TECHNISCHE UNIVERSITÄT DRESDEN

FACULTY OF COMPUTER SCIENCE
INSTITUTE OF SOFTWARE ENGINEERING
CHAIR FOR COMPILER CONSTRUCTION
PROF. DR. JERONIMO CASTRILLON

Diplomarbeit

zur Erlangung des akademischen Grades
Diplom-Informatiker

A Rust Backend for Lingua Franca

Clément Fournier
(Born 23 January 1998 in Le Havre, France, Mat.-No.: 4754263)

Tutor: Dipl. Ing. Christian Menard

Dresden, December 6, 2021

TECHNISCHE UNIVERSITÄT DRESDEN

cfaed CENTER FOR ADVANCING ELECTRONICS DRESDEN

Fakultät Informatik Institut für Technische Informatik, Professur Compilerbau

Task Description for Diplomarbeit

for:	**Clément Fournier**
Major:	Diplom Informatik, Year 2018
Matriculation Nr.:	4754263
Title:	**A Rust Backend for Lingua Franca**

Lingua Franca is a polyglot coordination language implementing the programming model of "Reactors". This model based on discrete events with timed semantics can be used to design cyber-physical systems in a deterministic fashion. Thereby, the polyglot approach of Lingua Franca allows the model to be used in multiple "back-ends", supporting different target languages and, by extension, platforms.
A language for which a Lingua Franca back-end does not exist yet is Rust. Rust is a systems-oriented language with a unique paradigm, namely that of its ownership model. Ownership in Rust forces the programmer to be explicit about memory pointers and references, which in turn enables a multitude of advantages. For example, Rust can automate garbage collection at compile-time or eliminate whole families of errors like many kinds of race conditions. In particular, a Rust compiler can reason precisely about the execution steps, allowing, in principle, execution that is predictable.
Predictability and precise control over the execution are both, desirable for cyber-physical systems, and goals of the Reactor model. Thus, Rust should be a great match for a Lingua Franca back-end. Although the unique properties of Rust's ownership model and synchronization routines impose challenges for implementing an efficient Rust runtime for Reactors, an previous work with an initial attempt at creating such a runtime suggests that these challenges can be overcome. Building on the existing initial runtime, the goal of this thesis is to create a complete Rust backend for Lingua Franca including a code generator and an optimized Rust runtime.
In particular, the student should understand and explain the semantics of Lingua Franca, the precise requirements of an efficient implementation of Reactors, as well as the benefits and limitations of Rust's ownership model and synchronization mechanisms. Understanding these, the student should implement a Rust code generation backend for Lingua Franca, supporting all stable language features, and extend the existing runtime implementation as needed. An important extension of the existing runtime would be support for parallel execution of reactions. In order to understand performance characteristics of the newly created Rust backend, the student should also implement selected benchmarks and compare the performance of Rust to existing Lingua Franca backends. Further optimizing the Rust runtime based on the benchmarks can be an optional extension of the base task. Similarly, exploring the comparability of the new Rust backend with ongoing work in facilitating federated execution of Lingua Franca programs can be an optional extension to this work.

Advisor:	Christian Menard, Felix Wittwer
1. Examiner:	Prof. Dr.-Ing. Jeronimo Castrillon
2. Examiner:	Dr.-Ing. Gerald Hempel
Issued:	14. June, 2021

Jeronimo Castrillon Mazo Digitally signed by Jeronimo Castrillon Mazo Date: 2021.06.01 14:51:37 +02'00'

Prof. Dr. rer. nat. Christel Baier
Vorsitzender des Prüfungsausschusses

Prof. Dr.-Ing. Jeronimo Castrillon
Verantwortlicher Hochschullehrer

Declaration of authorship

I hereby declare that I wrote this thesis on the subject

A Rust Backend for Lingua Franca

independently. I did not use any other aids, sources, figures or resources than those stated in the references. I clearly marked all passages that were taken from other sources and cited them correctly.

Furthermore I declare that – to my best knowledge – this work or parts of it have never before been submitted by me or somebody else at this or any other university.

Dresden, December 6, 2021

[illegible]

Clément Fournier

Abstract

The *reactor model* is a deterministic model of concurrent computation, designed to match the safety and performance requirements of safety-critical real-time systems. The model underpins the semantics of *Lingua Franca* (LF), a coordination language that allows specifying reactor programs at a high-level of abstraction, while allowing the programmer to write the business logic of the program in any of its supported *target languages*, which include C and C++.

This thesis aims to add support for *Rust* as a target language for LF. Using an ownership model, Rust's powerful type system allows automating memory management without runtime garbage collection. Rust programs enjoy strong guarantees about memory safety and thread safety, which makes Rust an interesting target language for LF. Rust's ownership model nevertheless places strong restrictions on the usage of pointers, which make designing some programs challenging.

Building on previous work [16], this thesis presents the implementation of the LF Rust target. It explains how Rust's strong typing discipline can be used to enforce invariants of the reactor model at compile-time, and how it influences the design of the Rust runtime library, and of the code generated from LF. Thanks to sensible optimization decisions, performance evaluation of the Rust target shows uplifting results, outperforming the existing C++ target on a variety of benchmarks.

Kurzfassung

Das *Reaktormodell* ist ein deterministisches Modell für nebenläufige Berechnungen, das entwickelt wurde, um die Sicherheits- und Leistungsanforderungen sicherheitskritischer Echtzeitsysteme zu erfüllen. Das Modell bildet die Grundlage für die Semantik von *Lingua Franca* (LF), einer Koordinierungssprache, die es ermöglicht, Reaktorprogramme auf einer hohen Abstraktionsebene zu spezifizieren, während der Programmierer die Geschäftslogik des Programms in einer der unterstützten *Zielsprachen* (Targets), zu denen C und C++ gehören, schreiben kann.

Ziel dieser Arbeit ist, *Rust* als Zielsprache für LF zu unterstützen. Das starke Typsystem von Rust verwendet ein Ownership-Modell um die Speicherverwaltung ohne Garbage Collection zu automatisieren. Rust-Programme haben starke Garantien für Speichersicherheit und Threadsicherheit, was Rust zu einer interessanten Zielsprache für LF macht. Das Eigentumsmodell von Rust schränkt jedoch die Verwendung von Zeigern stark ein, was die Implementierung einiger Programmiermuster zu einer Herausforderung macht.

Aufbauend auf früheren Arbeiten [16] stellt diese Diplomarbeit die Implementierung des LF Rust-Targets vor. Sie erklärt, wie die starke Typisierungsdisziplin von Rust genutzt werden kann, um Invarianten des Reaktor-Modells zur Kompilierzeit durchzusetzen, und wie sie das Design der Rust-Laufzeitbibliothek und des von LF generierten Codes beeinflusst. Dank vernünftiger Optimierungsentscheidungen zeigt die Leistungsbewertung des Rust-Targets erbauliche Ergebnisse, und übertrifft das bestehende C++-Target in einer Vielzahl von Benchmarks.

Résumé

Le modèle des réacteurs est un modèle de calcul concurrent déterministe, conçu pour répondre aux exigences de sûreté et de performance de systèmes temps réel modernes. Le modèle fonde la sémantique de Lingua Franca (LF), un langage de coordination qui permet de spécifier un programme en termes de réacteurs, tout en permettant au programmeur d'écrire le cœur de la logique du programme dans n'importe lequel des langages cibles (*targets*) supportés, par exemple C ou C++.

L'objectif de cette thèse est de supporter Rust comme langage cible pour LF. Le système de types de Rust utilise un modèle de propriété (*ownership model*) pour automatiser la gestion mémoire sans nécessiter de récupérateur mémoire à l'exécution. Les programmes Rust jouissent de solides garanties en matière de sûreté, le système de types éliminant par exemple par construction les situations de compétition (*data races*). Cela fait de Rust un langage cible intéressant pour LF ; néanmoins, le modèle de propriété de Rust impose de fortes restrictions sur l'utilisation des pointeurs, ce qui rend la conception de certains programmes difficile.

S'appuyant sur des travaux antérieurs [16], cette thèse présente l'implémentation de la *target* Rust. Elle explique comment le typage fort de Rust peut être utilisé pour vérifier les invariants du modèle des réacteurs à la compilation, et comment il influence la conception de la logique d'exécution, et du code Rust généré à partir de LF. Grâce à des décisions d'optimisation judicieuses, la *target* Rust présente des résultats encourageants lors des tests de performance, surpassant la *target* C++ existante sur une sélection de benchmarks.

Contents

List of Figures

List of Tables

List of Listings

1. Introduction

The *reactor model* is a deterministic model of concurrent computation, designed to match the safety and performance requirements of modern cyber-physical systems. The model underpins *Lingua Franca* (LF), a coordination language that allows programmers to write reactor programs in any of its supported *target languages*: C, C++, Python, or TypeScript. While the high-level structure of the reactor program is expressed in LF, the business logic of the program is written in the target language, a general-purpose programming language. Before execution, the LF program is compiled to translate the LF structure into a program consisting entirely of target code. This target program can be interpreted or compiled into a binary, depending on the target language.

The goal of this thesis is to add support for *Rust* as a target language for LF. Rust is a systems programming language, whose type system leverages an *ownership model* to prove safety properties of the program at compile-time, like memory safety and data race freedom. This paradigm also allows Rust to perform automatic memory management without runtime garbage collection, which allows Rust programs to run on memory-constrained systems. Rust's support for efficient, safe low-level programming makes it an interesting target language for LF, whose paradigm is geared to benefit primarily safety-critical systems, including embedded cyber-physical systems.

However, Rust's type system can be an obstacle to writing some programs, e.g. because it places restrictions on the sharing of references. While previous work [16] shows that Rust's type system can accommodate the port-based communication of the reactor model, the runtime performance of that prototype runtime is not satisfactory. It also lacks support for parallel execution of reactions, a feature whose implementation may be further complicated by Rust's type-level verification of thread safety.

This work is focused on addressing these shortcomings, and delivering performant multi-threaded execution capabilities. The runtime implementation described in this thesis has been thoroughly optimized, and benchmarks indicate that it can perform competitively compared to other LF targets and a state of the art actor framework, Akka [6]. The runtime's application programming interface (API) uses Rust's powerful type system to guarantee some of the invariants of the reactor model at compile-time, thereby improving on the current LF targets. Where the implementation of the runtime has to subvert Rust's strong static verification with so-called *unsafe code*, it does so backed up by formal analysis of the reactor model's constraints.

This thesis is structured as follows. Chapter 2 presents the reactor model, Lingua Franca and related work. It also includes an introduction to the Rust language. Chapter 3 explains the design of the runtime, and its main differences with the runtime prototype and other LF targets. Chapter 4 presents the form of the code generated by the LF Rust code generator, and the compiler implementation. Chapter 5 evaluates the performance of the Rust runtime, and explains some of the main optimizations done to the runtime during its development. Chapter 6 presents possible avenues for future work. Finally, Chapter 7 concludes this thesis.

2. Background

Concurrent programming is widely regarded as a necessary means to address the performance and scalability requirements of modern computer systems [43]. Concurrent programs define actions that are to be performed simultaneously, instead of sequentially. These actions may be carried out by multiple processor cores, or, in distributed systems, by multiple machines running simultaneously. Many common abstractions to manage concurrency introduce nondeterminism in the program, i.e., situations where the behaviour of the program does not depend solely on the program's initial state and input. Such abstractions include threads [10], remote procedure calls [39], and actors [21, 1]. Nondeterminism makes concurrent programs hard to understand and debug, as concurrency bugs are poorly reproducible [30, 37]. For applications with high safety requirements, testability of the program is an important property, which is undermined by nondeterminism. Building those safety-critical applications using a deterministic model is therefore desirable.

2.1. The reactor model

The *reactor model* [34] is a model of computation for concurrent and distributed programming with deterministic semantics. It borrows concepts from other related models of computation, like discrete-event systems [47], synchronous-reactive languages [20, 4], and dataflow models [26, 31]. The following sections informally introduce the key concepts of the model.

2.1.1. Structure of reactors

Reactors *Reactors* model individual components of the system. A reactor is a collection of routines, called *reactions*, that have access to a shared internal *state*. This may sound similar to the basic premise of object-oriented programming, but while an object's external interface is usually composed of routines (methods) that mediate access to the state variables, reactions are also internal to the reactor. Instead of exposing methods, reactors expose tagged data streams, reified by *ports*: *input ports* describe what data is depended on by the reactor, *output ports* describe what data may be produced by the reactor.

Reactions are the basic unit of execution in a reactor. They can read from input ports of the containing reactor, and write to output ports, provided these dependencies on components are explicitly declared. A reaction may be scheduled for execution when one of its declared *triggers* produces an event. Triggers include ports, which emit events when they are set to a new value, actions, and timers.

Actions are event sources internal to the reactor, which may be scheduled explicitly by a reaction, periodically, or through an external physical event. *Timers* are similar to actions, but trigger periodically and may not be scheduled explicitly.

Events *Events* are the messages sent from one reactor to another (through ports), or emitted internally by actions and timers. Events bear a timestamp, called a *tag*, that orders them along a *logical timeline*, where two events that have the same tag are logically instantaneous. Events are handled in tag order, and simultaneous events are handled in a predefined, deterministic order, that is determined by the dependencies of reactions. That ordering is an important component of the reactor model, as it models causality between events of a reactor program. It is a central topic of Chapter 3.

The handling of an event involves executing the reactions that have registered a dependency on its source. These reactions execute at the same logical time as the tag of the event: reactions are logically instantaneous. Ports written to by a reaction may trigger new reactions at the same logical time step, causing a "chain reaction". Statically checking that dependencies between ports are acyclic ensures that chained execution of reactions terminates. Actions triggered by a reaction only trigger new reactions at a strictly greater logical time, which allows for cyclic and periodic control flow.

Composition Reactors compose: a reactor may contain other reactors, and connect their ports to each other, or to its own reactions and ports. Contained reactors are black boxes, which can only be used by binding to their ports. A composition of reactors forms a hierarchical tree, where each reactor only has access to its direct children. The root of the tree is the outermost reactor, and it cannot have inputs or outputs.

The model also describes a way to reconfigure the dependencies between components at runtime, called *mutations* [34]. Mutations are special reactions that may create new children reactors and alter connections of the dependency graph. While mutations have been described in the reactor model, they are not yet part of the main implementation of the model, the LF language. For this reason, they are not described in this document.

Dependency graphs All together, dependencies between reactor components (reactions, ports, actions, sub-reactors) can be represented as a directed *dependency graph*, as illustrated in Figure 2.1. The outermost box represents a reactor. Ports are represented by the black triangles; inputs are on the left, and outputs on the right. In this figure, black arrows represent instantaneous dataflow. For instance, if the port i_0 receives some data, n_0 will execute at the same tag, as if instantaneously. In turn, n_0 may push data to the port i_0 of the contained reactor R_0, which will also be observed at the same tag. Notice that ports may be connected together, as is the case between the ports o_0 and i_0 of the child reactors R_0 and R_1, respectively. This means any value produced by o_0 will also be produced by i_0 at the same tag.

Yellow arrows represent control flow that is mediated by actions, and is therefore not instantaneous. For instance, n_1 may schedule the action a_1, which triggers n_5. The

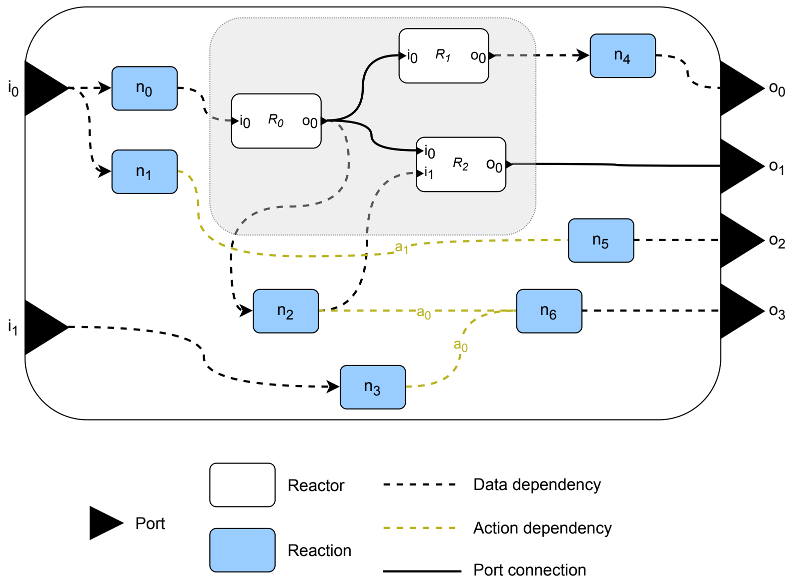


Figure 2.1.: Schematic representation of a reactor. State variables are omitted. Connections between components are derived from their dependencies. For instance, reaction n_3 declares that it reads the input i_1, and may schedule the action a_0. Notably, the subgraph in black (data dependencies, and port connections) must be acyclic: it describes dependencies between reactions that execute the same logical time step.

release of a_1 will occur at a strictly greater tag than that at which the action was scheduled within n_1. Only the subgraph in black is required to be acyclic.

Note about terminology In this thesis, I refer to all relations between components as their *dependencies*, regardless of their orientation. In a narrower sense, the term "dependency" would only describe backward edges (e.g., a reaction *depends* on a port that triggers it), and the term "anti-dependency" is then used for forward edges (e.g., a reaction effecting an action) [34]. Similarly, I use the term *dependency graph* in the abstract sense of "graph of dependencies", without implying that it only contains backward edges. In fact, the "dependency graphs" presented in Chapter 3 only contain forward edges and could be termed *dataflow graphs* instead. Both are equivalent up to the orientation of their connections.

2.1.2. Timing in reactors

The model distinguishes *physical time*, which is measured by clocks, from *logical time*, which is an artificial construct. Logical time does not progress while a reaction is executing: outputs are (logically) simultaneous to inputs. Access to physical time is still useful to reactor programs, because it helps deliver real-time behaviour. For this reason, both timelines are related in a way that preserves deterministic semantics.

Physical and logical time are measured in the same units, and aligned at the start of execution. During execution, logical time is never allowed to get ahead of physical time. This is enforced by waiting for physical time to match the logical stamp of the expected processing of actions.

This synchronization of the logical timeline with the physical timeline is done on a best effort basis. This depends on the real-time capabilities of the execution platform, and the runtime load on the app. For instance, congestion in the event queue may cause logical time to increasingly lag behind physical time. However, those external factors will not influence the order of processing of events, which is still determined by the logical time stamps of the events, so it does not affect determinism of the program.

Superdense time The reactor model supports a concept known as *superdense time*: two logical time values that appear to be the same are not necessary simultaneous. Conceptually, for every logical time value, there is an infinite sequence of *microsteps* that are not simultaneous.

This explains how logical actions can be scheduled with a zero logical time delay. Since they're still required to execute strictly after the time they're scheduled at, they're scheduled at least one microstep in the future.

2.1.3. Distributed execution

The port-based communication of reactors abstracts away the communication medium used to distribute messages. In fact, reactor programs on different machines may communicate with one another over a network. In such a distributed setting, the multiplicity of clocks requires attention to preserve determinism in the whole system. One problem is that imperfectly synchronized clocks between two hosts could lead messages to be improperly ordered, which could break causality in the system. Similarly, because of the transit time of messages across the network, when we pick an event to process, it is possible that some prior event emitted by another machine is on its way, but was not yet received.

Following the ideas used by PTIDES [47], the reactor model uses assumptions on characteristics of the network, and the clock synchronization error across hosts, to determine a *safe-to-process* time for incoming messages. This is possible because of the special relation between physical time and logical time [34]. Clock synchronization errors may be further kept in check by a time synchronization protocol like the Precision Time Protocol (PTP).

2.2. Lingua Franca

Lingua Franca (LF) [33] is a coordination language designed to write reactor programs. The language provides syntax to describe the high-level structure of the program using the concepts of the reactor model. Within this structure, the business logic of the application is written in the *target language*, a general-purpose programming language. Before execution, the LF program is compiled to translate the LF structure into a program written entirely in the target language. The supported target languages are C, C++, TypeScript, Python, and thanks to this work, Rust.

An LF program consists of a source file which contains the declaration of a *main reactor*. This file may reference other LF source files using import statements. Source files are loaded by the compiler recursively, and together with the main file constitute the sources of the LF program.

```
1 target Cpp;
2 main reactor Hello {
3   reaction(startup) {=
4     std::cout << "Hello, world!\n";
5   =}
6 }
```

Listing 2.1: A Hello World program in LF.

Listing 2.1 shows a simple "Hello World" program written in LF using the C++ target. The target is declared on line 1. This will instruct the LF compiler to use the C++ code generation backend. On line 2, we can see the declaration of the main reactor. The main reactor is the entry point of the program. It may contain other reactor instances, and together, all reactor instances of the program form a tree rooted in the main reactor. On line 3, the `reaction` keyword is used to define a reaction. The contents of the parentheses that follow are a list of *triggers* for the reaction, here only the `startup` trigger, which is triggered when the program starts. The block delimited by `{= ...=}` defines the body of the reaction. This code will be executed when a trigger for the reaction fires, here, at startup. These special delimiters are used to enclose target code in many places in the LF grammar.

Reactions Reactions declare their dependencies in their *signature*, namely,

— their *triggers* (timers, actions, inputs, or outputs of contained reactors). These components emit events that cause the reaction to be scheduled for execution;

— the components that the reaction *reads* from but that do not trigger it;

— the components that the reaction *effects*, e.g. ports whose values are set by the body of the reaction.

A more complete example of a reaction declaration is shown in Listing 2.2, showing the placement of those different dependency kinds in the signature.

Reactions are anonymous: they can only be invoked following an event related to their triggers, never invoked explicitly by, e.g., a method call.

```
reaction(triggers) reads  → effects {=
  /* target language code */
=}
```

Listing 2.2: Syntax for a reaction declaration.

Composing reactors Reactors may contain other reactors, and create connections between their ports and the ports of the reactors they contain. For instance, Listing 2.3 defines a nested reactor, and binds its own ports to it using the → operator.

The `ParentReactor` has no vista into the internals of the child reactor; it may only see its ports. From the point of view of the child reactor, the `ParentReactor` is completely invisible, and only ports are used to represent the external world. This port-based design contrasts with the actor model, where every actor has a direct reference to every other actor it may send messages to.

```
reactor ParentReactor {
    input in:  int;
    output out: int;

    printer = new PrintReactor();
    in  → printer.in;
    printer.out  → out;
}
```

Listing 2.3: A reactor which contains an instance of the PrintReactor, defined in Listing 2.4.

2.2.1. Event sources

Reactions are only scheduled in response to an event of one of their triggers. The main sources of events that have already been mentioned are ports and actions.

Ports A port emits an event when it receives a new value, which causes its downstream reactions to be triggered.

Ports are either declared as an *input* or an *output* of a reactor. In Listing 2.4, the `PrintReactor` declares an input named `inp` and an output port named `out`. Both port declarations must specify the type of the data after the colon, here, `int`. This type is target-code, and is not parsed or interpreted by the LF compiler. The reaction on line 5 just forwards the value from the input to the output, printing it in the process. The `set` call will cause reactions that have a trigger dependency on `out` to be executed at the same tag.

```
reactor PrintReactor {
  input inp: int;
  output out: int;

  reaction(inp) →  out {=
    int port_value = *inp.get();
    std::cout << port_value << '\n';
    out.set(port_value);
  =}
}
```

Listing 2.4: A C++ reactor that forwards an input to an output through a reaction.

Actions Actions are either physical or logical. Physical actions can be scheduled asynchronously, while logical actions may only be scheduled from within a reaction. "Scheduling" an action here means committing to executing the dependent reactions at some future tag. That expected release time is the current physical time for physical actions. Logical actions are scheduled relative to the current logical time, because they're scheduled from within reactions (which execute on the logical timeline, conceptually).

Actions may be scheduled with a value, as shown in Listing 2.5 on line 14. This value may then be retrieved within reactions triggered by the event, as shown on line 7.

```
1  target Cpp;
2  reactor PhysicalAction {
3    physical action sensorData: int;
4    output out: int;
5
6    reaction(sensorData) →  out {=
7      out.set(*sensorData.get());
8    =}
9
10   reaction(startup) →  sensorData {=
11     auto thread = std::thread([&] () {
12       while (true) {
13         if (/* sensor receives something */)
14           sensorData.schedule(/*get sensor value*/)
15       }
16     });
17     thread.join();
18   =}
19 }
```

Listing 2.5: A C++ reactor which declares a physical action and uses it to handle asynchronous events produced by another thread.

Timers Timers are event sources that schedule events periodically on the *logical* timeline. For instance in Listing 2.6, a timer named `t` is declared, which will emit its first event 1 second after program startup, and then emit a new event every 100 milliseconds. This will schedule the reaction periodically, since the reaction declares that it is triggered by the timer (line 5). Contrary to the events produced by actions, the events produced by timers may have no associated value.

```
1 reactor TimedReactor {
2   timer t(1 sec, 100 msec);
3   output out: bool;
4 
5   reaction(t) → out {=
6     out.set(true);
7   =}
8 }
```

Listing 2.6: A C++ reactor whose reaction is triggered by a timer.

Startup and shutdown Two special triggers are available in all reactors: `startup` and `shutdown`. The first is triggered only on the very first tag processed by the application. Reactions triggered by `startup` are commonly used to initialize reactors, and, for instance, to set up asynchronous threads that will produce events through physical actions.

Conversely, `shutdown` is triggered on the last tag processed by the program. The application might shut down because of a timeout, or because a reaction requested that it stop (each target provides a function for reactions to do so), and in both cases, `shutdown` is triggered at that moment. Any events produced with a tag greater than the shutdown tag are ignored.

2.2.2. Target properties

LF provides a mechanism to configure target-specific code generation properties, and parameters of the runtime. These are called *target properties*, and are declared in a block within the target declaration, as illustrated in Listing 2.7.

Target properties are defined as key-value pairs, with a syntax reminiscent of JSON [24]. In Listing 2.7, the property `build-type` is bound to the value `"Release"`. This property is defined by the C++ target to integrate with the CMake build tool; here, it instructs CMake to optimize the generated program. The property `keepalive` is a configuration parameter for the runtime. Here, it instructs the runtime to continue execution even if the event queue of the program is empty. The current LF targets have to use this if asynchronous threads may produce new events (through physical actions) even after all events of the program have already been processed.

```
target Cpp {
  build-type: Release,
  keepalive: true,
};
```

Listing 2.7: Example target declaration featuring target properties.

2.2.3. The LF compiler

The LF compiler, called LFC, has a single compiler frontend for all targets, and one backend code generator for each target. It is developed in the GitHub repository lf-lang/lingua-franca[1]. Various satellite projects are hosted in the same GitHub organization namespace, for instance to host the runtime libraries for the various targets (including the Rust target, at lf-lang/reactor-rust[2]), or integrated development environment (IDE) plugins.

The compiler frontend parses LF source into an abstract syntax tree (AST), on which various attribution and validation passes are performed. Some semantic checks use dependency graph representations of the reactor program, which are built by the frontend. The AST, together with these dependency graphs, are used as input to the backend code generators once the frontend is done validating the program's structure.

Compared to a traditional source-to-binary compiler, LFC's backend emits source code in the target language. The target compiler, if any, is then called to compile the generated sources. The target compiler performs type-checking and other semantic analyses on the target program, so LFC does not implement those in its frontend. This strips LFC's frontend of a lot of complexity associated with those passes; LFC does not even have to parse the target code.

Figure 2.2 shows a high-level overview of the compilation pipeline for an LF program with C++ target. The LF source program is first handed to the frontend, which parses an AST and validates it. The tree is then passed on to the appropriate backend code generator, selected by the target declaration in the LF source file, e.g. `target Cpp`. The alternative backend code generators are not used here, but all code generators are included in the LFC binary. The code generator then emits a C++ program, the *target program*.

LFC then uses C++ tools to build an executable from the target program. This step of the pipeline is target-specific, for instance, the Python target does not need a final compilation step, since Python is an interpreted language.

One characteristic all targets share, is that the generated target program depends on a *runtime library*, which implements the runtime scheduling infrastructure for reactor programs. How this runtime library works determines how the generated program binds to it, and so it determines the form of the generated target program. There is therefore some degree of coupling between each code generator and its target-specific runtime

[1]`https://github.com/lf-lang/lingua-franca`
[2]`https://github.com/lf-lang/reactor-rust`

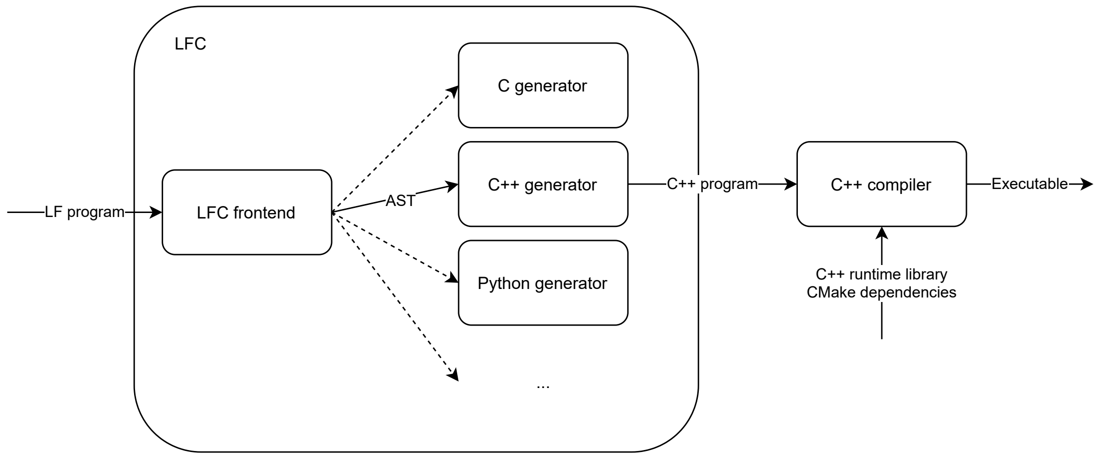


Figure 2.2.: High-level overview of the compilation pipeline for an LF program with C++ target.

library.

2.2.4. Targets

This section gives an overview of the existing targets and their respective specificities.

C

The C language is ubiquitous in the world of embedded systems, because of its being universally supported by embedded platforms. It is used as a *lingua franca* by foreign function interface (FFI) implementations of many other programming languages. Because C code can be made very efficient, the LF C target was initially conceived to evaluate the minimal amount of runtime overhead incurred by LF's deterministic execution semantics.

The LF C target features two runtime implementations, that provide the same API to LF reactions. The *unthreaded runtime* is built to have minimal dependencies, so that it is able to run on low-level embedded platforms. The second runtime depends on a POSIX thread library. This enables parallelization of reactions, and adds support for handling asynchronous events (physical actions), which the unthreaded runtime lacks.

Given C's weak type system, and its lack of language features to support encapsulation or memory management, enforcing that the code of reactions is well-behaved (e.g., that it does not access state of neighbouring reactors) is challenging. Target code is therefore *assumed* to respect the semantics of the reactor model. This makes the programmer responsible for ensuring it does, which may be a source of bugs, compared to stricter languages like C++.

To date, the C target is one of the three targets (along with Python and TypeScript) that support *federated execution*, LF's distributed execution mechanism (cf. Section 2.1.3).

C++

The C++ target is based on a standalone library that implements concepts of the reactor model using object-oriented programming (OOP), only depending on the Standard Template Library (STL). The library, reactor-cpp[3], can be used on its own as a C++ framework, where users implement new reactors as classes extending base classes defined by the framework.

The LF code generator was developed a posteriori, and links generated reactor programs to the existing framework. The use of OOP allows the C++ generator to map LF reactor classes directly to C++ classes, without having to compute the parameters of every reactor *instance* in the program at compile-time (as C does). In effect, the C++ generator translates LF ASTs directly to target code, while the C generator needs to unroll the full dependency graph of the program at compile-time. This makes the implementation of the C++ code generator significantly simpler than the C code generator. The C++ runtime framework, on the other hand, creates this dependency graph at runtime, during initialization of the program, to compute the dependencies of reactions.

Contrary to the C target, the C++ target is able to take advantage of generics to enforce type safety of port connections at compile-time. Ports are indeed defined as a generic class, and can only be connected to ports with the same value type. C++ ports also leverage smart pointers to enforce some invariants of the reactor model, for instance, that a value set into a port cannot be mutated anymore by the calling reaction. These smart pointers implement ownership semantics reminiscent of Rust's ownership model, which will be covered in Section 2.4.2. C++ can also be used to enforce stronger encapsulation of reactor internals than C, by only exposing ports as public members of the reactor class. However, C++ reaction code can still break the semantics of the reactor model, for instance, by sending a reference to its internal state to another reactor. While this is in some respect impossible to prevent, short of statically verifying the target code, the C++ target still offers significantly more safety against accidental programmer mistakes than the C target.

Python

Python is a high-level dynamically typed language, that boasts a vast ecosystem of libraries and frameworks, accessible through an officially supported package management system (Pip). CPython, the reference implementation of the language, provides a convenient FFI for interfacing with C. It allows library writers to write module implementations in C for better performance, while exposing idiomatic Python bindings for consumption by client Python code. This is the approach taken by the Python target, as its runtime library wraps the C target's runtime.

The Python code generator generates Python classes for each reactor class, to expose

[3]`https://github.com/lf-lang/reactor-cpp`

an idiomatic API to reactions. To support Python's dynamic type system, LF state variables, parameters and ports do not need to specify static type annotations.

TypeScript

TypeScript is a scripting language, which extends JavaScript with a static type system. This enables TypeScript projects to implement *gradual typing*, whereby initial development is accelerated by omitting type annotations, and the codebase is subsequently gradually consolidated by the addition of static type annotations.

The TypeScript target uses Node.js [45], a JavaScript runtime, as it allows LF programs written for the TypeScript target to integrate with the large library ecosystem provided by the Node package manager (NPM). Node.js does not provide access to threads, and the TypeScript target can therefore not support parallel execution of reactions. Compared to the unthreaded C runtime though, physical actions are still supported.

2.3. Related work

The reactor model has several close relatives, mostly part of a family of models of computation emphasizing explicit message passing between components of the system [32]. This contrasts with approaches to concurrency that rely on shared memory for communication [37].

Actors [21, 1] are a model of computation in which the system is modelled as a collection of concurrent objects communicating via asynchronous message passing. One problem with actors is their inherently nondeterministic communication style: even assuming messages are delivered in the order they were sent by the communication medium (e.g., a network), very simple actor networks still may exhibit nondeterminism. This is illustrated in Figure 2.3. In this figure, even though both messages A and B may be sent in a determinate order from the Source actor, making message A transit through the Relay actor causes nondeterminism: the Target actor may receive both messages in either order. Despite their shortcomings, actor programs exhibit a high degree of concurrency, and are widely used for their high performance. Notable actor-based programming languages include Erlang [3] and Pony [7]. The latter uses its type system to enable highly parallel garbage collection, and guarantee data race freedom, like the Rust type system. Library implementations include Akka [6], an actor framework written in Scala, and Actix [44], a Rust framework. Akka was used in this thesis to evaluate the performance of the LF Rust target. Actix relies on Rust's support for cooperative scheduling (`async` / `await`) to deliver high-performance, and is mostly geared towards high-throughput web applications.

Other models resembling the actor model have deterministic semantics. In dataflow models [31, 26, 17, 38], connections between components represent the flow of data from a producer to a consumer. Computation in the consumer actor is triggered when its input data is available, i.e., has been produced by the upstream actor. Dataflow networks provide deterministic concurrency, and static information on connections enables exten-

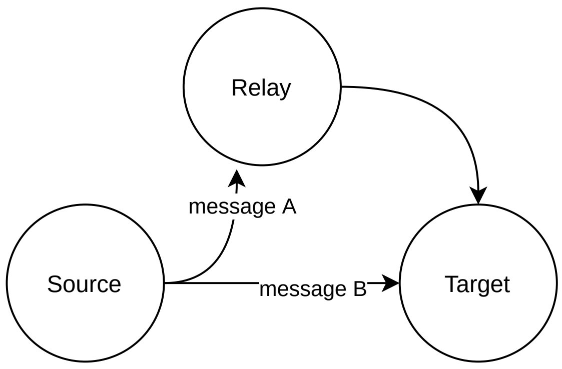


Figure 2.3.: A network of three Hewitt actors.

sive static analysis and efficient ahead-of-time scheduling. On the other hand, dataflow models have generally no way to provide real-time behaviour, which is required of most cyber-physical systems.

Under synchronous/reactive (SR) [20, 4] approaches, actor reactions are conceptually aligned on the ticks of a global logical clock. Reactions produce their outputs at the same logical time as their inputs, so conceptually, reactions are assumed to be instantaneous (the *synchrony hypothesis*). This family of techniques can be given deterministic semantics, and compiled very efficiently [20]. They have also been applied to distributed systems [5, 42]. However, contrary to LF's approach, clocks in these languages are purely logical, which undermines their ability to provide real-time behaviour.

Discrete-event (DE) systems are another model, where all messages sent between actors bear a timestamp, and are processed in timestamp order. DE systems synchronize their logical timeline, defined by the order of message timestamps, to the physical timeline (wall clock time). This allows these systems to provide real-time behaviour, but DE systems have historically been confined to rather specialized use cases (e.g. hardware simulation). LF's tagged events are inspired by the semantics of DE systems.

2.4. Introduction to Rust

This section introduces some of the syntax and features of the Rust language. The goal of this section is to give the reader unfamiliar with Rust a general feel for the language, and allow them to understand Rust code listings in later chapters. The section also serves as a reference for the reader to jump back to, whenever later chapters mention language features of Rust they are unfamiliar with.

As is customary, let us begin our tour of Rust with a Hello World example, shown in Listing 2.8. Like in main other languages, the entry point for rust programs is a function called `main`. Function declarations in Rust start with the `fn` keyword, followed by the name of the function and its parameter list.

The body of the function is a block. Here, line 2, we start by defining an intermediate variable called `world`, using the `let` keyword. The type of the variable is inferred from

```
1 fn main() {
2   let world = "world";
3   println!("Hello, {}!", world);
4 }
```

Listing 2.8: A program which prints the string "Hello, world!" to standard output.

the right-hand side, here it is the type of a string literal. Rust has very elaborate type inference, which allows most types in local variables to remain implicit. Rust is still strongly, statically typed.

The statement that follows, line 3 calls the *macro* `println`. Rust macro invocations look like function invocations, but they use an exclamation mark (`!`) in their identifier. Macros are the topic of Section 2.4.5. The `println` macro takes a format specifier as its first argument, which it parses at compile-time to generate efficient printing code, and statically ensure that the arguments match the format specifier. In the format specifier, the string `{}` is a placeholder which will be replaced by the second argument to the macro, here, the variable `world`.

The following section introduces the basic concepts and data model of the language.

2.4.1. Data model

Primitive types

Integer types Primitive types in Rust include integer types like `i32`, a signed integer type of 32 bits, or `u8`, an unsigned integer type of 8 bits (a byte). Contrary to C and C++, Rust specifies an exact size for primitive integer types. Arithmetic overflow is prohibited by default and results in a runtime error, for applications built with assertions enabled. This is useful as unintended overflow is a source of bugs in C and C++, and can easily cause memory corruption [46].

String slice type Rust also predefines a string type called `str`, which represents a sequence of UTF-8 encoded characters (a *string slice*). `str` is most often used with a reference, like `&str`, as it has variable size and cannot be placed on the stack. It is to be contrasted with `String`, which is a mutable-size character buffer.

Unit Another primitive type of interest is the *unit* type, written `()` in Rust. This type has a single value, also written `()`, and is the type of blocks and functions that don't return an expression (of another type), for instance, the type of the empty block `{}`. The presence of this type makes the type system more consistent than the `void` type of C, or that of Java: in Rust, all functions return a value, even if that value is `()`.

Many constructs that are statements in C or C++ are expressions in Rust, meaning, they have a type and evaluate to a value. For instance, an `if/else` statement has the same type as its branches. Blocks have the type of the last expression that is written in them, or if it is missing, the unit type. This allows writing a function that returns a value without an explicit **return** statement, such as the one in Listing 2.9. Notice that the return type of the function is written postfix, after the → symbol.

```rust
fn sum(a: u32, b: u32) -> u32 {
  a + b
}
```

Listing 2.9: Definition of a Rust function that implicitly returns its last expression.

Expressions that have type `()` are commonly called statements. For instance, an **if** without an else branch is a proper statement, and has type `()`.

User-defined types

The most important user-defined data types in Rust are *tuples*, *structs* and *enums*[4].

Tuples Tuple types are anonymous data aggregates. For instance, the type `(u32, u32)` is a pair with two `u32` fields. Individual fields bear numeric identifiers. In Listing 2.10, the sum function of Listing 2.9 is rewritten to use a tuple parameter.

```rust
fn sum(pair: (u32, u32)) -> u32 {
  pair.0 + pair.1
}
```

Listing 2.10: Example of field access on a tuple.

Structs Struct are named data aggregates, similar to C's structs. The memory layout of structs is flat, so that structs may not contain a field of their own type recursively. The memory layout of Rust data types may be controlled with compilation flags, to be highly interoperable with C/C++ [8].

Structs can be defined in three ways, illustrated in Listing 2.11. The declaration for `Instant` (line 2) defines a struct with two named fields, `time` and `step`.

The declaration of `ReactorId` (line 5) defines a *tuple struct*, with a single unnamed field of type `u16`. This form is often used to define so-called *newtypes* over another type. A newtype is a design pattern in languages like Haskell, whereby one defines a type wrapping another without necessarily providing access to the wrapped instance. This can be used to provide new behaviour, or on the contrary, to constrain the available behaviour on an data type. These are usually used to communicate semantic information about the intended usage of the instance, for instance the `ReactorId` struct defined on line 5 is implemented as an integer of width 16, but this is an implementation detail that

[4]Rust also supports defining C-like unions, but they are not memory-safe and so are rarely used.

```
1 /// A struct that defines two fields of type Instant and i32 (integer)
2 struct Instant { time: Instant, step: i32 }
3
4 /// A "tuple struct", with a single unnamed field.
5 struct ReactorId(u16);
6
7 /// A zero-sized struct
8 struct Token;
```

Listing 2.11: Different forms of struct declarations.

code getting access to a `ReactorId` should not rely on. With the newtype pattern, the `u16` field is hidden from the users of `ReactorId`.

The declaration of `ReactorId` (line 8) defines a struct that has a unique value. Since the struct has no fields, it also has size zero at runtime. Such structs are sometimes useful for type-level programming.

```
let instant = Instant {
  time: Instant::now(),
  step: 0
};
```

Listing 2.12: Construction of an instance of the `Instant` struct defined in Listing 2.11. Struct initialization expressions resemble the struct declaration, but for each field, the part right of the colon is its initial expression, instead of its type.

To create a struct instance, all fields must be initialized, as shown in Listing 2.12. This contrasts with languages like Java, where the code of a constructor can observe uninitialized fields (set to their default value, e.g. `null`), and also give out a reference to a partially initialized `this`. This is why Rust, like OCaml for instance, does not need a null value.

Enums Enums are tagged unions, that is, union types which carry a *discriminant* at runtime that identifies which variant they represent. They can be used as simple enumerated types which define constants, but are much more powerful.

```
enum Request {
  Put { url: URL, content: String },
  Get { url: URL }
}
```

Listing 2.13: An enum declaration.

Listing 2.13 shows the declaration of an enum `Request`. The enum declares two variants, which themselves declare an anonymous struct: they can declare fields, but do not declare a type. The expression `Request::Get { url: someUrl }` creates an instance of the second variant, and has type `Request` (not `Request::Get`). Enums can only be used by pattern-matching on their variants.

Pattern-matching Struct and enums implement the product and sum types commonly found in functional languages like OCaml, and Rust uses this type-level structure to check the exhaustivity of pattern matching constructs. Pattern-matching allows concisely expressing variable bindings and structural tests that differentiate enum variants. The match expression is used for that purpose. In Listing 2.14, the match expression on line 2 destructures the `request` parameter. Since `Request` is an enum type, an exhaustive match statement must have a branch for each of the enum variants `Put` and `Get`. On line 5, the `Put` pattern binds each field of the variant to a local variable binding with the same name. If the request parameter is a `Put` variant, the match expression will execute the code in that branch and print "Put(url=..., content=...)". Note that **`match`** is an expression, which has the same type as its branches (here, `()`).

```
1 fn print_request(request: Request) {
2   match request {
3     // this branch binds fields of the request to
4     // variables on the right of the arrow
5     Put { url, content }
6       => println!("Put(url={}, content={})", url, content),
7     Get { url } => println!("Get(url={})", url),
8   }
9 }
```

Listing 2.14: Pattern-matching on a `Request` instance (enum defined in Listing 2.13)

Match-expressions are required by the Rust language to be *total*, i.e., any possible value of the subject expression must be covered by a branch. In Listing 2.14, both enum variants for `Request` are covered, which makes the match total. Total patterns can also be used as the left-hand side of the assignment operator, a usage which is usually referred to as *value destructuring*. For instance in Listing 2.15, a tuple value is destructured on line 2, thereby binding new variables to its component fields. The pattern `(x, y)` is total on the type `(u32, u32)`, as it cannot fail.

```
1 fn destructure(tuple2: (u32, u32)) {
2   let (x, y) = tuple2;
3 }
```

Listing 2.15: Destructuring a tuple.

2.4.2. References and ownership

Perhaps the most salient feature of Rust is its use of an ownership model to manage memory at compile-time. Each value in Rust is *owned* by exactly one variable, called its *owner*. Ownership can be transferred from a variable to another by *moving*, and assignment on most types has move semantics. When a value is moved from one variable to another, the previous variable cannot be used anymore. As this is very restrictive, ownership may be temporarily transferred by *borrowing*, which creates a reference to the value.

The language uses its type system to enforce its ownership discipline, by having types carry ownership information. For any given type `T`, a variable with type `T` owns its value, while a variable of type `&T` (read “reference to T”) is only *borrowing* the value from its owner (of type `T`). The following example illustrates the difference between the types `Vec<i32>`, which has exclusive ownership of the value, and `&Vec<i32>`, a *shared reference* to the value, which does not have exclusive ownership (`Vec` is similar to `std::vector` in C++).

```
1  fn consume(pv: Vec<i32>) { } //   pv has exclusive ownership
2  fn consume_ref(rv: &Vec<i32>) { } // rv borrows ownership
3
4  fn main() {
5    let v1: Vec<i32> = vec![1, 2]; // this macro creates a new vector
6    let v2: Vec<i32> = vec![1, 2];
7
8    consume(v1); // v1 is moved into the function parameter
9    println!(v1); // error: the value in v1 was moved
10
11   consume_ref(&v2); // let the function borrow v2
12   println!(v2); // v2 still has exclusive ownership of the value
13 }
```

Listing 2.16: Example of moving and borrowing ownership.

In Listing 2.16, the call to `consume` (line 8) *moves* ownership of the vector value into the function parameter `pv`, because `pv` is declared with an owned type line 1. The caller loses access to the moved value, and so the `println!(v1)` statement fail to compile. In contrast, `v2` is only borrowed, using the & operator. This means the value is accessible in the body of `consume_ref` through a reference, but does not have exclusive ownership. When that function returns, `v2` still has exclusive ownership of the value, because the shared reference is provably dead.

Notice that the program does not do any explicit memory management. The compiler tracks when each owned value goes out of scope to insert a call to its destructor. For `v2`, this happens at the end of the `main` function, while for `v1`, this happens at the end of the `consume` function, since its ownership has been moved.

Mutability and aliasing The Rust language places restrictions on the interaction of mutation and aliasing. Several immutable references to the same data may be live simultaneously (thus *aliasing* the memory location), but mutation of the data is impossible while it is aliased. The Rust compiler enforces that by tracking the liveness of borrowed references, and emitting an error when the lifetime of a mutable borrow overlaps with the lifetime of any other borrow of the same data. This restriction is meant to statically prevent race conditions that occur when aliased data is accessed in an unsynchronized fashion [25, 28].

Listing 2.17 shows an example where a mutable and immutable borrow overlap, which is prohibited by the compiler. The last statement line 10 uses `refmut` to extend its scope, so that it overlaps with the immutable borrow line 8.

```
1  fn main() {
2    let mut v1: Vec<i32> = vec![1, 2]; // v1 owns the value
3
4    let refmut: &mut Vec<i32> = &mut v1;
5    refmut.push(3); // we can mutate the vector through the mutable reference
6    println!("{}", refmut[3]); // we can also use it for read operations
7
8    let refshared = &v1; // error: v1 is already borrowed as mutable
9
10   refmut.push(4);
11 }
```

Listing 2.17: Two overlapping borrows.

Summary In summary, for any type `T`, three variants have different ownership semantics:

— `T` has exclusive ownership, assignment on variables of type `T` moves the value;

— `&T` is a shared reference, many of them can be alive at the same time, but the data cannot be mutated;

— `&mut T` is a mutable reference, which has to be the only live reference to its data during its whole lifetime.

Lifetime quantification

Rust statically guarantees that references point to valid data while they are live, thereby preventing bugs associated with “dangling pointers”. This property is derived by the compiler, by checking that the object referred to (the *referent*) by a reference *outlives* the reference itself. For instance Listing 2.18 declares a reference `r` and sets its value to a borrow of `x`, which is declared in a smaller scope. Because `x` goes out of scope before the next usage of `r`, the usage of `r`, line 7, is invalid, as its referent variable does not

```
{
  let r: &i32;
  {
    let x = 5;
    r = &x;
  }
  println!("r: {}", r); // usage of r
}
```

Listing 2.18: An attempt to use a reference whose referent has gone out of scope — this produces an error [27].

```
struct Relay<'a, T> {
  pointer: &'a T
}
```

Listing 2.19: A generic struct which declares one lifetime parameter `'a` and a field of a reference type.

exist anymore. This produces a compile-time error, as the variable `x` does not live long enough.

The compiler is able to check this by comparing the *lifetime* of the reference `r` to the lifetime of the referent `x`. The lifetime of a variable is roughly equivalent to its lexical scope.

Most Rust code manipulating references does not need to deal with reference lifetime explicitly. In Listing 2.18, for instance, the compiler infers the lifetimes of the borrows from the structure of the code. Sometimes it's necessary to annotate code in order to deviate from the default inference behaviour.

For instance, when structs contain references, those must have an explicitly quantified lifetime. The struct must in this case declare a *lifetime parameter*, which looks like a generic parameter, but whose identifier starts with a single quote. For instance in Listing 2.19, the struct declares one lifetime parameter `'a`.

Any struct that contain an object of type `Relay` must also specify a lifetime *argument* for the struct type, for instance in Listing 2.20). Lifetime parameters need to be repeated at all layers of a composition of objects, which can be noisy when objects are deeply nested.

```
struct RelayContainer<'a> {
  relay: Relay<'a, i32>
}
```

Listing 2.20: A generic struct which contains a `Relay` object (cf. Listing 2.19).

Static lifetime A special lifetime is the predeclared `'static` lifetime, which is the lifetime of references that life forever. This means their referent is part of the memory image of the executable, like string literals or constants.

2.4.3. Memory safety

Rust's ownership discipline is too restrictive for many low-level use-cases, for instance, to implement mutual exclusion or inter-thread communication. As an escape hatch, Rust has *pointer types* (written `*const T` or `*mut T`), which have no associated lifetime. Pointers are not verified to obey the rules of Rust's ownership model, and hence do not benefit from the same safety guarantees as references. Pointers may point to uninitialized or freed memory, or alias a mutable location in a way that may cause data races. Since dereferencing pointers is not checked at compile-time to be safe, Rust only allows this in `unsafe` blocks.

It is important to understand that unsafe blocks do not allow the programmer to violate Rust's ownership model. Rather, they delimit regions of code where the programmer has to uphold the ownership model themselves, without the help of the compiler. Failure to comply with the ownership model results in *undefined behaviour* (UB). For instance, breaking the reference aliasing rules by aliasing a mutable reference is considered UB [9].

The syntactic overhead on using unsafe code discourages its use, and promotes its confinement to implementations, and not APIs. Use cases that need unsafe code are encouraged to encapsulate it into a safe API: given that safe APIs compose, they can then be used anywhere, while the unsafe code is concentrated only in one place [25].

Smart pointers One particular use case for encapsulated unsafe code is to create types that mediate access to a value of another type, i.e., smart pointers. Smart pointers in Rust are elevated to the rank of design pattern, the standard library is full of them. For example, a `Mutex<T>` provides atomic access to values of type T. One can only acquire the inner value by acquiring the lock, possibly blocking the thread until the lock has been released by other threads.

Similarly, since it's impossible to move the same value into several places, one cannot share a value in safe Rust without using a borrowed reference, and having to deal with lifetimes. An alternative is to use the `Rc<T>` smart pointer, which can be freely cloned into a new owned `Rc<T>`. This smart pointer performs reference-counting, which is one way to ensure the inner value is only dropped once, even though a multitude of `Rc` may be created. An `Rc` hence allows multiple readers on the same value, without dealing with lifetimes. However, `Rc` does not allow mutable access to its data, since it could be used to duplicate a memory location.

Smart pointers are very useful in practice, as they extend the language semantics; for instance, `Rc` allows an object to have several owners, which is not possible with regular Rust types. Similarly, the smart pointer `RefCell` allows an object to be mutated even though it is accessed through a shared reference. This pattern is called *internal mutability*, and is very useful in practice. However, this may cause data races in a concurrent setting, as shared references are not unique. The Rust compiler performs static analysis to ensure that internally mutable types are not used where a thread-safe object is expected.

2.4.4. Abstraction in Rust

Software abstractions enable decoupling client code from implementation details they don't need to know about. This section introduces some of the language features that enable data encapsulation and polymorphism in Rust.

Namespacing and visibility

Member functions Rust types can define member functions of two kinds: *methods* and *associated functions*. Associated functions are regular functions that live in the namespace of the type. Methods are associated functions that also take an instance of the type as its first parameter. Both are defined in an ***impl*** *block*, as shown in Listing 2.21.

```
1 struct Point(x: u32, y: u32);
2 impl Point {
3   fn origin() -> Self {
4     Self { x: 0, y: 0 }
5   }
6   fn manhattan(&self) -> u32 {
7     sqrt(self.x * self.x + self.y * self.y)
8   }
9 }
```

Listing 2.21: Member function definitions on a `Point` struct.

Within an `impl` block, the keyword `Self` can be used as an alias for the current type (here, `Point`). We also call this the *self type*. On line 3, we define an associated function, which does not take parameters. This can then be called with the expression `Point::origin()` from outside of the type.

On line 6, we define a method, because it has a `self` parameter. It takes its parameter by reference, since that parameter is declared `&self`. Within the body of the function, `self` is an expression that has type `&Self`, and so we can access fields of the point on it, line 7. The method can be called with the syntax `point.manhattan()`, where `point` is any expression of type `Point`. It can also be called like a regular associated function (`Point::manhattan(&point)`), but this syntax is less convenient as it cannot be chained and requires an explicit borrow.

Module system Types define a namespace for their members, but this is not the only namespacing system of Rust. To provide a namespace for types themselves, Rust uses *modules*. A module is an *item* (like functions or type declarations) that may contain other items. Listing 2.22 shows an example of two module trees. The module `m` defined

```
1  mod m {
2    fn a() {
3      nested::b();
4    }
5    mod nested {
6      use super::a;
7      fn b() {
8        a(); // m::a
9      }
10   }
11 }
12 mod n {
13   pub use super::m::a;
14 }
15 mod sibling;
```

Listing 2.22: Nested module declarations and use statements.

on line 1, defines a namespace containing two items — the function `a` and the module `nested`. To use items of `nested` within the body of `a`, we have to qualify the namespace using the `::` operator, like on line 3.

In order to use items of another module without qualification, it is possible to import them in the current namespace with a `use` declaration, like on line 6. Here, this allows calling the function `a` using its simple name on line 8. Within the path `super::a`, `super` is a keyword that identifies the parent module, here, `m`.

`use` declarations may also be used to *re-export* members under a different namespace, as shown on line 13. Here, the `pub` keyword (for *public*) makes other modules able to use the path `n::a` an alias for `m::a`. This is commonly used to split implementation of a module into separate submodules, then exporting them all under the same namespace so as not to expose the internal module structure to clients.

Not only can modules be declared explicitly with the `mod` keyword, all Rust files implicitly define a module with the same name as the file. In Listing 2.22, the file containing the listing code is hence a module itself, whose members are the modules `m` and `n`. This is why it is possible for line 13 to refer to `m` using the path `super::m`, where `super` refers to the implicit parent module.

Modules declared without a body, like on line 15, are taken by the Rust compiler to be defined in a sibling file with the same name. In this example, `mod sibling` will have all the items declared in a `sibling.rs` file in the same directory. This allows splitting the implementation of a module into separate files.

Crates Rust projects are called *crates*. A crate may contain several *artifacts*, including binary programs, and a *library artifact*. Each of those is defined by a Rust file which acts as the root of a module tree. The root module of a library artifact has the same name as the containing crate. This module is used by dependent projects to refer to the contents of the crate. Crates that only define a library artifact (and no executables) are called *library crates*.

Visibility The default visibility of all items in Rust is *private*, meaning, the item is only accessible within the innermost enclosing module. This is also true of struct fields (which are not items). The modifier `pub` (for *public*) is used to make an item accessible from outside of its containing module.

The `pub` modifier also has several other forms. With `pub(crate)`, an item is made visible to all modules of the declaring crate, but not to other crates. With `pub(<module_path>)`, e.g. `pub(super)`, an item can be made visible in only a subset of the modules of the crate, i.e., those modules that are descendants of the module referred to between parentheses.

An item t is only visible to a client module if there exists a path to t in which every segment (module, type, and t) is visible to the client. It is common in Rust to declare most items public, but instead restrict the visibility of the *modules* that contain them. This hides the structure of the implementation modules, and decouples the public module tree of a crate.

Traits

Behaviour that is shared between types is modelled with *traits*. Traits define an interface for a type as a set of functions supported by the type. They are similar to type classes in Haskell, as they can be implemented even on types that were defined in another project. For example, the Rust standard library defines the trait `Ord`, for totally ordered types. The simplified trait declaration is printed in Listing 2.23:

```rust
pub trait Ord: Eq + PartialOrd<Self> {
  fn cmp(&self, other: &Self) -> Ordering;

  fn max(self, other: Self) -> Self {
    match self.cmp(&other) {
      Ordering::Less | Ordering::Equal => other,
      Ordering::Greater => self,
    }
  }
  // ...
}
```

Listing 2.23: Simplified declaration of the `Ord` trait.

The trait declaration starts with the declaration of the name of the trait, which in this case is followed by a list of *trait bounds* (`Eq + PartialOrd<Self>`). These are other traits that need to be implemented by any type that implements `Ord`. Here, these requirements express that any totally ordered type must necessarily also be partially ordered, and must support a total equality operation.

A trait is not a concrete type, it is simply an interface, i.e., a set of members supported by the implementer. Members may be methods, associated types, associated functions, and associated constants. Within the trait declaration, the concrete type of the implementer can be referred to generically using the `Self` keyword. Trait bounds constrain the `Self` type of an implementation.

The `Ord` trait declares a `cmp` method line 2, which performs a three-way comparison between `self` and another object of the same type (`Self`). This method has no body, which means an implementer must define the implementation of this method. By contrast, the definition of the `max` method on line 4 has a body, which acts as a default implementation, defined using `cmp`.

We can implement `Ord` for the `Instant` struct with a dedicated `impl` block, shown in Listing 2.24. Within this `impl` block, the `Self` type is an alias for `Instant`. `self` also has type `Instant`. The block must implement all unimplemented functions of the `Ord` trait in order to compile, here, `cmp`. Additionally, the presence of an `impl` block for `Instant` makes the compiler require that `Instant` also implement `Eq` and the other trait bounds on `Ord`.

```rust
impl Ord for Instant {
  fn cmp(&self, other: &Self) -> Ordering {
    self.time.cmp(&other.time)
      .then(self.step.cmp(&other.step))
  }
}
```

Listing 2.24: Implementation of the trait `Ord` for the `Instant` struct (cf. Listing 2.11).

Operator overloading Rust uses that `Ord` trait, as well as a few other traits blessed by the compiler, to implement *operator overloading*. The `Ord` trait overloads comparison operators like `<=` or `>`, which will be desugared to a call to `Ord::cmp`. The `Eq` trait (a bound of the `Ord` trait), overloads the equality operators `==` and `!=`. It is possible to overload other select operators, like `+` through the trait `Add`.

Contrary to C++ though, the assignment operator `=` cannot be overloaded. It is also not possible to define new custom operators. This design decision limits the complexity of the language and its parser, and is meant to make Rust programs easier to read.

Generics Rust has practically no subtype polymorphism, and parametric polymorphism is the preferred strategy to achieve code reuse. Structs, enums, traits and functions may declare type parameters, and be used with a range of type. Type parameters may be constrained by *trait bounds*, which require that the actual type argument implement the trait. This allows using functions of that trait on instances of the type parameter. For example, the function in Listing 2.25 requires that its type parameter `T` implement `Ord`, which allows using the method `Ord::cmp` on `a`:

All generic method invocations, and parameterized structs, are monomorphized during compilation. Each monomorphic variant is optimized separately, which is one of the reasons Rust boasts to offer *zero-cost abstraction*. At the same time, this means that type information must be entirely known at compile-time everywhere. For example, there is no equivalent to the Java type `List<?>`, where the type argument is unknown (a "list of some unknown thing").

Virtual dispatch *Virtual dispatch* (also known as *dynamic dispatch*) [36] is a feature of most object-oriented languages, whereby a method call is only linked at runtime based on the type of a receiver object. A single virtual method call site may hence call several implementations of a method. In a language like Java, virtual dispatch is a crucial tool for abstraction.

Java relies on subtype polymorphism to use dynamic dispatch: a variable declared with a class type `C` may contain instances of subclasses of `C`. Calling a method on the variable will dispatch the method according to the actual class of the value (which may be a subclass).

However, as said earlier, Rust has no subtype polymorphism: one cannot use a trait as the type of a variable, or parameter, or return value. This is because variables are allocated on the stack, and so their size must be known at compile-time. But since traits may be implemented by types of different sizes, a value whose type is simply `Ord`, for instance, has no known size, and cannot be put on the stack. This is the reason traits are not considered to be *types* in Rust.

The solution to this problem is to use references: since they are implemented by pointers, references of different types all have the same (machine-specific) size. To do so in Rust, we use variables with type `&dyn Trait` (dynamic reference to some trait `Trait`), which may contain a reference to any concrete implementer of the trait. Note that using references like this is exactly how Java (and many other languages) implements subtype polymorphism: in Rust, it's simply opt-in. Contrary to Java though, where every value contains a reference to its class (and hence to its vtable), in Rust, the *reference* itself needs to carry around the vpointer. Because of this implementation strategy, `&dyn` pointers are called *fat* pointers, since they actually consist of two references.

```
fn max<T : Ord>(a: &T, b: &T) -> &T {
  match a.cmp(b) {
    Ordering::Less | Ordering::Equal => b,
    Ordering::Greater => a
  }
}
```

Listing 2.25: Example of a generic function.

```
1  fn count_if_starts_with(strings: &Vec<&str>, prefix: &str) →  usize {
2    strings.into_iter()
3          .filter(|s| s.starts_with(prefix))
4          .count()
5  }
6
7  fn count_if_starts_with(strings: &Vec<&str>, prefix: &str) →  usize {
8    let mut count = 0usize;
9    for s in strings {
10     if s.starts_with(prefix) {
11       count += 1;
12     }
13   }
14   count
15 }
```

Listing 2.26: Example of using closures together with iterators.

Functional programming

Closures and iterators Rust has some features that pertain to the functional programming paradigm, for instance, it supports *closures*, i.e., anonymous functions that may capture part of their environment. This is especially useful for transforming collections in a functional style, with classic functions like `map` or `filter`. Rust's *iterators* support this programming style, as shown in Listing 2.26. On line 2, a new iterator is created, which is then filtered on the next line using a provided closure function. In Rust's closure syntax, the parameter list is enclosed within pipes (here, `|s|`), and followed by an expression, which is the return expression of the closure. The closure defined here to filter the iterator captures the variable `prefix` from its environment.

The second function defined in Listing 2.26 (line 7) is equivalent to the first, but is written in a more imperative style. On line 9, the `for` loop demonstrates Rust's first-class support for iterators: any object implementing the trait `IntoIterator`, which provides the `into_iter` function called on line 2, can be iterated on in a for loop.

`Option` and `Result` Other features of Rust usually found in functional programming languages are its monadic `Option` and `Result` types. The first represents a possibly absent value, whereas the second represents the result of a possibly failed computation. Their declarations are reproduced in Listing 2.27.

2.4.5. Meta-programming

Macros Rust features a meta-programming system directly integrated into the language. Contrary to C/C++ preprocessor directives, which transform the source file before the compiler even reads it, in Rust, *macros* transform *token streams* produced by the compiler before the parsing phase. This allows the compiler to restrict where macros

```
enum Option<T> {
  None,
  Some(T)
}
enum Result<T, E> {
  Ok(T),
  Err(E)
}
```

Listing 2.27: Declarations of `Option` and `Result`.

are allowed to be used, and most importantly, how they modify their enclosing scope (the declarations they introduce). These restrictions are called *hygiene* by Rust language designers, because their intent is to make macro more readable and less error-prone than preprocessor macros. A nice side-effect of this structured approach is that the compiler is able to produce much more helpful error messages than in C/C++.

Macro invocations look like function calls, except the identifier is suffixed with an exclamation mark. For instance, `println!("abc")` is a macro call, which prints to standard output. Since the contents of the parentheses are interpreted by the macro, they don't need to be a well-formed Rust expression or so. This allows defining convenient domain-specific language (DSL)s that look like human-language syntax.

Attributes Attributes are metadata attached to items, be they functions, type declarations, modules, or otherwise. Attributes allow specifying code generation options for the compiler. They are also used to automatically derive trait implementations from the implementation of a struct or enum, as shown in Listing 2.28.

```
#[derive(Clone, Eq)]
struct Foo {
  a: u32
}
```

Listing 2.28: Example of a derive attribute on a struct declaration.

The attribute on the first line specifies that the `Foo` struct will get an automatically derived implementation of `Clone` , which will clone all fields recursively, and `Eq`, which also will compare all fields recursively. Both of these are possible because all the types of all fields (here, a single `u32`) also implement `Clone` and `Eq`.

Conditional compilation Rust features first-class support for conditional compilation, which is also achieved through attributes. It is for instance possible to provide a different implementation of a module depending on the target platform, like shown in Listing 2.29. In this example, the contents of the `cfg` attribute specify a condition that is evaluated by the compiler during compilation to determine whether the item is to be compiled or not. Since both `cfg` attributes in the example have mutually exclusive conditions, a unique implementation for the module will be compiled, and client code can just depend on it normally.

```rust
#[cfg(os = "windows")]
pub mod os_specific_api {
  pub fn print_os() {
    println!("windows!")
  }
}
#[cfg(not(os = "windows"))]
pub mod os_specific_api {
  pub fn print_os() {
    println!("not windows!")
  }
}
```

Listing 2.29: Example of a conditionally compiled module using the `cfg` attribute.

2.4.6. Cargo

Rust has a rich ecosystem of open-source libraries, accessible using its build tool and package manager, *Cargo*. As a build tool, Cargo can run the Rust compiler (*rustc*), run tests and benchmarks, collect lints and format code. Cargo also manages project dependencies, and can download external crates and treat them as input to the build. It integrates with *crates.io*[5], the Rust package registry, and can package and publish crates artifact with little setup on the part of the developer.

Cargo configuration Cargo is configured using a configuration file written in the *TOML* language [41]. TOML is a file format for configuration files, which is designed to be easy to read and write for humans. Like YAML and JSON, TOML defines a dictionary of configuration entries, which can themselves contain arbitrary values, including other nested dictionaries. Listing 2.30 shows an example Cargo configuration file for a crate named `my_crate`. The file starts by defining metadata about the crate, like the crate name and current version. The `[package]` header defines a TOML *table*, which is a dictionary of key-value pairs. Here, its keys are `name`, `version` and `authors`. The `[dependencies.log]` table describes that the crate has a dependency on the `log` crate, Rust's standard logging façade. The version string provided uses *semantic versioning* [40], and is therefore interpreted as a range of allowed versions. Here, the crate's `log` dependency must have a version between 0.4.0 (inclusive) and 0.5.0 (exclusive).

[5] `https://crates.io`

```toml
[package]
name = "my_crate"
version = "0.1.0"
authors = ["Clément Fournier"]

[dependencies.log]
version = "0.4"
```

Listing 2.30: Example TOML configuration file for Cargo.

Conditional compilation and Cargo Cargo defines a number of conditional compilation variables that can be used with the `cfg` attribute (cf. Paragraph 2.4.5§Attributes). For instance, code that should only be used in tests can be annotated with *`#[cfg(test)]`*. Cargo only sets the `test` variable when running test code, so the annotated items are not part of the released binary.

Cargo also supports crate-level *features*, which are conditional compilation flags usable by dependent crates. A feature can be enabled when specifying a dependency on the crate. For instance, Listing 2.31.a defines a Cargo feature named `cli` (line 9), presumably, to provide an optional command-line interface (CLI) for the crate. The feature references an optional dependency on a crate named `clap`. Unless the feature is enabled, this dependency is excluded from the build. This makes `my_crate` lighter for those users who depend on it, but do not need a CLI. The code of `my_crate` can also use the conditional compilation attribute *`#[cfg(feature="cli")]`* to compile code only if this feature is enabled.

```toml
1 [package]
2 name = "my_crate"
3
4 [dependencies.clap]
5 version = "1.3.0"
6 optional = true
7
8 [features]
9 cli=["clap"]
```

(2.31.a) Definition of a Cargo feature named `cli`, in the `my_crate` crate.

```toml
[dependencies.my_crate]
version = "0.1.0"
features=["cli"]
```

(2.31.b) Usage of the feature from a dependent crate.

Listing 2.31: Example definition and use of a Cargo feature.

3. Runtime design

Motivation Existing LF targets all rely on a code generator that translates LF code constructs into target code. The generated target program binds to a runtime library written in the target language. The runtime library factors out logic common to all reactor programs, in particular, the reaction scheduling logic. This design simplifies the code generator, compared to generating a completely standalone program. It allows the runtime library to be developed and tested independently of the compiler codebase. The runtime library can even be made reusable without LF, for instance, the C++ runtime can be used to write reactor programs in pure C++.

Design goals The design of the runtime library and that of the generated code is naturally co-dependent. To guide this design, the following goals have been identified:

— The Rust target should be able to integrate within the Rust ecosystem, for instance, it should allow the use of Rust libraries in target code.

— The target should reuse existing tools and libraries where possible. For instance, there are Rust libraries that provide abstractions that ease parallel programming, and the Rust runtime should be able to rely on those libraries where sensible.

— The coupling between the runtime and the code generator should be minimized. Loose coupling here ensures that implementation details of the runtime can be changed without having to update the code generator.

To allow a smooth integration with the Rust ecosystem, both the runtime and the generated code rely on Cargo, the Rust packaging and build tool (cf. Section 2.4.6). The runtime is packaged as a standalone library crate, which the generated code can bind to using Cargo's dependency resolution mechanisms.

To minimize coupling between the runtime and the code generator, the runtime is architected as a *framework*, i.e., a library that uses *inversion of control* to call into the generated code. The generated code binds to this framework using well-defined extension points, defined by a number of traits published by the runtime crate. Implementing these traits is declarative in nature: the generated code implements the API without having to know the details of how it is used by the runtime. The exact workings of the runtime are hence isolated from the generated code. The relatively low number of public extension points makes it easier to provide API stability guarantees, which the code generator relies on.

The following main components of the runtime have been identified:

(1) an API to model reactor programs, to be used by the generated code;

(2) infrastructure to execute the program using the constructed model;

(3) an API for reactions to interact with the execution infrastructure and the components of the reactor program at runtime, from within user-written reaction code.

Items (1) couples the runtime library to the code generator, and therefore benefits from having a stable, minimal surface. Item (2), the execution infrastructure, which is also called *the scheduler*, is mostly internal to the runtime crate. A public entry point acts as a single point of coupling between this component and the generated code (which has to call it in its `main` function). Item (3) is an API that couples the runtime crate to all reactor programs written in LF. Since making incompatible changes to this API could require updating the code of reactions in many reactor programs, the stability of this API is essential. It therefore also benefits from having minimal surface.

3.1. Chapter outline

This chapter is concerned with the design of the runtime crate. It does not discuss the form of the generated code itself, which is instead presented in Chapter 4, Section 4.1. The code listings in this chapter are to be understood as very simplified, so as to discuss the implementation at a relatively high level of abstraction. Finer implementation details related to performance are discussed in Chapter 5.

The rest of this chapter is structured as follows.

Section 3.2 gives an overview of the runtime crate's main components and internal structure. Section 3.3 then presents the basic principles of how reactors are represented in generated code, and how the runtime interacts with them.

Section 3.4 presents a high-level overview of how executing a reactor program works, and what pieces are required to implement the scheduler, distinguishing synchronous and asynchronous behaviour. Section 3.5 presents a formal description of how the reactor model and dependency graphs can be used to derive an execution strategy for the synchronous part of the scheduler. Sections 3.6 through 3.11 present different aspects of the synchronous part of the scheduler. Section 3.12 presents how the scheduler handles asynchronous events. Section 3.13 shows how the scheduler can be instantiated from within the generated target program. This concludes the explanation of the workings of the scheduler.

Section 3.14 goes on to explain how the data structures required by the scheduler are produced during the assembly phase. Section 3.15 explains how the runtime crate makes use of Cargo conditional compilation features.

Section 3.16 compares the workings of the scheduler with an older runtime design, presented in [16]. Finally, Section 3.17 provides some insight about the main differences between the runtime presented here and existing LF runtimes.

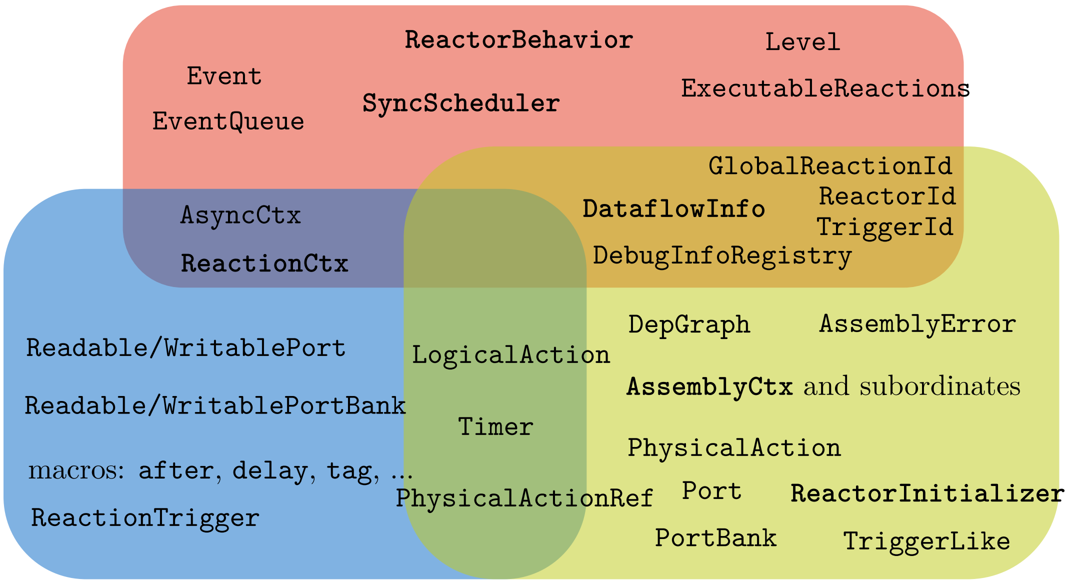


Figure 3.1.: Venn diagram that categorizes some APIs of the runtime crate into three different sets: red is the scheduler, blue is the API provided for code in reactions, yellow are types used to model the reactor program.

3.2. The runtime crate

The runtime crate was developed in lf-lang/reactor-rust[1] over the course of this work. It has been released with an MIT License. Figure 3.1 represents most of the types defined by the runtime crate and categorizes them according to the API they belong to.

The red surface represents the *scheduler*, which is responsible for driving the execution of the reactor program at runtime, by reacting to events and executing reactions. In this implementation, the scheduler is reified by an instance of `SyncScheduler`. Most of the contents of the red surface are internal types depended on by `SyncScheduler`, for instance, `Event` and `DataflowInfo`.

In blue, the reaction API enables interaction between the target code of reactions and the scheduler at runtime. It is centred around the `ReactionCtx` type, which mediates requests between reaction code and the scheduler internals (red). `AsyncCtx` also plays an important role in this interaction.

The yellow surface corresponds to the types used to model the reactor program. It contains types for reactor components, like `Port` or `Timer`. Some of those are also part of the reaction API (blue), while others are not communicated to reactions directly. For instance, `Port` is exposed to reactions as a `ReadablePort` or `WritablePort` instance, to limit the amount of interactions available to reactions. The model is built during initialization of the reactor program in a dedicated *assembly phase*, which revolves around the types `AssemblyCtx` and `ReactorInitializer`. The intersection between the yellow

[1]`https://github.com/lf-lang/reactor-rust`

and red surface contains types that are built during the assembly phase for the use of the scheduler, most importantly, `DataflowInfo` and `DebugInfoRegistry`.

3.3. Modelling reactors

Inspired by the existing C++ runtime (cf. Section 2.2.4), the Rust runtime models reactors using OOP concepts. This makes the code generator relatively simple, as reactor definitions are conceptually similar to a class in the OO sense: they define a general template for reactor instances. As introduced in Section 2.4.4, Rust's object system is based on *structs* implementing *traits*. The runtime hence bases its design on the principle that each reactor definition corresponds to a struct definition, which defines fields for reactor components, like actions, timers, and ports. The types of these components are structs exported by the reactor runtime.

To make custom reactor structs manipulable by the runtime, the runtime publishes traits that are implemented by the reactor struct.

```rust
pub trait ReactorBehavior {
    /// Id of this reactor.
    fn id(&self) -> ReactorId;
    /// Execute a reaction with the provided context.
    fn react(&mut self, ctx: &mut ReactionCtx, id: LocalReactionId);
    /// Perform cleanup actions to prepare for the next tag.
    fn cleanup_tag(&mut self, ctx: &CleanupCtx);
}
```

Listing 3.1: Declaration of the `ReactorBehavior` trait (most documentation elided).

The trait `ReactorBehavior` (cf. Listing 3.1) is the interface through which the runtime interacts with a reactor at runtime. The most relevant method is `react`, which executes a reaction on the reactor. The implementation is generated by the compiler to dispatch over the given identifier to the correct implementation method (cf. Section 4.1.3).

The trait `ReactorInitializer` (cf. Listing 3.2) is used to build reactors during the initialization of the application. The trait's only function is `assemble`, which uses an `AssemblyCtx` to produce a new instance of the self type. The other argument to the `assemble` function stands for an entire parameter list, which are the constructor arguments for the reactor. The type is associated with each particular implementation of the trait, using the associated type `Self::Params` (line 5).

Since the assembly is fallible, the result is wrapped within Rust's `Result` type, here hidden behind a type alias `AssemblyResult`.

The runtime use two separate traits for a practical reason: to implement the scheduling logic, it uses dynamic dispatch on the `react` method of `ReactorBehavior`. As explained in Section 2.4.4, this requires using a fat pointer `&dyn ReactorBehavior` to access reactor instances of different types under a common trait type. However, using a fat pointer requires the trait bound (here `ReactorBehavior`) to be *object-safe*, and

```rust
pub trait ReactorInitializer: ReactorBehavior {
    /// Type of the user struct
    type Wrapped;
    /// Type of the construction parameters
    type Params;
    /// Exclusive maximum value of the ‘id‘ parameter of ‘react‘.
    const MAX_REACTION_ID: LocalReactionId;

    /// Assemble the user reactor, ...
    fn assemble(args: Self::Params, assembler: AssemblyCtx<Self>)
      → AssemblyResult<FinishedReactor<Self>> where Self: Sized;
}
```

Listing 3.2: Declaration of the `ReactorInitializer` trait (most documentation elided).

object-safe traits cannot define associated constants, because that's unsound in general [19]. The solution to this problem is to factor out those associated members into the separate `ReactorInitializer` trait. This is appropriate because they are only used during initialization of the program, where we can use stronger typing with generics.

Both traits are hence used to interact with the same struct during distinct phases of the program's lifetime. While `ReactorInitializer` is used during *assembly*, the initialization phase, `ReactorBehavior` is used during the execution phase.

3.4. Overview of the execution logic

Executing a reactor program can be reduced to executing a certain set of reactions at the startup tag. Each reaction execution may trigger the execution of new reactions, either at the same tag (when a port is set), or at a future tag (when an action is scheduled). At a given tag, any reaction is executed at most once, and since the set of reactions in the program is finite, the processing of a tag terminates iff each executed reaction terminates. The program may also receive asynchronous events, which may trigger a number of reactions at a particular (future) tag. These events may be produced by threads executing concurrently to the scheduler; in the reactor model, they correspond to the scheduling of a physical action (cf. Section 2.2.1).

All events are processed in tag order.

Problem Structure From these requirements, we can identify two relatively independent sub-problems that are to be tackled by the scheduler instance:

(1) The processing of a single tag, which includes logic to let reactions trigger downstream reactions by setting a port, and to order reaction invocations so as to respect their dependencies;

(2) The maintenance of an *event queue*, that stores future events in an order determined by their tag, and supports the insertion of new events by asynchronous threads.

With the right data structures, the first sub-problem can be conceived as a simple function of a tag and an initial set of reactions, that produces side effects on the event queue and on the internal states of reactors.

The second sub-problem is about maintaining the current state of the execution, which in the implementation described here is encapsulated by an instance of the struct `SyncScheduler`. That instance lives as long as the program, and is what is usually referred to when the author writes "the scheduler".

3.5. Representing data dependencies

Processing a particular tag (problem (1) of Paragraph 3.4§Problem Structure) requires executing reactions in a particular order, that is derived from the data dependencies of reactions. This ordering provides for the determinism guarantee of Lingua Franca. A definition of that ordering based on a formalism of the reactor model follows.

3.5.1. Definition

Let us first introduce a couple of ancillary definitions.

Definition 1 (port binding)**.** A port d may be *bound* to a port $u \neq d$, which we write $u \to d$. This means that whenever u is present, d is present with the same value. We call u the *upstream* of d, and we say that d is *in the downstream* of u, as ports can be bound to only one upstream, but may be the upstream of many ports. Bound ports cannot be set by reactions. The relation $\to$ is called the *connection relation* on ports. Connections between ports must be acyclic, i.e., for any port p, there is no path in $\to$ from p to itself. The transitive, reflexive closure of $\to$ is called $\overset{*}{\to}$.

Definition 2 (container function)**.** Within a reactor program, the *container function* C maps every component, reaction and reactor to the reactor that contains it. C is undefined for the main reactor.

Definition 3 (priority relation)**.** Reactions of the reactor program are related by a partial order $\prec$, called the *priority relation.* Within a given reactor, any two reactions can be compared by $\prec$, i.e., for m, n such that $C(n) = C(m)$, either $n \prec m$ or $m \prec n$.

Definition 4 (immediate dependency)**.** A reaction n has an *immediate dependency* on a reaction $m \neq n$, written $m <_1 n$, iff either

(i) there exists ports $u \in \text{effects}(m)$, and $d \in \text{triggers}(n) \cup \text{reads}(n)$ such that $u \overset{*}{\to} d$, or

(ii) $C(m) = C(n)$ and $m \prec n$.

Intuitively, condition (i) means that n depends on m if m produces data that n observes. Through setting a port, m itself may even trigger n, and so m must always be executed before n. Condition (ii) is required because reactions share access to the state variables of their container reactor. To capture a deterministic ordering of accesses to that shared state, the reactor model requires that reactions that access shared state variables be totally ordered by $<_1$. This is the case here: if $C(n) = C(m)$ and condition (i) of Definition 4 does not apply, then either $m \prec n$ or $n \prec m$, so condition (ii) applies[2].

Definition 5 (dependency relation)**.** The transitive closure of $<_1$ is called $<$, the *dependency relation* of the program, and is a strict partial order on reactions. If $m < n$, then m should be executed before n at a given tag.

From this definition follows that if neither $n < m$ nor $m < n$, the computation of n and m are independent. When that is the case, those reactions may be executed in parallel for better performance.

3.5.2. Implementation

Since the scheduler is entirely part of the runtime crate, it is agnostic to the actual structure of the reactor program's dependencies. In other words, that structure is a *parameter* of the scheduler, and needs to be represented as a first class value — like a dependency graph. In fact, the relation-based definition of $<$ would be straightforward to implement using a dependency graph. However, ordering reactions using a graph at runtime would be inefficient, so the runtime uses a practical approximation of $<$, described in the following.

During initialization of the reactor program, the runtime builds a dependency graph, which represents reactions, ports, and actions as nodes, and their mutual dependencies as edges. Edges are oriented in the direction of data flow.

Each reaction is then labelled with its *level* in that graph:

Definition 6 (level)**.** Given a directed acyclic graph (DAG), the *level* of a vertex is the length of the longest path from any root of the graph to that vertex.

By construction of the graph, for any reactions n, m, $m < n$ implies $\text{level}(m) < \text{level}(n)$. By contraposition, two reactions that have the same level are independent and may be executed in parallel. Given any set of reactions, we can partition it by level, then execute each level one after the other, in ascending order, processing reactions of the same level in parallel.

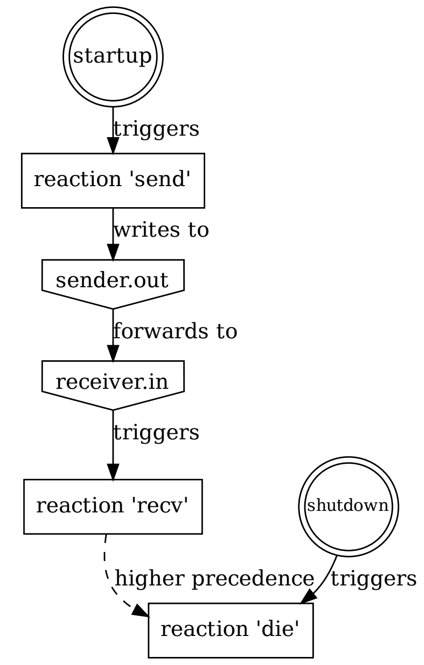
The entire scheduler module is built with this model in mind, as it provides a simple way to simultaneously respect reaction dependencies and exploit some[3] of the parallelism

[2] Lingua Franca relates *every* reaction of each reactor with each other by priority. This is stricter than necessary if the reactions do not access shared state, which the LF Rust code generator exploits when generating synthetic reactions to implement timers (cf. Section 4.1.5).

[3] It should be noted, that $\text{level}(m) < \text{level}(n)$ does not imply $m < n$. Specifically, it may be the case that $\text{level}(m) \neq \text{level}(n)$ and neither $m < n$ nor $m > n$, i.e., n and m are independent. Since we only parallelize when $\text{level}(m) = \text{level}(n)$, this represents some missed opportunities for parallelism.

```
target Rust;
reactor Sender {
  output out: u32;
  // @label send
  reaction(startup) → out {=...=}
}

reactor Receiver {
  input in: u32;

  // @label recv
  reaction(in) {=...=}
  // @label die
  reaction(shutdown) {=...=}
}

main reactor {
  sender = new Sender();
  receiver = new Receiver();

  sender.out → receiver.in;
}
```

(a) LF code of a simple program with two reactors communicating over connected ports.



(b) Corresponding dependency graph, as built internally by the runtime. Reactions closer to the roots (startup and shutdown) have a lower level, and should be executed first. Note that the level of “die” is 5, not 1.

Figure 3.2.: Example of a dependency graph for a simple LF program.

in the application. Sequential execution is a degenerate case of this model, where instead of parallelizing reactions that have the same level, the scheduler just executes them sequentially. Note that this implementation strategy is not original, but was inspired by the C++ runtime, and a former C implementation.

Figure 3.2 shows an example of a dependency graph for illustration. In this example, the reaction `send` produces some data that is consumed (through a port) by the reaction `recv`. Accordingly, in the graph, `recv` is reachable from `send` over the port nodes, and level(`send`) < level(`recv`), so `send` is always executed before `recv`. Note the presence of an edge between `recv` and `die`, which represents their relative priority explicitly.

3.5.3. Dependency graphs

For a given reactor program, the runtime defines a dependency graph $G = (V, E)$ as the following DAG:

— The vertex set V is the union of the set of reactions $\mathcal{N}$ and triggers $\mathcal{T}$ of the program. $\mathcal{T}$ contains all actions, timers, and ports of the program, and the special `startup` and `shutdown` triggers;

— The edge set E is the union of:

1. The connection relation on ports $\rightarrow$,
2. The priority relation on reactions $\prec$,
3. The set of effect edges $\{ (n, p) \mid n \in \mathcal{N},\ p \in \text{effects}(n) \wedge p \text{ is a port} \}$,
4. The set of trigger edges $\{ (c, n) \mid n \in \mathcal{N},\ c \in \text{triggers}(n) \}$,
5. The set of read edges $\{ (c, n) \mid n \in \mathcal{N},\ c \in \text{reads}(n) \}$.

Note that only effect edges for ports are included, and not actions or timers (item 3). This information is not needed by the runtime to order reactions, as the dependency relation on reactions only cares about ports. Actions and timers are still included in the graph, as are special `startup` and `shutdown` triggers, so that the set of reactions they trigger can be computed the same way as for ports. The graph as defined above must be acyclic, which is checked at runtime.

Note that edges are oriented forward, in the direction of effects, and of port connections. For that reason, the dependency graph is actually a dataflow graph.

In Section 3.5 we assumed that the execution strategy for parallel reactions is adequate, i.e., we can rely on level information in the dependency graph to approximate the dependency relation $<$ on reactions. We now prove that it is.

Let $G = (V, E)$ a dependency graph. We name E_* the edge set of the transitive, reflexive closure of G. For $a, b \in V$, if there exists a path from a to b in G (or $a = b$), then we write $(a, b) \in E_*$. Obviously, since $\rightarrow \subset E$, we have $\overset{*}{\rightarrow} \subset E_*$.

By definition of level (Definition 6), if $(a, b) \in E$, then $\text{level}(a) < \text{level}(b)$. By extension, if $(a, b) \in E_*$, then $\text{level}(a) \leq \text{level}(b)$.

Lemma 1. *For any two reactions $m, n \in V$, $m <_1 n$ implies $\text{level}(m) < \text{level}(n)$.*

Proof. Let $m, n \in V$ be reactions such that $m <_1 n$. Let us consider the two cases in the definition of $<_1$ (Definition 4) separately.

(i) There exists ports $u \in \text{effects}(m)$, $d \in \text{triggers}(n) \cup \text{reads}(n)$ such that $u \overset{*}{\rightarrow} d$.

Since $u \overset{*}{\rightarrow} d$, we know that $(u, d) \in E_*$, and hence $\text{level}(u) \leq \text{level}(d)$. By definition of E, we know that $(m, u) \in E$ (item 3) and $(d, n) \in E$ (items 4 or 5). Finally, we have $\text{level}(m) < \text{level}(u) \leq \text{level}(d) < \text{level}(n)$.

(ii) $C(m) = C(n)$, and $m \prec n$. Then, since $\prec \subset E$, $(m, n) \in E$ and $\text{level}(m) < \text{level}(n)$. □

The extension to $<$ is immediate:

Theorem 1. *For any two reactions $m, n \in V$,*

$$m < n \textit{ implies } \text{level}(m) < \text{level}(n).$$

$$\begin{tikzcd}
 & u_1 \arrow[dl, "*"'] \arrow[dr, "*"] & & u_2 \arrow[dl, "*"'] \arrow[dr, "*"] & \\
p & & s & & q
\end{tikzcd}$$

(a) Initial situation.

$$\begin{tikzcd}
u_1 \arrow[dd, "*"'] \arrow[dr, "*"] & & u_2 \arrow[dl, "*"'] \arrow[dd, "*"] \\
 & x_0 \arrow[d] & \\
p & s & q
\end{tikzcd}$$

(b) Placement of x_0 as the upstream of s.

Figure 3.3.: Visual representation of the connections in the proof of transitivity for $\overset{*}{\leftrightarrow}$.

Proof. Let $m, n \in V$ be reactions such that $m < n$. There must be an ascending chain in $<_1$ that starts with m and ends with n. By our lemma, each member of the chain has a greater level than the previous one, so, by transitivity, $\text{level}(m) < \text{level}(n)$. ☐

3.5.4. Port communication

Using $\rightarrow$, we can derive a relation $\overset{*}{\leftrightarrow}$ between ports: $p \overset{*}{\leftrightarrow} q$ iff there is a port u such that $u \overset{*}{\rightarrow} p$ and $u \overset{*}{\rightarrow} q$.

Theorem 2. *$\overset{*}{\leftrightarrow}$ is an equivalence relation on ports.*

Proof. The relation $\overset{*}{\leftrightarrow}$ is trivially symmetric. Reflexivity is also easy to show, since $p \overset{*}{\rightarrow} p$ for any p, which implies $p \overset{*}{\leftrightarrow} p$.

The proof of transitivity is less obvious. Let p, s, q be ports such that $p \overset{*}{\leftrightarrow} s$ and $s \overset{*}{\leftrightarrow} q$. There must be ports u_1, u_2 such that $u_1 \overset{*}{\rightarrow} p$, $u_1 \overset{*}{\rightarrow} s$, $u_2 \overset{*}{\rightarrow} s$, $u_2 \overset{*}{\rightarrow} q$ (cf. Figure 3.3a). Since a port may only be bound once, s has at most one upstream port. Either:

(i) s has no upstream port, in which case $s = u_1 = u_2$ and $p \overset{*}{\leftrightarrow} q$.

(ii) s has an upstream port x_0 (i.e., $x_0 \rightarrow s$), $u_1 \overset{*}{\rightarrow} x_0$, and $u_2 \overset{*}{\rightarrow} x_0$ (cf. Figure 3.3b). Since x_0 itself has at most one upstream port, we can apply the same reasoning repeatedly. Since $\rightarrow$ is finite, we must end up on the base case $u_1 = u_2$, and therefore $p \overset{*}{\leftrightarrow} q$. ☐

Within an equivalence class $\mathcal{C}$ for $\overset{*}{\leftrightarrow}$, all ports observe the same values at the same tags. Ports connected to each other by $\rightarrow$ are all in the same equivalence class for $\overset{*}{\leftrightarrow}$.

Theorem 3. *Within any non-empty equivalence class $\mathcal{C}$ for $\overset{*}{\leftrightarrow}$, there is exactly one port $w_{\mathcal{C}}$ that is not bound.*

Proof. By contradiction. Assume $\mathcal{C}$ contains $n \neq 1$ ports that are not bound.

(i) If $n = 0$, then all ports are bound. Let us assign a numeric label to every port in $\mathcal{C}$ as follows: let x_0 be a port in $\mathcal{C}$. For $i \geq 0$, the port x_i is bound, and has a unique upstream in $\mathcal{C}$, which is labelled x_{i+1}. There are only $|\mathcal{C}|$ distinct x_i, so, by the pigeonhole principle, there exists k, $0 \leq k < |\mathcal{C}|$ such that $x_{|\mathcal{C}|} = x_k$. The ports $\{x_i\}_{k \leq i < |\mathcal{C}|}$ form a cycle, which contradicts the definition of $\rightarrow$.

(ii) If $n > 1$, then there must be two distinct ports $p, q \in \mathcal{C}$ which are not bound. Since they have no upstream, there cannot exist a port u such that $u \overset{*}{\to} p$ and $u \overset{*}{\to} q$, which contradicts $p \overset{*}{\leftrightarrow} q$. $\square$

The port $w_\mathcal{C}$ is the only one that can be set by a reaction (i.e., be in its effects set), since bound ports cannot be set.

This hints that all ports of a given equivalence class $\mathcal{C}$ can share access to a common data cell. Since $w_\mathcal{C}$ is unique, and any reaction which effects that port precedes (in $<$) any reaction that depends on a port in $\mathcal{C} \setminus w_\mathcal{C}$ (the readers of the data cell), no two reactions will have concurrent read and write access to the data cell (even though many may share concurrent read-only access to it). Similarly, if several reactions effect $w_\mathcal{C}$, they must be in the same reactor —$C(C(w_\mathcal{C}))$ if $w_\mathcal{C}$ is an input, $C(w_\mathcal{C})$ otherwise—, and hence be related by $<$. In other words, no two reactions can share concurrent mutable access to the port.

This shows that the rules of the reactor model are strict enough to guarantee that sharing a single data cell between ports equivalent by $\overset{*}{\leftrightarrow}$ will not result in data races at runtime, and that usage of the cell will in practice respect Rust's borrowing rules. This conclusion is relevant for optimization of the runtime, as it provides a basis to elide locks and runtime checks that would be mandated by the Rust compiler otherwise (cf. Section 3.10.1).

The rules of the reactor model that have to be respected to yield the data race freedom property are summarized as follows:

— a bound port has a unique upstream (it can only be bound once)

— bound ports cannot be set by any reaction, they only receive values from their upstream

— an input i can only be set by reactions in $C(C(i))$

— an output o can only be set by reactions in $C(o)$

Those are verified at compile-time by the code generator, as they are part of the semantics of LF. They may also be checked at runtime by the reactor runtime, when building with assertions enabled.

3.5.5. Multiports

Multiports are a shorthand syntax to declare several ports at once. These ports are called the *channels* of the multiport. In the dependency graph, each channel of the multiport is represented as a separate independent port. An additional node is added to the graph for each multiport, that has an outgoing edge towards each channel node. Connecting two multiports generates an edge between each individual channel, and no edge between the multiport nodes themselves. Figure 3.4 shows how the graph looks like for the simple multiport connection created in the program of Listing 3.3.

The addition of multiports to dependency graphs does not change the connection relation $\rightarrow$. The only change required to the formalism of this section is to add a case to the definition of $<_1$ (Definition 4):

(iii) $m <_1 n$ if there exists a multiport u in effects(m), a port $d \in$ triggers$(n) \cup$reads(n), and a channel c of u such that $c \xrightarrow{*} d$.

It is easy to see how the proof of Lemma 1 can be adapted to accomodate this new case, thereby preserving the main result of Theorem 1.

```
reactor Node(size: usize(4)) {
  input[size] in: u32;
  output[size] out: u32;
  // @label writer
  reaction(startup) -> out {=...=}
  // @label reader
  reaction(in) {=...=}
}

main reactor {
  nodes = new Node(size=4);
  nodes.out -> nodes.in;
}
```

Listing 3.3: Simple LF program that binds two multiports together.

3.6. The `process_tag` routine

The method `process_tag` of the scheduler implements problem (1) of Paragraph 3.4§Problem Structure. This central piece of the scheduler executes reactions at a given tag, starting from an initial set of reactions scheduled to execute at that tag. Listing 3.4 shows an implementation of this method.

The set of remaining reactions to execute is named `todo` (line 6). For each level in increasing order, the outer loop retrieves the triggered reactions with that level. The inner loop, line 11, executes each of those reactions. This inner loop may be parallelized, which will be the topic of Section 3.10. On line 12, the reaction body is invoked, using the method `execute_reaction`. This will be the topic of Section 3.7. The object `reaction_ctx`, the *reaction context*, is used to accumulate events produced by the reaction execution. After execution, the newly triggered reactions for this tag are accumulated into the `todo` set (line 15). They will be handled in the next iterations of the outer loop, since by construction of the dependency graph, they necessarily have a strictly greater level than the current level. Finally, those reactions that are scheduled to trigger at a future tag are pushed into the *event queue* on line 16 (Section 3.12).

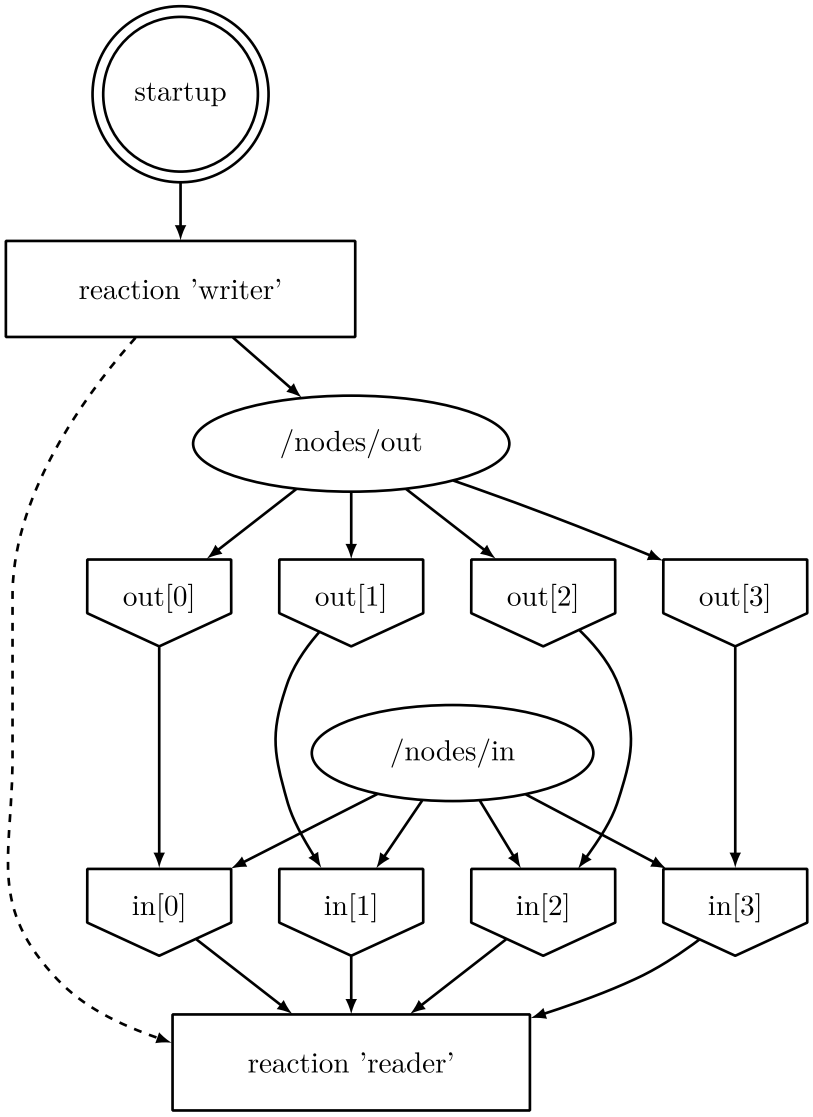


Figure 3.4.: Dependency graph for the reactor program of Listing 3.3.

```
1  fn process_tag(
2      &mut self, // SyncScheduler
3      tag: EventTag,
4      initial_reactions: ReactionSet,
5  ) {
6    let mut todo = initial_reactions;
7    let mut cur_level = 0;
8    while !todo.is_empty() {
9      let reactions_at_this_level = todo.remove_all(|r| level(r) == cur_level);
10     let mut reaction_ctx = self.new_ctx(tag);
11     for n in reactions_at_this_level {
12       self.execute_reaction(n, &mut reaction_ctx);
13     }
14     // executing reactions may have triggered new ones
15     todo = todo.union(reaction_ctx.get_triggered_reactions());
16     self.push_events(reaction_ctx.get_future_events());
17
18     cur_level += 1;
19   }
20 }
```

Listing 3.4: Pseudo-Rust code for the `process_tag` function, which processes a tag.

3.7. Representation of reactions

In Listing 3.4, the method `execute_reaction` (line 12) executes a reaction. The type of the parameter `n` is not specified, but it is a first-class object that represents the reaction to execute. To make `process_tag` as efficient as possible, we need a type for this reaction object that is cheap to copy, and to store in sets like `todo` (line 6). We also want a low-overhead strategy to implement `execute_reaction`.

In the C++ and TypeScript targets, each component and reaction is an object that contains lists of dependencies and anti-dependencies, which contain pointers to other components it depends on. In safe Rust though, such a pointer-heavy design is not impossible to implement, but not idiomatic, because it requires using smart pointers, or unsafe code. In previous work [16], the Rust runtime used to represent reactions with objects that are directly executable. This was a major strain on performance (cf. Section 3.16), so the current runtime uses a different approach.

In the current implementation, reactions are represented with simple symbolic identifiers (IDs). Reaction identifiers are as wide as a 32-bit integer[4], so they can be very efficiently copied and compared. Compared to the approach of C++ , or the previous Rust prototype, these identifiers do not give immediate access to the reaction they stand for. This design separates the problem of *where* the addressee is in memory, from the

[4]The runtime also supports using 64-bit integers, with a conditional compilation flag. For the sake of exposition we assume this feature is not used. Confer Section 3.15.

problem of having to symbolically refer to it and copy such references. It allows us to manipulate those references much more easily and flexibly compared to actual references ("pointers"), whose usage is constrained by the ownership system. Rust references also need to have a referent, but as we will see, in this framework, reifying reactions with unique objects is unnecessary.

To make reaction identifiers directly executable by the scheduler, we give them additional structure: an ID contains both an identifier for the container reactor, and one for the reaction within the given reactor (cf. Listing 3.5).

```rust
pub(crate) struct GlobalReactionId {
    container: ReactorId,
    local: LocalReactionId,
}
struct ReactorId(u16);
struct LocalReactionId(u16);
```

Listing 3.5: Simplified declaration of `GlobalReactionId`. Instances have the same layout as a single `u32`, which makes them very efficient to copy and hash.

Using these identifiers, we can now express that e.g., setting a port triggers a set of reactions, by using a set of `GlobalReactionId` to represent those reactions. At runtime, to execute a given reaction of this set, the runtime can map the `ReactorId` to the instance of the corresponding reactor, and make a virtual call to `ReactorBehavior::react` with the `LocalReactionId` as a parameter.

3.8. Layout of reactor instances

To implement the execution strategy for `GlobalReactionId` described in the previous section, the runtime needs to associate each `ReactorId` with its corresponding instance. A simple solution is to make the scheduler store all reactor instances within a vector. Each reactor's `ReactorId` is its index in the vector, so that it can be fetched in constant time. To allow storing reactors of different types within the same vector, we have to use a fat pointer[5] using the trait bound `ReactorBehavior` (presented in Section 3.3). These special references allow for dynamic dispatch on the `react` method. The vector is a field of the scheduler instance, so because of Rust's memory safety rules, reactors cannot contain references that live for a shorter time than the scheduler itself. In practice, this means they can only contain references with a static lifetime. This constraint applies to port values and state variables.

Listing 3.6 shows how this strategy is used to implement the `execute_reaction` function used on line 12 of Listing 3.4.

Storing reactors within a vector makes for a straightforward, constant time strategy to fetch reactors without using pointers or references. However, this vector structure is

[5]Confer Paragraph 2.4.4§Virtual dispatch.

```rust
fn execute_reaction(&mut self, n: GlobalReactionId, ctx: &mut ReactionCtx) {
  let GlobalReactionId { container, local } = n;
  // self.reactors: Vec<Box<dyn ReactorBehavior>>
  let reactor = &mut self.reactors[container];
  reactor.react(reaction_ctx, id);
}
```

Listing 3.6: Pseudo-Rust code that executes a single reaction in a given context.

also hard to change at runtime, since reactors need to be stored at the index of their reactor ID. This may cause difficulties to support mutations in the future (cf. 6.1.4).

3.9. The reaction context

The reaction context (type `ReactionCtx`) is used to mediate between the scheduler and the code of reactions. It provides an API for the reaction to manipulate and query components of the reactor program.

In C and Python, that API is provided by functions that have access to a global scheduler instance. In Rust, this is not idiomatic, and global mutable state anyway requires unsafe Rust code.

In C++ , all components are objects that can be directly manipulated by reactions. This works because components contain a reference to the scheduler, and may mutate it, for instance to trigger reactions when a port is set. In Rust, reactor components are also objects, but they pre-exist the scheduler, so they cannot be constructed with a reference to it. In fact, without using unsafe Rust (or a lot of smart pointers), they *cannot* contain a reference to the scheduler, since that would be duplicating a mutable reference.

For that reason, the Rust runtime treats reactor component objects as simple data types, and the `ReactionCtx`[6] centralizes the implementation of operations on them. `ReactionCtx` serves as a façade for the internals of the scheduler, and is the single coupling point between the user-written target code in LF and the internals of the scheduler.

3.9.1. API

The API of `ReactionCtx` is structured in the following way:

— Methods to query the current state of the logical subsystem: `get_current_tag`, `get_logical_time`, `is_shutdown`, etc.;

— Methods to query ports, actions and timers: `get`, `set`, `is_present`, etc.;

— Methods to schedule logical actions: `schedule`, `schedule_with_v`[7];

[6] `https://lf-lang.github.io/reactor-rust/reactor_rt/prelude/struct.ReactionCtx.html`
[7] For *schedule with a value.*

— A method to use asynchronous physical actions: `spawn_physical_thread`[8];

— A method to shutdown the application: `request_stop`.

Those methods act on reactor components that are injected in reactions (cf. Section 4.1.3), like `LogicalAction` and `Timer`. Importantly, while ports are implemented by a `Port<T>` struct, we never directly inject this struct into reactions. Instead, we wrap references to them within the types `ReadablePort` and `WritablePort`. This restricts the possible interactions of the reaction with the port: you can't write into a `ReadablePort`, nor read from a `WritablePort`. Simply using Rust's mutable and immutable reference types wouldn't achieve this goal, as a mutable reference may be both written to and read from. Note that this level of indirection is most likely completely eliminated by the Rust compiler.

Back in Figure 3.1, the components and API that are made available to reactions and can be manipulated with `ReactionCtx` are all in the blue surface. The reactor runtime crate exports them in a module called `prelude`[9], so that importing all this API can be done with a single import statement: `use reactor_rt::prelude::*`.

```
reaction(input_port) -> output_port {=
  // Rust
  if let Some(value) = ctx.get(input_port) {
    ctx.set(output_port, value);
  }
  // Rust (2), equivalent
  ctx.set_opt(output_port, ctx.get(input_port));
  // C++
  if (input_port.is_present()) {
    output_port.set(input_port.get());
  }
  // C
  if (input_port -> is_present) {
    SET(output_port, input_port -> value);
  }
=}
```

Listing 3.7: Comparison of the API to get and set port values in Rust, C++ and C. The `ctx` object is an injected `ReactionCtx` instance.

Listing 3.7 compares the API to get and set port values in Rust, C++ and C. The compared code samples have equivalent behaviour: if the value is not present, then no error occurs. In Rust, the fact that values may be present or not is made explicit by using an `Option` type. The `Option` type forces the user to pattern-match on the value in order to access it.

[8] `spawn_physical_thread` is the topic of Section 3.12.1.
[9] `https://lf-lang.github.io/reactor-rust/reactor_rt/prelude/index.html`

Note that `get`, its variants, and `is_present`, are generic and work uniformly for ports, actions and timers. There is an invariant, that for any `c`, `ctx.is_present(c)` returns true if and only if `ctx.get(c)` returns a non-empty `Option` instance. The method `is_present` is not very useful for ports, as pattern-matching on the result of `get` is a more ergonomic way to safely access the port value. However, timers, and many actions, do not carry values — in this case, their value type is set to `()`, the unit type. For these components, `is_present` is a useful shorthand. The generic API uniformly offers all those methods for all those components, as it allows reducing the number of methods of `ReactionCtx` (Rust does not support overloading).

Note that `set` and `schedule` are not generic, and only work on ports and actions, respectively.

3.9.2. Implementation

The high-level API of `ReactionCtx` hides a divide in the underlying implementation. When setting a port, not only needs the data cell of the port be updated, but reactions that declared a dependency on that port[10] need to be triggered at the current tag. Similarly, scheduling an action triggers the dependent reactions at a future tag. Those two types of interactions are handled somewhat differently.

Reactions that are triggered at the current tag are accumulated into a set, that is merged into the next reactions to execute (cf. Listing 3.4 line 15), and handled immediately. Instead, scheduling an action at a future tag causes the creation of an *event* that is pushed to a global event queue. An event is basically a tuple of tag and triggered reactions, and the event queue keeps these events sorted by tag.

An early implementation of `ReactionCtx` captured a mutable reference to the scheduler. While this allowed pushing events directly into the event queue, it falls short of supporting parallel execution, as several contexts might be alive simultaneously in different threads, and the event queue is not synchronized. At the time the runtime was updated to support parallel execution the `ReactionCtx` implementation was updated not to require a mutable borrow of the scheduler, and events are placed into a temporary queue local to the `ReactionCtx` before being pushed into the main event queue (cf. Listing 3.4 line 16).

`ReactionCtx` still borrows some of the fields of the scheduler immutably. This required careful placement and scoping of the different field borrows in the `process_tag` method. But these borrowed fields are vital to the functionality of `ReactionCtx`, as they include a crucial piece of information: the object instance that records, for each trigger component (port, action, etc), which reactions they trigger. As already mentioned, trigger components have been kept intentionally simple, and do not store themselves this set of reactions. This has a practical reason: if all information, for all trigger components, is stored in a single separate data structure, then we can *borrow* those sets of reactions from that data structure instead and avoid cloning them. The reasons why this is so will be explored in Section 5.1.2.

[10]in fact, on any port in the same equivalence class for $\overset{*}{\leftrightarrow}$.

3.10. Parallel execution

The principles behind parallel execution of reactions were exposed in Section 3.5. Translated into pseudo Rust-code, this means the inner loop of Listing 3.4 may be parallelized.

Rust has a large array of libraries that ease writing data-parallel code, like Crossbeam[11], easy-parallel[12], and crates from the async ecosystem like Tokio[13]. The Rust runtime uses Rayon[14]. This choice was mostly motivated by simplicity, as this library offers simple combinators to turn a sequential loop into a parallel one. Rayon also has simple APIs to implement thread pooling, which is a feature of the C and C++ runtimes.

```
1  // reactors: &mut Vec<Box<dyn ReactorBehavior>>
2  reactions_at_this_level
3    .iter()
4    .par_bridge()
5    .fold_with(ctx, |mut ctx, reaction_id| {
6      // execute the reaction
7      let GlobalReactionId { container, local } = n;
8      let reactor = &mut reactors[container];
9      reactor.react(ctx, id);
10
11     ctx
12   })
13   .fold(|| Default::default(), ReactionCtx::merge)
14   .reduce(|| Default::default(), ReactionCtx::merge)
```

Listing 3.8: Simplified example of how Rayon's combinators are used to parallelize the inner loop over the set of reactions at a given level (Listing 3.4 line 11). Rayon provides the method `par_bridge` as an extension to regular iterators. It returns an instance of the trait `ParallelIterator`, which provides the combinators seen below: `fold_with`, `fold`, `reduce`. In Rayon, fold operators are not terminal, which is why they have to be followed with `reduce`.

Listing 3.8 sketches how Rayon combinators are used to parallelize the inner loop of `process_tag` (cf. Listing 3.4). The `fold_with` combinator takes a reaction context instance (`ctx`), and a function of a context and reaction ID (here, a closure). Rayon clones [15] the context instance so that each worker thread gets its own instance, and executes the provided callback. The result of the `fold_with` call is another parallel iterator containing all reaction contexts that ended up being created and used. The remaining `fold` and `reduce` combinators, lines 13 and 14, merge these reaction contexts together in

[11]https://docs.rs/crossbeam

[12]https://docs.rs/easy-parallel

[13]https://docs.rs/tokio

[14]https://docs.rs/rayon

[15]Rayon relies on the `Clone` trait for this. If implemented directly by `ReactionCtx`, `Clone` could be used by reaction code to subvert the semantics of the reactor model. Instead, the newtype pattern is used to implement `Clone` on a separate wrapper type, not shown in the listing.

parallel, using the provided combination function `ReactionCtx::merge`. This produces a single context which contains the set of reactions and future events triggered by all the parallel reaction executions. All those combinators implicitly use Rayon's thread pool, which the reactor runtime configures at the start of execution. As an optimization, this logic is not executed when there is a single reaction to execute for a given level. In that case, the overhead introduced by Rayon cannot be compensated by parallelism, so the scheduler just executes the reaction inline.

Enhancing the runtime with parallel execution capabilities required a large refactoring of the internals of `ReactionCtx`, as explained in Section 3.9.2. The other large piece of work required that was required for this is related to Rust's type system. In Rust, the type system is used by the compiler to guarantee the absence of data races. The compiler automatically implements the `Send` and `Sync` marker traits on data types that are thread-safe. APIs that require thread safety, like Rayon's combinators, can then use `Send` or `Sync` as regular trait bounds on type parameters for user-provided data. As soon as one uses such an API in an existing sequential program, the compiler starts checking that the data types that are used are thread-safe. Importantly, the compiler does not consider internally mutable types to be thread-safe. Since instances of those types may be mutated through a shared reference, different threads that each have a reference may cause a data race. In the reactor runtime, internal mutability is crucial to the implementation of ports, so by extension, ports, and reactors that contain them, are not declared thread-safe by the compiler. The following section explains how this was solved.

3.10.1. Thread-safety of ports

Section 3.5.4 showed that constraints on the declaration of port connections and reaction dependencies give us the property that a port will not be used concurrently by reactions, unless all those concurrent accesses are read-only. Hence, the way we use ports is safe, and cannot exhibit a data race.

This conclusion was drawn using knowledge about the structure of $\rightarrow$, namely, knowledge of the constraints on $\rightarrow$ imposed by the reactor model (cf. Section 3.5.4). This information is lost to the Rust compiler's borrow checker. To prove to the compiler that ports are not causing data races (which is undefined behaviour in Rust), we need to use unsafe code. While there is no way around it, Rust's standard library allows using safe smart pointers which encapsulate that unsafe code. The smart pointer `RefCell` checks at runtime that Rust's borrowing rules are respected: at any point in time, either exactly one mutable borrow, or zero or more immutable borrows. Its implementation does not use atomic operations, so `RefCell` is not thread-safe. Using the thread-safe equivalent, `AtomicRefCell`[16], incurs a large runtime overhead (cf. Section 5.1.3). Since this performance penalty is incurred even though we know the program is safe, the Rust runtime uses unsafe code manually to avoid them.

[16]`https://docs.rs/atomic_refcell/`

3.10.2. Sharing reactors

The astute reader will probably realize that there is a problem with Listing 3.8. Line 8 borrows the reactor to execute the reaction. But this mutable borrow might happen multiple times concurrently, on the same `reactors` vector, which violates Rust's borrowing rules. Here, there is no way around unsafe code: we have to use pointers instead of references, and explicitly tag this pointer as `Send` (using a newtype). There is no data race occurring, because the reactions within the set `reactions_at_this_level` are all from different reactors. This is guaranteed by the construction rules of the dependency graph: if two reactions are in the same reactor, then an edge connects them in the dependency graph to represent their priority relation. Hence, no two reactions of the same reactor have the same level.

Note that here, there is truly no way to write this in purely safe code using e.g. a mutex or other standard smart pointers[17]. The root of the problem is not thread-safety proper, it is that all reactors are stored within the same vector, and the compiler cannot prove that distinct indices are accessed.[18]

3.11. Tag cleanup

LF semantics specify that ports and actions may be either *present* or *absent* at a given tag. At the start of every tag, all ports must be absent, and they only become present if they are set by some reaction executing at that tag. Similarly, actions are only present at a given tag if they have been scheduled for that tag.

In the current implementation, port and action values are contained in the port or action instances. This means that these instances need to be emptied of their value after processing a tag, to make sure that their value is not observable in future tags. This is implemented by a virtual call to the `cleanup_tag` function of `ReactorBehavior` (cf. Listing 3.1, line 7). As a simple implementation strategy, this method is called on all reactors of the application at the end of the `process_tag` function, which might be a performance bottleneck for some programs. Improving this implementation is left for future work (cf. Section 6.1.5).

3.12. Events and asynchrony

While processing a tag by synchronously executing reactions, new reactions may be scheduled at a future time point, because of logical action. Using *physical* actions, it is even possible for asynchronous code to schedule reactions. In the runtime framework, these future triggerings are called *events*, and they are reified by the struct `Event`.

Listing 3.9 shows the declaration of `Event` and `EventTag`. An `Event` contains the tag at which it is to be executed, and a set of reactions to execute. When handling the

[17]Short of wrapping the reactor vector itself in a mutex, thereby making our loop entirely sequential.

[18]It is perhaps interesting to note that it is the only instance where using unsafe code in the runtime cannot be avoided.

event, these reactions will be fed to `process_tag` as the initial set of reactions for the tag. The event may also cause termination of the program after the tag is processed. Termination events are otherwise treated like regular events, so that it is possible for timers and actions to be triggered normally even on the shutdown tag. Termination events are produced by calls to `ReactionCtx::request_stop`[19].

```
struct Event<'x> {
    tag: EventTag,
    reactions: ReactionPlan<'x>,
    terminate: bool,
}
/// Represents a tag
struct EventTag {
    offset_from_t0: Duration,
    microstep: MicroStep,
}
```

Listing 3.9: Simplified declaration of `Event` and `EventTag`.

`EventTag` is represented as a time offset from the start time of the program. For much of the history of the runtime, `EventTag` was actually based on absolute time instants. While both are equivalent in terms of the ordering they define on tags, using relative times makes for more readable trace messages, and basing `EventTag` directly on relative times means we do not need the scheduler's initial time to perform relativization when pretty-printing tags. Another benefit of relative offsets is that tags can be created in a vacuum from a duration. This is leveraged by the `tag!`[20] macro, which allows creating a tag with the syntax `tag!(T0 + 20 ms)` for instance. This facility is used extensively in LFC's integration tests.

Events are processed in increasing order of their tag. Before it is time to process them, they are stored in an event queue, which stores them in ascending tag order. Implementation details are the focus of Section 5.1.4.3.

3.12.1. Asynchronous events

Events may be produced by asynchronous threads, which is useful to handle input coming from the external world (e.g. keyboard input, or a pressure reading from a probe). Such events are associated with a physical action, and have, as their tag, the current *physical* time at the point where the physical action was scheduled (plus a possible offset). This ensures that they are scheduled in the future (with respect to the latest tag processed by the logical subsystem, or currently being processed), since the current logical time is always lagging behind physical time.

[19] `https://lf-lang.github.io/reactor-rust/reactor_rt/prelude/struct.ReactionCtx.html#method.request_stop`
[20] `https://lf-lang.github.io/reactor-rust/reactor_rt/macro.tag.html`

Events produced by asynchronous threads need to be forwarded to the event queue of the scheduler. Since we can't just share a mutable reference to the event queue between threads, we instead use a channel abstraction, provided by the Crossbeam[21] library. Channels are a multi-producer single-consumer[22] abstraction, which (in our case) uses a shared unbounded buffer[23] to communicate. The receiver remains in the ownership of the scheduler, while the senders can be cloned and given out to new threads.

API Asynchronous threads that want to communicate with the scheduler cannot be allowed all interactions permitted to reactions by `ReactionCtx`. For instance, functions like `get_logical_time`, and functions that manipulate components like ports and actions should not be callable asynchronously, as they would be racing with the main scheduler thread. For this reason, the restricted API provided to emit asynchronous events is not provided by `ReactionCtx`, but by the struct `AsyncCtx`[24].

`AsyncCtx` only allows scheduling physical actions, as this is the way provided by the reactor model to produce events outside of synchronous reaction execution. Physical actions are scheduled using an offset from the current physical time, not the logical time. `AsyncCtx` communicates with the scheduler through the event channel. It owns a sender instance bound to the scheduler's receiver.

`AsyncCtx` instances are produced by reaction context instances using the method `spawn_physical_thread`[25]. The `AsyncCtx` instance captures immutable references to the internals of the scheduler. To ensure that these references do not become dangling if the scheduler is dropped and the asynchronous thread is still running, we use *scoped threads* (provided by Crossbeam). Scoped threads are bound to a scope, which is a lexical scope reified by a scope object. When the scope object is dropped, the thread is joined, i.e., the destructor of the scope blocks until the thread terminates. This ensures that we can use references within the thread, provided the references outlive that scope. The reactor runtime creates a scope on program startup that joins asynchronous threads when the scheduler shuts down, so that the references captures by threads to the scheduler are guaranteed to be valid. The lifetime of this global scope is symbolized by a lifetime parameter on the scheduler and on `ReactionCtx`. The Rust type system otherwise ensures that we do not capture references that do not outlive that scope.

Listing 3.10 shows an example usage of `spawn_physical_thread`. The only references that may be captured by the thread closure are those that outlive the thread scope (they are internals of `AsyncCtx`). Importantly, the thread cannot capture e.g. `ctx`, the reaction context, because the compiler can prove that the `ctx` reference injected into the reaction does not live long enough.

[21]`https://docs.rs/crossbeam`

[22]Crossbeam channels support multiple consumers, but in the reactor runtime, the receiver is unique.

[23]Using a bounded buffer would have advantages, cf. Paragraph 3.17§Concurrency.

[24]`https://lf-lang.github.io/reactor-rust/reactor_rt/prelude/struct.AsyncCtx.html`

[25]`https://lf-lang.github.io/reactor-rust/reactor_rt/prelude/struct.ReactionCtx.html#method.spawn_physical_thread`

```
physical action act: u32;

reaction(startup) -> act {=
  // clone to gain ownership
  let act = act.clone();
  ctx.spawn_physical_thread(move |link| {
    std::thread::sleep(Duration::from_millis(200));
    // This will send an event through the channel.
    // The event tag is the current physical time
    // at the point this statement is executed.
    link.schedule_physical_with_v(&act, Some(123), Asap).unwrap();
  });
=}
```

Listing 3.10: Example usage of `spawn_physical_thread` in LF.

3.12.2. Main event loop

The main event loop of the scheduler drains events from the `Receiver` regularly, and inserts them into the sorted event queue. To select which event to process, the scheduler picks the event with the earliest tag, and then waits until the current physical time matches the logical time of the tag.

Listing 3.11 shows the implementation of the event loop. It starts by processing the initial tag (line 2). Reactions that declared a dependency on `startup` are executed using the `process_tag` routine. On line 4, the program enters an unconditional loop using the **`loop`** keyword. On line 6, pending asynchronous events are flushed from the channel receiver. The macro `push_event` is used to insert those events into the event queue, which reorders them according to their tag. The call to `try_iter` does not block to wait for events, and the loop proceeds to line 10 if the channel is empty. On line 10, the call to `take_earliest` attempts to remove the earliest event from the event queue.

If the event queue is non-empty, the identifier `evt` is bound to the removed event, and we proceed into the body of the **`if`** statement. On line 11, the event is checked to be earlier than the timeout of the application, if there is one. If it is past the timeout, the event is ignored and the shutdown routine is called (line 13). This will execute reactions that depend on the `shutdown` trigger, using the timeout as the logical time of the shutdown tag. The program ends, because of the **`return`** statement. If the tag of the event is not past the timeout, on line 16, the call to `wait_until` will pause the current thread to wait for the tag of the event, in case the logical time of the event tag is greater than the current physical time. When this call returns, the current physical time is greater than the tag of the event, and it can be processed. Then, if it was a termination event (produced by a `request_stop` call), the shutdown tag is processed and the program ends (line 18). Otherwise, `process_tag` is called to execute the reactions triggered by the event (line 20). The loop then proceeds to its next iteration.

If the event queue is empty, `take_earliest` will return `None` (line 10). In this case, an

```
fn launch_event_loop(mut self) {
  self.startup();

  loop {
    // flush pending events, this does not block
    for evt in self.rx.try_iter() {
      push_event!(self, evt);
    }

    if let Some(evt) = self.event_queue.take_earliest() {
      if let Some(timeout) = self.timeout_tag {
        if timeout < evt.tag {
          return self.shutdown(timeout, None);
        }
      }
      self.wait_until(evt.tag); // may sleep, async wake up code elided
      if evt.terminate || self.shutdown_time == Some(evt.tag) {
        return self.shutdown(evt.tag, evt.reactions);
      }
      self.process_tag(evt.tag, evt.reactions);
    } else if let Some(evt) = self.receive_event() { // this call may block
      push_event!(self, evt);
    } else {
      // all senders have hung up
      return self.shutdown(EventTag::now(), None);
    }
  }
}
```

Listing 3.11: Outline of the main event loop of the scheduler.

attempt is made to wait for an asynchronous event by blocking on the channel receiver (line 21). If this succeeds, the new event is pushed into the event queue, and the loop proceeds to its next iteration, where the event will be processed. Otherwise, this means that the channel returned a `RecvError`, because it is disconnected. This happens if no live asynchronous thread holds a reference to a sender, which means that the event queue will remain empty forever. In this case, the shutdown routine is executed, and the program ends (line 25).

3.12.3. Timeline synchronization

The call to `wait_until` on line 16 (Listing 3.11) lets physical time catch up with logical time, to resynchronize the two timelines.

The wait is implemented by blocking on the channel receiver with a timeout. The precision of the response time is unknown as of yet, and future experiments should be dedicated to evaluating it. However, compared to using a better specified wait function like shuteye[26], this offers a very significant advantage: the scheduler wakes up as soon as an asynchronous event is produced. A naïve implementation with e.g. `thread::sleep` would indeed have the scheduler ignore any new events while it sleeps.While the reactor model specifically allows logical time to lag behind physical time, this is to be avoided, as timely event handling might be safety-critical.

Implementing the same behaviour with `shuteye` or `thread::sleep` would require implementing a busy wait with exponential backoff. While Rust libraries like Crossbeam[27] provide facilities to ease this, a prior investigation to determine the precision of the channel's timeout is required.

3.12.4. Unbounded waiting and `keepalive`

An empty event queue does not necessarily mean the program should be terminated: if asynchronous threads exist, they may still push events to the scheduler. Other LF targets have no way to check that this condition holds, so they rely on configuration to determine whether the event loop should terminate, or wait. This is the purpose of the `keepalive` target property: if true, the program will wait for asynchronous events when the event queue is empty, possibly indefinitely (modulo timeout).

In Rust though, a better solution is possible. We know that asynchronous threads can only produce events through `Sender` instances that the scheduler gives out. Channels also have a built-in disconnection mechanism: if no sender exists, the channel is considered *disconnected*, which means that any attempt to receive a message will result in a `RecvError` or equivalent (without blocking). If the event queue is empty, the scheduler could just try waiting for a message, and exit the program if it receives a `RecvError`.

This sounds like it should work out of the box. However, this overlooks an important aspect of the channel API: the only way to create a new sender is by cloning an existing

[26]`https://crates.io/crates/shuteye`
[27]`https://crates.io/crates/crossbeam-utils`

one. If the sender count drops to zero, no more senders can be created. In both std[28] and Crossbeam channels, this makes the transition to a disconnected state irreversible.

Since the scheduler has to be able to produce a new sender whenever a reaction asks for it, it has to keep a sender alive all the time, and the channel never gets disconnected. This is solved by patching Crossbeam's channel implementation to make the `Receiver` instances able to produce new senders at will. This was a complex change, both from a purely technical standpoint (channel internals are complex, low-level unsafe code), and also to create a safe Rust API for this new feature of `Receiver`. The problem with the API is namely, that `Sender` and `Receiver` offer a unified façade over different channel implementations, and not all of them should be able to produce new senders at will.

The patch is currently waiting for a review[29] to be included in the upstream Crossbeam repository.

3.13. Entry point for execution

The members of `SyncScheduler` are nearly entirely private: the type only exposes a public entry point as an associated function. An example usage of that entry point is shown in Listing 3.12. The type `SchedulerOptions` (line 2) configures the runtime itself, while the other argument (built on line 6) are construction parameters for the main reactor type. The call to `run_main` (line 8) builds an instance of the main reactor, which recursively assembles the entire reactor tree for the program. `run_main` then transforms the built dataflow graph into a `DataflowInfo` instance, builds a scheduler instance with it, and launches the event loop shown in Listing 3.11.

```rust
1  fn main() {
2      let mut options = SchedulerOptions::default();
3      options.timeout = Some(delay!(15 ms));
4      options.threads = 8;
5
6      let main_args: MainReactorType::Params = ...;
7
8      SyncScheduler::run_main::<MainReactorType>(options, main_args);
9  }
```

Listing 3.12: Prototypical skeleton of a `main` function for a reactor program, whose main reactor has type `MainReactorType`.

[28] `https://doc.rust-lang.org/std/sync/mpsc/index.html`
[29] `https://github.com/crossbeam-rs/crossbeam/pull/750`

3.14. Assembly phase

The execution infrastructure for reactors requires runtime data structures that are derived from the structure of the reactor program. These include, for instance, a record of which reactions are triggered by which port, which is required to handle setting a port correctly.

This information is collected from a global dependency graph during initialization of the program, in a dedicated *assembly phase*. Building a dependency graph at runtime allows the runtime to verify the program structure, and to not have to trust the LF code generator completely. It also makes for a simple mapping from the reactor model concepts to an execution strategy, as explained in Section 3.5. Note however, that the dependency graph is thrown away before execution starts, as the scheduler only uses more efficient data structures to drive the rest of the execution.

3.14.1. The assembly module

Types related to assembly are mostly concentrated in the `assembly`[30] module. The contents of the assembly module, and other related types are graphed in the yellow surface of Figure 3.1 (page 35). The intersection with the scheduler (red) represents the data structures built during assembly, and used at runtime by the scheduler, the most important of which being `DataflowInfo`. The yellow surface contains types for reactor components, like `Port` or `Timer`. Some of them are also part of the reaction API (blue), while others are not communicated to reactions directly (like `Port`).

The most important public types in the assembly module are `AssemblyCtx` and `ReactorInitializer`, whose purpose and interaction have been cursorily presented in Section 3.3. Each reactor has to implement the trait `ReactorInitializer` (cf. Listing 3.2), whose `assemble` function creates an instance of `Self` using an `AssemblyCtx` parameter. The latter provides a declarative API to create new reactor components and declare their dependencies. That API encapsulates all the details of the graph construction, so as to limit the coupling of `ReactorInitializer` to the internals of the assembly module.

Reactor components are owned by the struct that implements `ReactorInitializer`, and we cannot place references to them in the graph. Instead, the graph use symbolic identifiers to refer to components: reactions are identified by a `GlobalReactionId`, while triggers use the type `TriggerId`.

`TriggerId` is a newtype over `u32` (`u64`, with a compilation flag). Trigger components implement `TriggerLike`, which allows the assembler to access their ID generically. The relevant declarations are reproduced in Listing 3.13.

Size constraints on reactor programs The structure of identifiers for reactions (Listing 3.5) and components (Listing 3.13) poses clear upper limits on the size of reactor programs:

[30] `https://lf-lang.github.io/reactor-rust/reactor_rt/assembly/index.html`

```rust
struct TriggerId(u32);
trait TriggerLike {
  fn get_id(&self) → TriggerId;
}
```

Listing 3.13: Simplified declarations of `TriggerId` and `TriggerLike`.

— The maximum number of reactors is 2^{16} (range of `ReactorId`)

— The maximum number of reactions per reactor is 2^{16} (range of `LocalReactionId`)

— The maximum number of total components in the program is $2^{32} - 2$ (range of `TriggerId` minus 2 reserved values for `shutdown` and `startup`)

With the appropriate conditional compilation flag (cf. Section 3.15), the runtime crate may also be built with 64-bit wide `GlobalReactionId` and `TriggerId`. In this case, the range of `ReactorId` and of `LocalReactionId` are both expanded to 2^{32} distinct values, while the range of `TriggerId` is expanded to $2^{64} - 2$.

3.14.2. Dependency graph implementation

The runtime uses a graph library called petgraph[31] to represent the dependency graph. Petgraph's `DiGraph` API assigns an opaque *index* to every node when it is first recorded in the graph. Operations on a digraph, e.g. recording an edge, always use these indices instead of the node itself. This scheme allows petgraph to simplify some graph algorithms, and also allows mutating the graph while traversing it (which would not be possible if the indices were references to the internals of the graph).

Since the reactor runtime already has its own specialized implementations of `TriggerId` and `GlobalReactionId`, the runtime implements a wrapper over a petgraph `DiGraph` called `DepGraph`. The dependency on petgraph and its specific index types is confined to the internals of `DepGraph` as an implementation detail.

`DepGraph` uses `TriggerId` and `GlobalReactionId` to represent triggers and reactions, not the reactor components themselves, which are owned by their enclosing reactor instance. This should be kept in mind when reading the mathematical formalism of this section, as it equates IDs and their referent.

3.14.3. Uses of dependency graphs

At runtime, the only information that the reaction context requires about the structure of the program is the set triggered(t) of reactions triggered by each trigger component $t \in \mathcal{T}$. This is necessary to implement functions like `ReactionCtx::set` (for ports) and `ReactionCtx::schedule` (for actions). When a port t is triggered through `set`, triggered(t) is merged into the set of reactions that are to be executed next at the same

[31] `https://docs.rs/petgraph/`

tag. When an action or timer t is triggered through `schedule`, triggered(t) is used to build an `Event` that is to be pushed to the event queue.

In terms of the reactor model, triggered(t) is defined as follows:

(i) If t is a port, then triggered(t) is the set of reactions which declare a trigger dependency on a port p such that $t \overset{*}{\leftrightarrow} p$.

(ii) Otherwise, t is an action, a timer, `startup`, or `shutdown`. triggered(t) is the set of reactions that have directly declared a trigger dependency on t.

In terms of dependency graphs, triggered(t) is the set of reactions reachable from t in one step, or if t is a port, the reactions reachable in one step from any port reachable from t through edges contributed by $\rightarrow$ (which are the only edges that connect ports together)[32].

Hence, a simple graph exploration suffices to compute a map data structure that corresponds to $t \in \mathcal{T} \mapsto$ triggered(t).

`DataflowInfo` The data structure that encapsulates the map described above is called `DataflowInfo`. In order to implement the `process_tag` function as shown in Listing 3.4, we would also need this data structure to contain level information for reactions, as in that listing, level is called on line 9 to filter reactions. This could be accomplished by saving e.g. a `HashMap<GlobalReactionId, u32>`. The actual runtime implementation does not need to do that explicitly, as reaction sets are actually pre-partitioned by level during the construction of `DataflowInfo`. This optimization and the way it rewrites `process_tag` are described in Section 5.1.1.

3.14.4. Debug information

Apart from building the dependency graph, the assembly phase also collects debug information for all reactions, reactors, and components into a specialized data structure, the `DebugInfoRegistry`. Apart from preserving identifiers of the LF source, it also contains compact data structures that preserve the containment relationship of reactor components. This information is used to perform validity checks at runtime, for instance, to check that the assumptions detailed in Section 3.5.4 are respected.

The API of `DebugInfoRegistry` is reproduced in Listing 3.14. The struct is internal to the scheduler, and is used to format trace messages, when logging is enabled. Keeping debug information outside of reactor components themselves makes the component objects lighter, and centralizes this concern.

[32] Note that this algorithm gives incorrect results for bound ports, since we only explore the downstream of t, not the entire equivalence class for $\overset{*}{\leftrightarrow}$. However, bound ports are never set explicitly, so we do not need to know the reactions they trigger and ignore them.

```rust
impl DebugInfoRegistry {
  fn get_debug_info(&self, id: ReactorId) → &ReactorDebugInfo;
  fn fmt_reaction<'a>(&'a self, id: GlobalReactionId) → impl Display + 'a;
  fn fmt_component<'a>(&'a self, id: TriggerId) → impl Display + 'a;

  fn get_container(&self, id: ReactorId) → Option<ReactorId>;
  fn get_trigger_container(&self, id: TriggerId) → Option<ReactorId>;
}
```

Listing 3.14: Selected members of `DebugInfoRegistry`.

3.15. Cargo integration

The runtime integrates with Cargo, the main advantage being easy access to a large number of Rust libraries. It also provides ancillary features, like documentation generation, and conditional compilation. Publicly supported conditional compilation flags[33] include:

parallel-runtime Unless this is provided, the runtime is built without parallel execution support. In which case, the Rayon dependency is not compiled at all. This reduces compile times and binary size for programs which do not exhibit a lot of concurrency, or platforms that do not support threading.

wide-ids This feature widens `GlobalReactionId` and `TriggerId` to be 64 bits wide, on 64-bit platforms. This allows the reactor runtime to address a greater number of reactors (2^{32} compared to 2^{16}), and may be useful for programs with very large numbers of reactors. Using this feature comes with a slight performance degradation, so the default ID size is 32 bits, even on 64-bit platforms.

no-unsafe Changes the implementation of ports to use check at runtime that Rust's borrowing rules are respected. The default implementation uses unsafe code, as explained in Section 3.10.1.

The runtime also defines conditional compilation features for internal use, for instance, to make internal implementation details of the crate available to benchmarking code.

3.16. Comparison with the prototype

3.16.1. The single `Reactor` trait

The design presented in [16] uses a single `Reactor` trait. The most important difference between this trait and the current `ReactorBehavior` is that `Reactor` uses strongly typed reaction identifiers, by using an associated type as shown in Listing 3.15. This provides additional type safety: it makes the `react` function total on the type of the reaction

[33]Cargo features, cf. Section 2.4.6.

ID, and makes reaction IDs of different reactors incomparable and unconvertible to each other. On the other hand, this also makes it impossible to define an object-safe version of the trait: reaction IDs of different reactors might have different sizes, and hence we can't use dynamic dispatch on that method. Instead, to execute reactions generically, a closure is created, that captures both a reference to the reactor and a copy of the (strongly typed) reaction identifier. The closure is then callable as a function of nothing more than the `ReactionCtx` parameter. Dynamic dispatch is used on the closure type, rather than on the reactor type.

This has several unintended drawbacks. The first is that each closure needs its own mutable reference to the reactor instance and its state. Of course Rust's ownership rules do not allow several simultaneous mutable borrows, so the reactor reference has to be wrapped within several layers of smart pointers to be shared. Similarly, since Rust checks thread-safety at compile-time, one layer of this stack of smart pointers has to be a `Mutex`, to guarantee the absence of data race for the reactor state. This mutex is acquired on every reaction execution, which in itself is a very significant performance bottleneck.

Another significant drawback of the previous design is that the closure objects that represent reactions are very expensive to clone, as to be thread-safe, the reference to the reactor needs atomic reference-counting operations. However, the scheduler needs to be able to represent e.g. a set of reactions to schedule, and in that previous design, these sets actually contain a clone of the closure. Due to otherwise suboptimal design, the amount of replication is large, which makes this another significant performance bottleneck. In the runtime presented in this thesis, `LocalReactionId` is on the contrary, very cheap to copy (cf. Section 3.7).

```rust
pub trait Reactor {
    type ReactionId;
    // Execute a reaction with the given ID
    fn react(..., reaction_id: Self::ReactionId, ...);
}
```

Listing 3.15: Declaration of the `Reactor` trait in an early runtime prototype [16]. At that time reactor instances were still assembled manually so there are no initialization-related methods on the trait.

3.16.2. Debug information

Another inefficiency of the older prototype is that, for the sake of debugging, it makes reactor identifiers carry information about all enclosing reactors. This makes reactor IDs larger and more expensive to copy.

Listing 3.16 shows the declaration of those IDs in the older runtime. That runtime uses `GlobalId` to reference both triggers and reactions, while the runtime presented in this thesis has distinct types, that have a different internal structure. `ReactorId` is

```
1  pub enum ReactorId {
2      Root;
3      Child {
4          // Rc = reference counting smart pointer
5          parent: Rc<ReactorId>,
6          name: &'static str
7      }
8  }
9  pub struct GlobalId {
10     container: ReactorId,
11     local: u32,
12     name: &'static str,
13 }
```

Listing 3.16: Declaration of `ReactorId` and `GlobalId` in the runtime prototype.

structured as a linked list of identifiers, using an `Rc` smart pointer (line 5). This way, every reactor ID knows about its container. The main benefit of that design is that these IDs can be pretty-printed easily. The reaction context can also check at runtime that some interactions are legal with respect to the declared dependencies of reactions.

These IDs need to be copied and passed around a lot, which is obviously very inefficient, not only because of the size of the IDs, but also because of the recursive cloning of the whole `Rc` chain.

The runtime described in this thesis uses more lightweight IDs, and the debug information is isolated into the `DebugInfoRegistry` (cf. Section 3.14.4). This means debug information does not get in the way of efficient copying of IDs.

3.17. Notable differences with other LF targets

Reactor internals The most important difference with other object-oriented target is that reactor instances are much simpler in Rust. They only contain their state, their own components and their ID, and different concerns are distributed across separate registries.

For instance, dependency information is centralized in the dependency graph. While the use of a dependency graph at runtime is similar to what the C++ target does, in Rust, the dependency graph is truly the only place where dependency information is registered. By contrast, in C++, each reactor component and reaction contains lists of dependencies and anti-dependencies, thereby making the object graph a dependency graph itself. This kind of design is not idiomatic in Rust, as the ownership model makes it extremely difficult to implement such self-referential, pointer-heavy data structures. The Rust implementation makes sure that reactors never contain dangling pointers to other components, as these pointers are not needed in the first place. Similarly, storing debug and topological information is offloaded to the `DebugInfoRegistry`, for use by the scheduler logging functions and for runtime assertions. The use of symbolic identifiers to refer to components and reactions and glue everything together is a unique feature of the Rust target.

References in states and ports That the state of reactors may not contain references, unless those references have the static lifetime. This is because reactor instances are owned by the scheduler, so the references contained by reactors must outlive the scheduler instance, which itself lives as long as the entire execution phase of the program. In practice, the static lifetime is nearly equivalent. Since port instances are owned by reactor instances, port values must also have the static lifetime.

This constraint is imposed by Rust's ownership model, and makes sure that reactors or ports never contain dangling pointers. It also enforces at compile-time that reactors cannot leak references to their internal state to other reactors, thereby possibly causing data races. The Rust runtime is currently the only LF runtime that can enforce this. Applications that cannot avoid sharing references, e.g., for performance reasons, can still use unsafe code to do so, or smart pointers.

Port implementation In the C++ runtime, ports have a pointer to their upstream, while in Rust, the data cell is actually a separate first-class object and no pointers to the ports themselves exist. In Rust, transitive port connections are completely reduced: all ports of a given equivalence class for $\overset{*}{\leftrightarrow}$ have the same pointer to the same data cell, and so the data is exactly one pointer indirection away. In C++, the number of data accesses is linear with respect to the length of the upstream chain (though these chains are typically very short).

Concurrency The Rust runtime delegates parallelization of reactions to the Rayon framework, which reduces the complexity of the implementation. By contrast, the C and C++ runtimes implement their own thread pooling logic.

The use of channels as a communication abstraction between asynchronous threads and the scheduler is also a specificity of the Rust runtime. As explained in Section 3.12.2, this removes the need to write custom waiting routines using `thread::sleep` or equivalent. It also makes the Rust target the only one to date not to need the `keepalive` target property. The channel implementation currently uses an unbounded buffer, which could cause event-producing threads to overwhelm the scheduler. In the C runtime, asynchronous threads have to compete for a mutex in order to schedule an event, giving the scheduler a natural mechanism for applying backpressure. In Rust, this could be achieved by changing the unbounded buffer to a bounded one, which would block event-producing threads in case the buffer is full.

The use of scoped threads to enable safe borrowing of stack-local data is also a specificity of the Rust target. C++ has libraries for scoped threads[34], so the C++ runtime could do that too in the future.

[34]`https://www.boost.org/doc/libs/1_76_0/doc/html/thread/ScopedThreads.html`

The types of port and action values are verified at compile-time to be thread-safe (using Rust's `Sync` trait). This ensures that executing reactions reading from the same port concurrently will not cause a race condition.

Safety The Rust type system helps to prevent data races in concurrent code, and to guarantee memory safety. It does so by forcing library authors to think thoroughly about the corner cases of their abstractions, and the actual safety conditions of their "unsafe" code. For this reason, the Rust runtime enjoys a high standard of safety, though it could be improved further by formal verification in future work (cf. Section 6.1.2).

Package management Since the runtime leverages Cargo, it has first-class integration within the Rust ecosystem. This allows the runtime to use features like conditional compilation, and documentation generation. The same can be said of the TypeScript and Python targets, but not so much the C and C++ targets.

4. The LF-Rust compiler

Aside from its runtime, the second piece of the LF Rust target is its code generator. The code generator generates a target Rust program that binds to the runtime library using the traits it publishes (cf. Section 3.3). The entry point of the target program is a `main` function that calls the entry point of the scheduler (cf. Section 3.13). This chapter describes the form of the generated Rust program in Section 4.1, and the code generator's architecture in Section 4.3.

4.1. Form of the generated code

This section describes the form of the generated code. The reader may refer to Section 3.3 for definitions of the traits that the generated code has to implement.

4.1.1. Project layout

To integrate with Cargo, the Rust code generator generates a Cargo project with a standard structure. Figure 4.1 shows the structure of the directory in which the target program is generated.

The top-level directory contains a generated `Cargo.toml` file, which is the configuration file for Cargo. The `rust-toolchain` file is also used as configuration. The `src` directory contains generated Rust sources, and the `target` directory contains build artifacts generated by Cargo.

Within the source directory, a `main.rs` file contains the entry point of the program (the `main` function). A subdirectory defines a module named `reactors`, which contains

```
/
├── Cargo.toml
├── rust-toolchain
├── src
│   ├── main.rs
│   └── reactors
│       ├── mod.rs
│       ├── reactor1.rs
│       └── ...
└── target
    └── ...
```

Figure 4.1.: Directory structure of a Rust target program.

a Rust file for each reactor class used in the LF program (in Figure 4.1, `reactor1.rs` is one of them). Each such reactor file declares a Rust module (cf. Section 2.4.4).

4.1.2. Reactor modules

Each reactor class corresponds to a Rust module in the generated program. This module contains a struct definition that implements the traits published by the runtime, `ReactorBehavior` and `ReactorInitializer` (cf. Section 3.3), and several other structs whose role is explained in this section and those that follow.

Listing 4.1 shows a simple LF reactor, and Listing 4.2 shows the outline of the corresponding reactor module. The reactor declaration generates a struct declaration containing the state variables of the reactor (line 4). This *state struct* has the same name as the LF reactor, as it is the only one with which user-written target code interacts directly (cf. Section 4.1.3). Reactor parameters are generated into a separate struct, here, `SimpleReactorParams` (line 13).

Another struct implements the traits of the reactor runtime, defined here on line 18. Since it implements the adapter design pattern, its name ends with `Adapter`. This struct contains an instance of the state struct in a field (line 20). It also contains fields for each reactor component defined by the reactor class, here, the port `out` (line 22).

The names of fields defined by the the code generator (like `__impl`) start with two underscores, which is an identifier prefix reserved by LF to avoid name collisions. Identifiers in LF may indeed not start with this prefix, which, e.g., prevents a port from being named `__impl`, thereby conflicting with other generated fields.

Parent module

The parent module for all reactor implementation modules is the `reactors` module, defined by the `mod.rs` file shown in Figure 4.1. For each reactor submodule, the parent module exports the type of the adapter struct and of the param struct, as shown in Listing 4.3. In this listing, the `mod` declaration links the sibling `simple_reactor.rs` file as a submodule of the `reactors` module. The exports that follow make it easier to refer to those types when reactors assemble children instances (cf. Section 4.1.4).

4.1.3. Reactions

The body of reactions may access state variables of the containing reactor. Reactions are declared as methods of the state struct, and can therefore access the state struct with the keyword `self`.

Listing 4.4 shows how the method for the reaction of `SimpleReactor` (cf. Listing 4.1) is declared, on line 9. This method is private (as it omits the `pub` modifier). It takes a `&mut self` parameter, for the state struct, a parameter for the reaction context, named `ctx`, and a parameter for the port it declared a dependency on (`out`). The body of the method is copy-pasted from the LF source file without further transformation. Notice

```
reactor SimpleReactor(param: u32(0)) {
  state x: u32(param);
  output out: u32;
  reaction(startup) -> out {=
    ctx.set(out, self.x);
  =}
}
```

Listing 4.1: LF code for a simple reactor.

```
use reactor_rt::prelude::*;

// state struct
pub struct SimpleReactor {
  x: u32,
}

impl SimpleReactor {
  // reaction implementations (elided)
}

// parameter struct
pub struct SimpleReactorParams {
  param: u32,
}

// adapter struct
pub struct SimpleReactorAdapter {
  __id: ReactorId,
  __impl: SimpleReactor,
  // component declarations
  pub out: Port<u32>,
}

impl ReactorInitializer for SimpleReactorAdapter {
  type Wrapped = SimpleReactor;
  type Params = SimpleReactorParams;
  // (elided)
}

impl ReactorBehavior for SimpleReactorAdapter {
  // (elided)
}
```

Listing 4.2: Outline of the reactor module defined for `SimpleReactor` (cf. Listing 4.1), in a `simple_reactor.rs` file. Reaction declaration elided.

```rust
mod simple_reactor;
pub use simple_reactor::SimpleReactorAdapter;
pub use simple_reactor::SimpleReactorParams;
```

Listing 4.3: Declarations in the parent module of `SimpleReactor` (cf. Listing 4).

```rust
 1 use reactor_rt::prelude::*;
 2 
 3 // state struct
 4 pub struct SimpleReactor {
 5   x: u32,
 6 }
 7 
 8 impl SimpleReactor {
 9   fn react_0(&mut self, ctx: &mut ReactionCtx, out: WritablePort<u32>) {
10     ctx.set(out, self.x);
11   }
12 }
```

Listing 4.4: Reaction declaration generated for the `SimpleReactor` of Listing 4.1.

the import statement on line 1, which brings useful types and macros of the reactor runtime crate in scope for reactions to use.

Dependencies the reaction declares in its signature are all injected as parameters of the method. For instance in Listing 4.4, the port `out` is injected as an instance of `WritablePort`, as the reaction declares an effects dependency on it. This type provides write-only access to the port (through the API of `ReactionCtx`). The type a dependency is injected with depends on the type of component and on the dependency kind, as shown in Table 4.1. As we can see in this table, the types of parameters injected for trigger and read dependencies are always shared references, so that Rust can prevent them from being mutated. Undeclared dependencies, and dependencies on `startup` or `shutdown`, do not create a parameter on the generated method.

Component	Trigger/read dependency	Effect dependency
Port of type `T`	`&ReadablePort<T>`	`WritablePort<T>`
Logical action of type `T`	`&LogicalAction<T>`	`&mut LogicalAction<T>`
Physical action of type `T`	`&PhysicalActionRef<T>`	`&mut PhysicalActionRef<T>`
Timer	`&Timer`	n.a.
Port bank of type `T`	`&ReadablePortBank<T>`	`WritablePortBank<T>`

Table 4.1.: Type of the parameter injected in a reaction as a function of component kind and dependency kind.

`ReactorBehavior` implementation The runtime invokes reactions through a virtual call to `ReactorBehavior::react` (cf. Section 3.7). The implementation of this method selects the correct reaction method using the `LocalReactorId` parameter. The trait `ReactorBehavior` is implemented by the adapter struct, not directly the state struct. Its implementation for `SimpleReactor` is reproduced in Listing 4.5.

`react` is implemented with a match statement, shown on line 14. Each branch corresponds to a separate reaction invocation. Here, there is a single reaction, whose ID is zero; the corresponding branch is defined on line 15. The right-hand side of the arrow calls the method `react_0` on the state struct instance. Here, the `out` field for the corresponding port is mutably borrowed, and this reference is wrapped in an instance of `WritablePort`. The last branch is taken if the local reaction ID does not correspond to a reaction of the reactor. It is defined to panic, as this is an error in the scheduling logic.

The implementation for the remaining methods of `ReactorBehavior` are also shown in Listing 4.5. The method `id` (line 9) just returns the reactor ID of the current reactor. This can be used by the scheduler to check at runtime that the reactor is correct. The method `cleanup_tag` clears ports and actions of the reactor, as explained in Section 3.11.

```rust
1  // adapter struct
2  pub struct SimpleReactorAdapter {
3    __id: ReactorId,
4    __impl: SimpleReactor,
5    pub out: Port<u32>,
6  }
7
8  impl ReactorBehavior for SimpleReactorAdapter {
9    fn id() -> ReactorId {
10     self.__id
11   }
12
13   fn react(&mut self, ctx: &mut ReactionCtx, rid: LocalReactionId) {
14     match rid.raw() {
15       0 => self.__impl.react_0(ctx, WritablePort::new(&mut self.out),),
16       unknown => {
17         panic!("Unknown reaction ID {}", unknown);
18       }
19     }
20   }
21
22   fn cleanup_tag(&mut self, ctx: &CleanupCtx) {
23     ctx.cleanup_port(&mut self.out);
24   }
25 }
```

Listing 4.5: Implementation of `ReactorBehavior` for the `SimpleReactor` of Listing 4.1.

4.1.4. Assembly

During assembly, the runtime uses the trait `ReactorInitializer` to create an instance of each reactor. The implementation of this trait is shown in Listing 4.6. On line 4, a constant is defined to represent the maximum reaction ID of the reaction.

The implementation of `assemble` follows, on line 6. This function uses an `AssemblyCtx` parameter to create a new instance of the reactor, including its components like ports and actions. It records the dependencies of all its reactions on these components using the API of the assembly context, which itself internally builds a dependency graph using this information.

The implementation starts by destructuring the parameter struct on line 9. This is used to bring individual parameters in scope, so that they are accessible in expressions for the constructor arguments of child reactors. Since the `SimpleReactor` does not feature children reactor instances, this is here somewhat pointless, but the following subsection will show an example usage.

The rest of the function is a call to `AssemblyCtx::assemble`, starting on line 10. This method takes a closure that needs to return an `AssemblyIntermediate` instance. Instances of this type can only be produced by two functions of the `AssemblyCtx`, one of which is `assemble_self`, which is called on line 11. The `assemble_self` method creates and initializes a new instance of the current reactor, using first a closure (line 12) that creates the reactor itself using the auxiliary function `user_assemble` (defined on line 30). This closure can use a `ComponentCreator` instance (the `cc` parameter here), and the `id` for the newly created reactor. The component creator is the only object that can create new reactor components, e.g., `Port` or `Timer`. When doing so, it registers them in the dependency graph that the `AssemblyCtx` encapsulates. The second closure used by `assemble_self`, on line 18, is used to register the dependencies of declarations. Its parameter `__assembler` is an instance of `DependencyDeclarator`, which has methods to declare dependencies, while building the dependency graph internally. The second parameter is called `__self`, a mutable reference to the reactor instance being created, which is used to access the actions and ports instances that are in its fields. The last parameter is an array of reaction IDs, created by `AssemblyCtx`. The body of the closure declares dependencies using the methods of the dependency declarator.

While this API may appear unintuitive and complicated compared to a more imperative style, it gives the `AssemblyCtx` a lot of freedom regarding the order in which it calls these closures and creates components like reaction IDs. The generated code is very declarative, which decouples the internals of the assembly context from the generated code. The API also uses types like `AssemblyIntermediate` and `FinishedReactor` to make sure that the implementation of `assemble` uses the API of the assembly context, instead of e.g. just returning an instance of `Self` without registering its components in the assembler. Various type-level assertions are performed here, for instance, that `MAX_REACTION_ID` is equal to the number of reactions created by `assemble_self`, and to the number of debug labels provided for reactions (line 16). It is hence difficult to write an incorrect implementation that compiles.

```rust
impl ReactorInitializer for SimpleReactorAdapter {
  type Wrapped = SimpleReactor;
  type Params = SimpleReactorParams;
  const MAX_REACTION_ID: LocalReactionId = LocalReactionId::new(1);

  fn assemble(__params: Self::Params, __ctx: AssemblyCtx<Self>)
     -> AssemblyResult<FinishedReactor<Self>> {

    let SimpleReactorParams { param, } = __params;
    __ctx.assemble(|__ctx|
      __ctx.assemble_self(
        |cc, id| Self::user_assemble(cc, id, SimpleReactorParams { param, }),
        // number of non-synthetic reactions
        1,
        // reaction debug labels
        [None],
        // dependency declarations
        |__assembler, __self, [react_0]| {
          // --- reaction(startup) -> out {= ...=}
          __assembler.declare_triggers(TriggerId::STARTUP, react_0)?;
          __assembler.effects_port(react_0, &__self.out)?;
          Ok(())
        }
      )
    )
  }
}

impl SimpleReactorAdapter {
  fn user_assemble(__assembler: &mut ComponentCreator<Self>,
                   __id: ReactorId,
                   __params: SimpleReactorParams) ->  AssemblyResult<Self> {
    let SimpleReactorParams { param, } = __params;
    let __impl = SimpleReactor { x: param, };

    Ok(Self {
        __id,
        __impl,
        out: __assembler.new_port::<u32>("out", PortKind::Output),
    })
  }
}
```

Listing 4.6: Implementation of `ReactorInitializer` for the `SimpleReactor` of Listing 4.1.

Creating children

The method `AssemblyCtx::with_child` is used to assemble children reactor instances. Listing 4.8 shows how this method is used within the assembly method of a `SimpleParent` reactor, defined in Listing 4.7. The method is called within the closure parameter of `assemble`, line 6. The type of the child is specified explicitly, as the assembler will use it to assemble the child recursively. The first parameter of the `with_child` call is the name of the child instance, which is stored in the debug information registry.

Next, the constructor arguments of the child are provided, using the child's `Params` associated type. These arguments may refer to parameters of the current instance, as they were destructured to bring them into scope (line 4). Notice that on line 8, the LF expression `{=param * 2=}` has been translated to a Rust block expression, `{param * 2}`. This allows using arbitrarily complex logic to create arguments, e.g., to introduce intermediary variables.

The last parameter, on line 9, is a closure that provides access to the assembly context, and the built child instance. The child instance (here, the `child` closure parameter) is only a mutable reference, as its ownership remains in control of the assembler. Since only the port fields are public (cf. Listing 4.2, line 22), the only thing the parent reactor can do with the child is bind its ports to its own.

```
reactor SimpleParent(param: u32(2)) {
  child = new SimpleReactor(param={=param * 2=});
}
```

Listing 4.7: LF definition of a reactor with a child reactor.

```
1  fn assemble(__params: Self::Params, __ctx: AssemblyCtx<Self>)
2     -> AssemblyResult<FinishedReactor<Self>> {
3
4    let SimpleParentParams { param, } = __params;
5    __ctx.assemble(|__ctx|
6      __ctx.with_child::<super::SimpleReactorAdapter, _>(
7        "child",
8        super::SimpleReactorParams::new({param * 2}),
9        |mut __ctx, child| {
10         __ctx.assemble_self(/* assemble self parameters */)
11       }
12     )
13   )
14 }
```

Listing 4.8: Example usage of `with_child` in the assemble implementation of the `ParentReactor` of Listing 4.7.

When several children instances are to be created, the `with_child` calls are nested within each other, each of them successively bringing a mutable reference to the new child into scope. Since `with_child` returns an `AssemblyIntermediate` result, and its closure parameter also needs to return an `AssemblyIntermediate`, these nested closures have to end with the `assemble_self` call.

Notice that the strict API of `AssemblyCtx` can only be used to access children of the current reactor. It is impossible to access children of the children, for instance. This is in line with the reactor model's scoping constraints, and adds a further layer of safety on top of the verification performed by the LF compiler ahead of time.

4.1.5. Code lowering

Some high-level features of LF are easier to implement using existing simpler features. This increases code reuse in the code generator and the runtime. In traditional compilers, this is usually implemented by *lowering* a high-level intermediate representation (IR) into a lower-level IR. While LFC does not do that, as it does not feature a traditional IR (instead working on ASTs), this section reuses the term *lowering* to describe how some of the high-level features of LF are implemented in terms of simpler features.

Timers

Timers have first-class support in the reactor runtime API, through the struct `Timer`. However, the reactor runtime does not do anything special with timers, except considering them as triggers during assembly. The periodic behaviour of timers is implemented by adding *synthetic reactions* to the generated Rust program. Each timer generates two synthetic reactions, a *bootstrap* reaction, which schedules the first release of the timer on startup, and a *periodic* reaction, which reschedules the timer each time it is triggered (if the timer is periodic). Listing 4.9 shows how those reactions would look like, were they to be written in LF.

These synthetic reactions are special in that they are not ordered by the priority relation ($\prec$) with respect to other reactions in the same reactor. This is required because priority edges may cause causality loops, and it is safe since the synthetic reactions never access state of the current reactor.

Port references

Reactions may reference ports of child reactors in their signature. Since child reactors are not contained by their parent (all reactors are flattened in the reactor vector, cf. Section 3.8), the port of the child is not accessible at runtime from within `ReactorBehavior::react`. To circumvent this, a port reference is actually code generated as a port of the current reactor that is connected to the port of the child. Listing 4.10 compares the LF source code to how the Rust code generator understands it. Note that the synthetic ports are neither inputs nor outputs, as for instance `__child__in` may be set by reactions of the current reactor (like an output), but also be the upstream of a child port (like an input).

4.2. LF extensions

The section reports on some of the functionality available within LF programs to programs written using the Rust target. Most of this functionality is provided by target properties (cf. Section 2.2.2) allow specifying code generation options, and configuring some of the behaviour of the runtime. The target properties supported by the Rust runtime are described in the following two subsections. Section 4.2.3 then describes the few instances where, the Rust target gives slightly different semantics to some LF code constructs.

4.2.1. Common target properties

The Rust target supports part of the target properties supported by all targets:

timeout The *timeout* property allows specifying the maximum amount of time the program is allowed to run for. The implementation of this feature is described in Section 3.12.2.

keepalive The *keepalive* property specifies that the program should not terminate as soon as the event queue is empty. As explained in Section 3.12.4, the property is not useful in the Rust target, as the runtime implements a better strategy to keep the program alive only as long as necessary. It is still supported for compatibility, but is ignored.

files The *files* property allows copying files to the generated sources directory. In the Rust target, it is not very useful, as including Rust files is instead done with the *rust-include* property (cf. Section 4.2.2).

The other common properties, that are currently not supported, are the following:

fast The LF *fast* property is currently not supported, although there is no particular obstacle to its implementation. It is the subject of Paragraph 6.3§Fast execution mode in the Future Work chapter.

logging The LF *logging* property is not supported, as the log levels it uses that are incompatible with Rust's standard logging levels. Specifically, Rust's standard log levels are `error`, `warn`, `info`, `debug`, and `trace`, while LF's are `error`, `warn`, `info`, `log`, and `debug`. Since both cannot be reconciled because of the `debug` level, the Rust target sticks to Rust's standard levels, and does not use this target property to configure log levels. Instead, it relies on an environment variable. As this variable can be set at runtime, it can be changed without recompiling the LF project, and is therefore more flexible.

```
reactor /* ...*/ {
  timer t(offset, period);

  // the bootstrap reaction
  reaction(startup) {=
    ctx.schedule(t, After(offset));
  =}

  // the periodic reaction
  reaction(t) -> t {=
    if !period.is_zero() {
      ctx.schedule(t, After(period));
    }
  =}
}
```

Listing 4.9: LF code showing generated synthetic reactions for timers.

```
reactor ChildReactor {
  input in: u32;
  output out: u32;
}
reactor ParentReactor {
  child = new ChildReactor();

  reaction() -> child.in {==}
  reaction(child.out) {==}
}
```

(4.10.a) LF code for a reactor whose reactions reference ports of a child reactor.

```
reactor ChildReactor {
  input in: u32;
  output out: u32;
}
reactor ParentReactor {
  child = new ChildReactor();

  // these ports are neither
  // inputs nor outputs
  port __child__in: u32;
  port __child__out: u32;

  reaction() -> __child__in {==}
  reaction(__child__out) {==}

  child.out -> __child__out;
  __child__in -> child.in;
}
```

(4.10.b) Pseudo-LF code showing how the Rust code generator interprets the `ParentReactor`.

Listing 4.10: Shows how child port references are implemented by the Rust code generator.

4.2.2. Rust-specific target properties

The generated code is a Cargo project, and for usability of the Rust target, some of the features of Cargo are accessible through target properties.

Dependency declarations The LF user can specify dependencies on arbitrary Cargo crates using the *cargo-dependencies* target property. An example usage is shown in Listing 4.11.a. On line 3, a dependency on the crate `rand` (a random number generation library) is registered, along with a version specification. The target property can also be used to configure the runtime crate, as shown on line 4. Here, the parallel execution feature is enabled. Any combination of the features described in Section 3.15 is possible. The corresponding generated `Cargo.toml` file is shown in Listing 4.11.b. Any dependency can be registered using the dictionary syntax shown on line 4. The keys and their meaning are defined by the Cargo specification, as they match the keys usable in a TOML dependency specification.

```
1 target Rust {
2   cargo-dependencies: {
3     rand: "0.8",
4     reactor_rt: {
5       features: ["parallel-runtime"]
6     }
7   }
8 };
```

(4.11.a) Rust target declaration that specifies some cargo dependencies.

```
[dependencies.rand]
version = "0.8"

[dependencies.reactor_rt]
features = ["parallel-runtime"]
```

(4.11.b) Corresponding excerpt of the generated `Cargo.toml` file.

Listing 4.11: Example usage of the *cargo-dependencies* target property.

Rust module inclusion An LF program can also include files written in pure Rust as submodules of the root module of the crate. The target property *rust-include* is used for this purpose; its value is an array of file paths that should be included. They are linked into the generated project by generating a corresponding `mod` declaration in the main file (cf. Paragraph 2.4.4§Module system). Using this property, an entire module tree can be included by specifying a directory containing a `mod.rs` file. It will then be recursively cloned to the generated sources directory.

CLI generation The generated crate also declares Cargo features (cf. Section 2.4.6), which are used for conditional compilation, and conditional dependencies. To enable features, the *cargo-features* target property is used, as shown in Listing 4.12. Currently, the only supported feature is named *cli*, as it adds a command-line interface (CLI) to the generated executable. When this feature is enabled, a CLI argument parser library is added as a dependency, and a conditionally compiled module is enabled within the crate. This module interacts with the parser library to configure the CLI argument parser. One CLI parameter is generated for each constructor parameter of the main reactor, and additional parameters are generated to set parameters of the runtime.

```
target Rust {
  cargo-features: ["cli"]
};
```

Listing 4.12: Example usage of the *cargo-features* target property.

4.2.3. Differences with other targets

Constructor parameters The LF Rust target has one significant divergence point from existing targets. In all LF targets, constructor parameters are available within the body of reactions. For instance in the Python target, parameters are stored in instance variables, like state variables.

In Rust, this is more complicated to implement, as LF constructor parameters have to be in scope in the argument expressions of a nested reactor constructor call. Destructuring the parameter struct to bring parameters in scope (cf. Listing 4.6, line 9) and then rebuilding it to save it for later access within reactions would cause surprising errors in some cases, as referencing a parameter in an argument to a child instance creation may move the parameter outside of the assemble function. One alternative would be to mandate that parameter types implement the `Clone` trait, and produce a clone of the parameter struct before destructuring it. This trait bound might be constraining for LF users, so currently, LF constructor parameters are only visible within nested reactor constructor calls, and state variable initializers. To access a constructor parameter from within a reaction, it is necessary to store it into a state variable explicitly.

This design makes LF constructor parameters simpler, as they are just treated as local variables. Persisting state is solely done with state variables, which also makes the code generator simpler, and those two LF features more orthogonal. Persisting parameters in state variables is still a common pattern in the Rust target, and it is verbose. Future work could make it more ergonomic (cf. Section 6.2.1).

State variable initialization Since `{= =}` code blocks are generated as Rust block expressions, they can include arbitrary initialization logic, including statements and intermediary variable declarations. This contrasts with other targets, where variable initializers are usually restricted to being a single expression, and more complex initialization logic has to be written in a `startup` reaction. Rust cannot use this pattern, as all fields of the state struct have to be initialized when it is created (cf. Paragraph 2.4.1§Structs). Deferred initialization in a reaction is much less practical in Rust.

To make this more ergonomic, in the Rust target, state variable initializers can refer to other state variables (provided there is no forward reference). This is another difference compared to other targets, where state variable initialization details are usually unspecified.

4.3. Compiler implementation

The LF compiler, LFC, was presented in Section 2.2.3. This section focuses on the implementation of LFC, and presents how the Rust code generator is implemented.

4.3.1. Technologies

Xtext The compiler frontend is built using the Xtext framework [13]. Xtext provides a grammar language that allows defining a lexer, parser, and the classes modelling the language's AST in a single file. Xtext automatically implements an attribution pass on the AST that link together code constructs that refer to each other. This even works across files, as Xtext has a built-in file import mechanism on top of which LF's `import` statement is built. Additionally, Xtext is able to automatically generate classes that plug in to Eclipse's APIs, to enhance Eclipse with language-specific IDE features like compilation, syntax highlighting, and reference resolution. Xtext is also able to generate a language server for the language that implements Microsoft's Language Server Protocol (LSP) [35], and hence can be used with IDEs that support it, e.g. Visual Studio Code. All this makes Xtext a powerful framework to bootstrap language development, and provide with relatively little effort a front-end and editing environments for language users.

Eclipse However convenient Xtext has been to get the compiler started, it also locks our compiler in the Eclipse environment, which is significantly split from the more mainstream Java world. Eclipse relies on OSGi [2], and hence cannot use dependencies that weren't packaged as an OSGi bundle. This excludes a lot of useful libraries that are available in package repositories like Maven Central[1], the main package repository for such widely used build tools as Apache Maven [14] and Gradle [23].

Languages The compiler runs on the Java Virtual Machine (JVM), and unsurprisingly, parts of it are written in Java. At the start of this work though, most of the compiler was written in Xtend [12], a language that also runs on the JVM and claims to improve on Java's perceived flaws. Xtend has very good interoperability with Java, and many convenience features such as extension methods, string interpolation, and null-coalescing operators. However, Xtend also has shortcomings of its own. Since 2021, the language is barely maintained, and might become unmaintained in the near future. For this reason, the LF team aims to gradually remove Xtend sources from the codebase.

The LF team has instead turned its eye to the Kotlin [18] language, which also promises an improved developer experience compared to Java. Kotlin is developed by JetBrains, and runs no risk of going unmaintained in the near future, as Google's endorsement

[1]`https://mvnrepository.com/repos/central`

of the language for Android Platform development in 2017 [29] gave the language a powerful ally. Xtend sources are to be rewritten either in Java or Kotlin.

4.3.2. Code generators

Historically, each target code generator has been written somewhat independently, and not necessarily by the same developers. At the start of this work, most of them were pretty monolithic Xtend classes. The exception was the C++ code generator, which had recently been rewritten in Kotlin and featured a more modular design. The TypeScript generator was soon to follow.

Most of the compiler (including the code generators) relies heavily on the monolithic Xtend class `ASTUtils`, which provides API that is missing from the AST classes through Xtend extension functions. Extension functions are static methods (in the Java sense) that can be called as if they were instance methods of their first parameter. Extension functions are a useful tool to extend an API which one does not control. This is the case with the AST classes, as they are generated by Xtext and cannot be modified manually. Kotlin also features extension functions, albeit with a more explicit syntax. Xtend extension functions are not recognized as extension functions by the Kotlin compiler.

4.3.3. The Rust code generator

The Rust code generator is written entirely in Kotlin. Its entry point is the class `RustGenerator`. `RustGenerator` virtually does not use its base class `GeneratorBase`, as this is a very complex class which contains much C-specific functionality, and is very hard to change. It still extends it for compatibility with other code generators.

Model classes Contrary to other existing code generators, the Rust code generator features a specialized set of model classes that abstract away the AST, and present Rust-specific code generation information. The Rust code generator first builds a tree of those model classes from the AST, then hands them to an *emitter* object, which emits the generated code using only the model classes as input.

For instance, the `State` AST node, which represents a state variable, is represented by the model class `StateVarInfo`. This class exposes the initializer expression as target code, and not as a child AST node. This simplifies the emitter object, as it is handed sanitized[2] target code, that it can then include directly into the string templates used for code generation, without further transformation.

The root of the tree of model classes is the class `GenerationInfo`, visible in Figure 4.2. It contains metadata about the generated crate, and information about the code generation options specified as target properties. It also contains a list of all `ReactorInfo` of the program, which are the model classes that represent reactor class declarations. `ReactorInfo` and related model classes are illustrated in Figure 4.3.

[2]For instance, identifiers may be escaped so as not to conflict with reserved Rust keywords.

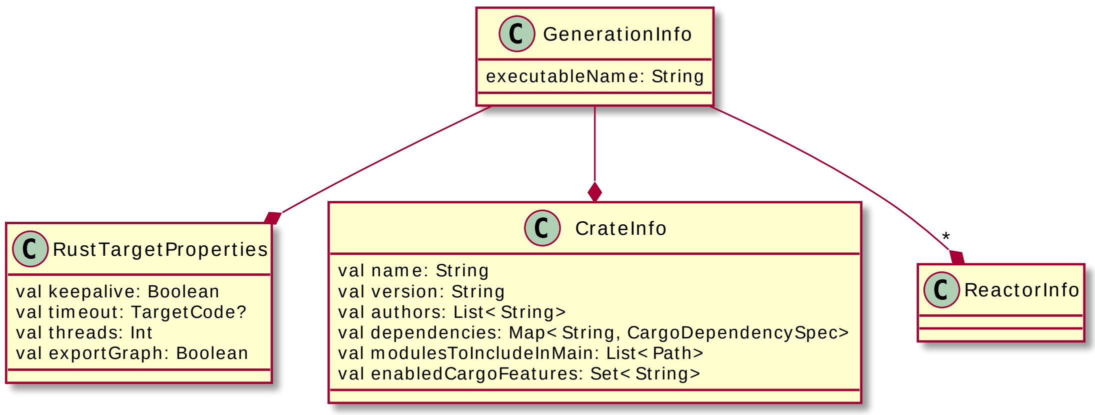


Figure 4.2.: Top-level classes of the Rust model classes.

Code emission After a `GenerationInfo` has been created, it is handed to the object `RustEmitter`, which generates the entire Cargo project. Several helper objects are responsible for emitting specific parts of the project; for instance, `RustCargoTomlEmitter` emits the `Cargo.toml` file that configures Cargo. Most of these objects rely heavily on Kotlin's string interpolation feature, which allows inserting the value of arbitrary Kotlin expressions inside a larger string template. In fact, splitting the generation into a separate code emission phase that works on intermediate model classes is intended to resemble the usage pattern of a template engine like Apache Velocity [15]. Compared to using a template engine though, writing the code emission logic in Kotlin is more flexible.

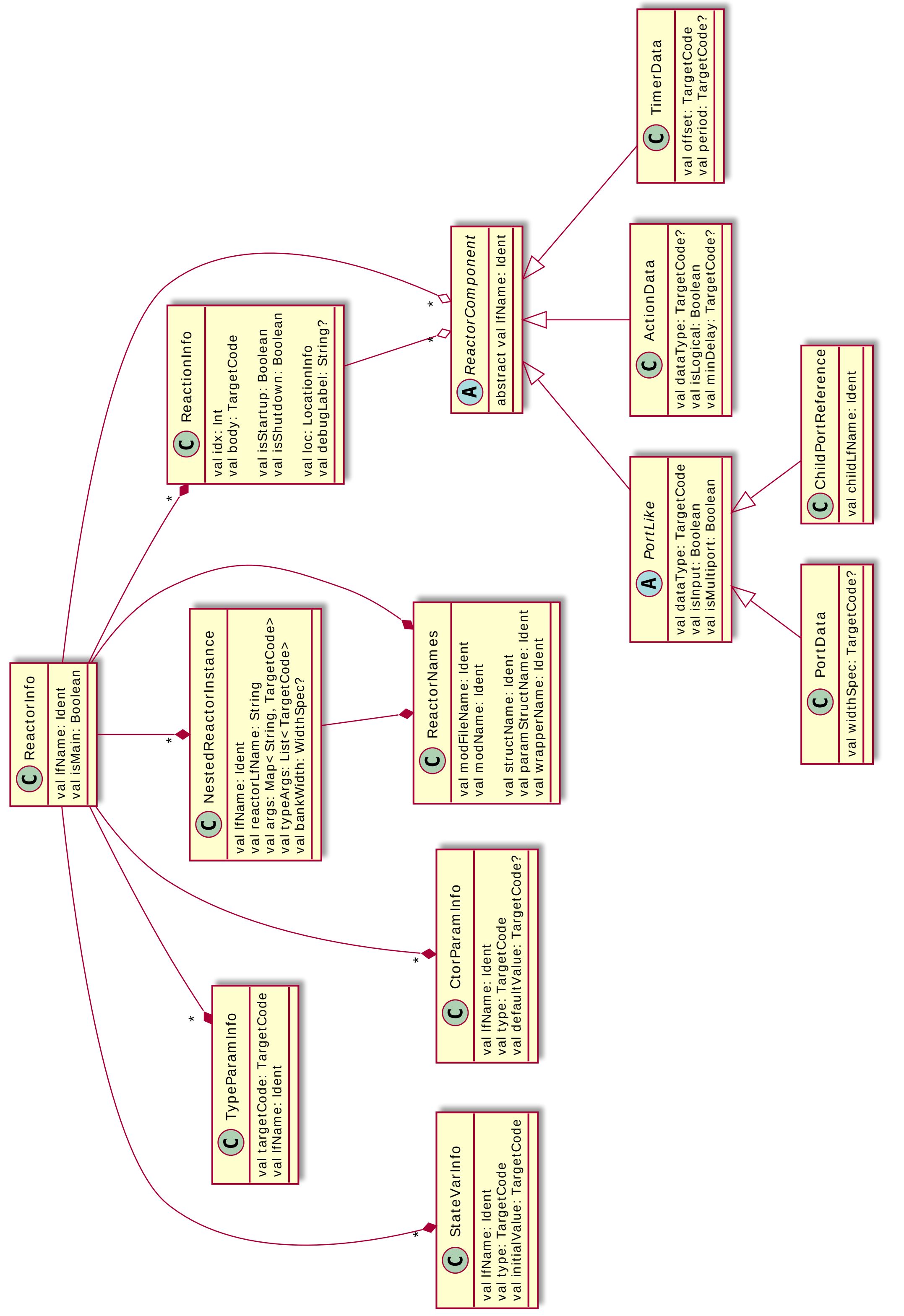


Figure 4.3.: Model classes for reactors and their components.

5. Evaluation

Message passing frameworks like actor frameworks and publish-subscribe architectures are popular in many applications. The nondeterminism introduced by the communication architecture is accepted as a necessary evil to reach the scalability and performance requirements of the system. Therefore, an interesting question is, how fast can a synchronous, deterministic framework like LF can run? Rust has a reputation for being a fast language, capable of rivalling with C and C++, but the Rust runtime prototype [16] had disappointing performance results. The runtime described in this thesis attempts to address the design flaws of the prototype (cf. Section 3.16), and has itself been significantly optimized since its inception.

In this chapter, Section 5.1 explains some of the optimizations performed on the runtime. The observations of this section can be a source of inspiration for future target implementations, and perhaps for the existing C++ and C runtimes. Section 5.2 presents some benchmark results to compare the Rust target to other LF targets and to an actor framework, Akka [6]. While its performance analysis does not attempt to be exhaustive, it certainly can give a first impression of how the Rust runtime fares when compared to those other frameworks.

5.1. Optimizing the Rust runtime

In order to optimize any program, it is necessary to first assess its performance. For the Rust runtime, the Ping Pong benchmark of the Savina benchmark suite [22] was used to confirm the impact of optimizations. Through profiling, that benchmark also served as a source of ideas for new optimizations.

Criterion Criterion[1] is a benchmarking framework for Rust, that plugs in to Cargo seamlessly. Compared to the standard benchmarking tools of Cargo, Criterion performs statistical analysis, and most importantly, it keeps a record of the latest run benchmark as a baseline. When running a benchmark, Criterion prints statistics about the relative change in performance compared to the latest run benchmark. This makes it a very useful tool to measure the impact of a change quickly. Criterion was used throughout the development of the runtime to compare the Rust runtime to itself on the same benchmark.

Workflow The use of a single benchmark is justified by the large time cost associated with maintaining runnable benchmark code, while simultaneously developing the run-

[1]`https://crates.io/crates/criterion`

Label	Commit	ΔT (%)	Description
a	`4cf2ed6`	-24.2	Avoiding copies using `Cow`
b	`1568091`	-12.0	Minimizing creation of new `Sender` instances
c	`1b18bf8`	-8.1	Removing hash maps within actions
d	`8022924`	-29.1	Introducing `ExecutableReactions`
e	`862ad4a`	-26.8	Eliding atomic runtime-checks in the internals of ports
f	`e43f9bc`	-18.2	Removing hash maps to index dependency sets
g	`98614f3`	-9.5	Representing reaction sets with a vector

Table 5.1.: Key for the labels in Figures 5.1 and 5.2.

time and the LF code generator. The benchmark is written in LF, but Criterion expects Rust files, so every incompatible change in LFC or the runtime requires recompiling the LF file, and manually patching the generated program to use the Criterion API.

The main problem with using a single benchmark as a measure of efficiency is the lack of variety of the measured usage patterns. There is a significant risk of over-optimizing for that specific benchmark, to the detriment of other usage patterns. This has been the case for instance with the data structure used by actions (cf. Section 5.1.4.4). The Ping Pong benchmark also does not benefit from parallelism and cannot be used to evaluate the performance of the code paths associated with reaction parallelization.

To alleviate this risk, a greater variety of benchmarks has been written outside of the runtime repository. These rely on the benchmarking infrastructure of the LF project, which was built to compare LF targets to each other, and to actor frameworks like Akka. The results of these are described in Section 5.2 These benchmarks show that the performance of the Rust runtime generalizes to other usage patterns.

Performance history Figure 5.1 presents a historical record of the execution time of the Savina Ping Pong benchmark. For the 180 most recent commits in the runtime crate (since `8d76a72`), 100 iterations of the benchmark were performed and their execution time averaged. Each iteration consists of 10^5 message exchanges between the two Ping and Pong reactors.[2]

Average execution time of the benchmark has been divided by nearly 3.5 between the oldest and most recent commit. Most of the improvements can be attributed to only a handful of optimizations, whose commits are labelled in red. Figure 5.2 presents the rate of change in execution time per commit, again with the same labels. These labels are described in Table 5.1. Each of these optimization is the topic of one of the following sections, in the following order:

Section 5.1.1 Introduction of `ExecutableReactions`, a specialized data structure to represent reaction sets (spike d);

Section 5.1.2 Avoidance of copies using `Cow` (spike a);

[2]The source code of the benchmark is provided in Appendix A.

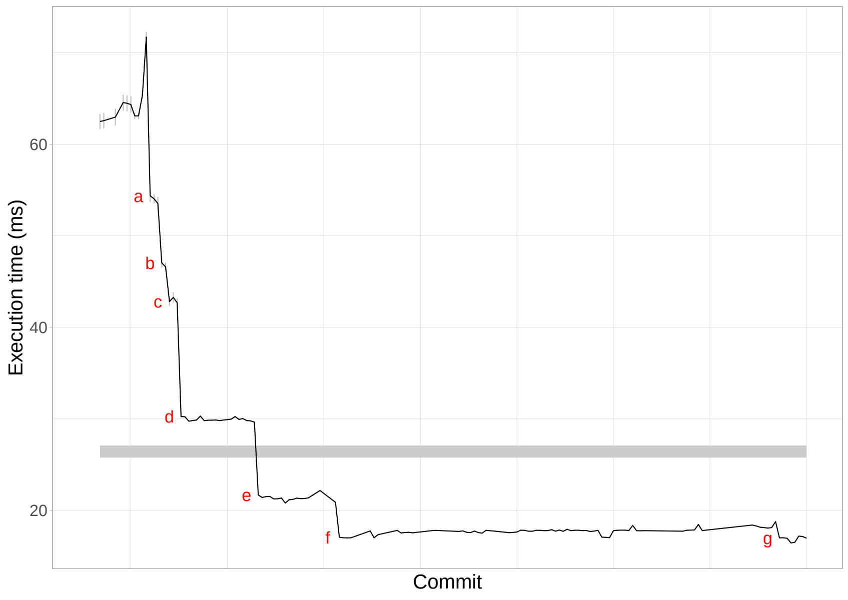


Figure 5.1.: Historical data for the average execution time of the Savina Ping Pong benchmark. The grey stripe represents a 95% confidence interval for the execution time of the same benchmark by the C++ target. Confer Table 5.1 for a description of the labelled points.

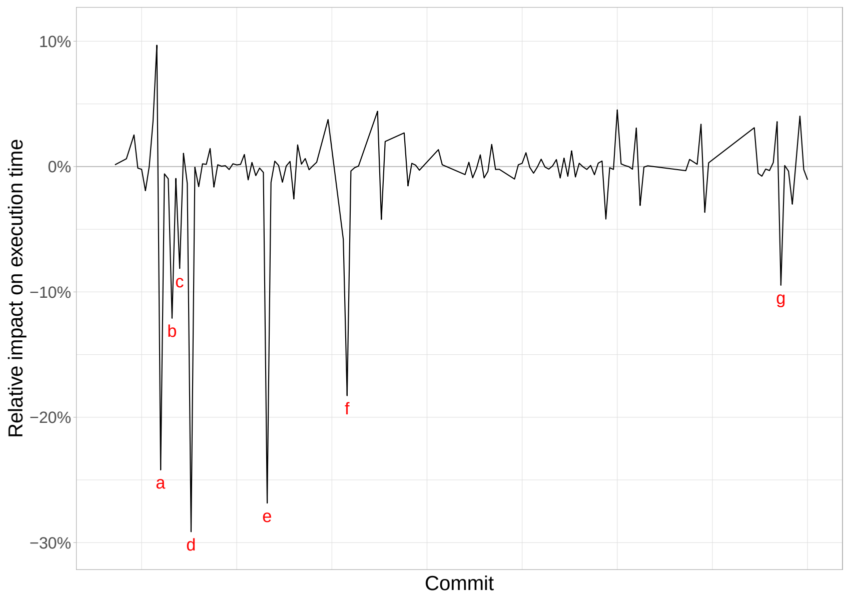


Figure 5.2.: Impact on execution time of the Savina Ping Pong benchmark by commit. Downward spikes correspond to very impactful optimizations. Confer Table 5.1 for a description of the labelled points.

Section 5.1.3 Minimizing atomic operations (spikes b and e);

Section 5.1.4.1 Swapping `HashMap` for `IndexVec` to index dependency sets (spike f);

Section 5.1.4.2 Swapping `HashSet` for a sorted `Vec` for reaction sets (spike g);

Section 5.1.4.4 Swapping `HashMap` within actions for a vector-based map (spike c).

5.1.1. Sparse reaction sets

The `process_tag` implementation shown in Listing 3.4 relies on using a set of reactions, named `todo`. The way this set is used is inefficient: on line 9, the function iterates through the entire set to fetch reactions at a particular level. It also iterates over the entire range of level indices, even though levels where at least one reaction is triggered are commonly sparse.

Another inefficiency is that the outer loop only terminates if the `todo` set is progressively emptied by the reaction, so it needs to be a mutable set. However, experience shows that in many programs, we can avoid copying sets of reactions and reuse some that are precomputed at initialization time. This will be the focus of Section 5.1.2.

The actual set implementation used avoids this by partitioning reactions according to their level. In effect, instead of using a `Set<GlobalReactionId>`, the data structure is similar to a `Map<LevelIndex, Set<GlobalReactionId>>`. This data structure is called `ExecutableReactions` in the runtime. The map implementation is a bespoke data structure, that is designed to store mappings in a vector, with keys in ascending order. This allows very fast merging of two `ExecutableReactions` instances (required at line 15 in Listing 3.4), as the two vectors can be zipped together. This implementation yields $O(\log n)$ complexity for random access, which is worse than `HashMap`'s constant time. However, the runtime does not need random access, as levels are only fetched in increasing order. Fetching the next level can be done in constant time, with a much lower constant factor than `HashMap`.

The outline of the actual outer loop used in `process_tag` is shown in Listing 5.1. The performance gain observed after implementing this is visible in Figure 5.2 as spike d.

5.1.2. Avoiding copies

Profiling the Ping Pong benchmark showed that a significant strain on the application is the copying of `ExecutableReactions` instances. This was so because such instances were being merged together at the end of the inner loop of Listing 5.1 (line 16).

Observing that many reactions only ever set at most one port, the runtime can be optimized to reuse existing `ExecutableReactions` instances instead of cloning them. Indeed, when a reaction sets several ports, all the reactions they trigger are fetched from `DataflowInfo`, and merged into a single set, that is then finally merged into the `todo` set in `process_tag`. This means the reaction context must own an instance of `ExecutableReactions`, in order to mutate it. However, usually, at most one `ExecutableReactions`

```rust
fn process_tag(
  &mut self, // SyncScheduler
  tag: EventTag,
  initial_reactions: ExecutableReactions,
) {
  let mut todo = initial_reactions;
  let mut min_level = 0;
  while let Some((level_no, reactions)) = todo.next_batch(min_level) {
    // level_no is the level of the current batch of reactions,
    // since the map is sparse, ‘next_batch‘ steps over empty levels.
    min_level = level_no + 1;
    for n in reactions {
      self.execute_reaction(n, &mut reaction_ctx);
    }
    // executing reactions may have triggered new ones
    todo = todo.union(reaction_ctx.get_triggered_reactions());
    self.push_events(reaction_ctx.get_future_events());
  }
}
```

Listing 5.1: More realistic implementation of the outer loop of `process_tag`. Compare with Listing 3.4.

is merged into this set. Therefore, in that best case, the reaction context could borrow a reference to the internal value yielded by `DataflowInfo`, and avoid cloning it.

Rust provides a smart-pointer for that purpose: `Cow`, for *copy-on-write*. Its declaration is reproduced in Listing 5.2. A `Cow` can be either a reference, or an owned value. The trait bound `ToOwned` constraints the type of borrowed data `B` to implement a function that turns it into owned data, which is called when a client asks for a mutable reference to the input data. Most commonly, this transformation just clones the referent.

```rust
pub enum Cow<’a, B: ToOwned + ’a> {
  Borrowed(&’a B),
  Owned(<B as ToOwned>::Owned),
}
```

Listing 5.2: Declaration of the `Cow` smart-pointer in the Rust standard library.

Since some reactions do not trigger new reactions at all, the runtime represents sets of executable reactions with a type `ReactionPlan`, whose declaration is reproduced in Listing 5.3.

Notice the lifetime parameter: since a `Cow` may contain a reference, its lifetime must be quantified. `ReactionPlan` instances are used extensively throughout the runtime, for instance, apart from being stored within reaction contexts, its instances are also used within events (cf. Listing 3.9). This lifetime parameter needs to be specified ev-

```
type ReactionPlan<'a> = Option<Cow<'a, ExecutableReactions>>;
```

Listing 5.3: Declaration of `ReactionPlan`, the type used to represent a possibly absent, possibly borrowed set of reactions.

erywhere in the runtime, and ultimately, since the references point to the internals of the `DataflowInfo`, the `DataflowInfo` instance need to outlive all those references. This means that `DataflowInfo` must outlive any event, and since events are contained by the event queue, which is contained by the scheduler instance, `DataflowInfo` must outlive the scheduler itself, and cannot be owned by it. Lastly, since these are shared references, `DataflowInfo` cannot be mutated while the scheduler is alive. So, while this optimization is very significant (cf. spike a in Figure 5.2), it also prevents the dataflow graph from being mutated at runtime, which is an obstacle to implementing reactor mutations in the future.

Another significant observation about the actual execution patterns of `process_tag` can be made: often, `ExecutableReactions` instances only contains reactions in a single level. After they have been executed in the inner loop of `process_tag` (line 12 in Listing 5.1), they will never be fetched again, as `process_tag` only queries reactions of strictly increasing level. This insight allows the runtime to avoid merging reaction plans in many situations. For instance, imagine `process_tag`'s `todo` set only contains reactions at level 3. After executing those reactions, the reaction context contains a set s of newly triggered reactions. Let us assume the reactions in s are all of level 5. Then, instead of merging s into `todo`, the reaction context can *set* `todo` to s. Since `process_tag` will never query level 3 again, merging both sets is unnecessary, and the loop can just carry on with those reactions of level 5. To implement this optimization, the union function called line 16 of Listing 5.1 has been rewritten to take the current level into account.

Together, these optimisations make up spike a in Figure 5.2.

5.1.3. Avoiding atomic operations

Atomic operations are necessary to implement thread-safe reference counting, for instance in the `Arc` smart pointer. Profiling showed that these operations are very costly, and identified two places in the runtime where they could be elided.

The first is the asynchronous channel used to transmit asynchronous events to the scheduler (cf. Section 3.12.1). The channel maintains a count of currently alive sender and receiver instances using atomic operations. To avoid these operations, it is necessary to avoid creating short-lived receiver or sender instances, as each instance requires two atomic operations, one when creating the new instance, and one when destroying it. Each reaction context instance requires a way to produce a new sender instance, so as to produce new `AsyncCtx` instances for use with physical actions. With the patched channel implementation described in Section 3.12.4, it suffices for the reaction context to have access to a receiver instance to create new senders. The reaction context initially owned a cloned receiver instance, which was destroyed at the end of `process_tag`. To avoid the atomic operations this usage pattern implies, the reaction context was updated to capture a reference to the receiver used by the scheduler. This optimization is visible in Figure 5.2 as spike *b*.

Atomic operations were also used in the internals of ports. As described in Section 3.10.1, ports contain a reference to a mutable data cell for the value, and the Rust compiler requires that accesses to this location not cause data races. To implement this, a smart pointer like `AtomicRefCell`[3] can be used, as it checks at runtime that Rust's borrowing rules are respected (one writer XOR zero or more readers), using atomic operations. By removing this smart pointer, and instead relying on unsafe code to access the data cell, a large performance improvement was observed. In Figure 5.2, it is labelled as spike *e*.

5.1.4. Data structure selection

This section focuses on some of the data structures used internally by the runtime, and how they have been optimized.

5.1.4.1. TriggerId internal representation

For much of the history of the runtime (until point *f* in Figure 5.1), `TriggerId` was represented just like `GlobalReactionId`, as a tuple of a `ReactorId` and a local identifier within the reactor. While this structure makes for a convenient implementation of `DebugInfoRegistry`, it also makes the only reasonable implementation of a map whose keys are `TriggerId` a hash map, since trigger IDs of different reactors are numerically far apart. Such a map data structure is stored in `DataflowInfo`, and is solicited every time a port is set or an action or timer is scheduled.

Contrary to reaction IDs, trigger IDs do not require access to the ID of their reactor, except to produce debugging information. Given that, a change in the structure of `TriggerId` enables using a much more efficient map implementation that `HashMap`. Namely, if trigger IDs are simplified to be a newtype over `u32`, and are allocated compactly for all trigger components of the program, then the map can be implemented as a vector of values, where the index of each value in the vector corresponds to the `TriggerId` key. This change corresponds to point *f* in Figure 5.2.

5.1.4.2. Reaction sets for levels

In Section 5.1.1, the data structure for reaction sets is explained to be conceptually a `Map<LevelIndex, Set<GlobalReactionId>>`. The data structure corresponding to the `Set<GlobalReactionId>` is called `Level`. While it must retain the set property (absence of duplicates), the most stressful operation on this data structure it iteration, not insertion. Indeed, levels are iterated over in the inner loop of `process_tag`, which happens more frequently than merging two levels. Most of the `ExecutableReaction`

[3] `https://docs.rs/atomic_refcell/`

instances are never mutated after their creation during assembly and are shared (cf. Section 5.1.2), which reduces the number of created level sets, and hence of insertions.

To optimize this set data structure for iteration, the `Level` data structure does not use a hash table, but instead a sorted vector of `GlobalReactionId`. While insertion in order becomes $O(n)$ compared to `HashSet`'s amortized $O(1)$, iteration has a lower constant factor than with `HashSet`, yielding a significant performance improvement on the Savina Pong benchmark (cf. spike g in Figure 5.2).

5.1.4.3. Event queue

The event queue is implemented on top of Rust's `VecDequeue`, a dequeue data structure based on a circular buffer. This offers excellent performance while the event queue is small. It exhibits best-case performance for insertion of an event at the start and at the end (i.e., events that are earlier or later than all current events). Despite these desirable characteristic, the theoretical complexity of inserting an event at a random location is high (linear). It is easy to build a pathological program that will always hit this worse case. However, this hasn't proved to be a problem in the LF programs built so far with the Rust target.

5.1.4.4. Action values

Getting the value of a port is equivalent to a couple of pointer reads, and is hence very fast. This can be made so efficient because ports only ever store one value at a time. In contrast, at any given time, any number of triggerings of an action may already have been scheduled for future tags. Each of those triggerings can feature a distinct value, and all those values need to be stored somewhere for future retrieval. For instance in Listing 5.4, the same logical action is scheduled three times, and each value has to be saved along with its tag.

```
main reactor {
  logical action act: u32;

  reaction(startup) → act {=
    ctx.schedule_with_v(act, Some(30), after!(30 ms));
    ctx.schedule_with_v(act, Some(50), after!(50 ms));
    ctx.schedule_with_v(act, Some(70), after!(70 ms));
  =}

  reaction(act) {=
    println!("Received {} at {}", ctx.get(act).unwrap(), ctx.get_tag());
  =}
}
```

Listing 5.4: Example program that schedules a logical action several times.

This is implemented with a mapping of tags to values, stored within the action instance. Although using a hash table (Rust's `HashMap`) is an accessible initial implementation, hashing keys was shown to take a lot of time. Since values are retrieved with a monotonically increasing time, the data structure should support fast removal for the smallest key. Based on the assumption that `schedule` is usually called with a monotonically increasing time, the chosen implementation should support fast insertion for large keys.

An optimal map implementation for this usage pattern would hence be a rotating vector of sorted key-value pairs, e.g. implemented with Rust's `VecDequeue`. With such a data structure, insertion at the end (largest key) uses amortized constant time, and removal at the start (smallest key) uses constant time.

Unfortunately, while the current implementation uses the same idea, it does not use a circular buffer. This yields amortized constant time insertion and removal at the end, but linear time insertion at the start. However, the degenerate case of a map of size 0 to 1 has constant time insertion and removal at both ends (which coincide). This seems to be the usage pattern of the Savina Ping Pong benchmark, which is why Figure 5.2 shows spike *c*. Improving this implementation by using a rotating vector is left for future work.

5.2. Comparison to other frameworks

The LF development team has implemented a benchmark suite based on the Savina suite [22]. The Savina suite is designed to cover a wide range of programming patterns for actor programs. Its reference implementation provides benchmark implementations for the Akka actor framework [6]. Most of the Savina benchmarks are also implemented in the LF C and C++ targets. A subset of those benchmarks were implemented[4] in the LF Rust target in order to perform an initial comparison with the existing LF targets and Akka.

Figure 5.3 shows the results of this comparison. Each data point corresponds to the mean execution time of 18 iterations[5] of the benchmark program, for a given amount of parallelism (specified as a number of threads), and for a given target framework. All measurements were performed on an Intel(R) Core(TM) i7-9700T CPU with 32 GiB RAM. Note that the Rust benchmark programs measured here have not used the Rust compiler's profile-guided optimization feature, which might improve the Rust results further.

Discussion While an in-depth discussion of these results is out of scope for this document, these initial experiments show that the LF Rust target can compete with those other frameworks.

The first two plots belong to the group of concurrency benchmarks. The LF targets show an adverse scaling behaviour as the number of available threads rise, especially

[4]Thanks to Johannes Hayeß for his help in this effort, particularly in fixing the Radix Sort benchmark.
[5]20 total iterations, including 2 warmup iterations that were removed from the final data.

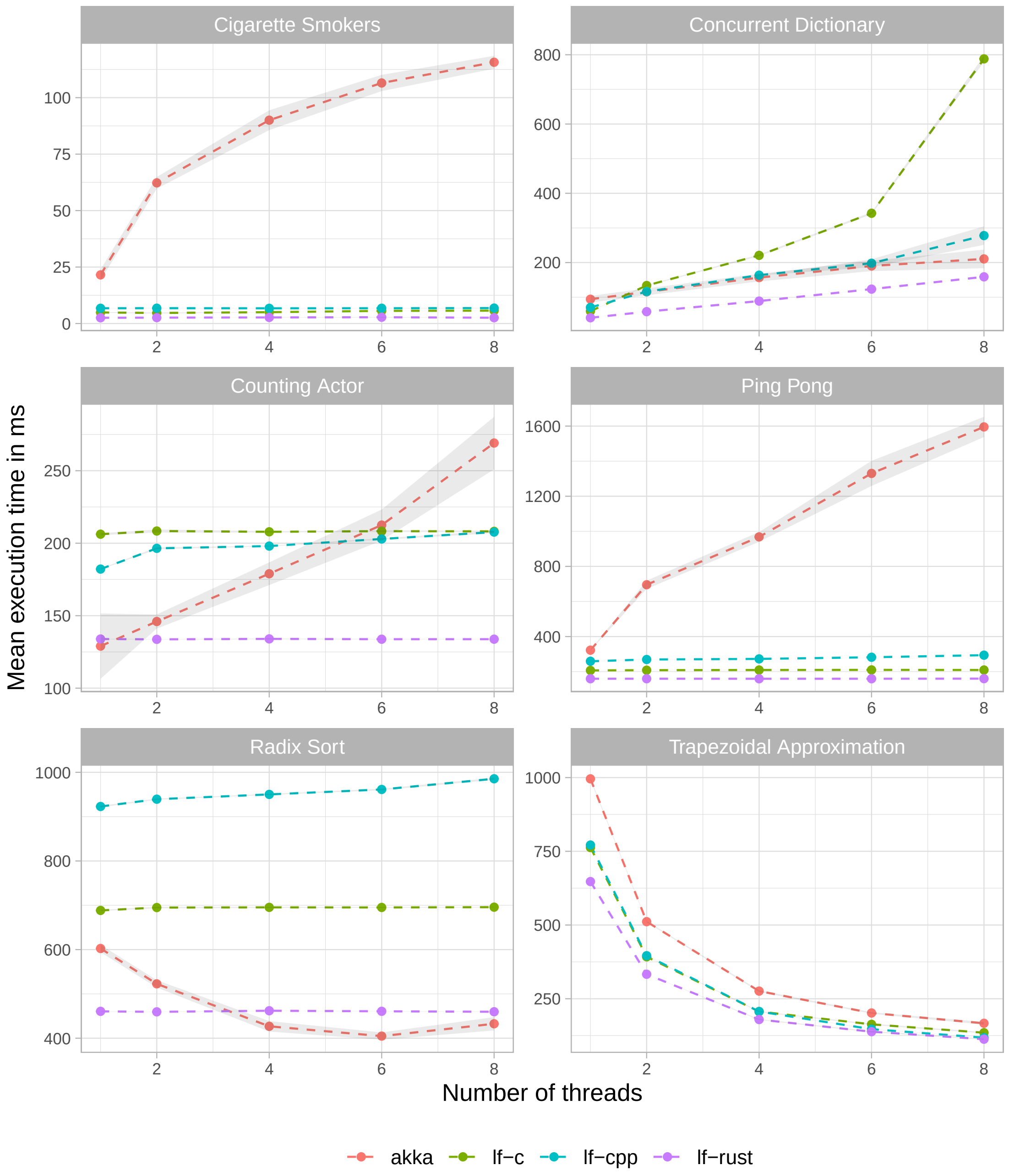


Figure 5.3.: Comparison of Akka and different LF targets on six benchmarks of the Savina benchmark suite.The grey halo shows a 95% confidence interval for the position of each point.

in the Concurrent Dictionary benchmark. That benchmark evaluates contention for a single data structure accessible by many worker reactors, and the C target performs significantly worse than the Rust and C++ targets. The next two plots (Counting Actor and Ping Pong) belong to the group of micro benchmarks. Those benchmark programs are entirely sequential, and expectedly, the curves are relatively flat. In those four plots, the Akka framework appears scales very badly. The Akka runtime might be distributing workloads across all available threads, which causes overhead due to context switching. By contrast, since in LF, communication between reactions is synchronous, the scheduler knows when it makes sense to dispatch different reactions to different threads, and when it does not.

The last two plots (Radix Sort and Trapezoidal Approximation) belong to the group of parallelism benchmarks. The Trapezoidal Approximation benchmark shows excellent scaling behaviour for all the measured frameworks. Akka performs better than LF for the Radix Sort benchmark though.

In all these experiments, the Rust target performs better than both the C++ and the C target. The speedup of the Rust target over the other frameworks was estimated at 1.76 for the LF-C++, 1.85 for LF-C, and 3.44 for Akka. This value was computed using the geometric mean of the speedups over individual benchmarks, for a thread count of 8.

The scaling behaviour of the Rust target appears identical to that of the C++ target. It is difficult to find an immediate culprit to account for the difference in performance between C++ and Rust here. Since the curves seem to differ only by a constant factor, it is likely that the algorithms used by both runtimes have similar performance characteristics, and that small differences in the data structures used add up to this large final difference. Without profiling the C++ target, it will be difficult to determine which optimizations can be made to improve it.

6. Future work

This chapter explores some possible avenues for future work, and gives some insight as to how they could be implemented.

6.1. Features

6.1.1. Scheduler unit tests

One problem with the scheduler as currently written is that it is difficult to test: writing reactors manually is very tedious, as it involves declaring all components and their dependencies explicitly. Most tests for the runtime are actually integration tests written in LF, in the repository where the compiler is being developed. One problem with these tests is that the LF program-under-test depends both on the correctness of the runtime scheduler, and on the correctness of the LF compiler.

The scheduler itself is a relatively small component, and our project would benefit from unit testing that component to cover more code paths. The principal obstacle is that the scheduler requires some reactors, and reactors are difficult to construct manually. A possible solution would be, instead of creating reactors, to use manual implementations of `ReactorBehavior`. This would require bypassing the assembly phase, and constructing a dependency graph manually. While this is tedious, the reactor runtime already has utilities to make this easier, which are used to unit test dependency graphs. One problem is that these utilities imitate the functionality of `AssemblyCtx`, and may get out of sync.

6.1.2. Scheduler verification

Going a step further than unit testing, the safety claims of the Rust runtime could be formally verified. Rust's design encourages encapsulating unsafe code within safe APIs. Projects like RustBelt [25, 11] attempt to verify that safe APIs actually compose, by determining automatically the verification conditions required for an API implemented by unsafe code to be safe. Using such a tool, we could discover what conditions on reactor implementations need to be satisfied in order for a reactor program to be safe. Such a result would be a significant token of confidence in the safety of programs written with the LF Rust target, and could inform the future design of LF.

6.1.3. Deadline violation detection

During execution of an LF program, logical time may lag behind physical time, in a way that compromises the real-time reactivity of the system. To counter this, LF provides a mechanism for specifying *deadlines* on reactions. A deadline specifies that the invocation of the reaction has to occur within some amount of physical time, starting from the logical time of the tag that triggers the reaction. For example, if a reaction is invoked at logical time t, and the reaction specifies a deadline of one hour, then the reaction has to be executed before the physical time of the platform advances to $t + 1$ h. If the deadline cannot be met by the runtime, i.e., the invocation occurs after the expected maximal physical time, a *deadline violation* occurs. Dealine violations should be detected at runtime, and handled by a user-specified *deadline violation handler*.

Listing 6.1 illustrates the LF syntax that defines a deadline and its handler on a reaction. In this example, if the reaction is executed later than the tag of `x` plus the deadline of 10 milliseconds, the deadline violation handler is executed instead of the reaction itself.

```
target Rust;
reactor Deadline() {
  input x: u32;
  output deadline_miss: u32;

  reaction(x) → deadline_miss {=
    info!("Normal reaction.");
  =} deadline(10 msec) {=
    warning!("Deadline violation detected.");
    ctx.set_opt(deadline_miss, ctx.get(x));
  =}
}
```

Listing 6.1: Example of the LF syntax to define a deadline alongside with its handler.

This feature could possibly be implemented entirely by the code generator. The dispatch code generated to implement `ReactorBehavior::react` (cf. Section 4.1.3) could be updated to check the current physical time, for those reactions that declare a deadline. If a deadline violation is detected, the handler should be invoked, with the same parameters as the reaction. The advantage of implementing this feature like this is simplicity of the runtime, which would not have to record a mapping of reaction IDs to their (possibly absent) deadline. It also would not add any overhead to reactions which do not have a deadline.

6.1.4. Mutations

Reactor *mutations* are special reactions that have the ability to reconfigure the connections between reactors components at runtime. In the runtime implementation described in this thesis, the main obstacle to implementing mutations is the fact that the dataflow information is summarized in a `DataflowInfo` object, and the dependency graph is destroyed before starting the execution. To avoid copying reaction sets unless necessary, events pushed to the event queue may contain references to the internals of the `DataflowInfo` instance. Rust's ownership model requires that the `DataflowInfo` not be mutated while there might be live references into its internals in the program. Given how the scheduler is currently structured, the Rust compiler assumes that such live references may exist as long as the scheduler instance is alive, and therefore forbids mutating the `DataflowInfo` instance during the entire program. However, the Rust compiler is too conservative in its analysis, as those references only actually exist while some events are pending. If the event queue is empty, then it is safe to change the `DataflowInfo`. Similarly, the vector that contains reactor instances can be changed at that time. Reactor IDs are indeed similar to references, only, their lifetime is implicit, and their validity is not checked by the Rust compiler. Only performing mutations when the event queue is empty is probably very restrictive. However, it seems difficult to support reinterpreting the contents of events after a mutation that destroyed a reactor, for instance.

Tangential to the design of mutations is the effort to support *modal models*, i.e., reactors switching between different *modes*, like a state machine. A reactor may have a different internal topology depending on its state. The relative staticity of reactor modes compared to mutations might be a better fit for Rust's strong type system. Efforts on supporting either can probably inform the design of the other.

In any case, implementing mutations will require reorganizing the scheduler, so as to change what lifetime information is exposed to the scheduler. Possibly, some type parameters of the scheduler will have to be removed, and some of its fields turned into local variables, so as to use inferred lifetimes instead of explicit ones.

6.1.5. Smarter tag cleanup

Section 3.11 explains how ports and actions need to be cleaned up at the end of a tag, to empty them of their value. This is currently implemented by resetting all ports and actions in the program at the end of the processing of a tag. This cleanup phase is a potential performance bottleneck, especially since not all ports and actions are necessarily present at all tags. Devising a better strategy to avoid this loop would be a good idea.

We could at the very least parallelize the loop with Rayon.

However, maybe a more lightweight approach would be to use the structure of port connections (cf. Section 3.5.4) to trigger cleanup code exactly when it is necessary. It might be possible, for instance, to determine the maximum level L at which a port is read, then create a synthetic cleanup reaction n and set up its dependencies such that the port triggers n, and $\text{level}(n) > L$. This would ensure cleanup is only executed if the port was present, and after all readers have had a chance to look at the data. Figuring out which reactor this reaction belongs to, and how to set up its dependencies, is the hard part.

Note that a further constraint on the tag cleanup is that setting a port to `None` drops the contained value, which calls its destructor. Since destructors may perform observable side effects (including panicking), the drop call must be called at a *consistent* time point.

6.1.6. Error handling

The Rust reactor runtime is not resilient to crashes in user-written code. A crash (in Rust lingo, a *panic*) in a reaction will just crash the entire application. Crashes in the destructors of port and action values are also not handled. The C++ and C targets currently have no good solution to handle these crashes either, however, low-level system code may crash, and a robust application should be able to recover from those faults. When a Rust program panics, the stack is *unwinded* until the first stack frame that has registered a handler. It would be desirable to protect reaction execution with such a panic handler, to allow the reactor program to recover. This should possibly only be done if the reaction was explicitly marked as fallible in LF. A possible syntax extension to tag reactions for this purpose is described in the following section.

6.2. Lingua Franca syntax

6.2.1. Annotations

Introducing an extensible attribute syntax could help make the Rust target more ergonomic. For instance, Section 4.2.3 describes how constructor parameters are not available within reactions, unless they're explicitly stored within a state variable. Listing 6.2.a shows how verbose this pattern currently is. Listing 6.2.b shows how this could be made easier by introducing a `@state` annotation. Clearly, this convenience feature could also be implemented by allowing the `state` keyword before a parameter, or some other first-class syntactic construct. The main benefit of an annotation syntax is extensibility, as any target can give meaning to the annotations they like. Compared to modifying the LF grammar itself, the overhead of adding a new annotation is minimal.

```
reactor Foo(param: u32(0)) {
  state param(param);
}
```

(6.2.a) Current pattern used to persist a parameter for use within reactions.

```
reactor Foo(
  @state param: u32(0)
) {}
```

(6.2.b) A `@state` annotation, which would be interpreted as the code of Listing 6.2.a.

Listing 6.2: Possible use case for an annotation syntax.

Other use cases for annotations abound. For instance, a `@may_panic` annotation on a reaction could instruct the Rust runtime to recover gracefully from any crash in the reaction (cf. Section 6.1.6). A `@stateless` annotation on a reaction could mean that it does not access the state variables of the enclosing reactor, and can thereby be freed from the priority ordering imposed on reactions of the same reactor. Annotations could be used to add verification assertions to reactor programs. For instance, an `@invokedExactly(1)` annotation could specify that a reaction should execute a specific number of times. Such annotations could help with static analysis of reactor programs, by letting the programmer write-down explicit verifiable contracts using their domain knowledge about the application. This idea is the object of a discussion on GitHub[1].

6.2.2. Required parameters

Lingua Franca syntax currently mandates reactor parameters to have a default value, which stems from simplifying assumptions that were made in the initial language design. While the default value is useful in some cases, in many cases it is not. In some cases, there is no possible default value writable by the programmer, which is the case with generic reactors, as shown in Listing 6.3. In this reactor, there is no possible expression that can replace the ..., as the `T` is unknown. Allowing parameters to have no default value would solve this, in which case they would be interpreted as *required*. Required parameters could also be useful for main reactors, in which case, in the Rust target, the user would need to provide a value on the command-line in order to run the program.

```
reactor Generic<T>(p: T(...)) {
  state s: T(p);
}
```

Listing 6.3: Example generic reactor with a parameter.

6.3. Other

First-class reactors in Rust Some of the current restrictions on Rust reactors can be attributed to the need to comply with the conservative static analysis of the Rust compiler. For instance, state variables cannot contain references, as the lifecycle of reactor instances is obscured from the Rust compiler by the use of dynamically sized types (fat pointers). These restrictions could be lifted if reactors were first-class citizens of the Rust language. The Rust compiler could perform very fine static analysis of the lifetime of, e.g., of port values. It would also improve the toolchain to write reactor programs, as the Rust compiler has excellent error messages, and Rust has wide IDE support.

Judging by historical decisions of the Rust team, any such language extension would probably still rely on a library for its implementation. Rust's `async/await` language feature is implemented this way, and the Rust language team has not promoted any of the several library implementations to be part of the standard library. This effort would probably require standardizing traits (similar to e.g., `ReactorInitializer`), that would be implemented automatically by the Rust compiler.

Note that macros can generate items, and could be used to integrate a "reactor DSL" to the Rust language with no change required in the compiler, and no particular input from the Rust language team. While such a DSL could help with the tooling situation,

[1]`https://github.com/lf-lang/lingua-franca/discussions/756`

as reactor programs would be pure Rust code, it is unclear whether they could be used to also perform program-wide static analysis of the reactor program at compile-time.

Better graph algorithms The implementations of graph algorithms in the runtime were not written with performance in mind. The level assignment algorithm even has worst-case exponential complexity, as it is linear in the number of paths in the graph (which, for some graphs, can be exponential in the number of nodes in the graph). This part of the runtime could make better use of dynamic programming.

LFC refactoring The design of the Rust code generator could be used in other code generators (cf. Section 4.3). The classes use to model the AST form a sort of IR, that LFC currently lacks. It would be possible to split off the Rust-specific functionality of these model classes, and reuse this IR in other code generators. Doing so could simplify some of the existing code generators, by extracting part of their data sanitization logic. For instance, the C++ code generator is currently relatively fragmented into small subclasses, and it is hard to understand how they all connect together.

Tracing The C++, C and Python runtimes support logging execution traces using a compact binary format, to minimize runtime overhead. The Rust runtime currently does not support this, although it can log human-readable debugging information. The C++ runtime currently uses LTTNG as a tracing framework, which is a tracing toolkit for Linux systems. A Rust crate[2] provides bindings to this framework, and could possibly be integrated to the Rust runtime. Support for this would most likely be locked behind a conditional compilation feature.

Fast execution mode All other LF targets support a `fast` target property, that makes the scheduler not wait between two tags, even if the current logical time is early. This might be used for simulation, and while simulation has not been a goal until now, it might be worth implementing.

Evaluate timing precision `Receiver::recv_timeout` is used to synchronize logical time with physical time, as explained in Section 3.12.2. This function has unknown precision, and an investigation would be useful to determine the resolution of the Rust runtime's logical timeline. LF has support for specifying time offsets as small as a nanosecond, but the current runtime is probably less precise.

[2]`https://crates.io/crates/lttng-ust`

7. Conclusion

This thesis presents a performant, safe Rust target for the Lingua Franca language. While a fully comprehensive evaluation is beyond the scope of this work, the initial performance results of the Rust target are very encouraging. Rust outperforms both the C and C++ targets on all implemented benchmarks, and shows good scaling behaviour on parallel computations. Future work should evaluate the runtime on a wider range of usage patterns, and possibly, against other actor frameworks. The steps taken to optimize the Rust runtime since its early prototype may in the future inform the design of other target runtimes. In particular, quantifying the impact of some optimizations provides useful insight into what data structures and operations are determining in the performance of reactor programs. This historical data can complement profiling information to guide optimization decisions.

The Rust ownership system definitely steered the design of the Rust runtime. Compared to the other existing runtime implementations, the final design presented here makes little use of references, and relies instead on symbolic identifiers. This allows a relatively simple representation of reactors, where each reactor is unaware of its neighbourhood. While this design helps remove syntactic overhead related to quantifying reference lifetimes, it is not yet known whether it will be an obstacle or an asset for the design of LF extensions like mutations.

The design of the Rust target leans into Rust's strong typing discipline to provide strong safety guarantees about the generated reactor program. Rust's type system allows verifying invariants of the reactor model at compile-time, and expressing constraints on user-provided types. For instance, port values are asserted by the compiler to be thread-safe, which is required for safe parallel execution of reactions. When the type system is too conservative, and rejects code that appears unsafe but is used safely, it can be circumvented by using unsafe code. This is done in select places in the runtime, but always encapsulated within a safe API. In those places, the safety claims made by the implementation are supported by analysis of the reactor model's constraints. The rules of the reactor model are checked both during compilation of the LF program, during compilation of the target program, thanks to the strict assembly API, and at runtime.

This work can be used as a starting point to implement more features of LF, and to enrich LF with new features, like mutations. The author hopes it can also serve as a basis for ambitious projects, like formal verification of the reactor implementation, or integrating reactors as a DSL embedded within Rust. These could provide stronger safety guarantees about reactor programs. Building trust in the Rust implementation could perhaps eventually justify reorganizing other language runtimes to use this trusted core, instead of having each language implement its own scheduling logic separately.

Bibliography

[1] Gul A Agha, Ian A Mason, Scott F Smith, and Carolyn L Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7(1):1–72, 1997.

[2] OSGi Alliance and The Eclipse Foundation. OSGi. `https://www.osgi.org/`. Accessed: 2021-12-04.

[3] Joe Armstrong. Erlang — a survey of the language and its industrial applications. In *Proc. INAP*, volume 96, 1996.

[4] Gérard Berry and Georges Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of computer programming*, 19(2):87–152, 1992.

[5] Gérard Berry and Ellen Sentovich. Multiclock Esterel. In *Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, pages 110–125. Springer, 2001.

[6] Jonas Bonér, Viktor Klang, Roland Kuhn, et al. Akka library. `https://akka.io`, 2011-2021. Accessed: 2021-12-04.

[7] Sylvan Clebsch, Juliana Franco, Sophia Drossopoulou, Albert Mingkun Yang, Tobias Wrigstad, and Jan Vitek. Orca: Gc and type system co-design for actor languages. *Proc. ACM Program. Lang.*, 1(OOPSLA), oct 2017.

[8] The Rust Community. The Rust reference – 10.3: Type Layout.

[9] The Rust Community. The Rust reference – 15.3: Behavior considered undefined.

[10] Eric C Cooper and Richard P Draves. C threads. Technical Report CMU-CS-88-154, 1988.

[11] Hoang-Hai Dang, Jacques-Henri Jourdan, Jan-Oliver Kaiser, and Derek Dreyer. Rustbelt meets relaxed memory. *Proceedings of the ACM on Programming Languages*, 4(POPL):1–29, 2019.

[12] Eclipse Foundation. Xtend website. `https://www.eclipse.org/xtend/index.html`. Accessed: 2021-11-24.

[13] The Eclipse Foundation. Xtext. `https://www.eclipse.org/Xtext/`. Accessed: 2021-12-04.

[14] Apache Software Foundation. Apache maven. `https://maven.apache.org/`. Accessed: 2021-12-06.

[15] Apache Software Foundation. Apache velocity. `https://velocity.apache.org/`. Accessed: 2021-12-06.

[16] Clément Fournier. Investigating compatibility of the reactor model with rust's ownership-based type system. Großer Beleg, Technische Universität Dresden, 2021.

[17] Jon Gjengset, Malte Schwarzkopf, Jonathan Behrens, Lara Timbó Araújo, Martin Ek, Eddie Kohler, M Frans Kaashoek, and Robert Morris. Noria: dynamic, partially-stateful data-flow for high-performance web applications. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 213–231, 2018.

[18] JetBrains GmbH. Kotlin. `https://kotlinlang.org/`. Accessed: 2021-12-06.

[19] Rewi Haar. Associated constants should not be object-safe. `https://github.com/rust-lang/rust/issues/26847`, 2019.

[20] Nicholas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.

[21] Carl Hewitt. Viewing control structures as patterns of passing messages. *Artificial intelligence*, 8(3):323–364, 1977.

[22] Shams M. Imam and Vivek Sarkar. Savina - an actor benchmark suite: Enabling empirical evaluation of actor libraries. In *Proceedings of the 4th International Workshop on Programming Based on Actors Agents & Decentralized Control*, AGERE! '14, page 67–80, New York, NY, USA, 2014. Association for Computing Machinery.

[23] Gradle Inc. Gradle. `https://gradle.org/`. Accessed: 2021-12-06.

[24] ECMA International. *Standard ECMA-404 – The JSON data interchange syntax.* 2 edition, 2017.

[25] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. Rustbelt: Securing the foundations of the rust programming language. *Proceedings of the ACM on Programming Languages*, 2(POPL):1–34, 2017.

[26] Gilles Kahn. The semantics of a simple language for parallel programming. In *Proc. of the IFIP Congress 74*, pages 471–475. North-Holland Publishing Co., 1974.

[27] Steve Klabnik, Carol Nichols, and the Rust Community. The Rust programming language: 10.3 - Validating References with Lifetimes.

[28] Steve Klabnik, Carol Nichols, and the Rust Community. The Rust programming language: 4.2 - References and Borrowing.

[29] Frederic Lardinois. Google makes Kotlin a first-class language for writing Android apps. `https://techcrunch.com/2017/05/17/google-makes-kotlin-a-first-class-language-for-writing-android-apps/`. Accessed: 2021-11-24.

[30] Edward A Lee. The problem with threads. *Computer*, 39(5):33–42, 2006.

[31] Edward A Lee and David G Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987.

[32] Edward A Lee, Stephen Neuendorffer, and Michael J Wirthlin. Actor-oriented design of embedded hardware and software systems. *Journal of circuits, systems, and computers*, 12(03):231–260, 2003.

[33] Marten Lohstroh, Christian Menard, Alexander Schulz-Rosengarten, Matthew Weber, Jeronimo Castrillon, and Edward A Lee. A language for deterministic coordination across multiple timelines. In *2020 Forum for Specification and Design Languages (FDL)*, pages 1–8. IEEE, 2020.

[34] Marten Lohstroh, Ínigo Íncer Romeo, Andrés Goens, Patricia Derler, Jeronimo Castrillon, Edward A Lee, and Alberto Sangiovanni-Vincentelli. Reactors: A deterministic model for composable reactive systems. In *Cyber Physical Systems. Model-Based Design*, pages 59–85. Springer, 2019.

[35] Microsoft. Language server protocol. `https://microsoft.github.io/language-server-protocol/`. Accessed: 2021-12-06.

[36] Scott Milton and Heinz W. Schmidt. Dynamic dispatch in object-oriented languages. Technical report, CSIRO – Division of Information Technology, 1994.

[37] MIT. Reading 17: Concurrency. `https://web.mit.edu/6.005/www/fa14/classes/17-concurrency/`, 2014. Accessed: 2021-12-04.

[38] Derek G Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 439–455, 2013.

[39] Bruce Jay Nelson. *Remote Procedure Call*. PhD thesis, USA, 1981.

[40] Tom Preston-Werner et al. Semantic versioning 2.0.0. `https://semver.org`, 2013. Accessed: 2021-12-04.

[41] Tom Preston-Werner, Pradyun Gedam, et al. TOML v1.0.0. `https://toml.io/en/v1.0.0`, 2021. Accessed: 2021-12-04.

[42] Lui Sha, Abdullah Al-Nayeem, Mu Sun, Jose Meseguer, and Peter C Olveczky. PALS: Physically asynchronous logically synchronous systems. Technical report, 2009.

[43] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software,. *Dr. Dobb's Journal*, 30(3), 2005.

[44] The Actix Team. Actix. `https://actix.rs/`, 2021. Accessed: 2021-12-04.

[45] Stefan Tilkov and Steve Vinoski. Node.js: Using JavaScript to build high-performance network programs. *IEEE Internet Computing*, 14(6):80–83, 2010.

[46] Huon Wilson. Myths and Legends about Integer Overflow in Rust. `https://huonw.github.io/blog/2016/04/myths-and-legends-about-integer-overflow-in-rust/`. Accessed: 2021-11-25.

[47] Yang Zhao, Jie Liu, and Edward A Lee. A programming model for time-synchronized distributed real-time systems. In *13th IEEE Real Time and Embedded Technology and Applications Symposium (RTAS'07)*, pages 259–268. IEEE, 2007.

A. The Savina Ping Pong benchmark

The Savina Ping Pong benchmark consists of two reactors sending messages back and forth over a port. The `count` parameter (line 40) determines how many message exchanges are executed. The Ping Pong benchmark is entirely sequential and does not benefit from parallel execution. Its definition is reproduced in Listing A.1.

```
target Rust;

reactor Ping(count: u32(1000000)) {
    input receive: u32;
    output send: u32;
    state pingsLeft: u32(count);
    logical action serve;

    reaction(startup, serve) -> send {=
        ctx.set(send, self.pingsLeft);
        self.pingsLeft -= 1;
    =}

    reaction (receive) -> serve {=
        if self.pingsLeft > 0 {
            ctx.schedule(serve, Asap);
        } else {
            ctx.request_stop(Asap);
        }
    =}
}

reactor Pong(expected: u32(1000000)) {
    state expected(expected);

    input receive: u32;
    output send: u32;
    state count: u32(0);

    reaction(receive) -> send {=
        self.count += 1;
        ctx.set(send, ctx.get(receive).unwrap());
    =}

    reaction(shutdown) {=
        assert_eq!(self.count, self.expected);
    =}
}

main reactor SavinaPong(count: u32(1000000)) {
    ping = new Ping(count=count);
    pong = new Pong(expected=count);
    ping.send -> pong.receive;
    pong.send -> ping.receive;
}
```

Listing A.1: Source code for the Savina Ping Pong benchmark.

Acronyms

API	application programming interface. 1, 12, 14, 23, 33–35, 48–52, 58–62, 72, 74, 77, 82, 83, 88, 99, 105
AST	abstract syntax tree. 11, 13, 77, 82, 83, 104
CLI	command-line interface. 32, 80, 81
DAG	directed acyclic graph. 39, 40
DE	discrete-event. 15
DSL	domain-specific language. 30, 103, 105
FFI	foreign function interface. 12, 13
ID	identifier. 46–48, 51, 60, 61, 63–65, 73, 74, 93, 100, 101
IDE	integrated development environment. 11, 82, 103
IR	intermediate representation. 77, 104
JVM	Java Virtual Machine. 82
LF	Lingua Franca. ix–xi, 1, 4, 6–8, 10–15, 33, 34, 39, 40, 43, 44, 48, 53, 56, 58, 60, 62, 66, 69–71, 76–82, 87, 88, 94–97, 99, 100, 102–105
LSP	Language Server Protocol. 82
OOP	object-oriented programming. 13, 36
SR	synchronous/reactive. 15
STL	Standard Template Library. 13
UB	undefined behaviour. 23, 52

Index