Vrije Universiteit Amsterdam

Master's thesis

Submitted in partial fulfillment of the requirements for the degree of Master of Science in Parallel and Distributed Computer Systems.

NAME TBD

Christopher Esterhuyse
(ID: 2553295)

supervisors

Vrije Universiteit Amsterdam	Centrum Wiskunde & Informatica
dr. J. Endrullis	prof. dr. F. Arbab

August 3, 2019

Abstract

TODO

Acknowledgments

TODO

Contents

List of Figures

Chapter 1

Introduction

(TODO) abstractions are everywhere in programming but coordination is still primitive. there is a large gap between protocol concept and implementation. makes code confusing, hard to maintain, brittle, buggy, and non-standardized. we wish to solve this problem by expressing the protocol at a higher level of abstraction and then compile down to something efficient. Reo is a language well-suited for this task. it is very expressive, you can describe relations between data visible at ports. can express state etc. the semantics for reo are value-passing; this is natural for composability and modularity. In the context of shared memory, the java backend is performant owing to its reliance on semantic value-passing, implemented by passing object references 'under the hood'. Currently, this implementation is not entirely correct; mutable datatypes are subject to data races for connectors which replicate data. there are incentives for adding support for systems languages such as C: they represent a large swathe of the possible user space, and their low-levelness means that they can more effectively leverage the information that protocol descriptions provide in the first place. Rust is a programming language similar to C++, and intended for a similar audience. aside from the comforts of modern programming languages (closures, generics, functional patterns, extensive macros) it is notable for its unique memory management system: it relies on affine types to statically manage variable bindings, implicitly freeing memory which goes out of scope in a predictible manner. its ownership rules also prevent the majority of data races and protect the programmer from undefined behavior such as accessing uninitialiedd memroy. its UNSAFE sub-language is very similar to C, and can be tapped-into explicitly to achieve optimizations that the compiler cannot prove are safe. Rust is also useful for its exceptionally expressive APIs, as the types themselves allow and require the OWNERSHIP of values to be specified, giving the callee more freedom to specialize their bejhavior according to these guarantees. in this work we detail the development of a Rust back-end for the Reo compiler to generate protocol objects which can act as the communication mediums of 'compute components'. furthermore, we explore how we are able to leverage the affine type system of the Rust language to provide liveness properties to our coordinating programs.

Chapter 2 introduces background information pertinent to Reo and our compile-target languages. Chapter 3 explains how our new Reo back-end compiles to the Rust language via a new intermediate representation. Chapter 4.3 describes our Reo-rs library which gives the Reo-generated code its runtime behavior as a coordination medium. Chapter 5 describes our orthogonal tool for statically checking protocol adherence to protocols at runtime. Chapter 6 evaluates the runtime performance of our protocol objects, offering comparisons to Reo's other back-ends, to hand-crafted coordination code, and exploring how the protocol's description influences runtime. Finally, Section 7 reflects on our contributions as a whole and highlights opportunities for future work.

Chapter 2

Background

2.1 Reo

Reo is a high-level language for specifying protocols. Here, we explore the motivation behind Reo's development, and how the language is used. The Reo language has applicability whenever there is a benefit in being able to formalize a communication protocol. However, this work primarily focuses on Reo's role in automatic generation of glue-code for applications.

2.1.1 Motivation

TODO

2.1.2 Language

TODO TODO textual and visual examples eg: alternator

2.1.3 Semantic Models

Reo took a number of years to take its present shape. It is recognizable as early as 2001, but was presented as a concept before it was formalized, leaving it as a task for future work [JA12]. Later, This several different approaches to formal semantics were developed. For our purposes, it suffices to concentrate only on the small subset of the semantics to follow. For additional information, the work of Jongmans in particular serves as a good entry point[JA12].

Starting with the fundamentals, a **stream** specifies the value of a variable from data domain D changing over the course of a sequence of events. Usually streams are considered infinite, and so it is practical to define them as a function $\mathbb{N} \mapsto D$. A **timed data stream** (TDS) takes this notion a step further,

$\mathbb{R}$	A	B
0.0	0	*
0.1	*	0
0.2	*	*
0.3	1	*
0.4	*	1

Table 2.1: Trace table comprised of TDS's for variables A and B. This trace represents behavior that adheres to the *fifo1* protocol with input and output ports A and B respectively.

annotating each event in the sequence with an increasing *time stamp*. A TDS is defined by some tuple $(\mathbb{N} \mapsto \mathbb{R}, \mathbb{N} \mapsto D)$, or equivalently, $\mathbb{N} \mapsto (\mathbb{R}, D)$ with the added constraints that time must increase toward infinity [ABRS04]. By associating one TDS with each *named variable* of a program, one can represent a *trace* of its execution. TDS events with the same time stamp are considered simultaneous, allowing reasoning about *snapshots* of the program's state over its run time. These traces can be practically visualized as **trace tables**, with variables for columns and time stamps for rows by representing the absence of data observations using a special 'silent' symbol *, referring *silent behavior*. In this work, we use 'trace tables' to refer to both the visualization and to a program trace as a set of named TDS's. The runs of finite programs can be simulated either by bounding the tables (constraining the TDS domain to be finite), or by simulating finite behavior as infinite by extending the 'end' forevermore with silent behavior. Table 2.1 gives an example of a trace table for some program with two named variables.

One of it's earlier *coalegebraic models* represented Reo connectors as **stream constraints** (SC) over such TDS tables in which variables are ports [Arb04]. Here, constraints are usually defined in first-order *temporal logic*, which allows the discrimination of streams according to their values both now and arbitrarily far into the future[1]. This model is well-suited for translating from the kinds of safety properties that are typically desired in practice. Statements such as 'A never receives a message before B has communicated with C' have clear mappings to temporal logic, as often it is intuitive to reason about safety by reasoning about future events. Table 2.1 above shows the trace of a program that adheres the *fifo1* protocol with ports A and B as input and output respectively.

SC are unwieldy in the context of code generation. In reality, it is easier

[1]Not all variants of temporal logic are equally (succinctly) expressive. It requires a notion of 'bounded lookahead' to express a notion such as 'P holds for the next 3 states' as something like $\Box^{1-3}P$ rather than the verbose $(\Box P \wedge \Box\Box P \wedge \Box\Box\Box P)$.

to predicate one's next actions as a function of the *past* rather than the future. Accordingly, **constraint automata** (CA) was one of the *operational models* for modeling Reo connectors that has a clearer correspondence to stateful computation. Where an NFA accepts finite strings, a CA accepts trace tables. Thus, each CA represents some protocol. Programs are adherent to the protocol if and only if it always generates only accepted trace tables. From an implementation perspective, CA can be thought to enumerate precisely the actions which are allowed at ports given the correct states, and prohibiting everything else by default. A CA is defined with a state set and initial state as usual, but each transition is given *constraints* that prevent their firing unless satisfied; each transition has both (a) the *synchronization constraint*, the set of ports which perform actions, and (b) a *data constraint* predicate over the values of ports in the firing set at the 'current' time step. For example, Listing 2.1 above is accepted by the CA of the *fifo1* connector with all ports of binary data type $\{0, 1\}$. Observe that here the automaton discriminates the previously-buffered value ('remembering' what A stored) by distinguishing the options with states q_{f0} and q_{f1}. As a consequence, it is not possible to represent a *fifo1* protocol for an infinite data domain without requiring infinite states.

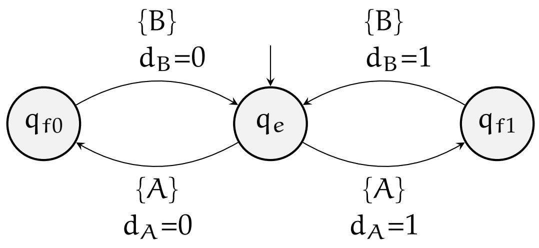


Figure 2.1: CA for the *fifo1* protocol with ports A and B sharing data domain $\{0, 1\}$.

Later, CA were extended to include *memory cells* (or *memory variables*) which act as value stores whose contents *persist* into the future. Data constraints are provided the ability to assign to their *next* value, typically using syntax from temporal logic (eg: m' is the value of m at the next time stamp). Figure 2.2 revisits the *fifo1* protocol from before. With this extension, the task of persistently storing A's value into the buffer can be relegated to m, simplifying the state space significantly. This change also makes it possible to represent connectors for arbitrary data domains, finite or otherwise.

For the purposes of Reo, we are interested in being able to compute the composition of CAs to acquire a model for the compositions of their protocols. Figure 2.3 shows an example of such a composition, producing *fifo2* by composing *fifo1* with itself. This new protocol indeed exhibits the desired be-

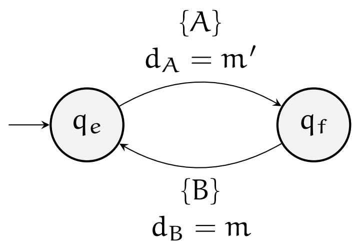


Figure 2.2: CA with memory cell m for Reo connector `fifo1` with arbitrary data domain D common to ports A and B. Two states are used to track to enforce alternation between filling and emptying m.

havior; the memory cells are able to store up to two elements at a time, and B is guaranteed to consume values in the order that A produced them. Even at this small scale, we see how the composition of such CA have a tendency to result in an *explosion* if state- and transition-space. When seen at larger scales, a *fifo*N buffer consists of 2^N states. The problem is the inability for a CA to perform any meaningful *abstraction*; here, it manifests as the automaton having to express its transition system in undesired specificity. Intuitively, the contents of m_0 are irrelevant when m_1 is drained by B, but the CA requires two transitions to cover the possible cases in which this action is available. In the context of accepting existing trace tables, data constraints are evaluated predictably. However, in the case of code generation we are able to treat the data constraint instead as a pair of (a) the *guard* which enables the transition as a function of the *present* time stamp, and (b) the *assignment*, which may reason about the next time step, and which we are able to guarantee by *assigning* variables. As such, data constraints are broken up into these parts where possible. Figure 2.3 and others to follow formulate their data constraints such that the guard and assignment parts are identifiable wherever it is practical to do so.

Evidently, memory cells provide a new means of enforcing how data persists over time. In many cases, it can be seen that the same connectors can be represented differently by moving this responsibility between state- and data-domains. **Rule-based automata** (RBA) are the cases of CA for which this idea is taken to an extreme by relying only on memory cells entirely; RBAs have only one state. Figure 2.4 models the *fifo1* connector once again, this time as an RBA. Aside from the added expressivity, RBAs benefit from being cheaper to compose. As the state space is degenerate, RBAs may be easily re-interpreted into forms more easy to work with. **Rule-based form** (RBF) embraces the statelessness of an RBA as a single formula, the *disjunction* of its constraints. In this view, Dokter et al. defines their composition of connectors such that, instead of exploding, the composed connector has transitions and memory cells that are

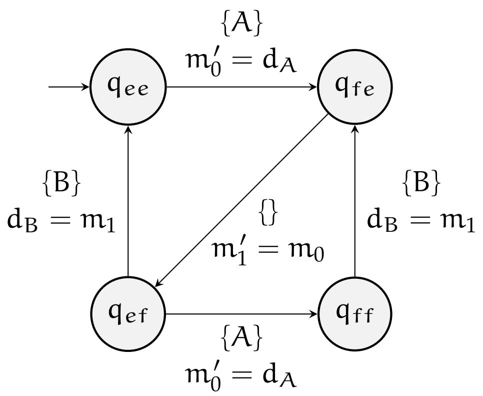


Figure 2.3: CA with memory cells m_0 and m_1 for the *fifo2* connector with an arbitrary data domain for ports A and B. Transitions are spread over the state space such that the automaton's structure results in the *first-in-first-out* behavior of the memory cells in series.

the *sum* of its constituent connectors [DA18].

RBAs have a structure more conducive to *simplification* of the transition space, such that one RBA transition may represent several transitions in a CA. Figure 2.5 shows how how this occurs for the *fifo2* connector. Where the CA in Figure 2.3 must distinguish the cases where A fills m_0 as two separate transitions, the RBA is able to use just one; likewise for the transitions representing cases where B is able to drain m_1. This 'coalescing' of transitions in RBAs is possible owing to the collapsing of their state space. Even without an intuitive understanding of why such transitions can be collapsed, such cases may often be identified only by inspecting the syntax of the data constraints. For another example of CA, a naïve translation to RBA might produce two transitions with data constraints $m = * \wedge X$ and $m \neq * \wedge X$ for some X, which are both covered by a single data constraint X. As both RBA and RBF share this property, we usually refer to RBA transitions and RBF disjuncts as *rules*, giving these models their name. By distinguishing CA transitions from RBA rules in terminology, we are perhaps more cognizant of the latter's increased ability to *abstract* away needless data constraints.

Typically, Reo has used the $\mathtt{Data}$ domains in both CA and RBA as parallels to the data-types of the ports. In most of the languages in which Reo protocols are implemented, the discriminants of such types are not distinguished statically. For example, the C language lacks a way to statically enforce a that function `void foo(int x)` is only invoked when x is prime. Instead, checks at runtime are used to specialize behavior. On the other hand, the state-space is simple enough to afford a practical translation into the structure of the pro-

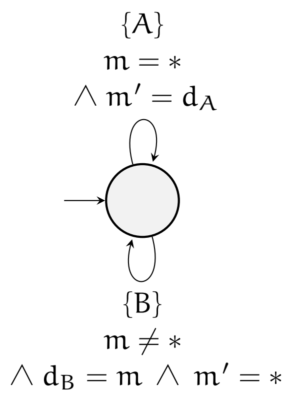


Figure 2.4: RBA of the *fifo1* connector for an arbitrary data domain common to ports A and B. Memory cell m is used both to buffer A's value, and as part of the data constraint on both transitions for *emptying* and *filling* the cell to ensure these interactions are always interleaved. Data constraints are formulated for readability such that the 'guard' and 'assignment' conjuncts are line-separated.

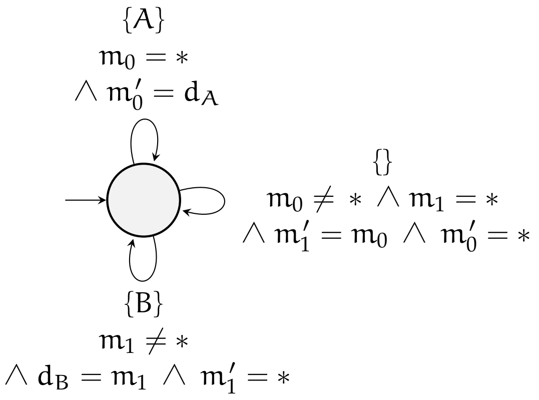
gram itself, requiring no checking at runtime. For example, Listing 1 shows an intuitive representation of a connector that alternates between states A and B, getting data x from its environment in A, and emitting x when $x = 3$. Observe that there is no need to protect operations behind a runtime-check of *which* state the corresponding CA is in. This observation has implications for the behavior of implementations of RBAs, as they 'cannot remember' which state they are in and must thus perform more checking. In practice, the overhead of this checking is manageable, and does not *explode* under composition as the state space of CAs tend to do. The representation of automata in programming languages is explored in more detail in Section 2.2.3.

2.1.4 The Reo Compiler

TODO

2.2 Target Languages

In this section we introduce terminology relevant to the languages targeted for code generation by the Reo compiler. We identify patterns and properties that are relied upon in later chapters.



{A}
$m_0 = *$
$\wedge\ m_0' = d_A$
{}
$m_0 \neq * \ \wedge m_1 = *$
$\wedge\ m_1' = m_0 \ \wedge\ m_0' = *$
{B}
$m_1 \neq *$
$\wedge\ d_B = m_1 \ \wedge\ m_1' = *$

Figure 2.5: RBA of the *fifo2* connector for an arbitrary data domain common to ports A and B. Memory cells m_0 and m_1 are drained by B in the order they are filled by A, and have a capacity of 2 elements. Data constraints are formulated for readability such that the 'guard' and 'assignment' conjuncts are line-separated.

2.2.1 Affine Type System

In a nutshell, affine types are characterized by modeling values as finite resources, operations on which *consume* them. This notion of 'affinity' has its roots in logic. In a type-theoretic proof system, one can attempt to derive some *judgment* $\Gamma \vdash t : \tau$ with *statements* assigning *types* to *terms*; here term t is stated to have type τ under context Γ. A context is simply a list of statements, which can be thought to correspond with *assumptions* or *premises* [NG14]. The judgment holds if one can construct a proof, starting with the judgment and select and applying *rules* until no 'dangling' judgments remain; the set of available rules characterizes the system. For example, simply-typed lambda calculus has a type-derivation rule for abstraction, substitution and application. Depending on the type system, *structural rules* may additionally be provided for manipulating ('massaging') the context such that other rules may be applied. For example, *weakening* is a common structural rule that allows one to arbitrarily discard statements from the context. Depending on the proof system, this may be useful or even necessary before other rules can be applied such that the proof can be completed. For example, $weakening$ is required to prove $A, t : \tau \vdash t : \tau$, as var cannot be directly applied; A is in the way. One can imagine that the ability to arbitrary replicate, discard and re-arrange terms effectively describes a type system that treats its context like a set.

$$(var) : \frac{}{t : \tau \vdash t : \tau} \qquad (weaken) : \frac{\Gamma \vdash \Sigma}{\Gamma, A \vdash \Sigma} \qquad (contract) : \frac{\Gamma, A, A \vdash \Sigma}{\Gamma, A \vdash \Sigma}$$

```java
void stateA() {
  this.value = get();
  if (this.value == 3) {
    stateB();
  } else {
    stateA();
  }
}
void stateB() {
  put(this.value);
  stateA();
}
```

Listing 1: An example of a program which implements a two-state automaton in the Java programming language. Observe that the behavior of states A and B are encoded implicitly in the *structure* of the program, while determining which of the two in A are available A requires a check ar runtime.

```
fifo1(a?, b!) {
  #RBA
  {a, ~b} $m = null, $m' = a
  {~a, b} $m != null, b = $m, $m' = null
}
```

Listing 2: Textual Reo specification of the *fifo1* connector using RBA semantics. Data is forwarded from input A to output B, buffered in-between in memory cell m.

Affine type systems are characterized by the absence of the *contraction* rule. Proofs cannot replicate statements at will, and thus they are a finite resource in the proof, *consumed* by use in rules [Wal05].

As type systems do in general, type affinity excludes some programs from being expressed. In the context of programming languages, why would we want this? Of course, the hardware has no problem replicating the bytes representing some integer. Why then do we limit ourselves? This argument can be made for type systems in general; the machine likewise has no problem re-interpreting the bytes storing a string as an integer. This limitation is a feature in and of itself as long as the programs lost are usually somehow 'undesirable'. For example, it is exceedingly common practice to dedicate a memory region to one type for the duration of the program. The reasons for this are primarily for the programmer and not the program; it is simply easier to distinguish values by type such that values only move between them when explicitly 'moving' in memory.

2.2.2 The Rust Programming Language

The Rust programming language is most similar to C++, being a general-purpose, imperative systems-level programming language with C-like syntax. What sets Rust apart is its memory model. Rust is not a memory-managed language, and has no runtime whatsoever. Instead, the language relies on its *ownership system* to predictably insert allocations and deallocations at the right moment such that it *runs* much as C++ would without exposing these details to the programmer. To make this possible, the Rust compiler keeps track of the variable binding which *owns* a value at all times. Owned values are affine, and associating them with new variable bindings invalidates their previous binding. In Rust, this is called *moving*, and doubles (at least conceptually) with the re-location of a value in memory. Listing 3 illustrates how this appears to a programmer; In `main`, the variable `x` is moved into the scope of `func`. The subsequent access of `x` on line 8 is invalid, preventing this program from compiling. Once an owned value goes out of scope, it is no longer accessible, and Rust performs any destruction associated with that type. Along with the RAII ('resource acquisition is initialization') pattern popularized by C++, programmers can rest assured that their resources are created and destroyed on demand, without the need for any bookkeeping at runtime.

Borrowing

On their own, movement is incredibly restrictive; there is no apparent way to use any resource without destroying it. To reclaim some vital functionality, Rust has the *borrow system* to facilitate the creation and management of types whose ownership is *dependent* on others. Similar to those in C++, programmers are able to create *references* to values (also called 'borrows' in Rust terms). These references are new types, and thus do not represent a transference of ownership to a new binding. Listing 4 demonstrates the example from before, but now passing the x by reference into `func` such that `x` is not invalidated. The Rust compiler's *borrow checker* relies on variable scoping to keep track of these borrows to ensure they do not out-live their referent, as these would manifest at runtime as *dangling pointers*. This relationship between value and reference is referred to in Rust as the reference's *lifetime*. Rust performs this static analysis at a per-function basis. As such, it is necessary for programmers to fully annotate the input and output types of functions, but they can usually be *elided* within function bodies. This has an important consequence; the compiler does cross the boundaries between functions to interpret their relationship.

For some types there is no practical reason to enforce affinity. This is usually the case for primitives such as integers. For these cases, the language uses the `Copy` trait to opt-out of Rust's affine management of these resources. Copy-types behave in ways familiar to C and C++ programmers. Listing 3 from before would compile just fine if Rust's 4-byte unsigned integer type `u32` was used in place of `Foo`.

C and C++ have no inherent support for preventing data races. The programmer is in full control of their resources. It is all too easy to create data races to C by unintentionally accessing the same resource in parallel precautions. One of the tenets of Rust's design is to use its ownership system to prohibit these data races at compile-time. For this reason, Rust has an orthogonal system for *mutability*. References come in two kinds: mutable and immutable. The distinction is made explicit in syntax with the `mut` keyword. Rust relies on a simple observation of the common ingredient for all data races: mutable aliasing; only *changes* to the aliased (one resource accessible by multiple bindings) resource manifest as data races. Rust's approach is thus simply to prohibit mutable aliasing by preventing these conditions from co-existing. Mutable references must be *unique* (prohibiting aliasing) and immutable references do not allow for any operations that would mutate their referent (prohibiting mutability). This is the same thinking behind the *readers-writer* pattern for the eponymous lock: there is no race condition if readers coexist, but if one writer exists, it must have exclusive access.

```
1  struct Foo;
2  fn func(x: Foo) {
3    // this function takes argument `x` by value.
4  }
5  fn main(){
6    let x = Foo; // instantiate.
7    func(x);     // Ok. `x` is moved into `func`.
8    func(x);     // Error! x is used after move.
9  }
```

Listing 3: Type `Foo` is affine. On line 7, x is moved into function `func`, consuming it. Accessing x is invalid, and so line 8 raises an error.

Traits and Polymorphism

The primary means of polymorphism in Rust is through generic types with *trait bounds*, also called 'type parameters'. Traits are most similar to interfaces in Java, categorizing a group of instantiable types by defining abstract functions for which implementors provide definitions. Unlike Java, Rust traits say noth-

```
struct Foo;
fn func(x: &Foo) {
  // this function borrows some `Foo` structure by reference.
}
fn main(){
  let x = Foo; // instantiate.
  func(&x);     // &x is created in-line. x remains in place.
  func(&x);     // another &x is created. x remains in place.
}
```

Listing 4: `Foo` is an affine resource. New references to `x` are created and sent into codefunc without changing the ownership of `x`. Rust's **borrow checker** ensures that these borrows do not outlive `x`.

ing about fields and data, thus describing only their behavior. In this manner, Rust traits are somewhat like C's header files, but rather to be defined per implementor type. Elsewhere in the program, functions and structures can make use of *generic* types. These types are arbitrary and thus opaque to any behavior save for those common to all Rust types; for example, any type at all can be *moved* or can be *borrowed* in Rust. To perform more specialized operations on these generic types, they can be *bounded* with traits. This acts as a contract between the generic context and its concrete call-site; the caller promises that the generic type is reified only with a type which implements the specified traits, and thus the generic can be used in accordance with the behavior these traits provide. This is demonstrated in Listing 5. Here, `something` is a function which can be invoked with `T` chosen to be any type implementing trait `T`, such as `String`.

Above, we see how a function can be defined such that it operates on a generic type. However, the function cannot be used until the type is chosen concretely for a particular instance. This choice is called *dispatch*, and Rust offers two options: static and dynamic. *Static dispatch* (also called 'early binding') is used when the called function can be informed which concrete type has been chosen statically, as the caller knows it at the call-site. During compilation, the generic function is *monomorphized* for the type chosen in this manner on a case-by-case basis, generating binary specialized for that type as if there were no generic at all. Static dispatch is used in C++ templates, and opted-out with the keyword `virtual`. This is Rust's second option: *dynamic dispatch* (also called virtual functions or late binding) where the generic function exists as only one instance, and all of the specialized operations on the generic type are resolved to concrete functions at runtime by traversing a layer of indirection: functions are *looked up* in a virtual function table (vtable). Java uses such virtualization

```rust
pub fn something<T: PrintsBool>() {
    T::print(true);
}
trait PrintsBool {
    fn print(bool);
}
impl PrintsBool for String {
    fn print(b: bool) {
        println!("bool is: {}", b)
}   }
```

Listing 5: Definition of a function `something` generic over some type `T`, where `T` implements trait `PrintsBool`. Types can implement this trait by providing a definition for all the associated functions, in this case, only `print_bool`.

extensively, which allows a lot of flexibility such as allowing functions to be *overridden* by downstream inheritants. Precisely how a language represents virtual functions and lays out the data in memory varies from language to language. Rust uses the *fat pointer* representation in its *trait objects*. Concretely, some generic object which is known only to implement trait `Trait` is represented as a pair of pointers; the first pointing to the actual object' data, and the second pointing to a dense structure of meta-data and function pointers for the methods of `Trait`, usually embedded into the text section of the binary by the Rust compiler itself. Both methods of dispatch are exemplified in Listing 6, demonstrating how static dispatch must *propagate* generics to the caller for resolution to concrete types at compile time, while one function using dynamic dispatch is able to handle any virtualized types by resolving their methods at runtime.

Rust uses traits for just about everything. Some traits are defined in the standard library, and have a degree of 'first class' status by having special meaning when used in combination with the language's syntax. For example, `Not` is a trait that defines a single function, `not`, which is invoked when the type is negated using the usual exclamation syntax, ie. `!true`. Some traits have no associated functions, instead exist for the purpose of communicating information to the compiler. Seen before, `Copy` is a trait which disables the Rust compiler's checks of a value's affinity. `Copy`-types may be passed by value. On the other hand, `Drop` associates the `drop` function with a type, which the compiler will invoke when it goes out of scope; this is the parallel to the definition of destructors in C++. It is common practice in Rust to rely upon common traits such as these for frequently-occurring cases. For example, it is considered good practice to implement `Debug` on your custom traits such that they can

```rust
trait Emits {
    fn emit(&self) -> usize;
}
impl Emits for String {
    fn emit(&self) -> usize {
        self.len()
    }
}
fn func_static<T: Emits>(x: &T) -> bool {
  x.emit() > 10
}
fn func_dynamic(x: &dyn Emits) -> bool {
    x.emit() > 10
}
fn main() {
  let value = String::from("Hello!");
  func_static::<String>(&value);
  func_dynamic(&value as &dyn Emits);
}
```

Listing 6: Static and dynamic dispatch in Rust exemplified. `func_static` shows the former, propagating the type parameter to the caller. `func_dynamic` shows the latter, relying on a virtual function table to resolve the concrete function at runtime. Function `main` shows how both appear at the call site.

be printed using *debug print* syntax (eg. `print!(":?", foo)`) for programmers depending on your work.

Enums and Error Handling

As in C, Rust usually relies on `struct` for defining its types. Each is defined as the list of its constituent fields. Creation of these structures necessitates building all of their constituents, and all fields exist at once. By contrast, *sum types* have *variants* only one of which may be present at a time. Arguably, the duck-typing of Python and flexible polymorphism of Java do the work of these sum types; a variable can be bound to *anything* and then its 'variant' can be reflected at runtime using some explicit operations (`ininstance` and `instanceof` respectively). C takes the approach fitting the language's philosophy; `union` types represent any one of its constituents but the program is at the mercy of the programmer to interact with it as the correct variant as they see fit. Rust's solution is similar to C's unions, but its focus on safety required the use of *tags*. In Rust, an `enum` type is defined with a list of variants, only one of which may exist per intance at a time. At runtime, the variant can be descriminated by explicitly *pattern-matching*, inspecting some implicit meta-data field of the enum stores to reflect which of the variants is in use. Like C's unions, each variant can be an arbitrary type (another struct for example), these variants can be of heterogeneous size, and thus are represeted by the largest of their variants *plus* the space for the tag.

Unlike Java and Python, Rust has no mechanism for *throwables* which override the default control flow, usually for the purposes of ergonomically handling errors. Instead, Rust represents all recoverable errors in the data domain as enums. The standard library defines `Option` and `Result` enums, which are monadic in that they *wrap* the 'useful' data as one of the variants, but represent the possibility for *other* variants also. They differ in that `Option::None` carries no data, and thus `Option` is generic only over one type, the contents of the `Some` variant. `Result`, on the other hand, has two generic types, one for its 'successful' `Result::Ok` variant, and one for its 'unsuccessful' `Result::Err` variant. Listing 7 gives an example of typical error-handling in Rust; here, `divide_by` relies on `Result` to propagate the error for the caller to handle. In circumstances where the error is unrecoverable, Rust uses a thread *panic*, which unrolls the control flow (printing debugging information if an environment variable is set). This is somewhat similar to Java's `Error`.

```rust
struct DivZeroError;  // contains no data

fn divide_by(numerator: f32, divisor: f32) -> Result<f32, DivZeroError> {
    if divisor == 0. {
        Result::Err(DivZeroError)
    } else {
        Result::Ok(numerator / divisor)
}   }

fn main(input: f32) {
    match divide_by(4.5, input) {
        Ok(x) => print!("Success! computed:{}.", x),
        Err(_) => print!("Something went wrong!"),
}   }
```

Listing 7: Demonstrating the Rust idiom of using a `Result` in return position to propagate exceptions to the caller for handling. Here, `main` must `match` the return value to acquire the result contained within the `Result::Ok` variant.

2.2.3 The Type-State Pattern

The *state* or *state machine* pattern refers to the practice of explicitly checking for or distinguishing transitions between and requirements of states in a stateful object[2]. Usually, these states are distinguished in the data domain of one or more types. Even the lowly `Option` type can be viewed as a small state machine as soon as some condition statement specializes operations performed with it. Although its uses are ubiquitous in application development in general, this pattern is particularly useful for those for which the added ability to manage complexity is necessary: video games, for example [Nys14].

As the name suggests, the *type state* pattern is an instance of the state pattern, characterized by encoding states as types, which usually are distinct from *data* in their significance to a language's compiler or interpreter. A common approach is to instantiate one of the state types at a time. As an example, consider the scenario where a program wants to facilitate alternation between invoking some functions `one` and `two` which repeatedly mutate some integer n. Listing 8 gives an example of what this might look like as a *deterministic finite automaton* in the C language. In this rendering, the expression `two(one(START)).n` evaluates to the expected result of $(0+1) \cdot 2 = 2$. Even for this simple example, the encoding of states as types in particular has its benefits; the expression `one(two(START))` may appear sensible at first glance, but the compiler is quick to identify the type mismatch on the argument to `one`, making clear that the expression does not correspond to a path through the automaton:

```
note: expected 'DoTwo' but argument is of type 'DoOne'
```

The type state pattern can be applied in any typed language, but it is particularly meaningful in languages where the compiler or interpreter *enforces* its intended use. The example above demonstrates some utility, but a language such as C has no fundamental way to prevent the programmer from *re-using* values. If the programmer misbehaves, they can retain their previous states when given new ones, and then invoke the transition operations as they please. It's not much of a state machine if all states coexist, is it? This is not always a problem in examples such as the previous. Here, the types prevent the construction of mal-formed *expressions*, and perhaps this is enough. However, we cannot so easily protect a resource from any side effects of `one` or `two`; imagine the chaos that would result from these functions writing to a persistent file descriptor.

An affine type system overcomes the shortcoming illustrated above. By treating instances of these types as affine *resources*, the programmer cannot

[2]Usually, we disregard the effects of terminating the program. Equivalently, this pattern only allows one to describe automata in which every 'useful' state reaches some final 'terminated' state.

```c
typedef struct DoOne { int n; } DoOne;
typedef struct DoTwo { int n; } DoTwo;
const DoOne START = { .n = 0 };

DoTwo one(DoOne d1) {
  DoTwo d2 = { .n = d1.n + 1};
  return d2;
}
DoOne two(DoTwo d2) {
  DoOne d1 = { .n = d2.n * 2};
  return d1;
}
```

Listing 8: An example of the type-state pattern in the C language. The alternating invocation of `one` and `two` is translated to type-checking the compiler can guarantee. This example guarantees that well-formed *expressions* can be interpreted as valid paths in some corresponding automaton, as the types must match.

retain old states without violating the affinity of the types. The example looks very similar when translated to Rust, but now a case such as that shown in Listing 9 will result in the compiler preventing the retention of the variable of type `DoOne`.

```rust
fn main(d1: DoOne) {
  let d2  = two(d1);
  let d1  = one(d2);
  let d2  = two(d1);
  let d2_again = two(d1); // Error! `d1` has been moved.
}
```

Listing 9: A demonstration of how the type-state encoding shown in Listing 8 can leverage affine types to ensure that not only expressions, but *a trace through execution* can be interpreted as valid paths through some corresponding automaton. The compiler correctly rejects this example, which corresponds with attempting to take transition `two` twice in a row.

2.2.4 Proof-of-Work Pattern

Section 2.2.3 demonstrates how the type-state pattern can be used as a tool to *constrain* actions the compiler will permit the program to do. Indeed, this is a natural parallel to the affinity of the type system, which guarantees that no resource is consumed repeatedly. The counterpart to affine types is *relevant* types, which defines correctness as each resource being consumed *at least once*. Type systems that are both relevant and affine are *linear*, such that all objects are consumed exactly once.

There is no way to create true relevance or linearity in user-space of an arbitrary affine type system; any program which preserves affinity is able to exit at any time without losing affinity. How are we able to enforce a behavior if it is correct to exit at any time? *Proof-of-work* is a special case of the type-state pattern which allows the expression of a relevant type *under the assumption that the program continues its normal flow*; ie. system exits are still permitted. The trick to enforcing the use of some object `T` is to specify that a type is a function which must *return* some type `R`, and to ensure that `R` can *only* be instantiated by consuming `T`. Clearly, we cannot prevent `T` from being destroyed in some other way, but we are able to prevent `R` from being *created* any other way.

Realistic languages have many tools for constraining what users may access. Java has *visibility* to prevent field manipulations. Rust has *orphan rules* to prevent imported traits from being implemented for imported types. Languages without any such features won't be able to prevent users from creating the return type `R` without consuming `T`. In these cases, another option is *generative types* which, among other things, allow us to further distinguish types with different origins. Here, generative types may be used to ensure not just *any* `R` is returned, but a particular `R` within our control. As this work uses the Rust language for concrete implementations, we will rely on its ability to prohibit the user from creating `R` by using *empty enum types* for types with no data nor type constraints, and by making its fields and constructors *private* otherwise[Gor].

Consider the following illustrative scenario: We wish to yield control flow to a user-provided function. Within, the user is allowed to do whatever they wish, but we require them to invoke `fulfill` exactly once (which corresponds to 'consuming `R`'). How can we express this in terms the compiler will enforce? Listing 10 demonstrates a possible implementation (omitting all but the essence of 'our' side of the implementation). The user's code would then be permitted to invoke `main` with their own choice of callback function pointer. Our means of control is the interplay between dictating both (a) the *signature* of the callback function and (b) prohibiting the user from constructing or replicating `Promise` or `Fulfill` objects in their own code.

```rust
struct Promise;
struct Fulfilled;

fn fulfill(p: Promise) -> Fulfilled {
  // invoked once per `main`
  return Fulfilled::new();
}

fn main(callback: fn(Promise)->Fulfilled) {
  // ...
  let _ = callback(Promise::new()); // `Fulfilled` discarded.
  // ...
}
```

Listing 10: A demonstration of proof-of-work pattern. Here, the user is able to execute `main` with any function as argument, but it must certainly invoke `fulfill` exactly once.

Chapter 3

Code Generation

In this chapter, we describe the process of translating the Reo compiler's internal representation of a protocol specification into an executable *protocol object* in the Rust language.

3.1 Two-Phase Code Generation

In this section, we explain and motivate our approach of segmenting the code generation process into two distinct phases. Throughout, we refer to the precedent set by the existing Reo compiler backend for generating code in the Java language, as it has seen the language most similar to Rust which has seen significant development.

3.1.1 Generation Sub-tasks

Reo specifications represent connectors declaratively as relations between ports. They are thus well-suited to reasoning about the protocol's properties. In contrast, our target imperative languages such as Java and Rust represent computation such that it corresponds more closely to machine instructions; they are imperative, laying out sequences of actions which together emerge as interaction at runtime. Where interactions in the former can be oriented around the synchronous observations of port values, interactions of the latter must be expressed as sequences of actions, laid out over time. Implementing algorithms for translating between these forms must take care that the translation procedure between these forms preserves the semantics as intended; choosing the incorrect ordering can change the nature of the emergent interaction in unexpected ways. For example, reading memory cell *before* writing it corresponds to a different interaction than reading it *after* writing it.

Rust and Java have in common their strongly-typed language, while is untyped. To facilitate the desired level of specificity for connectors, the *textual Reo* language has support for optional *data type annotations* on ports, which are retained by the Reo compiler for use by the relevant backend. How these annotations are used is not specified, as their interpretation is only sensible for target languages in a case-by-case basis. To us, these annotations are understood as types which the associated ports must be capable of transmitting. For our typed languages, typing information must be injected such that the emitted target code is valid.

Regardless of the intermediate representation, protocol objects must ultimately be emitted in the target language. Aside from expression in the correct syntax, the end result must make explicit any work required to make it *executable* with the desired runtime behavior. Even simple concepts require the support of auxiliary book-keeping structures to maintain the protocol object's state, and specialized *concurrency primitives* are needed to ensure that actions compose into interactions at runtime in the expected way. Clearly, this is very particular to the target language, as they vary greatly on how they fundamentally express operations on data at a granular level.

In summary, we identify and name three sub-tasks of generating target language protcol objects from a Reo protocol description:

T_{act} Synchronous interactions must be decomposed into asynchronous actions, with the roles of each participant laid out in a sequence.

T_{type} Ports must be given data types such that they agree with any (optional) type-annotations in the Reo specification, and successfully type-check in the target language.

T_{run} Details necessary to make the result runnable are included. Symbolic actions are represented as concrete operations on data.

3.1.2 Decoupling the Reo Compiler from Rust

The Reo compiler has an existing backend for generating Java code. It works by generating Java according to the structure of a *template generator*. In this manner, it can be thought of as performing all code-generation sub-tasks at the same time directly from the compiler's intermediate representation. However, the extent to which the Reo compiler is *coupled* to the Java language is reduced through the reliance on a Java library for the granular implementations of structures that are common to all protocol objects; rather than generating these classes each time, the Reo compiler simply generates a dependency. For example, the library defines a `Component` interface, for which the code generator produces a protocol-specific implementor class. Consequently, a significant part of T_{run} is delegated to this library.

For T_{type}, help comes from the Reo compiler itself, which in its current form was developed with support for Java in mind. This is visible in its internal representation. For example, types are given a *default* type if the input specification omits a type-annotation; this is no trouble for Java, which can upcast all classes to `Object` without losing the capabilities on which the protocol relies at runtime, namely (1) equality checks, and (2) object replication[1].

Only T_{act} is performed almost entirely by the template generator. For simple protocols, this task is relatively easy, as there is not much to add when actions are largely concurrent. For example, replicating the contents of a memory cell into a set of others is simply-done in Java by first reading an object reference, and then overwriting the others one at a time. However, ordering dependencies must be resolved very carefully in the general case. The current Java code generator is susceptible to erroneously observing value x at memory cell M in the event that the observation is synchronous with M's value being overwritten by x. Even with the help of the template generator, this translation is sufficiently complex to make detection of these bugs difficult.

Rust is able to mimic Java's approach to create a similar backend through the explicit use of *dynamic dispatch*, such that types can be collapsed to something analogous to Java's `Object` class. If done naïvely, the resulting backend would inherit the problems of its predecessor, and new ones to boot; the Java-like approach is not idiomatic in the context of Rust; it would not make good use of the extensive control of systems resources unique to a systems language. Chapter 4.3 to follow goes into detail about the properties of the protocol runtime. Here, it suffices to say that we wish to implement a runtime that does not rely on heap-allocation of its port-values, and thus cannot rely entirely on dynamic dispatch. Furthermore, our runtime wishes to perform more extensive optimizations, relying on the unique abilities of our systems language to manipulate its resources at a low level. All these extensions pose a problem in particular for T_{run}, as runtime properties directly influence how the executable protocol objects are represented. Our work in unremarkable in its solution to this problem: we delegate T_{act} to a Rust library. However, we make this separation more extreme. In a nutshell, we wish to partition the work of code generation into two clear *phases*, the former of which performs tasks T_{seq} and T_{type}, and the latter of which performs T_{act}. To minimize coupling between the modules performing these tasks, the interface between phases is made terse, unambiguous and explicit in the definition of a new intermediate representation of protocol connectors: the *imperative form*.

[1]In the chapter to follow, we discuss the consequence of this approach, as it necessarily assumes that operations for checking equality and replication can be resolved for any types.

3.1.3 Temporary Simplifications

Our intention is to isolate the Reo backend from Rust's specifics as extensively as possible. In this manner, we decouple the modules responsible for the code-generation subtasks in accordance with good software engineering practices. Furthermore, it facilitates the *re-use* of the first phase of the code generation process for *other* imperative programming language targets. The section to follow defines imperative form to be as target-language agnostic as possible. However, for the sake of minimizing the disturbance to the Reo tooling ecosystem, we still embrace the per-target structure for Reo code generation for now. As such, the Reo compiler still specifies a Rust language target, and emits executable Rust source as a result. Our representation of the *imperative form* is expressed in Rust syntax (as the `ProtoDef` type) such that this reliance on an intermediate representation is invisible to the end user. As far as they are concerned, Reo generates native Rust that just happens to *somehow* make minimal use of Rust-specific syntax. `ProtoDef` corresponds closely to the definition of imperative form, facilitating this decision being overturned in future with minimal effort.

3.2 Imperative Form

In this section, we define our new intermediate representation of Reo protocol specifications. We include an intuitive look at how it captures the details of the Reo compiler's internal representation, but such that only T_{act} remains to be performed before the finished Rust source code can be emitted.

3.2.1 Intuition

Imperative form makes explicit the *ordering* of events for which the ordering cannot be arbitrary. In other words, synchronous concurrency can still be expressed, but operations whose correctness relies on it are ordered explicitly. For very simple protocols, imperative form is very similar to the compiler's RBA-like internal representation; they have in common that they partition rules into (1) *guards* which contain the *synchronization-* and *data-constraints*, and (2) *assignments* which mutate the protocol's state and produce observable events such as boundary ports putting and getting.

Imperative form differs from that of the compiler's intermediate representation the more extensively the protocol makes use of synchronous manipulations of its state, ie. the more the protocol's state is mutated *within* a single rule. For example, consider a protocol in RBF with data constraint $f(P_0) = f(P_1)$ and synchronization constraint $\{P_0, P_1\}$ with input (putter) ports P_{0-1}. This rule can be understood as "P_{0-1} fire *if* the results of function f on their put-values are equivalent". Here, the results of f clearly cannot be compared until *after* they are computed, but these intermediate values must be *stored* somewhere in the meantime. How can we correctly store these values (mutating the state) *before* we know whether this rule will fire and should thus mutate the state? Our solution is to ensure that these sorts of situations can occur only if these 'transient' state-mutations can be *undone* without observable effects; erroneous state mutation is not a problem if it is immediately reverted. Thus, imperative form models a synchronous rule firing as a *transaction* by partitioning its actions on either side of some *commit*-instant. Actions thus then fall into one of two categories:

- **Transient Actions**
  Actions *before* committing may only perform mutations of the protocol state that can be *rolled back* reversing their effects. Implicitly, this prohibits any actions that are visible to an outside observer. Actions are also allowed to trigger a rollback and abort.

- **Persistent Actions**
  Action *after* committing may irrevocably mutate the state and produce observable effects, but may not trigger a rollback.

For our example RBF, the check for equality acts as a *guard*, but its computation involves the values put by P_{0-1}. To be valid, f is required to be a pure function which cannot be observed to *consume* the values of P_{0-1} (eg. by reading them via immutable reference), and is invoked to produce three temporary variables, which can subsequently be checked for equality. If the check fails, all actions *roll back*, discarding our temporary variables and aborting the rule firing. This corresponds to a guard that was not *satisfied*. On the other hand, if the equality passes, the rule *commits* and the values of P_{0-1} are irrevocably consumed (and in this case, discarded). In this way, P_{0-1} only observe their data being consumed if the rule fires.

Conceptualizing an imperative form rule purely as action-sequences gives good intuition for the intention behind the representation. However, this would require concurrent events to be sequentialized. However, many actions are not given in the compiler's internal representation in this form. For example, the ordering of assignments is left unspecified. There is no harm in sequentializing these actions if the end result is indeed sequential. For example, the Java protocol object indeed assigns to memory cells sequentially. Rather than assuming this will always be the case[2], we represents persistent actions as *concurrent* movements from sources to destinations. In the same manner, is it unnecessary to sequentialize the observations in the *guard*; it does not matter the order

[2]Indeed, our final implementation dispatches data movements sequentially, but they are completed concurrently. See Section 4.3.4.

in which ports are checked for availability. For this reason, imperative form is ultimately represented with three distinct structures (1) unordered transient actions performed before (2) ordered transient actions with rollback, and finally (3) unordered persistent actions.

3.2.2 Definition

A protocol description in imperative form must be provided by a single structure which provides mappings for symbolic names such that they can be resolved as they are encountered when traversing the rest of the definition. This *name definition* structure is similar to a *symbol table*. The behavior of the connector itself is defined by a set[3] of *imperative rules*, each corresponding to an RBA rule, given by a tuple (P, I, M) with:

1. **Premise** P
   A tuple of three *identifier* sets (P_R, P_F, P_E). P_R contains the set of identifiers whose values must be 'ready', and are thus in stable state, involved in the rule. The subset of identifiers belonging to ports thus encodes the *synchronization constraint* of the associated RBA. P_F and P_E are the sets of *memory variables* which must be known to be full and empty respectively, such that it is known whether they can be read or written from. The rule can certainly not consider firing unless all ports are ready and all memory cells are in the specified states.

2. **Instructions** I
   A list of reversible *instructions* which are performed in sequence. These instructions have no immediately observable effects, such that they can be reverted in the event of a *rollback*. Concretely, each instruction is one of:

   - $\mathtt{check}(p)$
     Trigger a rollback if predicate p over data is satisfied.
   - $\mathtt{fill}_P(m, p)$
     Fill an empty memory variable m with the result of a predicate p over available data. It's data type is implicitly *boolean*.
   - $\mathtt{fill}_F(m, f, a)$
     Fill an empty memory variable m with the result of invoking function f with parameters a, a list of references to data variables with length matching the arity of f. It is incorrect for f to *mutate* its arguments, as this would result in observable effects which cannot be rolled back.

[3] In our implementation, these rules are provided in an ordered list, primarily for the purpose of making for more comprehensible error messages.

- $swap(m_0, m_1)$
  Swap[4] the values in two memory variables m_0 and m_1.

  If a rollback is triggered by $\mathtt{check}$, any swapped memory cells are swapped back, and any memory cells whose values were created by $\mathtt{fill}_P$ or $\mathtt{fill}_F$ are destroyed.

3. **Movements** M
   A mapping from *resources* to any identifiers that can act as getters (getter ports and empty memory cells). This represents the *observable effects* of the rule firing after instructions are performed without triggering rollback.

As an example to demonstrate this representation, the RBA rule in the previous section with data constraint $f(P_0) = f(P_1)$ and synchronization constraint $\{P_0, P_1\}$ can be represented in the imperative-form rule with:

rule element	value
premise	$(\{P_0, P_1\}, \{\}, \{t_0, t_1\})$
instructions	$[\mathtt{fill}_F(t_0, f, [P_0]), \quad \mathtt{fill}_F(t_1, f, [P_1]), \quad \mathtt{check}(t_0 = t_1)]$
movements	$\{\}$

For our purposes, omitting a source identifier s from M can be interpreted as $s \to \{\}$ by default, understood to mean 'discard s'. For other applications, it may be preferable to require that each source have an explicitly-defined destination set. For example, this would be useful in the context of a *linear type system* in which data cannot trivially be *destroyed*.

3.3 Code Generation Pipeline

In this section we explore the translation procedure of protocols, starting from their declarative Reo specification in the Reo compiler's internal representation to executable Rust. This is done in two distinct phases, as explained in Section 3.1.2.

3.3.1 Phase 1: Reo Compiler Backend

The Rust backend of the Reo compiler translates from the compiler's intermediate representation to Rust. Section 3.1.2 explains how this task is partitioned such that the compiler itself only performs the first of the two phases that comprise this process. Rust is still the target, but rather than generating the protocol object to be executed directly, Reo outputs a source file that contains a `ProtoDef`, the Rust-syntax version of a protocol in *imperative form*.

[4]In principle, any reversible data-agnostic manipulation is possible, but swapping values is sufficiently expressive and intuitive for our purposes.

Action Sequencing

The most obvious role the backend plays in the translation process is to spread the contents of the relational, interaction-oriented internal representation over the imperative, action-oriented *imperative rules*. This task is called T_{act} in the previous sections. The process begins given (1) the set of port putters in the synchronization set, (2) a mapping from port-getter to `Term`, (3) a mapping from port-putter to `Term`, and (4) a `Formula`, representing the rule's guard. Terms and formulas are recursive data structures, and can both be thought of as *sum types*. The former describes a an expression to be computed at runtime. For example, `MemoryCell(A)` and `True` are both terms that can be assigned to memory cells and retrieved by getter-ports. `Formula` is similar, but used in the context of predicates; formulas are essentially terms known to have a boolean data-type, and thus come with variants for the usual boolean operations. For example, `True`, `And(Eq(A,B), Eq(B,C))` are examples of formulas. The backend traverses the compiler's internal representation, constructing imperative form rules one at a time, also populating the associated `NameDef` symbol table. For each such rule, it helps to consider each of its distinct structures in isolation, initially without any optimizations:

P **Premise**

The first task of the backend is to build the imperative rule's *premise*. Observe that the this step is nearly trivial, as the synchronization set, set of port-getters, and set of port-putters are already provided. The only exception is for memory cells which are *read*, which must be derived by encountering their occurrences inside terms and formulas for the guard and assignments. Clearly, no information is added by explicitly isolating the set of readable memory cells; we are able to derive it despite being absent in the internal representation, and the term is only meaningful if one cannot read from a memory cell *unless* it is marked as readable. We nevertheless perform this preprocessing as it makes explicit that the memory cell must be available for the interaction without having to 'look ahead' into terms and formulas. Furthermore, its inclusion provides us a simplifying property: if an identifier does not occur in the premise, it must refer to a *temporary variable*.

I **Instructions**

Next, the backend generates a list of *instructions*, representing the *tentative actions* that may be rolled back if the rule aborts instead of committing. Here, $\mathtt{fill}_P$ and $\mathtt{fill}_F$ instructions are inserted to compute the results of any evaluations of predicates or functions and bind them to temporary variables. Next, special cases of each memory cell M synchronously read-from and written-to are detected. For each, a temporary variable T_M is created, and an instruction $swap(M, T_M)$ is appended to I. As a result, data movements can be conducted in parallel without fear of a race condition emerging; although M is logically involved both in reading and writing, these actions are distinguished by name; M is as independent from T_M as any two distinct memory cells. M receives and stores its new (written) value M, and its old value can concurrently be read from T_M without interference. Finally, a *check* instruction is appended to the list, parameterized with an analog to the compiler's `Formula` type.

M **Movements**
All port-getters and memory cells which are *written to* are grouped by source to produce M, a mapping from sources to destination sets. Each mapping is additionally annotated with a flag to indicate whether the value is retained at the source, ie. the mapping is of type $\mathtt{Identifier} \Rightarrow (\mathtt{boolean}, \{\mathtt{Identifier}\})$. As an example, moving a value from A to B results in A mapping to $(\mathit{false}, \{B\})$.

Clearly, many opportunities exist for optimization of the imperative form, as it affords some degree of *over*-specification and can represent the same connector with a different sequence of instructions. As an example, it is possible to fragment our guard into several *check* instructions. This can be used to achieve *short-circuiting* behavior; as long as no guard reasons about a temporary variable *prior* to its definition, a check may be moved earlier such that rollbacks may be triggered earlier and work can be avoided. These opportunities were not explored extensively in this work, as instructions in practice tend to be simple enough not to afford many such opportunities. More significant is the *omission* of trivial checks entirely. This can be done by traversing instructions with a model of the protocol's state, initialized to match the premise. Checks known to always hold can be omitted, and checks known to never hold can result in the rule being discarded entirely (it would never fire!). This optimization is incorporated, and works to great effect in practice.

Type Classification and Constraining

Previously referred to as T_{type}, the backend must extract data-type information about its ports and memory cells from the compiler's internal representation such that the generated Rust is valid. Our approach is to begin by assuming that every port, memory cell and temporary variable has its own unrelated type. The types are then *constrained* as a result of discovering the ways in which they are related from the protocol definition. For example, the presence of a term `Eq(A,B)` 'collapses' the types of A and B into one equivalence class. Still, at this level, types are purely symbolic.

These symbolic types exist only in the first phase of the code generator. They are not only resolved prior to phase two, but they are not present in that phase at all. Our imperative form does not deal with the complexity of symbolic (ie. generic or parametric) types. Instead, the Reo compiler relies on the Rust idiom of relying on *dispatch* to resolve concrete types at the last possible moment: at the call site. To achieve this result, Reo generates the `ProtoDef` object wrapped in a *wrapper function*, whose role is primarily to expose generic type parameters for instantiation to the caller, and construct a `ProtoDef` instance within the body, to be invoked with concrete types. Listing 11 gives an example of how this generic function appears to the end user for some trivial protocol. Observe how `protocol_1` relies on generics to let the caller, `main_1` which type to select for `T`. The `TypeInfo` structure represents a port's data type as data. The details of this type are provided in Section 4.3.4.

```rust
fn new_protocol_a() -> ProtoDef {
    ProtoDef {
        name_defs: hashmap! { "A" => Memory(TypeInfo::of::<String>) },
        rules: vec![],
}   }
fn main_a() {
    let _ = protocol_0();
}

fn new_protocol_b<T>() -> ProtoDef {
    ProtoDef {
        name_defs: hashmap! { "A" => Memory(TypeInfo::of::<T>) },
        rules: vec![],
}   }
fn main_b() {
    let _ = protocol_1::<String>();
    let _ = protocol_1::<Bar>();
    let _ = protocol_1::<Foo>();
}
```

Listing 11: Comparison between concrete and generic function definitions for building `ProtoDef` structures. `new_proto_a` uses the concrete `String` type, and will be compiled to a single function as expected. Reo uses the approach of `new_proto_b`, which determines the choice of the generic type on demand at the *call site*.

Unlike Java, Rust makes very few promises about the properties of some generic type. It cannot be certain that instances of some generic type `T` will have a defined operation for checking equality, or for safe *value replication*. To facilitate *useful* polymorphism, Rust relies on *type constraints* to act as a contract between caller and callee: the caller will ensure that only types which satisfy the bound may be selected for the type parameter, and the callee is able to interact with its generic arguments in accordance with *behavior* the bounds guarantee the type will define. This approach may be familiar to programmers of Java or C, where this concept manifests as interfaces and declarations (usually in header files) respectively.

The Reo backend cannot guarantee the generated Rust code is sound unless it is careful to add the necessary type bounds to its generic arguments. The internal representation is inspected for cases which will necessitate the use of specialized operations (such as replication) and will annotate its generic types with *trait bounds*. Ultimately, the resulting coalesced, bounded type parameters are reflected in the generated code as part of the function declaration. The second phase of the code generator can rest assured that for every specialized operation inserted, Reo will have already anticipated the need to guarantee the bound is satisfied. Listing 12 gives an example of a generated trait bound for the case where two memory cells are related by being the same, and requiring the definition of an equality operation.

```
1  fn new_protocol<T: Eq>() -> ProtoDef {
2      ProtoDef {
3          name_defs: hashmap! {
4              "A" => Memory(TypeInfo::of::<T>),
5              "B" => Memory(TypeInfo::of::<T>),
6          },
7          rules: vec![
8              premise:   /* omitted */,
9              instructions: vec![Check(Eq("A", "B"))],
10             movements: /* omitted */,
11         ],
12 }   }
13 fn main() {
14     let _ = new_protocol::<String>(); // ok! String implements Eq
15     struct Foo;
16     let _ = new_protocol::<Foo>(); // ERROR! Foo does not implement Eq!
17 }
```

Listing 12: Reo-generated `ProtoDef` building functions with a generic type `T`. Reo inserts trait bounds to ensure the type chosen for `T` has all the needed behavior. In this case, `T: Eq`, ensuring that instances can be checked for equality, as necessitated by the instruction on line 9. Line 16 generates a compile-time error, as `Foo` does not meet the requirements.

Initial Protocol State

The initial state of a protocol object is unusual in that it is included in the textual Reo protocol definition, but omitted from the `ProtoDef`. This design choice reflects how a protocol's initial state is unassuming at first glance, but is significantly different from the rest of the protocols definition: it is the only part of specification that cannot generally be replicated. Rules and name definitions describe *behavior*, and do not involve any elements of the port's data domain directly. By contrast, a protocol's initial state does not *describe* data, it *is* data. By extracting this facet of the specification, `ProtoDef` becomes pure in its role as a *blueprint* for a protocol's behavior.

Nevertheless, textual Reo is able to specify the initial states of memory cells. To support this functionality, the Reo compiler itself generates code which *builds and injects* the initial values for any initially-filled cells in a controlled environment. To mirror the textual descriptions that define the initial values, these memory types acquire a final trait bound such that their initial values can be constructed per protocol object instantiation *within* the confines of the function that the Reo compiler generates. Listing 13 exemplifies a trivial protocol where A is a memory cell defined as being initialized by some value represented by the string `'hello'`. The generated code inserts a trait dependency, ensuring that the type parameter `T` defines this operation. Observe how now the return result is no longer `ProtoDef`, but rather `ProtoHandle`. While the former represents a re-usable *blueprint* for instantiating a `ProtoHandle`, the latter represents an instantiated protocol, initialized and ready to run; the only user-facing means of generating a new object of the same protocol is to again invoke `protocol`.

3.3.2 Phase 2: Rust Library

Our work follows the precedent set by the Java code generator in relying on a library in the target language to define the lion's share of the behavior for our runtime protocol objects. For Rust, these definitions are bundled into the **Reo-rs** library, which is added as a dependency to the code generated by the Reo compiler's backend. Chapter 4.3 explores the architecture and behavior of our executable protocol objects in detail. For now, it suffices to say that our approach is to represent executable protocols as extensively preprocessed *data structures* which then drive the behavior of a lightweight *interpreter* at runtime. This data-representation is often called *commandification* [Nys14].

```rust
fn new_protocol<T: From<&str>>() -> ProtoHandle {
    ProtoDef {
        name_defs: hashmap! { "A" => Memory(TypeInfo::of::<T>) },
        rules: vec![],
    }.build(MemInit::default().with<T>("hello".into())).unwrap()
}
fn main() {
    let _ = new_protocol::<String>(); // ok! String: From<&str>
}
```

Listing 13: `new_protocol` instantiates a runnable protocol object by internally building a `ProtoDef` description, and then immediately instantiating it with a `MemInitial`, whose contents are constructed from parsed strings. In this manner, Reo can control the initialization of protocol instances for any suitable type `T`. Note that `&str` is the immutable reference type of a sized string slice, while `String` is an owned, mutable character buffer. Both are common types from the Rust standard library.

Soundness Check

Our backend is novel in that the work of constructing the executable protocol object requires crossing an API-boundary. Rather than trusting the well-formedness of the Reo-generated `ProtoDef`, Reo-rs will check that its input is internally-consistent. By adding these checks, the dependency between the Reo compiler and Reo-rs is *unidirectional*; users are free to safely construct protocol objects using Reo-rs in their applications directly. The most obvious advantage to this approach is an additional layer of safety, allowing for the compiler and Reo-rs to be maintained separately, eg. if the compiler acquires a bug from a new update, the error cannot propagate into Reo-rs unnoticed. Another advantage is the avenues for future work this opens up. Our approach treats protocol structures as data, facilitating their mutation at runtime, resulting in *dynamic protocol reconfiguration*, although exploring this further is beyond the scope of this work.

In native Rust, the usual variable scoping rules apply to ensure that a symbolic identifier is resolved to a meaningful memory position. These systems are so familiar to us that we usually take the complexity of their work for granted. Rules that are second nature to us require explicit enforcement to replicate the work of the checker. As `ProtoDef` *commandifies* the behavior to be later executed, the Rust compiler does not interpret these actions in the normal way, and we must mimic its behavior manually. We take this idea a step further by relying on the `ProtoDef`'s *premise* to facilitate a mechanism that mimics the Rust *borrow checker* system; rules trace which variables are *valid* (ie. initialized) throughout the rule's execution top to bottom, tracking changes as a result of actions filling or swapping their values. In this manner, we are able to catch invalid memory accesses during `build`, rather than encountering them at runtime. An additional perk of mimicking this system is our ability to detect the occurrence of values which *must* be consumed during the rule's firing, but whose consumption is not included in the specification. For example, an *imperative rule* may include some putter port P in its ready-set (and thus, its synchronization constraint), but associate no *movement* with P's value. A naïve implementation which overlooks such occurrences may introduce *memory leaks* for such cases if it takes the specification at face value. Instead, our custom borrow checker will reach the end of of the specified actions and conclude that as P was not explicitly emptied, it will insert a trivial movement $P \mapsto \{\}$ to ensure the value is consumed. This is analogous to how Rust's borrow checker inserts `drop` calls to destroy local variables which go out of scope unconsumed. By performing this extensive checking, Reo-rs affords an expressive `build` function, capable of giving detailed error information in response to an invalid input. The signature of this function is given in Listing 14.

```
fn build_proto(&ProtoDef, MemInitial) -> Result<ProtoHandle, BuildError>;
type BuildError = (Option<usize>, BuildErrorInfo);
enum BuildErrorInfo {
    ConflictingMemPremise { name: Ident },
    PutterCannotGet { name: Ident },
    MovementTypeMismatch { getter: Ident, putter: Ident },
    GetterHasMuliplePutters { name: Ident },
    InitialTypeMismatch { name: Name },
    /* 15 more variants */
}
type Ident = &'static str; // static string literal type
```

Listing 14: Signature of the `build` function. Its inputs are (1) an immutable reference to a `ProtoDef`, which is used to determine the protocol's behavior, and (2) a `MemInitial`, which stores initialized memory cells to be incorporated into the protocol's state. The return result is an enumeration type, returning `ProtoHandle` upon success, and a tuple on failure, whose elements are, respectively (1) the index of the imperative rule where the error occurred if applicable, and (2) another sum type, communicating the nature of the error with additional information.

By restricting our API such that all executable protocol objects are *only* created by `build`, our runtime interpreter is able to rely on the properties we guarantee and avoid checking them itself. In this way, checking for soundness is also an optimization.

Protocol Initialization

As far as Reo is concerned, the entry point to Reo-rs is the `build` method, which uses a `ProtoDef` as a read-only 'blueprint', and consumes a `MemInitial` object to build and initialize the state of `Proto` instance on the heap. This object is returned indirectly, via a `ProtoHandle`, such that access the `Proto` instance can be shared by *cloning* the handle. Listing 13 demonstrates how this appears to the caller.

`ProtoDef` and `Proto` are very similar structures, which both correspond closely to the language-agnostic definition of *imperative form* given in Section 3.2.2. However, they represent the same information differently as they have different purposes.

	`ProtoDef`	`Proto`
purpose	Defines a protocol's behavior.	Is efficiently interpretable at runtime to execute the defined behavior.
identifiers	Symbolic strings.	Indices, keys and pointers directly into data structures.
redundancy	Minimizes redundancy for brevity and to minimize error surface.	Replicates data to optimize for access time.
state	Is stateless, but describes protocol state.	Is stateful, contains conventional protocol state in memory cells, and meta-state for 'bookkeeping' thread access, ie. stateful mutex locks.

`build` initializes the state of the runnable protocol object, `Proto`. Reo-rs defines behavior for bringing these types to life at runtime, complete with granular synchronization primitives and various optimizations. Thus, `build` completes the final sub-task of code generation, $T_{\mathtt{run}}$. Chapter 4.3 to follow details the properties of these runnable protocol objects, including how they are represented and how they behave to act as coordination mediums according to their `ProtoDef` specifications.

Chapter 4

Protocol Runtime

In this section we explore the Rust implementation of Reo-generated protocol objects. Rather that generating the needed structures and behaviour from scratch each time, the Rust back-end follows the precedent of the well-established Java back-end and relies on a single, re-usable dependency for the work common to all protocols. Here, we explore the implementation of this **Reo-rs** library, picking up where we left off from the generation step in Chapter 3 above.

4.1 Examining the Java Implementation

The work of this project can draw from the efforts of previous work on the Reo Compiler. The Java implementation in particular has seen the most frequent and recent updates. This section treats the Java code generator as a touchstone for Reo-generated application code in general. We give a brief overview of the properties inherent to the generated code, and consider the effects of projecting the underlying ideas to the Rust language.

4.1.1 Architecture

Fundamentally, the generated code adheres closely to Reo's literature, revolving around the interplay between `Port` and `Component` objects. From the perspective of a developer looking to integrate a generated Java protocol into their application, the entry point is the `Protocol` component (where 'Protocol' is the name of the associated Reo connector).

Running a system requires an initialization procedure: (1) a `Port` is instantiated per logical port, (2) a `Component` is instantiated per logical component, and (3) pairs of components are linked by overwriting a port-field for both objects with the same instance of `Port`. To get things going, each component must be provided a thread to enter it's main loop; in idiomatic Java, this manifests as calling `new Thread(C).start()` for each component `C`. A simplified example of the initialization procedure is shown in Listing 15 for the simple 'sync' protocol which acts as a one-way channel. In this example, the ports are of type `String`.

```java
Port<String> p0 = new PortWaitNotify<String>();
Port<String> p1 = new PortWaitNotify<String>();

Sender c0 = new Sender();
Receiver c1 = new Receiver();
Sync c2 = new Sync();

p0.setProducer(c0); c0.p0 = p0;
p0.setConsumer(c2); c2.p0 = p0;
p1.setProducer(c2); c2.p1 = p1;
p1.setConsumer(c1); c1.p1 = p1;

new Thread(c0).start();
new Thread(c1).start();
new Thread(c2).start();
```

Listing 15: A simplified example of initialization for a system centered around a `Sync` protocol object, which acts as a channel for transmitting objects of type `String`. Both ports and components are constructed before they are 'linked' in both directions: each port stores a reference to its components, and each component stores references to its ports. The system begins to *run* when each component is given a thread and started.

In a sense, this implementation primarily hinges on `Port` as a communication primitive between threads, and equivalently, between components. For matters of concurrency, operations on port-data involves entering a *critical region*. In contrast, `Components` are used only to store their ports and to be used as name spaces for their `run` function which implements their behavior (which corresponds to RBA rules in the case of the protocol component). Essentially, anything that interacts with `Port` objects can reify a logical component, whether or not this is done by an object implementing the `Component` interface.

4.1.2 Behavior

The representation of protocol rules is very intuitive; a rule is implemented as a block of code which operates on a component's ports. Once generated into Java, the only obvious sign that a component was generated from Reo is its linkage to multiple other components[1]. The (simplified) generated `Component` code of the 'sync' protocol from the previous section is shown in Listing 16. This demonstrates that rules are indeed *commandified*, in that their behavior is encoded in discernible structures (appropriately called `Command`).

The behavior and structure of a component go together, and are generated by Reo at a relatively granular level. As such, the encoding of memory cells is natural also. Memory cells can be found next to ports in the fields of a `Component`.

4.1.3 Observations

It is very easy to see the correspondence between a generated Java protocol and its Reo definition. This carries over to how components and ports are used by an application developer. Next, we consider their higher-level properties that follow from the observations in the previous sections:

1. **Protocol Event Loop**
   Protocols are fundamentally *passive* in that they do not act until acted upon. Nevertheless, protocols each have their own dedicated thread that waits in a loop for a *notification* from its monitor. Notifications originate from a component's own `Ports` in the event of a `put` or `get` invocation. For this reason, protocols and components are related in both directions, afforded by setting a port variable in one direction, and functions `setProducer` and `setConsumer` in the other.

   True to the intuition behind the RBA model, the protocol must *check* which (if any) commands can be fired, and keep spinning, trying rules while *any* guard is satisfied. This is unfortunate, as this approach requires guards to be evaluated repeatedly. As the protocol relies on the actions of other components to make progress, it is counter-productive for it to spend a lot of system resources evaluating guards to *false*. In cases where threads must share processor time, the excessive work of the protocol component will begin to get in the way of other components making progress, in turn leading yet more guards to evaluate to *false*.

2. **Reference Passing**
   Java is a managed programming language whose garbage collector is central to how the language works. To support the transmission of arbitrary data types, `Port` is generic over a type. The language only supports this kind of polymorphism for objects. Unlike primitives (such as `int`), the

[1]The distinction between 'protocol' and 'compute' components is tenuous at the best of times. If compute components are allowed to interact directly with one another, the distinction observed here disappears also.

```java
private static class Sync implements Component {
    public volatile Port<String> p0, p1;

    private Guard[] guards = new Guard[]{
        new Guard(){
            public Boolean guard(){
                return (p1.hasGet() && (!(p0.peek() == null)));
    }   },  };

    private Command[] commands = new Command[]{
        new Command(){
            public void update(){
                p1.put(p0.peek());
                p0.get();
    }   },  };

    public void run() {
        int i = 0;
        while (true) {
            if(guards[i].guard())commands[i].update();
            i = i==guard.length ? 0 : i+1;
            synchronized (this) {
                while(true) {
                    if (p1.hasGet() && (!(p0.peek() == null))) break;
                    try {
                        wait();
                    } catch (InterruptedException e) { }
}   }   }   }   }
```

Listing 16: A simplified example of a Reo-generated Java protocol class for the *sync* connector. By convention, it is started by invoking `start`, which is a method inherited from the `Runnable` interface which `Component` extends. This method assumes that all ports are correctly initialized and linked to another ‘compute’ port. Its RBA-like behavior comes from an array of guards and commands which it iterates over in a loop, firing rules as possible forever.

data for objects is stored on the heap and is garbage collected by the Java Virtual Machine. Variables of such objects are therefore moved around the stack by *reference*. Moving and replicating values is cheap and easy, as they always have a small (pointer-sized) representation on the stack.

A minor drawback is the need for indirection when performing operations that need to *follow* the reference. For example, comparing two `Integer` objects requires that the `int` primitives backing them on the heap be retrieved and compared. Equality is an example of an operation that the Reo protocol thread can be expected to perform frequently. The cost of this indirection depends on a myriad of factors, but is at its worst when it results in new, spread-out locations each time. This case might arise, for example, if the `Sender` continuously created new `Integer` objects and sent them through its port. Another drawback is the *requirement* to allocate primitives on the heap before they can be sent through a port. This is not usually a problem in the case of Java, as in practice, almost everything is going to be stored on the heap with or without Reo.

This aspect of the generated Java code will require the most change for the Rust version, as Rust has a very different model for memory management; it does not use a garbage collector by default, and structures are stored first and foremost on the *stack* as in the C language.

3. **Two Hops for Data**
As protocols are components like any other, even the most trivial of data-movements require values to hop at least twice: into the protocol, and out of the protocol. Fortunately, as stated above, the cost of the 'hop' itself is trivial, as it will always be a small reference. The problem is the time delay *between* the hops, as it will often involve actions of three distinct threads in series (with the protocol in the middle).

4. **Vulnerable to User Error**
The construction and linking of components with ports is not something the protocol itself is concerned with. Indeed, *every* component assumes that their port-variables will be initialized by their environment. At the outermost level, this environment is in the application developer's hands. Components make no attempt to verify that they are correctly linked according to the specification; currently, there is not any infrastructure in place to support this checking if it were desired. As a result, it is possible make mistakes such as fusing two of a protocol's ports into one. Whether this is a problem worth solving depends on the burden of responsibility that Reo intends to place on the end user. These difficulties cannot be completely avoided, but approaches exist to minimize these opportunities for mistakes.

   While ports are clearly directional 'from the inside out' (ports store distinct references to their producer and consumer components), the same is not so 'from the outside in'. Neither of a port's components is prevented from indiscriminately calling `put` or `get`. The assignment of a port's values for 'producer' and 'consumer' component is in user-space also. As a consequence, these fields may not agree with the components that interact with the ports at all. In fact, any number of components may store a reference to a port, each arbitrarily calling `put` and `get`. If done unintentionally, this would lead to *lost wakeups*; the thread blocking for a notification after calling acting on the port is not the same as the thread receiving the notification. Solutions can be conceived to *wrap* ports in objects that constrain the API of a port to one of the two 'directions'. However, without affine types, there is no obvious way to ensure the *number* of components accessing a port is correct. In Rust, limiting these accesses becomes feasible.

5. **Port Data Aliasing**
   In Reo, it is common for connectors to replicate port data. Owing to the nature of Java, this is currently achieved by duplicating references, where replication is also known as *aliasing*. For immutable objects, aliasing has no observable side effects, and thus does not threaten Reo's value-passing semantics. However, Reo ports permit instantiation with *any* object-type. Even if the operations are thread-safe, this causes *incorrect* behavior, as a component might observe their data changing seemingly under their feet. Worse still, objects which are not thread-safe can cause undefined behavior. This is a result of Java's view on memory safety having inverted priorities to Rust. In Java, operations are unsafe by default, and the programmer must go out of their way to protect themselves from data races, access of invalid memory and corruption. In Rust, the *ownership system* is based on the prohibition of mutably-aliased variables. Achieving replication in Rust will require some effort to convince the compiler of safety before a program will compile.

6. **Non-Terminating Protocols**
   Currently, Reo-generated protocol objects loop forever unless they raise an exception and crash. For protocols that can perform actions with observable side-effects in the absence of other components, this is perhaps a good idea. However, in the majority of realistic cases, protocols are indeed passive, and cannot do meaningful work as the only component. Reo semantics tend to reason about *infinite* behaviors. However, real programs often do *end*, and it is desirable that the program's exit is not held up by an endlessly-blocked protocol thread.

7. **Protocol Components Cannot be Composed at Runtime**
   (TODO is this the place to explain this?) Ports allow data to move from the putter (or 'producer') and getter (or 'consumer') components as an *atomic* operation by delaying `put` or `get` operations until their counterpart is called also. This causes problems for the implementation of RBAs with rules whose guards are predicated by the data they move. How

can a protocol *decide* if it should fire as a function of values it can only obtain *by* firing? This ability to reason about the future is currently still a luxury limited to models such as TDS. The Java implementation gets around this problem by introducing *asymmetry* between 'compute' and 'protocol' components. Protocols are allowed to *cheat*. The `Port` object has additional operations to inspect a value without consuming it: `peek` and `hasGet`. However, this asymmetry means that composing two Java protocol components (by linking them with ports) does *not* result in a component with their composed behavior. Solving this problem in earnest requires continuously-connected protocols to reason about their distributed state, which is a problem beyond the scope of this work. Reo's relationship with *liveness properties* is explored in Section 5.

8. **Sequential Coordination**
The Java implementation is structured such with *ports* being the critical region between components. As protocols have multiple ports, at first glance it may appear that coordination events could occur in parallel. However, no communication through protocol `P` happens without the single thread in `P`'s `run` method. Indeed, `put` and `get` operations can be *started* in parallel by the boundary components, but `P` can only complete it's half of these operations sequentially.

4.2 Requirements and Guidelines Defined

The Reo compiler's Java code generator were examined in Section 4.1, resulting in the extraction of some high-level observations, enumerated in Section 4.1.3. Our goal is to build upon what exists, and to improve where possible. The goal of this section is to make concrete the priorities behind the design process of Reo-rs. This serves a dual purpose: (1) Their existence helps guide the design process of Reo-rs, helping to make design decisions and making the reader cognizant of the library's priorities. (2) Readers most interested in the finished product are able to use the evaluation in Section 4.4 as a means of overviewing the chapter.

First, we identify a number of functional requirements. These determine the functionality that Reo-rs prioritizes, and which safety properties the end user is able to rely upon.

$\mathbf{R}_{value}$ Preserve Reo's value passing semantics. No user interaction should contradict these semantics. This precludes data races as a result of *mutably aliasing* resources which the user considers to be independent values.

R_{init} Prevent the protocol from being initialized in an inconsistent state. Prevent port objects from being unitialized, unsafely accessed in parallel, or incorrectly connected to the protocol.

R_{ffi} Facilitate a foreign function interface with other systems languages C and C++. Ports and protocol objects should be accessible from those languages as well as Rust, such that they can be constructed, used and destroyed.

Furthermore, we identify *guidelines* for softer, qualitative properties of the implementation of Reo-rs. These primarily address matters of feature priority, and provide a basis for making the assumptions necessary for meaningful optimization. The Java implementation is the closest parallel to Reo-rs for performance, and is far too different to form the basis of any thorough quantitative testing. Hence, these guidelines are not formulated such that they can be absolutely satisfied. Instead, performance benchmarking in Chapter 6 *motivates* the extent to which the guidelines are followed by Reo-rs.

G_{data} Allow the transmission of large data types without requiring the user to move the from the stack to the heap. Minimize the number of times data must be *moved* in memory such that data transmission remains performant for types with large representations.

G_{fast} Minimize the overhead of *control operations* for the protocol object to route payloads and perform bookkeeping. In particular, minimize the cost of *evaluating* the rules before one is selected for firing.

G_{end} Facilitate the protocol object being destroyed and its resources freed. Facilitate a means of detecting termination which is correct and ergonomic.

4.3 Protocol Runtime

Here, we explore the Reo-rs library implementation which provides user-facing types that work together to behave as coordination primitives at runtime in accordance to the imperative-form-like `ProtoDef` used to configure them.

First, Section 4.3.1 explains the functionality of Reo-rs through the lens of the user-facing API. Beyond this surface level, there is a large design space for implementing the internals. As such, this project necessitated the exploration of several of these possibilities before settling on a design. Section 4.3.2 begins by summarizing these possibilities and motivating the choice for this work. Section 4.3.3 details how this design is laid out as more concrete type definitions in the library (most of which the user will never see). Finally, Section 4.3.4 describes how these structures interact to emerge as coordination.

4.3.1 Application User Interface

The Reo compiler generates protocol descriptions in imperative form, which then are transformed by Reo-rs into runnable objects. The user therefore interacts mostly with Reo-rs itself, and Reo provides only the entry point for building particularly instances of protocol object. In this section we explain which functionality of Reo-rs is user-facing, focusing primarily on which requirements are satisfied.

Construction and Destruction

Reo-rs is built to interface with the Reo compiler, but it is not dependent on it. The entry point for protocol objects is the `ProtoDef` type, which is a concrete realization of the (logical) imperative form. Instances of this type are visible in Listings **??** and **??**, but its definition is omitted for brevity (available in the source code). Regardless of whether the constructed `ProtoDef` was Reo-generated, it is instantiated along with any initial memory cells (in the `MemInitial` structure) to produce a `ProtoHandle`. This type has a small, pointer-sized shallow representation (ie. 32 or 64 bits) to the `Proto` structure on the heap. The handle is opaque to the user, at first glance offering no functionality other than to replicate the handle (aliasing the `Proto`) or to destroy the handle (in Rust this is done implicitly by letting its binding go out of scope, or explicitly by invoking `mem::drop`). Once the last handle to a `Proto` instance goes out of scope, its resources are freed. This is achieved by relying on the Rust-idiomatic `Arc` type, ('atomic reference-counted') for the definition of `ProtoHandle`.

The system cannot come to life until the user acquires some of the protocol's *ports*. In Reo-rs, there is indeed a singular type for a port object (`PortCommon`), but the user does not acquire it directly; instead, users are able to create and destroy two distinct types that *wrap* `PortCommon`: `Putter` and `Getter`. In this manner, the *direction* of the logical type is enforced; only `Putter` can put, and only `Getter` can get. Both are created by the `claim` function, which is invoked parameterized with (1) a `Protohandle`, (2) the data type of the port as a `TypeInfo`, and the name of the port to identify it ('name' is a string, corresponding to the one used in the definition of the `ProtoDef`, exemplified in Listing **??**). In this manner, the `Proto` is involved in the creation of these port objects and is able to enforce that (1) ports have the correct *direction*, (2) no logical port may have more than one putters or getters at a time, and (3) the port type is instantiated with the correct *data type*. All of these types and operations are also perfectly thread-safe, relying on the proto-lock, to be discussed in Section 4.3.3. When used from Rust, `Getter` and `Putter` have generic type paramters that do not influence their memory representation, but rather enforce that the data-type with which they are claimed matches that used in puts and gets. These generic arguments are not available in C; instead, a variant of these port types are available that store the `TypeId` as data, and reflect on it whenever putting or getting to preserve safety.

Putters and getters store replicas of their proto-handles (by necessity), which can be examined for subsequent claims; this means that the original `ProtoHandle` can be safely discarded. The `Proto` is destroyed only when all proto-handles *and* all of its ports are destroyed. `Proto` is also involved in the destruction of ports, which affords the user discarding and *reclaiming* ports at will. These port objects can also be treated as data, sent across threads, stored in buffers or moved to the heap. However, there is no safe means of *replicating* a port object, ensuring that no two threads can be putting or getting on the same logical port at once.

Port Operations

Putters and getters rely on Rust's *mutability* rules for ensuring that at most one thread is accessing each of a protocol's logical ports at a time; concretely, `Get` has method `get` with signature `fn get(&mut self) -> T`, and likewise for `put`. Otherwise, the operations of these two types are distinct.

Getters offer `get`, which blocks the calling thread, and returns an object of type `T`, where `T` is the data type with which the putter was claimed. Other variants of this method exist which may be more ergonomic for the user depending on the circumstance. `get_timeout` takes an extra `Duration` parameter, and will attempt to return once the duration has elapsed without the port being involved in a rule firing. Rather than `T`, this method returns `Option<T>`, where the `Option::None` variant communicates that timeout was exceeded before a value was acquired. For cases where the getter wishes to participate in a rule firing, but for whatever reason does not actually want the data, `get_signal` is available; this method behaves like `get`, but returns no value. Finally, `get_signal_timeout` behaves as expected, returning a boolean `true` to communicate whether the getter participated in a rule firing before timing out; in either case, no element of the port's data type is returned.

Putters also have variants of `code` available that mirror those of the getter; `put_timeout` behaves as expected. However, putters do not have a parallel to the getters' `get_signal`. At the moment the datum is offered, it is unknown whether any getters (directly involved in the firing, or indirectly acquiring it after it is stored in the protocol's memory) will want the data. As putters 'go first', they are subservient to the choice of the getters, and must be ready to deal with the situation in which their put-datum was involved in a rule firing, but not consumed. Reo-rs attempts to avoid the costs of creating or destroying port-data values wherever possible, as these operations may be arbitrarily expensive. As such, rather than the expected signature of `fn(&mut self, T)`, the `put` method returns a value of type `Option<T>`; if their value is not consumed, it is instead *returned* to the putter. The expectation is that this option is beneficial to the user, they are able to re-send the same datum, or use it elsewhere; Here, a return value of `Option::None` communicates that the value was consumed. If the putter is not concerned with retaining an unconsumed value, the method variant `put_drop` is available which returns the datum regardless of the actions of getters.

Both putters and getters transfer their data into and out of the stack frames of their port operations respectively *by value*. Owing to its focus on resource affinity, Rust relies on its *move semantics* for both (a) relocating bytes in memory, and (b) transferring a semantic resource binding. For cases where the relocation of bytes is not necessary, LLVM can often be relied upon to optimize the memory movements away, passing consumed resources by reference 'under the hood'. However, these optimizations are not guaranteed. Users of the Rust language have grappled with this shortcoming for years, but the search for a satisfying solution remains an open problem [Mat15]. For us, this problem it presents itself when defining our API. Neither of the available options satisfy all of our requirements; $\mathbf{R}^{\mathbf{F}}_{\mathbf{correct}}$ asserts that we cannot rely on reference passing from user-controlled code, as it becomes impossible to enforce Reo's value passing semantics. On the other hand, relying on Rust's move semantics has the potential to insert up to two copies of our values, working against our satisfaction of $\mathbf{R}^{\mathbf{N}}_{\mathbf{data}}$. Our choices are as follows:

1. **Value passing.**
   We expose safe functions `put` and `get` to consume and return data *by value*, guaranteeing correctness. Depending on the compilation environment, it may require up to two moves of the data.

2. **Reference passing. User provides correctness.**
   We expose unsafe functions `put_in_place` and `get_in_place` to consume and return data *by reference*, relying on the user's care to initialize or drop the corresponding value themselves. According to Rust's idiom, these functions are marked with the *unsafe* keyword, moving the burden of preserving correctness to the user. Their use necessitates explicitly wrapping them in `unsafe` blocks, such that it is easy to identify weak spots in a program's correctness.

   These functions are ideal for a *foreign function interface* (to C, for example), for which *unsafe* is often necessary anyway.

3. **Value pass a 'referring' type.**
   The user passes type `Q` through ports by value, where `Q` *represents* the real type, `T`, ie. the user re-interprets what Reo-rs will consider to be the

port's data type. As far as the protocol is concerned, `put` and `get` is called as in case (1). The simplest approach is to mimic that of Java, passing type `Box<T>`, representing an owned heap-allocated resource as an *owned pointer*. Other approaches may be simple (eg: transmitting an integer which keys into a shared map) or arbitrarily complicated (Several Rust libraries exist for decoupling an object's data from its *ownership*, such as `rent_to_own`, `managed` and `swapper`[2]).

Options (1) and (3) are already supported by the methods described thus-far. To cover the remaining case, we rely on Rust's *unsafe* API idiom to expose this trade-off for users to make on a case-by-case basis. Additional 'raw' method variants are available which mimic Rust's move semantic by consuming and producing port data at the source and destination of a provided *pointer*. Users are responsible for safely initializing and uninitializing the datum appropriately to avoid leaking memory or encountering undefined behavior. As is the idiom, these functions are explicitly marked with *unsafe*, communicating to users that their use requires additional care to use these functions in accordance with their documentation. This option is therefore well-suited to the foreign-function interface with C.

In sections to follow, we ignore this complication and presume the safer value-passing variants are used only.

Interface with C and C++

Programming languages rely on an *ABI* (application binary interface) for translating functions and types into binary according to a dependable convention. When languages agree on this interface, it ensures that both caller and callee with agree on the minutia of the calling convention, and how structures are laid out in memory. It is common for Rust, C and C++ to interface using the C ABI, and as such, there is syntactic support for selecting the ABI to be used per function and per structure. Reo-rs makes use of this feature to expose C-friendly types and functionality as part of its foreign function interface. In most cases, this requires nothing more than the addition of preprocessor annotations, ie. `#[repr(C)]` and `#[no_mangle]`, and visibility keywords, ie. `extern`.

Rust makes frequent use of generic type-parameters, which rely on the Rust compiler for *dispatch* at the call site (see Section 2.2.2). C has no analog for this feature, and thus, some structures and functions are be provided secondary *concrete* representations. As an example, Rust represents the `Port` structure with a generic argument, affixing its data-type at compile-time. This structure has a *foreign-function* analog type without the generic parameter. Consequently,

[2]These libraries are publically available on `crates.io`.

some of the safety guarantees of Reo-rs are unavailable when accessed via this interface; for such cases, functions are marked with the `unsafe` keyword according to the Rust idiom to communicate that the programmer takes on additional responsibility to invoke these functions only in the event they can guarantee that the relevant requirements are met.

Rust has no analog to C's header files[3]. To facilitate the sort of workflow that is idiomatic to C, whereby definitions and declarations can be distinguished such that *compilation* can be distinguished from *linkage*, the Rust ecosystem offers an addon for *generating* C header files from Rust source. This module can be loaded as the **bindgen**[4] addon to *cargo*, rust's package manager (comparables to *pip* for python, *npm* for node-js, and perhaps *maven* for Java). With this tool, we are able to generate C header files without much friction, and include them in any distributions such that downstream dependents on Reo-rs can incorporate them into their applications as they would do for any other C library. Once compiled and linked, their C or C++ applications would execute as would any other binary, and the use of Rust for compiling Reo-rs is no longer visible. Owing to its nature, calls that cross the Rust-C boundary do not induce any significant overhead at all [KN18].

4.3.2 Design Process

Many designs for the implementation of Reo-like coordinators are possible. Their structure and workings all depend on how information is arranged, and how multiple threads come together to coordinate on an a-priori unknown task without stepping on one another's toes. In our case, we concentrate on the case where all participants in the system share a memory address space, which opens up many means of exchanging data between threads. As is typical in multi-threading, the problem is not accessing the data, but rather restraining oneself from accessing the data at the wrong time. Before we can approach any design decisions, we examine what we know for certain: ports invoke `put` or `get`, each from their own thread. They cannot return immediately, as this would not result in the correct system behavior; when not aligned in time, getters will often (unknowingly) read uninitialized data, and putters will write their data, never to be read. They wish to exchange data in accordance with some defined protocol, but a priori have no knowledge of the protocol, nor their role in it. The aim is to facilitate rule firing 'greedily' as opportunity allows: ie. we do not wish to delay the firing of some rule x if some y can be fired sooner.

[3]Rust does have trait-associated functions, which can declare functions without defining them. However, they are of no use here, as they are not usable from C either, as they are inherently coupled with Rust's generic system.

[4]url: `https://crates.io/crates/bindgen`.

The Coordinator

The most obvious starting point is asking *who decides which rule to fire?* Reaching consensus prevents the system from reaching some mal-formed state where two rules are being committed to in tandem, violating the protocol or deadlocking on some resource they have in common. It is easy to contrive of such examples where numerous ports are involved. For example, consider a case where two rules disagree on which of two putter ports distribute their datum to a set of getters. If not done carefully, some getters may receive one value, and the rest another. The most approachable solution is to stick more closely to the Reo model by introducing a specialized *coordinator* for each protocol. Consensus is trivial when one participant is elected the leader for every circumstance. Unfortunately, we cannot rely on some port x being involved in every rule firing such that they are the coordinator. Many protocols do not have such an x that can be relied upon to be present. The Java back-end solves this problem by adding a fresh 'protocol' thread whose task is to *only* coordinate the others. This approach is easy to think about, as there is a clear mapping from threads to roles. However, the protocol thread is not *inherently* coupled to the actions of ports. It has to *wait* for opportunities to coordinate, necessitating the transmission of explicit events from compute-threads to the coordinator. These messages can use a channel, or use something like semaphores or monitors to send *signals* instead, and then relying on the coordinator to *re-discover* which ports are ready by reading the state of shared memory. Next, one must decide *who* organizes the actions into an interaction. The Java backend's approach is to spread ports out over space such that they can become ready concurrently[5]. The coordinator then treats the port structures like messaging pigeonholes, and performs the task of moving data around itself. The coordinator's notification to the ports is subtle; taking the form of `put` and `get` calls which release port-local locks, unblocking the compute-threads, completing the interaction. This solution is effective, but has some downsides, as discussed in Section 4.1.3.

Event Handling

A minor change with the potential for improvement is to remove the necessity of the protocol thread to *re-discover* the nature of the event which generated a wakeup signal. Rather than signals with no payload, we can use *events* which carry explicit information, eg: 'Port x is ready to get!'. With this approach, the coordinator waits in an *event loop*, handling incoming events. The Rust ecosystem has a number of libraries for defining event loops built atop system signal handles. An early implementation made use of the `mio` crate for sending events

[5]Conceptually this could be in parallel, but the actual implementation the exchange necessitates the use of a *class monitor lock* to prevent interfering with the protocol thread itself.

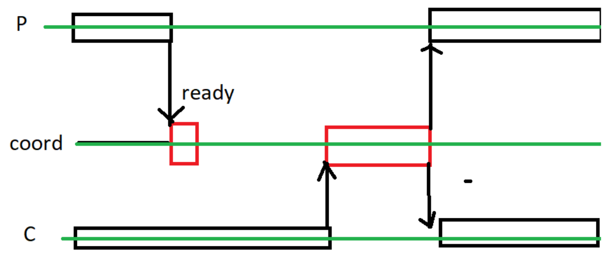


Figure 4.1: TODO. Figure is placeholder too.

which communicated *which* port has become ready. With this minor change, the coordinator does not need to inspect the contents of ports directly, which, owing to their modification by multiple threads, inherently cause several *cache misses* for the coordinator. Rather, the coordinator is able to manage a private, dense, *redundant* record of which ports are ready. Aside from the unfortunate data duplication, this optimization contributes greatly to the satisfaction of $\mathbf{R}^{\mathbf{N}}_{\mathtt{fast}}$. Unfortunately, regardless of how fast `mio` may be, the event must still cross the boundary between threads. In over-encumbered systems, this has the consequence of causing context switches.

Threadless Protocol

We observe that the use of a protocol thread results in a strange property for rules involving one or more ports: Always at least one thread is not doing any meaningful work. To understand why, consider an example protocol which simply forwards data from producer P to consumer C. Figure 4.1 shows a possible sequence of events. The coordinator cannot act until a rule can be fired. As soon as the rule is ready, all the involved ports are blocked. Until the coordinator has finished completing the interaction, the other threads wait, blocked until the coordinator releases them. Even the most efficient event-handling system cannot help but to make matters worse by adding delays in both directions.

One pivotal decision of our final design is to attempt to alleviate this problem. If compute-threads are going to block, waiting for progress anyway, why not have them do the coordination *themselves*? In our approach, we discard the dedicated protocol thread and re-interpret the coordinator as a *role* which the other threads take turns adopting. Conceptually, this change is a minor one; there is ultimately still at most one thread acting as coordinator. Until now, we have taken for granted that the coordinator can complete interactions with impunity; as they are the only elected leader of their kind, there is no concern for data races. In our new model, if *anyone* can be a coordinator, we must go our of our way to prevent two threads from adopting the role at once. Where before the bottleneck existed (implicitly) as a single coordinator thread processing a sequence of events *one at a time*, we now make the lock explicit: upon becoming ready, every thread attempts to acquire the *protocol lock*, the holder of which acts is the only coordinator for the duration.

Delegating the Task of Data Exchange

Owing to our focus on values at the systems level, we do not have the simplifying luxury of the Java backend to presume that moving data is cheap. In Rust, as in C and C++, values are not represented by indirect references by default; often, their shallowest representations are all there is to them. To satisfy $\mathbf{R}^{\mathbf{N}}_{\mathbf{data}}$, the Java backend's (admittedly intuitive) representation of ports as data pigeonholes is wasteful. For many realistic Reo protocols, data often moves *through* protocols synchronously, moving from `Putter` to `Getter` without any storage in memory cells in-between.

Our implementation introduces a new idea in an attempt to capitalize on this observation: getters fetch their data directly from the *source*. In this approach, the coordinator does necessarily handle data itself. Rather, it decides which rule to fire and *delegates* the task of moving the data to the getters themselves. In addition to skipping a redundant 'data hop' from putter to coordinator, this also facilitates the dissemination of a putter's datum to all its getters *in parallel*. This change requires extra messaging form the coordinator, as getters are given more responsibility. Where before a signal from coordinator to getter sufficed ('Your datum is ready!'), coordinators must now communicate the location of the getters' source ('Your datum can be found at P!').

4.3.3 Architecture

`Proto` is a type corresponding closely to its imperative-form specification type, `ProtoDef`. While they represent the same thing, their differences in structure and contents are a result of them being used for different *purposes*. Despite the increase in granularity from Reo's RBA-like form, `ProtoDef` still represents a *specification* of the protocol; as such, it strives to minimize redundancy to simplify parsing and minimize the surface for internal inconsistency. On the other hand, `Proto` is structured to facilitate *execution* at runtime.

1. **Layout optimized for speed.**
   As discussed in Section **??**, the `build` method is the only user-accessible means of constructing `Proto` instances. The methods of this type can rely on this to assume that their contents are internally consistent, thereby avoiding the cost of performing many checks at runtime. For example,

if a *rule guard* includes an equality check between the values in memory cells m_0 and m_1, `Proto` is able to assume that these cells have the same type; it is safe to use the result of m_0's definition of type-equality.

Additionally, concise structures can be re-arranged such that their layout facilitates less computation time. A general example of this paradigm is caching. Here, a clear example is the replacement with symbolic *names* for ports, functions and memory cells (represented by strings in the `ProtoDef` type), using *integers* for keying into vectors, maps and other such structures directly.

2. **Additional data for primitive concurrency.**
   Where `ProtoDef` can leave information implicit, `Proto` must be explicit. Interface ports require some data structures for storing concurrency primitives. For example, coordinators must send *control messages* to getters, as explained in Section 4.3.2 above.

Critical Region

Section 4.3.2 explains how threads initiating actions at boundary ports of a protocol assume the role of coordinator. It is fruitful to examine the fields of `Proto` in accordance to which *roles* access them, and how their access is safely controlled. The most coarse-grained distinction is that between fields inside and outside the protocol's lock-protected *critical region*. This divide is so fundamental, that is is immediately apparent by looking at the definition of `Proto` itself, seen in Listing 17. `ProtoCr` stores all of the fields manipulated only by the coordinator, such as the *allocator*, to which the coordinator delegates the task of storing persistent values. This is explained further in Section 4.3.4 to follow. Also observe the field responsible for managing which ports are *claimed*. While this is not a task traditionally associated with the coordinator, it's mutually-exclusive access between threads necessitates that this structure is protected by the lock.

Spaces

Clearly, `ProtoCr` can contain only the data that is not contended by multiple threads. Some structure is still needed for threads to *rendezvous* such that information can be exchanged and actions can be aligned in time. In the Java implementation, the class `Port` served two distinct purposes: (1) stored the value being exchanged by two threads, and (2) acted as the rendezvous for putter and getter. As explained in Section 4.3.2 above, the former of these tasks does not involve the coordinator in Reo-rs. However, the latter is still relevant. To meet this need, `ProtoR` associates *space* for every identifier (for ports and memory cells alike). The difference is name exists to distinguish them from ports, to which they are certainly related, but not identical. For every identifier, its `Space` contains precisely the data needed for it to communicate with its peers. For ports, this includes a `MsgBox`, which serves as a control-message channel from coordinator to compute-thread. Spaces are discussed further in Section 4.3.4 to follow.

```
pub struct Proto {
    cr: Mutex<ProtoCr>, // accessed by coordinator with lock
    r: ProtoR, // shared access
}
struct ProtoR {
    rules: Vec<Rule>,
    spaces: Vec<Space>,
    name_mapping: Map<Name, LocId>,
    port_info: HashMap<LocId, (IsPutter, TypeInfo)>,
}
struct ProtoCr {
    unclaimed: HashSet<LocId>,
    is_ready: BitSet,
    memcell_state: BitSet, // presence means mem is FULL
    allocator: Allocator,
    ref_counts: HashMap<Ptr, usize>,
}
```

Listing 17: Definitions of the most coarse-grained structures of a protocol instance. `Proto` is the entry-point, composed of `ProtoCr` in the critical section, accessed by only the coordinator, and `ProtoR` outside it, accessed by all.

4.3.4 Behavior

This section explains how the data structures of the `Proto` type comes to life at runtime to emerge as coordination according to the protocol with which it was configured.

Rule Interpreter

Unlike the Java implementation, Reo-rs moves the specification of the protocol type very late into the pipeline from Reo to the final application. Rather than relying on Reo to generate *native* application code, in this work we make more extensive use of the *virtualization* pattern. At runtime, the coordinator traverses rules in data form, performing tasks as a rudimentary *interpreter*. The tasks associated with such a `Rule` correspond closely to the conceptual interpretation of the imperative form; as such, they are treated like *transactions* at runtime.

TODO more detail and talk about temporary variables

Minimizing the Bottleneck

Reo-rs shares its centralized locking architecture with the Java back-end. Regardless of whether the coordinator and the thread that performs the role are decoupled, the importance of providing it mutual exclusion is clear; two coordinators in tandem would not be safe in the knowledge that the state does not change between evaluating the guards and changing the state. Methods exist for *fragmenting* protocols such that the locking becomes finer grained as protocols are into sets of smaller ones. As such, the Reo compiler internally produces a *protocol set* as its output, though the work on this feature is ongoing. Nevertheless, we consider this decomposition an orthogonal concern and consider it no farther. Reo-rs embraces this central lock, but takes measures to minimize the duration for which it is held. In this section we discuss these measures and how they work together to help satisfy $\mathbf{R}^{\mathbf{N}}_{\mathbf{fast}}$. To structure our reasoning, we identify the tasks a coordinator performs from the moment to acquires the lock (accepting its role), to the moment it releases it (relinquishing the role).

1. **Initialization**
   Section 4.3.1 explains how the time spent purely on *overhead* is diminished by avoiding the event-signal interaction used by the Java implementation, necessary to wake a sleeping protocol thread. Once the coordinator has acquired its lock (a task needed in both versions), transitioning into the work of the coordinator is nothing more than the time taken to invoke the `coordinate` function call.

2. **Checking Readiness and Memory State**
   Imperative form shares the explicit representation of the *synchronization set* with RBAs, encoding precisely which ports are involved with the firing. Clearly, a rule cannot fire until all ports involved are *ready*. Per port, this is a boolean property which can be represented by a single bit flag. Owing to the simplicity of this data, each of these sets can be represented as a single *bit-vector*, a data structure for which set-operations are exceptionally fast. Reo-rs takes this optimization a step further by extracting another boolean property per memory cell: *fullness*. The idiomatic encoding for memory cells storing data of type `T` in Rust would be the `Option<T>` type, such that `Option::None` represents *emptiness*. Instead, the relevant flags for fullness are extracted, separated from their

data and instead coalesced into another bit-vector. With just a handful of fast bit-wise operations, the coordinator is able to quickly detect whether a rule *cannot* fire, as a result of a port not being ready, or a memory cell being full when it should be empty, or empty when it should be full. In practice, the vast majority of cases where a rule's guard is unsatisfied are detected in this step.

3. **Instructions**
   Instructions are relatively expensive compared to the other steps in a rule's interpretation. Their cost scales with their complexity, as they can be defined as arbitrarily large and deep formula terms. Even individually, the cost of each operation can be high, as they include arbitrary user-defined function invocations, arbitrary user-defined equality checks, and allocation space for newly-created data objects. Section 4.3.4 explains how the cost of memory allocation is mitigated such that the allocation itself is amortized to constant time. For the rest of these operations, there is not much that can be done to avoid the cost; for the most part, they would be expensive even if each rule were performed by a native Rust function. Fortunately, the vast majority of rules for Reo connectors require no instructions at all. In practice, Reo connectors tend not to *inspect* the data whose flow they coordinate. The more intrusive the protocol's routing logic becomes, the more it begins to resemble *computation*, a task for which Reo should probably not be optimized. For the protocols without instructions (including *fifo1*, *alternator*, *sequencers*, *sync*, *lossy sync* and more), the support for instruction parsing costs no more than the time to determine that there are zero instructions to execute.

4. **Movements**
   Once a rule is committed, the role of the coordinator is to kick any getters into action, delegating the data exchange to them. Each *movement* encodes one *resource* (`Putter` or memory cell) being distributed amongst a set of *recipients* (each a `Getter` or memory cell). This meta-interaction is not synchronous; getters may take arbitrary time before waking up and actually participating in the data exchange. This is not the case for memory cells; as part of the *state* of the protocol, this is manipulated by the coordinator only. As such, operations which move values *into* memory (where memory cells act as getters) are performed first. Section 4.3.4 explains this procedure in more detail. Here, it suffices to say that the movement of memory between memory cells is fast.

   For port-getters, the coordinator does not move the value itself. Rather, the work is delegated to the compute-thread by sending a *control message* to the getter's `MsgBox`.

Usually, the coordinator does not have to interact with the resource (acting as putter) at all. It can rely on getters to 'clean up'. The coordinator returns, releasing the protocol lock. The only exception is for movements with *zero getters*. Such cases can represent a resource being destroyed. In these cases, there is no getter to perform the cleanup, and so, the coordinator does it itself. For a `Putter`, this is no more than sending a control message, releasing it. For memory cells, this may require running the *destruction* function associated with the memory cell's data type. Section 4.3.4 provides more detail on how these are managed.

Data Exchange

Eventually, each `Getter` waiting at their `MsgBox` receives a control message from the coordinator, revealing to them the identity of their *resource*. Their task is to locate the corresponding `Space` and contend with an unknown number of fellow getters to complete the *movement*. The correctness of this exchange relies on the satisfaction of a number of properties:

1. **One getter cleans up the resource**
   Regardless of whether the resource is a `Putter` or a memory cell, the set of getters are responsible for cleaning up the resource to finish the interaction. In the case of a putter, this takes the form of sending them a *control message*, notifying them that everyone has finished inspecting their datum and they may return to the caller. Clearly it is unsafe for anyone to release the putter *before* some getter has finished reading the datum; by returning, the putter may *invalidate* the memory region storing the datum.

   In the case of a memory cell resource, cleanup takes the form of re-setting its *ready flag* inside `ProtoCr`, signifying that the memory cell is in a stable state can again be involved in rule firings. This is necessary as there is no dedicated thread guaranteed to set this flag in future, as is the case for getters and putters. Section 4.3.4 to follow also explains how these memory cells are *emptied* in these events such that they can again store new values. This manipulates the protocol's state, potentially making new rules' guards satisfiable. As such, this last getter must once again acquire the protocol lock and attempt to *coordinate*.

2. **At least $\mathrm{N} - 1$ getters `clone`**
   Rust generalizes the operation for *replicating* a datum to produce another instance from it. It is idiomatic to rely on the standard trait `Clone` with single operation `clone` to implement this behavior. This approach covers cases for which there is a non-trivial means of replicating objects; some-

times, performing a bit-wise copy of the structure's shallowest representation is not enough. Consider the example of `Arc` ('atomic reference-counted') in Rust's standard library. This type consists of just a pointer to some heap-allocated tuple `(refcount, data)`, and is used for shared, reference-counted ownership of `data`. For this type, coping the pointer to the tuple is not sufficient. Cloning must *follow* the pointer and increment `refcount`.

3. **One getter *moves* instead of cloning**
   Data movements represent the *transmission* of data from source to a set of destinations. Generally, the value is no longer present at the source afterwards. Naïvely, the original must be *destroyed* to complete the interaction. However, it is wasteful and illogical to replicate an object only to destroy the original. Instead, we wish to *move* a value between threads, much as Rust's move semantics allow the movement of affine types between bindings. This cannot be done in the conventional way, as movement is defined is generally within the context of a single thread and scope. Regardless of Rust's expressiveness, it is nonetheless an *action-based* language, and does not offer the *interaction* we need.[6]

   When orchestrated correctly, we are able to implement a safe *move* operation between threads by invoking a pair of *unsafe* operations, one on either end. In *unsafe Rust* it is possible to *copy* a value without influencing the original. If not done correctly, this can easily lead to *double-frees*. On the other hand, it is possible to leak resource memory with `forget`, an operation of Rust's standard library which causes the compiler to consider the value *moved* without invoking `drop`. These pitfalls should be familiar to C programmers, as *unsafe Rust* gives one the capability to interact with *raw pointers* in a fashion similar to that of C. Together, these actions constitute the inter-thread move primitive we need.

   We elaborate our task by requiring an election between getters, such that one is designated the *mover*, and the rest are *cloners*.

4. **All clones must be complete before the move**
   It is unsafe to *move* a value before or while performing `clone` on the original. Essentially, every data exchange must proceed in two strict phases such that all clones occur in the first, and the move in the second. Consider again the example of type `Arc` by examining this sequence of events

[6]Rust is able to understand uni-directional movement of values into *new* threads using the same mechanism by which closures can *enclose* variables in their parent scope. More complex types are able to also create their own notions of safe 'movement' by composing actions as we suggest in this section. As in our case, they require the use of `unsafe`, as by definition the Rust compiler cannot reason about their correctness in the usual way.

that results in undefined behavior: (1) `Arc` x represented by a pointer to heap region at p is moved to binding y. (2) y goes out of scope, it's `refcount` is reduced to zero, and so its heap allocation is freed. (3) `Arc::clone` is invoked with x, which traverses its pointer to memory position p, and attempts to increment `refcount`. p is no longer allocated, and arbitrary memory corruption ensues. To prevent such cases, Reo-rs must take care to order all clones of some value before it can be moved, as the Rust compiler would do.

Many solutions are possible, but have in common that these getters must exchange some meta-information safely across thread boundaries. Our solution uses a pair of atomic variables for this purpose, *count* and *mover*, initialized by the coordinator a priori to N and *true* respectively. In a nutshell, *mover* is true if no getter has yet claimed the role of mover, which represents both (1) the responsibility to clean up, and (2) the privilege of moving the original value, rather than cloning it. Part of the procedure at large is a *pair* of *elections* between getters to determine a *mover* and a *last* getter. We elect a mover first. The time between the elections gives the losers (the 'cloners') the opportunity to clone, safe in the knowledge that the mover will not clean up until they are finished. If the mover is also elected last, they clean up and return immediately, as all clones must already be complete. Otherwise, the mover must await a signal from whomever is last before cleaning up.

This process is complete enough to implement the desired functionality for Reo-rs. However, we identify two important optimization opportunities which have the unfortunate consequence of complicating the data-exchange procedure further.

1. **Not all getters want data**
   Getters participating as a result of the `get_signal` operation will not return a value. Clearly these getters cannot avoid participating in the mover-election, as then nobody would clean up. These getters specialize their interactions by participating in the last-election first. The intuition is that if they lose this election, it is safe for them to return without participating in the mover-election; clearly this covers the case of no getters wanting the data. It is also safe to re-arrange these elections in this case; these getters have no intention to `clone`, and thus are not a threat to the invariant that required these elections to be ordered in the first place: all clones are complete by the time the last-getter is

2. **`Copy`-types can be replicated without `clone`**
   Section 2.2.2 explains how `Copy` marks types for which have a trivial destructor, and are safe for multiple getters to replicate by *copying* their

value bit-wise. This is the case for primitives, and structures composed entirely out of primitives, such as arrays of integers.

For copy-types, the mover and the *copiers* may copy the original datum in parallel. Afterward, only a last-getter is elected to clean up, safe in the knowledge that all copies are finished.

The full data-exchange procedure is spelled out in Rust-like pseudocode in Listing 4.3.4 in the Appendix.

Memory Cells

Section 4.3.3 explains that per *location* (generalizing ports and memory cells), Reo-rs maintains a persistent `Space` structure at a fixed location on the heap such that threads have a predetermined location to *rendezvous* on communication primitives. Section 4.3.4 follows up, explaining how these structures are also pivotal to data exchange. When getters converge on the space of a `Putter`, they rely on the presence of a prepared *data reference* in the space to the location of the putter's datum on its own stack. In this manner, values moving between ports are never moved to the heap at all. The memory-alignment of the putters datum generally differs per data exchange, necessitating that their space's reference be *updated* to the location of their value each time.

Memory cells differ from putters in that their value *persists* beyond the lifetime of any individual thread participating in the protocol; consequently, the data itself *must* be stored on the heap. A naïve implementation treats memory cells similarly to putters by continuously *updating* the data-reference in the associated `Space` such that it points to a freshly-allocated value on the heap every time the memory cell is filled.

We are able to rely on a property of Reo for an optimization: memory cells have pre-defined types. Instead of shifting the pointer around to a fresh allocation each time, we are able to *pre-allocate* the space needed to store one value per memory cell. In this model, the references do not change. Instead, each has a single allocation which is repeatedly re-used. Whenever the cell is empty, the contents of the allocated space are *uninitialized*. This can be done safely by relying on auxiliary structures for tracking when memory cells are empty; Section 4.3.4 explains how *bit vectors* serve this purpose for Reo-rs. This approach removes the cost of creating and allocating spaces at runtime. Unfortunately, this approach suffers a drawback inherited from its strict interpretation of *value-passing semantics*: moving data between memory cells is expensive. While small optimizations are possible for some circumstances (eg. we are able to swap references when the contents of two memory cells are *swapped*), they are only applicable in a handful of situations.

Requirements $\mathbf{R}^{\mathbf{N}}_{\mathbf{data}}$ and $\mathbf{R}^{\mathbf{N}}_{\mathbf{fast}}$ incentivize a more extensive optimization. Reo-rs intentionally decouples memory cells (including their spaces and their fullness flags) from *storage*, which describes where the contents of the cells is kept on the heap. We observe that Reo protocols perform logical replication of values often, while mutating existing values rarely. As such, many situations exist in which we are able to safely *alias* values between memory cells by relying on *reference counting*. We extend the idea of re-using allocations, but rather that fixing them per memory cell, we allow all memory cells of the same data type to draw from a shared pool of re-used allocations; this is often referred to as an *arena allocator*. The intricacies of this process are delegated by the coordinator to the `Allocator`, which tracks which *storage cells* of a type are available (free) and which are occupied. Rules which replicate, destroy or move data between memory cells thus can often moving data altogether, instead manipulating only the references within spaces, and reference counters of storage cells. For example, a rule which empties memory cell m_0 (destroying the contents) needs to only decrement the reference counter. Only when the counter reaches zero does the allocator need to be involved, invoking the value's destructor in-place and freeing the storage slot. This approach has another advantage: `clone` is invoked *lazily*, in some cases being avoided altogether. Consider a connector for which values originate from putters, get stored in memory slots, are replicated repeatedly, only to be destroyed before ever being emitted to a getter. In this example, `clone` is never necessary. This approach has an additional consequence; the data exchange operation explained in Section 4.3.4 may be initialized such that *nobody* is permitted to move. The procedure already given (in Listing 4.3.4) is able to handle this case.

Type Reflection

TODO refactor. PortDatum is no longer public-facing

Section 2.2.2 explains how Rust offers both *static* and *dynamic* dispatch for executing generic code, similar to how it is done for C++. These options offer a trade-off in runtime speed, binary size and flexibility. The interpreter approach taken by Reo-rs makes it difficult to make effective use of static dispatch, as this requires that the concrete type be known at the *call site*. While it is common for Rust libraries to be written in terms of generic arguments which the user will statically dispatch in their own code, this is not particularly ergonomic in our case. As their types are distinguished by context alone, it is impossible to store generic types together in typical collections (such as maps or arrays). This presents a problem for defining objects for generic Reo protocols, which vary also *number* and *arrangement* of data types.

Cases such as ours which require more extensive flexibility usually rely on dynamic dispatch instead. With this approach, generic type parameters behave less like static *macros*, and more like *interfaces* in Java, whose virtual functions are resolved at runtime. As Java interfaces define their virtual functions, Rust achieves dynamic dispatch by hiding some concrete type behind a particular *trait*, which provides the interface henceforth. Appropriately, the Rust term for these dynamic objects is *trait objects*, identifiable by the keyword `dyn`, for example, `Box<dyn Foo>`. In our case, we define trait `PortData` to encapsulate precisely the operations Reo needs: (1) a function for checking equality, (2) an accessor function for the concrete type's *layout*, which allows us to create new heap allocations, (3) the type's destructor, and (4) the associated `clone` operation. These type-specific operations are distinguished at runtime by traversing *virtual function tables* of the data types to resolve their specific functions. This lookup incurs some extra overhead at runtime, but we argue why this is acceptable for Reo-rs:

1. **Only pay for what you use**
   The most significant argument in favor of dynamic dispatch for our case is the observation that the only significant downside, the added overhead to type-specific operations, are seldom incurred. Much of the work performed by the coordinator is agnostic to the data-type. Section 4.3.4 explains how usually getters perform moves themselves, while coordinators interact with spaces, send messages and compare bit vectors. Furthermore, usually the argument in favor of reducing overhead is in the context of *hot path* computation; one wouldn't use dynamic dispatch to compute force vectors between every pair of atoms in a physics simulation, as these would dominate the computation. Protocols are low level but simply not *that* low.

2. **Reduced binary size**
   In some cases virtualization is able to *increase* performance indirectly by reducing the overall size and heterogeneity of the binary. When a program has enough different port types, static dispatch would (statically) cause a lot of code replication that interacts poorly with the *instruction cache*.

3. **Data type can change at runtime**
   Dynamic dispatch is a method of relegating the task of distinguishing type behavior to runtime by relying on *data* to distinguish types. Regardless of whether this is done by necessity, this opens up new opportunities for *changing* behavior at runtime. If you change the data, you change the behavior. Projects beyond the scope of this thesis may rely on dynamic dispatch to implement *dynamic reconfiguration*. In a sense, the interpreter approach taken by Reo-rs represents this idea taken to the ex-

treme: changing the contents of the `Proto` is able to influence more than just the *types* of protocol rules, but alter them altogether.

Rust's chosen representation of trait objects is the *fat pointer*, which is able to represent an indirect object by appending a *virtual function table* ('vtable'). These two pointers can be thought of as representing the data and behavior respectively. The necessity for the data itself being accessed via indirection is clear; objects implementing the trait may have different sizes in memory, but these sizes must be unified to create a common trait object with a known size. The second pointer is used by Rust 'under the hood' to access the concrete functions of the original object such that it's data is accessed by functions of the correct concrete type. These trait objects can be thought to carry their behavior around with them by moving with their vtables. While ergonomic in general, this is often redundant in the case of Reo, where values are guaranteed to only move between ports and memory cells of the same type anyway. In our case, vtable pointers would be repeatedly overwritten by the same vtable pointer, coupled to a new datum. Instead, we split these pointers, and store the vtables at fixed positions; one per memory cell, and some in the *allocator*. In this manner, we are able to use Rust's native dynamic dispatch, but move only *data* around as is done with static dispatch.

Our internal use of dynamic dispatch is not visible to the user. The API of putters and getters uses *static dispatch* instead, as it is most suitable for the context; port objects retain the same type throughout their lifetimes. We rely on a custom type, `TypeInfo`, to bridge the gap. This type serves a dual purpose: (1) it acts as a key, which can be checked for equality, and (2) it *contains* the vtable of the type it represents. Listing 18 shows the `TypeInfo::of` function, which is the user's only means of creating new `TypeInfo` instances. This function is used by the user themselves to *specify* the types of in the Rust-representation of a protocol's *imperative form*, `ProtoDef`. It is used once more inside the `claim` function for creating a new `Putter` and `Getter`, which guarantees that the new port object represents a logical port associated with a matching `TypeInfo`.

`TypeInfo::of` works by extracting an object's vtable from the text section of the generated binary, which Rust will guarantee is included by virtue of being used. To demonstrate how this works at runtime, we see the assembly generated for the `TypeInfo::of` function with an extremely simplified version of `PortDatum` consists of only function `get_num`, where `u32` implements the trait with `fn get_num(&self) -> usize{45}`. Listing 19 shows the result of compilation where `main` returns the `TypeInfo` for `u32`. Here, `.L__unnamed_1` shows the text region of the compiled binary containing a 24-byte-long vtable for `u32` when dynamically dispatched as implementors of trait `PortDatum`. In Rust, all vtables carry information about the type's *layout* (`u32` occupies 4 bytes, and must be 4-byte aligned in memory), as well as a pointer to its `drop` function for

```
1  struct TypeInfo(VtablePtr); // VtablePtr field is private (opaque to user).
2  impl TypeInfo {
3      fn of<T: PortDatum>() -> TypeInfo {
4          // 1. fabricate bogus (uninitialized) data pointer to some type T.
5          let x: Box<T> = unsafe { MaybeUninit::zeroed().assume_init() };
6          // 2. SAFE cast to trait object (Rust appends vtable pointer)
7          let fat_x = x as Box<dyn PortDatum>;
8          // 3. convert to "raw" trait object: a pair of pointers with UNSAFE cast.
9          let raw: TraitObject = unsafe { transmute(fat_x) };
10         // 4. discard the bogus data. Return wrapped vtable only
11         TypeInfo(raw.vtable)
12 }   }
```

Listing 18: 'Tricking' the Rust compiler into retrieving the vtable of a given type `T` for dynamic dispatch to virtual functions of trait `PortDatum`. The safe cast on line 7 inserts a pointer to a vtable which the compiler will ensure is present in the program text. `TypeInfo` structures can later be used for type reflection, by manually appending this pointer to reconstruct the *fat pointers* that Rust natively uses for dynamic dispatch.

deallocation (which for `u32` is trivial, simply returning). The remaining field points to the single function pointer associated with the trait, returning 45 as expected.

4.4 Requirements and Guidelines Evaluated

In this section, we give a summary of the means by which the requirements and guidelines of Section 4.2 are satisfied and adhered to respectively. This section serves as an overview of this chapter at large, motivating its points by referring to the relevant subsections above.

R_{value} Values passing through ports preserve value passing. This is achieved even in the presence of reference-passing optimizations 'under the table' by leaning on the same philosophy that Rust uses to prevent data races: prohibit mutable aliasing. Objects are only aliased (accessible via multiple bindings) if they should be identical. Section 4.3.1 explains how we prohibit aliasing into and out of the protocol by relying on value-passing port operations. On the other hand, Reo-rs aliases values, but only *until* they are mutated. Section 4.3.4 explains how memory values are safely aliased. TODO refer to the section that treats transform functions

R_{init} Section 4.3.1 explains how users are shielded from the granular initialization procedure by exposing an API with explicit constructor functions

```
core::ptr::real_drop_in_place:
        ret
example::main:
        lea     rax, [rip + .L__unnamed_1]
        ret
<u32 as example::PortDatum>::get_num:
        mov     eax, 45
        ret
.L__unnamed_1:
        .quad   core::ptr::real_drop_in_place
        .quad   4
        .quad   4
        .quad   <u32 as example::PortDatum>::get_num
```

Listing 19: The resulting assembly showing the vtable of type `u32` for dynamic dispatch of trait `PortDatum` (here simplified to only having a single function '`get_num`', returning an integer). Function `main` is exposed, creating and returning the `TypeInfo` of type `u32` using the function shown in Listing 18. Observe how this function simply returns `.L__unnamed_1`, the vtable of `u32`, included by the Rust compiler.

`build` and `claim` of protocols and ports respectively. Protocol objects are extensively customizable by the expressiveness of *imperative form*, with which `build` is parameterized. At the same time, these structures are kept internally-consistent by the preservation of invariants, and relying on `build` as the only user-facing means of instantiation.

R_{ffi} C and C++ foreign-function interfaces are provided by relying simply declarations with the C ABI where possible. As C cannot support Rust's notion of generics, where necessary, the `ffi` module provides generic-free alternatives for data types and functions where generics are represented as data instead. Reo-rs can thus be compiled once into either a statically- or dynamically-linked library for use in these other languages without any additional runtime overhead.

G_{data} Reo-rs facilitates the transmission of any fixed-size data-types by value. This permits but does require data to be heap-allocated. Sections 4.3.4 and 4.3.4 explains how Reo-rs has a value-passing API, but relies on reference-passing to minimize the number of times values are moved in memory.

G_{fast} Section 4.3.2 explains Reo-rs coordinates the actions of multiple threads while minimizing the overhead of inter-thread communications. Section 4.3.4 explains how meta-operations are represented such that they can be batched, allowing the coordinator to reduce the overhead of processing rules for firing.

$\mathbf{G}_{end}$ Section 4.3.2 explains how protocol objects are not given their own threads, trivially facilitating termination detection if no ports remain to interact with it. Section 4.3.1 describes how protocol structures are implicitly cleaned up once all of their ports are destroyed.

Chapter 5

Generating Static Governors

A protcol's *governor* acts to ensure that all the actions of a component are *adherent* to the protocol with which it interfaces, guaranteeing that its actions will not violate the protocol. In this section, we develop a means of embedding governors into Rust's affine type system. As a result, an application developer may ergonomically opt-into checking protocol adherence of their own compute-code using their local Rust compiler, whereafter successful compilation guarantees adherence to the protocol.

In more precise terms, let protocol A be *protocol adherent* to protocol B if and only if the *synchronous composition* of A and B is language-equivalent to B; equivalently, A is adherent to B and A adheres to B. This can be understood as A contributing no constraints to the composed system that B did not have already.

5.1 The Problem: Unintended Constraints

A central tenet of Reo's design is the *separation of concerns*, part of which is the desire to minimize the knowledge a compute component must have of its protocol. In this view, coordinating the movements of data is not a concern relevant to the task of computation. A desirable balance is possible with the observation that protocol objects are able to partially impose protocol adherence on their neighbors; External ports may instigate a `put` or `get` at any moment, and the *coordinator* will complete them as soon as the protocol definition allows it. In this way, coordinators possess a crucial subset of the features of *governers*: aligning the *timing* of two actions that compose an interaction. Unfortunately, in the properties of the realm of sequential, action-centric programming itself *implicitly* imposes constraints on the behavior of the system: `put` or `get` block until their interaction is completed, and no subsequent code (potentially, other port operations) will occur until they do. This is beyond the capabilities of the coordinator to influence.

In the context of application development, this has an interesting consequence; the behavior of the system is influenced by the behavior of (potentially) all of its components. This is sensible in theory, but becomes unwieldy in practice. Even small changes to the behavior of a compute component influences the system's behavior in unexpected ways, as we are not used to thinking about synchronous code as a composable protocol, nor are we able to intuit the *outcome* of the composition. For example, Listing 20 gives the definition of a compute function which a user may write to interact with a protocol. When `p` and `g` are connected to a *fifo1* protocol (which forwards `p` to `g`, buffering it asynchronously in-between), it runs forever and the output will be something like: `I saw true. I saw false. I saw true. I saw false. (...)`.
However, when connected with the *sync* protocol (which forwards `p` to `g` synchronously), the system has no behavior. The problem is that even though *fifo1* and *sync* have the same *interface*, `transform_not` is *compatible* (can be made to adhere) with the former and not the latter. By definition, *sync* fires when *both* `p` and `g` are ready, but `transform_rot` does not `put` until the `get` is completed. Once the intricacies of these programs grow beyond a programmer's ability to keep track of these relationships, the composed system may have *unintended* behavior. This property may be obvious at the small scale of this example, but it becomes more difficult the larger and more complex the program becomes. In the worst cases, an innocuous change makes an interaction becoming unreachable, manifesting at runtime as *deadlock*.

```
fn transform_not(p: Putter<bool>, g: Getter<bool>) {
    loop {
        let input: bool = g.get();
        print!("I saw {}.", input);
        p.put(! input);
}   }
```

Listing 20: A function in Rust which can be used as a compute component in a system, connected to a protocol component.

5.2 Governors Defined

In this work, we accept that it is necessary to write compute code that has blocking behavior. Rather than attempting to empower the coordinator with the ability to further influence its boundary components, we introduce explicit governors into our applications such that from the protocol's perspective, the components appear to manage themselves. A particular compute component requires a particular governor as the behavior permitted to the compute component is a function of its *interface* with the protocol.

Ultimately, all governors have in common that they enforce adherence to a given protocol on the components they govern. However, governors may differ on *when* and *how* this enforcement manifest. For example, a governor may intercept and filter network messages at runtime, while another checks for deviations *statically* and emitting compiler errors.

In this work, we leverage the unique expressiveness of the Rust language by creating a tool which generates protocol-specific governor code. When used by an application developer, these governors assess the protocol adherence of compute functions *statically*, and prevent compilation if deviations are detected. As such, these governor are absent from the compiled binary.

5.3 Solution: Static Governance with Types

TODO

5.4 Making it Functional

This section details the workings of the **Governor generator** tool which generates Rust code given (a) a representation of a protocol's RBA, and (b) the set of ports which comprise the interface of the compute component to be governed.

5.4.1 Encoding CA and RBA as Type-State Automata

The *type state* pattern described in Section 2.2.3 provides a means of encoding finite state machines as affine types. Their utility is in guaranteeing that all runtime traces of the resulting program correspond to runs in the automaton. For this class of machines, the encoding is very natural, as there can be a one-to-one correspondence between the states of the abstract automaton, and the types required to represent them. This is also the case for transitions and functions; in the worst case, this mapping is one-to-one also. For an arbitrary transition from states a to b with label x, a function can be declared to consume the type for a, return the type for b, and perform the work associated with x in its body.

The encoding is more complicated for CA, where not only states but data constraints must be encoded into types and must interact with transitions. One approach is to treat *configurations* as *states* were treated before by enumerating them into types. For example, the configuration of state q_0 with memory cell $m = 0$ is represented by type `q_0_0`, while state q_0 with $m = 1$ is represented by `q_0_1`. On a case-by-case basis, one might be able to represent several configurations using one type in the event these configurations are never *distinguished*. For example, a connector may involve positive integers, but only distinguish their values according to whether they are *odd* or *even* and nothing else; in this case {`q_0_0`, `q_0_1`, `q_0_2`, ...} may be collapsed to {`q_0_odd`, `q_0_even`}. For an arbitrary case unique types are needed for every combination of state with every value of every variable. As RBAs are instances of CA, we are able to represent them using the same procedure. As RBAs are used by both the Reo compiler and Reo-rs, they are the model of choice for governors also.

5.4.2 Rule Consensus

The protocol works by firing rules at runtime which correspond to those of the RBA which defines its Reo connector. Section 5.4.1 above explains how various compute-components are able to proceed in lockstep with the protocol's RBA in a type-state automaton of their own. For deterministic RBAs, this is easy enough; everyone can trivially know which action occurs *next*, and they can transition through configuration space independently, safe in the knowledge that their representations of the run will stay aligned. This ignores temporary mis-alignments in *time* for transitions in which the compute component does not communicate with the protocol; for these cases, one may work head, leaving the other in a previous state. However, they will catch up eventually when they both reach a transition that involves them communicating (which is ultimately all that matters). This process becomes more complicated when the protocol can reach configurations with *multiple* choices for the next transitions. Without a priori agreement on how these situations are handled, the choice is defined to be nondeterministic. Clearly, all is well as long as all parties agree on this choice; problems only present themselves when compute components and protocols diagree on what may happen next. If the view of a governor are out of sync with its protocol it is generally unable to guarantee that the actions it permits are adherent, or it may prohibit something it ought not to, resulting in unintended constraints of the intended behavior (in the worst case, deadlock). This is a problem of *consensus*.

Many means of creating consensus exist. We are able to enforce a *meta-protocol* a priori between governors and protocol such that consensus emerges at runtime. This can be achieved without any overhead by making the decision based only on information statically available. For example, peers may rely on a shared, total *priority* ordering on rules to remove all nondeterministic choice. Many such meta-protocols are possible, each making assumptions about the desired system behavior.

This work takes the approach of statically 'electing' the *coordinator* as the leader in every case, and having all governors follow the lead of its arbitrary choice by 'asking' it what to do next *dynamically*. This approach is primarily motivated by its *flexibility*; by supporting an arbitrary choice on the part of the protocol, we make the choice itself an *orthogonal* concern for future work. Electing the coordinator in particular as the leader is also somewhat natural; It is only actor in the system with a complete view of the protocol's state, allowing it to make the choice as a function of the state, ie. the protocol is capable of making the best-informed decisions.

In terms of implementation, we make a modification to the encoding of our governor's automaton such that it can represent all choices available from a particular configuration. Before proceeding, the governor 'collapses' these options to match the choice of the protocol by communicating with the coordinator. Concretely, the Rust function for a rule no longer returns a *particular* type-state token, but rather a `StateSet` which enumerates the options. This object is collapsed as a result of calling `determine`. Handling the returned variants branches the governor's control flow in a manner akin to a `match` (similar to a `switch` statement in other languages), with each arm given a single state token to proceed. Naïvely, this must be encoded as a distinct `enum` type with a variant for every possible outcome. Clearly creating new type definitions for every conceivable combination of branches is prohibitively expensive[1]. Ideally we wish to be able to create enumeration types on-the-fly with precisely the variants needed on a case-by-case basis. Rust provides *tuples* for this purpose in the case of `struct` (product types), it has no parallel for `enum` (sum types). This feature has been requested for some time [rh14]. If it is supported one day, this may be ideal fore representing these `StateSets` with minimal code generation.

Until anonymous sum types are supported, our solution to the problem of representing the generic `StateSet` type relies on Rust's traits to encode the variants as a *tail-recursive* list of nested tuples. Matching the elements of the lists is achieved by repeatedly attempting to match the *head*. Listing 21 shows one possible representation which uses a final *sentinel* list element to make for an ergonomic definition of the `StateSetMatch` trait, which provides distinct head-matching behavior for *singleton* lists from that of larger ones. From the user's perspective, `StateSet` objects are opaque, and prevent the automaton from proceeding with transitions until the object is collapsed to some usable `State` object.

[1]Early versions of our implementation indeed enumerated these types with a relatively effective *powerset* construction. However, it was unable to avoid explosion if there simply were many solutions to be found. The nail in the coffin was the changing from the exponential base from 2 to 3 as a result of the modification explained in Section 5.5.2.

```
trait StateSetMatch {
    type MatchResult;
    fn match_head(self) -> Self::MatchResult;
}
struct StateSet<L> { data: /* omitted */ }
impl<H,Ta,Tb> StateSetMatch for StateSet<(State<H>, (Ta, Tb))> {
// a list with 2+ elements. match_head definition omitted
    type MatchResult = Result(State<H>, StateSet<(Ta,Tb)>>;
}
impl<S> StateSetMatch for StateSet<(State<S>, ())> {
// singleton set. `()` acts as a sentinel element. match_head definition omitted
    type MatchResult = State<S>;
}
fn example(ab_set: StateSet<(State<A>, (State<B>, ()))>) {
    match ab_set.match_head() {
        Ok(a) => /* matched A */,
        Err(b_set) => {
            let b = b_set.match_head();
            /* matched B */
}   }   }
```

Listing 21: Definition of type `StateSet`, which acts as an anonymous sum type by encoding its variants as a tail-recursive tuple in its generic argument. Two non-overlapping definitions of trait `StateSetMatch` are provided to make the type behave as expected in response to associated method `match_head`. Function `example` demonstrates how the arbitrary number of variants are matched two at a time by repeatedly attempting to match the first element of the list (the head), translating it into a conventional `Result` enum which Rust can pattern-match as usual. The result of this match can depend on the contents of field `data`, which is instantiated dynamically at runtime by interacting with the coordinator.

5.4.3 Governed Environment

TODO

5.5 Making it Practical

With a basic outline for the implementation, we are able to realize some functional, yet naïve governors. However, there is a long way to go before these systems can be applied in any realistic scenario. In this section, we explain which problems remain to be solved, whether for the sake of managing complexity, or for the user's ergonomics.

5.5.1 Approximating the RBA

The approach to generating a type-state automaton from an RBA was given in 5.4.1. Our type-state automaton suffer the same state-space explosion of Constraint Automata, prior to the inclusion of memory (explained in Section 2.1.3). We cannot hope to represent realistic programs with this approach alone, as the type-state automata would be wildly unmanageable in its number of states and transitions. In this section, we explain how the type-state automaton is adapted to *approximate* the protocol's configuration space such that we strike a balance between accuracy and simplicity, without any effect on the governor's correctness.

Data Domain Collapse

We abandon the goal of faithfully representing the entirety of the protocol's configuration space in favor of representing an approximation by assuming all data types to be the trivial *unit-type*. With this assumption, memory cells may be in one of two states: (a) empty, (b) filled with 'unit'. Converting existing RBAs may see large sub-expressions of *data constraints* becoming constant, including checks for equality and inequality between port values. This assumption is justified by its relation to Assumption **??** from Section **??**. In this context, it can be understood to mean that *usually*, two configurations that are only distinguished by having different *data values* in memory cells or begin put by putters satisfy precisely the same subset of the RBA's guards. Consequently, they do not need to be distinguished. This simplification greatly reduces the total number of types to encode an RBA's configuration space. However, it is still necessary to consider the possible *combinations* of all empty and full memory cells, requiring potentially 2^N types for N cells. Rather than enumerating these types explicitly, we can rely on the *structure* the RBA provides by simply encoding each automaton configuration as a *tuple* of types `Empty` and `Full`. In a sense, each tuple is indeed its own type, but neither the code generator nor the compiler need to pay the price of enumerating all the combinations eagerly. For example, a configuration of three empty memory cells would be represented by type `(Empty,Empty,Empty)`. For brevity, we will henceforth abbreviate these tuples by omitting commas, and shortening `Empty` and `Full` to `E` and `F` respectively.

As before, we are able to represent an RBA rule as a *function* in the Rust language by encoding a configuration change from q to p determines its *declaration* such that it consumes the type-state of p and returns the type-state of q. The naïve approach of generating functions per type-state is susceptible to the same *exponential explosion* that plagued CAs in the first place. Fortunately, tuple-types have inherent structure which Rust's generic type constraints are able to understand. The use of generics to *ignore* elements of the tuple coincides with an RBA's ability not *ignore* memory values. Consequently only one function definition per RBA rule is required. The way the rule's data constraint manifests is somewhat different, as our function must *explicitly* separate the *guard* and *assignment* parts and represent them as constraints on the parameter-type and return-type respectively. As an example, Listing 22 demonstrates the type definitions and rule functions for the *fifo2* protocol first seen in Section 2.1.3 with the associated RBA shown in Figure 2.5. Observe that the concrete choices for tuple elements act as *value checks* for memory cells in either empty or full states. Omission of a check must be done explicitly using a *type parameter* such that the function is applicable for either case of `Empty` or `Full`, and to ensure the *new* state preserves that tuple element; this causes memory cells to have the expected behavior of *propagating* their values into the future unless otherwise overwritten by assignments. This serves as an example of a case where our simplification coincides with a faithful encoding of the original protocol as *fifo2* never discriminates elements of the data domains of A and B.

```rust
enum E {} // E for "Empty"
enum F {} // F for "Full"
fn start_state() -> (E,E);

fn a_to_m0<M>(state: (E,M)) -> (F,M);
fn m0_to_m1  (state: (F,E)) -> (E,F);
fn m1_to_b<M>(state: (M,F)) -> (M,E);
```

Listing 22: Type-state automaton for the *fifo2* protocol in Rust. The three latter functions correspond to the three rules seen for the RBA in Listing 2.5. Function bodies are omitted for brevity. Note that `M` is not a type, but rather a generic *type parameter* to be instantiated at the call site.

RBA Projection

When a protocol's interface is provided as-is to a compute component, its model itself (an RBA in our case) defines precisely what it is permitted to do, just with the *direction* of operations reversed; for the component to be compatible, it must put on port P whenever the protocol gets on P, and get on port Q whenever the protocol puts on port Q. In such cases, the procedure for encoding the RBA described in Section 5.5.1 can be applied directly. Otherwise, the interface of a compute component does not subsume the entirety of the interface of its protocol. In such systems, the protocol interfaces with

rule	guard	assignment
0	$m_0 = *$	$\wedge\, m_0' = d_A$
1	$m_0 \neq * \wedge m_1 = *$	$\wedge\, m_1' = m_0 \wedge m_0' = *$
2	$m_1 \neq *$	$\wedge\, d_B = m_1 \wedge m_1' = *$

Table 5.1: RBF of the *fifo2* protocol, equivalent to the RBA in Figure 2.5. Formatted with an outermost disjunct per line such that guard and assignment parts per rule are discernible.

several compute components. Indeed such cases form the majority in practice; compute components tend to only play a small role in a larger system.

The contents of Section 5.5.1 are sufficient to generate some functional governors. We consider a system containing protocol P and connected compute component C with interfaces (port sets) I_P and I_C respectively such that $I_P \supseteq I_C$. We wish to generate governor G_C whose task is to ensure that C adheres to P. As a first attempt, we translate P's RBA to Rust functions and types as-is. We would quickly notice that the RBA's data constraints represent port-operations that are excluded from I_C. These interactions involve no actions on C's part; from the perspectives of C and G_C, these actions are *silent*. Equivalently, we do not use the RBA of P directly, but consider instead its *projection* onto I_C, which *hides* all actions that are not in the interface projected upon.

```
fn a_to_m0<M>(state: (E,M)) -> (F,M) {
  // A puts
}
fn m0_to_m1  (state: (F,E)) -> (E,F) {
  // silent
}
fn m1_to_b<M>(state: (M,F)) -> (M,E) {
  // silent
}
```

Listing 23: Type-state automaton rules which govern the behavior of a compute component with interface ports $\{A\}$ for the *fifo2* protocol. Function bodies list the *actions* which the component contributes to the system. Observe that rules but 0 are silent.

As an example, we once again generate a governor for a compute-component with interface $\{A\}$ with the *fifo2* protocol. This time the protocol is represented as an RBF in Table 5.1 to make the correspondence to the generated governor in Figure 23 more apparent. Observe that all but one of its rule functions are *silent*, serving no purpose but to advance the state of the automaton by consuming one type-state and producing the next. As demonstrated here, this approach to generating governors is correct, but has two undesirable properties:

1. **API-clutter**
   The end-user is obliged to invoke functions which correspond with rules in the protocol's RBA. In many cases, these rules will serve no purpose other than to consume a type-state parameter, and return its successor.

2. **Protocol Entanglement**
   The type-state automaton captures the structure and rules of the protocol's RBA in great detail. This is a failure to *separate concerns*, which further couples the compute component to its protocol. This has the immediate effect of making components difficult to re-use (their implementations are more protocol-specific), as well as making them brittle to *changes* to the protocol, making them difficult to maintain.

RBA Normalization

Section 5.5.1 introduced a procedure for generating governors, but also discussed a significant weakness; all governors are represented by type-state automata based on the original protocol's rules. In this section, we introduce a notion of *normalization* that intends to *specialize* the governors according to its needs such that it is still 'compatible' with the protocol's RBA in all ways that matter, but has greatly reduced *api-clutter* and *protocol-entanglement*.

Let an RBA be in normal form if it has no silent rules. We observe that the presence of silent rules contributes to both api-clutter and protocol entanglement. Ideally, we wish to abstract away the workings of the protocol as much as possible; at all times, the governor only needs to know which actions the component must perform *next*. To make this notion more concrete, we introduce some definitions which build on one another to define the term we need: our normalization procedure should generate an RBA with starting configuration which *port-simulates* the protocol's RBA in its starting configuration:

- $\mathsf{Act}(r)$ of an RBA state r:
  The set of ports in r which perform actions (ie: are involved in interactions).

- *Rule sequence* from c_0 to c_1 of RBA R:
  Any sequence of rules in R that can be applied sequentially, starting from configuration c_0 and ending in configuration c_1.

- P-*final* wrt. port set I:
  A rule sequence of RBA R, with *last* rule r_{last} is P-final with respect

to port set I if $\mathrm{Act}(r_{last}) \cap I = \{P\}$ and for all rules r in the sequence, $r = r_{last} \vee \mathrm{Act}(r) \cap I = \varnothing$.

- RBA R_1 in config. c_1 *port-simulates* R_2 in config. c_2 wrt. Interface I:
  If for every P-final rule sequence of R_2 starting in c_2, ending in c_2' there exists some P-final rule sequence of R_1 starting in c_1, ending in c_1' such that R_1 in c_1' port-simulates R_2 in c_2'.

The intuition here is that it does not matter how the governor's RBA structures its rules. It is unnecessary for governors to advance in lockstep with the protocol to the extent that they agree on the protocol's *configuration* at all times. It suffices if the protocol and governor always agree on which *actions* the ports in their shared interface do next. Figure 5.1 visualizes this idea; observe how the normalized RBA has entirely different transitions (different labels and configurations), but is ultimately able to pair actions of the protocol for ports in its interface with its own local actions.

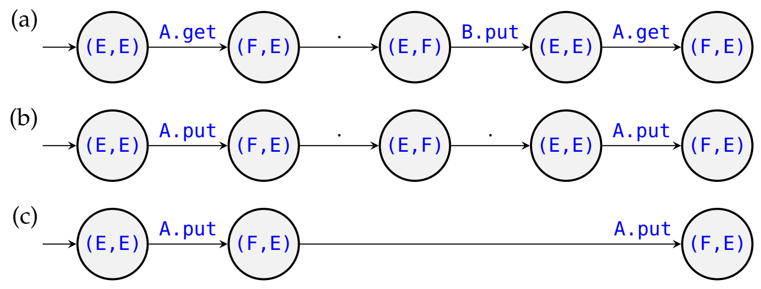


Figure 5.1: Rules being applied to walk three RBAs in lockstep, with time horizontally, showing the (simplified) configurations traversed, and annotating rules by showing which port actions they involve.
(a) RBA of protocol *fifo2*. (b) RBA of *fifo2* projected onto port set $\{A\}$. (c) RBA of *fifo2* projected onto port set $\{A\}$ and normalized to remove silent rules.

The final normalization procedure is given in Listing 24 in the form of simplified Rust code. It works intuitively for the most part: silent rules are removed, and new rules are added to retain their contribution of moving the RBA through configuration space. The function `normalize` ensures that the returned rule set is in the same configuration as the protocol after matching a non-silent, but the configuration is allowed to 'lag behind' while the protocol performs rules which it considers to be silent. New rules must be added to 'catch up' to the protocol after any such sequence of silent rules. The procedure does this by building these *composed* rules from front to back, ie. replacing every silent rule x with a *set* of rules $x \cdot y$, where y is any other rule. Once

```
fn normalize(mut rules: Set<Rule>) -> Set<Rule> {
    let (mut silents, mut not_silents) = rules.partition_by(Rule::is_silent);
    while silents.not_empty() {
        let removing: Rule = silents.remove();
        if removing.changes_configuration() {
            for r in silents.iter() {
                if let Some(c) = removing.try_compose_with(r) {
                    silents.insert(c);
            }   }
            for r in not_silents.iter() {
                if let Some(c) = removing.try_compose_with(r) {
                    not_silents.insert(c);
    }   }   }   }
    return not_silents;
}
```

Listing 24: Normalization procedure, expressed in (simplified) Rust code. In a nutshell: while one exists, an arbitrary silent rule x is removed, and the list of rules is extended with composed rules $x \cdot y$ such that y is another rule.

completed, the RBA may contain rules that are subject to *simplification*. For example, $\{m = * \wedge n = *,\ m \neq * \wedge n = *\}$ can be represented by only $n = *$.

The normalization algorithm is **correct** as clearly it does not have silent rules once it returns (`not_silent` containing zero silent rules is invariant). Observe that for each silent rule removed, it does not consider composing with *itself*. The immediate result is that the algorithm never inserts some rule $x \cdot x$ for silent rule x. This is not a problem, as all *silent* rules of our approximated RBAs are *idempotent* with respect to their impact on the configuration. The algorithm is able to take for granted that the result any *chain* of silent rules $x \cdot x \cdot x \cdot ...$ is covered by considering x itself. Furthermore, the incremental removal of rules prohibits the creation of any silent cycles at all. This is due to the reasoning above being extended to any sequences also. (TODO PUMPING LEMMA).

The normalization algorithm is **terminating**. It consists of finitely many *algorithm steps* in which the RBA A is replaced by RBA $B = (A \setminus \{r\}) \cup \{r \cdot x | r \in A \setminus \{x\} \wedge composable(x, r)\}$ for some silent rule $x \in A$. Initially, A is the input RBA with silent rules. The algorithm terminates, returning B when A is replaced by B where B has no silent rules. Let $P(x)$ be the set of *acyclic paths* through RBA x's configuration space. Observe that initially, $P(A)$ is finite. It suffices to show that in each algorithm round, $|P(A)|$ strictly decreases. Within a round, for every 'added' p in $P(B) \setminus P(A)$, p contains a rule $m \cdot n$ such that there exists p' in $P(A) \setminus P(B)$ identical to p but with a 2-long sequence of rules m, n in the place of x. From this we know that $|P(A)| \geqslant |P(B)|$. However, the

rule	guard	assignment
0	$m_0 = *$	$\wedge\ m_0' = d_A$
2	$m_1 \neq *$	$\wedge\ d_B = m_1\ \wedge\ m_1' = *$
$1 \cdot 0$	$m_0 \neq * \wedge m_1 = *$	$\wedge\ m_0' = d_A \wedge m_1' = m_0$
$1 \cdot 2$	$m_0 \neq * \wedge m_1 = *$	$\wedge\ m_0' = *$

Table 5.2: RBF of the *fifo2* protocol, projected onto port set $\{A, B\}$ and normalized. Rules 0 and 2 are retained from Table 5.1, and new rules $1 \cdot 0$ and $1 \cdot 2$ are composed of rules from the original RBF.

1-long path of x itself is clearly in $P(A) \setminus P(B)$. Thus, $|P(A)| > |P(B)|$. QED.

To demonstrate the normalization procedure, Table 5.2 shows the result of projecting the *fifo2* connector's RBF onto port set $\{A, B\}$ and normalizing. The two additional rules can be understood to 'cover' the behavior lost as a result of omitting the silent rule 1 from the original RBF.

5.5.2 User-Defined Protocol Simplification

Recall, the purpose of a governor is to preserve a system's *liveness*. They do this by ensuring that their governed compute component performs port operations that allow the interfacing protocol (and the system around it) to progress. Governors do this by enforcing that their compute component's implementation *covers* each possible transition with code that performs the required task, and ensuring it is chosen correctly in accordance with the wishes of the protocol. Section 5.4.2 explains how our type-state automaton represents this by each configuration requiring the definition of a *set* of transitions, one for each action. We say the implementation 'covers' each of these cases by defining the component's behavior in each cases, including the invocation of the relevant port operation.

An overzealous governor which requires to cover *additional* (unnecessary) cases would still serve its purpose. In effect, such a governor would enforce adherence to some *other*, more permissive protocol. However, liberty of the protocol means responsibility to the compute-component: the more the protocol *might do*, the more the compute-component *must consider doing*. There is incentive for governors to do this: permissive protocols are simpler to enforce.

This conservatism becomes a problem when it infringes on the component's ability to express its behavior as intended. Consider the example of a compute-component X that forwards values from its input port A to its output port B. Perhaps this component is used in a pipeline as intended such that the component is involved in an endlessly-alternating sequence, represented by regular expression $(AB)^*$. Perhaps there is a sensible way for X to implement the more

rule	guard	assignment
0	$m_0 = *$	$\wedge\ m_0' = d_A$
1	$m_0 \neq * \wedge m_1 = *$	$\wedge\ m_1' = d_A \wedge m_0' = *$
2	$m_0 = m_1 \neq * \wedge m_2 = *$	$\wedge\ m_2' = d_A \wedge m_0' = m_1' = *$
3	$m_0 = m_1 = m_2 \neq *$	$\wedge\ d_B = m_0 \wedge m_0' = m_1' = m_2' = *$

Table 5.3: RBF of the *a7b1* connector, which is characterized by cycling through a predictable sequence of period 8, where A inputs seven times and B outputs once. It works by encoding its configuration in an 8-long cycle as a three-bit integer using the fullness of memory cells m_{0-2}.

permissive protocol which permits B firings to be omitted, expressed $(A(B|\lambda))^*$. P has no problem discarding values input from A. However, if the governor takes it a step further such that 'anything goes' (expressed $(A|B)^*$), X cannot meaningfully represent its work. How on earth can it forward a message to B before receiving it on A? Not even clairvoyance can help; what if A never fires at all? This is how the user would experience the problem of a governor infringing on the component's own behavior; in a sense, P has a protocol of its own which must be preserved on its interface ports which the governor violates.

Nevertheless, there is value in providing a compute-component with a simplified (permissive) view of the protocol where possible. As a motivating example, consider the *a7b1* connector, given as RBF in Table 5.3. This connector uses the fullness of three memory cells to count in binary from zero to seven (using the binary alphabet of memory cell states {`E`, `F`}), and cycling back again to zero. Configurations in this cycle are distinguished by specifying different behaviors on A and B. Here, the projection and normalization of the protocol's RBF is trivial, as no rules are silent. Without the ability to simplify, the Y must be implemented such that it corresponds exactly with the protocol's (predictable) walk through its approximated configuration space, given in Figure 5.2. As all states are distinguishable, so too are their corresponding *state types* distinct. Now consider this protocol interfacing with some compute-component Y, which is always ready to consume and emit some data element *Q*. Without simplification, the resulting governor would require that the traversal through configuration space be spelled out; the user would be forced to distinguish these states, even though Y has no need for this specificity. Most likely, the resulting implementation will be repetitive and verbose, if the behavior is the same for configurations `(EEE)`, `(EEF)`, et cetera.

Our solution to this problem is to introduce a third type for representing the state of a memory cell which may be *either* full or empty: `Unknown` (abbreviated as `U`). Rather than corresponding to a specific configuration of the

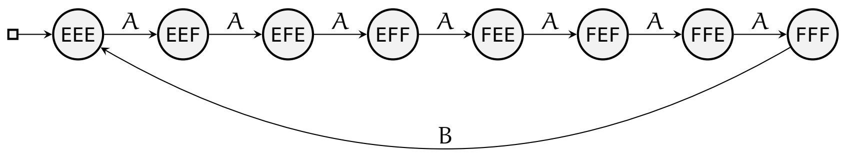


Figure 5.2: Rules transitioning through configuration space of approximated RBAs for *a7b1* connector, with states named after the 'count' the three memory cells represent in base 2 (in binary alphabet {`E`, `F`}). Here, the normalization procedure with interface set {A, B} is trivial as no transitions are silent.

(approximated) RBA, the governor now reasons about the *set* of states which the protocol may be in. For example, type `(UUE)` encapsulates all the concrete configuration types {`(EEE)`, `(EFE)`, `(FEE)`, `(FFE)`}, and is liable to covering the *union* of the rules applicable to any of those states. In this manner, it is safe for the programmer to arbitrarily 'forget' the state of a memory cell, replacing its element in the tuple type with `U`. To be clear, `U` is not special as far as Rust is concerned; we have changed to a ternary alphabet for representing memory cells in types. However, `U` does not correspond to any real configuration that memory cells are ever 'really' in at runtime; they are always either empty or full. `U` is a stand-in for an empty *or* full memory variable, an abstraction in which the protocol is not (explicitly) involved. With this tool in their belt, the implementation of the compute component is able to arbitrarily *unify* the state-types of multiple branches. Our example component Y above is able to implement its behavior to the satisfaction of its governor with transitions through configuration space in Figure 5.3. This weakening can be communicated quite ergonomically, resulting in something very close to what the user would implement themselves: a single loop where the four rules (numbered 0-3) may be applied to configuration type `(UUU)`, each resulting again in `(UUU)`.

5.5.3 Match Syntax Sugar

Section 5.4.2 explains how the set of transitions to be covered by a configuration type can be represented in Rust's type system as a tail-recursive list. This alleviates the problem of having to explicitly enumerate the needed sets each as their own enumeration type. This is necessary, as the upper-bound[2] of state

[2]Many factors reduce this number drastically in practice. For example, state-sets are usually not large because they are only ever encountered when reached by *transitions* from some state.

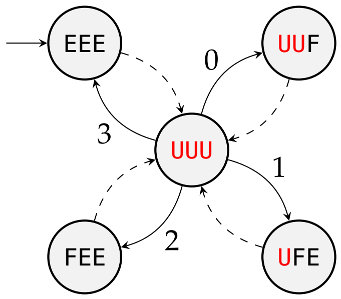


Figure 5.3: Rules transitioning between configurations of the *a7b1* connector shown in Figure 5.2. Here, the user employs *weakening* to convert (dashed arrows) state tokens to configurations to the configuration-set '???' containing *all* concrete configurations. RBA rules firing are shown with solid arrows, annotated with the rule name, corresponding to those given in Table 5.3.

sets is 2^{3^M}, where M is the number of memory cells[3]; suffice it to say, it is *a lot*. Unfortunately, these are not natively-supported enumeration types, and thus cannot be *matched* as is idiomatic in the Rust language. However, Rust has extensive support for abstract syntax tree *macros*, allowing us to have the best of both worlds; the user interacts with `StateSet` types by using a match-like macro which enumerates the branches, but there is no need for concrete `enum` classes to be defined for all the conceivable combinations. Figure 25 gives an example of how these cases compare to one another.

[3]The number of unique state sets is 2^S, where S is the number of configurations (automaton state types). This, in turn is 3^M, as each memory cell's state is represented by a type in {`E`, `F`, `U`}.

```
enum StateSetXyz { X(X), Y(Z), Z(Z) }
fn match_standard(set_xyz: StateSetXyz) {
    use StateSetXyz::{X, Y, Z};
    match set_xyz {
        X(x) => x.foo(),
        Y(y) => y.bar(),
        Z(z) => z.baz(),
}   }
fn match_recursive(set_xyz: StateSet<(X, (Y, (Z, ())))>) {
    use StateList::{Head, Tail};
    match set_xyz.match_head() {
        Head(x) => x.foo(),
        Tail(set_yz) => match set_yz.match_head() {
            Head(y) => y.bar(),
            Tail(z) => z.baz(),
}   }   }
fn match_macro(set_xyz: StateSet<(X, (Y, (Z, ())))>) {
    match_set! { set_xyz;
        x => x.foo(),
        y => y.bar(),
        z => z.baz(),
}   }
```

Listing 25: Example of three methods for matching a *state set*, representing a sum type of three variant types simplified here to `X`, `Y` and `Z`. First, `match_standard` shows how this is done in idiomatic Rust, requiring an enum type `StateSetXyz` be explicitly defined. `match_recursive` shows how the same state set represented by a tail-recursive `StateSet` type can be similarly matched by exhaustively 'unzipping' head elements using a function `match_head`. finally, `match_macro` functions identically to the second case, but relies on a sugaring macro `match_set` to mimic the syntax of Rust's `match` statement, seen in the first function.

Chapter 6

Benchmarking

In this section we evaluate the performance of the Reo-rs library, focusing primarily on the optimizations described in Section 4.3. TODO add references to requirements and guidelines from Reo-rs chapter

6.1 Experimental Setup

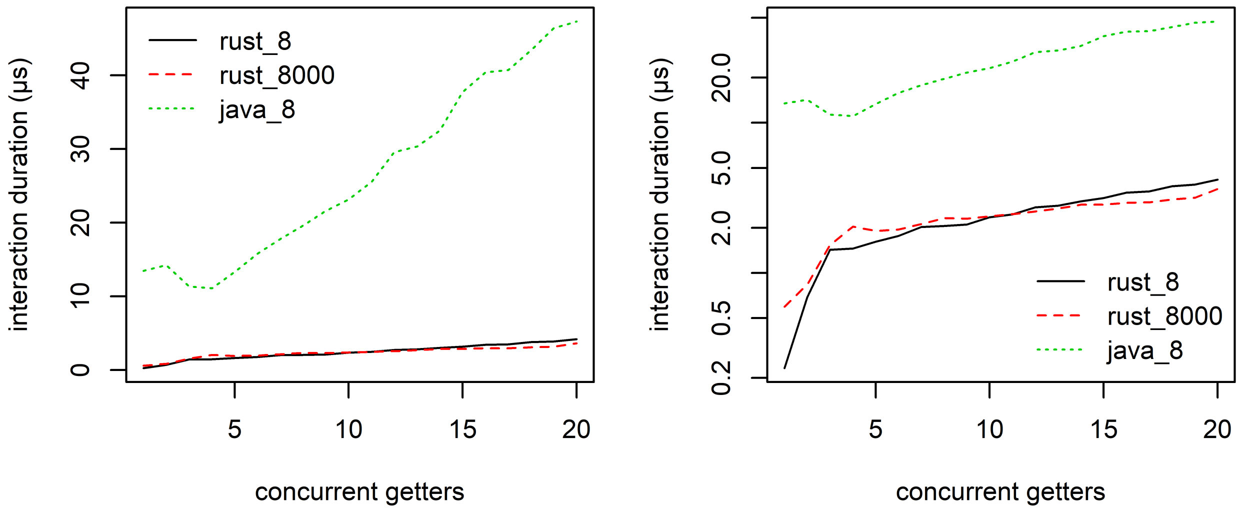
TODO

6.2 Reo-rs in Context

This section compares the performance of Reo-rs to its various competitors: (1) existing Reo back-end for the Java language (2) Hand-crafted Rust protocol code. The goal is to provide the reader with an understanding on the strengths and weaknesses of Reo-rs in a broader context.

6.2.1 Versus the Java Implementation

We begin by making the most intuitive benchmark to get an understanding of how effectively Reo-rs has been optimized for its task; we compare it to the work of the Reo compiler's Java code-generator. This comparison spans two vastly different systems with different goals, but also compare a memory-managed language to a system's language. The reader should bear this in mind when interpreting the measurements. As our test scenario, we have a set of N getters repeatedly copying some memory value M, retained inside the protocol from initialization. By involving a contended resource, we are able to test and



(a) (b)

Figure 6.1: Comparison of interaction time for the *fetch* connector for both Java and Rust backends moving small-size values. 'rust_8' and 'java_8' both move a payload of 8 bytes (to match the reference size of the 64-bit JVM). 'rust_8000' gives an example of how the runtime of Reo-rs can change with respect to modest changes in data size. The two sub-figures mirror one another except for the linear and logarithmic y-axes respectively.

compare the *scalability* the generated programs, both in terms of number of ports and the size of the transmitted data.

The unfairness of our comparison cuts both ways, as there is not a clear means of comparing the transmission of large values; the Java version relies entirely on object-aliasing, effectively implementing different semantics. For Java, the size of values transmitted is largely irrelevant. We begin by a comparison on the only common ground; Figure 6.1a shows both Java and Rust are transmitting pointer-sized objects. Aside from the order of magnitude difference in runtime, we observe a different *shape*. Reo-rs is observed to be significantly faster for a single-getter case. This is easy enough to explain; the type relied upon for protecting the coordinator's *critical region* is `Mutex` from the `parking_lot` crate, which provides implementations of these kinds of concurrency primitives. `Mutex` is documented as having a *fast path* optimization for when the lock is acquired *uncontested*. Runs with one getter are thus able to take advantage of this optimization every time.

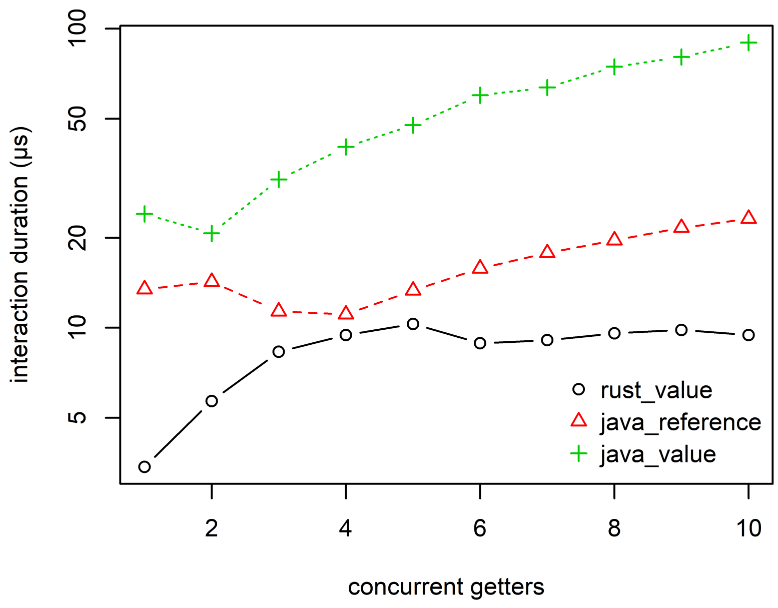


Figure 6.2: Comparison of interaction time for the *fetch* connector for both Java and Rust backends moving 64-kilobyte-sized data. The Rust backend moves the datum by value, while the Java parallel 'java_reference' aliases the object (moving by reference). This backend does not support semantics-preserving value-passing. We achieve safety here by the coordinator injecting `clone` operations of byte arrays to mirror Rust's bit-wise copy, called 'java_value'. Note the logarithmic y-axis.

Figure 6.2 attempts to draw the same comparison as before, but in the case of large data-types. The Java-generated protocol objects do not do true value-passing. It is out of the scope of this project to attempt to implement this

6.2.2 Versus Hand-Crafted Programs

Clearly, Reo-rs cannot out-perform hand-optimized Rust on a case-by-case basis; whatever Reo-rs does, the hand-optimized code can mimic and specialize to surpass its performance. The utility of the library is to handle arbitrary Reo-generated protocol descriptions, hiding the details away from the user with an API that strikes a balance between flexibility, safety and performance. Here, we attempt to gauge the performance gap between Reo-rs for some small examples of connectors.

Firstly, we examine a case for which Reo-rs is expected to do poorly: the

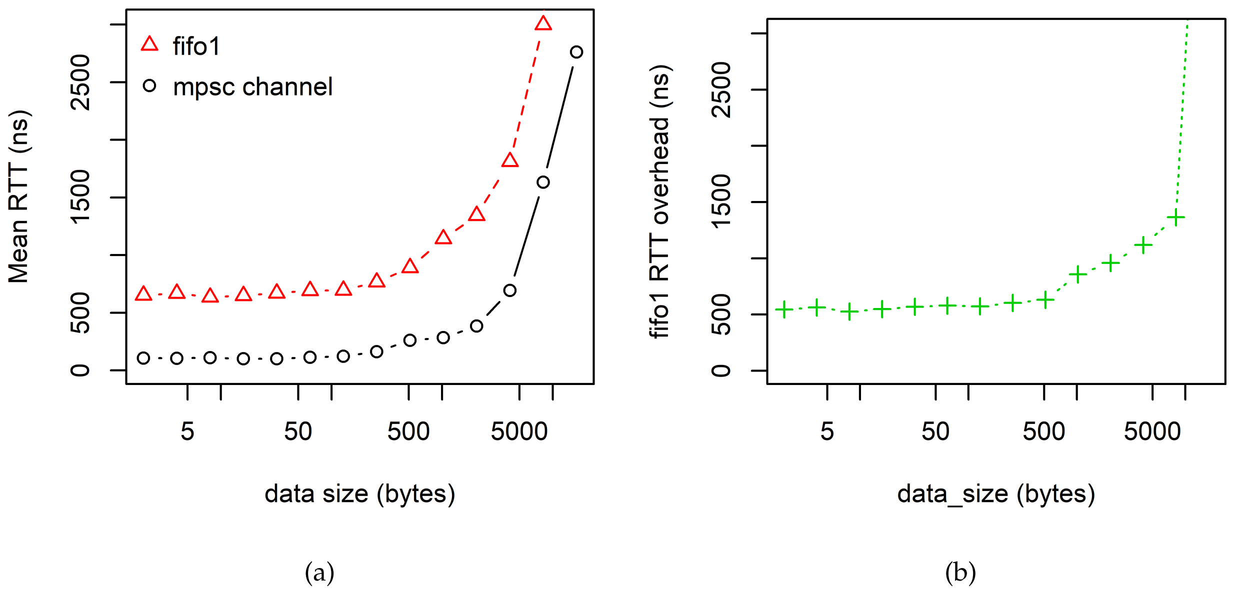


Figure 6.3: Time from beginning of `put` to end of `get` in connector *fifo1* compared to the time taken to send and received over an `mpsc` channel from the Rust standard library. Plots show the overhead in response to the size of the moved data. Figure (b) shows the difference between the curves in Figure (a), showing the overhead of Reo-rs more explicitly.

simple *fifo1* connector. For this case, Reo-rs appears to overlook a myriad of optimization opportunities that are not available for connectors in general; for example, the port removing elements (the getter) wastes time checking if any other getters contend their removal of the buffered value (clearly not). Figure 6.3 compares the total 'round trip time', measuring the mean duration of a 'round trip' of a single element, ie. starting from the moment `put` begins to the moment `get` ends. Sub-figure 6.3a shows this in contrast to the simplest channels in Rust's standard library: `mpsc` ('multiple producer, single consumer').

These measurements show the range of times taken in response to changes in the data's size. The figure makes clear that compared to `mpsc`, Reo-rs experiences significant overhead in all cases. Some uniform overhead is to be expected, as `mpsc` is purpose-built and optimized for precisely this scenario. Reo-rs lacks the ability to optimize for this protocol to the same extent. For example, as explained in Section 4.3.2, all port-operations involve the acquisition of a shared protocol lock. Conceptually, Reo-rs has all the information it needs optimize for this protocol and particular and match the performance of `mpsc`. The generality of Reo-rs inhibits its ability to identify and specialize for these kinds of optimizations to the same extent as was done by hand for `mpsc`. A hand-optimized fifo1 channel could take advantage of the fact that the two ends of the channel need not share a lock at all. For this protocol, partitioning the coordinator such that each part only considers *local* rules would be correct and cause less contention.

Figure 6.3b makes clear that Reo-rs is experiencing overhead that increases with the size of the moved value. At first glance, one can be forgiven for attributing this overhead to the presence of redundant movements of the data, which Section 4.3.1 explains can occur if llvm fails to optimize away the *physical* movements from Rust's *semantic* movements of values to new variable bindings. This turns out not to be the cause in this case. Rather, the overhead is caused by a more granular implementation detail; `mpsc` has a different method of moving its values. `mpsc` has some means to optimize data movements in the case of an x86-64 processor by 'in-lining' it, spelling it out into a sequence of smaller movements hundreds of lines long. In this manner, `mpsc` is optimized to avoid the overhead of system calls[1].

Things become more interesting when we apply Reo-rs to more complex problems that actually make use of its more expressive features such as synchrony and state. The *alternator* is a canonical Reo circuit that routes messages from putters $\{P_0, P_1\}$ to getter G in an alternating fashion (starting with P_0). The semantics of this connector are subtly different from that of a *sequencer*; namely, the alternator only routes data when all three participants are simultaneously ready, after which G receives a pair of messages in succession. The connector's semantics achieve this by transmitting the first value synchronously, and the dispatching latter asynchronously. Listing 6.4 shows an implementation of this protocol 'by hand' in Rust using `Barrier` from the standard library, and the bounded-channel type from the ubiquitous `crossbeam` library with bounds 0 and 1 to act as synchronous (AKA rendezvous) and asynchronous (AKA buffered) channels for data. Figure 6.4 shows the experimental results. Reo-rs never out-performed the hand-crafted code, but it was never far behind. As seen before, these optimized channels are able to more efficiently move data by using in-lining optimizations, increasing the gap as data gets larger. For values large enough to benefit from it, the figure also shows the speedup achieved by using the *unsafe* reference-passing API, described in Section 4.3.1.

Reo-rs is designed to handle the expressivity that characterizes the Reo language. The more synchronous and stateful a connector becomes, the more Reo-rs-*like* the handcrafted implementation will be.

[1]Listing 27 in the Appendix gives a glimpse of the assembly for this procedure.

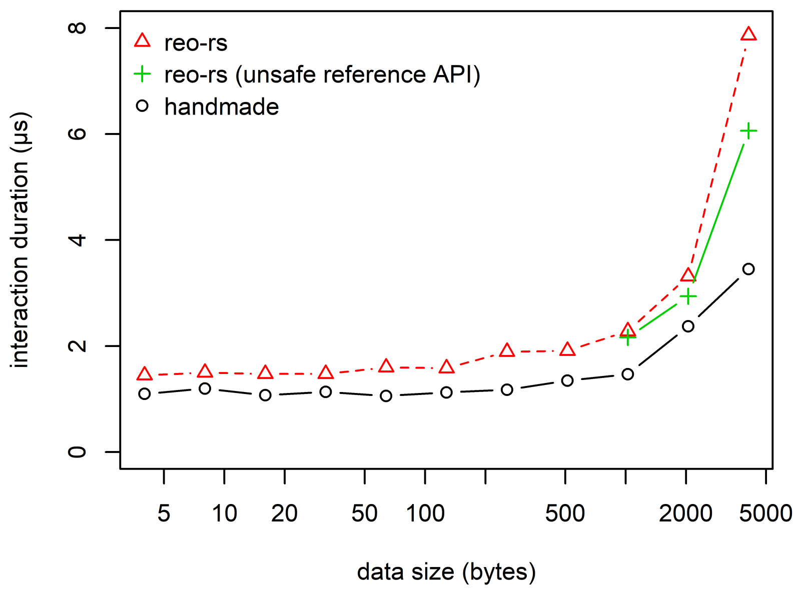


Figure 6.4: TODO.

6.3 Overhead Examined

Here, we examine the performance characteristics of Reo-rs in more detail under various circumstances. The goal is to understand how Reo-rs uses its computational resources, and how performance responds to properties of the specific protocol.

6.3.1 Port-Operation Parallelism

For connectors as simple as *fifo1*, overhead is overhead. However, we are particularly interested in understanding how this overhead is partitioned; as connectors become more complex, different parts of this overhead impact *parallelism* in different ways. Section 4.3 explains the nature of the *coordinator* role, and how its operations are performed holding the lock for the `Proto` instance which connects ports as their common communication medium. Table 6.1 shows measurements for an experiment that attempts to understand which proportion of our overhead is incurred *inside* the critical region, ie. by the coordinator. In the case of this experiment, our protocol has rules for movement which can be rendered as a *bipartite graph*, allowing data flow from any putter in $\{P0, P1, P2\}$ to any getter in $\{G0, G1, G2\}$. As explained in Section 4.3.4, movements such as these do not buffer the data elements inside the protocol; getters take values from putters directly. As a consequence, putters are both the first and last to parttake in any of our rule firings' data movements. The table shows the mean duration for which each putter was involved in such a firing. Along with the total duration of the run, we are able to compute to which extent these putters were able to work in parallel. We distinguish between four cases, corresponding to rows in Table 6.1. The first three cases do not involve the `clone` operation, and are observed to have insignificant differences for all measurements. For this experiment, with modestly-sized values, we conclude that there is no large difference in performance between these three cases:

move Values are moved from putter to getter synchonously.

copy Putters retain their values, and getters replicate them with a bit-wise copy that does not mutate the original.

signal Getters do not return any data. They return after releasing putters.

The final **clone** case attempts to observe the effects of intentionally delaying getters outside of the lock by necessitating the use of an explicit `clone` operation whose duration is artificially lengthened[2]. For these runs, putters retained their original values, but the datum was not marked with the `Copy` trait. In all cases, we observed that even at this coarse granularity, there was significant parallelism. For the majority of the time, new rules were able to fire whilst interactions were being completed outside of the critical region. The final case in particular was within a small rounding error of perfect parallelism.

6.3.2 Overhead Inside the Critical Region

The previous section, we saw an experiment with data moving between putters and getters. These runtimes included the putter's time spent not only on the movement itself, but also on the time spent in the role of *coordinator*. Here, we examine the work that pertains to this role and how changes to the definition of rules influence overhead. Figure 6.5 shows the overhead incurred by a coordinator that traverses *unsatisfied* rules before finding one to fire. It is apparent that in all cases, the overhead scales linearly, as expected. The time taken to evaluate the satisfaction of a rule varies greatly dependent on its definition; the evaluation of a rule involves many different operations mirroring the intricacies of the *imperative form* it models, described in Chapter 3. To represent the

[2] `sleep` calls were out of the question, as its variability is overwhelming at this scale. Instead, `clone` perform thousands of chaotic integer computations on the replica before returning it. This is intentionally obtuse such that the Rust compiler is unable to identify a trivializing optimization.

	mean active time			run	mean
	p0	p1	p2	duration	parallelism
move	2.037µs,	2.031µs,	2.017µs	214.991ms	2.83
copy	1.744µs,	1.852µs,	1.844µs	197.676ms	2.75
signal	1.962µs,	1.956µs,	1.937µs	207.181ms	2.83
clone	94.021µs,	94.082µs,	94.058µs	9420.148ms	2.995

Table 6.1: Runs of 3 putters greedily sending their data directly to any of 3 getters, 100000 times. This test was performed with 4 variants, differing on the properties of the data and whether the putter retained the original. The last column is derived, showing to what extent these putters were able to work in parallel.

possibility space, the figure shows measurements for a simple protocol which encounters replicas of one unsatisfied rule repeatedly, where its nature comes in four distinct variants:

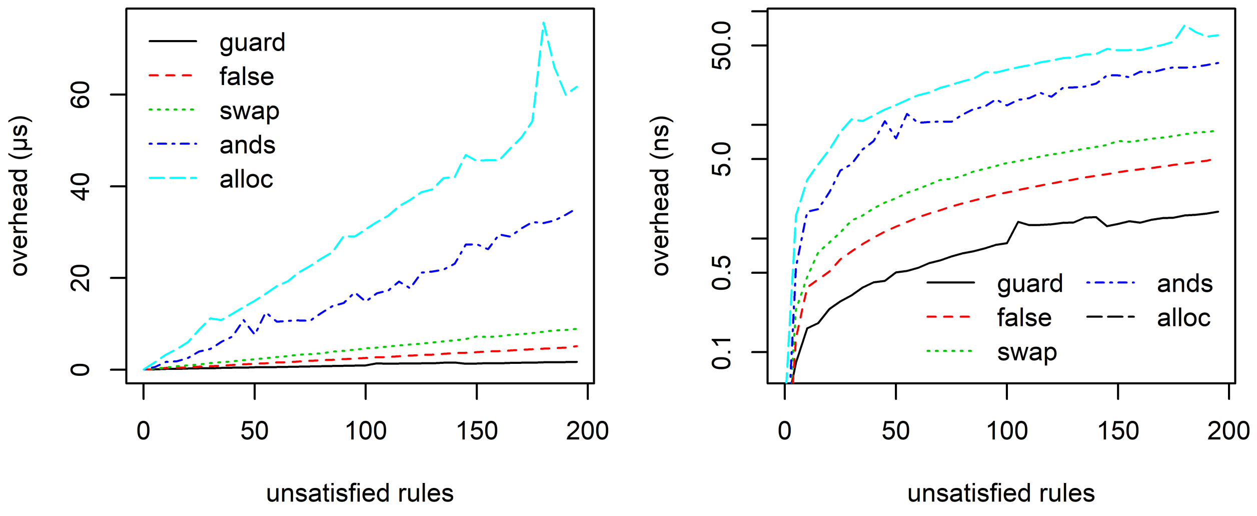
guard A rule where some port is not ready. This is detected almost immediately by cheap *bit vector* operations, as explained in Section 4.3.4. Evaluation takes 8.76ns.

false A rule whose first instruction checks the predicate *false*. Evaluation takes 18.91ns.

ands A rule whose first instruction is a tree-like formula structure of twenty-five conjunctions, only the last of which is *false*. Evaluation takes 180.72ns.

alloc A rule whose first instruction allocates a fresh boolean-type resource with value *false*. The second instruction checks that this temporary value is true. Upon failure, the allocation must be rolled back, discarding the temporary value. Evaluation takes 316.51ns.

These results meet our expectations. Rules can be arbitrarily complex, and perform an arbitrary amount of work before concluding that they are not satisfied, and will not fire. Even for this small set of relatively simple examples, we are able to see orders of magnitude in difference between the best case (in which the rule is skipped as early as possible by evaluating a bit-vector), to the worst case (involving several complex instructions). Fortunately, realistic Reo connectors will be defined almost entirely by rules that are either skipped as a result of evaluating these bit-vectors, or not at all. Even more expensive guards can expect to incur overhead in the order of nanoseconds as long as they involve no more than a handful of reasonable instructions.



(a) (b)

Figure 6.5: Overhead as a result of evaluating a sequence of identical unsatisfied rules before firing something. Experiment is repeated for four variants of unsatisfied rules varying in the complexity of the operations they before before being deemed unsatisfied. The two sub-figures show the same information, with (b) representing it with a logarithmic y-axis to accentuate the small-scale differences.

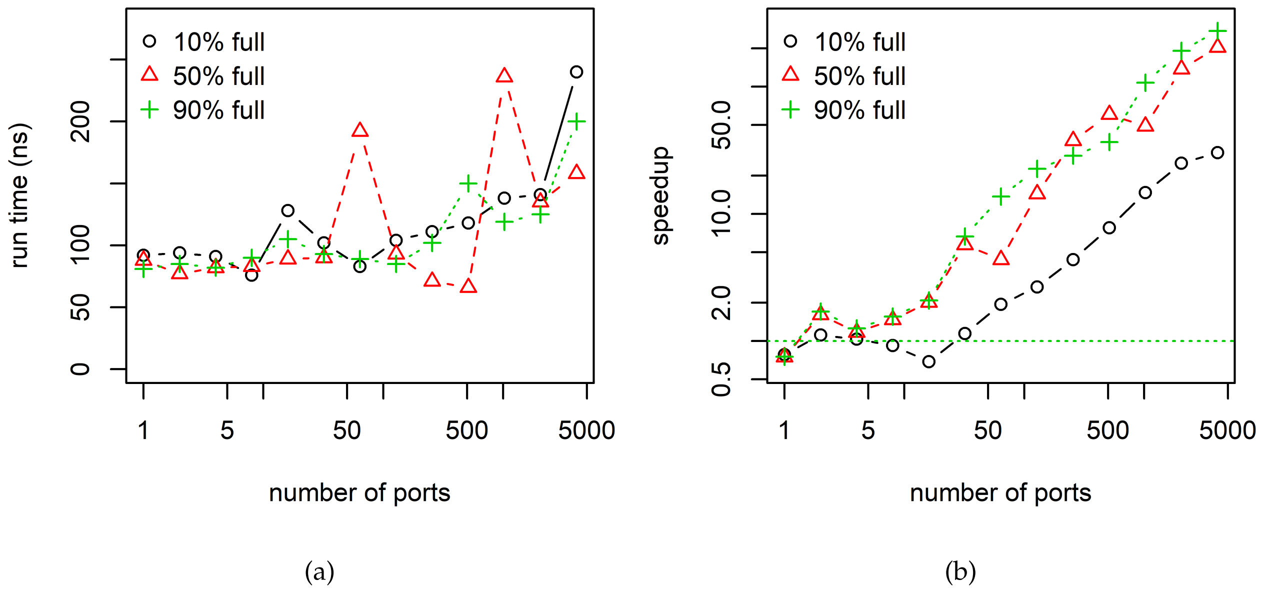


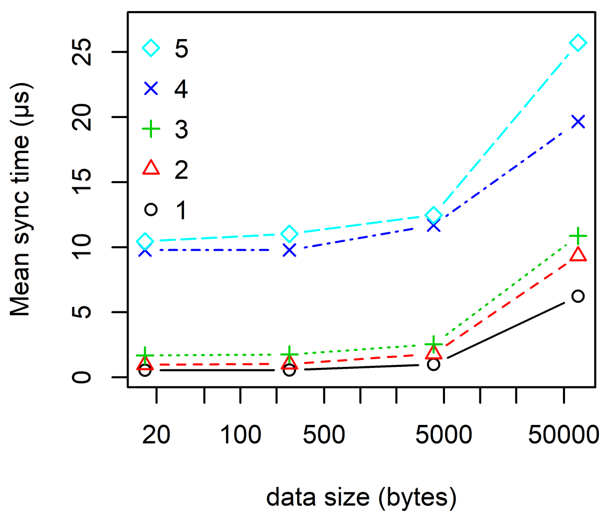
Figure 6.6: Run time of the `is_subset` operation for bit vectors and canonical hash sets. This operation is used very frequently by the coordinator to determine whether a rule is satisfied. Figures show (a) run times for the bit vector in response to a changing maximal element, ie. number of ports, and (b) the speedup of the bit vector in comparison to the hash set. Note the logarithmic axes.

The efficiency of the bit vector *subset* operation is key to the speed of the coordinator. This is the first first three things evaluated for each and every rule, checking whether all involved ports are ready, and if all the relevant memory cells are full or empty. The bit-set capitalizes on how unusually often we need this operation. Figure 6.6 shows how bit vectors are very efficient at checking whether one is a subset of another. Here we see the time taken to evaluate the *positive* case, representing the best-case scenario for our speedup. It is guaranteed to occur at least three times every time a rule fires. Figure 6.6a shows how low the cost of the operation stays, even when there are very many ports involved. Figure 6.6b shows how significant the speedup over the subset operation of canonical `HashSet` type. Admittedly, the majority of realistic Reo circuits are on the low-end with respect to number of ports; if nothing else, this is a result to encourage the development of more complex connectors. Observe that the cost of the operation is agnostic to the *fullness* in the case of the bit-vector. This is not so for the hash set, for which a fuller hash set makes for a more expensive operation.

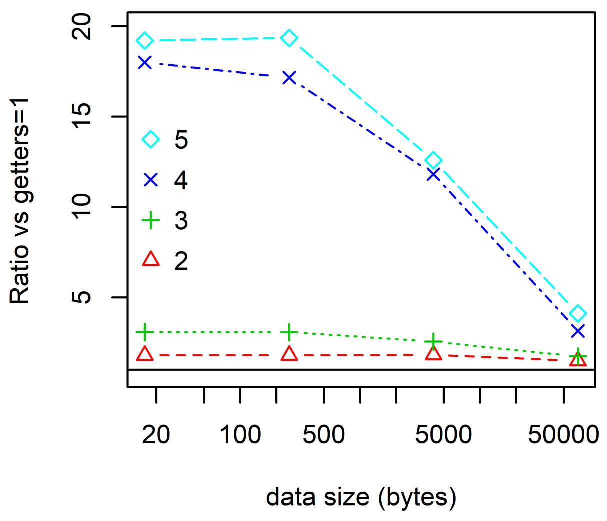
6.3.3 Overhead Outside the Critical Region

After data exchanges are initiated by the coordinator, the protocol's lock is released. Time spent exchanging can therefore only impact the threads that play a part directly. Figure 6.7 shows measurements of the *siso* protocol, which synchronously distributes a putter's datum to a set of getters. First, we concentrate on the case where this datum implements the `Copy` trait, communicating to Reo-rs that it is safe to replicate elements of the ports' data type with bit-wise shallow copies. The experiment shows the mean time to finish one such interaction, varying in response to the size of the datum, and repeated for a different number of getters. As expected, runs with more getters took longer to complete, as the putter cannot return until everyone else is finished. The cost of the copy operations can also be seen to grow approximately linearly. Figure 6.7b shows the runs normalized to that of the case with a single getter; here we observe that as the data gets larger, the overhead of the additional getters decreases in proportion to that of the single getter. The measurements for four and five getters are unexpectedly high compared to the rest. We are not able to fully explain this, and imagine it has something to do with these threads sharing resources with one another. Regardless, runs with these slower getters adhere to the prior observations, suggesting that whatever is influencing their performance is multiplying their overhead by a constant factor.

The experiment with connector *siso* is shown repeated in Figure 6.8. This time, the data type is not marked `Copy`. Section 4.3.4 explains how the dissemination of these values to multiple getters are handled, as there are multiple requirements on the order of operations such that correctness can be guaranteed.



(a)



(b)

Figure 6.7: Duration of a synchonization event from the perspective of a putter, putting to a set of getters. The data type implements `Copy`, allowing getters to copy the value bit-wise in parallel. Plots show five variants, corresponding to getter sets of sizes one through five, plotted in response to a data type with a changing size in memory.

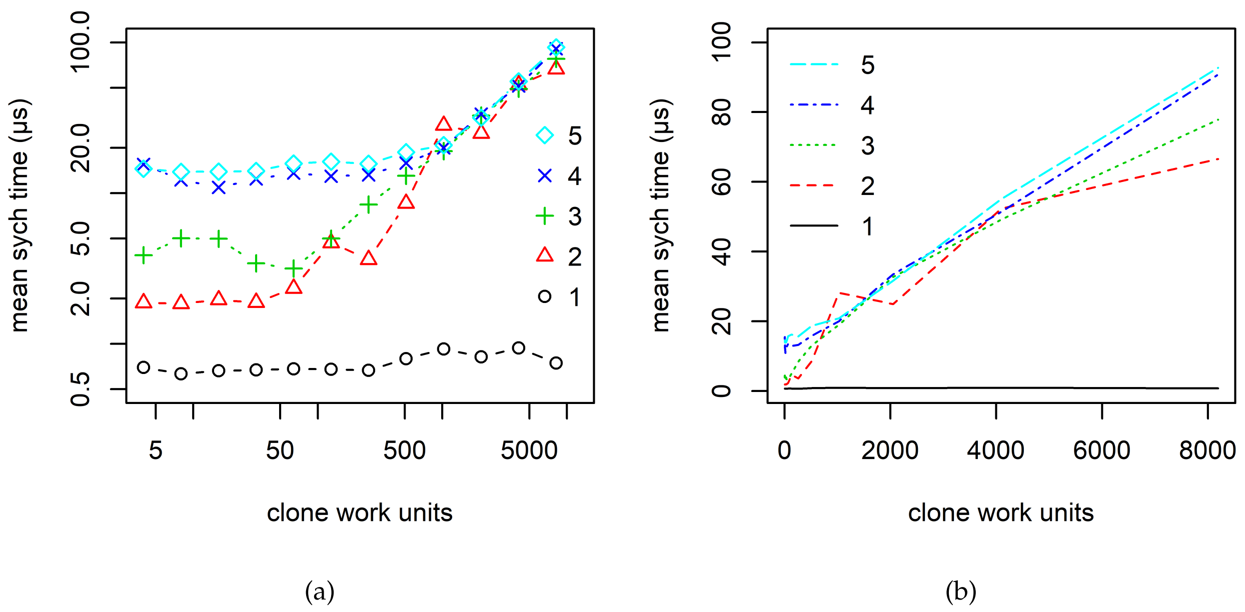


Figure 6.8: Duration of an interaction in a *siso* connector. Runs are distinguished by the number of getters per interaction with one putter, plotted over an increasingly expensive `clone` operation. Figure (b) repeats the information of (a) to make the subtle differences more visible.

In this context, Figure 6.8a shows that the case of a single getter is exceptionally fast. This is to be expected, as even for non-copy data-types, all but one getter must clone; someone gets the putter's original. These results demonstrate the benefits of this optimization; it is worth noting that invoking a clone also incurs the overhead of an added *virtual function call*. In this experiment, the cost of `clone` was manipulated synthetically as before. As expected, beyond a threshold, the cost of the clone operation dominates, causing runtimes to converge, regardless of the number of getters.

Figure 6.9 omits the exceptional single-getter case to show the effects of competing cloners. Measurements for cheap clone operations differ in runtimes both as a result of increased contention on concurrency primitives, but also as a result of simply having a larger group of getters to wait for, giving more opportunities for stragglers to delay completion.

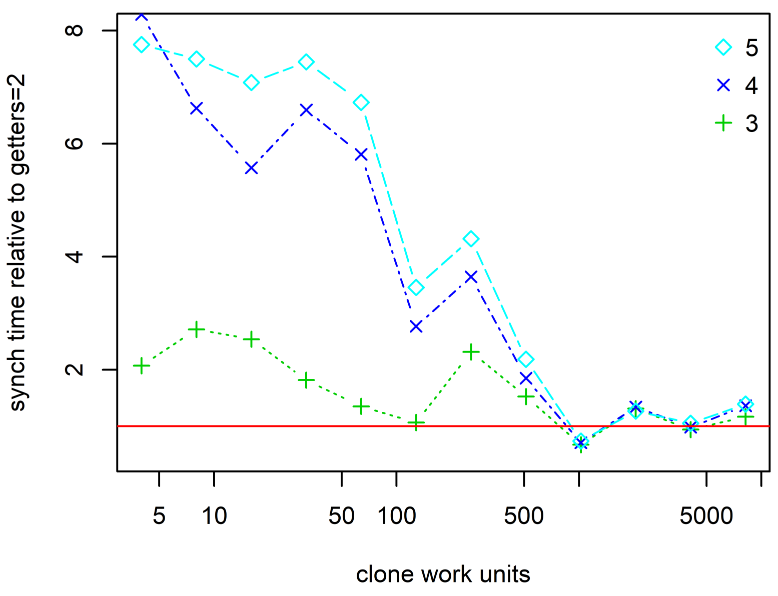


Figure 6.9: Measurements of all runs involving the `clone` operation from Figure 6.8, shown plotted relative to that of the case with 2-getters.

Chapter 7

Discussion

7.1 Future Work

TODO short intro

7.1.1 Imperative Form Compiler

we use an interpreter. previously, we mention our reasons for doing so. in other circumstances, the overhead of the interpretation may be undesirable. other language targets may be supported by instead again pre-processing imperative form to generate tailor-made target code as the Java back-end does today.

7.1.2 Distributed Components

TODO refer to Reowolf. talk about how protocols can become distributed by partitioning their memory space. problem is complex. involves consensus. refer to reowolf project. refer to farhad's and sung's previous work

7.1.3 Imperative Branching

as propositional formulas can be converted to DNF (with disjunctions on the outermost layer), so too can imperative form rules be split over OR branches into new rules (EXAMPLE). This idea can be taken to the extreme: splitting over the elements of the data domain and once again enumerating the transition space, resulting in something with the most degenerate RBAs possible with an explosion in rules. In some cases, moving in this direction is desirable, as it has the effect of making the rule GUARDS do all of the checking; as seen earlier, boolean guarded variables can be efficiently checked in bulk.

THe extreme of the spectrum is unlikely to be more efficient: the same transform values will be computed and discarded repeatedly, and as the number of boolean variables increases, at some point even batch-computing them falls behind. Future work could investigate this balance between a small number of rules, and determinisms in rules.

7.1.4 Runtime Governors

TODO more abstract suggestion. runtime governers. makes it possible to use them without affine types. eg in C

7.1.5 Further Runtime Optimization

Didn't implement everything for variious reasosn but lots is possible

1. simplify or remove predicates based on implicit information if rules have a KNOWN ordering. EG:: before [if P, if not P]. after [if P, true]
2. stuffed pointers for data types with representations no larger than ptr size. no need for allocator. complicates the refcounter mechanism
3. imperative form preprocessing. some redundancy exists in how you can represent things. eg: MemSwap sometimes can be achieed just by reorganizing your movements. more examples: pruning check subtries of tautologies and contradictions. this can happen in either reo or reo-rs

7.1.6 Avoid Lock Re-Entry

when you empty a memcell you need to coordinate again. would be nice if we can avoid locking a second time by either detecting when we know it will be unnecessary or just some mechanism to delegate the work

7.1.7 Runtime Reconfiguration

our protocol is data. we can change it and change its behavior. care must be taken because you are reconfiguring the structure laid out for speed

7.2 Conclusion

TODO

Bibliography

[ABRS04] Farhad Arbab, Christel Baier, Jan Rutten, and Marjan Sirjani. Modeling component connectors in reo by constraint automata. *Electronic Notes in Theoretical Computer Science*, 97:25–46, 2004.

[Arb04] Farhad Arbab. Reo: a channel-based coordination model for component composition. *Mathematical structures in computer science*, 14(3):329–366, 2004.

[DA18] Kasper Dokter and Farhad Arbab. Rule-based form for stream constraints. In *International Conference on Coordination Languages and Models*, pages 142–161. Springer, 2018.

[Gor] Manish Goregaokar. Exotic sizes. `https://doc.rust-lang.org/nomicon/exotic-sizes.html`. Accessed: 2019-06-01.

[JA12] Sung-Shik TQ Jongmans and Farhad Arbab. Overview of thirty semantic formalisms for reo. *Scientific Annals of Computer Science*, 22(1), 2012.

[KN18] Steve Klabnik and Carol Nichols. *The Rust Programming Language*. No Starch Press, 2018.

[Mat15] Niko Matsakis. Owned references to contents in an earlier stack frame. issue 998. rust-lang/rfcs, Mar 2015.

[NG14] Rob Nederpelt and Herman Geuvers. *Type theory and formal proof: an introduction*. Cambridge University Press, 2014.

[Nys14] Robert Nystrom. *Game programming patterns*. Genever Benning, 2014.

[rh14] User 'rust highfive'. Anonymous sum types. issue 294. rust-lang/rfcs, Sept 2014.

[Wal05] David Walker. Substructural type systems. *Advanced Topics in Types and Programming Languages*, pages 3–44, 2005.

Appendices

```rust
if T::IS_COPY { // irrelevant how many copy
    if let Some(dest) = maybe_dest {
        do_move(dest);
        m.visit();
    }
    let was = count.fetch_dec();
    if was == LAST {
        let [visited_first, retains] = m.visit();
        finalize();
    }
} else {
    if let Some(dest) = maybe_dest {
        let [visited_first, retains] = m.visit();
        if visited_first && !retains {
            let was = count.fetch_dec();
            if was != LAST {
                mover_await();
            }
            do_move(dest);
            finalize();
        } else {
            do_clone(dest);
            let was = count.fetch_dec();
            if was == LAST {
                if retains {
                    finalize();
                } else {
                    mover_release();
        }   }   }
    } else {
        let was = count.fetch_dec();
        if was == LAST {
            let [visited_first, retains] = m.visit();
            if visited_first {
                finalize();
            } else {
                mover_release();
}   }   }   }
```

Listing 26: A getter's procedure for retrieving a value from a putter or memory cell. Getters must coordinate such that one is elected the *mover* with all others cloning. The mover must go last, and once everyone is done, the resource must be cleaned up.

```
std::sync::mpsc::spsc_queue::Queue<T,ProducerAddition,ConsumerAddition>::pop:
        push    rbp
        mov     rbp, rsp
        push    r15
    /* 10 lines omitted */
        je      .LBB2_2
        mov     byte ptr [rsp + 288], 2
        movaps  xmm0, xmmword ptr [rsp + 288]
        movaps  xmm1, xmmword ptr [rsp + 304]
        movups  xmm2, xmmword ptr [rax]
        movaps  xmmword ptr [rsp + 272], xmm2
        movups  xmm15, xmmword ptr [rax + 16]
        movups  xmm2, xmmword ptr [rax + 32]
        movaps  xmmword ptr [rsp + 256], xmm2
        movups  xmm2, xmmword ptr [rax + 48]
        movaps  xmmword ptr [rsp + 240], xmm2
        movups  xmmword ptr [rax + 16], xmm1
        movups  xmmword ptr [rax], xmm0
        movups  xmm0, xmmword ptr [rax + 64]
        movaps  xmmword ptr [rsp + 224], xmm0
        movups  xmm0, xmmword ptr [rax + 80]
        movaps  xmmword ptr [rsp + 192], xmm0
        movups  xmm0, xmmword ptr [rax + 96]
        movaps  xmmword ptr [rsp + 208], xmm0
        movups  xmm0, xmmword ptr [rax + 112]
        movaps  xmmword ptr [rsp + 160], xmm0
        movups  xmm0, xmmword ptr [rax + 128]
    /* 153 lines omitted */
```

Listing 27: Snippet out of the x86-64 assembly generated by receiving a large datum through `recv` from a simple channel from the Rust standard library. It unrolls the movement of the entire object into a large sequence of smaller operations rather than invoking a system call. This is the case for the receipt of a `Copy`-type represented by 512 bytes.