Panopticon: A Universal Method Invocation Library for Rust

XUEYING QIN, University of Southern Denmark, Denmark and The University of Edinburgh, UK

In this paper, we present a universal method invocation (UMI) library for Rust. The UMI library supports location transparency by encapsulating the message-passing details and provides programmers an interface that allows them to migrate a monolithic program into a distributed setting, while preserving the semantics and without massive changes to the syntax of the program. This study provides a perspective on designing distributed systems: A monolithic program can be viewed as an abstraction of a distributed program, specifying *what* functionalities a program attempts to achieve instead of *how* these functionalities are achieved in a distributed setting by abstracting away the details of distributed memory management and message passing over a network.

CCS Concepts: • **Theory of computation** → **Operational semantics**; • **Computing methodologies** → **Distributed programming languages**.

Additional Key Words and Phrases: remote procedure call, memory management, ownership

1 INTRODUCTION

Distributed computing has extensive application in areas such as cloud computing, big data processing, web services, and blockchain systems, driving the development of modern, large-scale, and resilient software systems. Distributed systems offer many significant advantages such as scalability, fault tolerance, resource sharing, and geographical distribution.

However, compared to monolithic systems, distributed systems are more complex and challenging to design, implement, test and debug due to the necessity for coordination, synchronisation, and communication among distributed components. Therefore, it is a common practice to start with a monolithic design when implementing a system, as this approach simplifies the initial implementation and deployment process. Such a monolithic system can later be re-structured and migrated into a distributed design when it needs to be expanded to a larger scale. However, it still requires non-trivial effort to migrate a monolithic system into a distributed design.

1.1 Contributions

To address these issues in the design and implementation of a distributed system as well as migrating a monolithic systems into a distributed setting, we propose our design of a UMI library in Rust. The design of the UMI library shares the same underlying idea of the *remote procedural call* (RPC) [Nelson 1981], where a method invocation on an object can be executed on a different node within the same network, abstracting over the underlying message-passing details. Such a design allows programmers to model a distributed system focusing on *what* functional features are required instead of *how* these functional features are achieved via complicated network communications. Moreover, with this framework, programmers can migrate applications from a monolithic design to a distributed architecture without massive changes to source code or the needs of high-level expertise in microservices. Last but not least, by choosing Rust as the language for implementing the UMI framework, we are able to avoid distributed memory management hassles like distributed garbage collection while extending Rust's memory safety and data-racing free guarantees into the distributed setting.

In summary, we make following contributions:

Author's address: Xueying Qin, University of Southern Denmark, Odense, Denmark and The University of Edinburgh, Edinburgh, UK, xyqin@imada.sdu.dk.

- We provide a usable *Rust implementation of the UMI framework* (section 3) and demonstrate how such a framework can be used in migrating monolithic programs into a distributed setting (section 3.7).
- We present a distributed borrow checking algorithm and an extended lifetime management mechanism for distributed memory management (section 3.5 and section 3.6).
- We formalise the *structural operational semantics* based on Pearce [2021]'s featherweight Rust (FR) for a core calculus of monolithic Rust programs and distributed Rust programs written in the UMI framework (section 4).
- We prove a *location transparency theorem*: With the UMI framework, when a monolithic program is deployed to multiple nodes, its semantics is preserved (section 4).

1.2 Limitations

There are some limitations of the UMI framework worth mentioning upfront. Firstly, the UMI framework does not handle network communication errors. Since technically it is difficult to handle these errors in a distributed program while maintaining the same interface of its monolithic counterpart without the language feature throwing and handling exceptions. Also, we plan to integrate this framework within micro-services platforms, where server errors are managed by cloud service providers. Supervision strategies can be employed to take snapshots and restart from failures, ensuring these issues do not pose a critical problem for the UMI framework's design.

In addition, both the implementation and the formalisation of the UMI framework are deterministic and sequential. In the current stage, such a design decision is sufficient to demonstrate of core concepts of the UMI framework including location transparency and memory safety. However, as potential future work, it would be nice to model the UMI framework that accounts for concurrency.

Moreover, there are limitations in the formal system we have presented, for instance, functions and structs are missing. These issues are caused by the formal system we build upon. Many different styles of formal systems of Rust have been surveyed for formalising the UMI framework, however it is rather hard to find a formal system that models sufficient core features of the surface language of Rust. We will discuss related formalisation of Rust in section 5. In order to have a better formalisation of our system, it is required to have a better formalisation of the surface language of Rust, which is out of the scope of this work.

2 BACKGROUND

In distributed computing, an RPC allows a method invocation to be executed on another computer on a shared network. One application of RPCs is that in an object-oriented programming paradigm, it enables a method to be invoked on an object stored on a different machine and exchange data across the network. Such a remote method invocation has the same encoding as a local invocation, without the programmer explicitly coding the details for the remote interaction. However, it is hard to support *location transparency*, i.e., in most existing frameworks (e.g., Java RMI [Pitt and McNiff 2001; Wollrath et al. 1996]), remote invocations do not have *the same semantics* as local invocations. In addition, *memory management* is hard in a distributed setting, for example, distributed garbage collection is complicated.

Rust [Klabnik and Nichols 2018] is a high-level system programming language which *guarantees memory safety* and *prevents data races* by its *ownership system* for memory management and *borrow checker* for tracking object lifetime of all references in a program during compilation. Since Rust's semantics guarantees memory safety, we can extend such guarantees to the distributed computing setting, allowing us to design a RPC framework that provides safe remote method invocations.

2.1 Remote Procedure Calls

The basic idea behind an RPC system is to make a remote invocation appear like a local invocation, abstracting away the underlying communication mechanisms like message passing and network protocols and simplifying distributed computing by providing a familiar programming model. When a client program calls a procedure, an RPC system will handle the task of transferring the procedure call request to the remote server, along with any necessary parameters or states. The server then executes the requested procedure and sends the results back to the client.

RPCs are particularly useful in distributed systems, where different components of an application are running on separate processes or machines. It allows these components to communicate and share resources efficiently, as if they were part of a single program. There are many common applications of RPCs. For instance, RPCs are used in distributed file systems, such as Network File Systems (NFS) [Corbin 2012; Tay and Ananda 1990], to enable clients to access and manipulate files on remote servers transparently. In designing web services, RPCs form the basis of many web service protocols, such as Simple Object Access Protocol (SOAP) [Box et al. 2000; Gudgin et al. 2003], which allows applications to communicate over the internet using XML-based messaging. In modern microservices architectures, RPCs are often used for inter-process communication between different microservices, enabling them to collaborate and share functionality. In object-oriented programming, RPCs are commonly implemented as remote method invocations (RMI), enabling objects on different machines to interact with each other [Seemakhupt et al. 2023; Sriraman and Wenisch 2018]. gRPC [gRPC Authors 2024] is a high performance RPC framework for such an application. The core feature of RMIs is that objects can interact with each other by invoking methods and passing data across the network. This is the application domain of this study.

2.2 Rust

As a system programming language with emphasises on safety, performance, and concurrency, Rust is designed to prevent some common programming errors, such as data races and dereferencing null pointers. Rust achieves these goals through its distinctive features of the ownership system, borrow checking, and lifetimes.

In Rust, each value has a variable designated as its *owner*. Each value can only have one owner at a time, and when the owner goes out of scope, the value is *dropped*, i.e., deallocated from memory. This ownership model ensures resources to be managed correctly without the need for a garbage collector. The ownership system serves as the basis for Rust's memory safety.

Rust allows functions and data structures to create references to values *without* taking ownership. This is called *borrowing*. When a value is borrowed, the original owner cannot modify the value until the borrowing ends. The *borrow checker* is part of Rust's compiler, which ensures that references are used safely and do not result in dangling pointers or other memory issues. Borrowing can be either mutable or immutable, where mutable references have additional constraints to prevent data races and undefined behaviour. *Lifetimes* in Rust express how long references should be valid. They assist the borrow checker in ensuring that references do not outlive the data they point to.

With the design of the ownership system, borrow checking mechanism, and lifetimes, Rust enforces strict memory safety guarantees, i.e., that all references point to valid memory, without requiring a garbage collector. These features also ensure that Rust programs do not have dace races by allowing only one mutable reference at a time or multiple immutable references.

```rust
#[proxy_me]
struct A { arg: u32 }
impl A {
  #[umi_init]
  new(arg: u32) -> A { A {arg: arg} }
  #[umi_struct_method]
  by_value(&self, a: A) {...}
  #[umi_struct_method]
  by_ref(&self, &a: A) {...}
  #[umi_struct_method]
  by_mut_ref(&self, &mut a: A) {...} }
```

```rust
fn main() {
  let a_remote =
    remote!(addr, A::new(10));

  let a_local1 = A::new(1);
  let a_local2 = A::new(2);
  let mut a_local3 = A::new(3);

  a_remote.by_value(a_local1);
  a_remote.by_ref(&a_local2);
  a_remote.by_mut_ref(&mut a_local3); }
```

Fig. 1. Migrating A Monolithic Application into A Distributed Setting with UMI

3 THE DESIGN AND IMPLEMENTATION OF THE RUST UMI LIBRARY

In the section, we present our design and implementation of the UMI framework as a library in Rust. With such a library, a monolithic program can be migrated into a distributed program while preserving the semantics of the original monolithic program.

3.1 Overview

To give a high-level overview of the design, in figure 1, we introduce an example of migrating a monolithic program into a distributed setting, by adding the *macros* provided by the UMI library. In this example program which allocates instances of the `struct A` and calls methods on them, the macro `#[proxy_me]` implicitly translates the declared type `A` from a `struct` that can only refer to local resources to an `enum` that can either hold local resources or be a *proxy* that refers to resources held on a remote node. The initialisation method is translated by the macro `#[umi_init]` to create an instance of the `enum A` instead of an instance of the `struct A`. Other methods are also translated by macros to allow both an invocation on a local instance of `A` and an invocation on a proxy of `A`. To create a proxy instance, the macro `remote!(address, ...)` is used, while the syntax of the initialisation of a local instance is unchanged. The invocations of the methods defined for translated `struct A` take the same form of the invocation of those original methods. We have to use different macros to identify syntactic scopes of different code blocks in a piece of Rust program to correctly produce the corresponding translations.

An invocation on a proxy is encapsulated into a serialised message and sent to the destination node of which the address is the address stored in the proxy, and then the message is deserialised and the invocation is executed at the destination. After the execution, the result of the invocation is again put into a serialised message and passed back to the calling node to be deserialised.

3.2 The Design of the Translation

As we have seen in the example discussed above, the syntax of a monolithic program is translated into a distributed program by a set of macros. For a declared `struct`, the macro `#[proxy_me]` performs the translation:

$$\texttt{struct}\ A\ \{\ \mathit{fields}\ \} \rightsquigarrow \texttt{enum}\ A\ \{\ \mathit{Local}\ (\ \mathit{fields}\ ), \mathit{Remote}\ (\ \mathit{Address}, \mathit{ID}, \mathit{IsOwner}\ )\ \}$$

where the *Address* is the type of address of the node which stores the resource of a proxy, the *ID* is the identifier of a proxy's resource in the resource table that will be discussed in section 3.3, and *IsOwner* denotes whether a proxy is an owned reference or a borrow reference. This macro can

translates an `enum` to allow it to represent a proxy by adding a new constructor which is the proxy:

$$\texttt{enum}\ A\ \{\ \textit{variants}\ \} \rightsquigarrow \texttt{enum}\ A\ \{\ \textit{variants}, \textit{Remote}\ (\ \textit{Address}, \textit{ID}, \textit{IsOwner}\ )\ \}$$

The translation of a `enum` does not affect its initialisation method, however, the translation of a `struct` requires its initialisation method to be changed accordingly — instead of creating an instance of a type which is a `struct`, an instance of a *Local* variant of an `enum` is created. For instance, in the example shown in figure 1, the `new(arg:u32)` method is translated by `#[umi_init]` into:

```
new(arg: u32) -> A { A::Local {arg: arg} }
```

The macro `#[umi_struct_method]` performs the translation of other methods of a `struct`. For instance, the method `by_value(&self, a:A)` is translated into:

```
fn by_value(&self, a: A) {
  match &self {
    Local(...) => { /* do something */ },
    Remote(...) => { /* remote do something */ }
}}
```

Note that within the pattern matching block for the *Remote* variant, the invocation is firstly put into a message and serialised. Then the serialised message is passed to the address stored in the proxy, and gets deserialised and executed. The result is again put into a message and get serialised. Once it is returned back to the original node, the result is extracted from the deserialised message. Such a communication process between nodes via sending and receiving serialisation/deserialisation messages is completely generated by the macro, freeing programmers from dealing with the message passing complexity. As for a method of a translated `enum`, the macro `#[umi_enum_method]` adds an additional pattern matching block for the proxy variant to the existing pattern matching.

3.3 Resource Management

To be able to use the UMI library for executing programs that access and manipulate memories of different nodes within a network, resources and computations need to be made available to and well-managed by all nodes in the network.

Firstly, a node should be able to store resources owned by different machines and deallocate those resources according to their lifetime. In Rust, if some resources are owned by a reference on the same node, and the reference has reached the end of its lifetime, these resources will be deallocated from the memory. With such a design, resources that are not owned by any reference on the same node are automatically deallocated. However, in our UMI library, while some resources on a node n_1 are not owned by any reference on the same node, they can be owned by a reference on a different node n_2. Although these resources do not have a local owner, the deallocation should not happen until the remote owner reaches the end of its lifetime. To achieve this goal, on each UMI server, we design a *resource table* shown in figure 2 on the left, which has the same lifetime as the server. We used it to identify and manage local resources involved in remote computations. Note that the ID in an entry of the table is the ID field in a corresponding proxy, which is globally unique. If a variable is created locally, it will be put into the table once it is passed into a remote

ID	Resource
uid0	...
uid1	...
...	...

Full Path Name	Type Information
A::new	u32, A
A::foo1	(&A, A), ()
...	...

Fig. 2. A resource table (L) and a method registration table (R)

computation. The entry will not be removed until the remote computation finishes. If a variable is created via a remote call, it will be put into table on creation and will be deallocated when its remote owner decides that it should be dropped.

Secondly, we need to make all nodes aware of all methods that can be invoked on a proxy in order to make computations available on all nodes. To achieve this goal, we use a macro to register all methods that are available for remote invocations in a *method registration table* shown in figure 2 on the right. The macro takes the form of `register!(name, arg_types, return_type)`. The method registration table holds the full path name, argument types, and return type of methods. When a serialised invocation message, which takes the form of a plain string, is received by a node, the method to be invoked is deserialised and reconstructed according to the type information recorded in the registration table.

3.4 Passing Remote Invocations via Messages

As briefly discussed in section 3.1 and section 3.3, remote invocations and results of executions are implicitly communicated via serialised and deserialised messages among nodes. We use the Serde [serde-rs 2023] framework to serialise and deserialise these messages and Rust data structures.

There are different types of messages for passing remote invocations. For instance, a remote invocation sent to an receiving node is represented as an invocation message which taking the form of `Message::Invoke(fname, variables, invoke_op)`, where `fname` is the full path name of the method, each variable is annotated with its ownership information (owned or immutably/mutably borrowed), and `invoke_op` specifies the ownership information of the return value. The result of the execution of a remote invocation is passed back to the calling node via a return message taking the form of `Message::Return(return_var)`, where the `return_var` is also annotated with its ownership information. Another important type of messages is the deallocation message which takes the form of `Message::Drop(id)`, where the `id` corresponds to an entry key in the resource table shown in figure 2. Such a message instructs some remotely owned resources to be deallocated.

3.5 Extending Borrow Checking into Distributed Settings

To execute a deserialised remote method invocation on the node which receives the invocation, the first step is to gather serialised data as well as the ownership information of each variable involved in the method. In this step, we do not perform any reconstruction of these variables; instead, variables are simply prepared in an appropriate format that can be reconstructed during the execution of the method. Such an format is implemented as `Argument`, which keeps the information of the variables related to borrow checking, and stores the data of the variable. The detailed implementation of this step is shown in listing 1, listing 2, and listing 3.

A serialised variable has a label indicating whether it is a piece of data copied or moved from the caller (`OwnedLocal`), a remote reference owned by the caller (`OwnedRemote`), a remote reference immutably borrowed by the caller (`RefRemote`), or a remote reference mutably borrowed by the caller (`MutRefRemote`). As shown in listing 1, if a variable is serialised data, which is copied or moved from the caller, it will be kept as serialised, since the deserialisation and reconstruction process will happen during the invocation of the method (line 5). If a variable is a proxy which is located at the receiver, then it will be obtained from the resource table shown in figure 2. According to its ownership information, if it is moved, then the corresponding entry will be removed from the resource table (line 6 — line 9).

Listing 1. Gathering variables from an invocation message: `OwnedLocal` and `OwnedRemote`

```
1 Message::Invoke(fname, variables, invoke_op) => {
2   let mut arguments: Vec<Argument> = Vec::new();
```

```
3   for v in &variables {
4     match v {
5       Variable::OwnedLocal(s) => { arguments.push(Argument::Serialised(s.clone())); },
6       Variable::OwnedRemote(serialise_remote, addr, id) => {
7         if addr == &local_address {
8           let (owned, is_ref) = mvtable.remove(id).unwrap().into_inner();
9           let arg_ref = Argument::Owned(owned); arguments.push(arg_ref);
10        } else { arguments.push(Argument::Serialised(serialise_remote.to_string())); }},
11      ...}} ... }
```

If it is immutably borrowed, as shown in listing 2. then the corresponding entry will be immutably borrowed from the table (line 5 — line 11).

Listing 2. Gathering variables from an invocation message: `RefRemote`

```
1 Message::Invoke(fname, variables, invoke_op) => {
2   let mut arguments: Vec<Argument> = Vec::new();
3   for v in &variables {
4     match v {...
5       Variable::RefRemote(serialise_remote, addr, id) => {
6         if addr == &local_address {
7           let borrow = mvtable.get(id).unwrap().borrow();
8           let ptr: *const (Box<dyn Any + Send + Sync>, bool) = &*borrow;
9           unsafe {
10            let back: &(Box<dyn Any + Send + Sync>, bool) = ptr.as_ref().unwrap();
11            let arg_ref = Argument::Ref(&back.0, back.1); arguments.push(arg_ref); }
12        } else {
13          arguments.push(Argument::RemoteRef(serialise_remote.to_string())); }}, ...}} ... }
```

If it is mutably borrowed, as shown in listing 3 then the corresponding entry will be mutably borrowed from the table and updated after the execution (line 5 — line 11).

Listing 3. Gathering variables from an invocation message: `MutRefRemote`

```
1 Message::Invoke(fname, variables, invoke_op) => {
2   let mut arguments: Vec<Argument> = Vec::new();
3   for v in &variables {
4     match v { ...
5       Variable::MutRefRemote(serialise_remote, addr, id) => {
6         if addr == &local_address {
7           let mut borrow_mut = mvtable.get(id).unwrap().borrow_mut();
8           let ptr: *mut (Box<dyn Any + Send + Sync>, bool) = &mut *borrow_mut;
9           unsafe {
10            let back: &mut (Box<dyn Any + Send + Sync>, bool) = ptr.as_mut().unwrap();
11            let arg_ref = Argument::MutRef(&mut back.0, back.1); arguments.push(arg_ref); }
12        } else {
13          arguments.push(Argument::RemoteMutRef(serialise_remote.to_string())); }}}} ... }
```

If an argument is a proxy that is *not* located at the caller, as shown in three `else`-cases demonstrated in listing 1 (line 10), listing 2 (line 12 — line 13), and listing 3 (line 12 — line 13), then the proxy will be passed into the method without any additional modification.

Once the information about all variables is gathered and processed, the invocation will be executed and the result will then be sent back to the caller. The implementation of the execution of this invocation is shown in listing 4 and listing 5. The method information, mainly ownership and type information of the arguments, and return value of a method are retrieved from the registration

table shown in figure 2 on the right. During the execution of the method via `f.call(arguments)`, serialised arguments and boxed argument entries retrieved from the resource table are reconstructed according to the registered type information.

The result of an execution is provided in two formats, serialised data and a boxed data entry. These two formats are used according to the required ownership information of the return value. As shown in listing 4, if the method produces an owned result annotated with `InvokeOp::Owned`, whether the serialised data `res` represents some local resources or a proxy, it will be kept as the serialised form and sent back via a return message (line 8). If the method is an initialisation call sent by the macro `remote!(...)` annotated with `InvokeOp::Init`, the boxed entry data will be inserted into the resource table and a unique `id` will be generated. In the return message, the address of the receiver, the `id`, and the ownership status which is `true` are included for the caller to construct a proxy that owns such a data entry on the receiver (line 9 — line 12).

Listing 4. Executing an invocation and returning the result: `Owned` and `Init`

```
1  Message::Invoke(fname, variables, invoke_op) => { ...
2    let f: &str = &*fname;
3    match lrtable.get(f) {
4      Some(f) => {
5        let ((res, is_local), b) = f.call(arguments);
6        let res_message: Message;
7        match invoke_op {
8          InvokeOp::Owned => { res_message = Message::Return(ReturnVar::Owned(res)); },
9          InvokeOp::Init => {
10           let id = (SystemTime::now(), m_id_gen.next());
11           mvtable.insert(id.clone(), RefCell::new((b, false))); // b is the resource
12           res_message = Message::Return(ReturnVar::OwnedInit(local_address, id, true));}, ... }
13       response(stream, res_message); },
14     None => { /* report unregistered function */ }} }, ...
```

For a return value that is an immutably or mutably borrowed reference, as shown in listing 5 there are two situations. If the borrowed reference is local to the receiver, the reference itself will be inserted into the resource table identified by a generated unique `id`. Such an `id` and the address of the receiver will be sent back to the caller for creating an proxy that mirrors this borrowed reference (line 9 — line 12 and line 15 — line 18). However, if a borrowed reference is not local to the receiver, meaning it already mirrors a reference on a different node, then it will not be stored in the resource table, instead, the serialised version of it will be sent back to the caller in a return message (line 13 and line 19).

Listing 5. Executing an invocation and returning the result: `Ref` and `MutRef`

```
1  Message::Invoke(fname, variables, invoke_op) => { ...
2    let f: &str = &*fname;
3    match lrtable.get(f) {
4      Some(f) => {
5        let ((res, is_local), b) = f.call(arguments); // b is a reference
6        let res_message: Message;
7        match invoke_op { ...
8          InvokeOp::Ref => {
9            if is_local {
10             let id = (SystemTime::now(), m_id_gen.next());
11             mvtable.insert(id.clone(), RefCell::new((b, true)));
12             res_message = Message::Return(ReturnVar::RefMirror(local_address, id));
```

```
13            } else { res_message = Message::Return(ReturnVar::RefBorrow(res)); }},
14          InvokeOp::MutRef => {
15            if is_local {
16              let id = (SystemTime::now(), m_id_gen.next());
17              mvtable.insert(id.clone(), RefCell::new((b, true)));
18              res_message = Message::Return(ReturnVar::MutRefMirror(local_address, id));
19            } else { res_message = Message::Return(ReturnVar::MutRefBorrow(res)); }}}
20        response(stream, res_message); },
21      None => { /* report unregistered function */ }}}, ...
```

3.6 Extending Lifetime Management to Distributed Settings

Recall that in section 3.3, we have introduced storing and deallocating remotely owned resources on a node. Here we discuss the design and implementation of a remote deallocation in detail.

In a monolithic Rust program, when a variable that owns some resources reaches the end of its lifetime, in most cases, out of a program's scope, the resources it owns will be automatically deallocated. We extend this feature to the distributed setting. As illustrated in listing 6, when the given proxy `a_proxy` is initialised, some resources are allocated to the receiver node with the address `addr` (line 5). Although these resources do not have an owner on the same node, they should not be deallocated until its remote owner `a_proxy` reaches the end of its lifetime (line 8).

Listing 6. An example of a remote deallocation

```
1 // on caller
2 fn main() { ...
3   // the data of a_proxy is in the table on the receiver with addr
4   // but it is owned by the caller and will be deallocated when its owner decides to drop it
5   let a_proxy = remote!(addr, A::new(10));
6   ...
7   a_proxy.by_value(...)
8 } // a_proxy is out of scope, its data on the remote machine is dropped
```

For monolithic programs, the deallocation is achieved via the `drop` method in the destructor trait `Drop`, which in most cases is automatically implemented for Rust types. We extend this `drop` method to handle the deallocation of remotely owned resources. The implementation is shown in listing 7. When a proxy that owns some resources on a node reaches the end of its lifetime, a serialised deallocation message is automatically sent to the node that holds these resources.

Listing 7. The implementation of a remote deallocation

```
1 impl Drop for #name {
2   fn drop(&mut self) {
3     match self {
4       Self::Remote(addr, id, is_owner) => {
5         if is_owner.load(Ordering::Relaxed)
6           { let msg = Message::Drop(*id); send(*addr, msg).unwrap(); }},
7       _ => {} }}}
```

As shown below, once a deallocation message is received by the targeted receiver, the entry with the corresponding `id` will be removed from the resource table (i.e., the `mvtable` in the listing).

```
1 Message::Drop(id) => { mvtable.remove(&id); }
```

3.7 Case Studies

We demonstrate the application of our UMI framework — to facilitate migrating monolithic programs to a distributed setting — by present two example distributed reminder programs migrated from a monolithic using UMI. Note that for the simplicity of presentation, we omit implementation details for some `structs` like `Entry`, application server nodes, and importing packages.

3.7.1 A Pull Reminder Application. Listing 8 and 9 show a distributed reminder application that extracts events that are due on request. It is very similar to the example shown in figure 1. Specifically, listing 8 shows that with those macros provided in the UMI library (line 1, line 4, line 6, and line 9), the `struct ReadyReminderServer` referring to local resources is translated into an `enum` referring to either local resources or remote resources. In addition, its methods can be invoke on both cases.

Listing 8. A Pull Ready Reminder Struct

```
1  #[proxy_me]
2  pub struct ReadyReminderServer { entries: BinaryHeap<Entry> }
3  impl ReadyReminderServer {
4    #[umi_init]
5    pub fn new() -> ReadyReminderServer { ReadyReminderServer { entries: BinaryHeap::new() } }
6    #[umi_struct_method]
7    pub fn submit_event(&mut self, content: String, ready_at: SystemTime)
8      { let entry = Entry::new(content, ready_at); (&mut self.entries).push(entry); }
9    #[umi_struct_method]
10   pub fn extract_event(&mut self) -> Option<Entry> {
11     let first = (&self.entries).peek();
12     match first {
13       Some(entry) => {
14         if entry.get_time() <= &SystemTime::now() {
15           let e = (&mut self.entries).pop(); return e; }
16         else { return None; }},
17       None => { return None; }}}}
```

For the reminder application client shown in listing 9, the only change to make it run in a distributed setting, i.e., initialising the `ReadyReminderServer` on a different (known) node in the network, is to replace the local initialisation for a monolithic program (line 2) with the remote initialisation that creates a proxy whose resources held on the remote node (line 3).

Listing 9. A Pull Ready Reminder Application Client

```
1  fn main() {
2    // let mut r = ReadyReminderServer::new(); // Old monolithic ReadyReminderServer creation
3    let mut r = remote!("127.0.0.1:3335", ReadyReminderServer::new, ReadyReminderServer);
4    r.submit_event("Goodbye World!".to_string(), SystemTime::now() + Duration::new(3, 0));
5    r.submit_event("Hello World!".to_string(), SystemTime::now() + Duration::new(1, 0));
6    println!("The first event is: {:?}", r.extract_event());
7    thread::sleep(Duration::new(4, 0));
8    println!("The first event is: {:?}", r.extract_event());
9    println!("The first event is: {:?}", r.extract_event()); }
```

3.7.2 A Push Reminder Application. Listing 10 and 11 show a distributed reminder application that actively extracts events that are due and pushes notifications to these events' corresponding reminder clients. This application demonstrates how a closure representing a notification (as a `CallBack`) can be passed to and executed on a remote node.

Listing 10 again shows the implementation of a `ReadyReminderServer` that is migrated from a monolithic implementation. Comparing to the original monolithic implementation, in addition to the usage of UMI marcos (line 1, line 4, line 6, and line 10), there are two more small changes. Firstly, each event `Entry` now records the address of the reminder client to which the event should be pushed as a `callback_addr` shown in line 8 — line 9. Secondly, instead of directly executing the callback of an event that is due (line 18), the callback is pushed to the event's corresponding reminder client to be executed (line 19).

Listing 10. A Push Ready Reminder

```
1  #[proxy_me]
2  pub struct ReadyReminderServer { entries: BinaryHeap<Entry> }
3  impl ReadyReminderServer {
4    #[umi_init]
5    pub fn new() -> ReadyReminderServer { ReadyReminderServer { entries: BinaryHeap::new() } }
6    #[umi_struct_method]
7    pub fn submit_event
8      (&mut self, callback: CallBack, callback_addr: String, ready_at: SystemTime) {
9      let entry = Entry::new(callback, callback_addr, ready_at);(&mut self.entries).push(entry);}
10   #[umi_struct_method]
11   pub fn run(&mut self) {
12     while (&self.entries).len() > 0 {
13       let first = (&self.entries).peek();
14       match first {
15         Some(entry) => {
16           if entry.get_time() <= &SystemTime::now() {
17             let e = (&mut self.entries).pop(); let c = e.clone().unwrap().callback;
18             // c.execute(); // Old monolithic ReadyReminderServer executes a callback locally
19             let c_addr = e.unwrap().callback_addr; remote!(c_addr, c);
20           } else { continue; }},
21         None => { continue; } }}}}
```

For the push reminder application client shown in listing 11, comparing to the pull reminder application client, more changes are made to the original monolithic push reminder application client. Similar to the migration of the pull reminder application client, the local initialisation (line 5) is replaced with a remote initialisation (line 6). Moreover, this application client is made into a `UMIEndpoint` assigned with a fixed address, a resource table, and a method registration table, enabling that, when its events are due, callbacks can be pushed back and executed according to its address.

Listing 11. A Push Ready Reminder Application Client

```
1  fn main() {
2    let mut table = RegistryTable::new();
3    let vtable = Arc::new(Mutex::new(ResourceTable::new()));
4    let handle = thread::spawn(move || {
5      // let mut r = ReadyReminderServer::new(); // Old monolithic ReadyReminderServer creation
6      let mut r = remote!("127.0.0.1:3335", ReadyReminderServer::new, ReadyReminderServer);
7      r.submit_event(CallBack::new("Goodbye World!".to_string()), "127.0.0.1:3336".to_string(),
8        SystemTime::now() + Duration::new(8, 0));
9      r.submit_event(CallBack::new("Hello World!".to_string()), "127.0.0.1:3336".to_string(),
10       SystemTime::now() + Duration::new(5, 0));
11     r.run(); });
12   let mut listener = UMIEndpoint::new("127.0.0.1:3336");
13   listener.start(table, vtable);
```

t ::=	`let mut` $x = t; t$	declaration
|	$w := t$	assignment
|	$t; t$	sequence
|	$()$	unit
|	$\{t\}$	block
|	`box` t	heap allocation
|	$\&w$	immutable borrow
|	`&mut` w	mutable borrow
|	$\#w$	move
|	$!w$	copy
|	v	value

(Term ($\mathcal{T}$))

w ::=	x	variable
|	$*w$	dereference

(LVal)

v ::=	$\perp$	
|	$()$	unit
|	i	integer
|	$\ell^\bullet$	owned reference
|	ℓ°	borrowed reference

(Value ($\mathcal{V}$))

$\ell \in \mathbb{A}$ (Location)

Fig. 3. The revised syntax of FR

```
14    handle.join().unwrap();}
```

Next, we discuss the formalisation of core concepts of the UMI library base on formalised operational semantics of a core language of Rust, and present the *location transparency theorem*.

4 THE OPERATIONAL SEMANTICS

We first provide a small-step operational semantics of a core language of Rust based on Pearce [2021]'s featherweight Rust (FR), which captures the core features of Rust including copy- and move-semantics, owned and immutably/mutably borrowed references, and lexical lifetimes. We then present dFR, which extends FR to include distributed features of the UMI framework such as remote copy- and move-semantics as well as remote references. We show that such an extension preserves the semantics of FR, and therefore, the type safety claims of FR is preserved by dFR.

Note that the goal of the formalisation we present is not to design a new formal semantics or type system of Rust, rather, we aim to utilise existing work which provide formal semantics accompanied with a type system with type safety proofs for a core surface language of Rust (i.e., FR). Hence, here we only focus on discussing the semantics extension and the location transparency theorem. For readers who are curious about FR's type system, please refer to appendix B.

4.1 The Revised Syntax and Semantics of FR

We present a revised syntax of FR allowing us to express substitutions easier in figure 3. Note that $\&w$ and `&mut` w represent immutable and mutable borrowing, where `&[mut]`w represents either a immutable or a mutable borrow term. $\ell^\bullet$ and ℓ° are owned and borrowed references where ℓ represents a location in a program state. A copy term is denoted as $!w$. In addition, different from the original FR and Rust which do not have explicit syntax for move, we use $\#w$ to express move explicitly. The notion of program state $\mathcal{S}$ is introduced in figure 4, which is a mapping from a location to a tuple of value and lifetime. When a value is replaced or its lifetime is expired, it will be removed from the program state.

We provide the operations of recursively removing values based on a location ℓ and a lifetime k from the state:

$$\begin{aligned} \left(\mathcal{S} \otimes (\ell^\bullet \mapsto (v, k))\right) \setminus \ell^\bullet &= \mathcal{S} \setminus v \\ \mathcal{S} \setminus v &= \mathcal{S} \quad \text{otherwise} \end{aligned}$$

$$\mathcal{S} : \mathbb{A} \rightharpoonup \mathcal{V} \times \mathcal{L}$$
$$\mathcal{S} \mid \ell \mapsto (v, m) \quad \text{where: } \ell \in \textbf{dom } \mathcal{S} \qquad \text{(update)}$$
$$\mathcal{S} \otimes \ell \mapsto (v, m) \quad \text{where: } \ell \notin \textbf{dom } \mathcal{S} \qquad \text{(extend)}$$

Fig. 4. The program state

and respectively:

$$\mathcal{S} \setminus k\ (\ell) = \begin{cases} \mathcal{S}(\ell) & \text{if } \mathcal{S}(\ell) = (v, k) \\ \textbf{undefined} & \text{otherwise} \end{cases}$$

Figure 5 shows the small-step operational semantics of FR, which takes the form of a reduction $S, t \xrightarrow{k} S', t'$, where S is the program state before the evaluation of the term t, and S', t' are the program state and the term after the evaluation. k is a lifetime, which we omit when it is irrelevant to the evaluation of a term.

Evaluating a copy term simply makes a copy of a value v at a given location ℓ, without modifying the program state, whereas evaluating a move term moves a value v out of a given location ℓ. A heap allocation `box` v puts the value v into a fresh location ℓ and gives it the *global lifetime* $\top$, which outlives all other lifetimes. The rule for evaluating a borrow term produces a borrowed reference of the give location ℓ. Assignment places a given value v' in the location ℓ, and recursively deallocates the old value v from the program state. Note that assignments to immutably borrowed references are prohibited by the type system provided by Pearce's [2021] original work, the discussion of the type system is omitted here since focus on the analysis of the SOS of FR programs. The evaluation of a declaration allocates a given value v to a fresh location ℓ and substitutes latter occurrence of the declared variable x with the owned reference $\ell^\bullet$.

$$\frac{\mathcal{S}(\ell) = (v, m)}{\mathcal{S}, !\ell^\bullet \longrightarrow \mathcal{S}, v}\ (\textsc{Copy}) \qquad \frac{}{\mathcal{S} \otimes \ell \mapsto (v, m), \#\ell^\bullet \longrightarrow \mathcal{S} \otimes \ell \mapsto \bot, v}\ (\textsc{Move})$$

$$\frac{\ell \notin \textbf{dom } \mathcal{S}}{\mathcal{S}, \texttt{box}\ v \longrightarrow \mathcal{S} \otimes \ell \mapsto (v, \top), \ell^\bullet}\ (\textsc{Box}) \qquad \frac{\ell \in \textbf{dom } \mathcal{S}}{\mathcal{S}, \&[\texttt{mut}]\ell^\bullet \longrightarrow \mathcal{S}, \ell^\circ}\ (\textsc{Borrow})$$

$$\frac{}{\mathcal{S} \otimes \ell \mapsto (v, m), \ell^\bullet := v' \longrightarrow (\mathcal{S} \setminus v) \otimes \ell \mapsto (v', m), ()}\ (\textsc{Assign Owned})$$

$$\frac{}{\mathcal{S} \otimes \ell \mapsto (v, m), \ell^\circ := v' \longrightarrow (\mathcal{S} \setminus v) \otimes \ell \mapsto (v', m), ()}\ (\textsc{Assign Borrowed})$$

$$\frac{\ell \notin \textbf{dom } \mathcal{S}}{\mathcal{S}, \texttt{let mut}\ x = v; t \xrightarrow{k} \mathcal{S} \otimes \ell \mapsto (v, k), t[\ell^\bullet / x]}\ (\textsc{Decl})$$

$$\frac{}{\mathcal{S}, \{v\} \xrightarrow{k} \mathcal{S} \setminus suc\ k, v}\ (\textsc{Block (base)}) \qquad \frac{\mathcal{S}, t \xrightarrow{suc\ k} \mathcal{S}', t'}{\mathcal{S}, \{t\} \xrightarrow{k} \mathcal{S}', \{t'\}}\ (\textsc{Block (suc)})$$

$$\frac{\ell \in \textbf{dom } \mathcal{S}}{\mathcal{S}, \ell^\bullet; t \longrightarrow \mathcal{S} \setminus \ell^\bullet, t}\ (\textsc{Seq-OwnedRef}) \qquad \frac{\ell \in \textbf{dom } \mathcal{S}}{\mathcal{S}, \ell^\circ; t \longrightarrow \mathcal{S}, t}\ (\textsc{Seq-BorrowedRef})$$

$$\frac{}{\mathcal{S}, i; t \longrightarrow \mathcal{S}, t}\ (\textsc{Seq-Int}) \qquad \frac{}{\mathcal{S}, (); t \longrightarrow \mathcal{S}, t}\ (\textsc{Seq-Unit})$$

Fig. 5. The semantics of revised FR

$$E ::= [\cdot] \mid E; t \mid v; E \mid \texttt{let mut}\ x = E; t \mid \texttt{let mut}\ x = v; E \mid \{E\} \mid \texttt{box}\ E \mid w = E$$

$$\frac{S, t \longrightarrow S, t'}{S, E[t] \longrightarrow S', E[t']}\ (\textsc{Context})$$

Fig. 6. Evaluation context

FR's lifetimes are based on the the lexical structure of programs. A block's lifetime k is based on the depth of the block. A block with deeper depth *suc* k lives shorter than a block with depth k. The evaluation of a block is the evaluation of the term inside the block. At the end of the evaluation, a single value v is obtained and values that have short lifetime than the current block are deallocated from the program state. The reduction rules for the evaluation of sequences are intuitive. We highlight the case for a sequence of which the first term is an owned reference. After evaluating the owned reference, it is recursively deallocated from the program state.

An evaluation context is a term with a placeholder $[\cdot]$. $E[t]$ is a term obtained by replacing the placeholder with a term t. The evaluation context and reduction rule for the evaluation context are shown in figure 6.

Next, we discuss the syntax and semantics of dFR, which extends FR with distributed computation features including remote references and values.

4.2 The Syntax and Semantics of dFR

Figure 7 shows the syntax of dFR, which is the syntax of FR discussed in section 4.1 extended with the remote declaration $\texttt{let mut@}n\ x = t; t$, remote heap allocation $\texttt{box}\ t$, remote values $v@n$, and remote terms $t@n$. All extensions are highlighted. Note that n is the address of a node and $\mathcal{N}$ is the set of addresses. Also, such an extension does not change the type system of FR.

Building upon the program state $\mathcal{S}$ for FR, in figure 8, we introduce the distributed program state $\mathcal{D}$, which maps addresses of nodes to their program states. The reduction rule takes the form of

t ::=	$\texttt{let mut}\ x = t; t$	declaration	
|	$\texttt{let mut@}n\ x = t; t$	remote declaration	
|	$w := t$	assignment	
|	$t; t$	sequence	
|	$()$	unit	
|	$\{t\}$	block	
|	$\texttt{box}\ t$	heap allocation	
|	$\texttt{box@}n\ t$	remote heap allocation	
|	$\&w$	immutable borrow	
|	$\&\texttt{mut}\ w$	mutable borrow	
|	$\#w$	move	
|	$!w$	copy	
|	v	value	
|	$v@n$	remote value	

Term ($\mathcal{T}$)

t_d ::=	$t@n$	(Remote Term)	
w ::=	x	variable	
|	$*w$	dereference	

LVal

v ::=	$\perp$	
|	$()$	unit
|	i	integer
|	$\ell^\bullet$	owned reference
|	ℓ°	borrowed reference

Value ($\mathcal{V}$)

$\ell \in \mathbb{A}$ (Location)

$n \in \mathcal{N}$ (Node)

Fig. 7. The syntax of dFR

$$\mathcal{D} : \mathcal{N} \rightharpoonup \mathcal{S} \qquad nt \in \mathcal{N} \times \mathbb{1} + \mathcal{N} \times \mathcal{T}$$
$$C : (nt)^* \qquad \text{Configuration} : \mathcal{D}, C$$

Fig. 8. Distributed program state and configuration stack

$\mathcal{D}, C \longrightarrow \mathcal{D}', C'$, where C and C' are *configuration stacks*. For each reduction, the term on the top of a configuration stack C gets evaluated. The element of the configuration stack can either be a pair of an address and a hole $(n, ?)$ or a pair of an address and a term (n, t). Utilising the distributed program state and the configuration stack, we provide the semantics of dFR in figure 9. We explain the reduction rules for each operation in detail.

To evaluate copying a remotely owned reference $\ell^\bullet@n'$ on a node with address n, the first step shown in the rule Copy (s1) is to update the configuration stack by changing the $(n, !\ell^\bullet@n')$ to $(n, ?)$, and pushing a new address-term pair $(n', !\ell^\bullet)$ to be evaluated onto the stack. It models that the computation is passed to the node n', which stores the resource owned by the reference $\ell^\bullet@n'$. Then the rule Copy (s2) indicates that the copy term $!\ell^\bullet$ gets evaluated on the node n', and the resulting value annotated with the address n' is passed back to fill in the hole.

Similarly, as for the semantics of moving the value out of a remote owned reference, the first step shown in Move (s1) is to replace the term on the node n with a hole, and pass the copy term to the node n' to evaluate. Then as shown in Move (s2), the resulting value v annotated with the address n' is passed back to the node n at the end of the evaluation to fill in the hole, and the value of the location ℓ at the program state of the node n' is replaced by $\perp$, indicating that the value is moved out of the location ℓ at the node n'.

The rules Box (s1) and Box (s2) show the evaluation of a remote heap allocation $\texttt{box}@n'\ v$ on the node n. Firstly, the heap allocation is passed to the node n' to be evaluated, and a hole on the node n is created and pushed onto the configuration stack. The value v is stored in a fresh location ℓ at the program state of the node n' and assigned with the lifetime $\top$ since it is a heap allocation, hence a owned reference $\ell^\bullet$ is created in the node n'. Such a owned reference is then passed back to the node n allowing the node n to own the location ℓ created by the heap allocation on the node n'. At the end of the evaluation, as shown in the rule Box (s2), on the top of the configuration stack, the hole create on the node n is filled by the owned remote reference $\ell^\bullet@n'$

The rules Borrow (s1) and Borrow (s2) show the evaluation of immutable and mutable borrow terms. On a node n, to borrow a remotely owned reference from a different node n', firstly a hole is created and pushed waiting for a term to be passed back, and the borrow term $\&[\texttt{mut}]\ell^\bullet@n'$ is passed to the node n' to be evaluated. After it being evaluated on the node n', the resulting remotely borrowed reference $\ell^\circ@n'$ is passed back into the node n in the hole on the configuration stack.

A remote assignment to an owned reference $\ell^\bullet@n' := v'$ on the node n assigns a new value v' to its remotely owned reference on the node n', which is shown in the rule Assign Owned (s1) and Assign Owned (s2). The first step is again leaving a hole awaiting to be filled on the configuration stack and passing the assignment to the node n' to be evaluated. In the next step, the evaluation of the assignment on the node n' modifies the program state on n' by recursively deallocates the old value v which is stored in the location ℓ. And then the location ℓ on the node n' is assigned with the new value v'. Since the assignment produces only a unit value $()$, it will be passed back and fill in the hole on the configuration stack.

The evaluation of a remote assignment to a borrowed reference is similar shown in rules Assign Borrowed (s1) and Assign Borrowed (s2). Note that again, since dFR extends FR without any modification of FR's type system, same to assignments in FR, remote assignments to immutable references in dFR are also prohibited by the type system.

$$\frac{\mathcal{D}(n')(\ell) = (v, m)}{\mathcal{D}, C \mathbin{+\!\!+} (n, !\ell^\bullet @ n') \longrightarrow \mathcal{D}, C \mathbin{+\!\!+} (n, ?) \mathbin{+\!\!+} (n', !\ell^\bullet)} \text{ (Copy (s1))}$$

$$\frac{\mathcal{D}(n')(\ell) = (v, m)}{\mathcal{D}, C \mathbin{+\!\!+} (n, ?) \mathbin{+\!\!+} (n', !\ell^\bullet) \longrightarrow \mathcal{D}, C \mathbin{+\!\!+} (n, v @ n')} \text{ (Copy (s2))}$$

$$\frac{}{\begin{array}{c}\mathcal{D} \otimes (n' \mapsto \mathcal{S} \otimes \ell \mapsto (v, m)), C \mathbin{+\!\!+} (n, \#\ell^\bullet @ n') \longrightarrow \\ \mathcal{D} \otimes (n' \mapsto \mathcal{S} \otimes \ell \mapsto (v, m)), C \mathbin{+\!\!+} (n, ?) \mathbin{+\!\!+} (n', \#\ell^\bullet)\end{array}} \text{ (Move (s1))}$$

$$\frac{}{\begin{array}{c}\mathcal{D} \otimes (n' \mapsto \mathcal{S} \otimes \ell \mapsto (v, m)), C \mathbin{+\!\!+} (n, ?) \mathbin{+\!\!+} (n', \#\ell^\bullet) \longrightarrow \\ \mathcal{D} \otimes (n' \mapsto \mathcal{S} \otimes \ell \mapsto \bot), C \mathbin{+\!\!+} (n, v @ n')\end{array}} \text{ (Move (s2))}$$

$$\frac{}{\mathcal{D}, C \mathbin{+\!\!+} (n, (\texttt{box} @ n'\ v)) \longrightarrow \mathcal{D}, C \mathbin{+\!\!+} (n, ?) \mathbin{+\!\!+} (n', \texttt{box}\ v)} \text{ (Box (s1))}$$

$$\frac{\ell \notin \mathbf{dom}\ \mathcal{D}(n')}{\mathcal{D}, C \mathbin{+\!\!+} (n, ?) \mathbin{+\!\!+} (n', \texttt{box}\ v) \longrightarrow \mathcal{D} \mid (n' \mapsto \mathcal{D}(n') \otimes \ell \mapsto (v, \top)), C \mathbin{+\!\!+} (n, \ell^\bullet @ n')} \text{ (Box (s2))}$$

$$\frac{\ell \in \mathbf{dom}\ \mathcal{D}(n')}{\mathcal{D}, C \mathbin{+\!\!+} (n, \&[\texttt{mut}]\ell^\bullet @ n') \longrightarrow \mathcal{D}, C \mathbin{+\!\!+} (n, ?) \mathbin{+\!\!+} (n', \&[\texttt{mut}]\ell^\bullet)} \text{ (Borrow (s1))}$$

$$\frac{\ell \in \mathbf{dom}\ \mathcal{D}(n')}{\mathcal{D}, C \mathbin{+\!\!+} (n, ?) \mathbin{+\!\!+} (n', \&[\texttt{mut}]\ell^\bullet) \longrightarrow \mathcal{D}, C \mathbin{+\!\!+} (n, \ell^\circ @ n')} \text{ (Borrow (s2))}$$

$$\frac{}{\begin{array}{c}\mathcal{D} \otimes (n' \mapsto \mathcal{S} \otimes \ell \mapsto (v, m)), C \mathbin{+\!\!+} (n, \ell^\circ @ n' := v') \longrightarrow \\ \mathcal{D} \otimes (n' \mapsto \mathcal{S} \otimes \ell \mapsto (v, m)), C \mathbin{+\!\!+} (n, ?) \mathbin{+\!\!+} (n', \ell^\circ := v')\end{array}} \text{ (Assign Borrowed (s1))}$$

$$\frac{}{\begin{array}{c}\mathcal{D} \otimes (n' \mapsto \mathcal{S} \otimes \ell \mapsto (v, m)), C \mathbin{+\!\!+} (n, ?) \mathbin{+\!\!+} (n', \ell^\circ := v') \longrightarrow \\ \mathcal{D} \otimes (n' \mapsto \mathcal{S} \setminus v \otimes \ell \mapsto (v', m)), C \mathbin{+\!\!+} (n, ())\end{array}} \text{ (Assign Borrowed (s2))}$$

$$\frac{}{\begin{array}{c}\mathcal{D} \otimes (n' \mapsto \mathcal{S} \otimes \ell \mapsto (v, m)), C \mathbin{+\!\!+} (n, \ell^\circ @ n' := v') \longrightarrow \\ \mathcal{D} \otimes (n' \mapsto \mathcal{S} \otimes \ell \mapsto (v, m)), C \mathbin{+\!\!+} (n, ?) \mathbin{+\!\!+} (n', \ell^\circ := v')\end{array}} \text{ (Assign Borrowed (s1))}$$

$$\frac{}{\begin{array}{c}\mathcal{D} \otimes (n' \mapsto \mathcal{S} \otimes \ell \mapsto (v, m)), C \mathbin{+\!\!+} (n, ?) \mathbin{+\!\!+} (n', \ell^\circ := v') \longrightarrow \\ \mathcal{D} \otimes (n' \mapsto \mathcal{S} \setminus v \otimes \ell \mapsto (v', m)), C \mathbin{+\!\!+} (n, ())\end{array}} \text{ (Assign Borrowed (s2))}$$

$$\frac{}{\mathcal{D}, C \mathbin{+\!\!+} (n, \texttt{let mut@}n'\ x = v; t) \longrightarrow \mathcal{D}, C \mathbin{+\!\!+} (n, t[?/x]) \mathbin{+\!\!+} (n', \texttt{let mut}\ x = v; x)} \text{ (Decl (s1))}$$

$$\frac{\ell \notin \mathbf{dom}\ \mathcal{D}(n')}{\begin{array}{c}\mathcal{D}, C \mathbin{+\!\!+} (n, t[?/x]) \mathbin{+\!\!+} (n', \texttt{let mut}\ x = v; x) \longrightarrow \\ \mathcal{D} \otimes (n' \mapsto \mathcal{S} \otimes \ell \mapsto (v, k)), C \mathbin{+\!\!+} (n, t[\ell^\bullet @ n'/x])\end{array}} \text{ (Decl (s2))}$$

$$\frac{\mathcal{S}, t \longrightarrow \mathcal{S}', t'}{\mathcal{D} \otimes (n \mapsto \mathcal{S}), C \mathbin{+\!\!+} (n, t) \longrightarrow \mathcal{D} \otimes (n \mapsto \mathcal{S}'), C \mathbin{+\!\!+} (n, t')} \text{ (Local Terms)}$$

$$\frac{\mathcal{D} \otimes (n' \mapsto \mathcal{S}), C \mathbin{+\!\!+} (n', t) \longrightarrow \mathcal{D} \otimes (n' \mapsto \mathcal{S}'), C \mathbin{+\!\!+} (n', t')}{\mathcal{D} \otimes (n' \mapsto \mathcal{S}), C \mathbin{+\!\!+} (n, t @ n') \longrightarrow \mathcal{D} \otimes (n' \mapsto \mathcal{S}'), C \mathbin{+\!\!+} (n, t' @ n')} \text{ (Remote Terms)}$$

Fig. 9. The semantics of dFR

Demonstrated in Decl (s1) and Decl (s2), the evaluation of a remote declaration is more complicated. The first step shown in Decl (s1) leaves a hole to be filled by the resulting term in the substitution of the declared variable x. Then the declaration is passed to the node n' to be evaluated. Shown in Decl (s2), once a fresh location ℓ on the node n' is allocated with the value v, the owned reference $\ell^\bullet$ will then be passed back to the node n and all occurrences of the declared variable x on the node n will be substituted with the remote owned reference $\ell^\bullet@n'$.

Lastly, all reduction rules for evaluating terms presented in FR are adapted into reduction rules for evaluating dFR via the rule Local Term. As for the evaluation of remote terms, shown in the rule Remote Term, if a local term t on the node n' is evaluated into t', then when it is treated as a remote term $t@n'$ on the node n, it will be evaluated to a remote term $t'@n'$ on the node n.

In the next section, we present and prove a location transparency theorem, which states that when translating a monolithic program written in FR into a distributed program written in dFR, the semantics of the monolithic program is preserved.

4.3 Preservation of Semantics when Translating a FR Program into a dFR Program

As we have mentioned in previous sections, by extending FR into dFR, the type system and static borrow checking of the validity of owning, immutably borrowing and mutably borrowing resources remain unchanged. By formalising the semantics of FR and dFR, we would like to show that, when we flatten a distributed program in dFR into a monolithic program in FR, the flattened result of the execution of the distributed program should be the same as the result of the execution of the flattened single node program. By demonstrating that the distributed program preserves the original semantics of the monolithic program, we can then conclude that the memory safety guarantees provided by FR's type system and static checking can be extended into distributed program in dFR.

Formally, we state this semantic preservation property of the distributed extension dFR in the *location transparency theorem* 4.1.

Before giving the definition of the location transparency theorem, we define a relation $\rightsquigarrow$ that *fuses* two reduction steps of an operation in dFR introduced in section 4.2. The reduction relation for each dFR operation is decomposed into two steps just for presentational purposes. The location transparency theorem states that fusing two reduction steps of a dFR operation gives the semantics of the operation, which preserves the semantics of its corresponding operation defined in FR as a single step reduction relation. Formally, the *fusion relation* is defined as:

Definition 4.1 (Fusion Relation). *For any term that contains remote components, for instance, a remote copy, a fusion relation is defined by two reductions in the operational model for dFR. For any term that does not contain remote components, for instance, a local term, a fusion relation is just a single reduction.*

$$
\begin{aligned}
&\mathcal{D}, C +\!\!+ (n, t) \rightsquigarrow \mathcal{D}', C +\!\!+ (n, t') \quad \textit{iff} \\
&\qquad (t \textit{ is remote} \implies \exists t'', C', \mathcal{D}''.\ \mathcal{D}, C +\!\!+ (n, t) \longrightarrow \mathcal{D}'', C' +\!\!+ (n, t'') \\
&\qquad\qquad\qquad \wedge \quad \mathcal{D}'', C' +\!\!+ (n, t'') \longrightarrow \mathcal{D}', C +\!\!+ (n, t')) \\
&\vee (\neg(t \textit{ is remote}) \implies \mathcal{D}, C +\!\!+ (n, t) \longrightarrow \mathcal{D}', C +\!\!+ (n, t'))
\end{aligned}
$$

With the definition of the fusion relation, we state the location transparency theorem as below. Note that the reverse direction of the theorem does not hold. Since to extend a given single node program into a distributed program by allocating the program state on arbitrary nodes and making the term involved in the execution containing remote components that live on arbitrary nodes may not lead to constructing a distributed program that can be successfully executed. However, because

our goal is to show that the distributed program preserves the same behaviour as if it is a single node program, having only one direction in the theorem is sufficient for our claim.

Theorem 4.1 (Location Transparency). For any term t, given an initial distributed program state $\mathcal{D}$ and an initial single node program state $\mathcal{S}$, where the flattened distributed program state equals to the single node program state, if a distributed execution of a term t that may be a remote term or contain remote component with the distributed program state $\mathcal{D}$ results in a distributed program state $\mathcal{D}'$ and a value v which can be either remote or local, then the execution of the flattened term t with the single node program state $\mathcal{S}$ will gives a state $\mathcal{S}'$ and value v', where the flattened resulting distributed program state $|\mathcal{D}'|$ equals to the S' and the flattened value v equals to v'. We make locations on all nodes distinct to simplify the proof.

$$\forall t \in \mathcal{T}.\ \mathcal{D}, C +\!\!+ (n,t) \rightsquigarrow \mathcal{D}', C +\!\!+ (n,v) \wedge |\mathcal{D}| = \mathcal{S}$$
$$\Rightarrow$$
$$\mathcal{S}, t|_{@} \longrightarrow \mathcal{S}', v' \wedge |\mathcal{D}'| = \mathcal{S}' \wedge v|_{@} = v'$$

where $|\cdot|$ and $\cdot|_{@}$ are operators that erase addresses of nodes from distributed program states and terms. Specifically, $|\cdot|$ is defined as:

$$|\mathcal{D}| = \bigcup_{\forall n \in \mathbf{dom}\,\mathcal{D}} \mathcal{D}(n)$$

and $\cdot|_{@}$ is defined as:

$$(\texttt{let mut@}n\ x = t; t)|_{@} = \texttt{let mut}\ x = t; t \qquad v@n|_{@} = v \qquad t@n|_{@} = t|_{@}$$
$$\texttt{box@}n\ t|_{@} = \texttt{box}\ t \qquad \ell^{\bullet}@n := v|_{@} = \ell^{\bullet} := v \qquad \ell^{\circ}@n := v|_{@} = \ell^{\circ} := v$$

Since a term t is always a closed term, we prove the theorem 4.1 by structural induction on the term t. The proofs are presented in appendix A, which focuses on presenting proofs of the cases concerning the remote extensions in dFR's reduction rules given in section 4.2.

4.4 Summary

In section 3, we have discussed the design and implementation of a UMI framework as a library in Rust. The core designed concepts of such a library — extending Rust's memory safety guarantees into a distributed setting — is presented in the formalisation of a distributed extension of a core calculus of Rust in this section. By proving the location transparency theorem 4.1, we demonstrate that a distributed program developed using the UMI framework preserves the semantics of a monolithic program from which it is translated. Therefore, with our UMI framework, Rust's memory safety guarantees provided by its type system, lifetime and ownership system, and borrow checking mechanism are indeed extended into a distributed setting.

5 RELATED WORK

Design and Implementations of Remote Procedural Call. Described in Nelson's [1981] thesis, RPC allows programs in separate address spaces to communicate synchronously. By experimenting with different implementations of RPC, Nelson [1981] argues that RPC is an efficient and effective programming tool for distributed systems. Java RMI [Pitt and McNiff 2001; Wollrath et al. 1996] implements the concept of RPC in a object-orientated programming language. It outlines a model for distributed objects within the Java environment which allows Java objects to communicate across different address spaces. This RMI framework is designed to integrate seamlessly with the Java language, preserving as much of the Java object model's semantics as possible. As for its memory management mechanism, it contains a design of distributed garbage collection, ensuring

that remote objects which are no longer referenced by any client should be automatically garbage collected. However, the Stub object in this framework does not always preserves the semantics of the Java object model. In addition, tarpc [Google 2024] implemented as a Rust library shares a similar underlying idea.

Our UMI framework is designed to be truly integrated with Rust as it is semantic preserving. The idea of location transparency and seamless integration of distributed computing presented in this UMI framework is related to the “seamless" distributed computing model where distributed programs maintain the same syntax and semantics as single-node executions, using node annotations for distribution, without requiring explicit communication handling [Fredriksson and Ghica 2012] and the formal model Krivine Nets presented by Fredriksson and Ghica [2014], which extends the classic Krivine abstract machine to support distributed execution. This extension allows a seamless and transparent RPC mechanism to handle higher-order functions without transmitting the actual code. Krivine Nets enable the seamless integration of distributed computing into programming languages by eliminating explicit communication and process management from source code.

The RPC calculus presented by Cooper and Wadler [2009] explores the design and implementation of a symmetrical location-aware programming language atop a stateless server. They address the challenge of maintaining control state transparently within a programming language, despite the stateless nature of web servers, which typically do not retain client-specific session information. They outline the RPC calculus λ_{rpc}, which is enriched with location annotations to indicate where code should execute, supporting semantics where computation steps can occur at designated locations; and a translation from λ_{rpc} to a first-order client-server calculus (λ_{cs}), which models an asymmetrical client-server environment. The subsequent work for such a RPC calculus present by [Choi et al. 2020] proposes a polymorphic RPC calculus that extends the typed RPC calculus with polymorphic locations. It introduces a new polymorphic RPC calculus that allows programmers to write succinct multi-tier programs using polymorphic location constructs and defines a type system for the polymorphic RPC calculus and proves its type soundness. In addition, it develops a monomorphisation translation that converts polymorphic RPC terms into monomorphic typed RPC terms, allowing existing slicing compilation methods for client-server models to be used. The type and semantic correctness of the monomorphisation translation are proven.

Distributed Memory Management. There are different approaches for distributed memory management for different usage scenarios. Distributed shared memory (DSM) provides the illusion of a shared memory space across multiple nodes, making it easier for programmers to write distributed applications as if they were writing for a shared-memory multiprocessor [Nitzberg and Lo 1991]. Distributed garbage collection used in Java RMI [Wollrath et al. 1996] as been an essential yet challenging research topic in distributed memory management. Abdullahi and Ringwood [1998] offers a comprehensive review of distributed garbage collection (GC) schemes applicable to autonomous systems connected by a network, particularly in the context of Internet programming languages such as Java. It highlights the evolution of garbage collection from single-address-space collectors to distributed systems due to the increasing prominence of languages like Java in Internet applications. Designed for the object-oriented programming language with actor model Pony [Clebsch et al. 2017], Orca is a concurrent garbage collection algorithm which manages memory without requiring stop-the-world pauses or synchronisation mechanisms, enabling zero-copy message passing and mutable data sharing among actors. Perhaps more relevant to our memory management mechanism, the region system Reggio [Arvidsson et al. 2023] accompanied with a type system for Verona [Microsoft 2019], which is a concurrent object-oriented programming language, organises objects into isolated regions, each with its own memory management strategy. It addresses the challenge of allowing manual memory management while maintaining memory safety by utilising

a combination of region-based memory partitioning and an ownership type system. In this paper, we take a different approach — we prove a concise yet safe memory management mechanism by extending Rust's ownership and borrow checking system into a distributed setting while abstracting over the message passing details.

Formalisations of Rust and Their Limitations. The formalism of a core calculus of the UMI framework presented in this paper is based on the formalism of a core calculus of Rust described by Pearce [2021]. Although as mentioned in previous sections, we are not completely satisfied with this formalism, it is the most suitable choice comparing to all other approaches of formalising Rust's semantics we have surveyed. In the following paragraphs, we discuss these approaches in detail.

There are many different approaches to formalise Rust, from different perspectives and aiming for different application domains. RustBelt [Jung et al. 2017] provide a formalised continuation-passing style MIR — λ_{Rust} — mechanised using Iris [Jung et al. 2015]. Similarly, the formalisation presented by Matsushita et al. [2021] is inspired by λ_{Rust}. These approaches do not provide a formal model which is close to the source-level language of Rust. Hence, it is not convenient for us to use it as the basis for formalising the distributed extensions of the features of the UMI framework.

Different from RustBelt, there are also attempts of formalising Rust from a source-level language perspective. For instance, Oxide [Weiss et al. 2021] attempts to formalise near-complete source-level Rust language features, providing a type system and small-step operational semantics. Due to the level of the complexity and obscurity of this formalism, we find that it is rather hard for us to gain a clear understanding of the modelling of Rust's borrow checking and non-lexical lifetime. In addition, without a concise and consistent presentation of the syntax, type system and formal semantics of the core features of Rust, it is again hard for us to use such a formalism to reason about properties of Rust programs and be convinced that the type system is indeed sound.

Back to the formalisation we choose to build our distributed language extension upon, instead of modelling the full Rust language, Pearce [2021] formalises a core calculus of Rust, which is FR, emphasising on the understanding of borrowing and lifetime of Rust. Although the possible extensions of FR to include tuples and functions are discussed, this formalism does not include essential Rust language features such as tuples, structs, functions and closures which makes it too minimalist. Also, the way that the let-binding is modelled made it hard to do substitutions, which is in my opinion the obstacle of having functions as a part of the formalism. In addition, its lexical treatment of Rust's lifetime is already obsolete.

Due to the reality of lack of a concise formalism of the source-level Rust that captures all key concepts of Rust's language features, our formalism of the UMI framework as a distributed extension of Rust is also not completely satisfying. However, it does demonstrate Rust's core memory management mechanisms, and allows us to prove the semantic preservation property of distributed programs implemented using the UMI framework. To have formalism of UMI capturing more language features would require us to develop yet another formalism of source-level Rust, which is out of the scope of this work.

6 CONCLUSION AND FURTHER WORK

In this paper, we first present our design and implementation of a UMI framework for Rust. This UMI framework allows programmers to express distributed computation in the same form of monolithic computation, abstracting away the internet communication complications and message passing details. As a distributed extension of Rust, our UMI framework extends Rust's memory safety guarantees into a distributed setting. We then present the formalism of a core calculus focusing on the core features relating to distributed memory management mechanisms in the UMI framework. We present FR, which is a core calculus of the surface language of Rust, formalising

the key concepts of Rust's ownership and lifetime in memory management. And we extend FR into dFR, to include distributed features of the UMI framework. By showing that distributed programs written in dFR preserves the semantics of monolithic programs written in FR via proving the location transparency theorem, we can conclude that the memory safety guarantees provided by Rust can be extended to distributed programs written with our UMI framework.

In the future, we would like to conduct some quantitative evaluations for the UMI framework. For instance, we could measure the performance overhead of distributed programs written using the UMI framework comparing to those using TCP requests or using other libraries such as tarpc. In addition, conducting user studies can be beneficial for assessing how straightforward the UMI framework is to use when writing distributed programs.

REFERENCES

Saleh E. Abdullahi and Graem A. Ringwood. 1998. Garbage collecting the Internet: a survey of distributed garbage collection. *ACM Comput. Surv.* 30, 3 (sep 1998), 330–373. https://doi.org/10.1145/292469.292471

Ellen Arvidsson, Elias Castegren, Sylvan Clebsch, Sophia Drossopoulou, James Noble, Matthew J. Parkinson, and Tobias Wrigstad. 2023. Reference Capabilities for Flexible Memory Management. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 270 (oct 2023), 31 pages. https://doi.org/10.1145/3622846

Don Box, David Ehnebuske, Gopal Kakivaya, Andrew Layman, Noah Mendelsohn, Henrik Frystyk Nielsen, Satish Thatte, and Dave Winer. 2000. Simple object access protocol (SOAP) 1.1.

Kwanghoon Choi, James Cheney, Simon Fowler, and Sam Lindley. 2020. A polymorphic RPC calculus. *Science of Computer Programming* 197 (2020), 102499.

Sylvan Clebsch, Juliana Franco, Sophia Drossopoulou, Albert Mingkun Yang, Tobias Wrigstad, and Jan Vitek. 2017. Orca: GC and type system co-design for actor languages. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 72 (oct 2017), 28 pages. https://doi.org/10.1145/3133896

Ezra EK Cooper and Philip Wadler. 2009. The RPC calculus. In *Proceedings of the 11th ACM SIGPLAN conference on Principles and practice of declarative programming*. 231–242.

John R Corbin. 2012. *The art of distributed applications: programming techniques for remote procedure calls.* Springer Science & Business Media.

Olle Fredriksson and Dan R Ghica. 2012. Seamless distributed computing from the geometry of interaction. In *International Symposium on Trustworthy Global Computing*. Springer, 34–48.

Olle Fredriksson and Dan R. Ghica. 2014. Krivine nets: a semantic foundation for distributed execution. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming* (Gothenburg, Sweden) *(ICFP '14)*. Association for Computing Machinery, New York, NY, USA, 349–361. https://doi.org/10.1145/2628136.2628152

Google. 2024. tarpc-v0.35.0. https://github.com/google/tarpc/tree/master

gRPC Authors. 2024. gRPC: A high performance, open source universal RPC framework. https://grpc.io

Martin Gudgin, Marc Hadley, Noah Mendelsohn, Jean-Jacques Moreau, Henrik Frystyk Nielsen, Anish Karmarkar, and Yves Lafon. 2003. SOAP Version 1.2. *W3C recommendation* 24 (2003), 12.

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2017. RustBelt: securing the foundations of the Rust programming language. *Proc. ACM Program. Lang.* 2, POPL, Article 66 (dec 2017), 34 pages. https://doi.org/10.1145/3158154

Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. *SIGPLAN Not.* 50, 1 (Jan. 2015), 637–650. https://doi.org/10.1145/2775051.2676980

Steve Klabnik and Carol Nichols. 2018. *The Rust Programming Language.* No Starch Press, USA.

Yusuke Matsushita, Takeshi Tsukada, and Naoki Kobayashi. 2021. RustHorn: CHC-based Verification for Rust Programs. *ACM Trans. Program. Lang. Syst.* 43, 4, Article 15 (Oct. 2021), 54 pages. https://doi.org/10.1145/3462205

Microsoft. 2019. Project Verona. https://www.microsoft.com/en-us/research/project/project-verona/

Bruce Jay Nelson. 1981. *Remote procedure call.* Ph. D. Dissertation. USA. AAI8204168.

Bill Nitzberg and Virginia Lo. 1991. Distributed shared memory: A survey of issues and algorithms. *Computer* 24, 8 (1991), 52–60.

David J. Pearce. 2021. A Lightweight Formalism for Reference Lifetimes and Borrowing in Rust. *ACM Trans. Program. Lang. Syst.* 43, 1, Article 3 (apr 2021), 73 pages. https://doi.org/10.1145/3443420

Esmond Pitt and Kathy McNiff. 2001. *Java.rmi: The Remote Method Invocation Guide.* Addison-Wesley Longman Publishing Co., Inc., USA.

Korakit Seemakhupt, Brent E Stephens, Samira Khan, Sihang Liu, Hassan Wassel, Soheil Hassas Yeganeh, Alex C Snoeren, Arvind Krishnamurthy, David E Culler, and Henry M Levy. 2023. A cloud-scale characterization of remote procedure

calls. In *Proceedings of the 29th Symposium on Operating Systems Principles*. 498–514.
serde-rs. 2023. *Serde*. https://serde.rs
Akshitha Sriraman and Thomas F Wenisch. 2018. μ suite: a benchmark suite for microservices. In *2018 ieee international symposium on workload characterization (iiswc)*. IEEE, 1–12.
Beng Hang Tay and Akkihebbal L Ananda. 1990. A survey of remote procedure calls. *ACM SIGOPS Operating Systems Review* 24, 3 (1990), 68–79.
Aaron Weiss, Olek Gierczak, Daniel Patterson, and Amal Ahmed. 2021. Oxide: The Essence of Rust. arXiv:1903.00982 [cs.PL] https://arxiv.org/abs/1903.00982.
Ann Wollrath, Roger Riggs, and Jim Waldo. 1996. A distributed object model for the javaTM system. In *Proceedings of the 2nd Conference on USENIX Conference on Object-Oriented Technologies (COOTS) - Volume 2* (Toronto, Ontario, Canada) *(COOTS'96)*. USENIX Association, USA, 17.

A THE PROOF FOR THE LOCATION TRANSPARENCY THEOREM

Case *Copy*. We assume that before the evaluation, given the distributed program state and monolithic program state as below:

$$\mathcal{D} = \mathcal{D}_0 \otimes (n' \mapsto (\ell \mapsto (v, m))) \qquad \mathcal{S} = \mathcal{S}_0 \otimes (\ell \mapsto (v, m)) \qquad |\mathcal{D}_0| = \mathcal{S}_0$$

We can say that, initially, the flattened distributed program state equals to the monolithic program state:

$$|\mathcal{D}| = \mathcal{S}$$

Following the dFR's reduction rule Copy (s1) and Copy (s2), the distributed execution of a remote copy term gives:

$$\mathcal{D}_0 \otimes (n' \mapsto (\ell \mapsto (v, m))), C \mathbin{+\!\!+} (n, !\ell^\bullet @ n') \rightsquigarrow$$
$$\mathcal{D}_0 \otimes (n' \mapsto (\ell \mapsto (v, m))), C \mathbin{+\!\!+} (n, v @ n')$$

Following FR's reduction rule Copy, the monolithic execution of the flattened remote copy term gives:

$$\mathcal{S}_0 \otimes (\ell \mapsto (v, m)), !\ell^\bullet @ n'|_@ \longrightarrow \mathcal{S}_0 \otimes (\ell \mapsto (v, m)), v$$

After the evaluation, the updated distributed program state and monolithic program state are shown as below:

$$\mathcal{D}' = \mathcal{D}_0 \otimes (n' \mapsto (\ell \mapsto (v, m))) \qquad \mathcal{S}' = \mathcal{S}_0 \otimes (\ell \mapsto (v, m))$$

To conclude, after the evaluation, the flattened updated distributed program state and the updated monolithic program state are equal, and the flattened resulting value obtained from the distributed execution and the value obtained from the monolithic execution are equal:

$$|\mathcal{D}'| = \mathcal{S}' \qquad v @ n'|_@ = v$$

Hence:

$$\mathcal{D}, C \mathbin{+\!\!+} (n, !\ell^\bullet @ n') \rightsquigarrow \mathcal{D}', C \mathbin{+\!\!+} (n, v @ n') \wedge |\mathcal{D}| = \mathcal{S}$$
$$\Rightarrow$$
$$\mathcal{S}, !\ell^\bullet @ n'|_@ \longrightarrow \mathcal{S}', v \wedge |\mathcal{D}'| = \mathcal{S}' \wedge v @ n'|_@ = v$$

□

Case *Move*. We assume that before the evaluation, given the distributed program state and monolithic program state as below:

$$\mathcal{D} = \mathcal{D}_0 \otimes (n' \mapsto (\ell \mapsto (v, m))) \qquad \mathcal{S} = \mathcal{S}_0 \otimes (\ell \mapsto (v, m)) \qquad |\mathcal{D}_0| = \mathcal{S}_0$$

We can say that, initially, the flattened distributed program state equals to the monolithic program state:

$$|\mathcal{D}| = \mathcal{S}$$

Following the dFR's reduction rule Move (s1) and Move (s2), the distributed execution of a remote move term gives:

$$\mathcal{D}_0 \otimes (n' \mapsto (\ell \mapsto (v, m))), C \mathbin{+\!\!+} (n, \#\ell^\bullet @ n') \rightsquigarrow \mathcal{D}_0 \otimes (n' \mapsto (\ell \mapsto \bot)), C \mathbin{+\!\!+} (n, v @ n')$$

Following FR's reduction rule Move, the monolithic execution of the flattened remote move term gives:

$$\mathcal{S}_0 \otimes (\ell \mapsto (v, m)), \#\ell^\bullet @ n'|_{@} \longrightarrow \mathcal{S}_0 \otimes (\ell \mapsto \bot), v$$

After the evaluation, the updated distributed program state and monolithic program state are shown as below:

$$\mathcal{D}' = \mathcal{D}_0 \otimes (n' \mapsto (\ell \mapsto \bot)) \qquad \mathcal{S}' = \mathcal{S}_0 \otimes (\ell \mapsto \bot)$$

To conclude, after the evaluation, the flattened updated distributed program state and the updated monolithic program are equal, and the flatted resulting value obtained from the distributed execution and the resulting value obtained from the monolithic execution are equal:

$$|\mathcal{D}'| = |\mathcal{D}_0| \otimes (\ell \mapsto \bot) = \mathcal{S}_0 \otimes (\ell \mapsto \bot) = \mathcal{S}' \qquad v @ n'|_{@} = v$$

Hence:

$$\mathcal{D}, C \mathbin{+\!\!+} (n, \#\ell^\bullet @ n') \rightsquigarrow \mathcal{D}', C \mathbin{+\!\!+} (n, v @ n') \wedge |\mathcal{D}| = \mathcal{S}$$
$$\Rightarrow$$
$$\mathcal{S}, \#\ell^\bullet @ n'|_{@} \longrightarrow \mathcal{S}', v \wedge |\mathcal{D}'| = \mathcal{S}' \wedge v @ n'|_{@} = v$$

□

Before proving the case for Box, we introduce a lemma stating a relation between a fresh location in a distributed program state $\mathcal{D}$ and such a location in a monolithic program state $\mathcal{S}$.

Lemma A.1. *Given a distributed program state $\mathcal{D}$ and a monolithic program state $\mathcal{S}$, n is a node in the domain of $\mathcal{D}$:*

$$|\mathcal{D}| = S \wedge \ell \notin \mathbf{dom}\ \mathcal{D}(n) \Rightarrow \ell \notin \mathbf{dom}\ \mathcal{S}$$

Proof. According to theorem 4.1, locations on all nodes in $\mathcal{D}$ are distinct. Given ℓ is a fresh location in $\mathbf{dom}\ \mathcal{D}(n)$, i.e., $\ell \notin \mathbf{dom}\ \mathcal{D}(n)$, ℓ is also not in any other nodes in $\mathcal{D}$. Hence we have:

$$\ell \notin \mathbf{dom}\ |\mathcal{D}|$$

Since $|\mathcal{D}| = S$, we can conclude that:

$$\ell \notin \mathbf{dom}\ \mathcal{S}$$

□

Case *Box*. We assume that before the evaluation, given the distributed program state $\mathcal{D}$ and monolithic program state $\mathcal{S}$ where the flatten distributed program state equals to the monolithic program state. In addition, we take a fresh location ℓ:

$$|\mathcal{D}| = \mathcal{S} \qquad \ell \notin \mathbf{dom}\ \mathcal{D}(n')$$

Following the dFR's reduction rule Box (s1) and Box (s2), the distributed execution of a remote heap allocation term gives:

$$\mathcal{D}, C \mathbin{+\!\!+} (n, \mathsf{box} @ n'\ v) \rightsquigarrow \mathcal{D} \mid (n' \mapsto \mathcal{D}(n') \otimes (\ell \mapsto (v, \top))), C \mathbin{+\!\!+} (n, \ell^\bullet @ n')$$

By lemma A.1, we have:

$$\ell \notin \mathbf{dom}\ \mathcal{S}$$

Following FR's reduction rule Box, the monolithic execution of the flattened remote heap allocation term gives:

$$\mathcal{S}, (\mathsf{box@}n'\ v)|_{@} \longrightarrow \mathcal{S} \otimes (\ell \mapsto (v, \top)), \ell^{\bullet}$$

After the evaluation, the updated distributed program state and monolithic program state are shown as below:

$$\mathcal{D}' = \mathcal{D} \mid (n' \mapsto \mathcal{D}(n') \otimes (\ell \mapsto (v, \top))) \qquad \mathcal{S}' = \mathcal{S} \otimes (\ell \mapsto (v, \top))$$

The updated distributed program state is flattened into:

$$|\mathcal{D}'| = |\mathcal{D}| \otimes (\ell \mapsto (v, \top))$$

Since we have $|\mathcal{D}| = \mathcal{S}$, we can conclude that the flattened updated distributed program state and the updated monolithic program state are equal:

$$|\mathcal{D}'| = |\mathcal{D}| \otimes (\ell \mapsto (v, \top)) = \mathcal{S} \otimes (\ell \mapsto (v, \top)) = \mathcal{S}'$$

Also, the flattened remote owned reference obtained from the distributed execution equals to the owned reference obtained from the monolithic execution:

$$\ell^{\bullet}@n'|_{@} = \ell^{\bullet}$$

Hence:

$$\mathcal{D}, C +\!\!+ (n, \mathsf{box@}n'\ v) \rightsquigarrow \mathcal{D}', C +\!\!+ (n, \ell^{\bullet}) \wedge |\mathcal{D}| = \mathcal{S}$$
$$\Rightarrow$$
$$\mathcal{S}, (\mathsf{box@}n'\ v)|_{@} \longrightarrow \mathcal{S}', \ell^{\bullet} \wedge |\mathcal{D}'| = \mathcal{S}' \wedge \ell^{\bullet}@n'|_{@} = \ell^{\bullet}$$

□

Before proving the case for Borrow, we introduce a lemma stating a relation between an existing location in a distributed program state $\mathcal{D}$ and such a location in a monolithic program state $\mathcal{S}$.

Lemma A.2. *Given a distributed program state $\mathcal{D}$ and a monolithic program state $\mathcal{S}$, n is a node in the domain of $\mathcal{D}$:*

$$|\mathcal{D}| = S \wedge \ell \in \mathbf{dom}\ \mathcal{D}(n) \Rightarrow \ell \in \mathbf{dom}\ \mathcal{S}$$

Proof. Given ℓ is an existing location in $\mathbf{dom}\ \mathcal{D}(n)$, i.e., $\ell \in \mathbf{dom}\ \mathcal{D}(n)$, we have:

$$\ell \in \mathbf{dom}\ |\mathcal{D}|$$

Since $|\mathcal{D}| = S$, we can conclude that:

$$\ell \in \mathbf{dom}\ \mathcal{S}$$

□

Case *Borrow*. We assume that before the evaluation, given the distributed program state $\mathcal{D}$ and monolithic program state $\mathcal{S}$ where the flatten distributed program state equals to the monolithic program state, and ℓ is a location with an allocated value:

$$|\mathcal{D}| = \mathcal{S} \qquad \ell \in \mathbf{dom}\ \mathcal{D}(n')$$

Following the dFR's reduction rule Borrow (s1) and Borrow (s2), the distributed execution of a remote borrow term gives:

$$\mathcal{D}, C +\!\!+ (n, \&[\mathtt{mut}\ell^{\bullet}@n']) \rightsquigarrow \mathcal{D}, C +\!\!+ (n, \ell^{\circ}@n')$$

By lemma A.2, we have:

$$\ell \in \mathbf{dom}\ \mathcal{S}$$

Following FR's reduction rule Borrow, the monolithic execution of the flattened remote borrow term gives:

$$\mathcal{S}, (\&[\texttt{mut}]\ell^{\bullet}@n')|_{@} \longrightarrow \mathcal{S}, \ell^{\circ}$$

To conclude, after the evaluation, the distributed program state and the monolithic program state remain unchanged, hence they are still equal. In addition, the flattened remote borrowed reference obtained from the distributed execution equals to the borrowed reference obtained from the monolithic execution:

$$\ell^{\circ}@n'|_{@} = \ell^{\circ}$$

Hence:

$$\mathcal{D}, C +\!\!+ (n, \&[\texttt{mut}]\ell^{\bullet}@n') \rightsquigarrow \mathcal{D}, C +\!\!+ (n, \ell^{\circ}) \wedge |\mathcal{D}| = \mathcal{S}$$
$$\Rightarrow$$
$$\mathcal{S}, (\&[\texttt{mut}]\ell^{\bullet}@n')|_{@} \longrightarrow \mathcal{S}, \ell^{\circ} \wedge |\mathcal{D}| = \mathcal{S} \wedge \ell^{\circ}@n'|_{@} = \ell^{\circ}$$

□

Case *Assign Owned*. We assume that before the evaluation, given the distributed program state and monolithic program state as below:

$$\mathcal{D} = \mathcal{D}_0 \otimes (n' \mapsto (\ell \mapsto (v, m))) \qquad \mathcal{S} = \mathcal{S}_0 \otimes (\ell \mapsto (v, m)) \qquad |\mathcal{D}_0| = \mathcal{S}_0$$

We can say that, initially, the flattened distributed program state equals to the monolithic program state:

$$|\mathcal{D}| = \mathcal{S}$$

Following the dFR's reduction rule Assign Owned (s1) and Assign Owned (s2), the distributed execution of a remote assignment to an owned reference gives:

$$\mathcal{D}_0 \otimes (n' \mapsto (\ell \mapsto (v, m))), C +\!\!+ (n, \ell^{\bullet}@n' := v') \rightsquigarrow$$
$$\mathcal{D}_0 \otimes (n' \mapsto (\ell \mapsto (v', m))), C +\!\!+ (n, ()@n')$$

Following FR's reduction rule Assign Owned, the monolithic execution of the flattened remote assignment to an owned reference gives:

$$\mathcal{S}_0 \otimes (\ell \mapsto (v, m)), (\ell^{\bullet}@n' := v')|_{@} \longrightarrow \mathcal{S}_0 \otimes (\ell \mapsto (v', m)), ()$$

After the evaluation, the updated distributed program state and monolithic program state are shown as below:

$$\mathcal{D}' = \mathcal{D}_0 \otimes (n' \mapsto (\ell \mapsto (v', m))) \qquad \mathcal{S}' = \mathcal{S}_0 \otimes (\ell \mapsto (v', m))$$

To conclude, after the evaluation, the flattened updated distributed program state and the updated monolithic program are equal, and the flatted unit value obtained from the distributed execution and the unit value obtained from the monolithic execution are trivially equal:

$$|\mathcal{D}'| = |\mathcal{D}_0| \otimes (\ell \mapsto (v', m)) = \mathcal{S}_0 \otimes (\ell \mapsto (v', m)) = \mathcal{S}' \qquad ()@n'|_{@} = ()$$

Hence:

$$\mathcal{D}, C +\!\!+ (n, \ell^{\bullet}@n' := v') \rightsquigarrow \mathcal{D}', C +\!\!+ (n, ()@n') \wedge |\mathcal{D}| = \mathcal{S}$$
$$\Rightarrow$$
$$\mathcal{S}, (\ell^{\bullet}@n' := v')|_{@} \longrightarrow \mathcal{S}', () \wedge |\mathcal{D}'| = \mathcal{S}' \wedge ()@n'|_{@} = ()$$

□

Case *Assign Borrowed*. We assume that before the evaluation, given the distributed program state and monolithic program state as below:

$$\mathcal{D} = \mathcal{D}_0 \otimes (n' \mapsto (\ell \mapsto (v, m))) \qquad \mathcal{S} = \mathcal{S}_0 \otimes (\ell \mapsto (v, m)) \qquad |\mathcal{D}_0| = \mathcal{S}_0$$

We can say that, initially, the flattened distributed program state equals to the monolithic program state:

$$|\mathcal{D}| = \mathcal{S}$$

Following the dFR's reduction rule Assign Borrowed (s1) and Assign Borrowed (s2), the distributed execution of a remote assignment to a (mutably) borrowed reference gives:

$$\mathcal{D}_0 \otimes (n' \mapsto (\ell \mapsto (v, m))), C +\!\!+ (n, \ell^\circ @ n' := v') \rightsquigarrow$$
$$\mathcal{D}_0 \otimes (n' \mapsto (\ell \mapsto (v', m))), C +\!\!+ (n, ()@n')$$

Following FR's reduction rule Assign Borrowed, the monolithic execution of the flattened remote assignment to a (mutably) borrowed reference gives:

$$\mathcal{S}_0 \otimes (\ell \mapsto (v, m)), (\ell^\circ @ n' := v')|_@ \longrightarrow \mathcal{S}_0 \otimes (\ell \mapsto (v', m)), ()$$

After the evaluation, the updated distributed program state and monolithic program state are shown as below:

$$\mathcal{D}' = \mathcal{D}_0 \otimes (n' \mapsto (\ell \mapsto (v', m))) \qquad \mathcal{S}' = \mathcal{S}_0 \otimes (\ell \mapsto (v', m))$$

To conclude, after the evaluation, the flattened updated distributed program state and the updated monolithic program are equal, and the flatted unit value obtained from the distributed execution and the unit value obtained from the monolithic execution are trivially equal:

$$|\mathcal{D}'| = |\mathcal{D}_0| \otimes (\ell \mapsto (v', m)) = \mathcal{S}_0 \otimes (\ell \mapsto (v', m)) = \mathcal{S}' \qquad ()@n'|_@ = ()$$

Hence:

$$\mathcal{D}, C +\!\!+ (n, \ell^\circ @ n' := v') \rightsquigarrow \mathcal{D}', C +\!\!+ (n, ()@n') \wedge |\mathcal{D}| = \mathcal{S}$$
$$\Rightarrow$$
$$\mathcal{S}, (\ell^\circ @ n' := v')|_@ \longrightarrow \mathcal{S}', () \wedge |\mathcal{D}'| = \mathcal{S}' \wedge ()@n'|_@ = ()$$

□

Case *Decl*. We assume that before the evaluation, given the distributed program state $\mathcal{D}$ and monolithic program state $\mathcal{S}$ where the flatten distributed program state equals to the monolithic program state. In addition, we take a fresh location ℓ:

$$|\mathcal{D}| = \mathcal{S} \qquad \ell \notin \mathbf{dom}\ \mathcal{D}(n')$$

Following the dFR's reduction rule Decl (s1) and Decl (s2), the distributed execution of a remote declaration gives:

$$\mathcal{D}, C +\!\!+ (n, \texttt{let mut@}n'\ x = v; t) \rightsquigarrow$$
$$\mathcal{D} \mid (n' \mapsto \mathcal{D}(n') \otimes (\ell \mapsto (v, k))), C +\!\!+ (n, t[\ell^\bullet @ n'/x])$$

By lemma A.1, we have:

$$\ell \notin \mathbf{dom}\ \mathcal{S}$$

Following FR's reduction rule Decl, the monolithic execution of the flattened remote declaration gives:

$$\mathcal{S}, (\texttt{let mut@}n'\ x = v; t)|_@ \longrightarrow \mathcal{S} \otimes (\ell \mapsto (v, k)), t[\ell^\bullet/x]$$

After the evaluation, the updated distributed program state and monolithic program state are shown as below:

$$\mathcal{D}' = \mathcal{D} \mid (n' \mapsto \mathcal{D}(n') \otimes (\ell \mapsto (v, k))) \qquad \mathcal{S}' = \mathcal{S} \otimes (\ell \mapsto (v, k))$$

To conclude, after the evaluation, the flattened updated distributed program state and the updated monolithic program are equal, and the substitution with the flattened owned reference obtained from the distributed execution and the substitution with the owned reference obtained from the monolithic execution are trivially equal:

$$|\mathcal{D}'| = |\mathcal{D}| \otimes (\ell \mapsto (v, k)) = \mathcal{S} \otimes (\ell \mapsto (v, k)) = \mathcal{S}'$$

$$t[\ell^\bullet @ n'/x]|_@ = t[(\ell^\bullet @ n')|_@/x] = t[\ell^\bullet/x]$$

Hence:

$$\mathcal{D}, C +\!\!+ (n, \texttt{let mut@}n'\ x = v; t) \rightsquigarrow \mathcal{D}', C +\!\!+ (n, t[\ell^\bullet @ n'/x]) \wedge |\mathcal{D}| = \mathcal{S}$$

$$\Rightarrow$$

$$\mathcal{S}, (\texttt{let mut@}n'\ x = v; t)|_@ \longrightarrow \mathcal{S}', t[\ell^\bullet/x] \wedge |\mathcal{D}'| = \mathcal{S}' \wedge t[\ell^\bullet @ n'/x]|_@ = t[\ell^\bullet/x]$$

□

The proofs for location transparency of the distributed and monolithic executions of local terms and remote terms should then be trivial.

B THE TYPE SYSTEM OF FR

Figure 10 presents the syntax of types of Pearce's [2021] FR without any modification. A primitive type such as the integer type `int` has copy semantics. A box type $\square T$ that represents a heap allocation has move semantics. A partial type may contain *undefined* components. An undefined component denoted by $\lfloor T \rfloor$ represents a currently inaccessible location as its value has already been moved. The dFR is merely a semantics extension of FR using the same type system as FR.

B.1 Preliminaries

These are support functions presented by Pearce [2021] for defining the typing rules.

Definition B.1 (Copy Types). *A type T has copy semantics, denoted by* `copy`*(T), when $T =$* `int` *or $T = \&\overline{w}$.*

Note that mutable references and boxes do not have copy semantics.

Partial Types	$\widetilde{T}$	::=	T	type
		|	$\square\widetilde{T}$	partial box
		|	$\lfloor T \rfloor$	undefined
Types	T	::=	ϵ	unit
		|	`int`	integer
		|	`&mut` $\overline{w}$	mutable borrow
		|	$\&\overline{w}$	immutable borrow
		|	$\square T$	box

Fig. 10. Syntax of Types

Definition B.2 (Type Strengthening). *For partial types $\widetilde{T_1}$ and $\widetilde{T_2}$, we say that $\widetilde{T_1}$ strengthes $\widetilde{T_2}$, denoted by $\widetilde{T_1} \sqsubseteq \widetilde{T_2}$, according to the following rules:*

$$\frac{}{\widetilde{T_1} \sqsubseteq \widetilde{T_1}}\ \text{(W-Reflex)} \qquad \frac{\widetilde{T_1} \sqsubseteq \widetilde{T_2}}{\square\widetilde{T_1} \sqsubseteq \square\widetilde{T_2}}\ \text{(W-Box)} \qquad \frac{u \sqsubseteq w}{\Gamma \vdash \&[\mathtt{mut}]\, u \sqsubseteq \&[\mathtt{mut}]\, w}\ \text{(W-Bor)}$$

$$\frac{T_1 \sqsubseteq T_2}{\lfloor T_1 \rfloor \sqsubseteq \lfloor T_2 \rfloor}\ \text{(W-UndefA)} \qquad \frac{T_1 \sqsubseteq T_2}{T_1 \sqsubseteq \lfloor T_2 \rfloor}\ \text{(W-UndefB)} \qquad \frac{\widetilde{T_1} \sqsubseteq \lfloor T_2 \rfloor}{\square\widetilde{T_1} \sqsubseteq \square\lfloor T_2 \rfloor}\ \text{(W-UndefC)}$$

Note that the rule W-Bor requires the mutability to be the same on both sides.

Definition B.3 (Type Join). *The join of partial types $\widetilde{T_1}$ and $\widetilde{T_2}$, denoted $\widetilde{T_1} \sqcup \widetilde{T_2}$, is a partial function returning the strongest $\widetilde{T_3}$ such that $\widetilde{T_1} \sqsubseteq \widetilde{T_3}$ and $\widetilde{T_2} \sqsubseteq \widetilde{T_3}$.*

Definition B.4 (Environment Strengthening). *Let Γ_1 and Γ_2 be typing environments. We say that Γ_1 strengthens Γ_2, denoted $\Gamma_1 \sqsubseteq \Gamma_2$, if and only if $\mathrm{dom}(\Gamma_1) = \mathrm{dom}(\Gamma_2)$ and for all $x \in \mathrm{dom}(\Gamma_1)$ where $\Gamma_1(x) = \langle \widetilde{T_1} \rangle^l$, we have $\Gamma_2(x) = \langle \widetilde{T_2} \rangle^l$ where $\widetilde{T_1} \sqsubseteq \widetilde{T_2}$.*

Definition B.5 (Environment Join). *The join of environments Γ_1 and Γ_2, denoted $\Gamma_1 \sqcup \Gamma_2$, is a partial function returning the strongest Γ_3 such that $\Gamma_1 \sqsubseteq \Gamma_3$ and $\Gamma_2 \sqsubseteq \Gamma_3$.*

Definition B.6 (LVal Typing). *An lval w is said to be typed with respect to an environment Γ, denoted $\Gamma \vdash w : \langle \widetilde{T} \rangle$, according to the following rules:*

$$\frac{\Gamma(x) = \langle \widetilde{T} \rangle^m}{\Gamma \vdash x : \langle \widetilde{T} \rangle^m}\ \text{(T-LvVar)} \qquad \frac{\Gamma \vdash w : \langle \square\widetilde{T} \rangle^m}{\Gamma \vdash *w : \langle \widetilde{T} \rangle^m}\ \text{(T-LvBox)}$$

$$\frac{\Gamma \vdash w : \langle \&[\mathtt{mut}]\overline{u} \rangle^n \qquad \overline{\Gamma \vdash u : \langle T \rangle^m}}{\Gamma \vdash *w : \langle \bigsqcup_i T_i \rangle^{\sqcap_i m_i}}\ \text{(T-LvBor)}$$

Definition B.7 (Path). *A path π is a sequence of zero or more path selectors ρ, which is either empty ($\pi \triangleq \epsilon$) or composed by appending a selector onto another path ($\pi \triangleq \pi' \cdot \rho$).*

Definition B.8 (Path Selector). *A path selector ρ is always a dereference ($\rho \triangleq *$).*

Definition B.9 (Path Conflict). *Let $u \triangleq \pi_u \mid x$ and $w \triangleq \pi_w \mid y$ be lvals. Then w is said to conflict with u, denoted $u \bowtie w$, if $x = y$.*

Note that $u \triangleq \pi \mid x$ denotes *destructuring* of an lval u into its base x and path π.

Definition B.10 (Type Containment). *Let Γ be an environment where $\Gamma(x) = \langle \widetilde{T} \rangle^l$ for some l. Then $\Gamma \vdash x \rightsquigarrow T_y$ denotes that variable x contains type T_y and is defined as $\mathtt{contains}(\Gamma, \widetilde{T}, T_y)$ where:*

$$\mathtt{contains}(\Gamma, \widetilde{T}, T_y) = \begin{cases} \mathtt{contains}(\Gamma, \square\widetilde{T'}, T_y) & \textbf{if } \widetilde{T} = \square\widetilde{T'}, \\ \mathit{true} & \textbf{if } \widetilde{T} = T_y, \\ \mathit{false} & \textbf{otherwise} \end{cases}$$

Definition B.11 (Read Prohibited). *In an environment Γ, an lval w is said to be read prohibited, denoted $\mathtt{readProhibited}(\Gamma, w)$, when some x exists where $\Gamma \vdash x \rightsquigarrow \&\mathtt{mut}\ \overline{u}$ and $\exists_i (u_i \bowtie w)$.*

Definition B.12 (Write Prohibited). *In an environment Γ, an lval w is said to be write prohibited, denoted $\mathtt{writeProhibited}(\Gamma, w)$, when either some x exists where $\Gamma \vdash x \rightsquigarrow \&\overline{u} \wedge \exists_i (u_i \bowtie w)$ or $\mathtt{readProhibited}(\Gamma, w)$ holds.*

The partial function $\texttt{move}(\Gamma, w)$ determines the environment after the value of an lval w is moved out:

Definition B.13 (Move). *Let Γ be an environment where $\Gamma(x) = \langle \widetilde{T_1} \rangle^l$ for some lifetime l, and w an lval where $w \triangleq \pi_x \mid x$. Then $\texttt{move}(\Gamma, w)$ is a partial function defined as $\Gamma[x \mapsto \langle \widetilde{T_2} \rangle^l]$ where $\widetilde{T_2} = \texttt{strike}(\pi_x \mid \widetilde{T_1})$:*

$$\texttt{strike}(\epsilon \mid T) = [T]$$
$$\texttt{strike}((\pi \cdot *) \mid \square \widetilde{T_1}) = \square \widetilde{T_2} \quad \textbf{where } \widetilde{T_2} = \texttt{strike}(\pi \mid \widetilde{T_1})$$

Definition B.14 (Mutable). *Let Γ be an environment where $\Gamma(x) = \langle \widetilde{T} \rangle^l$ for some lifetime 1, and w an lval where $w \triangleq \pi_x \mid x$. Then $\texttt{mut}(\Gamma, w)$ is a partial function defined as $\texttt{mutable}(\Gamma, \pi_x \mid \widetilde{T})$ that determines whether w is mutable:*

$$\texttt{mutable}(\Gamma, \epsilon \mid T) = \mathit{true}$$
$$\texttt{mutable}(\Gamma, (\pi \cdot *) \mid \square T) = \texttt{mutable}(\Gamma, \pi \mid T)$$
$$\texttt{mutable}(\Gamma, (\pi \cdot *) \mid \&\texttt{mut}\ \overline{w}) = \bigwedge\nolimits_i \texttt{mut}(\Gamma, \pi \cdot w_i)$$

Definition B.15 (Environment Drop). *The environment drop deallocates locations by removing them from an environment Γ: $\texttt{drop}(\Gamma, m) = \Gamma - \{x \mapsto \langle \widetilde{T} \rangle^m \mid x \mapsto \langle \widetilde{T} \rangle^m \in \Gamma\}$.*

Definition B.16 (Well-formed Type). *For an environment Γ, a type T is said to be well-formed with respect to a lifetime l, denoted $\Gamma \vdash T \geq l$, according to rules:*

$$\frac{}{\Gamma \vdash \texttt{int} \geq l}\ \text{(L-Int)} \qquad \frac{\Gamma \vdash u : \langle T \rangle^m \qquad m \geq l}{\Gamma \vdash \&[\texttt{mut}]\, u \geq l}\ \text{(L-Borrow)} \qquad \frac{\Gamma \vdash T \geq l}{\Gamma \vdash \square T \geq l}\ \text{(L-Box)}$$

Definition B.17 (Compatible Shape). *For an environment Γ, two partial types $\widetilde{T_1}$ and $\widetilde{T_2}$ are shape compatible, denoted as $\Gamma \vdash \widetilde{T_1} \approx \widetilde{T_2}$, according to the following rules:*

$$\frac{}{\Gamma \vdash \texttt{int} \approx \texttt{int}}\ \text{(S-Int)} \qquad \frac{\Gamma \vdash \widetilde{T_1} \approx \widetilde{T_2}}{\Gamma \vdash \square \widetilde{T_1} \approx \square \widetilde{T_2}}\ \text{(S-Box)} \qquad \frac{\forall_{i,j} (\Gamma \vdash u_i : \widetilde{T_1} \approx \widetilde{T_2} : w_j \dashv \Gamma)}{\Gamma \vdash \&[\texttt{mut}]\, \overline{u} \approx \&[\texttt{mut}]\, \overline{w}}\ \text{(S-Bor)}$$

$$\frac{\Gamma \vdash T_1 \approx \widetilde{T_2}}{\Gamma \vdash \lfloor T_1 \rfloor \approx \widetilde{T_2}}\ \text{(S-UndefL)} \qquad \frac{\Gamma \vdash \widetilde{T_1} \approx T_2}{\Gamma \vdash \widetilde{T_1} \approx \lfloor T_2 \rfloor}\ \text{(S-UndefR)}$$

Definition B.18 (Write). *Let Γ be an environment where $\Gamma(x) = \langle \widetilde{T_1} \rangle^l$ for some lifetime l and lval w where $w \triangleq \pi_x \mid x$. Then, write$^k(\Gamma, w, T)$ is a partial function defined as $\Gamma_2[x \mapsto \langle \widetilde{T_2} \rangle^l]$ for some rank $k \geq 0$ where $(\Gamma_2, \widetilde{T_2}) = \texttt{update}^k(\Gamma, \pi_x \mid \widetilde{T_1}, T)$:*

$$\texttt{update}^0(\Gamma, \epsilon \mid \widetilde{T_1}, T_2) = (\Gamma, T_2)$$
$$\texttt{update}^{k \geq 1}(\Gamma, \epsilon \mid T_1, T_2) = (\Gamma, T_1 \sqcup T_2)$$
$$\texttt{update}^k(\Gamma_1, (\pi \cdot *) \mid \square \widetilde{T_1}, T) = (\Gamma_2, \square \widetilde{T_2}) \quad \textbf{where } (\Gamma_2, \widetilde{T_2}) = \texttt{update}^k(\Gamma_1, \pi \mid \widetilde{T_1}, T)$$
$$\texttt{update}^k(\Gamma, (\pi \cdot *) \mid \&\texttt{mut}\ \overline{u_i}, T) = (\bigsqcup\nolimits_i \Gamma_i, \&\texttt{mut}\ \overline{u_i}) \quad \textbf{where } \overline{\Gamma_i = \texttt{write}^{k+1}(\Gamma, \pi \mid u_i, T)}$$

$$\frac{\sigma \vdash v : T}{\Gamma \vdash \langle v : T \rangle_\sigma^l \dashv \Gamma} \text{ (T-Const)} \qquad \frac{\Gamma_1 \vdash \langle t : T \rangle_\sigma^l \dashv \Gamma_2}{\Gamma_1 \vdash \langle \mathsf{box}\ t : \square T \rangle_\sigma^l \dashv \Gamma_2} \text{ (T-Box)}$$

$$\frac{\Gamma \vdash w : \langle T \rangle^m \qquad \mathsf{copy}(T) \qquad \neg \mathsf{readProhibited}(\Gamma, w)}{\Gamma \vdash \langle !w : T \rangle_\sigma^l \dashv \Gamma} \text{ (T-Copy)}$$

$$\frac{\Gamma \vdash w : \langle T \rangle^m \qquad \neg \mathsf{writeProhibited}(\Gamma_1, w) \qquad \Gamma_2 = \mathsf{move}(\Gamma_1, w)}{\Gamma_1 \vdash \langle \#w : T \rangle_\sigma^l \dashv \Gamma_2} \text{ (T-Move)}$$

$$\frac{\Gamma \vdash w : \langle T \rangle^m \qquad \mathsf{mut}(\Gamma, w) \qquad \neg \mathsf{writeProhibited}(\Gamma, w)}{\Gamma \vdash \langle \&\mathsf{mut}\ w : \&\mathsf{mut}\ w \rangle_\sigma^l \dashv \Gamma} \text{ (T-MutBorrow)}$$

$$\frac{\Gamma \vdash w : \langle T \rangle^m \qquad \neg \mathsf{readProhibited}(\Gamma, w)}{\Gamma \vdash \langle \&w : \&w \rangle_\sigma^l \dashv \Gamma} \text{ (T-ImmBorrow)}$$

$$\frac{\Gamma_1 \vdash \langle t_1 : T_1 \rangle_\sigma^l \dashv \Gamma_2 \qquad \ldots \qquad \Gamma_n \vdash \langle t_n : T_n \rangle_\sigma^l \dashv \Gamma_{n+1}}{\Gamma_1 \vdash \langle \overline{t} : \overline{T_n} \rangle_\sigma^l \dashv \Gamma_{n+1}} \text{ (T-Seq)}$$

$$\frac{\Gamma_1 \vdash \langle t : T \rangle_\sigma^m \dashv \Gamma_2 \qquad \Gamma_2 \vdash T \geq l \qquad \Gamma_3 = \mathsf{drop}(\Gamma_2, m)}{\Gamma_1 \vdash \langle \{t\} : T \rangle_\sigma^l \dashv \Gamma_3} \text{ (T-Block)}$$

$$\frac{x \notin \mathbf{dom}(\Gamma_1) \qquad \Gamma_1 \vdash \langle t_1 : T \rangle_\sigma^l \dashv \Gamma_2 \qquad \Gamma_3 = \Gamma_2[x \mapsto \langle T \rangle^l] \qquad \Gamma_3 \vdash t_2 : T' \dashv \Gamma_3}{\Gamma_1 \vdash \langle \mathsf{let\ mut}\ x = t_1; t_2 : T' \rangle_\sigma^l \dashv \Gamma_3} \text{ (T-Declare)}$$

$$\frac{\begin{array}{c}\Gamma_1 \vdash w : \langle \widetilde{T_1} \rangle^m \qquad \Gamma_1 \vdash \langle t : T_2 \rangle_\sigma^l \dashv \Gamma_2 \qquad \Gamma_2 \vdash \widetilde{T_1} \approx T_2 \\ \Gamma_2 \vdash T_2 \geq m \qquad \Gamma_3 = \mathsf{write}^0(\Gamma_2, w, T_2) \qquad \neg \mathsf{writeProhibited}(\Gamma_3, w)\end{array}}{\Gamma_1 \vdash \langle w = t : \epsilon \rangle_\sigma^l \dashv \Gamma_3} \text{ (T-Assign)}$$

Fig. 11. Typing Rules for FR

B.2 Typing Rules

Utilising the helper functions defined in perious section, FR's typing rules are shown in figure 11. Note that we do not make any major modification to these typing rules. The only minor changes are: 1) The syntax of the lifetime of a block is now implicit; and 2) Since the syntax of declaration is changed from the original form `let mut` $x = t$ to `let mut` $x = t_1; t_2$ to allow substitutions, the typing rule is slightly modified accordingly.

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009