-3-319-47958-3_20

David Rajchenbach-Teller and Franois-Régis Sinot. 2010. Opa: Language Support for a Sane, Safe and Secure Web. http://owasp.org/www-pdf-archive/OWASP_AppSec_Research_2010_OPA_by_Rajchenbach-Teller.pdf. In *Proceedings of the OWASP AppSec Research* (Stockholm, Sweden). Accessed: 2023-07-30.

Bob Reynders, Frank Piessens, and Dominique Devriese. 2020. Gavial: Programming the Web with Multi-Tier FRP. *The Art, Science, and Engineering of Programming* 4, 3, Article 6 (Feb. 2020), 32 pages. https://doi.org/10.22152/programming-journal.org/2020/4/6

Manuel Serrano, Erick Gallesio, and Florian Loitsch. 2006. Hop, A Language for Programming the Web 2.0. In *Companion to the 21th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Portland, OR, USA) *(OOPSLA Companion '06)*. ACM, New York, NY, USA.

Manuel Serrano and Vincent Prunet. 2016. A Glimpse of Hopjs. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming* (Nara, Japan) *(ICFP '16)*. ACM, New York, NY, USA, 180–192. https://doi.org/10.1145/2951913.2951916

Gan Shen, Shun Kashiwa, and Lindsey Kuper. 2023. HasChor: Functional Choreographic Programming for All (Functional Pearl). *Proceedings of the ACM on Programming Languages* 7, ICFP, Article 207 (Aug. 2023), 25 pages. https://doi.org/10.1145/3607849

Alex K. Simpson. 1994. *The Proof Theory and Semantics of Intuitionistic Modal Logic*. Ph. D. Dissertation. http://hdl.handle.net/1842/407

Friedrich Steimann. 2013. Content over Container: Object-Oriented Programming with Multiplicities. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software* (Indianapolis, Indiana, USA) *(Onward! '13)*. ACM, New York, NY, USA, 173–186. https://doi.org/10.1145/2509578.2509582

Friedrich Steimann. 2015. None, One, Many – What's the Difference, Anyhow?. In *1st Summit on Advances in Programming Languages (SNAPL '15) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 32)*, Thomas Ball, Rastislav Bodik,

Shriram Krishnamurthi, Benjamin S. Lerner, and Greg Morrisett (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 294–308. https://doi.org/10.4230/LIPIcs.SNAPL.2015.294

Hana Teyeb, Nejib Ben Hadj-Alouane, Samir Tata, and Ali Balma. 2017. Optimal Dynamic Placement of Virtual Machines in Geographically Distributed Cloud Data Centers. *International Journal of Cooperative Information Systems* 26, 03 (2017). https://doi.org/10.1142/S0218843017500010

Mirko Viroli, Jacob Beal, Ferruccio Damiani, Giorgio Audrito, Roberto Casadei, and Danilo Pianini. 2019. From Distributed Coordination to Field Calculus and Aggregate Computing. *Journal of Logical and Algebraic Methods in Programming* 109 (2019). https://doi.org/10.1016/j.jlamp.2019.100486

Jim Waldo, Geoff Wyant, Ann Wollrath, and Sam Kendall. 1994. A Note on Distributed Computing. https://scholar.harvard.edu/waldo/publications/note-distributed-computing. *Sun Microsystems Laboratories* (1994). Accessed: 2023-07-30.

Pascal Weisenburger, Mirko Köhler, and Guido Salvaneschi. 2018. Distributed System Development with ScalaLoci. *Proceedings of the ACM on Programming Languages* 2, OOPSLA, Article 129 (Oct. 2018), 30 pages. https://doi.org/10.1145/3276499

Pascal Weisenburger and Guido Salvaneschi. 2019. Multitier Modules. In *Proceedings of the 33rd European Conference on Object-Oriented Programming (ECOOP '19)* (London, UK) *(Leibniz International Proceedings in Informatics (LIPIcs), Vol. 134)*, Alastair F. Donaldson (Ed.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, Article 3, 29 pages. https://doi.org/10.4230/LIPIcs.ECOOP.2019.3

Pascal Weisenburger, Johannes Wirth, and Guido Salvaneschi. 2020. A Survey of Multitier Programming. *Comput. Surveys* 53, 4, Article 81 (Sept. 2020), 35 pages. https://doi.org/10.1145/3397495

A.K. Wright and M. Felleisen. 1994. A Syntactic Approach to Type Soundness. *Information and Computation* 115, 1 (1994), 38–94. https://doi.org/10.1006/inco.1994.1093

Fan Yang, Jinfeng Li, and James Cheng. 2016. Husky: Towards a More Efficient and Expressive Distributed Computing Framework. *Proceedings of the VLDB Endowment* 9, 5 (Jan. 2016), 420–431. https://doi.org/10.14778/2876473.2876477

George Zakhour, Pascal Weisenburger, and Guido Salvaneschi. 2023. *Type-Safe Dynamic Placement with First-Class Placed Values*. https://doi.org/10.5281/zenodo.8148841

Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew C. Myers. 2002. Secure Program Partitioning. *ACM Transactions on Computer Systems* 20, 3 (Aug. 2002), 283–328. https://doi.org/10.1145/566340.566343

Yongluan Zhou, Beng Chin Ooi, and Kian-Lee Tan. 2005. Dynamic Load Management for Distributed Continuous Query Systems. In *21st International Conference on Data Engineering (ICDE '05)*. 322–323. https://doi.org/10.1109/ICDE.2005.54

Mulki Indana Zulfa, Rudy Hartanto, and Adhistya Erna Permanasari. 2020. Caching strategy for Web application – a systematic literature review. *International Journal of Web Information Systems* 16, 5 (Jan. 2020), 545–569. https://doi.org/10.1108/IJWIS-06-2020-0032

A DYNO ANTENNAPOD IMPLEMENTATION

The following code listing shows the data types and the signatures of the important functions in our reimplementation. `App` uses the `metadata` map to track whether a remote media has been saved (when the remote media is in the keyset of the map), and whether it has been downloaded (when the reference to the local file is a `Some`). The `save` method takes a reference to a remote file and puts it into the `metadata` map, mapping to `None`. The `download` method on `App` instructs the `LocalFS` peer to download a copy of the byte-stream locally and return a reference to it which it will store in the `metadata` map. The `toByteStream` method maps a media file that is dynamically placed into a byte-stream that is dynamically placed. Similarly, `play`, whose source code is explained later, takes a dynamically placed media file and player to play the file.

```
1 class Media(url: String)
2 trait Player { def play(s: Array[Byte]): Unit }
3
4 val metadata: Map[Media at RemoteFS, Option[Media at LocalFS]] on App
5 def save (p: Media at RemoteFS): Unit on App
6 def download (p: Media at RemoteFS): Media at LocalFS on App // deals with URLs
7
8 def toByteStream (p: Media at (LocalFS | RemoteFS)): Array[Byte] at (LocalFS | RemoteFS) on App // deals with URLs
9 def play (m: Media at (LocalFS | RemoteFS), p: Player at (LocalPsmp | CastPsmp)): Unit on App
```

To illustrate how architectural constraints are enforced and do not need to be handled explicitly, we expand the definition of `play` in the code listing that follows. Lines 4 to 5 compute a stream from references using the `toByteStream` function. Once the stream reference is obtained, an explicit pattern match on the placement of the player must be performed, otherwise dereferencing the media would fail at compile-time (Lines 9 and 13). The exhaustive pattern match on the dynamically placed value has to handle each placement case (Line 11).

```
1 def play (m: Media at (LocalFS | RemoteFS),
2           p: Player at (LocalPsmp | CastPsmp): Unit = on[App] {
3
4     val s: Array[Byte] at (LocalFS | RemoteFS) = toByteStream((m.toEither[LocalFS, RemoteFS] flatMap {
5         remoteMedia => metadata.get(remoteMedia).toLeft(remoteMedia) }).fromEither)
6
7     p.toEither[LocalPsmp, CastPsmp] match {
8         Left(p) => on[LocalPsmp].run.capture(p, s) {
9             p.deref.play(on(s.peer).run.capture(s) { s.deref }.asLocal) },
10        Right(p) => s.toEither[LocalFS, RemoteFS] match {
11            Left(_) => error("Cannot play local file remotely")
12            Right(s) => on[CastPsmp].run.capture(p,s) {
13                p.deref.play(on(s.peer).run.capture(s) { s.deref }.asLocal) } } } }
```

Worthy of note is that the media byte-stream will only ever be sent across the wire when the reference is dereferenced, i.e., the data will only travel across the tie between a player and a file system without any hops.

Qualitative Conclusions. Qualitatively, the example of the `play` function illustrates two differences between AntennaPod's version and Dyno's. First, due to the representation of placement as an implementation of a common interface in AntennaPod, there are two `play` functions for each placement, and the setup before playing is duplicated verbatim. Our single `play` function eliminates code duplication and uses explicit language constructs to direct control flow dynamically to different peers. Second, these constructs guarantee safe access to dynamically placed values in contrast to AntennaPod, where placement checks – as shown earlier – are incomplete and hidden behind nested function calls.

B PROOFS

This appendix lists the proof of every result from Sections 6.4 and 6.5. When we deem parts of proofs not insightful we present them crudely without delving too deep in the details. The presentation of the proofs follows the same order as they appear in Section 6.

Proof of Progress (Theorem 1). The proof follows the traditional proof by structural induction on the typing derivation. Case T-Var: does not apply. Cases T-Abs, T-Label, T-Peer: follows trivially from the fact that the expression is a value. Case T-App: follows by the induction hypothesis or by applying E-App. Case T-Broad: follows from applying E-Context or E-Broad. Case T-Peerof: follows from applying E-Context or E-Peerof. Case T-Match and T-PMatch: follows from applying E-Context or E-Match or E-PMatch. The latter two rules' precondition follow immediately from the precondition of the typing rule. Case T-Deref: follows from applying E-Context or either E-Deref-OK or E-Deref-Err. One of the latter two rules must apply since the disjunction of their preconditions is always true when the codomain of σ contains values. If it does not then E-Conc-Lab applies. Case T-Remote: follows from E-Context or by the induction hypothesis or by E-Rem-Move, E-Rem-Eval, E-Rem-OK, or E-Rem-Err. □

Proof of Preservation (Theorem 2). The proof follows the traditional proof by structural induction on the typing derivation. The cases for T-Peer, T-Label, T-Var, and T-Abs do not apply because no evaluation rules exists to reduce the expression. Case T-Broad: follows by the induction hypothesis or by using E-Broad and T-Broad again. Case T-Peerof: follows by the induction hypothesis or by using E-Peerof and T-Peer. Cases T-App, T-Match, and T-PMatch: follow by the induction hypothesis or using the fact that the type is preserved under substitution. Case T-Deref follows from the induction hypothesis, or either from the reduction E-Deref-OK or E-Deref-Err followed by the preservation of types under substitution. T-Remote follows from the induction hypothesis, or from the preservation of types under substitution, or from using T-Label.

If at any time E-Conc-Rem or E-Conc-Lab are used then the theorem follows by the induction hypothesis and the assumption that σ is well-typed and that Ω is coherent with it. □

Proof of Store Well-Typeness Preservation (Lemma 1). By structural induction on the evaluation rules. Cases E-Dist, E-App, E-Broad, E-Match, E-PMatch, E-Peerof, E-Deref-OK, E-Deref-Err, E-Rem-Move: follows trivially because the store is not modified. Cases E-Context and E-Rem-Eval: follows from the induction hypothesis. Cases E-Rem-Err: follows from the definition of a well-typed store and the fact that σ is well-typed. Cases E-Rem-OK and E-Conc-Rem: follows from the preconditions of T-Remote which is assumed to apply. Case E-Conc-Lab: Using the induction hypothesis and Theorem 2. □

Proof of Configuration Coherence Preservation (Lemma 2). By structural induction on the evaluation rules. Cases E-Context, E-Rem-Eval, and E-Conc-Lab: follows from the induction hypothesis. Cases E-Conc-Rem, E-Rem-OK, E-Rem-Err, E-Broad, and E-Peerof: follows from the assumption with the observation that no new labels or peers are created. Cases E-App, E-Match, E-PMatch, E-Deref-Err, and E-Deref-OK: from the observation that coherence is preserved under substitution. □

Lemma 3 (Single Reference Creation). *Let $\Omega \vDash \mathcal{A}$ and $\pi : \mathsf{p} \in \Omega$ such that Ω is picked from a consistent and coherent sequence of configurations with e and a well-typed store σ under $\mathcal{A}; \Gamma$ be given. If $\mathcal{A}; \Gamma \vdash \mathsf{e} : \mathsf{T}$ on p and $\Omega; \sigma \triangleright \mathsf{e} \rightarrow^{\pi} \Omega'; \sigma' \triangleright \mathsf{e}'$ then $dom\ \sigma' = dom\ \sigma$ or $\sigma' = \sigma, \ell_{\mathsf{T}}$ at $\pi^{\mathsf{p}} : \mathsf{e}''$ or $\sigma' = \sigma, \ell_{\mathsf{T}}$ at $\pi^{\mathsf{p}} : \bot$.*

Proof. By structural induction on the evaluation then either the first case follows for E-App, E-Peerof, E-Broad, E-Match, E-PMatch, E-Deref-OK, E-Deref-Err, E-Rem-Move, and E-Dist because the store is left unchanged. For the cases E-Context, E-Rem-Eval, and E-Conc-Lab then the lemma follows using the induction hypothesis. For the cases E-Rem-OK, E-Rem-Err, and E-Conc-Rem then the last case follows from the observation that a single reference is being added to the store. □

Proof of Non-Circular Store (Theorem 3). Using induction on the number of evaluations performed. First, by definition the empty store $\cdot$ is non-circular. Next, assume that σ is non-circular. From the single reference creation lemma (Lemma 3) we know that after an evaluation then $\text{dom}\ \sigma' = \text{dom}\ \sigma$ or $\sigma' = \sigma, \ell_\mathsf{T}$ at $\pi^\mathsf{p} : \mathsf{e}''$ or $\sigma' = \sigma, \ell_\mathsf{T}$ at $\pi^\mathsf{p} : \bot$.

The first case holds by another reasoning by induction on the number of derivations where we attempt to prove that a reference not in the domain of σ cannot occur in its codomain. The inductive argument is as follows: the base case where the store is empty is vacuously true. Reasoning by contradiction, assume that the label to be added to the domain of σ already occurs in its codomain. The only rules that allow adding a reference are E-Rem-OK and E-Conc-Rem whose precondition requires that the label not be in the codomain. Armed with this result, observe that the only rule worth discussing that doesn't degenerate to $\sigma' = \sigma$ is E-Conc-Lab. Observe that its precondition prevents a cycle of length 1 from occurring. Moreover, the result we are armed with prevents any cycle of length larger than 1 from occurring. Therefore σ' is non-circular.

The second case can only happen if the evaluation rule used is E-Rem-OK or E-Conc-Rem. In both cases the precondition requires that the newly created reference to not be in σ's domain nor codomain. Therefore, by definition of non-circularity, σ' is non-circular.

The last case is by definition non-circular. □

Proof of Local Placement (Theorem 4). By structural induction on the typing derivations. All cases either follow from the induction hypothesis and from the observation that every typing judgement in the preconditions requires the expression to type on the same peer. The only exception is the T-Remote rule. In that case it is crucial to observe that the remote expression is typed only under the variable names of the captured expressions which are placed on the peer where the remote execution occurs. In other words, the slice of that typing environment is identical to itself which justifies using the induction hypothesis. □

Proof of Correct Reference Placement (Corollary 1). The first result, that $\mathsf{P} = \overline{\mathsf{p}_2}$ follows immediately from the type preservation theorem (Theorem 2). The second result, that $\mathsf{p}_3 = \mathsf{p}_{2,i}$ for some i follows from the precondition of T-Label and the type preservation theorem. □

Proof of Architectural Soundness (Theorem 5). By structural induction on the evaluation rules done on the first evaluation then all cases either are trivial or follow directly from the induction hypothesis. Only two cases are not direct.

Case E-Deref-OK: σ_1 and σ_2 agree on all expressions except for those at π_2. We consider two cases, the label being dereferenced is and is not on π_2. It it is not then by the definition of agreeing stores the result holds. Since the two peers are connected then the E-Deref-OK rule can be applied to derive different expressions. However this contradicts the assumption that the term is well-typed because T-Deref requires there to be a tie between p_1 and p_2 and there is none.

Cases E-Conc-Rem and E-Rem-Eval: The argument as the case of E-Deref-OK holds. With the exception that the contradiction happens with T-Remote. □

Received 2023-04-14; accepted 2023-08-27