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.

Safety and Performance in Generated Coordination Code

Christopher Esterhuyse
(ID: 2553295)

supervisors

Vrije Universiteit Amsterdam
dr. J. ENDRULLIS

Centrum Wiskunde & Informatica
prof. dr. F. ARBAB

August 16, 2019

Abstract

Reo is able to act as a domain specific language by compiling a high-level protocol specification to coordination glue code in a target general purpose language. In the case of Java, these generated programs take advantage of optimization opportunities, but struggle to preserve Reo's semantics. In this work, we extend the Reo compiler to support the Rust language. We show that the resulting program implements several existing and novel performance optimizations, whilst relying on Rust's expressive type system to preserve meaningful safety and liveness properties at compile time. Our design also facilitates the reconfiguration of the protocol at runtime, but still achieve performance in the order of handcrafted programs for non-trivial protocols.

Acknowledgments

I would like to express my gratitude to everyone at the CWI in the Formal Methods group. I am particularly grateful to Hans-Dieter Hiep, Sung-Shik Jongmans, and Roy Overbeek, with whom I worked the most. Everyone was incredibly generous with their time, and very willing to share their insights. I'm grateful to both of my supervisors, Farhad Arbab and Joerg Endrullis, who have made my thesis project as pleasant as possible by being agreeable, helpful and making themselves available to the best of their abilities. I am thankful for the university friends I saw regularly, who also tended to be my collaborators; Zakarias Nordfält-Laws, Dalia Papuc and various other members of the PDCS track were ever-involved in my projects. I'd also like to thank the lecturers in the theoretical computer science department at the VU, whose courses were enjoyable and informative without exception. Special thanks to course coordinator Wan Fokkink for continuously helping me with administrative troubles, and being an excellent teacher to boot. Old friends Peter Atkinson, Berend Baas, and Henry Ehlers have contributed in some way or another with their input and support. Many thanks to my family, with particular emphasis on my parents, aunts, uncles, cousins, and grandmothers, who have continuously checked in on my well-being and supported me financially from across the world for all these years. Lastly, special thanks to my girlfriend, Ronja Duwe, for being my most continuous and immediate supporter in all things.

Contents

List of Figures

Chapter 1

Introduction

Traditional, sequential programming has been changing for decades. Over time, languages acquired more and more tools to manage the level of abstraction, such that programs were of higher quality, and with a lower cost to develop [Sha84]. This trend continues to this day. For example, conventionally imperative languages such as Java and C++ have since added functional features, such as closures, to capitalize on their brevity and lack of side effects. Concurrent programming has undergone a similar process. Various paradigms have emerged to offer their own solutions to managing abstraction. *Coordination languages* attempt to reduce the coupling between the logical coordination of a program, and its implementation. They work by isolating that which makes concurrency distinct from sequential programming: the coordination of actions into interactions. The means by which this is achieved varies. Linda is a well-known example of a coordination language which abstracts away the implementation of coordination actions. Programs are written in terms of read and write operations over a global namespace of tuple variables. In this manner, coordination actions occupy a higher level of abstraction, and are thus more terse, simple and reusable [Gel85].

Reo is a coordination language developed at the CWI in Amsterdam. As with Linda, Reo is able to augment another general-purpose programming language by allowing a more abstract expression of coordination. Unlike Linda, Reo facilitates the 'extraction' of a program's coordination logic to a self-contained *protocol specification*. Reo does away with action-centricism; protocols constrain the ways in which data is permitted to flow through a system of graphical nodes, i.e., protocols define their permitted interactions [Arb05]. The specification is translated by the Reo compiler to a the target language. The result is a *protocol object*, a data structure which acts as a coordination medium for the system's actors. At runtime, components send and receive data through *ports*

managed by the protocol. Components can rely on the protocol to coordinate their actions such that the system at large behaves as specified.

In this work, we aim to extend the Reo compiler and its related tooling such that it can be integrated into the development pipelines of programs in systems languages such as C, C++ and Rust. Chapter 3 details the development of a Rust language target for the Reo compiler, whose outputs are interoperable with these languages. We discuss the various representation changes that a Reo protocol specification must undergo before valid Rust can be emitted.

Reo has intuitive *value-passing* semantics, such that data moving through ports is transferred in its entirety, without interference from the actions of other ports. In the realm of shared memory, this has an undesirable naïve implementation which must move the contents of large values repeatedly. Reo-generated Java code makes use of an optimization whereby values are kept in place, and their references are moved through ports instead. Currently, these Java programs are circumstantially subject to data races as a result of applying this optimization. This violates Reo's semantics; actors cannot be certain that a value acquired through a local port is not being accessed by someone else concurrently. Chapter 4 explains how our Reo-generated Rust combines the language's systems-level memory management with the explicit protocol descriptions of Reo to perform reference-passing such that Reo's semantics are preserved. Other optimizations are also explored. Chapter 5 provides an evaluation of our implementation at runtime, comparing it to Reo-generated Java, handcrafted Rust, and breaking down its performance in response to properties of the input Reo specification.

Reo's protocols specify which behaviors systems are and are not permitted to exhibit. At runtime, protocol objects organize actions on their boundary ports into permitted interactions. Actions do not succeed until they are permitted. In this manner, port actions at the wrong time will not cause the program to deviate from the protocol, but may cause a loss of program liveness if the operation blocks, waiting for an interaction that never comes. For such cases, Chapter 6 explores the design of *governors*, structures which allow the programmer to statically verify that local actions will not impede program liveness.

Chapter 2

Background

To begin, we introduce some terminology and background information to be used in the chapters to follow.

2.1 Reo

Reo is a high-level language for specifying protocols. In this section, we touch on the motivation behind Reo's development, and explain how it is used. The 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 coordinating interactions within executable programs.

2.1.1 Motivation and Purpose

Traditionally, coordination programming is approached much like sequential programming, by laying out program logic as sequences of actions, tracing the path of one control-flow at a time. In such programs, concurrency and parallelism are emergent properties; interactions are not represented explicitly, instead they must be derived from actions sprinkled throughout the program [Arb11]. The more complex the program, the more difficult it is for programmers to make sense of this coordination; actions that contribute to interactions become entangled with local computation. Depending on the language, there may also be a large conceptual gap between the coordination's underlying concept and the granular operations the program uses to implement it. The larger the gap, the more tightly-integrated they become, obfuscating the concept within implementation specifics and making it difficult to change one without changing the other.

Over time, various tools and languages have emerged, each with their own way to raise the level of abstraction. For example, various process calculi can represent coordination actions symbolically, such that it is more easily understood and manipulated by humans and machines alike [Arb11]. Chapel is an example of a general-purpose programming language with a focus on parallelism, offering a (more traditional) 'local view' mode for concurrent procedures, but also a 'global view' mode for data-parallelism that reads much like a sequential program [CCZ07]. *Coordination languages* are primarily oriented toward expressing the means by which concurrent systems coordinate the actions of individual actors into interactions. Their domain-specific nature often makes them suitable for augmenting other general-purpose languages by increasing their expressive power. *Linda* is said to be the first of its kind [Wel05]. In Linda, communication between actors takes the form of simple manipulations of tuples in a global value store it calls the *tuple space*. Programmers can thereby separate their concerns of *what* coordination logic is implemented from *how* it is implemented [GC92].

Reo is a coordination language founded by the Formal Methods group at the CWI in Amsterdam. It has much in common with Linda, but attempts to do away with what it considers to be a vestige of the world of sequential programming: *action-centricism*, making explicit only individual actions, relegating interactions to a derived concept. Reo represents a program's coordination logic in terms of interactions between participants explicitly; actions of participants can be derived later [Arb11]. As with Linda, Reo is a domain-specific language, able to take over only the coordination work of a program written in another language. Where Linda embeds tuple operations into its host language, Reo embeds ports; program logic is interspersed with (logical) message-passing port operations. Unlike their Linda equivalents, ports are local structures which are entirely oblivious to their environment, exchanging data with an unknown peer in an unknown system context. Actions serve only to drive local computation tasks. Building a complex system is a matter of *composing* these modular *components* by interfacing their ports. Reo's main purpose is to provide an interaction-centric, declarative language for the specification of a protocol component which acts as an intermediary between other components in the system. The *Reo compiler* sees to the translation from the Reo specification to the equivalent executable code in the target language [Arb11].

Reo's approach to programming comes with two advantages: (1) Programmers may view and manipulate the coordination logic of their program via the Reo specification, whose nature is well suited for tasks such as verification, and (2) protocol and compute components are loosely coupled to their environment, making it possible to reuse them in different programs, and to understand or alter their behavior in isolation.

2.1.2 Language

Reo is a graphical language which represents a protocol as a *connector*, a multigraph which defines the relationships between its nodes [Arb11]. Nodes represent logical 'locations' which may observe a single datum at a time. Ultimately, a connector's relations boil down to Reo's underlying semantics, of which there are several to choose from (explained in Section 2.1.3 to follow). They have in common that they constrain the synchronous observations of data at nodes. For example, a connector may enforce that nodes A and B always observe the same data element. This is known as the `sync` channel, a primitive sufficiently ubiquitous to be represented simply as an arrow ($\longrightarrow$) in Reo's graphical syntax. In this fashion, a connector specifies the ways in which data is permitted to synchronously 'flow' through its nodes. The literature often uses terms such as 'circuit' to evoke appropriate metaphors. As with electricity or fluid pressure, flow propagates forwards, and blockages propagate backwards.

A tenet of Reo's design philosophy is compositionality (and consequently, modularity). Rather than defining each and every connector anew in terms of the underlying semantics, all but the simplest primitive connectors are built from the composition of others [Arb11]. At each level of this hierarchy, the connector exposes a subset of its nodes to the connector above, wherein they are called *ports*. The set of a connector's ports is called its *interface*, beyond which is its *environment*. A connector with complex semantics emerges when the ports of its constituent connectors are linked together.

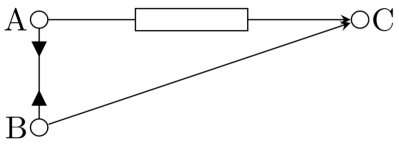
As Reo's various metaphors suggest, we usually think of data as flowing in a particular direction. To this end, ports are usually defined along with an annotation of their *orientation*: *input* ports accept data from the environment (moving downward in the connector hierarchry), and *output* ports are the reverse. With these orientations, port interactions orient the direction of their flow from *source* nodes to *sinks*. Connectors can define internal nodes of *mixed* orientation by sources to sinks in various configurations. In this way, connectors can specify complex relationships in their *hidden* (internal) nodes.

Some Reo primitives (and consequently, some connectors that incorporate them) define relationships in terms of variables whose states can persist into the future. In this way, Reo can model asynchronous data flow also, breaking up contiguous networks of synchrony, and constraining future observations in terms of its changing state. The primitive `fifo1` connector is canonical for this purpose; in observing some datum X on its single input port, an empty connector becomes full, whereafter the next observation on its single output port must be X, making it empty once again. Despite its simplicity, `fifo1` is sufficient for the expression of arbitrary asynchronous events and for encoding

the persistence of any state.[1] These memory variables is explored further in Section 2.1.3 to follow.

Conventionally, Reo is expressed using graphical syntax, representing nodes as one would expect. Building a connector graphically involves drawing connections between nodes as edges for the usual binary primitives (sync, fifo1, etc.), or as 'black boxes', exposing ports at their boundaries. In this work, we focus on Reo's textual syntax [DA18b] instead, as it makes for a more practical interface between human and compiler. In this context, the language is fundamentally the same, but relies on textual identifiers for nodes and connectors as is typical for general-purpose programming languages. Listing 2.1 demonstrates both methods using a canonical example connector: alternator2.

More complete descriptions of how Reo works, how its nodes behave under composition and details about the canonical Reo primitive connectors are available elsewhere [ABRS04, Arb05, Arb11].




```
1 alternator2(A?, B?, C!){
2   syncdrain(A, B)
3   fifo1(A, C)
4   sync(B, C)
5 }
```

Figure 2.1: Reo specification of the alternator2 connector with input ports {A, B}, and output port C using graphical and textual syntax. Data flows to C from A and B in an alternating fashion. Connector is distinct from a sequencer by B's transmission being synchronous with all three ports. Figure is taken from [ZHLS19].

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 et a. 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,

[1] The initially-full variant of fifo1 allows Reo to express connectors initialized with any imaginable initial state.

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

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

annotating each event in the sequence with an increasing *time stamp*. A TDS is defined by some tuple $(\mathbb{N} \mapsto \mathbb{R}, \mathbb{N} \mapsto D)$, or equivalently, $\mathbb{N} \mapsto (\mathbb{R}, D)$, with the requirement that time stamps 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 to *silent behavior*. In this work, we introduce *trace tables* as a term for both the tabular visualization, and to a program trace as a set of named TDS's. The runs of finite programs can be simulated either by bounding the tables (constraining the TDS domain to be finite), or by simulating finite behavior as infinite by extending the 'end' forevermore with silent behavior. Table 2.1 gives an example of a trace table for some program with two named variables.

One of it's earlier coalegebraic models represented Reo connectors as **stream constraints** (SC) over such TDS tables in which variables are ports [Arb04]. Here, constraints are usually defined in first-order temporal logic, which allows the discrimination of streams according to their values both now and arbitrarily far into the future.[2] 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.

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

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 *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 $\{0, 1\}$, shown in Figure 2.2. 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.

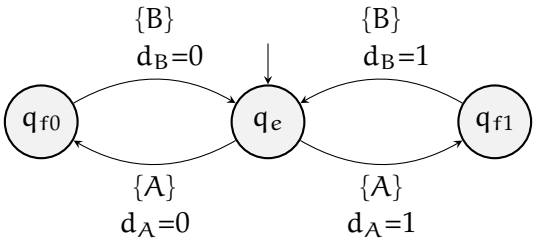


Figure 2.2: CA for the `fifo1` protocol with ports A and B sharing data domain $\{0, 1\}$.

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) [ABdBR07]. Figure 2.3 revisits the `fifo1` protocol from before. With this extension, the task of persistently storing the value of A in 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 protocols. Figure 2.4 shows an example of such a composition, producing `fifo2` by compos-

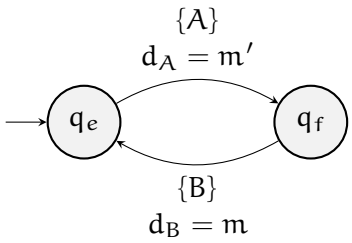


Figure 2.3: CA with memory cell m for Reo connector `fifo1` with arbitrary data domain D common to ports A and B. Two states are used to track to enforce alternation between filling and emptying m.

ing `fifo1` with itself (by connecting A of one to B of the other). 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 spaces. 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 verbosely. 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 by checking the contents of the table.

CA transitions have another interpretation in a context where the present is decided, but the future is not. If we have influence over the future values of memory cells, we are able to influence the walks through the automaton. 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 we ensure is satisfied. As an example, consider a transition with data constraint $m = d_A \wedge m' = d_B$; as long as $m = d_A$ holds now, we are able to take the transition and assign d_B to m afterward. Figure 2.4 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 [DA18a]. Figure 2.5 models the `fifo1` connector once again, this time as an RBA. Aside from the added expressiveness, RBAs benefit from be-

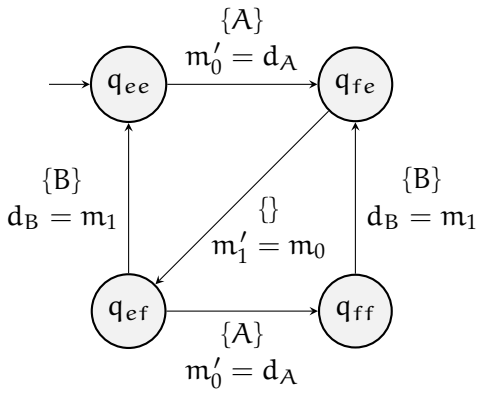


Figure 2.4: CA with memory cells m_0 and m_1 for the fifo2 connector with an arbitrary data domain for ports A and B. Transitions are spread over the state space such that the automaton's structure results in the 'first-in, first-out' behavior of the memory cells in series.

ing cheaper to compose. As the state space is degenerate, RBAs may be easily reinterpreted 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.

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 2.6 shows how how this occurs for the fifo2 connector. Where the CA in Figure 2.4 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 proto-

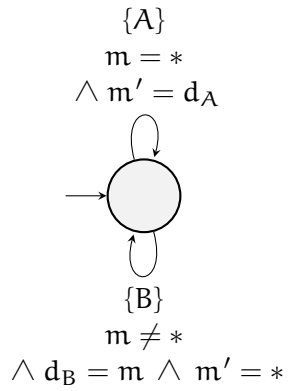


Figure 2.5: RBA of the `fifo1` connector for an arbitrary data domain common to ports A and B. Memory cell m is used both to buffer A's value, and as part of the data constraint on both transitions for emptying and filling the cell to ensure these interactions are always interleaved. Data constraints are formulated for readability such that the 'guard' and 'assignment' conjuncts are line-separated.

cols are implemented, the discriminants of such types are not distinguished statically. For example, the C language lacks a way to statically enforce a that function `void foo(int x)` is only invoked when x is prime. Instead, checks at runtime are used to specialize behavior. On the other hand, the state space is simple enough to afford a practical translation into the structure of the 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 by which state the corresponding CA is in at runtime. 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

An ecosystem of tooling has emerged around the Reo language, each exploiting Reo's explicit connector specifications for some purpose or another, ranging from verification to code generation. An overview of these tools can be seen on

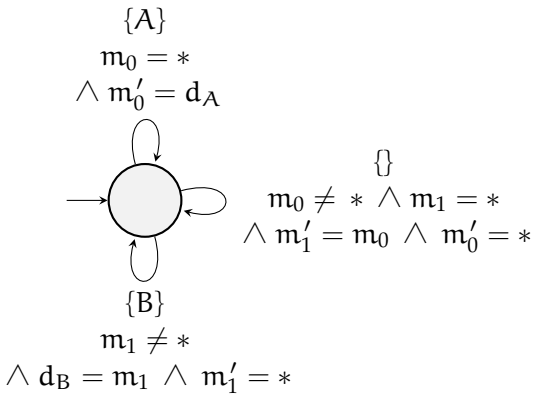


Figure 2.6: RBA of the `fifo2` connector for an arbitrary data domain common to ports A and B. Memory cells m_0 and m_1 are drained by B in the order they are filled by A, and have a capacity of 2 elements. Data constraints are formulated for readability such that the 'guard' and 'assignment' conjuncts are line-separated.

Reo's website[3]. In this work, we are particularly interested in the *Reo compiler*. Previously, code generation was a feature of Reo's *Extensible Coordination Tools* platform plugins for the Eclipse IDE. Since then, the compiler has become a standalone program, and has added several compilation targets as the result of ongoing research and development [JSS+12, JSA15, DA18a].

Given a textual Reo specification, the Reo compiler generates and emits a *protocol object* with runtime behavior corresponding with the input specification. The output is in a *target language* selected by the user. At time of writing, languages Java, Promela and Maude are supported. C11 is available also, but has been deprecated. Once parsed, the compiler translates the Reo-language representation into that of Rule-based form, involving operations hiding, projection, merging and so on. For our purposes, it suffices to trust that the compiler performs this transformation to our satisfaction, resulting in a representation with the intended semantics. Detailed information about this process is available in the literature [Arb04, Arb05, DA18a]. Accordingly, the textual Reo language has support for the definition of primitives in such models directly [DA18b]. Listing 2 demonstrates this with the primitive `fifo1` connector.

In this work, we concentrate on Reo's use for the generation of glue code between components in a target language, such that the program adheres to the associated Reo specification at runtime. The Reo compiler is responsible for this translation quickly and reliably. Programmers are able to exploit the

[3] http://reo.project.cwi.nl

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

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

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

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

correspondence between program and specification by using the representation that best fits the task at hand; it runs with all the characteristics of the target language, but programmers may manipulate or verify its coordination properties via its specification.

2.2 Target Languages

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

2.2.1 Affine Type Systems

In a nutshell, affine types are characterized by modeling values as finite resources, operations on which *consume* them. This notion of 'affinity' has its roots in logic. In a type-theoretic proof system, one can attempt to derive some *judgment* $\Gamma \vdash t : \tau$ with *statements* assigning *types* to *terms*; here term t is stated to have type τ under *context* Γ. A context is simply a list of statements, which can be thought to correspond with assumptions, or the premise [NG14]. The judgement holds if once can complete a proof by selecting and applying *rules* to arrive at the judgements. In practice, this is often done backwards: starting with the judgment, the proof is completed by applying rules until no 'dangling' judgments remain. Different systems can prove different judgements, as they may differ in the rules they permit. Consequently, rules characterize the system. As an example, simply-typed lambda calculus has a type derivation rule for abstraction, substitution and application. Depending on the type system, *structural rules* may additionally be provided for manipulating ('massaging') the context such that other rules may be applied. Consider the following structural rules:

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

In order of their appearance, var terminates proof branches by identifying tautologies: statements appearing directly in the context. $weaken$ allows us to arbitrarily grow our context, weakening the strength of the formula as a whole by adding to our assumptions. $contraction$ allows us to treat statements as if they are idempotent; we are allowed to discard duplicate statements at will.

Depending on the proof system, such structural rules may be necessary to make a formula provable [NG14]. For example, $weaken$ rule is required to prove $A, t : \tau \vdash t : \tau$. var cannot be directly applied, because A is 'in the way'. Substructural rules also help to characterize systems. For example, the ability to arbitrary replicate, discard and rearrange context expressions characterizes a type system whose context is 'set-like'; in such a system, the order of statements in the context has no impact on whether a formula can be proven.

Affine type systems are characterized by the absence of $contraction$ rule. Proofs cannot replicate statements at will, and thus they are a finite *resource* in the proof, consumed by use in rules [Wal05]. As type systems do in general, type affinity excludes some programs from being expressed. In the context of programming languages, why would we want this? Of course, conventional computer 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 reinterpreting 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, making a case for the feasibility of statically-typed local variables. These limitations are primarily for programmers and not for the program; they facilitate strictly enforcing on one another (or oneself) what would otherwise only be good programming practices.

2.2.2 The Rust Language

Rust is an imperative, general-purpose, systems programming language most similar to C++, even (mostly) sharing its C-style syntax. What sets it 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 deallocations (called `drop` in Rust) 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 relocation 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 the type's `drop` function is invoked to destroy it. Along with the RAII ('resource acquisition is initialization') pattern popularized by C++, programmers can rest assured that their resources are created and destroyed implicitly, at predictable moments.

```
1 struct Foo { val: u32 };
2 fn func(y: Foo) { // this function moves its parameter into its scope (by value).
3 }                 // `y` goes out of scope and is (trivially) dropped.
4 fn main(){
5     let x = Foo { val: 5 };
6     func(x);      // Ok. `x` is moved into `func`.
7     func(x);      // Error! x is used after move.
8 }
```

Listing 3: Type `Foo` is affine. On line 6, `x` is consumed by being moved into function `func`, transferring ownership to the binding `y`. Afterward, access of `x` is invalid, and so line 7 raises an error.

Borrowing

On its 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* (also called *borrows*) to values, which are indistinguishable from pointers at runtime, but have special static semantics. These references are new objects, not representing a transference of ownership as moves do. Listing 4 demonstrates the example from before, but now passing `x` by reference (as `&x`) 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 outlive 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 inferred within function bodies. This has an important consequence; the compiler does cross the boundaries between functions to interpret their relationship.

```
1  struct Foo { val: u32 };
2  fn func(y: &Foo) { // this function borrows some `Foo` structure by reference.
3  }                  // dropping returns the `borrowed' ownership to the caller.
4  fn main(){
5    let x = Foo { val: 5 };
6    func(&x);      // &x is created in-line. x remains in place.
7    func(&x);      // another &x is created. x remains in place.
8  }
```

Listing 4: `Foo` is an affine resource. New references to `x` are created and sent into `func`, temporarily lending ownership to the function, but retrieving it after it returns. Rust's borrow checker ensures that a borrow does not outlive its owner.

For some types there is no practical reason to enforce affinity. This is usually the case for primitives such as integers. Such types must be explicitly marked by implementing the (trivial) `Copy` trait, communicating to the compiler that this type can be *copied* in circumstances where affine types are moved, creating a new independently-owned value. Copy-types behave in ways familiar to C and C++ programmers. For example, Listing 3 from before would compile if `Foo` was marked with `Copy`.

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. 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 coexisting. 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 only sreaders coexist, but if one writer exists, it must have exclusive access.

Traits and Generics

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 function declarations for implementors to define. Unlike Java, Rust traits say nothing about fields and data, thus describing only their behavior. Effectively, they play the role of Java's interfaces, and also C's header files.

In Rust, structures and functions may be defined in terms of *generic types* (also called *type parameters*) similarly to those in C++. In a generic context, only operations that may be applied to any suitable type may be applied. For example any generic type `T` may be borrowed or moved. Move behavior becomes available if one *constrains* the generic type by expressing a requirement that implements some trait. For example, Rust can be certain that some generic `T` implements the method `clone` by constraining it with the requirement: `T: Clone`. Unlike Java interfaces, it is common to constrain a generic type with multiple traits; as they define only behavior, their trait implementations are orthogonal. In this manner, generic code can be written such that it can be applied to exactly the types which possess the required functionality. Listing 5 gives an example of how traits are defined, and used.

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-

```rust
pub fn something<T>(val: &T) where T: PrintsBool {
    val.greet()
}
trait Greets {
    fn greet(&self);
}
impl Greets for String {
    fn greet(&self) {
        println!("hello!")
}   }
```

Listing 5: Definition of the `Greet` trait, whose implementation requires that `greet` be defined to match the specified declaration. Function `something` can invoke this function, dispite not knowing concretely which type is chosen for generic type `T`.

by-case basis, generating binary specialized for that type as if there were no generic at all. Static dispatch is used in C++ templates, and opted-out with the keyword `virtual`. This is Rust's second option: *dynamic dispatch* (also called virtual functions or late binding) where the generic function exists as only one instance, and all of the specialized operations on the generic type are resolved to concrete functions at runtime by traversing a layer of indirection: functions are 'looked up' in a virtual function table (vtable). Java uses such virtualization 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 for these dynamic objects. Concretely, some generic object which is known only to implement trait `Trait` is represented as a pair of pointers; the first pointing to the actual object' data, and the second pointing to a dense structure of meta-data and function pointers for the methods of `Trait`, usually embedded into the text section of the binary by the Rust compiler itself. Both methods of dispatch are exemplified in Listing 6, demonstrating how static dispatch must propagate generics to the caller for resolution to concrete types at compile time, while one function using dynamic dispatch is able to handle any virtualized types by resolving their methods at runtime.

Rust uses traits for just about everything. Some traits are defined in the standard library, and have a degree of 'first class' status by having special meaning when used in combination with the language's syntax. For example, `Not` is a trait that defines a single function, `not`, which is invoked when the type is negated using the usual exclamation syntax, i.e., `!true`. Some traits have no associated functions, instead exist for the purpose of communicating

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

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

information to the compiler. Seen before, `Copy` is a trait which disables the Rust compiler's checks of a value's affinity. As another example, `Drop` allows the programmer to specify any *additional* behavior when an instance of the type is dropped, allowing one to implement arbitrary *destructors* similar to those of C++. Several standard traits exist for behaviors which are frequently useful. As a result, it is common to encounter unfamiliar types which implement familiar traits. For example, it is considered good practice to implement (or derive the implementation of) the `Debug` trait, which allow your type to be printed with the standard format strings.

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 occupied by data 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 'vari-

ant' can be reflected at runtime using some explicit operations (`ininstance` and `instanceof` respectively). C takes the approach fitting the language's philosophy; `union` types represent any one of its constituents but the program is at the mercy of the programmer to interact with it as the correct variant as they see fit. Rust's solution is similar to C's unions, but its focus on safety required the use of *tags*; this makes explicit which variant is currently in use. At runtime, an enum's variant can be discriminated by explicitly *pattern-matching*. Matches are most similar to *switch statements* in Java or C, but the *arms* are guarded by *patterns*, enabling the conditional de-structuring of enums and so on.

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 fields of one 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 similar to Java's `Error`.

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

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

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

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

2.2.3 The Type-State Pattern

The *state machine* pattern refers to the practice of explicitly checking for or distinguishing transitions between and requirements of states in a stateful object.[4] Usually, these states are distinguished in the data domain of one or more types. Even Rust's lowly `Option` type can be viewed as a small state machine as soon as some condition statement specializes operations performed with it. Although its uses are ubiquitous in application development in general, this pattern is particularly useful for those for which the added ability to manage complexity is necessary: video games, for example [Nys14].

As the name suggests, the *type state* pattern is an instance of the state pattern, characterized by encoding states as types in particular. Usually, the embedding in types in particular differs only in that, unlike two values of the same type, two distinct types are checked equality (or otherwise related meaningfully) at *compile time*. A common approach is to instantiate one of the state types at a time. We primarily concentrate on state types which do not contain any data (i.e, they have a trivial data domain). We refer to these objects as *tokens*, evoking a comparison with coins to suggest their small size and role as a sort of 'currency'; their role is to facilitate other things.

Type state automata are useful for encoding stateful control flow in a form the compiler will be able to understand and enforce. 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 type-state 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 example above demonstrates some utility, but a language such as C has no fundamental way to prevent the programmer from reusing values, including state tokens. 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 malformed expressions, and perhaps this is enough. However,

[4] Usually, we ignore program termination. Equivalently, this pattern only allows one to describe automata in which every 'useful' state reaches some final 'terminated' state.

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.

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

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

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

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

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

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`. In this way, it is possible to ensure that the program's control flow corresponds with a walk through the automaton.

2.2.4 Proof-of-Work Pattern

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

There is no way to create true relevance or linearity in user space of an arbitrary affine type system; any program which preserves affinity is able to exit at any time without losing affinity. How are we able to enforce a behavior if it is correct to exit at any time? *Proof-of-work* is a special case of the type-state pattern which allows the expression of a relevant type under the assumption that the program continues its normal flow; i.e., 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. Effectively, we use a return type to model relevance.

Realistic languages have many tools for constraining what users may access, such as Java's visibility keywords. 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 types with user-inaccessible fields. This prevents the user from creating instances of the type themselves, forcing them to acquire them by means we dictate. We are able to prevent *anyone* from creating instances of a type by using *empty enums*, which have no safe means of instantiation [Gor].

Consider the following illustrative scenario: We wish to yield control flow to a user-provided function. Within, the user is allowed to do whatever they wish, but we require them to invoke `fulfill` exactly once (which corresponds to 'consuming `R`'). How can we express this in terms the compiler will enforce? Listing 10 demonstrates a possible implementation (omitting all but the essence of 'our' side of the implementation). The user's code would then be permitted

to invoke `main` with their own choice of callback function pointer. Our means of control is the interplay between dictating both (a) the signature of the callback function and (b) prohibiting the user from constructing or replicating `Promise` or `Fulfill` objects in their own code.

```
struct Promise;
struct Fulfilled;

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

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

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

Chapter 3

Protocol Translation

The task of the Reo compiler is to translate a Reo protocol specification into the target language. The resulting code must interface with other components written in the language such that, at runtime, the resulting system behaves as specified. In this chapter, we explain how we extend the Reo compiler to support the Rust language target.

Section 3.1 examines the task of the backend, the translation from the Reo compiler's internal representation ('RIR') to an executable protocol object in the target language. This task is broken down into smaller, more specialized subtasks. Section 3.1.2 organizes these subtasks into a pipeline that sees the translation through from Reo specification to executable Rust protocol object. To bridge the gap between Reo and Rust compilers, Section 3.2 defines *imperative form* ('IF') as a novel representation of a protocol's behavior in a manner conducive to imperative languages, but not yet fine-grained or inherently coupled to one language in particular. Section 3.3 goes into detail about the implementation of the translation pipeline by explaining how the previously-defined subtasks are completed in stages. This includes an example of the output of the Reo compiler: Rust source code whose contents are primarily a Rust-embedded representation of IF.

3.1 Structuring the Translation Process

In this section, we explore the nature of the task which characterizes the Rust backend for the Reo compiler. Section 3.1.1 breaks the problem down into simpler subtasks, and explains their relationship to one another and how they apply in the case of a different target language. Section 3.1.2 explains how

these tasks are organized by defining their place within a translation pipeline. The implementation of the pipeline itself is given in Section 3.3.

3.1.1 Translation Subtasks

The Reo compiler's frontend parses its input in the Reo language with textual syntax. It performs significant transformations on its internal representation before arriving at what we call RIR. Everything that follows is the task of the backend, transforming it further until code in the target language can be emitted. Reo and Rust differ significantly in how they represent work. Accordingly, a protocol expressed in the former must be transformed significantly before it can be expressed in the latter. In this section, this task is decomposed into *subtasks* which serve a dual purpose (1) smaller tasks are more easily understood, and help to characterize Reo and Rust by identifying their differences, and (2) only isolated subtasks can be separated, allowing the entire task to be performed in stages, as the *pipeline*. Here, we explain how the pipeline is structured; the implementation of the subtasks is left to Section 3.3.

Input Representation

RIR embodies the completion of several of the operations on Reo connectors described in the literature [BSAR06, DA18a]; namely, *composition*, *merging*, *hiding*, and so on. For our purposes, it suffices to say that RIR is presented in a form corresponding closely to an RBA, one of the semantic models described in Section 2.1.3. RIR is self-contained, and defines a list of *rules* which correspond 1-to-1 to interactions between the protocol's ports. It captures the intuition of an RBA as they are usually understood in an imperative context; rules are subdivided into parts *guard* and *assignment*. Helpfully, RIR presents the latter not as a monolithic formula, but rather as a mapping from identifier to `Term` objects. In our imperative context, terms can be understood as expressions to be evaluated at runtime. For example, `True` is a boolean type term, while `Port("A")` may be understood as the value put by port A.

Subtasks Involving Actions and Data Types

Reo specifications represent connectors declaratively as relations between ports. They are thus well suited to reasoning about the protocol's properties. In contrast, our target imperative languages such as Java and Rust represent computation such that it corresponds more closely to machine instructions; they are *imperative*, laying out sequences of actions which together emerge as interaction at runtime. Where interactions in the former can be oriented around the synchronous observations of port values, interactions of the latter must be

expressed as sequences of actions, laid out over time. RIR is somewhere in between. Per RIR rule, the guard is distinguished from the assignment, suggesting a coarse-grained ordering of operations: the guard is evaluated before values are assigned. This is a step in the right direction, but requires further transformation before it can correspond to executable Rust. For example, RIR is able to disambiguate the order in which memory cells are read and written to by implying an ordering by annotating their identifiers with qualifiers comparable to those in temporal logic (in both syntax and semantics). For example, the assignment corresponding to $m = A \wedge m' = B$ does not explicitly express an ordering, but nevertheless implies it by annotating m with a qualifier to represent its 'next' value. Rust's imperative nature requires that operations on values occur in the order of their appearance in the program's control flow, as this determines the order in which they will be executed (Rust compiler optimizations aside).

Java, Rust and Reo have in common that they are strongly-typed languages. In the broader sense, 'type' describes the classification of just about everything in Reo, including connectors and primitives. The Reo compiler's internals perform transformations that handle the majority of what could be considered 'type checking'. The only exceptions are (1) the data types of each port, determining the types of values they transmit, and (2) the types of functions applied to values within the channel, which are expressed as relations between an identifier and a list of arguments (each expressed as `Term`). For the sake of programmer ergonomics, Reo permits the data types of ports to be omitted, such that they can later be derived in context. As this task is not performed by the Reo compiler's frontend, it becomes the responsibility of the backend to resolve these types. In compiling to Rust, the backend must take care that these types adhere to Rust's rules for types as well as Reo's. For imperative languages without types, it suffices to represent them with a universal type, e.g., `Object` in Java.

Subtasks for Abstract and Concrete Targets

Regardless of any intermediate representation, protocols must ultimately be emitted in the target language at the required level of specificity. Imperative languages place the burden of defining *how* their work is performed squarely on the shoulders on the programmer. For example, where a declarative language might not distinguish *merge sort* from *bubble sort*, Rust certainly does; a Rust program's operations on variables correspond (relatively) closely to machine instructions as they will be executed at runtime. This is also true in the case of our problem; what is implied in Reo must be made explicit in Rust. This includes the initialization of system resources, operations on concurrency

primitives, and all the minutia necessary to implement the optimizations described in Section 4.3.4.

We observe that one can perform meaningful protocol translation work, resulting in its expression in imperative manner without committing to a particular language's minutia. On the other hand, an abstract specification may be made concrete by translating it into a particular language. We distinguish these notions by differentiating between specification and implementation.[1]

It is beneficial to recognize that a significant portion of the translation from Reo to Rust would be shared by the same procedure to a similar language; the more similar language, the more we can expect their respective transformations to have in common. For example, despite their significant differences, Rust and Java are more similar to each other than they are to Reo. We attempt to generalize Rust, Java and languages like them in accordance with common terminology. We characterize *imperative* languages by a need to make explicit its *control flow* by imposing a total order on its actions. Recall the example of a memory cell, for which both a read and a write are expressed as part of an interaction $m = A \wedge m' = B$. An imperative language would require that these two distinct actions be laid out in a sequence: namely, $[m = A, m = B]$. Observe that with the order made explicit, the temporal ('next') qualifier can be discarded without introducing ambiguity.

Toward compilation to Rust, we are forced to resolve the data types of ports and functions. This resolution is a property of Reo itself, which has its own notion of port data types. For this reason, we are able to reason about the *properties* that characterize a port type, insofaras the releation is meaningful to Reo itself (and not introduced only when implementing the protocol). For example, we may reason that a ports type must facilitate its values being *replicated* if an interaction exists that replicates a value of its type. However, Reo does not prescribe what these types are called in the target language, nor does it prescribe any additional relationships between them. For example, two types distinguished by Reo may be unified in implementation.

In the sequel, we consider translation work *abstract* if is produces a representation valuable to any imperative language, and *concrete* otherwise.

Translation Subtasks Defined

Ultimately, we partition the task of translating from RIR to Rust as four subtasks, where those that are *abstract* precede those that are *concrete*:

[1] These abstract concepts tend to fall apart under scrutiny, as they depend on what is meant by 'computation' at all. Nevertheless, this observation is helpful in the context of our problem, as we prescribe the relationship between Reo and Rust by using the former to 'model' the latter.

	Action	Data Type
Abstract	T_{AA}: From each abstract interaction, a sequence of imperative actions are laid out.	T_{AT}: Ports are mapped to data types characterized by properties.
Concrete	T_{CA}: Abstract actions are reified into concrete, executable Rust code.	T_{CT}: Abstract data types are resolved to satisfactory concrete Rust types.

3.1.2 Pipelining Subtasks

Code generation is an unusual problem, as it introduces a spectrum of possibilities in response to questions that usually have trivial answers. For example, in which language is a concept expressed? Reo specifies the coordination behavior of the generated Rust code, but (by design) nothing more. This freedom makes room for questions of 'where' and 'when' the behavior that emerges at runtime is made concrete.

Toward an answer, we begin by considering one of the possible extremes: the Reo compiler performs as much of the work (i.e., as many of the subtasks) as possible. Whatever behavior is desired in the executable program is spelled out in detail, and reflected explicitly in the Rust code the Reo compiler produces. This solution is arguably the most intuitive, and it has many advantages. For example, we are able to 'front-load' as much computation work as possible, such that the generated Rust code can represent operations in a preprocessed form. We are also given fine control over the behavior of the final binary. However, this strength is what makes this approach impractical: the Reo compiler's ability to specify Rust's behavior in detail also implies a responsibility to do so. By reasoning about the Rust-compiled program directly, Reo must model Rust's language and tooling environment. Recreating this existing work is a poor use of the available software resources. Worse still, it results in Reo compiler becoming tightly coupled to the Rust language, not only syntactically, but in the fine-grained logic necessary for implementing our desired performance optimizations in full detail. At the same time, all flexibility is taken away from the user; they have no ability to understand or influence the concrete implementation themselves. Essentially, this approach trivializes the contributions of Rust compiler.

As one might expect, the opposite extreme trivializes the contributions of the Reo compiler. If hardly any transformations at all are applied before Rust code is emitted, the representation can only be very similar to that of RIR. As explained previously, these forms are simply too different for the Rust compiler to use as-is. By necessity, either a new Rusty-RIR to Rust translation tool would have to be introduced, or the translation would have to occur inside the Rust-

generated program at runtime. Either way, the work of performing our abstract transformations is simply postponed to later stage in the pipeline.

Between these extremes there is a vast spectrum. Ultimately, we wish to choose a balance that partitions the work between the compilers of Reo and Rust in a manner that befits the interests of the language, minimizing the extent to which Reo models Rust or vice versa. Section 3.1.1 touched on the observation that a portion of the work in translating from RIR to Rust is common to other imperative language targets. Our solution is for the Reo compiler to perform only the 'abstract' subtasks (T_{AA}, T_{AT}), translating RIR to a form for which imperative computation is natural, but is otherwise as agnostic to the target language as possible. Reo emits abstract behavior for the Rust compiler to make concrete. Clearly, this abstract representation must be understood by both Reo and Rust compilers, as it crosses the boundary between their stages in the pipeline. To follow, Section 3.2 defines this representation as *imperative form*, embodying the behavior of an abstract imperative language.

Clearly, the Reo compiler backend itself performs the abstract subtasks, but what exactly performs the concrete subtasks? The Reo compiler has an existing backend for generating Java code. It works by generating Java according to the structure of a *template generator*, which defines a hierarchy of string macros for formatting RIR objects in Java's syntax. At first glance, this backend exemplifies the extreme of Reo modeling the target language, performing all of the subtasks itself. However, the extent of the associated coupling to Java is mitigated through the reliance on a purpose-built Java library. Within, definitions are provided for objects that all Reo-generated Java programs will have in common. For example, the library defines a `Component` interface, for which the code generator produces a protocol-specific implementor class. This approach works to minimize the 'surface' of the generated code, by having Reo generate behavior at a higher level of abstraction. Reo generates Java in Java-specific terms, but must generate less overall.

Our approach follows the precedent set by the Java backend; we introduce *Reo-rs*, a purpose-built Rust library which reduces the Rust-specific surface exposed to the Reo compiler. Specifically, Reo-rs defines types that 'hide' their Rust-specific complexities behind an abstract API. We reduce the burden on the Reo compiler further by reducing the granularity of the protocol representation as it appears in the Rust source code; the Reo compiler emits a single `entrypoint` function, which acts as a thin wrapper around the initialization of a `ProtoDef`. This structure is provided by Reo-rs, and corresponds with IF, as it is defined in Section 3.2. By expressing the protocol's behavior in this form, the Reo compiler takes responsibility only of the abstract transformation steps: T_{AA} and T_{AT}. Rust itself completes the translation to its language specifics, partly at compile time and partly at run time. Figure 3.1 gives an overview

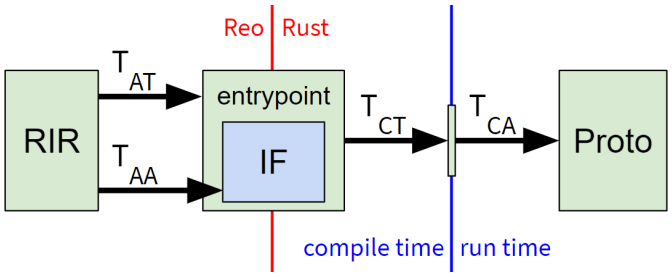


Figure 3.1: The translation pipeline from the Reo compiler's internal representation ('RIR') to the executable Rust `Proto` object. Translation phases correspond to subtasks defined in Section 3.1.1. The majority of the specification is represented in the imperative form ('IF'), which serves as the representation at the boundary between Reo and Rust. The boundary between compile and run time is relative to that of the user's Reo-coordinated program.

of the entire process from start to finish, ultimately resulting in objects which coordinate the components of a program written in the Rust language. Section 3.3 explains the implementation of these stages, and their relationships in more detail.

3.1.3 Enabling Future Expansion

Previously, we observed that the translation work from Reo to its targets can be expected to have significant overlap. For our purposes, we characterize imperative languages by defining IF in Section 3.2, and build our translation pipeline around its use as an intermediate representation.

Performing the translation in clear stages is beneficial to both Reo and its target languages. The benefits to new targets is clear; they may acquire Reo support with less effort than otherwise. Existing target languages are able to benefit also, as they are able to implement the same IF using different concrete translation procedures, i.e., they may implement T_{CA} and T_{CT} differently. Regardless of concrete implementation, protocol objects stemming with an IF specification in common are safe in the knowledge taht they will agree on their abstract action sequences, and the abstract types of their ports.

For the sake of limiting the scope of the project, we relegate IF to a tool unique to the generation of the Rust language; as far as the user is concerned, the Rust backend emits Rust directly, which they are able to use as part of their Rust programs. Future work may remove this restriction by having Reo support IF as a compilation target directly.

3.2 Imperative Form

In this section, we define *imperative form*, a novel intermediate representation of the behavior of Reo protocols, such that they more closely correspond to their final representation in some imperative language. In the protocol translation pipeline, this form is the result of completing translation subtasks T_{AA} and T_{AT}, as they are defined in Section 3.1.1. At this stage, we take for granted that ports and functions have been given abstract types by the Reo compiler.

3.2.1 Concept

RIR does not ergonomically facilitate execution, primarily because it does not define the *order* in which values are accessed, created and moved. Programmers using imperative, sequential languages are very used to thinking in terms of procedures which manipulate the state of variables *in scope* with the order of actions expressed as the control flow. Often, interpreters or compilers provide safety properties by tracing over the control flow, and emitting errors whenever a variable access is invalid.

IF makes explicit the ordering between symbolic *actions*; if executed in the specified order, it is guaranteed that (1) variable accesses are always valid, and (2) a rule fires unambiguously; the effects of a rules actions are only observable if and only if the rule has fired.

Rules as Transactions

RIR partitions the work of a rule into its *guard* and *assignments*. This is already a step in the direction of imperative computation, observing that some work (the guard) must be performed prior to deciding whether the rule fires, in which case the assignment follows. As the protocol does not define the moments when it will evaluate the guard, it is an error to define a guard whose evaluation 'leaks' meaningful behavior.

We are able to interpret a RIR rule as a two-action *transaction* which *commits* after the first, guaranteeing the second will occur. Prior to the moment of commitment, actions are (1) obligated to have a defined means of being *reversed*, undoing their observable effects on the environment, leaving no trace of their execution, and (2) able to initiate an *abort*, reversing their effects. In this view, the guard's evaluation instigates an abort if it evaluates to $\mathtt{false}$.

IF adopts the notion of ordered actions from RIR, but generalizes it to a sequence of any length of at least one. For our purposes, it suffices to have a fixed moment of commitment: immediately before the last action. Accordingly, we retain the term 'assignment' for this last action to mirror RIR. All other actions are transient, and behave as described above. One final stipulation is

needed: after an action's effects are reversed, its predecessor action is reversed also (if they exist). Consequently, an abort propagates up through the transient actions in the opposite order they were originally performed, reversing all of their effects on the state one by one. However, once the last action is reached, the rule's firing has committed, and the effects of all actions will be observed.

Action Granularity

IF represents a protocol's defined interaction as actions to be computed in the specified sequence. At this stage, our representation is still symbolic; actions do not necessarily correspond 1-to-1 with concrete operations in the target imperative language, and their representation of actions is unspecified as long as they preserve the properties of IF. We represent actions at the coarsest granularity possible to avoid overspecifying the ordering of concurrent operations by leaving for meaningful implementation choices in subtask T_{CA}.

The simplest IF rules can be represented with a single action; implicitly, the rule has a trivial guard (consisting of zero actions), consisting entirely of some guaranteed assignment. For example, a rule with a trivial data constraint may be represented as a single, trivial action; the rule always fires to no effect.

The utility of our generalization is the ability to break up a single action into multiple in a manner we are able to represent. For example, we are able to reason about actions that create *temporary variables*, corresponding to the creation of new values at runtime. These actions may be transient, as reversal is easily defined: discard the value. To illustrate our ability to split actions, we represent a protocol, expressed in RBF with data constraint $X = f(X)$ and synchronization constraint $\{X\}$ with only input (putter) port P. This rule can be understood as "X fires if and only if the results of function f on its put value is equivalent to the value itself". Here, the result of f clearly cannot be inspected until *after* it is computed. We are able to represent this rule with an action sequence of length three: (1) Create temporary value f_X by executing f given argument X. (2) Trigger an abort if $f_X \neq X$. (3) The rule has fired; do nothing other discarding values X and f_X.

3.2.2 Definition

Here we define *imperative form*, and explain how its definition corresponds with the intuition behind it. Firstly, an IF contains a structure which corresponds to a *symbol table*; this does the work of assigning symbolic *data types* to ports and memory cells. Ports must also be annotated with an explicit *orientation* (i.e., input or output). Other symbols are also represented here, for example, the names and the argument types for any named functions.

More interesting are the *imperative rules* listed for an IF, corresponding to rules of an RBA or RBF. Each rule is given by a tuple (P, I, M) where:

1. **Premise** P
   The premise is another tuple of three *identifier* sets (P_R, P_F, P_E). P_R is the *synchronization constraint*, i.e., the set of ports identifiers whose values must be 'ready'. P_F and P_E are the sets of *memory values* which must be known to be full and empty respectively, such that it is known whether they can be read from or written to. The rule can certainly not consider firing unless all ports are ready and all memory cells are in the specified states.

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

   - $\mathtt{check}(p)$
     Trigger an *abort* if predicate p over data is satisfied.
   - $\mathtt{fill}_P(m, p)$
     Fill an empty memory variable m with the result of a predicate p over available data. The value's data type is implicitly *boolean*.
   - $\mathtt{fill}_F(m, f, a)$
     Fill an empty memory variable m with the result of invoking function f with arguments a, a list of references to value identifiers with length matching the arity of f. It is incorrect for f to *mutate* the values of its arguments, as this would result in observable effects which cannot be rolled back.
   - $\mathit{swap}(m_0, m_1)$
     Swap the values in two memory variables m_0 and m_1.

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

3. **Movements** M
   A mapping from identifiers of *values* to the identifiers of getter ports and empty memory cells ('destinations'). This represents the final action of an imperative rule executed if and only if the rule *fires*.

An imperative rule aims to model a sequential computation from top to bottom. Instructions are able to (non-destructively) read values, create new variables, and swap the values of variables. Starting from the premise, one

is able to populate a set of values' identifiers *in scope*, and then traverse the instructions and rules. For this reason, it is beneficial to distinguish P from an instruction: P also establishes the initial scope.

The IF is well-formed only if no instruction no the movement is *invalid*. Here, it suffices to say that validity models the usual scoping rules in an imperative language. For example, one cannot read from an uninitialized value. Other restrictions are in place to ensure that instructions are always reversible. For example, $\texttt{fill}_F$ may only be used if the value it populates is previously uninitialized. The full enumeration of constraints is available in the source code, visible in the `build` function of the `ProtoDef` type (this is explained further in Section 3.3.3, including a listing with some example errors).

M is defined with a representation that makes it trivial to distinguish the cases where values are discarded (0 destinations), moved linearly (1 destination) or replicated (multiple destinations). This design is convenient for languages that require their values to be more explicitly managed. For example, languages with *affine types* (e.g., Rust) must simulate the replication of values by creating new affine resources from the original, and managing the replicas explicitly. *Relevant* data types (which must be used *at least once* [Wal05]) must handle empty destination sets by either emitting errors, or by simulating destruction.[2] There are many other reasons a language may want to specialize the way its values are used. For example, an implementation in C++ may need to inject `free` calls to avoid leaking memory in cases where pointer-values are discarded.

As an example to demonstrate IF, the RBA rule in the previous section with data constraint $X = f(X)$ and synchronization constraint $\{X\}$ can be represented in the imperative rule with:

	value
premise	$(\{X\},\ \{\},\ \{f_X\})$
instructions	$[\texttt{fill}_F(f_X, f, [X]),\ \texttt{check}(X = f_X)]$
movements	$\{X \to \emptyset,\ f_X \to \emptyset\}$

3.3 Translation Pipeline

This section details the implementation of the translation pipeline from RIR of Reo protocols to executable Rust. The section is structured to describe the translation process as a sequence of sequential stages. Unbeknownst to the user, the pipeline extends well after the Reo compiler has emitted Rust code.

[2] A relevant language may simulate the destruction of a value by moving it to some `Destroyed` destination with special semantics.

Subsections are titled according to *when* the translation takes place, and are presented in the order they are performed.

3.3.1 At Reo Compile-time

The Reo compiler is extended with a backend for translating RIR to a Rust source file. This translation stage is concerned with performing T_{AA} and T_{AT}, and representing them as a single rust *entrypoint* function in the emitted Rust source. The user is able to import this source as a dependency into their own project, whereby the *entrypoint* serves as a means for their program to construct the executable protocol object.

Action Sequencing

T_{AA} necessitates transforming a each of the protocol's rules into a sequence of symbolic actions. The most significant work occurs as a result of how differently *values* are represented. RIR is declarative, representing the result of a rule's firing as an *assignment*, mapping each *destination* (getter port and empty memory cell) to a `Term`. RIR already represents a significant transformation from RBF in isolating these values on a per-destination basis.

To begin, we describe the naïve approach to translate RIR to IF, one rule at a time. The translation procedure initializes all three fields $\{P, I, M\}$ of an imperative rule as initially empty, and populates them incrementally by recursively traversing the RIR rule's assignments. Each such assignment is ultimately represented in M, where terms are rather represented by identifiers. For some terms the mapping to identifier is trivial. For example, the value put by a port can use the identifier of the port itself. For others, it may be necessary to introduce fresh identifiers, representing temporary variables to be created. In either case, the term is traversed recursively to (1) collect these identifiers, and to (2) populate the premise P such that the rule is fired given access to all of the relevant memory cells and ports.

I is populated last by three kinds of actions. Firstly, the exceptional cases for which memory variable q will be both read and written to are treated. If necessary, a fresh temporary variable is introduced by appending an instruction $swap(q, q_{temp})$ where q_{temp} is some fresh (uninitialized) variable; q's previous and next values may be read and written unambiguously, distinguished by identifiers q_{temp} and q respectively. Second, I is appended with $fill_P$ or $fill_F$ instructions to create every other temporary variable in a manner befitting the term that represented them in the RIR's assignments, i.e., the result of invoking a function with port values as arguments. Finally, I ends with a single $check$ to evaluate the rule's guard, initiating an abort if it is evaluates to $false$.

As it was described thus far, our procedure is able to correctly render any RIR rule in IF with the necessary properties. For the sake of minimization or performance at runtime, at least three optimization opportunities may elaborate on this procedure, producing semantically-equivalent results.

1. Terms that occur repeatedly within assignments throughout the same RIR rule may have their **values deduplicated** by assigning them all the same identifier. Care must be taken to ensure that the instruction to create its value is inserted only once, sufficiently early that its creation precedes its earliest access. Note that each original occurrence still corresponds with a destination in the resultant M mapping. To clarify, consider the example with getter ports A and B both assigned terms corresponding to $f(C)$ where f is some function and C is a putter port. Here, one temporary variable f_C to store the result of $f(C)$ is sufficient; it is simply moved to two distinct destinations, reflected by the mapping $f_C \to \{A, B\}$ in M.

2. The large, monolithic `check` instruction that acts as a guard to the rules firing can be fragmented into **numerous guard instructions**. The utility of this is the ability to specify how its parts are ordered. For best results, it is beneficial to move checks as early as possible, such that less work is performed prior to an abort whenever the check fails at runtime. To be correct, care must be taken not to move guards so early such they precede the creation of any temporary variables their evaluation accesses. For example, consider an RIR whose guard is $A \wedge B$, where A and B are subformulas that reason about subterms whose evaluation necessitates the creation of temporary values t_A and t_B. By fragmenting $\texttt{check}(A \wedge B)$ into $\texttt{check}(A)$ and $\texttt{check}(B)$, we are able to move the former such that it follows the creation of A, but not of B. Effectively, the rule is able to *short circuit* its evaluation at runtime, circumstantially avoiding the creation and destruction of the temporary value identified by t_B.

3. **Static analysis** of values may conclude that a `check` instruction is a tautology, making it safe to omit. Similarly, the presence of even one contradictory `check` makes it possible to discard the rule entirely (as it will never fire). This optimization is particularly useful in combination with optimization (2).

Type Classification and Constraining

Our backend performs task T_{AT} to generate the IF such that the identifier of every port, memory cell, and temporary variable is assigned a symbolic type annotation, such that:

1. the types of identifiers match if they exchange values or are checked for equivalence.
2. data types are boolean if they occur in a context in which only boolean types are permitted, i.e., as the predicate of a formula.
3. the type is constrained by *properties* which guarantee the type defines operations which Reo may apply to its values.

Ultimately, ports and functions are mapped to a set of symbolic types, each of which is annotated with properties. These properties cover the most fundamental ways Reo interacts with values of a data type, aside from moving them through ports (which we assume to be inherent to all port types). Namely, the values of a type may be constrained such that its value may be (1) replicated, (2) checked for equality, returning a boolean value, (3) initialized from a given string, i.e., parsed.

Our backend performs this work in tandem with the work of T_{AT} described in the previous section. Initially, every identifier is assigned a fresh symbolic type with no properties, representing a data type unrelated to any other, and having no need of any defined operations. In traversing the RIR rules, properties are collected and associated to the relevant types. In some cases, a relationship between identifiers causes their types to be *unified*, resulting in a new type with the union of their properties. For example, a data movement from putter P to getter G unifies their types. If the requirements on types are *contradictory*, an error is emitted by the Reo compiler. For example, it is an error to provide types A and B with different explicit data type annotations, yet have them exchange data. Ultimately, the constructed IF associates a symbolic type with every mapping in its symbol table.

Rust Formatting

Once all the data has been prepared, the Reo compiler emits its contents formatted according to Rust's syntax. Listing 11 gives an example of the result: a single `instantiate` function reflecting the cumulation of the work of this phase. This function serves as the user's entrypoint for creating protocol instances with the corresponding behavior. Observe that in this example, a single symbolic data type `T` was identified, and annotated with property `FromStr` (which ensures the type can be parsed from a given string). As expected, the majority of the information is contained in the `ProtoDef` type (beginning on line 5), which is nothing more than a Rust-formatted rendering of the protocol's IF behavior specification. The sections to follow explain what happens next.

3.3.2 At Rust Compile-time

A Rust programmer makes use of the generated Rust code by importing it into their own program as a library. To interface with its contents, they are required to import Reo-rs as well. Section 4.3.1 explains the API this library presents to users for acting on these protocol objects from their own code.

As the name suggests, the `instantiate` function serves as the user's entry-point for instantiating protocol objects. This function is invoked from their own Rust code in the usual way. Previously, Section 3.3.1 explained that the output of the Reo compiler is a Rust source file containing a single entrypoint function. Rather than implementing it ourselves, our solution is for the definition of this function to effectively delegate subtask T_{CT} to the Rust compiler itself. From the user's perspective, the entrypoint is a function like any other; at the call-site within their own programs, they are able to specify concrete choices for the generic types themselves (Section 2.2.2 explains how Rust dispatches generic functions). This approach has three benefits: (1) We make use of an existing resource, which is not only easier, but is also good practice as it avoids the fragility that would otherwise follow from redundancy, (2) the result is idiomatic for the Rust language, and ergonomic for users to use in conjunction with other generics in their own programs, and (3) once Reo has emitted the entrypoint, a Rust programmer is able to use it to construct protocol instances for any choice of concrete types.

Using the Rust compiler in this way is achieved by the Reo compiler crafting the entrypoint such that its generic type arguments are annotated with the appropriate Rust trait bounds. In effect, we communicate to the Rust compiler the operations which the user's code chosen types must support. Line 2 in Listing 11 gives an example of a Reo-generated entrypoint function, `instantiate`, where `T` is expressed with a trait bound `FromStr`.

3.3.3 At Application Runtime

The user's program has been compiled by the Rust compiler, the resulting binary can be directly executed. Whenever the entrypoint function is executed, an instance of `Proto` is constructed and returned, indirected behind a `ProtoHandle`. These types and how they work to implement their associated Reo protocol at runtime is explained in Section 4.3. Here, it suffices to say that all Reo-generated entrypoints return the same `Proto` type, but the behavior and interfaces of these instances vary to reflect that of their specification. The final subtask of protocol translation, T_{CA}, occurs at runtime, in the execution of the entrypoint function itself. Owing to Rust's imperative nature, what happens next occurs a sequence of four distinct steps, resulting from the four distinct variables initialized in the body of `instantiate` in Listing 11.

Type Erasure: `type_info`

To make it possible to represent any and all protocol objects with the single `Proto` type, it is necessary to *erase* the types of port values and functions, representing their types as data instead. *Reflection* is the counterpart to this operation, allowing the port types of `Proto` objects to be distinguished at runtime; this occurs elsewhere, and is explained in Section 4.3.4 in the following Chapter. As can be seen in the example, this first step is trivially represented by the Reo compiler, relying on the definition of `TypeInfo` in Reo-rs.

Memory Initialization: `mem_init`

In the original textual Reo specification, the initial values of a protocol's memory cells is defined by strings (as a result of the textual syntax). To afford the user's choice of arbitrary types, we require that the original text can be translated into a value of the correct type to initialize the protocol's memory cells. Rust defines the `FromStr` trait to characterize types which have the property that their instances can be constructed by parsing a string at runtime. The entrypoint function is safely able to rely on the corresponding `from_str` operation to be defined for the type, as care was taken to include `FromStr` as a type constraint. These constraints are based on Rust's trait system (see Section 2.2.2), and correspond with the symbolic type properties explained in Section 3.3.1. The result is a `MemInitial` structure, which stores instances of initialized memory variables.

Imperative Specification: `proto_def`

As explained previously, the Reo backend embeds the imperative form specification in Rust's syntax such that the result is a dependency which the Rust compiler can understand. At runtime, this step necessitates that a `ProtoDef` instance be built, only to be read on the next line and subsequently discarded. Conceptually, this step could be performed at compiler-time by defining the `ProtoDef` in terms of types that the Rust compiler is able to evaluate at compile time (embedding it into the text section of the binary). This would avoid the work of constructing the `ProtoDef` with every instance, as (to follow) we see that one `ProtoDef` is able to instantiate any number of `Proto` instances. Instead, this object defined such that it requires some simple initialization at runtime. The overhead is inconsequential, particularly as `instantiate` is not a performance-sensitive function. However, the benefits of this dynamic definition are the ability to manipulate these definitions at runtime.

Construction: `built_proto`

The combination of `ProtoDef` and `MemInitial` represent (a specification of) the behavior, and initial state of an executable protocol object respectively. The final step is to put it all together and perform the only remaining subtask: T_{CA}. Reo-rs encapsulates this work in the `build` method defined for the `ProtoDef` type, visible in Listing 11 on line 32.

Completing subtask T_{CA} in this context consists of (1) initializing a protocol object complete with auxiliary bookkeeping structures, and (2) implementing the behavioral specification.

Our design for the implementation of executable protocol objects relies on a lightweight interpreter, which traverses a terse datastructure whose contents correspond to IF rules. For this reason, we are able to delay T_{CA} until the program's runtime. Section 4.3.2 explains how they function at runtime. Here, we focus on how the `Proto` structure is initialized.

Many of the fields of `Proto` are included to facilitate the granular operations that are defined by Reo-rs, without a clear parallel in the imperative form specification. Nevertheless, their presence is essential at runtime: for example, semaphores and control message channels. The fields that remain correspond to definition of the `ProtoDef`. Both its symbol table, and its rules are incorporated into the `Proto` instance after they undergo *preprocessing*, which performs two vital functions:

1. **Optimize Representation for Execution**
   `Proto` represents the ultimate departure from the initial Reo specification. in Chapter 4 to follow, we explain how the contents of `Proto` are accessed directly while in use as a communication medium at runtime. Due to their different purposes, these types have different representations, each specialized to its own purpose. Where `ProtoDef` prioritizes terseness and readability in the use of symbolic port and function names, the translation to `Proto` resolves them to concrete data structures for cheaper access, e.g., indexes into a vector replace symbolic port names.

2. **Ensure Internal Consistency**
   As Reo's backend for Rust is under our control, we are safe in the presumption that the Reo compiler can be trusted to create only internally consistent `ProtoDef` structures. However, the code generation process crosses a boundary between two compilers, and is designed to minimize their coupling by condensing the data that passes between them. As is good software development practice, Reo-rs works to minimize its dependency on the Reo compiler. In particular, Reo-rs avoids assuming that the Reo-generated `ProtoDef` describes *some* valid, sensible protocol. This

precaution is motivated because IF simply exposes more opportunities for inconsistencies than RIR (owing to its increased explicitness), which may cause modifications to the Reo compiler to introduce bugs.[3]

To make error handling ergonomic, Reo-rs adheres to Rust's idiom for error handling. In the event that an inconsistency is detected during `build`, an error variant is returned with information about the inconsistency. Listing 12 shows the resulting type signature of `build`, including some examples of possible error variants.

Listing 11 demonstrates how the result of `build` is returns the resulting protocol object. Observe that it is returned indirectly, represented by a `ProtoHandle`. The relationship between these objects and the definition of their behavior at runtime is detailed in Chapter 4 to follow.

[3] A user may tamper with their Reo-generated Rust code such that a different, consistent protocol results. We cannot distinguish this from intended behavior, and so users take responsibility for their own misfortune in tampering with generated code.

```
1  use reo_rs::*;
2  fn instantiate<T: FromStr>() -> ProtoHandle {
3      let type_info = TypeInfo::of::<T>();
4      let mem_init = MemInitial::default().with("m", "VALUE".from_str());
5      let proto_def = ProtoDef {
6          name_defs: {
7              "A" => Port { putter: true,  type_info },
8              "B" => Port { putter: false, type_info },
9              "m" => Memo(type_info),
10         },
11         rules: [
12             RuleDef {
13                 premise: Premise {
14                     ready_ports: {"A"},
15                     full_mem:    {   },
16                     empty_mem:   {"m"},
17                 },
18                 instructions: [],
19                 movements: { "A" => {"m"} }
20             },
21             RuleDef {
22                 premise: Premise {
23                     ready_ports: {"B"},
24                     full_mem:    {"m"},
25                     empty_mem:   {   },
26                 },
27                 instructions: [],
28                 movements: { "m" => {"B"} }
29             }
30         ],
31     };
32     let built_proto = proto_def.build(mem_init).unwrap();
33     return built_proto;
34 }
```

Listing 11: The Reo-generated Rust source given the `fifo1` connector's Reo specification as input. Section 3.3 explains how this representation bridges the gap between the Reo and Rust languages. The `ProtoDef` type on line 5 specifies the protocol's behavior in imperative form, as it appears embedded into Rust's syntax.

```rust
fn build(&ProtoDef, MemInitial) -> Result<ProtoHandle, BuildError>;

type BuildError = (Option<usize>, BuildErrorInfo);
enum BuildErrorInfo {
    MovementTypeMismatch     { getter: Ident, putter: Ident },
    ConflictingMemPremise    { name: Ident },
    PutterCannotGet          { name: Ident },
    GetterHasMuliplePutters  { name: Ident },
    InitialTypeMismatch      { name: Ident },
    GetterHasMultiplePutters { name: Name },
    EqForDifferentTypes,
    CheckingNonBoolType,
    /* 12 more variants */
}
type Ident = &'static str; // static string literal type
```

Listing 12: Signature of the `build` function. Its inputs are (1) an immutable reference to a `ProtoDef`, which is used to determine the protocol's behavior, and (2) a `MemInitial`, which stores initialized memory cells to be incorporated into the protocol's state. The return result is an enumeration type, returning `ProtoHandle` upon success, and a tuple on failure, whose elements are, respectively (1) the index of the imperative rule where the error occurred, if applicable, and (2) `BuildErrorInfo`, another sum type communicating the nature of the error with additional information.

Chapter 4

Protocol Runtime

Previously, Chapter 3 described how a Reo protocol specification is translated by the Reo compiler into the Rust language as an executable protocol object. In this chapter we discuss how these objects are able to act as the communication medium between a set of communicating components. This approach allows the user's component code to exchange data with its environment through the protocol object's exposed ports. Recall that components make no assumptions about the world beyond their ports, and consequently, have no notion of the system in which they play a part. From a user's perspective, ports are entirely opaque, and their components may use them to exchange data with their environment without any concern for global coordination.

Internally, protocol objects orchestrate the actions on their boundary ports into interactions defined by its protocol specification. As much as possible, the protocol will work to facilitate data flow. However, whenever a boundary port initiates an action which does not yet fall into a suitable interaction, the protocol exercises its power to block its completion until the time is right.

Section 4.1 begins by examining the Reo-generated protocol objects for the Java language, allowing us to use this previous work as a touchstone for our own. Within, Section 4.1.3 observes opportunities for our implementation to improve upon it by the addition of safety properties, and exploiting opportunities for optimization. Section 4.2 makes our goals explicit by defining the requirements and guidelines used to inform our design process and determine the assumptions used to facilitate out implemented optimizations. Section 4.3 follows with the implementation of the *Reo-rs* library, which defines our protocol and port objects. Within, Section 4.3.1 explains how we leverage Rust's affine type system to expose a safe user-facing API. Sections 4.3.2–4.3.4 explain how the protocol object behaves at runtime, detailing the implementation of optimizations which enable it to (1) coordinate actions without needing a ded-

icated thread, (2) increase parallelism by delegating data movement to component threads, and (3) internally perform reference counting and reference passing while preserving Reo's semantics. Finally, Section 4.4 gives an overview of how our requirements and guidelines are satisfied, including references to the sections containing the relevant details.

4.1 Examining the Java Implementation

The Reo compiler has seen extensive development for its Java code generator in particular. In this section, we examine the properties of the source code it generates. Later, Section 4.1.3 makes particular note of opportunities for our own version to improve upon this design, or at least deviate to the end of specializing its implementation to better suit the Rust language.

4.1.1 Architecture

Fundamentally, the generated code adheres closely to Reo's literature, revolving around the interplay between implementors of the `Port` and `Component` interfaces. (For brevity, we will refer to classes that implement these interfaces by these names also). From the perspective of a programmer, the entrypoint is the constructor of a generated protocol `Component`.

Running a system requires an initialization procedure: (1) a `Port` is instantiated per logical port, (2) a `Component` is instantiated per logical component, and (3) pairs of components are linked, by overwriting a port field for both objects with the same instance of `Port`. To get things going, each component must be provided a thread to enter it's main loop; in idiomatic Java, this manifests as calling `new Thread(C).start()` for each component C. A simplified example of the initialization procedure is shown in Listing 13 for the simple `sync` protocol. Observe that the data type of the ports (here, `String`) are represented in the generic argument of `Port`.

The design revolves around `Port` as a communication primitive between threads, and equivalently, between components. As such, ports themselves are the *critical region*, with contents only accessible to the holder of its lock. In contrast, `Components` are used only to store their ports and to be used as *namespaces*, the static environment within which their behavior is defined by the `run` function. In the case of the Reo-generated protocol component, the contents of `run` implements its Reo specification, with logic broken into the rules, assignments, and guards reminiscent of RBAs. Essentially, any behavior that interacts with a `Port` can be considered a component, whether or not it implements the `Component` interface.

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

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

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

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

Listing 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.

4.1.2 Behavior

The representation of protocol rules is very intuitive; a rule is implemented as a block of code which operates on a component's ports. Once generated into Java, the only obvious sign that a component was generated from Reo is its linkage to multiple other components. The (simplified) generated `Component` code of sync, seen previously, is shown in Listing 14. This demonstrates that rules are indeed *commandified*, i.e., their behavior is encoded as discernible data 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. Components represent memory cells as fields, and use `null` to represent emptiness, i.e, the absence of a data element.

4.1.3 Observations

Reo-generated Java objects have a very clear correspondence to their declarative Reo specification. This carries over to how components and ports are used by an application developer. For example, Port objects act both as points of

```
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     private Command[] commands = new Command[]{
11         new Command(){
12             public void update(){
13                 p1.put(p0.peek());
14                 p0.get();
15     }   },  };
16
17     public void run() {
18         int i = 0;
19         while (true) {
20             if(guards[i].guard())commands[i].update();
21             i = i==guard.length ? 0 : i+1;
22             synchronized (this) {
23                 while(true) {
24                     if (p1.hasGet() && (!(p0.peek() == null))) break;
25                     try {
26                         wait();
27                     } catch (InterruptedException e) { }
28 }   }   }   }   }
```

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.

data exchange and as primitive concurrency mechanisms aligning `put` with `get`. From this design, we observe the following noteworthy properties:

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 port in the event of a `put` or `get` invocation. For this reason, protocols and components are related in both direc-

tions, afforded by setting a port variable in one direction, and functions `setProducer` and `setConsumer` in the other. This is visible in Listing 13 on lines 8–11.

True to the RBA model, the protocol must check which (if any) commands can be fired continuously. The running thread achieves this by evaluating all guards and firing rules as possible. If no progress can be made, the thread awaits a notification. This is unfortunate, as this approach requires guards to be evaluated repeatedly, which can be arbitrarily expensive. As the protocol relies on the actions of other components to make progress, it is counterproductive 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 ('JVM'). Variables of such objects are therefore moved around the stack by reference. Moving and replicating values is cheap and easy, as they always have small (pointer-sized) representations.

   As is usual for Java, generics may only be defined for classes that inherit from `Object` (excluding primitives such as `int` and `float`). As is the idiom, primitives are represented by immutable object classes (`Integer`, `Float`, etc.) in these cases. Users are not able to transmit small values without introducing this indirection. The storage that backs their primitive is also out of their control, and is decided by the JVM. As a consequence, transmitting even primitive values through ports may result in heap-allocation. 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. Consider a protocol P with boundary components A and B. Data transmitted from A to B requires values to hop at least twice: $A \to P$, and $P \to B$. Fortunately, as stated above, the cost of the 'hop' itself is trivial, as only the pointer-sized reference is moved. The real problem is the overhead resulting from the hop

requiring three threads to exchange data in series. If this movement is defined as within a synchronous interaction, the round trip holds up P's execution of the rule, and the entire system around it.

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*, where the thread blocking for a notification differs from the thread receiving the notification, and so never wakes up. Java solutions exist to wrap ports in objects that constrain the API such that a component is only permitted to use the port as intended (e.g., putters can `put`, but not `get`). However, without affine types, there is no obvious way to ensure that each port is used by at most one putter and one getter.

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. Replication of references is called *aliasing*, as the referent is accessible via multiple bindings ('aliases'). If no referee can observe the presence of another, aliasing is not a problem. This is the case for some objects. As an example, `Integer` is an immutable object which also does not define any behaviors that can lead to further aliasing. This is not the case in general. Objects often facilitate the mutation of their fields, or define

methods which access other objects which do. When accessed by multiple threads in parallel, this can result in undefined behavior in the user's program. This contradicts Reo's value-passing semantics. Users cannot interact with values they acquire through ports without knowledge of their environment, as they cannot otherwise be sure if operations on these values are safe.

The Rust compiler statically enforces the impossibility of data races (see Section 2.2.2). Safe reference-passing between threads is nontrivial in Rust; users typically rely on thoroughly engineered libraries to provide safe abstractions for such things, rather than implementing the primitives themselves by introducing and managing `unsafe` code. Section 4.3.1 explains our solution for preserving Reo's semantics.

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. **Linked Protocols Deadlock**
   Section 2.1.3 explains that RBAs in our context do not work as *acceptors* of observed port values. Instead, they distinguish guards from assignments such that they can perform an active role in deciding the system's next state. In implementation, this manifests as protocol threads being passive, reacting to the actions on their boundary ports. Concretely, the protocol will rely on the ability to `peek` into the contents of a port, allowing it to reason about a value *before* it has been transmitted. In this manner, protocols rely on the ability to impose synchrony on the actions at their boundary. When two protocol components are linked by a port, each will passively wait for the action of the other. No protocol will act 'first'. The problem of dynamic synchronous decomposition necessitates each protocol participating in difficult distributed problem solving. The solution is not present in the Java implementation, and is out of the scope of our work as well. The problem here is one of safety: the Java implementation permits the linkage of two protocols, despite their behavior being incorrect.

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. In effect, a Reo-coordinated system is bottlenecked by the protocol component. Each rule's firing is a substantial task, and only one is able to occur at a time.

4.2 Requirements and Guidelines Defined

Following the observations of Reo-generated Java objects in Section 4.1, we identify and make explicit the design choices and goals that inform the design and implementation of our own protocol objects for the Rust language. First, we identify a number of functional requirements, representing the goals which can be assessed for satisfaction unambiguously.

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

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

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

Not all useful properties can be meaningfully quantified by strict requirements. We identify a set of guidelines intended to focus the design process, and to form a basis for the assumptions that make meaningful optimizations possible. By their nature, these guidelines cannot be clearly satisfied. Instead, the extent to which guidelines are satisfied is motivated by argumentation, and supported by experimental evaluation in Chapter 5 where applicable.

$\mathbf{G}_{data}$ Allow the transmission of large data types without requiring the user to move them 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.

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

$\mathbf{G_{end}}$ Facilitate a correct and ergonomic means of detecting the termination of a protocol object, i.e., protocol objects for which no more rules can be fired. Facilitate the protocol object being destroyed and its resources freed.

4.3 Protocol Objects

Here, we detail the structure and behavior that cause our Rust protocol object type, `Proto`, to coordinate the actions of boundary ports at runtime in accordance with its associated Reo specification. Section 4.3.1 details the user-facing API, explaining how its helps to provide safety properties. Protocol objects offer an expansive design surface, and so Section 4.3.2 relates our implementation at a conceptual level to that of the Java version that came before. Section 4.3.3 lays out the structural definition of `Proto`, which is relevant for understanding its connection to the code generation process and imperative form explained in Chapter 3. Finally, Section 4.3.4 details the implementation of `Proto`'s behavior at runtime, explaining the roles of the boundary components, how actions are arranged into interactions, and the effects of our implemented optimizations.

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 interacts mostly with Reo-rs itself, and Reo provides only the *entrypoint* function for building particularly instances of protocol object (explained in Section 3.3). In this section we explain which functionality of Reo-rs is user-facing, and explain how the API helps to satisfy our requirements.

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. For a concrete example, the previous chapter includes Listing 11, showing the `ProtoDef` for the `fifo1` connector. 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 (i.e., 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 replication (safely aliasing the `Proto`) or destruction. If the the last handle to a `Proto` instance is destroyed, all of its resources are freed. This is achieved by relying on Rust's canonical `Arc` type ('atomic reference-counted') from the standard library for the definition of `ProtoHandle`.

The protocol remains inert until the user acquires some of its ports. However, they cannot be constructed independently. Instead, the user must invoke `claim` method of a `ProtoHandle`, which is parameterized by the port's logical name; this corresponds with the symbolic name as it appears in the imperative form. By encoding both its orientation (i.e., `Putter` or `Getter`) and its data type as type parameters for each port object, `claim` is able to reflect on these properties and return an error (Section 2.2.2 explains how Rust represents exceptions with enumeration types) if they incongruous with the protocol's definition, or if the port of that name is currently claimed. Similarly, port objects notify their `ProtoHandle` that their name is again available to be claimed in the event of their destruction. All together, this API is able to guarantee that (1) every logical port has at most one port object at once, and that (2) the types of ports enforce that their orientation and data type align with their specification.

`Proto` objects are stored on the heap, but their ownership is shared between all of their existing `ProtoHandle` replicas. Internally, ports contain a `ProtoHandle` each also. The `Arc` within ensures that these handles are moved, acted upon and destroyed in a thread-safe manner. A `Proto` instance is destroyed when its last handle is destroyed, freeing all of its resources in the process. `Proto` objects do not rely on any static variables, allowing any number of them to be instantiated and used throughout a program without them interfering with one another's execution (except, of course, their sharing of the underlying hardware resources).

Port Operations and Variants

The API that defines `put` and `get` operations for ports is partitioned over port types in the expected way. Concretely, ports are represented in Reo-rs by distinct types `Putter<T>` and `Getter<T>`, each generic over their data type, `T`. Appropriately, `get` is only defined for getters, and `put` is only defined for putters. In both cases, the operations rely on Rust's borrowing rules to ensure that even if the port objects is shared within the user's program, it is impossible to act on a port from two threads concurrently (without circumventing the Rust compiler with `unsafe` operations). For example, the signature of `get` is specifies that the getter is accessed via a mutable reference[1] (written `&mut`).

[1]This terminology is often confusing; despite the name, 'mutability' is more often used to mean 'mutually exclusive', as is the case here. The Rust compiler will only permit this operation in a context where it is certain that the port is not concurrently being used elsewhere.

The `get` operation blocks the calling thread until an element of type `T` is returned. Users are able to customize their involvement in the interaction by invoking one of `get`'s variants. For example, `get_timeout` blocks the thread up to a specified timeout, and returns an `Option<T>` to distinguish success from failure. The latter case represents the failure to participate in an interaction, allowing the caller to reclaim the control flow and continue working. Getters also have the option of calling `get_signal`, which expresses a wish to participate in an interaction, but no desire to acquire the value itself. The utility of this option is that Reo-rs will alter its behavior to potentially avoiding a `clone` operation, or other associated overhead (explained in Section 4.3.4). The effects of this variant on performance are shown in Section 5. Finally, `get_signal_timeout` combines the functionalities of the other operations as expected.

Putters have access to `put_timeout`, which varies from `put` as before. Both operations have the potential to return the putter's value. This may occur even if the value was involved in an interaction, but was not acquired by any getters. Returning the value allows the user's program to decide what to do with the value. For example, a user may decide to put the value again until it is read. If this behavior is undesired, putters offer the `put_lossy` and `put_timeout_lossy` variants which will not return the value regardless of the actions of getters, dropping it if needs be.

Value-Passing API Semantics

The Rust language conflates the movement of values to new variable bindings with two meanings: (1) the value's ownership is transferred to the new binding and scope, and (2) the value's shallowest representation is moved in memory. Reo's semantics require that the data transmitted through ports is truly *moved*, transferring ownership. Clearly, not doing so would violate the semantics; values acquired through ports might be dropped or acted upon at the whims of their original putter, of which the getter would have no knowledge. As is idiomatic for the Rust language, our API transfers ownership into the scope of `put` and out of the scope of `get` by moving the value itself. Naïvely, this has significant performance implications, as the cost to move a value is dependent on the size of its representation. For example, a 10MB array is significantly more costly to move than a byte. For cases where the relocation of bytes is not necessary, Rust programmers can often rely on LLVM to optimize the memory movements away, passing consumed resources by reference 'under the hood' (but retaining the semantic movement of ownership). 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]. The only way to guarantee that the data is passed by reference at runtime is to expose an `unsafe` API, which relies on the user to pass

references and manage the associated ownership manually. Our requirements prioritizing safety ($\mathbf{R}_{value}$) and performance ($\mathbf{G}_{data}$) are in conflict. Our solution is to expose these options to the user as variants for `put` and `get`, and allow them to decide on a case-by-case basis:

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.**
   Operations are parameterized by references (or raw C-like pointer types) which Reo-rs writes to and reads from directly. The caller takes responsibility for ensuring that the value at the pointer's destination is initialized or dropped as necessary to correspond with Rust's usual ownership system. For example, this version of `put` is given a pointer to initialize memory, which the operation will initialize.

3. **Value pass a 'referring' type.**
   As far as Reo-rs is concerned, this is indistinguishable from case (1). However, the user intentionally reinterprets the data types of their ports such that they represent indirections. For example, `Box<T>` is a pointer-sized owned type which will be transmitted by Reo-rs much like any other pointer-sized integer, oblivious to the fact that it indirectly represents another type `Q`. Taken to the extreme, a naïve solution replaces all data types with heap-allocated indirections. Other approaches may be simple (eg: transmitting an index to a shared vector) 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].

To reflect our priority of safely, the 'default' port operations use value passing, corresponding to options (1) and (3). Users are able to take safety into their own hands by opting into `*_raw` port operation variants. We rely on Rust's idiom of marking such operations as `unsafe`, communicating to the user that they are adopting the responsibility of reading the API documentation to determine and provide the necessary guarantees, as is usual in languages such as C. Invoking `unsafe` function requires the user's code to be explicitly qualified as an `unsafe` block, making it difficult to unintentionally overlook this requirement.

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

Interface with C and C++

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

Rust makes frequent use of generic type parameters, which rely on the compiler for dispatch at the call site (see Section 2.2.2). C has no analog for this feature, and thus, some structures and functions are be provided secondary concrete representations. As an example, Rust represents the `Putter` and `Getter` types with a generic argument, affixing its data-type at compile-time. Reo-rs preserves safety guarantees by relying more extensively on reflection at runtime for these cases.

Section 2.2.2 explains that Rust does not use header files. Instead, function signatures may be expressed with traits, which are able to declare functions without defining them. However, traits are of no use to C, as they are inherently coupled with Rust's generic system. To facilitate the sort of workflow that is idiomatic to C, such that *compilation* can be distinguished from *linkage*. The Rust ecosystem offers *Bindgen*[3], an addon for generating C header files from Rust source. Bindgen is installed as a module for *Cargo*, Rust's package manager (comparable to Pip for Python, NPM for Node-js, and perhaps Maven for Java). With this tool, we are able to generate C header files without much friction, and include them in any distributions such that downstream dependents on Reo-rs can incorporate them into their applications as they would do for any other C library. Once compiled and linked, their C or C++ applications would execute as would any other binary, and the use of Rust for compiling Reo-rs is no longer visible. Owing to its nature, calls that cross the Rust-C boundary do not induce any significant overhead [KN18].

4.3.2 Design Process

Many designs for the implementation of Reo-like coordinators are possible. Their structure and workings all depend on how information is arranged, and how multiple threads come together to coordinate on an a priori unknown task without stepping on one another's toes. In our case, we concentrate on the

[3] https://crates.io/crates/bindgen

case where all participants in the system share a memory address space, which opens up many means of exchanging data between threads.[4] As is typical in multithreading, the problem is not accessing the data, but rather restraining oneself from accessing the data at the wrong time. Before we can approach any design decisions, we examine what we know for certain: ports invoke `put` or `get`, each from their own thread. They cannot return immediately, as this would not result in the correct system behavior; when not aligned in time, getters will often (unknowingly) read uninitialized data, and putters will write their data, never to be read. They wish to exchange data in accordance with some defined protocol, but a priori have no knowledge of the protocol, nor their role in it. The aim is to facilitate rule firing 'greedily' as opportunity allows: i.e., protocol objects should fire rules as frequently as possible such that the behavior of the system is not constrained *beyond* the constraints of the protocol itself.

The Coordinator

The most obvious starting point is asking 'who decides which rule to fire?' Reaching consensus prevents the system from reaching some malformed 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 rediscover 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

[4] This assumption provides context for our work, but is not an inherent assumption of all of our design choices. Wherever possible, we make this distinction in the text.

concurrently.[5] The coordinator then treats the port structures like messaging pigeonholes, and performs the task of moving data around itself. The coordinator's notification to the ports is subtle; taking the form of `put` and `get` calls which release port-local locks, unblocking the compute threads, completing the interaction. This solution is effective, but has some downsides, as discussed in Section 4.1.3.

Event Handling

A minor change with the potential for improvement is to remove the necessity of the protocol thread to rediscover the nature of the event which generated a wakeup signal. Rather than signals with no payload, we can use events which carry explicit information, eg: 'Port x is ready to get!'. With this approach, the coordinator waits in an event loop, handling incoming events. The Rust ecosystem has a number of libraries for defining event loops built atop system signal handles. An early implementation made use of the `mio` crate for sending events which communicated *which* port has become ready. With this minor change, the coordinator does not need to inspect the contents of ports directly, which, owing to their modification by multiple threads, inherently cause several cache misses for the coordinator. Rather, the coordinator is able to manage a private, dense, redundant record of which ports are ready. Aside from the unfortunate data duplication, this optimization contributes greatly to the satisfaction of $\mathbf{G}_{\mathtt{fast}}$. Unfortunately, regardless of how fast `mio` may be, the event must still cross the boundary between threads. In overburdened systems, this also has the consequence of causing context switches, introducing yet more overhead.

Threadless Protocol

If the coordinator has its own 'protocol thread', threads focus on their own work; the coordinator coordinates, and port threads interact with their ports. However, for all interactions that involve one or more ports (which are frequent in practice) we observe that despite the presence of multiple specialized threads, their tasks are not concurrent. Port threads perform external work until they instigate a port operation, and block. They do not wake up until the coordinator wakes up itself, and completes the port operation. Port threads and coordinator cannot know when to act, and must rely on notifications from one another. Threads must wake up and go to sleep frequently.

One pivotal decision of our final design is to attempt to alleviate this problem. If threads of other components must wait for progress anyway, why not

[5] Conceptually this could be in parallel, but the actual implementation the exchange necessitates the use of a *class monitor lock* (a structure for coordinating actions for all instances of a class) to prevent interfering with the protocol thread itself.

have them do the coordination themselves? In our approach, we discard the dedicated protocol thread and reinterpret 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. If only the protocol thread coordinates, there is no risk of data races. In our new model, anyone can be a coordinator; care must be taken 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 $\mathbf{G}_{\mathtt{data}}$, the Java backend's (admittedly intuitive) representation of ports as data pigeonholes is wasteful. For many realistic Reo protocols, data often moves through protocols synchronously, moving from `Putter` to `Getter` without any storage in memory cells in between.

Our implementation introduces a new idea in an attempt to capitalize on this observation: getters fetch their data directly from the source. In this approach, the coordinator does not necessarily handle data itself. Rather, it decides which rule to fire and delegates the task of moving the data to the getters themselves. In addition to skipping a redundant 'data hop' from putter to coordinator, this also facilitates the dissemination of a putter's datum to all its getters in parallel. This change requires extra messaging form the coordinator, as getters are given more responsibility. Where before a signal from coordinator to getter sufficed ('Your datum is ready!'), coordinators must now communicate the location of the getters' source ('Your datum can be found at P!'). Note that this idea is applicable in a context where components are distributed, i.e., they do not all share an address space. In such cases, it becomes vitally important to manage the task of data movement for the sake of performance, which is likely to complicate this optimization further.

4.3.3 Architecture

The protocol is reified in the `Proto` type. Internally, these objects store structures which correspond closely to its imperative form specification type, `ProtoDef`.

As such, one can think of `ProtoDef` as a behavioral specification of a `Proto`. Their differences in structure and contents are a result of them being used for different purposes. `ProtoDef` strives to minimize redundancy in order to simplify parsing, and to minimize the surface for internal inconsistency. On the other hand, `Proto` is structured to facilitate execution at runtime. As discussed in Section 3.3.3, the `build` method is the only user-accessible means of constructing `Proto` instances. Section 3.3.3 from the previous chapter explained how this final translation step is performed. The definition of the resulting `Proto` is provided in Listing 15. To follow, we explain its fields and their role in facilitating the behavior described in Section 4.3.4.

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

Listing 15: Definitions of the most coarse-grained structures of a protocol instance. `Proto` is the entrypoint, composed of `ProtoCr` in the critical section, accessed by only the coordinator, and `ProtoR` outside it, accessed by all.

Critical Region

Section 4.3.2 explains how threads initiating actions at boundary ports of a protocol assume the role of coordinator. It is fruitful to examine the fields of `Proto` in accordance to which roles access them, and how their access is safely controlled. The most coarse-grained distinction is that between fields inside and outside the protocol's lock-protected critical region. This divide is so fundamental, that is is immediately apparent by looking at the definition of `Proto` itself, seen in Listing 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* (See Section 4.3.1). While this is not a task traditionally associated with the coordinator, it's mutually exclusive access between threads necessitates that this structure is protected by the lock.

Spaces

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

4.3.4 Behavior

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

Rule Interpreter

Unlike the Java implementation, Reo-rs moves the specification of the protocol type very late into the pipeline from Reo to the final application. Rather than relying on Reo to generate native application code, in this work we make more extensive use of the virtualization pattern. At runtime, the coordinator traverses rules in data form, performing tasks as a rudimentary *interpreter*. The tasks associated with such a `Rule` correspond closely to the conceptual interpretation of the imperative form. At this granulairty, interactions do not exist explicitly. Rather, the interpreter must perform the work associated with each rule interaction as a sequence of actions which, all together, appear to the

observer as an interaction. For simple rules, it is clear to see how such interactions can be created. For example, consider a simple rule with constraint $P = C$ where P and C are putter and getter ports respectively. Here, P's value is simply moved to C if both ports are ready. As RBF rules become more complex, more actions become necessary to achieve the results of the interaction. Section 3.2 explains how by imperative form's restrictive representation already captures the result of this action-centric breakdown such that the interpreter does not need to compute it at runtime. These actions preserve their interaction-based semantics by behaving as *transactions*, with actions clearly divided into two sequences around an instant where the rule can be thought to *commit*. As long as their effects can be reversed, action prior to the commit can create temporary variables and trigger aborts as they please. This approach is flexible enough to represent Reo's *transform* channels, allowing values to be created and destroyed synchronously by being represented inside the transaction itself.

Minimizing the Bottleneck

Reo-rs shares its centralized locking architecture with the Java backend. 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 set of protocols as its output, though the work on this feature is ongoing. Nevertheless, we consider this decomposition an orthogonal concern and develop it no farther. Reo-rs embraces this central lock, but takes measures to minimize the duration for which it is held. In this section we discuss these measures and how they work together to help satisfy $\mathbf{G_{fast}}$. To structure our reasoning, we identify the tasks a coordinator performs from the moment to acquires the lock (accepting its role), to the moment it releases it (relinquishing the role).

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

2. **Checking Readiness and Memory State**
   Imperative form shares the explicit representation of the synchronization

constraint 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 (i.e., not coordination), a task for which Reo should probably not be optimized. For the protocols without instructions (including `fifo1`, `alternator`, `sequencer`, `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 configuration 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 `drop` 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 identifier from which they must fetch their value. Their task is to locate the corresponding `Space` and contend with an unknown number of fellow getters to complete the movement. The correctness of this exchange relies on the satisfaction of a number of properties:

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

   In the case of a memory cell resource, cleanup takes the form of resetting 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 `(refcount, data)`, and is used for shared, reference-counted ownership of `data`. For this type, coping the pointer to the tuple is not sufficient. Cloning must follow the pointer and increment `refcount`.

3. **One getter moves instead of cloning**
   Data movements represent the transmission of data from source to a set of destinations. Generally, the value is no longer present at the source afterwards. Naïvely, the original must be dropped to complete the interaction. However, it is wasteful and counter-intuitive 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-centric language, and does not offer the interaction we need.[6]

   When orchestrated correctly, we are able to implement a safe move operation between threads by invoking a pair of `unsafe` operations, one on either end. In unsafe Rust, it is possible to copy a value without influencing the original. If not done correctly, this can easily lead to double frees. On the other hand, it is possible to leak resource memory with `forget`, an operation of Rust's standard library which causes the compiler to consider the value moved without invoking `drop`. These pitfalls should be

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

familiar to C programmers, as unsafe Rust gives one the capability to interact with 'raw' pointers in a fashion similar to that of C. Together, these actions constitute the inter-thread move primitive we need.

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

4. **All clones must be complete before the move**
   It is unsafe to move a value before or while performing `clone` on the original. Essentially, every data exchange must proceed in two strict phases: all clones occur in the first, and the move occurs in the second. Consider again the example of type `Arc` by examining this sequence of events that results in undefined behavior: (1) Let x have type `Arc`, containing a pointer to heap region at p. (2) x is moved to binding y. (3) y goes out of scope, it's `refcount` is reduced to zero, and so its heap allocation is freed. (4) `Arc::clone` is invoked with x, which traverses its pointer to memory position p, and attempts to increment `refcount`. p is no longer allocated, and arbitrary memory corruption ensues. To prevent such cases, Reo-rs must take care to order all clones of some value before it can be moved, as the Rust compiler would do.

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

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

1. **Not all getters want data**
   Getters participating as a result of the `get_signal` operation will not return a value. Clearly these getters cannot avoid participating in the mover election, as then nobody would clean up. These getters specialize their interactions by participating in the last election first. The intuition is that

if they lose this election, it is safe for them to return without participating in the mover election; clearly this covers the case of no getters wanting the data. It is also safe to rearrange these elections in this case; these getters have no intention to `clone`, and thus are not a threat to the invariant that required these elections to be ordered in the first place: all clones are complete by the time the last getter is finished.

2. **`Copy`-types can be replicated without `clone`**
   Section 2.2.2 explains how `Copy` marks types for which have a trivial destructor, and are safe for multiple getters to replicate by copying their value bit-wise. This is the case for primitives, and structures composed entirely out of primitives, such as arrays of integers.

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

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

Memory Cells

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

Memory cells differ from putters in that their value persists beyond the lifetime of any individual thread participating in the protocol; consequently, the data itself *must* be stored on the heap. A naïve implementation treats memory cells similarly to putters by continuously updating (i.e., overwriting) 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 predefined types. Instead of shifting the pointer around to a fresh allocation each time, we are able to preallocate 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 reused. 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 Reo's value-passing semantics: moving data between memory cells is expensive. While small optimizations are possible for some circumstances, they are only applicable in a handful of situations. For example, memory cells can swap references to swap their (logical) contents.

Requirements $\mathbf{G}_{\mathtt{data}}$ and $\mathbf{G}_{\mathtt{fast}}$ incentivize a more extensive optimization. Reo-rs intentionally decouples memory cells (including their spaces and their fullness flags) from *storage*, which describes where the contents of the cells is kept on the heap. We observe that Reo protocols perform logical replication of values often, while mutating existing values rarely. As such, many situations exist in which we are able to safely alias values between memory cells by relying on reference counting. We extend the idea of reusing allocations, but rather that fixing them per memory cell, we allow all memory cells of the same data type to draw from a shared pool of reused allocations; this pattern 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 can avoid moving values altogether, instead manipulating only the references within spaces, and reference counters of storage cells. For example, a rule which empties memory cell $\mathfrak{m}_0$ (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 `drop` function 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. Often, this pattern is known as *copy-on-write*. 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. Fortunately, the procedure previously given (in Listing 4.3.4) handles this case correctly.

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. Reo-rs cannot hope to rely on static dispatch to resolve the concrete types of port data, as they are

only discovered later in the moment our `Rule` structures are interpreted. The idiomatic approach to such situations is to rely on dynamic dispatch, which virtualizes the operations on some generic type by adding indirection which is resolved at runtime through the traversal of function pointers. As with C++, Rust uses *virtual function tables* ('vtables') for this purpose. Dynamic objects are stored in place alongside a pointer to the relevant vtable, and operations traverse the table according to a statically-defined layout to resolve the concrete functions. Clearly, this is only possible if the method creating the dynamic object and the operations on it agree on the vtable's contents. To this end, Rust relies on its trait system: dynamic objects are created and interacted with in terms of some trait, which provides it with both an interface and a type. As such, Reo-rs defines a trait `PortDatum`, which defines all the operations belonging to all port data: (1) how is the object laid out in memory, (2) how are objects checked for equality, (3) how are objects cloned, etc. Effectively, we define a *supertrait* of the traits that characterize a type for which all of the operations are defined which Reo *may* require. Concretely, these *subtraits* correspond with the abstract type properties seen in Section 3.3.1. Two problems present themselves with Rust's idiomatic approach to dynamic dispatch:

1. Who defines `PortDatum` for the user's data types? The idiomatic approach is to expose the trait and simply require the user to implement the trait's associated operations to their type. However, if we do not trust the user entirely, some of our desired optimizations become impossible.[7] For example, users must mark their objects as `Copy`, communicating that their shallow representation can be safely copied in memory.

2. How do we express types of `PortDatum` which do not implement all of the subtraits? (e.g., a type which cannot be replicated). One is able to express Reo connectors which will not use these operations, (and thus is is correct not to require them), but we cannot know whether they are used statically.

We solve both of these problems by using an experimental feature the Rust language not yet available in the stable version: *specialization*. With this feature, we solve the first problem by defining `PortDatum` for every conceivable generic type ourselves, with their fields populated as a function of the type's properties. In this manner, `PortDatum` can be made entirely private, benefiting the user in alleviating their need to implement it, and benefiting Reo-rs by guaranteeing it is implemented correctly for every type. This also solves the

[7] This is a limitation of Rust's trait system, which prohibits the inclusion of associated functions and properties for traits used for the creation of dynamic objects. Rust does not support representing them in the vtable. This limitation may be removed in future.

second problem; as `PortDatum` is under our control, we can provide dummy implementations for operations which the type does not define, such that all `PortDatum`-implementor types can use the same vtable layout, despite implementing different sets of `PortDatum`'s subtraits. Conceptually, we can represent undefined functions with null pointers in the vtable. For safety's sake, we instead define default functions whose bodies trigger an explicit *panic*, unwinding the stack and throwing unrecoverable errors. If all goes well, these dummy functions are never executed. However, if a programming oversight results in Reo invoking a dummy function (which implies that the values data type was incorrectly specified), the program crashes with an unmistakable error message. Listing 16 gives a simplified[8] view of the `PortDatum` trait, and its implementation for any[9] data type, `T`.

Elaborating on what was explained previously, Rust's chosen representation of dynamic objects is the *fat pointer*. Each dynamic object is represented as a pair of pointers, one to its *data* (i.e., some structure with fields), and one to its *behavior* (i.e., the vtable). Dynamic objects can be thought to carry their behavior around with them. 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. Reo-rs would repeatedly overwrite these tuples in order to overwrite the *data*, redundantly overwriting the vtable pointer with an identical one. This is a symptom of Rust's design for dynamic objects: concrete operations are reflected per invocation. This approach is detrimental to Reo-rs for two reasons:

1. Dynamic objects are only accessible through their trait interface. Behind this interface, their concrete types are erased. There is no means to check equality between the concrete types behind the dynamic indirection. This ability is required for the `build` procedure (see Section 3.3.3) to ensure that memory is initialized with the expected type and so on.

2. Dynamic objects carry their vtable pointers with them. This increases the size of their representation. For small types, the size increase is (proportionately) significant.

[8] The real trait definition contains more fields, and must perform some manipulations of raw pointers to get around Rust's restrictions on which traits may be used for dynamic dispatch. For this reason, it is important for us to control its definition for any type `T`. These details are omitted for brevity.

[9] In the final implementation, we must include some trait bounds for all `PortDatum` types. `Send` and `Sync` are common Rust marker traits for types that can be passed between threads by value and reference respectively. They are implemented by default for all reasonable types such that users almost never need to consider them [KN18] (they are derived for user-defined types by default also), but this requirement covers some prickly safety pitfalls.

Our solution is to implement our own dispatch system that makes use of Rust's native vtables and dynamic dispatch, but without the above properties. Essentially, we split Rust's fat pointers into their *data* and *behavior* components, using the former as data as usual, and using the latter as a *key* to reflect on the concrete type's behavior as needed. Section 3.2 introduced the `TypeInfo` type, which appears to the user as nothing more than some *identifier* for its type. Under the hood, this value is the vtable pointer itself. Types with identical implementations have identical vtables, and so we are able to compare equality of the vtable itself to check equality of the erased (concrete) type. Listing 17 shows how function `TypeInfo::of` provides the user with the only means of creating a `TypeInfo` for some `T`. Only in the `ProtoDef` type does Reo-rs accept the user's provided `TypeInfo` directly, as it would be unsafe to rely on the user providing some `T` with a matching `TypeInfo`. Instead, the API of Reo-rs includes one layer of static dispatch into the library where necessary such that the creation of the `TypeInfo` can be trusted. For example, when populating some `MemInitial` structure with the initial values of memory cells (as described in Section 3.3.1), values can only be input using the provided `with` function. The user uses Rust's safe dispatch system, oblivious to Reo-rs translating their concrete objects into dynamic ones behind the scenes. For example, the user might do: `MemInit::default()::with("X", Foo::new())`, resulting in Reo-rs storing a dynamic `PortDatum` object named X, with `TypeInfo::of::<Foo>()`.

Listings 27 and 28 in the appendix demonstrate how `TypeInfo::of` appears in the generated binary.

4.4 Requirements and Guidelines Evaluated

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

$\mathbf{R}_{value}$ Values passing through ports preserve value-passing. This is achieved even in the presence of reference-passing optimizations 'under the table' by leaning on the same philosophy that Rust uses to prevent data races: prohibit mutable aliasing. Objects are only aliased (accessible via multiple bindings) if they are be identical. Section 4.3.1 explains how protocols limit aliasing to their internals by relying on value-passing port operations. On the other hand, Reo-rs aliases values, but only until they are mutated. Section 4.3.4 explains how memory values are safely aliased.

Section 3.2 explains how Reo-rs interprets an imperative form protocol description at runtime, relying on a transaction-like model to safely allow

the creation of new values to be incorporated in synchronous interactions. In this manner, protocols whose rules create and reason about temporary values can be faithfully represented.

R_{init} Section 4.3.1 explains how users are shielded from the granular initialization procedure. Reo-rs exposes an API with explicit constructor functions `build` and `claim` of protocols and ports respectively. Protocol objects are extensively customizable by the expressiveness of imperative form, with which `build` is parameterized. At the same time, these structures are internally-consistent by `build` as the only user-facing means of instantiation.

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

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

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

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

```rust
trait MaybeCopy { // private helper trait
    const IS_COPY: bool = false;
}
impl<T> MaybeCopy for T {} // default case. NOT COPY
impl<T: Copy> MaybeCopy for T { // specialize for COPY TYPES
    const IS_COPY: bool = true;
}
////////////////////////////////////////////////////////////////////
trait MaybeEq { // private helper trait
    fn maybe_eq(&self, _: &Self) -> bool {
        panic!("This type cannot check equality!")
}   }
impl<T> MaybeEq for T {} // default case. NOT PartialEq
impl<T: Eq> MaybeEq for T {
    fn maybe_eq(&self, other: &Self) -> bool {
        return self == other
}   }
////////////////////////////////////////////////////////////////////
trait PortDatum { // main PortDatum trait. also private
    fn is_copy(&self) -> bool;
    fn eq(&self, other: &Self) -> bool;
}
impl<T> PortDatum for T {
    fn is_copy(&self) -> bool {
        <Self as MaybeCopy>::IS_COPY
    }
    fn eq(&self, other: &Self) -> bool {
        <Self as MaybeEq>::maybe_eq(self, other)
}   }
```

Listing 16: Using Rust's specialization feature to define `PortDatum` (simplified) for every generic type `T` by relying on `T` always implementing helper traits `MaybeCopy` and `MaybeClone`. `MaybeCopy` can be implemented for any `T`, defining a default behavior in one block, and then overriding it for a more specialized behavior in the other. The Rust compiler will resolve which block to use based on the static properties of `T`, deriving a `PortDatum` implementation with precisely the desired definition. In this manner, `PortDatum` can be made inaccessible to the user, allowing Reo-rs to trust that it was defined in the expected manner. The helper traits are necessary to satisfy the requirements of the specialization feature: there must be a strict ordering on the specificity of implementation blocks for the same trait.

```
1  struct TypeInfo(VtablePtr); // VtablePtr field is private. User can only interact with
   ↪  `of` function.
2  impl TypeInfo {
3      pub fn of<T>() -> TypeInfo {
4          // 1. fabricate bogus (uninitialized) data pointer to some type T.
5          let x: Box<T> = unsafe { MaybeUninit::uninit().assume_init() };
6          // 2. SAFE cast to trait object (Rust appends PortDatum vtable pointer for T)
7          let fat_x = x as Box<dyn PortDatum>;
8          // 3. convert to "raw" dynamic 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         return 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.

Chapter 5

Benchmarking

Chapters 4 described protocol objects generated by Reo for the Rust language. In this chapter, we evaluate the performance of these objects at runtime. To begin, we place their performance in context with their Java counter parts, and with handcrafted Rust code in Section 5.2. Their performance characteristics are examined more closely in Section 5.3, investigating the effect the implemented optimizations, and examining the relationship between properties of the input specification and performance at runtime.

5.1 Experimental Setup

For experiments throughout this chapter, we used a machine with the properties given in Table 5.1. Our experiments are available for inspection in the source code in `experiments.rs`. For all tests, the Rust compiler (`rustc`) used was version 1.38.0-nightly (bc2e84ca0 2019-07-17) contemporary with stable release version 1.36.0 (a53f9df32 2019-07-03). Section 4.3.4 explains the need for this compiler version to support currently-experimental features: (1) `specialization` and `raw` in facilitating type reflection at runtime.

Experiments were run using the inbuilt testing functionality of Rust's package manager (i.e., invoking `cargo test`), using release-level compiler optimizations. All measurements shown are the mean of all measurements of runtimes for which protocol structures were built A times, and then runs were measured B times for each, i.e., the mean of $A \times B$ total repetitions. The values of A and B are specified in the captions of figures or tables in which the measurements appear. If left unspecified, A and B are 100 and 1000 respectively.

Component	Properties
Operating System	Windows 10.0.2.1000
Processor	4×2 intel core i7-7500U CPU (64-bit) at 2.70 GHz
Memory	12Gb DDR4 2400MHz
Storage	Micron 1100 SATA 5122 GB SSD

Table 5.1: Component properties of the machine used for experiments in this chapter, included for the sake of reproducibility.

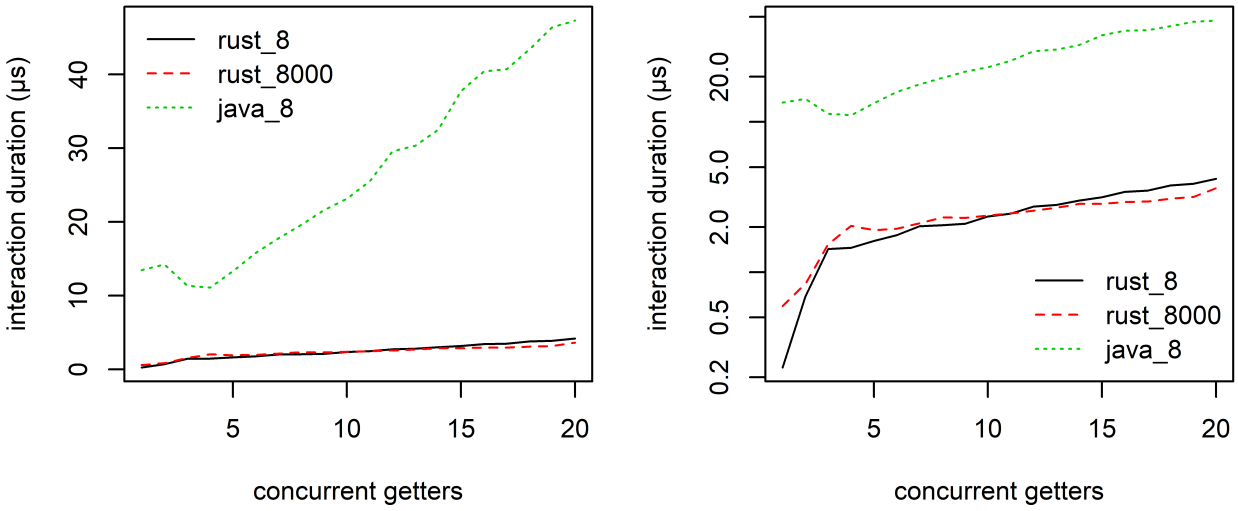
5.2 Reo-rs in Context

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

5.2.1 Versus the Java Implementation

We begin by making the most intuitive benchmark to get an understanding of how effectively Reo-rs has been optimized for its task; we compare it to the work of the Reo compiler's Java code-generator. This comparison spans two vastly different systems with different goals, but also compare a memory-managed language to a system's language. The reader should bear this in mind when interpreting the measurements, and focus primarily on the differences in performance relative to other measurements of the same system, i.e., we are most interested in comparing the 'shapes' of response curves. As our test scenario, we have a set of N getters repeatedly copying some memory value M, retained inside the protocol from initialization. By involving a contended resource, we are able to test and compare the scalability the generated programs, both in terms of number of ports and the size of the transmitted data.

The unfairness of our comparison cuts both ways, as there is not a clear means of comparing the transmission of large values; the Java version relies entirely on object aliasing, effectively implementing different semantics. For Java, the size of values transmitted is largely irrelevant. We begin by a comparison on the only common ground; Figure 5.1a shows both Java and Rust are transmitting pointer-sized objects in the `fetch` connector. Aside from the order of magnitude difference in runtime, we observe a different 'shape'. Reo-rs is observed to be significantly faster in the case of a single getter. This is easy enough to explain; the type relied upon for protecting the coordinator's critical region is `Mutex` from the `parking_lot` crate, which provides implementations of these kinds of concurrency primitives. `Mutex` is documented as having a 'fast



(a) (b)

Figure 5.1: Comparison of interaction time for the `fetch` connector for both Java and Rust backends moving small values. `rust_8` and `java_8` both move a payload of 8 bytes (to match the reference size of the 64-bit JVM). `rust_8000` gives an example of how the runtime of Reo-rs can change with respect to modest changes in data size. The two sub-figures mirror one another except for the linear and logarithmic y-axes respectively.

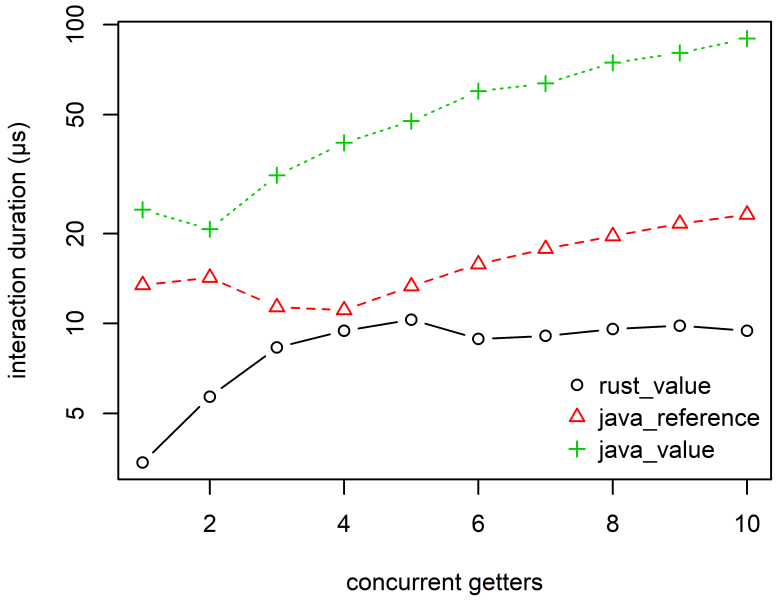


Figure 5.2: Comparison of interaction time for the `fetch` connector for both Java and Rust backends moving 64-kilobyte-sized data. The Rust backend moves the datum by value, while the Java parallel `java_reference` aliases the object (moving by reference). This backend does not support value-passing that preserves Reo's semantics. We achieve safety here by the coordinator injecting `clone` operations of byte arrays to mirror Rust's bit-wise copy, called `java_value` . Note the logarithmic y-axis.

path' optimization for when the lock is acquired uncontested. Runs with one getter are thus able to take advantage of this optimization every time.

Figure 5.2 attempts to draw the same comparison as before, but in the case of large data types. The Java-generated protocol objects do not do true value-passing. It is out of the scope of this project to attempt to implement this efficiently. Instead, we reason about the possible measurements we might observe for a hypothetical, correct value-passing Java implementation. In the figure, the region is bounded by (existing, unsafe) reference passing on the one hand, and naíve value-passing on the other. The latter is implemented as a modification to the protocol component's `run` method: values send to getters are first `cloned`.

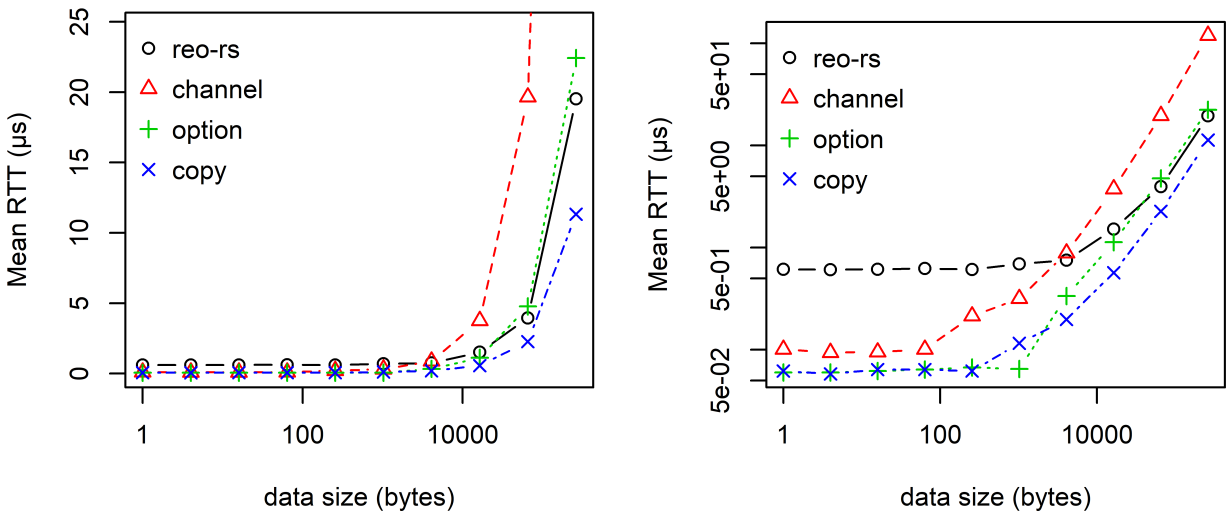
5.2.2 Versus Handcrafted Programs

Clearly, Reo-rs cannot outperform hand-optimized Rust on a case-by-case basis; whatever Reo-rs does, the hand-optimized code can mimic and specialize to surpass (or at least match) its performance. Reo-rs has the burden of balancing performance with other concerns such as flexibility and safety. Here, we measure the performance gap between Reo-rs and the handcrafted implementation of a small set of test protocols.

Firstly, we examine a case for which Reo-rs is expected to do poorly: we apply Reo-rs to a very simple protocol. To make matters worse, we perform the experiment in a circumstance in which explicit synchronization bookkeeping is unnecessary: port operations are accessed sequentially. Concretely, we examine the `fifo1` connector. At these small scales, it matters considerably how we perform our optimization. Figure 5.3 shows runtimes for Reo-rs compared to three handcrafted solutions. `channel` is the most intuitive solution, relying on the ubiquitous `crossbeam` channel for its efficient message-passing channels. The `option` version represents a memory cell as the the `Option` type from the Rust standard library; it can implement the reading and writing of a memory cell with methods `replace` and `take`. Finally, `copy` uses unsafe Rust, shirking Rust's idioms to access a pre-allocated heap buffer directly. We see that the runtime of Reo-rs is never the fastest solution. Particularly for small port-values, the simplicity of this protocol is not worth the overhead Reo-rs incurs by traversing rules, comparing guards and so on. Still, it is surprising that Reo-rs overtakes the `crossbeam` channel, whose implementation clearly does prioritize the efficient movement of very large values.

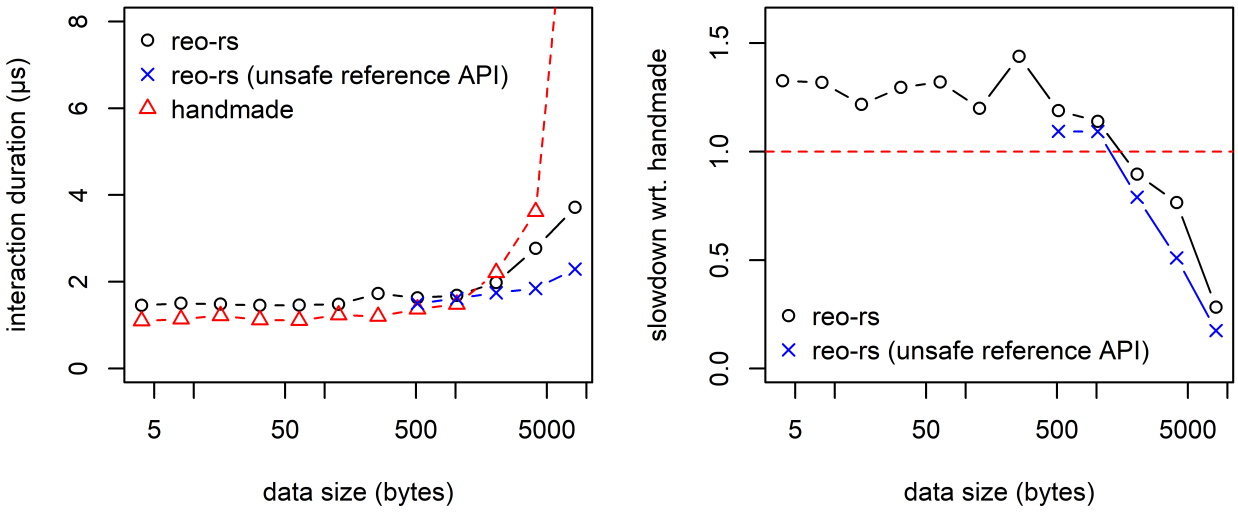
The comparison become more interesting when we apply Reo-rs to less trivial connectors, as they begin to necessitate synchronization. The *alternator2* is a canonical Reo circuit that routes messages from putters $\{P_0, P_1\}$ to getter G in an alternating fashion (starting with P_0). The semantics of this connector are subtly different from that of a *sequencer*; the first value is transmitted once all ports are synchronously ready, after which the second value is transmitted asynchronously.

Listing 5.4 shows how Reo-rs fares against a simple handcrafted solution which maps Reo-rs channel primitives and the connector's synchronization constraint to `crossbeam`'s channels and the `Barrier` in Rust's standard library, implemented to closely correspond with the specification, available for inspection as Listing 25 in the appendix. Figure 5.4b shows that, as before, the performance of Reo-rs far surpasses that of our handcrafted solution for very large values, likely as a result of relying on `crossbeam`, whose slowdown was previously observed in Figure 5.3. More interesting is the observation that while Reo-rs is still inferior for small values, the gap has closed significantly across the board, suggesting that the synchronization overhead is more similar in both



(a) (b)

Figure 5.3: Time from beginning of `put` to end of `get` in connector `fifo1` compared to handcrafted Rust alternatives of various sorts. `channel` is intuitive, using a `crossbeam` 1-capacity buffered channel. `option` stores the temporary variable in an `Option` type, and writes and reads from it using `replace` and `take` operations. `copy` uses unsafe rust to perform reads and writes to an allocated heap buffer directly. In both figures, the x-axis is logarithmic. The second plot displays information with a logarithmic y-axis also. Results are the mean of 100×1000 measurements



(a) (b)

Figure 5.4: Mean interaction duration of the Reo-generated `alternator2` connector in comparison to an intuitive implementation based on channel primitives in the `crossbeam` crate, and the `Barrier` from the Rust standard library showing (1) absolute runtimes, and (2) the slowdown factor of Reo-rs compared to the handmade code. Note the logarithmic x-axis. Measurements are the mean of 500×5000 repetitions.

versions. For completeness sake, and to put out best foot forward, we include the effects of the C-like `unsafe` API variants of Reo-rs port operations explained in Section 4.3.1.

The alternator circuit is still simple enough that the handcrafted implementation is expressed in Rust code that is only a few lines long. It is in our best interest to compare the performance of Reo-rs to handcrafted solutions closer to the expected level of complexity for the use case of Reo. However, as programs become more complex, it becomes more difficult to design handcrafted programs such that the results facilitate fair, and meaningful comparisons. As their complexity increases, there are more opportunities for optimizations which necessitate making objective design choices. We leave a more complete analysis of the performance of Reo-rs to handcrafted programs to future work. For now, we conclude that Reo-generated Rust is unlikely to outperform handcrafted programs for protocols at the expected level of complexity, but will be generally competitive with reasonable implementations of non-trivial protocols.

5.3 Overhead Examined

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

5.3.1 Parallelism Between Interactions

For connectors as simple as `fifo1`, overhead is overhead; there source of the overhead is clear, and does not require thorough investigation to understand. However, we are particularly interested in understanding how this overhead is partitioned; as connectors become more complex, different parts of this overhead impact performance in different ways.

Section 4 explains the nature of the coordinator role, and how its operations are performed holding the lock for the `Proto` instance which connects ports as their common communication medium. Table 5.2 shows measurements for an experiment that attempts to understand which proportion of our overhead is incurred *inside* the critical region, i.e., by the coordinator. In the case of this experiment, our protocol has rules for movement which can be rendered as a bipartite graph, allowing data flow from any putter in $\{P0, P1, P2\}$ to any getter in $\{G0, G1, G2\}$. As explained in Section 4.3.4, movements such as these do not buffer the data elements inside the protocol; getters take values from putters directly. As a consequence, putters are both the first and last to parttake in any of our rule firings' data movements. The table shows the mean duration for which each putter was involved in such a firing. Along with the total duration of the run, we are able to compute to which extent these putters were able to work in parallel. We distinguish between four cases, corresponding to rows in Table 5.2. The first three cases do not involve the `clone` operation, and are observed to have insignificant differences for all measurements. For this experiment, with modestly-sized values, we conclude that there is no large difference in performance between these three cases:

move Values are moved from putter to getter synchonously.

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

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

The final **clone** case attempts to observe the effects of intentionally delaying getters outside of the lock by necessitating the use of an explicit `clone` operation whose duration is artificially lengthened. At this scale, `sleep` calls were out of

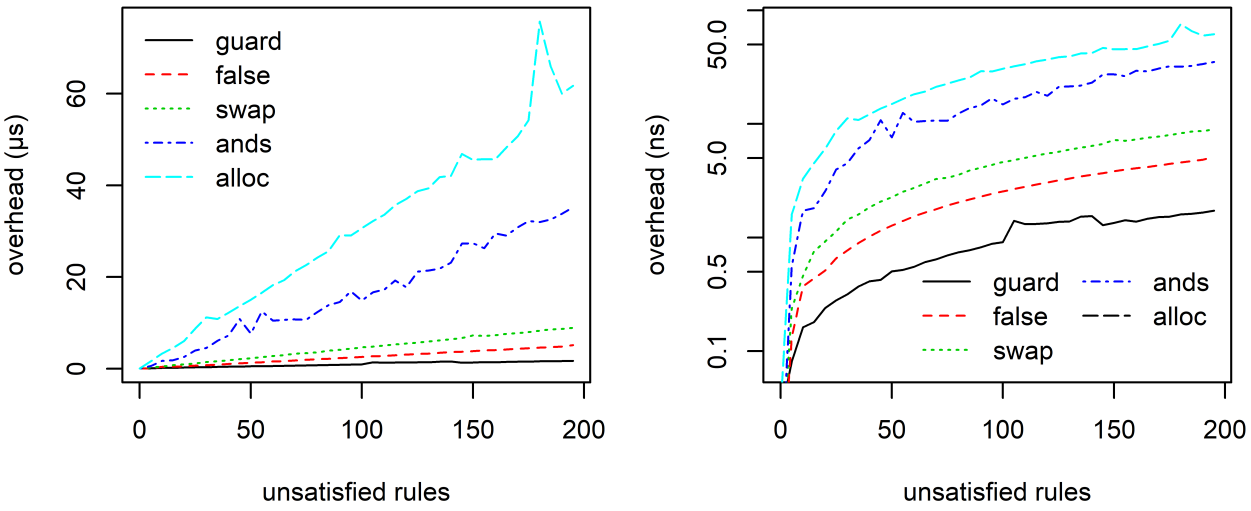
	mean active time			run	mean
	p0	p1	p2	duration	parallelism
move	2.68µs	2.594µs	2.993µs	31.1705ms	2.652
copy	2.737µs	2.4µs	2.673µs	28.7161ms	2.720
signal	2.351µs	2.282µs	1.943µs	24.7852ms	2.653
clone	4.451ms	4.461ms	4.416ms	44.609s	2.988

Table 5.2: Runs of 3 putters greedily sending their 2048-byte data directly to any of 3 getters, 10 000 times. The rightmost column computes how many putters were involved in an interaction at any given instant, i.e., indicating the extent to which their work was performed in parallel. This test was performed with 4 variants, differing on the properties of the data and whether the putter retained the original. Measurements are the mean of $100 \times 10\,000$ repetitions.

the question, as its variability overwhelms any meaningful measurements. Instead, `clone` perform thousands of chaotic integer computations on the replica before returning it. For these runs, putters retained their original values, but the datum was not marked with the `Copy` trait. In all cases, we observed that even at this coarse granularity, there was significant parallelism. For the majority of the time, new rules were able to fire whilst interactions were being completed outside of the critical region. The final case in particular was within a small rounding error of perfect parallelism, owing to the high cost of `clone`.

5.3.2 Time Inside the Critical Region

The previous section, we saw an experiment with data moving between putters and getters. These runtimes included the putter's time spent not only on the movement itself, but also on the time spent in the role of coordinator. Here, we examine the work that pertains to this role and how changes to the definition of rules influence overhead. Figure 5.5 shows the overhead incurred by a coordinator that traverses unsatisfied rules before finding one to fire. It is apparent that in all cases, the overhead scales linearly, as expected. The time taken to evaluate the satisfaction of a rule varies greatly dependent on its definition; the evaluation of a rule involves many different operations mirroring the intricacies of the imperative form it models, described in Chapter 3. To represent the possibility space, the figure shows measurements for a simple protocol which encounters replicas of one unsatisfied rule repeatedly, where its nature comes in four distinct variants:



(a) (b)

Figure 5.5: Overhead as a result of evaluating a sequence of identical unsatisfied rules before firing something. Experiment is repeated for four variants of unsatisfied rules varying in the complexity of the operations they before before being deemed unsatisfied. The two sub-figures show the same information, with (b) representing it with a logarithmic y-axis to accentuate the small-scale differences. Measurements are the mean of $100 \times 10\,000$ repetitions.

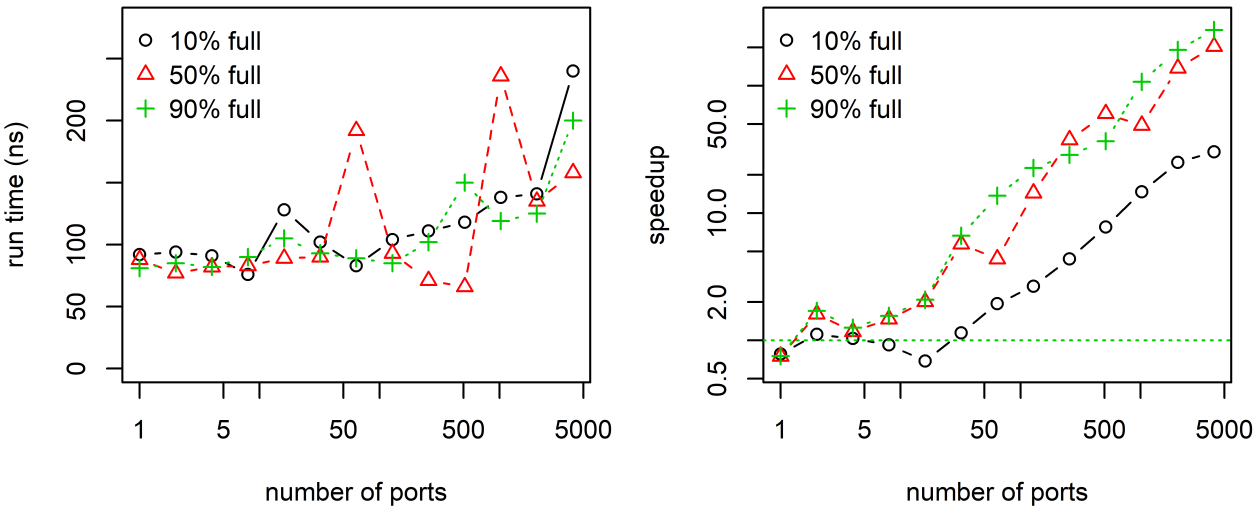
guard A rule where some port is not ready. This is detected almost immediately by cheap bit vector operations, as explained in Section 4.3.4. Evaluation takes 8.76ns.

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

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

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

These results meet our expectations. Rules can be arbitrarily complex, and perform an arbitrary amount of work before concluding that they are not sat-



(a) (b)

Figure 5.6: Run time of the `is_subset` operation for bit vectors and canonical hash sets. This operation is used very frequently by the coordinator to determine whether a rule is satisfied. Figures show (a) run times for the bit vector in response to a changing maximal element, i.e., number of ports, and (b) the speedup of the bit vector in comparison to the hash set. Note the logarithmic axes.

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

The efficiency of the bit vector subset operation is key to the speed of the coordinator. This is the first first three things evaluated for each and every rule, checking whether all involved ports are ready, and if all the relevant memory cells are full or empty. The bit-set capitalizes on how unusually often we need this operation. Figure 5.6 shows how bit vectors are very efficient at checking whether one is a subset of another. Here we see the time taken to evaluate the positive case, representing the best-case scenario for our speedup. It is guaranteed to occur at least three times every time a rule fires. Figure 5.6a

shows how low the cost of the operation stays, even when there are very many ports involved. Figure 5.6b shows how significant the speedup over the subset operation of canonical `HashSet` type. Admittedly, the majority of realistic Reo circuits are on the low-end with respect to number of ports; if nothing else, this is a result to encourage the development of more complex connectors. Observe that the cost of the operation is agnostic to the fullness in the case of the bit-vector. This is not so for the hash set, for which a fuller hash set makes for a more expensive operation.

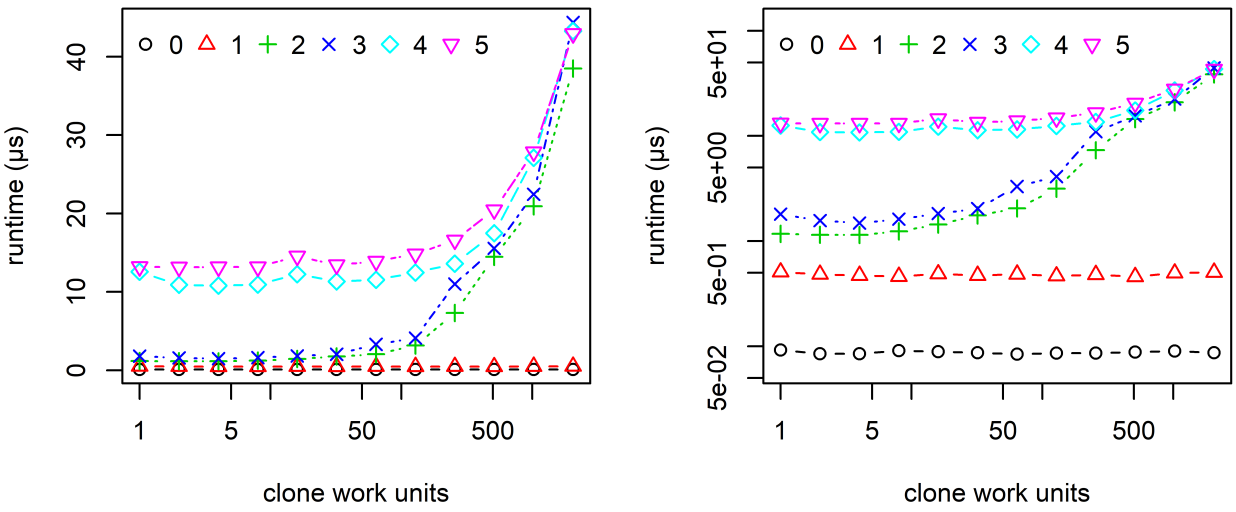
5.3.3 Parallelism Within Interactions

After data exchanges are initiated by the coordinator, the protocol's lock is released. Time spent exchanging can therefore only impact the threads that play a part directly. Figure 5.7 shows measurements of the `simo` ('single input, multiple output') protocol, which synchronously distributes a putter's datum to a set of getters. Figure 5.7 shows mean interaction times from the putter's perspective, measured in response to the cost of the data type's `clone` operation and the number of recipient getters. For the sake of the experiment, we use an artificial `clone` operaton as before, to simulate a computation of the desired intensity. We use an arbitrary 'work unit' as a relative metric of this duration. It's absolute meaning is not necessary; all that matters is that it is defined such that its contribution to runtime is proportionate.

We observe that with fewer than two getters, the runtime does not scale with the work units. Section 4.3.4 explains that amongst a set of getters, one is elected the *mover*, both responsible for freeing the putter and given permission to move the putter's original if possible. The vast difference made by having any getters at all can be explained by the implementation of the `Mutex` type protecting the protocol's critical region. With only one interacting port, the mutex lock is always uncontested, and able to take the 'fast path'.[1]

For few work-units, the duration of the interaction was always greater the more getters were involved. Counter to our expectations, this was not a linear relationship. The precise reason for this is uncertain, but owing to its repeatability, we conclude that it's a property of the system used for testing having to share physical cores. Regardless, we observe that for all cases with numerous getters, their durations converge toward more costly clone operations. Figure 5.8 confirms that as the parallelizable clone-work increases in significance, getters parallelize their work more effectively.

[1] Our `Mutex` comes from the `parking_lot` crate. The relevant documentation is found at `https://amanieu.github.io/parking_lot/parking_lot/struct.Mutex.html`.



(a) (b)

Figure 5.7: Mean interaction duration from the perspective of the putter into the `simo` connector. Results are distinguished by how many getters synchronously acquire the putter's datum in each interaction. Results are shown in response to the cost of the `clone` function, in arbitrary *work units*. Note the logarithmic x-axes in both figures, and the logarithmic y-axis in (b). Measurements are the mean of 100×3000 repetitions.

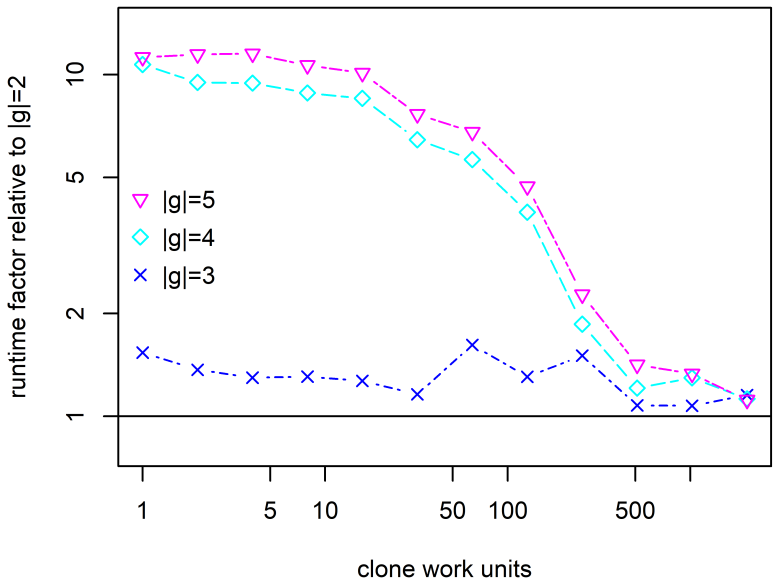


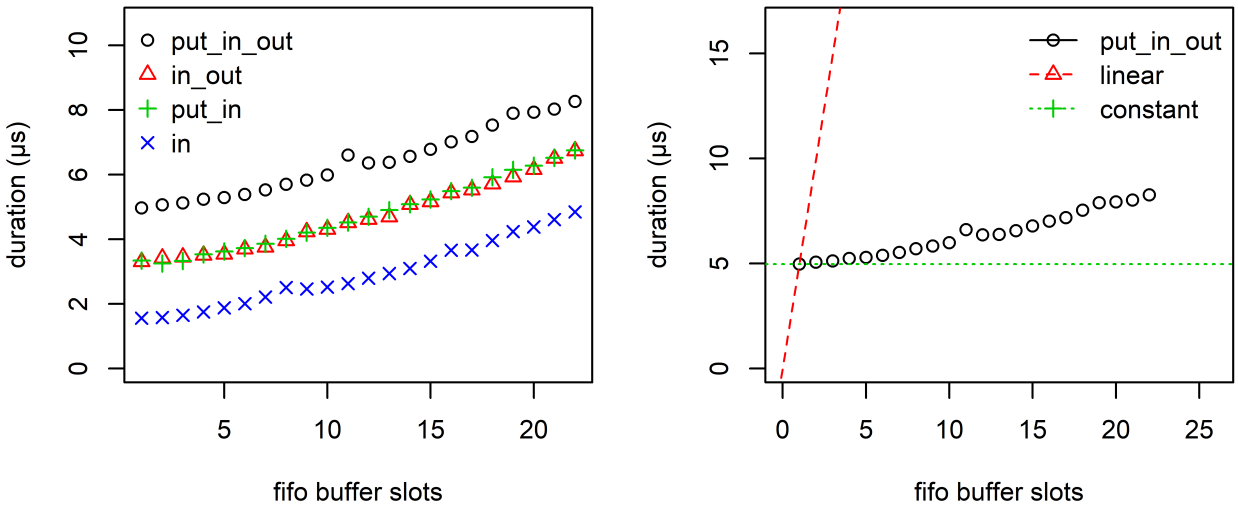
Figure 5.8: Mean interaction duration in the `simo` connector, showing slowdown relative to that of runs with two getters. Results are shown in response to the cost of the `clone` function, in arbitrary *work units*. Note the logarithmic axes. Measurements are the mean of 100×3000 repetitions.

5.3.4 Reference-Passing Optimization

Finally, we perform an experiment to verify that the reference-passing optimization described in Section 4.3.4 is working as intended. Figure 5.9 shows the results of the mean time taken for a datum to pass through a `fifoN` connector. This protocol makes trivial use of an `m`-long chain of memory cells. Values originate as input on one end, shifting between cells from head to tail, and finally being output at the tail. All measurements are dominated by the work of moving the 2^{13}-byte values in memory from one place to another. `shift_get` represents the most intuitive run, where values are moved twice: once *into* the protocol's storage, and once *out* to the getter. For this protocol, the Rust compiler failed to optimize the logical data-movement of the safe API's `put` operation. Runs using this safe variant are prefixed with `put_`, and include an additional value-movement.

As expected, runtimes in Figure 5.9a are seen stratified according to the number of movements they perform. In all cases, longer chains of memory cells indeed require more time (preventing the runtime to be constant with respect to `m`), but the overhead is relatively small, and does not appear to be affected by the baseline cost of movement; this is expected, as the cost of reference-passing is unrelated to the value's size, or data type in any way.

Figure 5.9b shows how the cost of reference-passing compares to the best- and worst-case scenarios for the response of interaction time to the length of the chain. Previously we have observed that Reo-rs experiences some constant overhead per port interaction (e.g., in Figure 5.3b), so interaction time would likely remain sublinear even if it performed value-passing between memory cells naïvely. However, in most examples (including this one) we can safely presume it's slope would be far steeper than it is now.



(a) (b)

Figure 5.9: Round trip time (RTT) of a 2^{13} byte value through a `fifoN` connector with m ranging from 1 to 20, measuring the time taken from the start of the `put` into the head of the chain, to the end of the `get` out of the tail. (a) The experiment was repeated using variations of port operations to control the number of memory copies. `put_*` runs move the datum using the safe value-passing API, and others use the unsafe C-like reference-passing API. `*_get` runs acquire the output by value, while others participate in synchrony by acquiring a signal. (b) `put_in_out` is compared to the best- and worst-case scenarios for interaction times in response to m. Measurements are the mean of $100 \times 10\,000$ repetitions.

Chapter 6

Generating Static Governors

The cumulation of the previous chapters describe a standalone contribution to the Reo compiler. We have seen that users are able to generate Rust source code from a Reo specification (Chapter 3) which may be used in conjunction with their own Rust code to behave according to the corresponding protocol specification at runtime (Chapter 4). The programmer is able to rely on these protocol objects to constrain the behavior of their program at runtime such that no deviations from the protocol specification may be observed.

In this chapter, we design the *governor generator*, a tool to augment the user's Reo-coordinated Rust programs. As with the Reo compiler before, this tool generates a Rust source file from the Reo protocol specification which the user may import as a dependency. Within, a Rust type which corresponds with a *governor*. If integrated into the implementation of the user's components, the Rust compiler will enforce that the user's code does not inhibit the behavior of the system at large. In effect, users are provided an ergonomic means to opt into their Rust compiler enforcing liveness properties of their programs.

Section 6.1 provides the formal definition of governors, and how they relate to protocols, components and liveness. Section 6.2 describes the problem the **governor generator** aims to solve from the perspective of a Rust programmer relying on Reo to coordinate their components' communication. Section 6.3 gives a high-level overview of our corresponding solution. Section 6.4 explains in detail how the governor generator works, concentrating only on what is required to make it minimally functional. Finally, Section 6.5 elaborates on the means by which the solution is made ergonomic and practical.

6.1 Governor Defined

A Reo protocol specification defines which interactions are permitted between its boundary ports. In other words, protocols constrain the behavior of any system in which they are a component. This is the result of Reo's semantics, which guarantees that the behavioral constraints of a system are the composition of that of its components, at any granularity. However, protocols do not prevent their boundary components from adding constraints of their own. As a consequence, the definition of a component viewed in isolation does provide any guarantees on which interactions *are* observable in the system at large. To make this possible, a component needs some knowledge about the behavior of the system beyond its own ports. Fortunately, Reo specifications take us halfway there, as they make explicit which behaviors they permit in an unambiguous form that can be transformed, and can be inspected. To proceed, we define the *governor*.

Protocol P with interface ports I_P defines G_{P,I_P}, its governor with respect to its interface ports. G_{P,I_P} shares interface I_P with P itself, and also constrains the interactions between them in the same manner as P. However, all port actions in G_{P,I_P} are the complement of those in P, i.e., all `put`s are `get`s and vice versa. G_{P,I_P} characterizes the behavior of a component that, if connected to P with interface I_P, would result in a composite system with the same behaviors of P itself. In other words, G_{P,I_P} describes the behavior which P would permit to occur, from the perspective of a component interfacing with all of P's ports.

Governors are generalized such that they are defined for any subset of a protocol's interface ports, i.e., a protocol P with interface I_P defines a set of governors $\{G_{P,X} \mid X \subseteq I_P\}$. Governors characterize the reverse-oriented behaviors as before, but such that these behaviors are *projected* onto the governor's own interface; projection is defined precisely by Baier et al. [BSAR06], and is explained in Section 6.5.1 when it is needed. Here, it suffices to say that the behavior as constrained by the governor omits the actions of all ports not in the governor's interface. This captures the intuition that the governor only specifies the interactions of its own boundary ports, not constraining the actions of other ports at all. As such, the behavior of a governor is at most as constrained as that of the protocol itself; for every port not included the interface, the governor potentially becomes more 'lenient'. This is made clear for the trivial case of a governor with the empty interface, which has no constraints whatsoever.

The utility of this construction is that it provides a means by which a component interfacing with some protocol P with any of its ports may compute precisely which behavior the protocol permits. By relying on the fact that protocol objects will facilitate any and all permitted interactions as possible at runtime, this boundary component is able to predict precisely the behavior the

component that arises from its composition with the protocol. In the simplest case, one is able to enforce that the resulting behavior matches the constraints of the protocol; equivalently, the protocol's boundary components do not contribute any behavioral constraints that the protocol itself did not define already. For such systems, one may understand the entirety of the system's coordination behavior by inspecting the definition of the protocol alone. In these cases, we say that the component is *adherent* to the protocol.

6.2 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; Section 4.3.2 explains that, while external ports may instigate a `put` or `get` at any moment, the coordinator will complete the operations only once the specification allows it. In this way, coordinators possess a crucial subset of the features of governers: aligning the timing of two actions that compose an interaction. Unfortunately, in the properties of the realm of sequential, action-centric programming itself implicitly imposes constraints on the behavior of the system: `put` or `get` block until their interaction is completed, and no subsequent code (potentially, other port operations) will occur until they do. This is beyond the capabilities of the coordinator to influence.

In the context of application development, this has an interesting consequence; the behavior of the system is influenced by the behavior of (potentially) all of its components. This is sensible in theory, but becomes unwieldy in practice. Even small changes to the behavior of a compute component influences the system's behavior in unexpected ways, as we are not used to thinking about synchronous code as a composable protocol, nor are we able to intuit the outcome of the composition. For example, Listing 18 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 adherent to the former, but not the latter. By definition, `sync` fires when both `p` and `g` are ready, but `transform_rot` does not `put` until the `get` is completed; effectively `transform_rot` constraints the behavior at its ports by imposing an ordering. 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. 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. In the worst cases, an innocuous change adds a new constraint such that no next interaction exists, observed at runtime as deadlock.

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

Listing 18: A function in Rust which can be used as a compute component in a system, connected to a protocol component.

6.3 Solution: Static Governance with Types

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 manage its boundary components, we empower the boundary components with the ability to manage *themselves*. To make this possible, we expose the protocol's specification to their components; specifically, we create a means by which we are able to generate the relevant governor for a given protocol and interface. The resulting governors ensure that any implementation is adherent to the protocol, by prohibiting it from adding any new constraints. Counter-intuitively, this takes the form of disallowing the component from performing the 'wrong' port operation, as the resulting blocking behavior would inhibit its ability to perform the 'right' port operation.

There is a vast possibility space for facilitating this enforcement on the users code given a governor as it is defined in Section 6.1; any solution that results in protocol adherence is sufficient. For example, a human with a keen eye and a steady hand would be able to check protocol adherence manually, by reasoning about the behavior of all components involved. Instead, we opt for a more ergonomic approach that leverages a tool the user is guaranteed to have at their disposal anyway: the Rust compiler. Our solution involves the interplay between two novel facets of the Rust language that follow from its affine type system;

1. We are able to model the protocol's underlying automaton (see Section 2.1.3) as a type-state automaton (see Section 2.2.3), such that the Rust compiler is continuously aware of the protocol's **state** throughout control flow of the user's code wrt. its execution at runtime,

2. and we are able to protect the API of Reo-rs's port objects (see Section 4.3.1) such that the Rust compiler statically allows their use only if doing so in the current **state** preserves protocol adherence.

From the user's perspective, they are able to opt into the Rust compiler enforcing protocol adherence in their component code. Concretely, the are able to modify their implementation such that it takes the form of a function with a declaration (i.e., type signature) as defined in the Rust dependency generated by the governor generator. The user defines the function with `put` and `get` operations interspersed with arbitrary Rust code as usual. However, any port operations that would violate protocol adherence will result in the Rust compiler generating a static type error; if their implementation is completed without type errors, the user can be certain that their component's behavior at runtime will be protocol adherent.

6.4 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.

6.4.1 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. For the sake of consistency with the previous chapters, we primarily use RBAs when reasoning about governors.

We extend this idea to the first of the two tenets of our design, as they appear in Section 6.3. Namely, we model the current configuration of the protocol as a type, updating it throughout our control flow such that it always corresponds with the protocol's configuration. From the user's perspective, they have precisely only token type in scope at all times. As defined previously, we are able to simulate 'updating' this type by overwriting it with a replacement token as the return result from a functions whose behavior alters the type. Concretely, our transition function consumes the old token (taking it by value as a parameter), and produces the new token as part of its return value. It follows that at all times we have precisely one token, whose type corresponds with the configuration of the protocol.

6.4.2 Rule Consensus

The protocol works by firing rules at runtime which correspond to those of the RBA which defines its Reo connector. Section 6.4.1 above explains how various compute components are able to proceed in lockstep with the protocol's RBA in a type-state automaton of their own. For deterministic RBAs, this is easy enough; everyone can trivially know which action occurs next, and they can transition through configuration space independently, safe in the knowledge that their representations of the run will stay aligned. This ignores temporary misalignments in time for transitions in which the compute component does not communicate with the protocol; for these cases, one may work ahead, leaving the other in a previous state. However, they will catch up eventually when they both reach a transition that involves them communicating (which is ultimately all that matters). This process becomes more complicated when the protocol can reach configurations with multiple choices for the next transitions. Without a priori agreement on how these situations are handled, the choice is defined to be nondeterministic. Clearly, all is well as long as all parties agree on this choice; problems only present themselves when compute components and protocols disagree on what may happen next. If the view of a governor

are out of sync with its protocol it is generally unable to guarantee that the actions it permits are adherent, or it may prohibit something it ought not to, resulting in unintended constraints of the intended behavior (in the worst case, deadlock). Clearly, this is a problem of consensus.

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

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

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

[1] Early versions of our implementation indeed enumerated these types with a relatively effective powerset construction. However, it was unable to avoid explosion if there simply were many solutions to be found. The nail in the coffin was the changing from the exponential base from 2 to 3 as a result of the modification explained in Section 6.5.2.

day, this may be ideal fore representing these `StateSets` with minimal code generation.

Until anonymous sum types are supported, our solution to the problem of representing the generic `StateSet` type relies on Rust's traits to encode the variants as a tail-recursive list of nested tuples. Matching the elements of the lists is achieved by repeatedly attempting to match the head. Listing 19 shows one possible representation which uses a final sentinel list element to make for an ergonomic definition of the `StateSetMatch` trait, which provides head-matching behavior for singleton lists, distinct from that of larger ones. From the user's perspective, `StateSet` objects are opaque, and prevent the automaton from proceeding with transitions until the object is collapsed to some usable `State` object.

```
1  trait StateSetMatch {
2      type MatchResult;
3      fn match_head(self) -> Self::MatchResult;
4  }
5  struct StateSet<L> { data: /* omitted */ }
6  impl<H,Ta,Tb> StateSetMatch for StateSet<(State<H>, (Ta, Tb))> {
7  // a list with 2+ elements. match_head definition omitted
8      type MatchResult = Result(State<H>, StateSet<(Ta,Tb)>>;
9  }
10 impl<S> StateSetMatch for StateSet<(State<S>, ())> {
11 // singleton set. `()` acts as a sentinel element. match_head definition omitted
12     type MatchResult = State<S>;
13 }
14 fn example(ab_set: StateSet<(State<A>, (State<B>, ()))>) {
15     match ab_set.match_head() {
16         Ok(a) => /* matched A */,
17         Err(b_set) => {
18             let b = b_set.match_head();
19             /* matched B */
20 }   }   }
```

Listing 19: Definition of type `StateSet`, which acts as an anonymous sum type by encoding its variants as a tail-recursive tuple in its generic argument. Two non-overlapping definitions of trait `StateSetMatch` are provided to make the type behave as expected in response to associated method `match_head`. Function `example` demonstrates how the arbitrary number of variants are matched two at a time by repeatedly attempting to match the first element of the list (the head), translating it into a conventional `Result` enum which Rust can pattern-match as usual. The result of this match can depend on the contents of field `data`, which is instantiated dynamically at runtime by interacting with the coordinator.

6.4.3 Governed Environment

So far, this section has described only the first of two facets of our design, as they appear in Section 6.3. Namely, we are able to model the protocol's configuration space, and trace its changes throughout the control flow of the user's component by continuously updating the type of the *token*, a small no-data structure used only for encoding information in its type. What remains is the step that connects this protocol configuration to its relationship with a given port operation.

We define the `Governed<Putter>` and `Governed<Getter>` types as wrappers for the `Putter` and `Getter` types existent in Reo-rs, corresponding to (logical) input and output ports respectively. As is idiomatic in the Rust language, our `Governed` type offers a type-safe means of 'managing' the API of the type it wraps (by enclosing the original API and providing its own), without altering the underlying data structure as it is represented at runtime. In other languages, this may be approximated with function overloading.

Everything comes together by defining the API of our wrapper types such that its port operations are specified to consume a particular token, and produce a replacement. For this to work as intended, the function's requirement for input and output token types must correspond with transitions in the type-state automaton. In other words, the API of our `Governed` type conflates two otherwise orthogonal actions, such that one cannot occur without the other: every port action a that changes the protocol's configuration from C_x to C_y (1) a occurs, (2) the token's type for C_x is updated to the type for C_y. The effect is the (static) prohibition of port operations which do not correspond to transitions through the protocol's configuration space, i.e., they do not occur 'next' for a protocol in the state matching the type of the token.

For cases there the token corresponds with a protocol configuration in which there are several possible next actions (described in Section 6.4.2), the user's component cannot safely commit to any choice, as it would necessarily constrain the component's behavior, violating its protocol adherence. Such cases are detected statically by having a `StateSet` with two or more elements. At runtime, this problem is solved by 'asking' the protocol which branch to take, after which we have a single state token and proceed as before. Statically, the choice is not yet known, and so the programmer must provide the definition of the resultant behavior for each case. Section 6.4.2 explains that our solution emulates matching by recursively defining behavior in response to suffixes of a list whose elements match those of the `StateSet`.

6.5 Making it Practical

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

6.5.1 Approximating the RBA

The approach to generating a type-state automaton from an RBA was given in 6.4.1. Our type-state automaton suffer the same state space explosion of Constraint Automata, prior to the inclusion of memory (explained in Section 2.1.3). We cannot hope to represent realistic programs with this approach alone, as the necessary automaton would be wildly unmanageable in its number of states and transitions. In this section, we explain how the type-state automaton is adapted to approximate the protocol's configuration space such that we strike a balance between accuracy and simplicity, without any effect on the governor's correctness.

Data Domain Collapse

We abandon the goal of faithfully representing the entirety of the protocol's configuration space in favor of representing an approximation by assuming all data types to be the trivial unit type. With this assumption, memory cells may be in one of two states: (a) empty, (b) filled with 'unit'. Converting existing RBAs may see large subexpressions of data constraints becoming constant, including checks for equality and inequality between port values. In practice, the vast majority of Reo protocols do not reason about the contents of memory cells beyond distinguishing fullness from emptiness. For these protocols, approximation has no effect. Note that this is the same rationale behind the optimization explained in Section 4.3.4. In this context, this means 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 configuration tuple is indeed its own type, but neither the code generator nor the compiler need

to pay the price of enumerating all the combinations eagerly. For example, a configuration of three empty memory cells would be represented by type `(Empty,Empty,Empty)`. For brevity, we will henceforth shorten `Empty` and `Full` to `E` and `F` respectively, and abbreviate tuples by omitting commas wherever doing so will not result in ambiguity.

As before, we are able to represent an RBA rule as a function in the Rust language by encoding a configuration change from q to p determines its declaration such that it consumes the type-state of p and returns the type-state of q. The naïve approach of generating functions per type-state is susceptible to the same exponential explosion that plagued CAs in the first place. Fortunately, tuple types have inherent structure which Rust's generic type constraints are able to understand. The use of generics to ignore elements of the tuple coincides with an RBA's ability to 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; this facet is borrowed from the Reo compiler's internal representation from Chapter 3.

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 2.6. 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 `E` or `F`, 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 domain shared by `A` and `B`.

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 orientation 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 6.5.1 can be applied directly. Otherwise, the interface of a compute component does not subsume the entirety of the interface of its protocol. In such systems, the protocol interfaces with several compute components. Indeed such cases form the majority in practice; compute components tend to only play a small role in a larger system.

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

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

Listing 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 2.6. Function bodies are omitted for brevity. Note that `M` is not a type, but rather a generic type parameter to be instantiated at the call site.

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 6.1: RBF of the *fifo2* protocol, equivalent to the RBA in Figure 2.6. Formatted with an outermost disjunct per line such that guard and assignment parts per rule are discernible.

The contents of Section 6.5.1 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* ports not in the interface projected upon, omitting the actions of those ports from the specification.

As an example, we once again generate a governor for a compute component with interface $\{A\}$ with the `fifo2` protocol. This time the protocol is represented as an RBF in Table 6.1 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

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

Listing 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.

here, this approach to generating governors is correct, but has two undesirable properties:

1. **API clutter**
   The 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 reuse (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 6.5.1 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:

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

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

- P-*final* wrt. port set I:
  A rule sequence of RBA R, with last rule r_{last} is P-final with respect to port set I if $\mathrm{Act}(r_{last}) \cap I = \{P\}$ and for all rules r in the sequence, $r = r_{last} \vee \mathrm{Act}(r) \cap I = \varnothing$.

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

The intuition here is that it does not matter how the governor's RBA structures its rules. It is unnecessary for governors to advance in lockstep with the protocol to the extent that they agree on the protocol's configuration at all times. It suffices if the protocol and governor always agree on which actions the ports in their shared interface do next. Figure 6.1 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, i.e., 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 $n = *$.

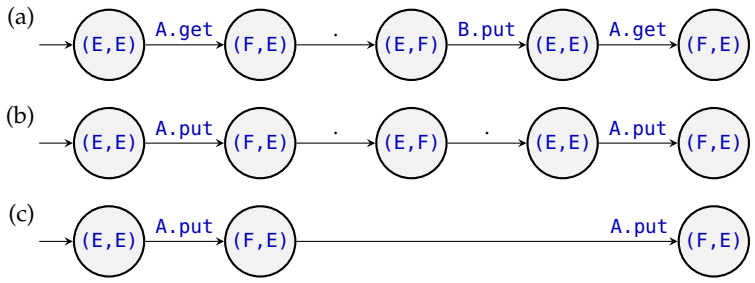


Figure 6.1: Rules being applied to walk three RBAs in lockstep, with time horizontally, showing the (simplified) configurations traversed, and annotating rules by showing which port actions they involve.
(a) RBA of protocol `fifo2`. (b) RBA of `fifo2` projected onto port set $\{A\}$.
(c) RBA of `fifo2` projected onto port set $\{A\}$ and normalized to remove silent rules.

The normalization algorithm is **correct**, as it does not have silent rules once it returns (`not_silent` containing zero silent rules is invariant). Observe that for each silent rule removed, it does not consider composing with itself. The immediate result is that the algorithm never inserts some rule $x \cdot x$ for silent rule x. This is not a problem, as all silent rules of our approximated RBAs are idempotent with respect to their impact on the configuration. The algorithm is able to take for granted that the result any chain of silent rules $x \cdot x \cdot x \cdot ...$ is covered by considering x itself. Furthermore, the incremental removal of rules prohibits the creation of any silent cycles at all. This is due to the reasoning above being extended to any sequences also[2]

The normalization algorithm is **terminating**. It consists of finitely-many algorithm steps in which the RBA A is replaced by RBA $B = (A \setminus \{r\}) \cup \{r \cdot x | r \in A \setminus \{x\} \wedge \mathtt{composable}(x, r)\}$ for some silent rule $x \in A$. Initially, A is the input RBA with silent rules. The algorithm terminates, returning B when A is replaced by B where B has no silent rules. Let $P(x)$ be the set of acyclic paths through RBA x's configuration space. Observe that initially, $P(A)$ is finite. It suffices to show that in each algorithm round, $|P(A)|$ strictly decreases. Within a round, for every 'added' p in $P(B) \setminus P(A)$, p contains a rule $m \cdot n$ such that there exists p' in $P(A) \setminus P(B)$ identical to p but with a 2-long sequence of rules m, n in the place of x. From this we know that $|P(A)| \geqslant |P(B)|$. However, the 1-long path of x itself is clearly in $P(A) \setminus P(B)$. Thus, $|P(A)| > |P(B)|$. QED.

[2]The reader may note the similarity between this observation and that made by the pumping lemma for regular languages. [Lin06]. In both cases, we observe that an arbitrary sequence of 'idempotent' cycles as part of a walk through configuration space have no influence on the rest of the path.

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

Listing 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.

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 6.2: RBF of the `fifo2` protocol, projected onto port set $\{A, B\}$ and normalized. Rules 0 and 2 are retained from Table 6.1, and new rules $1 \cdot 0$ and $1 \cdot 2$ are composed of rules from the original RBF.

To demonstrate the normalization procedure, Table 6.2 shows the result of projecting the `fifo2` connector's RBF onto port set $\{A, B\}$ and normalizing. The two additional rules can be understood to recreate the behavior lost as a result of omitting the silent rule 1 from the original RBF.

6.5.2 User-Defined Protocol Simplification

Recall, the purpose of a governor is to preserve a system's liveness. They do this by ensuring that their governed compute component performs port operations that allow the interfacing protocol (and the system around it) to progress. Governors do this by enforcing that their compute component's implementation *covers* each possible transition by providing code that performs the re-

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

Table 6.3: RBF of the a7b1 connector, which is characterized by cycling through a predictable sequence of period 8, where A inputs seven times and B outputs once. It works by encoding its configuration in an 8-long cycle as a three bit integer using the fullness of memory cells m_{0-2}.

quired task, and ensuring it is chosen correctly in accordance with the wishes of the protocol. Section 6.4.2 explains how our type-state automaton represents this by each configuration requiring the definition of a set of transitions, one for each action.

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

This overzealousness becomes a problem when it infringes on the component's ability to express its behavior as intended. Consider the example of a component X that forwards values from its input port A to its output port B. Perhaps this component is used in a pipeline as intended such that the component is involved in an endlessly alternating sequence, represented by regular expression $(AB)^*$. Perhaps there is a sensible way for X to implement the more permissive protocol which allows B firings to be omitted, expressed $(A(B|\lambda))^*$. P has no problem discarding values input from A. However, if the governor takes it a step further such that 'anything goes' (expressed $(A|B)^*$), X cannot meaningfully represent its work. How on earth can it forward a message to B before receiving it from A? Not even clairvoyance can help; what if A never fires at all? This is how the user would experience the problem of a governor infringing on the component's own behavior. P has permissible constraints of its own which the governor requires be relaxed.

Nevertheless, there is value in providing a compute component with a simplified (permissive) view of the protocol where possible. As a motivating example, consider the a7b1 connector, given as RBF in Table 6.3. This connector uses the fullness of three memory cells to count in binary from zero to seven (using the binary alphabet of memory cell states {E, F}), and cycling back again

to zero. Configurations in this cycle are distinguished by specifying different behaviors on A and B. Here, the projection and normalization of the protocol's RBF is trivial, as no rules are silent. Without the ability to simplify, the Y must be implemented such that it corresponds exactly with the protocol's (predictable) walk through its approximated configuration space, given in Figure 6.2. As all states are distinguishable, so too are their corresponding state types distinct. Now consider this protocol interfacing with some compute component Y, which is always ready to consume and emit some data element Q. Without simplification, the resulting governor would require that the traversal through configuration space be spelled out; the user would be forced to distinguish these states, even though Y has no need for this specificity. Most likely, the resulting implementation will be repetitive and verbose, defining the same behavior for cases of configuration `(EEE)`, `(EEF)`, et cetera.

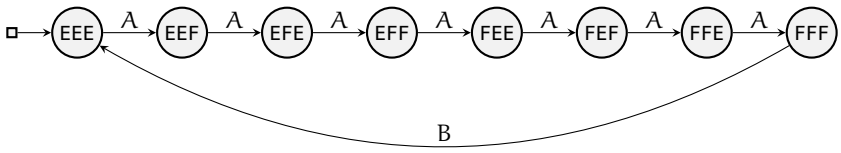


Figure 6.2: Rules transitioning through configuration space of approximated RBAs for the a7b1 connector, with states named after the 'count' the three memory cells represent in base 2 (in binary alphabet {`E`, `F`}). Here, the normalization procedure with interface set {A, B} is trivial as no transitions are silent.

Our solution to this problem is to introduce a third type for representing the state of a memory cell which may be either full or empty: `Unknown` (abbreviated as `U`). Rather than corresponding to a specific configuration of the (approximated) RBA, the governor now reasons about the set of states which the protocol may be in. For example, type `(UUE)` encapsulates all the concrete configuration types {`(EEE)`, `(EFE)`, `(FEE)`, `(FFE)`}, and is liable to covering the union of the rules applicable to any of those states. In this manner, it is safe for the programmer to arbitrarily 'forget' the state of a memory cell, replacing its element in the tuple type with `U`. To be clear, `U` is not special as far as Rust is concerned; we have changed from a binary to a ternary alphabet for representing memory cells in types. However, `U` does not correspond to any real configuration that memory cells are ever 'really' in at runtime; they are always either empty or full. `U` is a stand-in for an empty or full memory variable; this is an abstraction in which the protocol is not (explicitly) involved. With this tool in their belt, the implementation of the compute component is able to arbitrarily unify the state types of multiple branches. Our example

component Y above is able to implement its behavior to the satisfaction of its governor with transitions through configuration space in Figure 6.3. This weakening can be communicated quite ergonomically, resulting in something very close to what the user would implement themselves: a single loop where the four rules (numbered 0-3) may be applied to configuration type `(UUU)`, each resulting again in `(UUU)`.

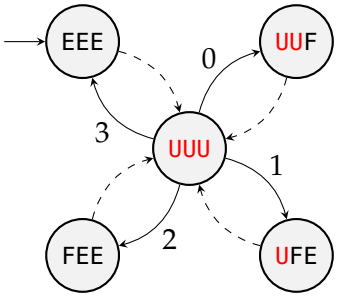


Figure 6.3: Rules transitioning between configurations of the a7b1 connector shown in Figure 6.2. Here, the user employs weakening to convert (dashed arrows) tokens to those of `(UUU)`, representing all concrete configurations. RBA rules firing are shown with solid arrows, annotated with the rule name, corresponding to those given in Table 6.3.

6.5.3 Match Syntax Sugar

Section 6.4.2 explains how the set of transitions to be covered by a configuration type can be represented in Rust's type system as a tail-recursive list. This alleviates the problem of having to explicitly enumerate the needed sets each as their own enumeration type. This is necessary, as the upper bound[3] of state sets is 2^{3^M}, where M is the number of memory cells;[4] suffice it to say, it is a large number. Unfortunately, these are not natively-supported enumeration types, and thus cannot be matched as is idiomatic in the Rust language. However, Rust has extensive support for abstract syntax tree macros, allowing us to have the best of both worlds; the user interacts with `StateSet` types by using a match-like macro which enumerates the branches, but there is no need for concrete `enum` classes to be defined for all the conceivable combinations. Figure 23 gives an example of how these cases compare to one another.

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

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

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

Listing 23: Example of three methods for matching a state set, representing a sum type of three variant types simplified here to `X`, `Y` and `Z`. First, `match_standard` shows how this is done in idiomatic Rust, requiring an enum type `StateSetXyz` be explicitly defined. `match_recursive` shows how the same state set represented by a tail-recursive `StateSet` type can be similarly matched by exhaustively 'unzipping' head elements using a function `match_head`. finally, `match_macro` functions identically to the second case, but relies on a sugaring macro `match_set` to mimic the syntax of Rust's `match` statement, seen in the first function.

Chapter 7

Discussion

In this chapter, we reflect on the work and findings in Chapters 3–5. This includes a subjective assessment of the results of the project as a whole, and identification of promising directions for future work.

7.1 Future Work

As with any project, there was insufficient time to investigate every topic encountered exhaustively. In this section, we highlight promising starting points for future work related to Reo, or to our contributions in Chapters 3–6.

7.1.1 Imperative Form Compiler

Chapters 3 and 4 explain how the Reo-rs runtime makes use of a lightweight interpreter to bring life to our protocol objects at runtime according to the appropriate specification. This commandification pattern has its advantages; namely, protocol behavior is alterable at runtime by manipulating the interpreted data. However, this flexibility does not come for free. The interpretation steps incur overhead both to the protocol construction procedure, and more importantly, to the work of port operations. Fortunately, our imperative form does not necessitate the use of an interpreter. Future work could investigate replacing the `build` procedure of Reo-rs with another compilation step such that the behavior is represented in native, directly executable Rust.

Future work might investigate the use of custom domain specific languages for compiling imperative form in a manner that it performs the same checking as in `build` statically. the obvious means of doing this is to build a compiler from scratch. However, other options exist that can make better use of existing tools. For example, Rust's *procedural macros* allow the programmer to define

arbitrary transformations of Rust's abstract syntax trees during compilation. Essentially, one is able to invoke arbitrary, precompiled Rust code inside the user's Rust compiler itself. In this manner, one can embed the needed domain-specific language into the Rust compiler.

7.1.2 Distributed Components

This work focuses on coordination between threads in shared memory. This approach can already be applied in the context of distributed components by abstracting ports behind local ones. However, we are unable to distribute the internals of our protocol components, as they presuppose a single, monolithic shared state in our current scheme. One can get around this by fragmenting protocols into smaller ones, and distributing those smaller protocols across the system. However, this cannot currently be done in all cases, as this fragmentation does not preserve synchrony.

Reo has a rich academic history in this distributed context. Examples include the works of Proencca et al. [PCDVA12], Koehler et al. [KAdV08], and Jongmans et al. [JSA14]. Future work might investigate how our contributions (e.g., reference-passing optimizations, static governors, etc.) can be applied in distributed systems.

7.1.3 Optimize Rule Branching

Reo-rs is able to represent protocols whose rules contain branching (i.e., logical disjunction). Rule-based form has already shown us the correspondence between our RBA rules and propositional logic, where the formula corresponds to a protocol, with disjuncts as rules [DA18a]. In the same way such terms can be manipulated until the formula is in disjunctive normal form, so too are we able to remove branching from our rules by splitting them. For example, a rule with data constraint $(P_0 = C_0 \vee P_1 = C_1) \wedge P_2 = C_2$ can be converted into two rules with data constraints $P_0 = C_0 \wedge P_2 = C_2$ and $P_0 = C_0 \wedge P_2 = C_2$ respectively. Such transformations are not particularly meaningful to the outside observer; clearly they have no influence on the protocol's semantics.[1] However, they do have interesting implications on performance. It is easy to contrive of examples for both extreme ends of the spectrum for which this splitting is either beneficial or detrimental to the performance of the protocol object, as we are able to both introduce and eliminate redundant work by splitting rules. As an example of a rule not worth splitting, consider one with very many instructions I before reaching a 3-way branch $a \vee b \vee c$ where the branch is not

[1] Changing the granularity of rules can be semantically meaningful once it affects our ability to express interesting properties. For example, fine-grained rules can be desirable when the protocol is lifted to consider *preference* between nondeterministic branching.

entirely nondeterministic (i.e., there are cases for which a cannot be chosen, etc.). Before splitting, I is computed once, and then one of $\{a, b, c\}$ will occur. After splitting, this is represented by three separate rules for a, b, and c. In the worst case, each is evaluated I before the third fires successfully. By splitting, we have introduced redundant computations of I.

In our case, only the Reo compiler's internals perform manipulations on protocol rules, while Reo-rs restricts itself to the treatment of tautologies and contradictions. Future work might investigate more extensive manipulation of protocol rules to optimize rules by conditionally splitting them to remove branching.

7.1.4 Runtime Governors

Chapter 6 explains how our design for static protocol governors is able to enforce protocol adherence at compile time. This approach is not always suitable, as it presupposes that we trust the compilation process enforcing the governance. If the situation calls for a degree of separation between the compilation process and its use at runtime, this may no longer be a good choice. For example, consider the use of Reo to coordinate peers in a distributed system, where the behavior of a component originates from a remote source, traveling over the network.

Our static governors also have limitations on how finely they can distinguish the states of the protocol. In a perfect world, governors would use perfect models of the protocol's state. Section 6.5.1 gives an example of some practical reasons to approximate the protocol instead, resulting in a governor that attempts to strike a non-trivial balance between accuracy and simplicity of its local automaton. Dynamic governors are given a far easier task, as they are able to make choices at runtime, when all the relevant information is available. Generally, these governors can therefore be more accurate. Future work might investigate how these two extremes of the spectrum compare, and to what extent it may be practical to use some facets of **both** to make an application more robust. There may be systems in which redundant checking is worth its cost to runtime performance.

7.1.5 Further Runtime Optimization

Section 4.3.4 discusses various optimizations of Reo-rs's runtime performance applied in this work, chief of which is arguably the use of reference-passing inside the protocol's state as part of the implementation of rules such that it preserves Reo's value-passing semantics. Other optimization opportunities presented themselves during the project, but were not investigated thoroughly

as they conflicted with our current goals, or were simply deemed less fruitful than other tasks. Future work might investigate these optimizations:

1. Simplify rules in the context of a known priority ordering to break nondeterminism. For example, consider rule r_a with priority over rule r_b. Clearly, r_a must always be given a chance to fire first, allowing r_b to presumes the negation of r_a's guard. In practice, this can often allow rules to be meaningfully simplified, particularly when they are created by splitting nondeterministic branches as discussed in Section 7.1.3. For example, consider rules with the data constraints $M_0 = M_1$ and $M_0 \neq M_1$. If they are prioritized in the order of their appearance here, the guard of the second rule becomes trivially $\mathtt{true}$.

2. Remove indirection inside Reo-rs when representing values smaller than the pointer-size, by 'stuffing' the value inside the pointer field. This optimization has complex interactions with the memory storage system described in Section 4.3.4, which uses pointers as keys to look up a value's reference count. Future work may investigate either (1) conditionally using stuffed pointers when it would not interact with the memory storage system, or (2) finding a way to make the memory storage system disambiguate these stuffed pointers

3. More extensively preprocess the imperative form as its executable object is built (explained in Section 3.3.3). For example, instructions can be fragmented and reorganized such that the effects of a fired rule are unaltered, but rules are able to detect and recover from unsatisfied guards by rolling back earlier. In this way, one can minimize the cost of evaluating rules with unsatisfied guards.

4. Reduce the number of atomic operations used for the exchange of metainformation during data exchange (Section 4.3.4). Assuming realistic numbers of ports we are able to collapse several atomic operations into one, aggregating distinct operations by using modulo arithmetic. For example, we are able to increment two (logical) numbers using a single atomic counter by adding $1 + 2^{32}$. In this fashion, the *move* and *countdown* variables may be unified to reduce lock contention and further increase performance and parallelism.

5. Some of the information currently exchanged between threads using atomics can be independently derived by reading the protocol's rules. For example, getters can deduce their putter and whether they are permitted to move the datum this way, rather than being told by the coordinator explicitly. It is unclear whether this is an improvement, as these threads

must spend extra time recovering this information, performing work redundant to that of the coordinator.

7.1.6 Avoid Lock Re-Entry

Section 4.3.2 motivates the lack of a dedicated coordinator thread to back protocol objects at runtime. As a consequence, port threads must share the responsibility of manipulating a shared protocol state in accordance with the protocol's movement through its configuration space. Protocols have non-trivial configuration spaces once they involve one or more memory cells in rules. To manipulate their contents safely, locks are required around the shared 'bookkeeping' structures that track these cells' states. Section 4.3.4 explains how our design optimizes for the concurrency of rule firings by moving the work of interacting with memory cells outside of the critical reason. A consequence of this approach is the occasional need for threads to reenter the critical region to update the state of a memory cell. For example, the first lock event instigates the rule firing and updates state, but marks memory cell M as 'busy' to ensure it cannot be involved in a rule firing until all data movements outside the critical region have completed. Once done, some thread has the responsibility to mark M as ready once again, necessitating a second lock event.

Future work might investigate more efficient mechanisms for achieving the same effect. As the mechanism is rather intricate, a vast space of possibilities exist. For example, one might investigate the effect of forgoing the second lock event in favor of leaving a message for a later coordinator to handle. This could take the form of an efficient parallel queue, highly optimized for the addition of new elements in parallel. More radical changes may also result in superior performance. Perhaps the locking can be avoided entirely if all the data structures representing the protocol's state become lock-free?

7.1.7 Runtime Reconfiguration

Chapter 4.3.4 explains how Reo-rs uses a lightweight interpreter to implement protocol behavior at runtime, reading rules from a dense data structure. A result of this approach is the ability to alter a protocol object's behavior at runtime arbitrarily by manipulating the data representing its rules. Future work might investigate the introduction of a reconfiguration procedure to change the protocol without tearing the instance down or influencing the compute components in motion. The use of an interpreter trivializes the work of manipulating the rules themselves, but care must be taken to change the protocol object's meta state safely such that it results in a new protocol which is again internally consistent (e.g., reconfiguring the structures used for primitive concurrency, message channels etc.).

7.2 Conclusion

The chosen design and implementation of a Rust code generator for the Reo compiler achieved satisfactory results. Although our protocol objects were usually slower than handwritten Rust programs, they were competitively performant in the case of non-trivial protocols. This is despite their data-oriented implementation, which has the added benefit of facilitating the reconfiguration of a protocol's behavior at runtime. Exploiting this feature was out of the scope of this project, but provides an entrypoint for interesting future work.

By the nature of Reo, correctness of a protocol object's behavior was always paramount. Still, this work also emphasized performance from its inception, motivating the choice of the Rust language in particular. The hope was to leverage its static ownership system to implement a powerful reference-passing optimization employed by the Java backend, but free from the associated safety problems. However, the Rust compiler was stumped by Reo's interactions transferring values between threads and between scopes. Ultimately, our solution followed the Rust idiom of manually managing ownership within a minimal unsafe scope, and wrapping it in an API that was safe once again. Here, Rust's ownership and mutual-access semantics were invaluable. This was the case for user-facing functionality in general, including that of creating and destroying protocols and ports. In the end, we were able to provide an API with strong, static safety guarantees provided the user does not intentionally circumvent Rust's semantics with unsafe code; a user can neither create nor encounter malformed protocol or port objects, and they cannot experience system behavior that contradicts the specification of the protocol (unintentionally or otherwise). Furthermore, our design includes other novel optimizations that result in benefits to the user. For example, protocols function without dedicated threads, increasing performance for sequentially accessed protocols, and trivializing the detection of a protocol's termination.

Rust's affine type system was instrumental in our design of static governors, which allow a programmer to verify that their components do not threaten the liveness of the system at large. This is done almost entirely at compile time, catching errors as early as possible, and minimizing runtime overhead. Our design demonstrates how an affine type system is able to communicate powerful correctness guarantees across an API boundary. In our case, we are able to embed a complex requirement 'the component does not perform a port action that contradicts the specification of a stateful protocol' into terms the compiler can understand and enforce: the program type checks. Users are provided with a means to opt into tasking their Rust compiler with performing this check, effectively allowing them to extend its verification capabilities with little more than an added library dependency.

The complexity of the translation procedure from Reo to Rust proved to be more complex than was expected. The imperative form intermediate representation was added out of necessity to curb the complexity of type checking and of applying various optimizations without bloating the Reo compiler itself with Rust-specifics. Unexpectedly, the introduction of this new form became integral to our protocol object's design, enabling us to extend its capabilities beyond what was originally intended. When targeting Rust, the Reo compiler supports ergonomic and safe use of more exotic Reo primitives such as *filter* and *transform*, which are able to perform tentative computations as part of synchronous interactions. Imperative form also shows promise as an intermediary step for languages similar to Rust; it is conceivable that existing targets such as Java (or others not yet implemented) can leverage this representation to reduce the work of adding new imperative language targets to the Reo compiler.

Bibliography

[ABdBR07] Farhad Arbab, Christel Baier, Frank de Boer, and Jan Rutten. Models and temporal logical specifications for timed component connectors. *Software & Systems Modeling*, 6(1):59–82, 2007.

[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.

[Arb05] Farhad Arbab. Abstract behavior types: a foundation model for components and their composition. *Science of Computer Programming*, 55(1-3):3–52, 2005.

[Arb11] Farhad Arbab. Puff, the magic protocol. In *Formal Modeling: Actors, Open Systems, Biological Systems*, pages 169–206. Springer, 2011.

[BSAR06] Christel Baier, Marjan Sirjani, Farhad Arbab, and Jan Rutten. Modeling component connectors in reo by constraint automata. *Science of computer programming*, 61(2):75–113, 2006.

[CCZ07] Bradford L Chamberlain, David Callahan, and Hans P Zima. Parallel programmability and the chapel language. *The International Journal of High Performance Computing Applications*, 21(3):291–312, 2007.

[DA18a] Kasper Dokter and Farhad Arbab. Rule-based form for stream constraints. In *International Conference on Coordination Languages and Models*, pages 142–161. Springer, 2018.

[DA18b] Kasper Dokter and Farhad Arbab. Treo: Textual syntax for reo connectors. *arXiv preprint arXiv:1806.09852*, 2018.

[GC92] David Gelernter and Nicholas Carriero. Coordination languages and their significance. *Communications of the ACM*, 35(2):96–108, 1992.

[Gel85] David Gelernter. Generative communication in linda. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(1):80–112, 1985.

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

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

[JSA14] Sung-Shik TQ Jongmans, Francesco Santini, and Farhad Arbab. Partially-distributed coordination with reo. In *2014 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 697–706. IEEE, 2014.

[JSA15] Sung-Shik TQ Jongmans, Francesco Santini, and Farhad Arbab. Partially distributed coordination with reo and constraint automata. *Service Oriented Computing and Applications*, 9(3-4):311–339, 2015.

[JSS+12] Sung-Shik TQ Jongmans, Francesco Santini, Mahdi Sargolzaei, Farhad Arbab, and Hamideh Afsarmanesh. Automatic code generation for the orchestration of web services with reo. In *European Conference on Service-Oriented and Cloud Computing*, pages 1–16. Springer, 2012.

[KAdV08] Christian Koehler, Farhad Arbab, and Erik de Vink. Reconfiguring distributed reo connectors. In *International Workshop on Algebraic Development Techniques*, pages 221–235. Springer, 2008.

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

[Lin06] Peter Linz. *An introduction to formal languages and automata*. Jones & Bartlett Learning, 2006.

[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.

[PCDVA12] José Proença, Dave Clarke, Erik De Vink, and Farhad Arbab. Dreams: a framework for distributed synchronous coordination. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, pages 1510–1515. ACM, 2012.

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

[Sha84] Mary Shaw. Abstraction techniques in modern programming languages. *IEEE software*, (4):10–26, 1984.

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

[Wel05] George Wells. Coordination languages: Back to the future with linda. In *Proceedings of the Second International Workshop on Coordination and Adaption Techniques for Software Entities (WCAT05)*, pages 87–98, 2005.

[ZHLS19] Xiyue Zhang, Weijiang Hong, Yi Li, and Meng Sun. Reasoning about connectors using coq and z3. *Science of Computer Programming*, 170:27–44, 2019.

Appendices

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

Listing 24: 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.

```rust
// initialization
let barrier_g  = Arc::new(std::sync::Barrier::new(3));
let barrier_p0 = barrier_g.clone();
let barrier_p1 = barrier_g.clone();
let (data_0_s, data_0_r) = crossbeam_channel::bounded(0); // synch  (unbuffered)
let (data_1_s, data_1_r) = crossbeam_channel::bounded(1); // asynch (1-buffered)

// port operation functions
let p0_put_function = || {
  barrier_p0.wait();
  data_0_s.send(P0_VALUE).unwrap();
};
let p1_put_function = || {
  barrier_p1.wait();
  data_1_s.send(P1_VALUE).unwrap();
};
let g_get_function = || {
  barrier_g.wait();
  let value_from_p0 = data_0_r.recv().unwrap();
  let value_from_p1 = data_1_r.recv().unwrap();
};
```

Listing 25: Handcrafted alternator implementation ini Rust based on channels from the `crossbeam` crate and a standard library `Barrier` for explicit synchronization. This simple design is chosen for its simplicity and its close correspondence to the Reo channels that constitute its specification.

```
        mov     eax, 131096
        call    __rust_probestack
        sub     rsp, rax
        lea     rdi, [rsp + 9]
        mov     edx, 65536
        mov     esi, 2
        call    qword ptr [rip + memset@GOTPCREL]
        mov     byte ptr [rsp + 8], 1
        mov     byte ptr [rsp + 65552], 0
        mov     eax, 32
.LBB0_1:
        movups  xmm0, xmmword ptr [rsp + rax - 24]
        movups  xmm1, xmmword ptr [rsp + rax - 8]
        movups  xmm2, xmmword ptr [rsp + rax + 8]
        movups  xmm3, xmmword ptr [rsp + rax + 24]
        movups  xmm4, xmmword ptr [rsp + rax + 65520]
        movups  xmm5, xmmword ptr [rsp + rax + 65536]
        movups  xmm6, xmmword ptr [rsp + rax + 65552]
        movups  xmm7, xmmword ptr [rsp + rax + 65568]
        movups  xmmword ptr [rsp + rax - 24], xmm4
        movups  xmmword ptr [rsp + rax - 8], xmm5
        movups  xmmword ptr [rsp + rax + 65520], xmm0
        movups  xmmword ptr [rsp + rax + 65536], xmm1
        movups  xmmword ptr [rsp + rax + 24], xmm7
        movups  xmmword ptr [rsp + rax + 8], xmm6
        movups  xmmword ptr [rsp + rax + 65552], xmm2
        movups  xmmword ptr [rsp + rax + 65568], xmm3
        add     rax, 64
        cmp     rax, 65538
        jb      .LBB0_1
        add     rsp, 131096
        ret
```

Listing 26: Snippet out of the x86-64 assembly generated by receiving a large datum through `recv` from a simple channel from the Rust standard library. It unrolls the movement of the entire object into a large sequence of smaller operations rather than invoking a system call. This is the case for the receipt of a `Copy`-type represented by 512 bytes.

```rust
#![feature(raw)]
use std::mem::{MaybeUninit, transmute};
use std::raw::TraitObject;
pub struct TypeInfo(usize);
impl TypeInfo {
    pub fn of<T>() -> TypeInfo {
        let x: Box<T> = unsafe { MaybeUninit::uninit().assume_init() };
        let fat_x = x as Box<dyn PortDatum>;
        let raw: TraitObject = unsafe { transmute(fat_x) };
        TypeInfo(raw.vtable as usize)
}   }
trait PortDatum {
    fn foo(&self) -> u32 { 123 }
}
impl<T> PortDatum for T {}
pub fn test() -> TypeInfo {
    TypeInfo::of::<u32>()
}
```

Listing 27: Example of how the `TypeInfo::of` function (1) ensures the compiled binary includes a vtable for the requested type `T` with `PortDatum` as its interface, and (2) returns the pointer to the vtable.

```
core::ptr::real_drop_in_place:
        ret
example::PortDatum::foo:
        mov     eax, 123
        ret
example::test:
        lea     rax, [rip + .L__unnamed_1]
        ret
.L__unnamed_1:
        .quad   core::ptr::real_drop_in_place
        .quad   4
        .quad   4
        .quad   example::PortDatum::foo
```

Listing 28: x86-64 assembly of Listing 27. `.L__unnamed_1` shows the binary representation of the vtable of the `u32` (32-bit unsigned integer) type and `PortDatum` trait. Rust's vtables have a predictable structure with three fields followed by trait-defined function pointers. Lines 10–12 store the concrete type's (1) `drop` function pointer, (2) size in bytes, (3) memory alignment in bytes.