



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.*

**Generating Safe and Fast
Coordination code in
Rust with Reo**

Christopher Esterhuyse

(ID: 2553295)

supervisors

Vrije Universiteit Amsterdam
dr. J. ENDRULLIS

Centrum Wiskunde & Informatica
prof. dr. F. ARBAB

July 22, 2019

Abstract

TODO

Contents

I	Preliminaries	5
1	Introduction	6
2	Background	8
2.1	Reo	8
2.1.1	Motivation	8
2.1.2	Language	9
2.1.3	Semantic Models	10
2.1.4	The Reo Compiler	15
2.2	Imperative Languages	17
2.2.1	Affine Type System	17
2.2.2	The Rust Programming Language	18
2.2.3	The Type-State Pattern	24
2.2.4	Proof-of-Work Pattern	26
II	Contributions	29
3	Imperative Form	30
3.1	Motivation	30
3.2	Role in the Reo Pipeline	30
3.3	Imperative Form Definition	32
3.4	From Reo to Imperative Form	33
3.4.1	Compiler Internal Representation	34
3.4.2	Action Sequencing	34
3.4.3	Compiler Output	34
3.5	From Imperative Form to Rust	35
3.5.1	Soundness Checks	35
3.5.2	Preprocessing	37
4	Protocol Runtime	40

4.1	Examining the Java Implementation.....	40
4.1.1	Architecture	40
4.1.2	Behavior	42
4.1.3	Observations	42
4.2	Requirements	47
4.2.1	Requirements Defined.....	47
4.2.2	Requirements Evaluated	48
4.3	Protocol Runtime	48
4.3.1	Application User Interface.....	48
4.3.2	Design Process.....	52
4.3.3	Architecture	56
4.3.4	Behavior	57
5	Generating Static Governors	70
5.1	The Problem: Unintended Constraints	70
5.2	Governors Defined.....	72
5.3	Solution: Static Governance with Types.....	72
5.4	Making it Functional	72
5.4.1	Encoding CA and RBA as Type-State Automata.....	72
5.4.2	Rule Consensus.....	73
5.4.3	Governed Environment	74
5.5	Making it Practical.....	75
5.5.1	Approximating the RBA	75
5.5.2	Unknown Memory State	81
5.5.3	Match Syntax	81
6	Benchmarking	82
6.1	Goal	82
6.2	Experimental Setup.....	82
6.3	Results	82
6.4	Observations	82
III	Reflection	83
7	Discussion	84
7.1	Future Work	84
7.1.1	Imperative Form Compiler	84
7.1.2	Distributed Components.....	84
7.1.3	Imperative Branching.....	84

7.1.4	Runtime Governors	85
7.1.5	Further Runtime Optimization	85
7.2	Conclusion	85

List of Figures

2.1	CA for fifo1 connector.	12
2.2	CA with memory for fifo1 connector.	13
2.3	CA with memory for fifo2 connector.	14
2.4	RBA for fifo1 connector.	15
2.5	RBA for fifo2 connector.	16
4.1	TODO	55
5.1	RBAs in lockstep with and without normalization.....	80

Part I

Preliminaries

Chapter 1

Introduction

these days useful abstractions are everywhere. communications are still relatively primitive: they are solved in a case-by-case basis but there isn't a general approach. the problem is that machines work with actions, and we think in interactions. there are many approaches to representing one with the other. for example, session types allow you to represent actions, and then allow you to predict your PARTNER's actions. Reo is neat in that its exogenous. it puts the coordination in one place so you extract the protocol from your code. this makes it easier to collect and reason about. you express your protocol in a high-level language and then use that as a specification. its explicitness makes it very useful for humans, but also for machines; the reo compiler is a tool for generating coordination glue code. the intuition is you abstract away the structure of your network and you no longer distinguish endpoints etc. you instead use PORTS as your interface. the compiler generates the details for you and then it behaves according to the protocol at runtime . the reo compiler has support for numerous backends such as java. 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 related to C++, 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 predictable manner. its ownership rules also prevent

the majority of data races and protect the programmer from undefined behavior such as accessing uninitialized memory. 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. 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'. Chapter BLAH deals with the translation process itself. Chapter blah describes how the generated rust code performs the role of a coordinator at runtime, detailing significant optimizations, particularly focusing on those that take advantage of components co-existing in shared memory. Chapter blah investigates the development of additional tooling for automatically detecting deviations from protocols at compile-time using no extra compilation steps. Chapter BLAH investigates the runtime characteristics of these systems at runtime. Part BLAH reflects on the progress of the project and suggesting directions 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, how the language is used, and (at high level) how it works. 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 focus on safety properties

Modern software development involves the construction of large and complex projects. Owing to their scale and the heterogeneity of the tasks required, many people are involved in the development of a program at once, and over its development lifetime. The industry has long-since established paradigms for managing the scale of these projects. One tenet of good software design is *modularity*, which describes a structure that, instead of being designed monolithically, is built out of smaller constituent modules. In addition to isolating modules such that they can be re-used in other projects, this design philosophy allows contributors to concentrate on a subset of all the modules at a time. These ideas are well-established in practice; code re-use and separation-of-concerns have been prevalent for some time.

Reo's utility is not only its ability to facilitate modularity. Reo is de-

signed such that properties of the individual modules are *preserved* when modules are *composed* into larger ones. This preservation marks the difference between *gluing* modules together (and hoping for the best), and *composing* them into something guaranteed to have the intended properties.

2.1.2 Language

Reo is a *coordination* language. This describes its focus on the specification of the *interactions* between distinct actors. This is in contrast to the usual *action-centric* model common to languages with their roots in sequential programming, where the programs or specifications describe *actions* of entities, relegating any associated interactions to requiring derivation from the actions. In a nutshell, Reo provides a language for describing the behavior of a *system* of actors by explicitly constraining the behavior of the *connector* which serves as their communication medium.

(TODO define connector. same as component just maybe structural?)

The Reo language is essentially graphic; each connector defines a relation over named *locations*. Complex connectors are defined as the composition of simpler connectors over its locations. This inherently visual language is also often seen in its textual form, usually in the context of machine parsing.

The simplest *primitive* connectors cannot be subdivided

by listing a set of constituent connectors. The simplest primitive Reo connectors are *channels*(TODO channel vs primitive). Channels by definition cannot be subdivided into constituent connectors, as they are defined by either (a) the model that provides Reo's semantics, or (b) opaque components defined in some target language such as C or Java. The nodes themselves are the other important aspect of the language. Ultimately, each node corresponds to a (logical) location which may hold up to one datum at a time. Reo is by default *synchronous*, and relationships between locations propagates that synchrony. *Locations* are divided into two classes according to whether

(TODO)

This motivates the Reo's metaphor of propagation of data and back-propagation of data-constraints, corresponding to its namesake, the Greek word for 'flow'. The compositional aspect is meaningful when locations are involved in multiple relationships.

forwards (by moving data several 'hops' at once) and backwards

In addition to re-using nodes inside a connector, connectors are able

to expose these nodes for re-use in the connectors *above them* by exposing the node in the connector's *interface*. These exposed nodes are called *ports*, leaning on the metaphor of the connector *moving* some data in and out of itself.

(data flow corresponds with what happens at runtime, except its SYNCHRONOUS by default. Relate to TDS. Talk about replication and equality checks).

(In the context of applications, components that cannot be composed compile to things managed by different threads. at the boundaries, they communicate with ports. Here, there is a meaningful difference between putter and getter. Include example of how a protocol that uses some port A three times still results in a putter-getter pair)

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, 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

\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.

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 future¹. 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 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

¹Not 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 $(\Box P \wedge \Box \Box P \wedge \Box \Box \Box P)$.

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 behavior; 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 *fifoN* 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 un-



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 .

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



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.

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 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 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.



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.

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 program 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 ask Sung to summarize the history of the Reo compiler. give a summarized story here.

The compiler aims to take the low-level implementation of a protocol out of the application developer’s hands. Given a protocol specification, the compiler generates the *glue code* and



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.

TODO focus on RBA

The steps from Reo specification to the generated glue code can be better understood when broken down into stages:

1. Specification expansion

The composed definitions of Reo components are unrolled to the channel-level until the protocol is represented by one large automaton with many nodes.

2. Minimization

Nodes not in the protocol’s interface are *hidden* and the RBA is minimized. This step produces a new, simpler automaton with the same behavior and interface.

3. TODO PUTTERS AND GETTERS

4. Linking and Code Generation

The finished source code is generated from the resulting internal-representation. Those associated with functions in the target language are linked accordingly, and the rest are parsed and translated from the operational semantics of Reo to suitable target-language operations (such as data movement and duplication). The rules of the internal state are translated to the runtime definition of a protocol

```
1 void stateA() {  
2     this.value = get();  
3     if (this.value == 3) {  
4         stateB();  
5     } else {  
6         stateA();  
7     }  
8 }  
9 void stateB() {  
10     put(this.value);  
11     stateA();  
12 }
```

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 at runtime.

```
1 f1fo1(a?, b!) {  
2     #RBA  
3     {a, ~b} $m = null, $m' = a  
4     {~a, b} $m != null, b = $m, $m' = null  
5 }
```

Listing 2: TODO.

component object. An entrypoint to instantiating this protocol object is generated with the appropriate interface. The specifics of this step vary per target language.

2.2 Imperative Languages

In this section we briefly introduce the properties relevant for discussion associated with imperative programming languages. In our case, these languages are the targets for the Reo compiler.

2.2.1 Affine Type System

The notion of ‘affinity’ as we use it has its roots in classical and intuitionistic 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 assigned type τ) under context Γ . A context is simply a list of statements, which can be thought to correspond with *assumptions* or *premises*. The judgment holds if one can construct a proof, starting with the judgment and select and applying rules until no ‘dangling’ judgments remain. For example, simply-typed lambda calculus has a type-derivation rule for abstraction, substitution and application [NG14]. 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.

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

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 the application of 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++ pro-

grammers. 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

```
1 struct Foo;
2 fn func(x: &Foo) {
3     // this function borrows some `Foo` structure by reference.
4 }
5 fn main(){
6     let x = Foo; // instantiate.
7     func(&x);    // &x is created in-line. x remains in place.
8     func(&x);    // another &x is created. x remains in place.
9 }
```

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`.

Java, Rust traits say nothing 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 spe-

```
1 pub fn something<T: PrintsBool>() {  
2     T::print(true);  
3 }  
4 trait PrintsBool {  
5     fn print(bool);  
6 }  
7 impl PrintsBool for String {  
8     fn print(b: bool) {  
9         println!("bool is: {}", b)  
10    }  
}
```

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`.

cialized 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 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's 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.

```
1 trait Emits {  
2     fn emit(&self) -> usize;  
3 }  
4 impl Emits for String {  
5     fn emit(&self) -> usize {  
6         self.len()  
7     }  
8 }  
9 fn func_static<T: Emits>(x: &T) -> bool {  
10     x.emit() > 10  
11 }  
12 fn func_dynamic(x: &dyn Emits) -> bool {  
13     x.emit() > 10  
14 }  
15 fn main() {  
16     let value = String::from("Hello!");  
17     func_static:::<String>(&value);  
18     func_dynamic(&value as &dyn Emits);  
19 }
```

Listing 6: TODO.

`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 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 (`instance` 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 instance at a time. At runtime, the variant can be discriminated 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 represented 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`.

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 object². 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].

²Usually, 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.

```
1 struct DivZeroError; // contains no data
2
3 fn divide_by(numerator: f32, divisor: f32) -> Result<f32, DivZeroError> {
4     if divisor == 0. {
5         Result::Err(DivZeroError)
6     } else {
7         Result::Ok(numerator / divisor)
8     }
9
10 fn main(input: f32) {
11     match divide_by(4.5, input) {
12         Ok(x) => print!("Success! computed:{}. ", x),
13         Err(_) => print!("Something went wrong!"),
14     }
15 }
```

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.

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.

```
1 typedef struct DoOne { int n; } DoOne;
2 typedef struct DoTwo { int n; } DoTwo;
3 const DoOne START = { .n = 0 };
4
5 DoTwo one(DoOne d1) {
6     DoTwo d2 = { .n = d1.n + 1 };
7     return d2;
8 }
9 DoOne two(DoTwo d2) {
10    DoOne d1 = { .n = d2.n * 2 };
11    return d1;
12 }
```

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.

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

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

```
1 fn main(d1: DoOne) {  
2   let d2 = two(d1);  
3   let d1 = one(d2);  
4   let d2 = two(d1);  
5   let d2_again = two(d1); // Error! `d1` has been moved.  
6 }
```

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.

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[[exo](#)].

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 `Fulfilled` objects in their own code.

```
1 struct Promise;
2 struct Fulfilled;
3
4 fn fulfill(p: Promise) -> Fulfilled {
5     // invoked once per `main`
6     return Fulfilled::new();
7 }
8
9 fn main(callback: fn(Promise)->Fulfilled) {
10    // ...
11    let _ = callback(Promise::new()); // `Fulfilled` discarded.
12    // ...
13 }
```

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.

Part II

Contributions

Chapter 3

Imperative Form

In this work we introduce another representation of Reo protocols intended to ease the transition from the Reo compiler’s internal representation to generated code in some target imperative language. In this chapter, we describe and exemplify this imperative form and describe the processes of translating to and from this new representation.

3.1 Motivation

3.2 Role in the Reo Pipeline

Reo is a declarative language that describes constraints as relations. This is a sensible choice because it provides an intuitive language for representing *interactions* as synchronous events, but also because this form lends itself well to the *composition* which the compiler must perform while unrolling specifications into their constituent connectors. When used to generate coordination glue code imperative languages, these high-level abstractions must at some point be decomposed into *actions* of individual participants. Currently, this translation is performed in the Reo compiler’s back-end. This is no simple task, and so the compiler leverages many tools such as *string template generators* to make the implementation more transparent and robust. Still, this approach has some undesirable consequences: (1) The Reo compiler itself becomes entangled with the syntax of all of its target languages; changes to target languages necessitate that it also be maintained to catch up. (2) Even template generators cannot fully abstract away the syntactic minutia of the target language; this can obfuscate

implementation errors¹. Rust in particular requires a significant amount of work to represent these rule firings, owing to its unique requirements for the user to take care that any variable bindings are *consumed* or *mutated* for the program to compile. While these concerns are unique to Rust, other languages will have concerns of their own. However, imperative languages have in common their expression of computation as sequences of operations. The set of actions comprising an interaction must thus be laid out with respect to run time, taking care only to operate on values in scope (Eg. certainly *after* the variables are declared).

The Reo compiler is also prepared to represent arbitrary *function calls* as part of the data constraints. Subtleties in the details of how these are declared have important implications on how these calls must be performed in an imperative setting. Consider a data constraint $f(P_0) = f(P_1) = C$ where P_0 and P_1 are *putters* (their ports are oriented to send data *into* the protocol or to peers), and C is a getter. Recall that the data constraints in an RBA are interpreted as ‘allowed’ synchronous observations of data. The protocol cannot control the values submitted by the putters, but it *can* (and does) control the value received by getters. In this view, such rules can be understood decomposed into *guards* and *assignments*, the former of which will be checked at runtime, and if satisfied, the latter will be performed. This case is interesting because the results of f are part of both the guard and the assignment; the rule does not fire unless $f(P_0) = f(P_1)$, but if it does, C is sent this value. This leads to an interesting conundrum: the value resulting from the execution of $f(P_0)$ and $f(P_1)$ are only needed based on a decision which is a function of $f(P_0)$ and $f(P_1)$; ideally; when the output is such that the rule is not fired, an outside observer should consider f never to be executed at all. Of course this is impossible when f is later instantiated into a target-language function. Instead, imperative form covers these cases by adopting semantics similar to that of *transactions*.

Imperative form has rules, each of which corresponds to an interaction (as with RBAs). Each rule consists of a sequence of actions that, when executed to completion, are observed to perform the interaction. However, these actions are partitioned such that there is a moment where the rule can be considered to *commit*. Once committed, any action performed guarantees all actions will be performed. On the other hand, actions *prior*

¹For example, the Java compiler contained a bug which resulted in swapped memory values as a result of generated code performing memory assignments in the wrong order. These errors are difficult to catch just by looking at the templates.

to commitment must be *reversible* such that they can be *rolled back*. This necessitates that no observable behavior be emitted prior to commitment.

3.3 Imperative Form Definition

A protocol in imperative form is a set of rules, each of which corresponds to a conditional interaction much like with RBAs. They are structured around the management of *resources*, so called not to be confused with ‘variables’ (which represent persistent memory slots in protocol objects). A resource is some initialized data item that is *available* at a moment in time. Memory cells which are filled are thus also resources. Concretely, an imperative rule is a tuple (P, I, M) with:

1. Premise P

A tuple of three *identifier* sets (P_R, P_F, P_E) . Set P_R contains the *synchronization constraint* identical to that of RBAs, referring to the set of ports that ‘fire’. In the context of a premise, this set describes which ports must be *ready*. P_F and P_E are the sets of *memory variables* which must be known to be full and empty respectively. By definition, any pair of these sets must have an empty intersection. The rule can certainly not consider firing unless all ports are ready and all memory cells are in the specified states. Omitting a port or memory cell represents *undefined state*, and thus is not considered an accessible resource.

2. Instructions I

A list of reversible *instructions* which are performed in sequence. These instructions can manipulate resources. As they are tentative, it is also possible to trigger a rollback. Concretely, each instruction is one of:

- $\text{check}(p)$
Trigger a rollback if predicate p over data is satisfied.
- $\text{fill}_P(m, p)$
Fill an empty memory variable m with the result of a predicate p over available data.
- $\text{fill}_F(f, a)$
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.

- `swap(m0, m1)`
Swap the values in two memory variables m_0 and m_1 ; in principle, any reversible data-agnostic manipulation is possible, but swapping values is sufficiently expressive.

If a rollback is triggered by `check`, any swapped memory cells are swapped back, and any memory cells whose values were created by `fillP` or `fillF` 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 $f(P_0) = f(P_1) = C$ and synchronization constraint $\{P_0, P_1, C\}$ will be represented in the imperative-form rule with guard $P = (\{P_0, P_1, C\}, \{\}, \{t_0, t_1\})$, instruction list $I = [\text{fill}:t_0 = f(P_0), \text{fill}:t_1 = f(P_1), \text{check}:t_0 = t_1]$, and with movements $\{t_0 \rightarrow \{C\}\}$. The explicit use of instructions to fill and check the values of temporary variables t_0 and t_1 .

Identifiers in the imperative form representation are *strings*, as it is a representation common to every realistic programming language. How these symbolic identifiers are represented ultimately is not specified, and implementations are free to choose whatever works best. Above, we have seen how these identifiers are used to link instructions, movements and predicates together such that relationships between them can be understood when reading the imperative form. To disambiguate the identifiers for ports, memory cells, temporary variables and function handles, the imperative form includes a field for *defining* names common to all rules.

3.4 From Reo to Imperative Form

This section details the procedure by which the Reo compiler transforms its internal representation to imperative form. Currently, this only occurs when Rust is chosen as the target language. As such, the current syntax for the imperative form is in Rust syntax, but otherwise corresponds closely to its description in the previous section.

3.4.1 Compiler Internal Representation

Internally, the Reo compiler represents connectors very similarly to RBAs. The two most significant differences are (1) the compiler collects annotations in the textual reo specification for the initial values of memory cells, if any are provided. (2) the *term* assigned to each getter-port and empty memory cell are identified and associated with the appropriate identifiers. At this stage, the behavior of the compiler is *specialized* according to the chosen target language. Concretely, the starting point for our back-end is a large *protocol* declaration object, consisting of (1) the ‘interface’, with port identifiers and whether they are putters or getters, (2) identifiers for memory variables and optionally a string representing their initial value, (3) a set of rules. Each of these rules, in turn, has (1) a representation of the logical guard, (2) the synchronization constraint, and (3) a partial mapping from identifiers to their assigned term.

3.4.2 Action Sequencing

TODO

3.4.3 Compiler Output

Observe that the definition of imperative form from Section 3.3 does not make any mention of the *initial state* of the protocol’s memory cells. This is a practical choice, as while there is a clear way to non-destructively create a runnable protocol object from its imperative form definition (the data types which comprise it are all under our control), the same is not true for the *values* a protocol object begins in its initialized memory cells. Instead, this design opts to separate the initial memory values from its imperative form description. This has the benefit of making it possible to instantiate protocol objects from a given imperative form *non-destructively*, such that one imperative form may be initialized into a runnable object repeatedly. Each time, it must be given initial values constructed anew.

Reo makes it possible to express protocols with a mix of generic and specific types. Ultimately, this allows for the creation of protocol descriptions for data types that are only determined *after* being emitted by the Reo compiler, but does not necessarily require it. Indeed, there are cases where it is desirable to be specific. For example, protocols may want to specify the type of internal memory cells to be *unit*, as they are used to encode state, but never store any user-facing data.

To meet these needs, Reo generates Rust source which contains a *generic function* which returns an imperative-form protocol description. The instantiation of the generic arguments determine the types and the initialized memory in the context of being *invoked* by code elsewhere. Section 2.2.2 provides more details on how Rust represents these generic arguments. Listing 11 gives an example of what Reo generates as output given a connector which *asynchronously* forwards some generic type `T` from `A` to `B`, where `B` receives the value transformed by some unspecified function `f`. Note that the result of this function is not the protocol definition, but rather the *instantiated* protocol object itself. This is clearly not the only use for such a structure. Listing 12 gives another example of a use for an imperative form generated in Rust. This is *not* currently done by the Reo compiler, but serves to illustrate what is trivially possible; here, `lazy_static` declares an immutable resource which is lazily initialized across threads such that it can be repeatedly built by reference using `&MY_PROTO`. This protocol illustrates the example from Section 3.3 where `f` is fixed as a function which returns the modulus of the input with 3. It demonstrates how a value computed from an arbitrary function can be incorporated into *both* the guard and assignment of one rule.

3.5 From Imperative Form to Rust

This section discusses how a finished *protocol object* is extracted from the imperative form representation in the `build` method. Rather than being performed by code generated for every particular protocol, this operation is provided in a dependency to the **Reo-rs** library. The details of this contribution are discussed in depth in Chapter 4.3 to follow. Here, it suffices to say that the library’s representation is very similar to the imperative form.

3.5.1 Soundness Checks

In a sense, a language is ‘declarative’ by expressing their values (or their computations) while intentionally omitting the control flow that performs the work. On the other hand, ‘imperative’ languages are characterized by doing the opposite; they make explicit the control flow of a program, and thus the values for bindings at a moment in time is a derived concept. Imperative form appropriately decouples values from their contents, exposing a larger surface for creating ill-formed rules. For example, it is incorrect to express a rule which will fill a memory variable if that variable

may already be filled; this would manifest as *overwriting* data at runtime. To aid with isolating the development of the Reo back-end from the Rust library, checks are performed to ensure that the input data structure can be interpreted as a well-formed protocol in imperative form.

One of Rust's most unique features is its static *borrow checker*, which has the task of emitting errors if it cannot determine, based on the header and body of a function alone, that all operations within the function adhere to Rust's *ownership rules*. These are described in more detail in Section 2.2.2. The borrow checker is limited to checking *actions* within the current function, and has no concept of interactions between concurrent threads. As such, implementing interaction primitives necessitates delving into *unsafe Rust* for actions which are unsafe when viewed in isolation. For example, it is unsafe to return type *T* acquired by dereferencing a pointer to *T* acquired through a message channel; whether the pointer is valid or whether the memory is initialized cannot be known. The Rust borrow checker is therefore unable to assist in ensuring that the data operations performed by our imperative instructions are safe. Instead, our translation procedure mimics the borrow checker. The reasoning is idiomatic for the affine language, following the control flow of each rule and keeping track of which resources are *available*. Instructions which would overwrite possibly initialized resources, or read from possibly uninitialized resources are *rejected* with appropriate error messages specifying the line number and the name of the variable in question.

Rust is not a memory-managed language. Instead, the ownership system keeps track of variables and inserts predictable de-allocation calls if they go out of scope before being consumed. For this reason, types must be annotated with their *ownership*, distinguishing references by whether their values are *logically* transferred to the scope of the function. Clearly it is incorrect for an imperative rule's *movement* to include a mapping for a resource which is not available. To mimic the borrow checker, it would be wise for the library to reject inputs for rules which leave resources unmapped; if left unchecked, this would result in leaking memory at runtime. To achieve the same result more ergonomically, Reo-rs opts to insert trivial movement-mappings for resources which must be freed but are not consumed, as the borrow checker would; ie. unconsumed resources *move* to the locations of an empty set of recipient identifiers. With this modification, the Rust runtime can rely on the set of mapped resources exactly corresponding to those consumed as the result of the rule's firing.

3.5.2 Preprocessing

Reo specifications include a lot of informations about what is expected of the protocol's observable behavior. That is indeed the point. By relying on `build` being the only publically-accessible means of constructing runnable protocol objects, checking well-formedness can be front-loaded to `build` such that they can be safely presumed to hold at runtime.

Along with the initial values for memory cells, `build` is responsible for setting up the state of the runnable protocol object which is needed for it to function. Conceptually, protocol state is limited to the contents of memory cells (and perhaps, a flag or `Option` representation for distinguishing *full* from *empty*). In practice, the coordination of threads requires some additional data structures that exist for guaranteeing mutual exclusion, helping threads find one another, and channels for inter-thread communication. These primitives are all constructed at the end of `build`, invisibly to the user. Chapter 4.3 to follow goes into detail about what happens next.

```
1 fn def_async_transform<T>(f: fn(&T) -> T) -> Proto {
2   ProtoDef {
3     name_defs: {
4       "A" => Port(Putter, T),
5       "B" => Port(Getter, T),
6       "mem" => Memo(Uninit, T),
7       "f" => Func(FuncHandle::new(f)),
8     },
9     rules: [
10      RuleDef {
11        premise: Premise {
12          ready_ports: {"A"},
13          full_mem: {},
14          empty_mem: {"mem"},
15        },
16        instructions: [],
17        movements: { "A" => {"mem"} }
18      },
19      RuleDef {
20        premise: Premise {
21          ready_ports: {"B"},
22          full_mem: {"mem"},
23          empty_mem: {},
24        },
25        instructions: [
26          CreateFromFunc("temp", "f", [Resource("mem")]),
27        ],
28        movements: { "temp" => {"B"} }
29      }
30    ],
31  }.build(MemInitial::empty())
32 }
```

Listing 11: Output out the Reo compiler with the Rust language target for an input connector which asynchronously forwards some generic data-type `T` from `A` to `B` through a transform function `f`. `T` and `f` are chosen by the user whenever `def_async_transform` is invoked, constructing a new *runnable* Rust protocol object with the `build` method. The syntax has been abbreviated slightly for readability, but is otherwise faithful.

```
1 lazy_static! {  
2     static ref MY_PROTO: ProtoDef = ProtoDef {  
3         name_defs: {  
4             "P0" => Port(Putter, u32),  
5             "P1" => Port(Putter, u32),  
6             "C"  => Port(Getter, u32),  
7             "f"  => Func(FuncHandle::new(|x: *const u32| *x%3 ),  
8         },  
9         rules: [  
10             RuleDef {  
11                 premise: Premise {  
12                     ready_ports: {"P0", "P1", "C"},  
13                     full_mem:    {},  
14                     empty_mem:   {},  
15                 },  
16                 instructions: [  
17                     CreateFromFunc("t0", "f", [Resource("P0")]),  
18                     CreateFromFunc("t1", "f", [Resource("P1")]),  
19                     Check(IsEq([Resource("T0"), Resource("T1")])),  
20                 ],  
21                 movements: { "t0" => {"C"} }  
22             }  
23         ],  
24     }  
25 }
```

Listing 12: Example of another use for the imperative form structure in the Rust language. Here, `&MY_PROTO` accesses the imperative form by reference such that it can instantiate runnable protocol objects repeatedly. Syntax is slightly abbreviated for readability. This protocol forwards $\text{modulo}(P_0, 3)$ to C only if it is equivalent to $\text{modulo}(P_1, 3)$.

Chapter 4

Protocol Runtime

In this section we explore the Rust implementation of Reo-generated protocol objects. Rather than 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 its 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 13 for the simple ‘sync’ protocol which acts as a one-way channel. In this example, the ports are of type `String`.

```
1 Port<String> p0 = new PortWaitNotify<String>();
2 Port<String> p1 = new PortWaitNotify<String>();
3
4 Sender c0 = new Sender();
5 Receiver c1 = new Receiver();
6 Sync c2 = new Sync();
7
8 p0.setProducer(c0); c0.p0 = p0;
9 p0.setConsumer(c2); c2.p0 = p0;
10 p1.setProducer(c2); c2.p1 = p1;
11 p1.setConsumer(c1); c1.p1 = p1;
12
13 new Thread(c0).start();
14 new Thread(c1).start();
15 new Thread(c2).start();
```

Listing 13: 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 components¹. The (simplified) generated `Component` code of the 'sync' protocol from the previous section is shown in Listing 14. 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

¹The 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.

```
1 private static class Sync implements Component {
2     public volatile Port<String> p0, p1;
3
4     private Guard[] guards = new Guard[]{
5         new Guard(){
6             public Boolean guard(){
7                 return (p1.hasGet() && !(p0.peek() == null));
8             },
9         };
10
11     private Command[] commands = new Command[]{
12         new Command(){
13             public void update(){
14                 p1.put(p0.peek());
15                 p0.get();
16             },
17         };
18
19     public void run() {
20         int i = 0;
21         while (true) {
22             if(guards[i].guard()) commands[i].update();
23             i = i==guards.length ? 0 : i+1;
24             synchronized (this) {
25                 while(true) {
26                     if (p1.hasGet() && !(p0.peek() == null)) break;
27                     try {
28                         wait();
29                     } catch (InterruptedException e) { }

---


```

Listing 14: 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.

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 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, ac-

cess 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

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. In this section, we lay out the requirements for Reo-rs. These requirements serve a dual purpose; firstly, it serves to structure the argumentation in sections to follow, primarily linking design choices together by providing over-arching motivations. Secondly, it provides a lens through which to view the work to follow, particularly for helping future readers to identify differences in their requirements such that different decisions can be made.

4.2.1 Requirements Defined

First, we enumerate and name the most vital **functional requirements** for the Reo-rs library. These requirements determine *what* functionality must be available to the end user of protocols and ports, and which safety properties must be preserved.

- R_{correct}^F 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}^F Prevent the protocol from being initialized in an inconsistent state. Prevent port objects from being uninitialized, unsafely accessed in parallel, or incorrectly connected to the protocol.
- R_{ffi}^F 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.

Next, we enumerate the requirements which are *qualitative*. By their nature, they cannot be met absolutely. Rather, they serve to guide the design of the implementation to prioritize the design decisions they affect.

For those associated with run time, Chapter 6 will assess the extent to which these goals are met.

- R_{data}^N 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.
- R_{fast}^N 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.
- R_{end}^N Facilitate the protocol object being destroyed and its resources freed. Minimize the hassle of detecting when a system has terminated.

4.2.2 Requirements Evaluated

TODO

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 11 and 12, 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 11). 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 parameters 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; R_{correct}^F 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 R_{data}^N . 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 ex-

ample), 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.

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

²These libraries are publically available on crates.io.

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.

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 concurrently³. 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` crate⁴ for sending events 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 R_{fast}^N . 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,

³Conceptually 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.

⁴<https://crates.io/crates/mio>

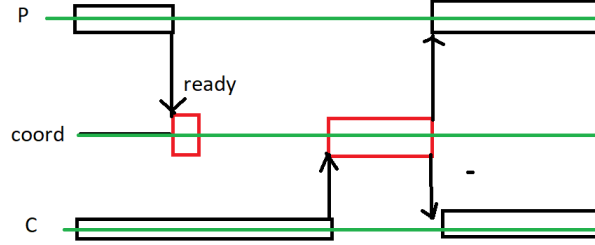


Figure 4.1: TODO

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 $\mathbf{R}_{\text{data}}^N$, 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 from 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 3.5, 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 ex-

plicit. 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 it is immediately apparent by looking at the definition of `Proto` itself, seen in Listing 15. `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, its 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 in 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.

4.3.4 Behavior

This section explains how the data structures of the `Proto` type come to life at runtime to emerge as coordination according to the protocol with

```

1 pub struct Proto {
2     cr: Mutex<ProtoCr>, // accessed by coordinator with lock
3     r: ProtoR, // shared access
4 }
5 struct ProtoR {
6     rules: Vec<Rule>,
7     spaces: Vec<Space>,
8     name_mapping: Map<Name, LocId>,
9     port_info: HashMap<LocId, (IsPutter, TypeInfo)>,
10 }
11 struct ProtoCr {
12     unclaimed: HashSet<LocId>,
13     is_ready: BitSet,
14     memcell_state: BitSet, // presence means mem is FULL
15     allocator: Allocator,
16     ref_counts: HashMap<Ptr, usize>,
17 }

```

Listing 15: 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.

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.

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 R_{fast}^N . To structure our reasoning, we identify the tasks a coordinator performs from the moment it 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*. The task of each getter is to locate the **Space** of this resource, and content with an unknown number of fellow getters to complete the *movement*. All together, this interaction between one resource and N getters is only correct if a number of requirements are satisfied:

1. **the last 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 $N - 1$ 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; sometimes, 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 (`count`, `data`), and is used for shared, reference-counted ownership of `data`. For this type, copying the pointer to the tuple is not sufficient. Cloning must *follow* the pointer and increment `count`.

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.⁵

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*.

⁵Rust 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.

4. No `clone` precedes the move

It is unsafe to *move* a value before or while performing `clone` on the original. Consider again the example of type `Arc` by examining this sequence of events 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, its reference count 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 `count`. `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.

5. Signal-getters prefer cloning

As explained in Section 4.3.1, the API of the `Getter` type offers variants of its `get` operation with slightly different semantics. Of note are those that communicate a lack of desire to actually acquire and return any datum; the getter is interested only in participating in synchrony. We observe that it is wasteful for such a signal-getter to be elected mover in the event the some *other* getter wanted the datum, and by losing the mover-election, was forced to `clone`.

With all of these requirements in mind, getters meet at the `Rendezvous` with fields `count` and `mover`, initialized to `N` and `true` by the coordinator a priori respectively. Listing 16 provides Rust-like pseudocode for this procedure from the perspective of some getter. The intuition behind this solution is the observation that eventually exactly one getter will decrement `count` for the last time (evaluating `count.sub_fetch(1)` to zero). At this moment, regardless of `N` of which getters wanted signals, all clones have been completed. The `Space` itself stores a *reference* to the value over which the getters contend. In the case of a `Putter`-resource, this reference points directly to the putter's stack. In this manner, compute-component threads move values from one another's stacks, without requiring any buffering in the protocol at all. As far as getters are concerned, only during *cleanup* does it matter where the space's value-pointer leads.

Another optimization exists for cases where the data type implements the `Copy` trait, communicating that this type has a trivial destructor, and it is perfectly safe for multiple getters to *copy* the data by replicating its shallowest representation in parallel. This is the case for primitives, and structures composed entirely out of primitives, such as arrays of integers.

```

1  if want_data {
2      let am_mover = mover.swap(false);
3      if am_mover {
4          if count.sub_fetch(1) == 0 { do_move() }
5          else { mover_signal.await() }
6          cleanup(was_moved=true);
7      } else {
8          do_clone();
9          if count.sub_fetch(1) == 0 {
10             // I was the last cloner. All cloners done.
11             mover_signal.release();
12         } }
13 } else { // want signal only
14     if count.sub_fetch(1) == 0 {
15         let am_mover = mover.swap(false);
16         if am_mover { cleanup(was_moved=false) }
17     } else { mover_signal.release() }
18 } }

```

Listing 16: 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.

For these types, each getter knows a priori that every getter will conclude that the type implements [Copy](#), and it can forgo the mover-election business. It is still necessary to elect one getter as the *last* to perform the cleanup (the putter must still be released, or the memory cell must still become ready).

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 $\mathbf{R}_{\text{data}}^N$ and $\mathbf{R}_{\text{fast}}^N$ 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 than 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 move 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

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 extreme: 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 17 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 18 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 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.

```

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 17: ‘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.

```

1 core::ptr::real_drop_in_place:
2     ret
3 example::main:
4     lea    rax, [rip + .L__unnamed_1]
5     ret
6 <u32 as example::PortDatum>::get_num:
7     mov    eax, 45
8     ret
9 .L__unnamed_1:
10    .quad   core::ptr::real_drop_in_place
11    .quad   4
12    .quad   4
13    .quad   <u32 as example::PortDatum>::get_num

```

Listing 18: 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 17. Observe how this function simply returns `.L__unnamed_1`, the vtable of `u32`, included by the Rust compiler.

Chapter 5

Generating Static Governors

A protocol'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 *governors*: 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 19 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_not* 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*.

```
1 fn transform_not(p: Putter<bool>, g: Getter<bool>) {  
2     loop {  
3         let input: bool = g.get();  
4         print!("I saw {}. ", input);  
5         p.put(! input);  
6     } }
```

Listing 19: 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

Thus far, we have reasoned about operations on RBAs that preserve their ability to simulate the non-silent port actions of the original. However, at runtime the protocol's state will follow a *particular* path, which may not be the only one possible. When two distinct paths branch out from the current configuration, which one should the governor follow such that it can enforce the correct actions? Consider the protocol *fifo2* once again, and observe that from type-state `(E,F)`, two rules may be fired next, one firing A and the other firing B. The governor must enforce A if and only if the protocol's state goes down the path for A, and likewise for B. This is an instance of the *consensus* problem; all RBAs in the system must agree on the path taken such that they can proceed in lockstep.

Many synthetic solutions are possible for creating consensus, as we can determine a meta-protocol ahead of time for both governors and protocol to follow such that consensus emerges at runtime. For example, by *ordering* the protocol's rules, and having all parties prioritize rules by ascending order, choice is statically eliminated. Even this simple solution

is deceptively nuanced, as the normalization procedure breaks the 1-to-1 correspondence between the rules of the protocol, and the rules of a governor.

Instead, 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. 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 a natural choice, as it is the only RBA in the system with a complete view of the protocol’s state, and thus can make the choice as a function of its *un-approximated* configuration, as well as the values of all port-putters dynamically.

In terms of implementation, the Rust function for a rule no longer returns a *particular* type-state token, but rather a `StateSet` object representing an indeterminate state-type which will *later* be chosen from the elements of the set. This type is entirely opaque other than a function to codedetermine it. At runtime, *determine* blocks until the governor receives a message from the protocol, communicating a particular choice. For cases where the `StateSet` is a singleton, *determine* simply unwraps the element. While the decision is made at *runtime*, the type-state automaton exists at runtime. To make this possible, the programmer must provide behavior for each *case*, corresponding to elements of the set, one of which will be chosen by *determine*. This use case describes a *sum type*, which is already present in rust as the `enum`, a union type with a set of *variants*. However, every new variant set would necessitate the definition of a new enum. This is impractical, as the number of combinations are large. Instead, we implement our own types which behave and appear to the user much like *anonymous sum types* (which do not currently exist in the rust language). This is achieved by relying once again on Rust’s trait system to encode *lists* in type-space, using nested tuples. *Matching* of these nested tuple types translates to peeling away tuple layers.

5.4.3 Governed Environment

TODO

5.5 Making it Practical

5.5.1 Approximating the RBA

Motivation

TODO

Data Domain Collapse

The approach to generating a type-state automaton from an RBA was given in 5.4.1. A major contributor to the size of these state spaces is the size of the data domain. To proceed 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)`.

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 *ig-*

nore 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 20 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.

```

1 enum E {} // E for "Empty"
2 enum F {} // F for "Full"
3 fn start_state() -> (E,E);
4
5 fn a_to_m0<M>(state: (E,M)) -> (F,M);
6 fn m0_to_m1 (state: (F,E)) -> (E,F);
7 fn m1_to_b<M>(state: (M,F)) -> (M,E);

```

Listing 20: 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.

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,

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.

the protocol interfaces with 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.

```

1 fn a_to_m0<M>(state: (E,M)) -> (F,M) {
2     // A puts
3 }
4 fn m0_to_m1 (state: (F,E)) -> (E,F) {
5     // silent
6 }
7 fn m1_to_b<M>(state: (M,F)) -> (M,E) {
8     // silent
9 }

```

Listing 21: 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 proto-

col is represented as an RBF in Table 5.1 to make the correspondence to the generated governor in Figure 21 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:

- 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 $\text{Act}(r_{last}) \cap I = \{P\}$ and for all rules r in the sequence, $r = r_{last} \vee \text{Act}(r) \cap I = \emptyset$.
- 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.

The final normalization procedure is given in Listing 22 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 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).

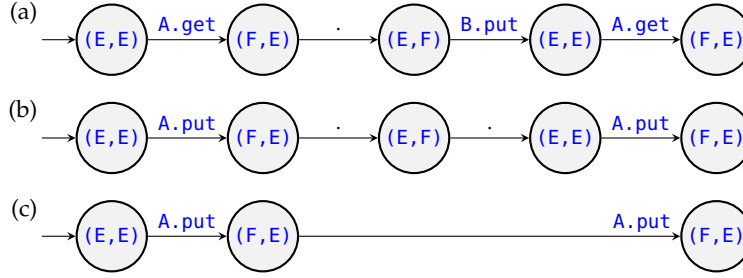


Figure 5.1: Rules being applied to walk three RBAs in lockstep, with time horizontally, showing the (simplified) configurations traversed, and annotating rules by showing which port actions they involve.

(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.

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 $x \cdot x \cdot x \cdot \dots$ 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 $B = (A \setminus \{r\}) \cup \{r \cdot x \mid r \in A \setminus \{x\} \wedge \text{composable}(x, r)\}$ 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 $|P(A)| \geq |P(B)|$. However, the 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.

```

1 fn normalize(mut rules: Set<Rule>) -> Set<Rule> {
2     let (mut silents, mut not_silents) = rules.partition_by(Rule::is_silent);
3     while silents.not_empty() {
4         let removing: Rule = silents.remove();
5         if removing.changes_configuration() {
6             for r in silents.iter() {
7                 if let Some(c) = removing.try_compose_with(r) {
8                     silents.insert(c);
9                 }
10            }
11            for r in not_silents.iter() {
12                if let Some(c) = removing.try_compose_with(r) {
13                    not_silents.insert(c);
14                }
15            }
16        }
17    }
18    return not_silents;
19 }

```

Listing 22: 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.

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 Unknown Memory State

5.5.3 Match Syntax

(TODO) 1. compare enum

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.

Chapter 6

Benchmarking

- 6.1 Goal
- 6.2 Experimental Setup
- 6.3 Results
- 6.4 Observations

Part III

Reflection

Discussion

7.1 Future Work

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

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

7.1.5 Further Runtime Optimization

Didn't implement everything for various reasons 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 achieved just by reorganizing your movements. more examples: pruning check subtrees of tautologies and contradictions.

7.2 Conclusion

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.
- [exo] Exotic sizes.
- [JA12] Sung-Shik TQ Jongmans and Farhad Arbab. Overview of thirty semantic formalisms for reo. *Scientific Annals of Computer Science*, 22(1), 2012.
- [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.
- [Wal05] David Walker. Substructural type systems. *Advanced Topics in Types and Programming Languages*, pages 3–44, 2005.