Rusty Junctions: Rich Asynchronous Concurrency in Rust

Sebastian Müksch

s1511595@sms.ed.ac.uk

4th Year Project Report
Computer Science and Mathematics
School of Informatics
University of Edinburgh

2020

Abstract

This report introduces the *Rusty Junctions Library* [26], a library-level implementation of join patterns as originally described in the Join Calculus [31]. The Rusty Junctions Library is implemented in Rust version 1.35.0 with a design inspired by the Joins Concurrency Library [35], a previous implementation of join patterns. It contributes the *junction* as a novel coordination unit for concurrent computation that groups dependent join patterns together and a scheme to use Rust's static type system to validate join pattern constructions at compile time. An algorithm for scheduling join pattern execution is also developed with an outline of a proof of its *strong fairness*. Examples of the Rusty Junctions Library in use are provided. In particular, an existing solution [29] to the Santa Claus problem [36] in Polyphonic C# [30] is translated into the Rusty Junctions Library, demonstrating concurrent programming with multiple junctions interacting with each other as well as the Rusty Junctions programming model in general.

Acknowledgements

First and foremost, I would like to thank my supervisor Dr. Ian Stark for his unparalleled support with this project. It is no exaggeration to say that the contributions of this project would not have been possible without him. He went above and beyond to help me with some of the greatest challenges implementing join patterns in a programming language that is famously strict about its type system and ownership semantics. On top of that, I truly enjoyed the long meetings working on this project, which often stretched far beyond schedule. Thanks to him I have a new found appreciation for my subject.

I would also like to thank two dear friends of mine, Martin and Julius, who have taken up the daunting task of deciphering my work to provide valuable feedback on this report. Without them taking the time to look over my report, I would have been blind to my own mistakes. Martin in particular is to thank as he invested countless hours forcing me to write correct, clear and concise.

Lastly, I would like to thank my mother, father, sister and grandmother. It will hopefully end up as a distant memory, but the final weeks of this project happened under incredibly challenging and uncertain times, not just for me personally, but globally. While writing this report, I ended up moving back to my home country due to the events that were unfolding. Suffice it to say that it was not left unaffected by the reality that everyone was living in at the time. However, my family has been absolutely incredible in supporting me through the long nights of work that were needed to get this project done. I am incredibly fortunate to have them by my side.

The reader may excuse me potentially breaking standard procedure at this stage, but I would like to address the following paragraph to my family in their native tongue as not everyone will be able to understand it if I write it in English. Not to worry, nothing is lost if the following is not understood, but I want it to be here for them:

Pre mamku, für Papa, Alexandra und Oma. Ich kann euch nicht genug danken für all die Unterstützung über die Jahre hinweg während meines Bachelors, aber vor allem in diesen letzten Wochen, die durchaus nicht einfach waren. In der Wissenschaft spricht man oft davon, auf den Schultern von Giganten zu stehen. Ich schätze mich unglaublich glücklich auf den Schultern von den mir geliebten Menschen zu stehen.

Contents

Chapter 1

Introduction and Motivation

Many problems, such as the Santa Claus problem [36] either require concurrency, naturally lean to a concurrent formulation or can be solved more efficiently using concurrency. For this, many programming languages provide various mechanisms to share data across multiple entities and synchronise it when necessary. These mechanisms span from simple, mutually exclusive locks to message passing concurrency. However, with all of these, synchronisation has to be handled manually, with various problems to avoid, such as race conditions and deadlocking, that are potentially hard to detect. This begs the question of whether there a programming model that is built from the ground up for concurrent computation is possible. Such a model may not only be able to avoid such issues but offer a richer form of asynchronous concurrency.

The Join Calculus, developed by Fournet and Gonthier as a process calculus for distributed and mobile computation, offers such an approach [31]. The Join Calculus introduces *join patterns*, which build on message passing concurrency and offer a declarative way to synchronise various messages from arbitrary places within a system and execute processes as a result of successful synchronisation. Polyphonic C# [30] and the Joins Concurrency Library [35] are prominent examples of implementations of join patterns. These have been shown to be able to solve problems in concurrency that require synchronisation of many entities, such as the Santa Claus problem [36] [29], in a purely declarative manner and even be scalable to larger applications [37]. This suggests that join patterns may be a valid approach to concurrency in a more system-oriented programming language such as Rust [1].

The Rust programming language [1] is a system programming language [34] designed with a focus on safe concurrency. It offers constructs for not only locking but also message passing concurrency in its standard library. Coined as a safe system programming language with a rich, static type system [34], this report poses the question: *How can join patterns as a rich concurrency construct be implemented in Rust and integrated with its type system to provide a within Rust novel, declarative approach to safe concurrency?* Answering this question, this report lays out the design and implementation of the *Rusty Junctions Library* [26], a new and published library contributed to the Rust programming language.

The Rusty Junctions Library is a library-level implementation of join patterns in Rust version 1.35.0 that employs message passing and takes inspiration from the Joins Concurrency Library [35] in its design. It contributes a novel unit for coordination, the *junction*, which provides a natural grouping of join patterns that depend on each other. It uses Rust's type system to validate join pattern constructions at compile time and automatically ensure thread-safety. A further contribution of the Rusty Junctions Library is an algorithm to schedule the execution of processes after successful synchronisation in join patterns.

This report starts in Chapter 2 with an introduction into the theory behind join patterns, the Join Calculus [31]. It explores existing implementations of them and an gives an overview of important Rust features potentially new to a reader unfamiliar with the language.

Chapter 3 lays out the design of the Rusty Junctions Library with the concepts behind its components. Chapter 4 provides technical details of the their implementation, issues encountered along the way and solutions to them. Additionally, it provides the algorithm used to schedule the execution of processes after a synchronisation in join patterns.

Chapter 5 provides examples of the currently published version 0.1.0 of the Rusty Junctions Library [26] in use, most notably a description of a solution to the Santa Claus problem [36]. This solution is based on, and essentially is, a translation of a Polyphonic C# solution due to Benton [29]. This exemplifies that the expressiveness of the library is on par with that of Polyphonic C# for a reasonably complex problem such as the Santa Claus problem [36].

Finally, Chapter 6 provides a qualitative analysis of the Rusty Junctions Library. It describes its limitations and examines properties of the scheduling algorithm for join pattern execution it contributes. Chapter 7 closes with final remarks and pointers for future work.

Chapter 2

Background: Join Calculus, Existing Implementations and Rust

This chapter presents the necessary background material to understand the functionality and design that is implemented in the Rusty Junctions Library [26]. This chapter will also serve to put the Rusty Junctions Library into context in the field of Join Calculus and join pattern implementations.

Section 2.1 discusses the Join Calculus [31] as the underlying theory behind join patterns. It also works through an example of a problem solved using the Join Calculus to provide insight into its programming model.

Section 2.2 is dedicated to a discussion of existing implementations of the Join Calculus and join patterns with Polyphonic C# [30] and the Joins Concurrency Library [35]. Example code is provided for both with the latter having inspired the design of the Rusty Junctions Library [26].

Finally, Section 2.3 provides explanations and sample code for features of the Rust programming language [1]. The features covered are particularly important for the design and implementation of the Rusty Junctions Library as discussed in Chapter 3 and Chapter 4, respectively.

2.1 Join Calculus

The *Join Calculus* was designed as a process calculus for distributed and mobile computation by Cédric Fournet and Georges Gonthier [31]. Processes within the Join Calculus are synchronised with the use of *pattern matching*, where inter-process synchronisation is specified declaratively [31].

This section presents a selected part of the formal definition of the Join Calculus necessary to understand the examples that follow. The examples themselves demonstrate the programming model of the Join Calculus. Definition 2.1 gives this selected part of the description of the Join Calculus [31], for the complete formal definition refer to Definition A.1 in Appendix A:

Definition 2.1 Selected parts of the Join Calculus definition as given by Fournet and Gonthier [31].

P,Q,R	$::=$		processes
		$\dots$	
	$\parallel$	$P \mid Q$	parallel composition
	$\parallel$	$\mathbf{0}$	inert process
	$\parallel$	$\texttt{return}\ \widetilde{E}\ \texttt{to}\ f$	return value(s) to function call
	$\parallel$	$\texttt{def}\ f(\widetilde{x}) \triangleright P\ \texttt{in}\ Q$	recursive function definition
E,F	$::=$		expressions
		$\dots$	
	$\parallel$	$\texttt{def}\ D\ \texttt{in}\ E$	process/function definition
D	$::=$		definitions
		$J \triangleright P$	execution rule
	$\parallel$	$D \wedge D'$	alternative definitions
	$\parallel$	$\top$	empty definition
J	$::=$		join patterns
		$x\langle\widetilde{y}\rangle$	message send pattern
	$\parallel$	$x(\widetilde{y})$	function call pattern
	$\parallel$	$J \mid J'$	synchronisation

The *join patterns* in Definition 2.1 are key to the programming model employed by the Join Calculus. The *message send pattern*, $x\langle\tilde{y}\rangle$, is used to declare a *channel* on which a message can be sent. A corresponding *function call* using the *function call pattern*, $x(\tilde{y})$, can receive the message and use it [31]. These two patterns may then appear in a *synchronisation* using the pipe $\mid$ operator [31]. The synchronisation occurs when each pattern has sent at least one value [31].

The *execution rule*, $J \triangleright P$, in Definition 2.1 ensures that process P *guarded* by join pattern J is executed with the values received in J [31]. The guarded process is only executed upon synchronisation in the join pattern. Note that if there is at least one function call pattern in J, then the process P may contain a $\texttt{return}\ \widetilde{E}\ \texttt{to}\ f$ to return one or more values to a function call f.

To illustrate the use of the Join Calculus [31] and join patterns, a storage cell written in the Join Calculus is now given. The storage cell provides a mechanism that synchronises a single value in a concurrent environment. The value can both be set, denoted with `put`, and received, denoted as `get`.

For a value to be stored without a `put` and `get` to synchronise together, i.e. just passing the value over, a channel to carry the value is added. This channel is denoted with `val` and can be thought of as carrying state asynchronously. Example 2.1 uses it together with `put` and `get` to implement the storage cell:

```
1 def storageCell(a) ▷
2     def put(v) | val⟨w⟩ ▷ val⟨v⟩ | return to put
3         ∧ get() | val⟨v⟩ ▷ val⟨v⟩ | return v to get
4     in val⟨a⟩ | return put, get to storageCell
5 in  ...
```

Example 2.1: Storage cell written using Join Calculus.

Example 2.1 is one *recursive function definition*, `def` $f(\tilde{x})$ $\triangleright$ P `in` Q, as given in Definition 2.1. The place of $f(\tilde{x})$ is taken by `storageCell(a)` in line 1, which takes as parameter the initial value for the storage cell. In place of outer process Q are ellipsis in line 5 indicating an arbitrary process. The inner process P is split across lines 2 to 4 for increased readability and provides the actual implementation.

The inner process P is made up of a *function definition*, `def` D `in` R, as given in Definition 2.1. Note that the process is R to not repeat names. The definition D can be further split into two *alternative definitions*, $D' \wedge D''$ using the appropriate rule from Definition 2.1. Names are changed again for disambiguation. The first alternative definition, D', is given in line 2 of Example 2.1 and is of the form of an *execution rule*, $J \triangleright T$:

```
2 def put(v) | val⟨w⟩ ▷ val⟨v⟩ | return to put
```

Following Definition 2.1 in the above, the join pattern J is a synchronisation of a function call pattern, `put(v)`, and a message send pattern, `val⟨w⟩`. The process T is a parallel composition, $U \mid S$. Process U is simply another message send pattern, `val⟨v⟩` and process S is a function return to `put`, `return to put`. Altogether, line 2 states that if there is a function call to `put` with value `v` and channel `val` has sent value `w`, then send on channel `val` the newly received value `v` and return to the `put` with no value.

Second alternative definition, D'', is given in line 3 of Example 2.1:

```
3 ∧ get() | val⟨v⟩ ▷ val⟨v⟩ | return v to get
```

It can be broken down analogously to line 2 of Example 2.1. The join pattern in line 3 requires a function call to `get`, i.e. the value being requested, and a value `v` available through channel `val`. If both is the case, the guraded process resends that value using channel `val` to ensure that it can be received in future and returns value `v` to function call `get`.

Together, line 2 and 3 of Example 2.1 implement the functionality of the storage cell. The value can be set through `put` and received through `get`, as soon as a synchronisation with a value through `val` is possible. Note that this means that neither `put` nor `get` will return before synchronisation.

Finally, line 4 of Example 2.1 makes up the process R of `def` D `in` R, the function definition in the inner process P:

```
in val⟨a⟩ | return put, get to storageCell
```

Again, a parallel composition is used. The first process is `val⟨a⟩`, which sends the initial value of the storage cell as provided by `storageCell(a)`. Without an initial value sent, neither `put` nor `get` could synchronise and would never return or have an effect.

The second process, `return put, get to storageCell`, returns `put` and `get` to the caller of `storageCell`. This allows them to manipulate the value of the storage cell after it has been created. Observe that `val`, the channel carrying the value asynchronously, is not returned to the caller because a call to it directly has no meaning in this example. Channel `val` can be considered auxiliary or *private*.

Example 2.1 demonstrates key elements of the programming model employed by the Join Calculus. Among these are the join patterns that declaratively describe behaviour through the synchronisation of multiple channels. Additionally, there is the use of private channels to carry state asynchronously.

This section ends with another example of the Join Calculus programming model. This time a mechanism for mutual exclusion is implemented in Example 2.2:

```
def mutex() ▷
    def acquire() | lock⟨⟩ ▷ return to acquire
        ∧ release() ▷ lock⟨⟩ | return to release
    in lock⟨⟩ | return acquire, release to mutex
in  ...
```

Example 2.2: Simple mutex written using Join Calculus.

Example 2.2 follows the same structure as Example 2.1. Therefore, it is not broken down in detail. However, the reader is encouraged to reason it through themselves.

While Example 2.2 may be deceptively simple, it carries valuable proof that any program that can be written with mutual exclusion can also be written using the Join Calculus and join patterns, since the former can be implemented in the latter.

This concludes the description of the Join Calculus and its programming model. The next section moves on to present existing implementations of the Join Calculus and join patterns.

2.2 Existing Implementations

With the Join Calculus [31] providing the theoretical foundations for a programming model built for concurrency, various attempts of implementing it and its join patterns have been made. Notable ones are:

- *JoCaml*, which extends the Objective Caml language with, amongst other things, support for concurrency and synchronisation with a programming model based on the Join Calculus [32];

- *Polyphonic C#*, which extends the C# programming language with concurrency constructs based on the Join Calculus [30]; and
- the *Joins Concurrency Library*, which implements join patterns as a library on top of C# 2.0 using generics and the .NET runtime [35].

This section describes the work done on Polyphonic C# and the Joins Concurrency Library. They pose an interesting case study in implementation approaches which inspired the design of the Rusty Junctions Library [26] described in Chapter 3.

2.2.1 Polyphonic C#

Polyphonic C# provides a *language-level* implementation of join patterns, which it names *chords*. This means they are implemented as a concurrency primitive [30]. It precedes the Joins Concurrency Library [35], which instead offers join patterns as a library using generics in C#. These were not available at the time Polyphonic C# was developed [35].

To offer join patterns as a concurrency primitive, Polyphonic C# extends the normal function definitions of C# [30]. It allows for join pattern declarations in a form similar to synchronisations in Definition 2.1. For this, function signatures include multiple function names with parameter lists and return types [30]. This parallels channels in the Join Calculus, see Section 2.1.

The function names with their parameter lists and return types are separated by the `&` operator [30], which takes the place of the `|` operator in synchronisations in the Join Calculus, see Definition 2.1. A notable consequence of this implementation is that join patterns are declared *statically*, i.e. not during runtime.

Paralleling message send patterns in the Join Calculus, see Definition 2.1, Polyphonic C# allows functions to be declared as *asynchronous*. It does so by introducing a new return type: `async` [30]. This return type is treated as a subtype of `void`.

Polyphonic C# restricts each join pattern to only include *at most* one *non*-`async` return type [30]. Together with other restrictions [30], this guarantees the well-formedness of the language [30]. It also disambiguates the function that is returned to if the `return` keyword be used in the body of a chord [30].

Example 2.3 implements the same storage cell that example Example 2.1 implemented in the Join Calculus, but now in Polyphonic C# to illustrate its syntax:

```
class StorageCell {
    StorageCell(int initialValue) {
        Val(initialValue);
    }

    public async Put(int v) & async Val(int old) {
        Val(v);
    }

10     public int Get() & async Val(int v) {
11         Val(v);
12         return v;
13     }
14 }
```

Example 2.3: Storage cell written in Polyphonic C#.

The structure of the implementation in Example 2.3 is identical to Example 2.1. Lines 6 to 8 of Example 2.3 declare a join pattern which states that if a call to set the value is made using `Put` and there is a value available through `Val`, then send `Val` with the value provided by `Put`.

Likewise, lines 10 to 13 of Example 2.3 declare a join pattern which states that if a request to get the value has been made with `Get` and there is a value through `Val`, resend the value so it is available in future and return it as well. Note that `Get` has a non-`async` return type. It is therefore *synchronous*, i.e. blocks the calling thread until the join pattern is executed. It is also the function receiving the value.

Observe that the initial value is sent by the constructor of the `StorageCell` class in line 2 of Example 2.3. Additionally, the `Put` and `Get` methods of the `StorageCell` class are declared as `public`. This parallels returning the channels in Example 2.1. Each instance of `StorageCell` is able to call these methods to manipulate its value.

The use of a class to implement the storage cell in Polyphonic C# is noteworthy, as it logically groups dependent functions and join patterns together. As Example 2.1 demonstrates, this is not necessarily done in the Join Calculus [31].

2.2.2 The Joins Concurrency Library

The *Joins Concurrency Library* came after the introduction of generics to C# in C# 2.0 [35]. Contrary to Polyphonic C#, the Joins Concurrency Library does not alter the language to introduce join patterns as a concurrency primitive. It provides join patterns as an optional library instead [35].

The programming model around join patterns in the Joins Concurrency Library is similar to Polyphonic C#. One change is using `Channel` classes, not functions, to build join patterns. Additionally, these are built *dynamically*, i.e. during runtime.

Example 2.4 shows how these changes manifest by again implementing the storage cell from Example 2.1 using the same principles:

```
class StorageCell<T> {
    public readonly Asynchronous.Channel<T> Put;
    public readonly Synchronous<T>.Channel Get;
    private readonly Asynchronous.Channel<T> Val;

    public StorageCell(T initialValue) {
        Join j = Join.Create();
8
9           j.Init(out Put);
10          j.Init(out Get);
11          j.Init(out Val);
12
13          j.When(Put).And(Val).Do((v) => { Val(v); });
14
15          j.When(Get).And(Val).Do((v) => { Val(v); return v; });
16
17          Val(initialValue);
18      }
19  }
```

Example 2.4: Storage cell written using the Joins Concurrency Library.

Lines 2 to 4 in Example 2.4 declare the channels used before in Example 2.1. Lines 7 to 11 initialise an instance of the `Join` class and associate the channels with it. This is a required step by the Joins Concurrency Library [35].

Lines 13 and 15 in Example 2.4 construct the join patterns. The one in line 13 states that `When` the `Put` channel has sent a message `And` the `Val` channel has sent a message, `Do` send the new value through the `Val` channel. The other also follows the logic of Example 2.1. After constructing the join patterns, line 21 sends the initial value. Channels are accessible as public members of the `StorageCell` class.

The Joins Concurrency Library [35] has notably been used in research to demonstrate that join patterns are scalable as a concurrency mechanism [37]. The throughput of a system using join patterns is increased through careful thread usage [37]. For instance, the thread that provides the last message necessary to synchronise in a join pattern also executes the join pattern's body [37].

This concludes the section on existing implementations of join patterns. Polyphonic C# implements join patterns by extending the language. It allows them to be *statically* declared in the source code before compile time [30]. The Joins Concurrency Library provides join patterns as a library feature. This allows them to be *dynamically* constructed during runtime [35]. The next section describes features of the Rust programming language [1] which have influenced the design and implementation of the Rusty Junctions Library [26] as described in Chapter 3 and Chapter 4, respectively.

2.3 Rust

Rust [1] is a statically typed system programming language with a particular focus on safe concurrent programming [34]. Rust builds on research done in linear types, affine types and alias types [28]. Rust also builds on research in region-based memory management [28]. As a result, Rust supports automatic memory management without a garbage collector at runtime through its *ownership* model [28].

This section covers three features of Rust's type and memory management system, which shaped the design and implementation of the Rusty Junctions Library [26] as described in Chapter 3 and Chapter 4, respectively:

- Ownership and Borrowing
- Lifetimes on References
- Traits and Trait Objects

Note that a full introduction is beyond the scope of this report, however, the reader may find "The Rust Programming Language" [27] an invaluable resource. The reader familiar with the above topics may skip this section.

2.3.1 Ownership and Borrowing

Ownership is mentioned at the beginning of this section as enabling an automatic memory management without a garbage collector [28]. The "The Rust Programming Language" describes the rules for ownership as follows [27, §4.1]:

(i) Each value has a variable called the *owner* used to access it.
(ii) There can only be *one* owner at a time.
(iii) If the owner goes out of scope, the value is *dropped*, meaning it is deconstructed, for instance by freeing the allocated memory.

These rules are checked at compile time [27, §4.1]. To build an understanding of these rules, this section will provide examples demonstrating them. For these, heap-allocated strings [18] are used, as the concept of deconstructing them by freeing memory should be familiar to most readers. Note that boilerplate code is omitted where possible.

Example 2.5 defines a heap-allocated string `s`, then assigns it to a new variable `t` and attempts to print `s` to the standard output using the `println!` macro [17]:

```
1 let s = String::from("ownership"); // Define string.
2 let t = s;                         // Assign it to t.
3
4 println!("{}", s);                // Print s to standard out.
```

Example 2.5: Ownership violation.

However, Example 2.5 fails to compile due to violating ownership rule (ii) [27, §4.1]. It requires two variables to own the string `"ownership"`. This is because after line 2, `t` is the new owner of the string that previously `s` was owning. In line 4, despite having handed over ownership, `s` is then attempting to access the string value for printing.

If both `t` and `s` were allowed access to the string `"ownership"`, it would be ambiguous which variable is required to drop the value and free the memory taken up by the heap-allocated string. Replace `s` in line 4 with the actual owner of the string `"ownership"`, `t`, for Example 2.5 to compile.

Line 2 in Example 2.5 *moves* the value from `s` to `t`, invalidating access rights of `s` to the string thereafter. Moving values has important ramifications for function calls, as

Example 2.6 demonstrates:

```rust
1 fn f(t: String) {/* ... */} // Define function taking a String.
2
3 let s = String::from("ownership");
4 f(s);
5
6 println!("{}", s);
```

Example 2.6: Moving with function call.

Example 2.6 defines a function `f` with a string parameter. The rest is similar to Example 2.5, except for the call to `f` in line 4 instead of defining a new variable `t`.

As with Example 2.5, Example 2.6 does not compile due to it violating ownership rule (ii) [27, §4.1]. This time, however, the solution is not as simple and requires the introduction of two concepts: *references* and *borrowing* [27, §4.2].

Example 2.6 demonstrates that the ownership rules [27, §4.1] alone would make functions potentially difficult to work with. In this case by not being able to use the parameter after the function call. It would be desirable for `f` to only *borrow* ownership to `s`, not transfer it. For this, Rust has references and borrowing [27, §4.2], which Example 2.7 demonstrates:

```rust
1 fn f(t: &String) {/* ... */} // Take a reference as parameter.
2
3 let s = String::from("ownership");
4 f(&s);                         // Borrow s.
5
6 println!("{}", s);
```

Example 2.7: Function call with reference.

Example 2.7 is the same as Example 2.6, with two exceptions:

(1) The function signature of `f` has changed to expect a reference `&String` instead of a `String` parameter;
(2) the variable `s` is *borrowed* in the function call in line 4, allowing `f` to access its value but not owning it.

With the above changes, Example 2.7 compiles. Function `f` can access the value `s` owns, without issues relating to the ownership rules [27, §4.1]. There is, however, an important caveat with regards to references and borrowing: *mutability*.

By default, variables in Rust are *immutable* [27, §3.1], meaning the value that is owned by the variable cannot be changed after it is initially assigned. This default immutability also holds for references [27, §4.2]. Therefore, if function `f` in Example 2.7 tried to alter the value of `s`, which it borrows, the compiler would throw an error [27, §4.2]. The reference that `f` takes can be explicitly declared as *mutable*, meaning that `f` is permitted to change the borrowed value:

```rust
1 fn f(t: &mut String) {/* ... */} // Mutable reference parameter.
```

Rust imposes an important restriction on the use of mutable and immutable references together to prevent data races [27, §4.2]. At any given time, there can *either* be a *single* mutable reference *or any number of* immutable references [27, §4.2]. In essence, Rust imposes a read-write lock for referencing.

This covers the parts of Rust's ownership system [27, §4.1] necessary to understand the technical restrictions on the Rusty Junctions Library [26]. The next section addresses a concern yet left untouched, which is how *dangling references*, i.e. references to values already dropped, are prevented by the Rust compiler [27, §4.2].

2.3.2 Lifetimes on References

References, as introduced in the previous section, are used to borrow values. To ensure a reference is valid, i.e. accessing the data it is intended to, Rust introduces *lifetimes* [27, §10.3]. Lifetimes are the scope for which references are valid [27, §10.3]. This section presents examples to illustrate them, starting with Example 2.8:

```rust
fn f(s: &str, t: &str) -> &str { // Return reference.
    &s[0..1]                     // Get first character.
}
```

Example 2.8: Lifetime issue in function return.

The `&str` syntax in Example 2.8 denotes a *string slice*, a reference to part of a string [27, §4.3]. For the purposes of this example, `&str` can be considered a reference to a string similar to that in Example 2.7.

The function `f` in Example 2.8 returns a reference to the first character of parameter `s`. However, Example 2.8 does not compile because the Rust compiler cannot *automatically* determine the scope in which the reference returned by `f` is valid, i.e. its lifetime. Manually inspecting `f` reveals it can only be as long as that of parameter `s`.

Typically, the Rust compiler is able to infer lifetimes [27, §10.3]. However, in Example 2.8, this is not possible for the by `f` returned reference, as its lifetime is ambiguous. Without inspection of `f`'s body, it is unclear what it depends on [27, §10.3]. The solution is to explicitly add lifetimes to disambiguate the code as is done in Example 2.9:

```rust
fn f<'a>(s: &'a str, t: &'a str) -> &'a str {
    &s[0..1]
}
```

Example 2.9: Explicitly annotated lifetimes.

Example 2.9 adds the explicit lifetime parameter `'a`, declared using `<'a>` after the function name. It is used after the ampersands that define the references. Therefore, the lifetime of the reference returned by `f` is the same as the *smaller* of the lifetimes of the parameters [27, §10.3]. This is sufficient to consider the code safe and compile it.

One special lifetime that is important to mention is the `'static` lifetime [27, §10.3]. The `'static` lifetime denotes that the lifetime of a reference *could* be as long as the entire runtime of the program [27, §10.3]. For instance, this is the case for references to string literals [27, §10.3]. Another important use of this lifetime is in Rust's `thread::spawn` function that spawns new threads, given a function to execute [23]:

```rust
pub fn spawn<F, T>(f: F) -> JoinHandle<T> where
    F: FnOnce() -> T,
    F: Send + 'static,
    T: Send + 'static,
}
```

Example 2.10: Function signature of `thread::spawn` [23].

Example 2.10 has the function signature of Rust's `thread::spawn` function. It declares that the function executed in the spawned thread cannot have a lifetime shorter than `'static` [23].

The function signature in Example 2.10 also specifies that the function executed in the spawned thread needs to implement the `Send` trait [11], discussed in Section 2.3.3. Therefore, `thread::spawn` has the following restrictions on the kinds of functions executable in the spawned thread [23]:

(1) The function has to *either*
  - be moved to the new thread, i.e. ownership passed over; *or*
  - has to provide a reference with `'static` lifetime [27, §10.3][23]; *and*

(2) the function has to be able to transfer *safely* across thread boundaries [11][23], described in Section 2.3.3.

This concludes the exploration of Rust's lifetime system. In particular, the description of the `'static` lifetime as an important concept for threads. The next section explains Rust traits [27, §10.2] and trait objects [27, §17.2], highlighting an important subset of the traits provided by the Rust standard library.

2.3.3 Traits and Trait Objects

Traits [27, §10.2] in the Rust programming language [1] are used to declare and define shared behaviour for different data types [27, §10.2]. One example for this would be the ability to be printed to the standard output, which the `Printable` trait in Example 2.11 is meant to capture:

```rust
trait Printable {
    fn print(&self);              // Declare abstract behaviour.
}

impl Printable for String {
    fn print(&self) { /*...*/ } // Define behavior for String.
}

impl Printable for u32 {
    fn print(&self) { /*...*/ } // Define behavior for u32.
11 }
```

Example 2.11: Trait for abstract print behaviour, not part of Rust's standard library.

Firstly, Example 2.11 defines the `Printable` trait in lines 1 to 3. This includes in line 2 the declaration of the behaviour, i.e. function, a data type with the `Printable` trait must have.

In lines 5 to 7, Example 2.11 then defines an implementation of the `Printable` trait and its behaviour for strings (`String`). Lines 9 to 11 do the same for unsigned 32-bit integers (`u32`).

One application of traits is *trait bounds* [27, §10.2]. They allow for generic data types to be limited to a subset of types that have certain traits [27, §10.2]. The `display` function in Example 2.12 demonstrates this:

```
1 fn display<T>(what: T)
2 where
3     T: Printable, // Require the Printable trait.
4 {
5     what.print()  // Use Printable's behaviour.
6 }
```

Example 2.12: Trait bound for generic parameter.

In Example 2.12, the `display` function takes a parameter of generic type. However, in lines 2 and 3 it requires the parameter to implement the `Printable` trait from Example 2.11. This allows line 5 to use the `print` function from the trait, despite not knowing the exact type of the parameter before compile time.

The `thread::spawn` function [23] in Example 2.10 uses trait bounds extensively. They cause the requirements on the function executed in the spawned thread as described at the end of Section 2.3.2. In particular, the `Send` trait [11] in the trait bounds declares that a type is transferable across thread boundaries [11]. This means that it can safely be used by multiple threads. A type that would *not* fall into this category is, for instance, a reference counted smart pointer without additional synchronisation affords between the threads to ensure consistency [11].

Closely related to the `Send` trait [11] is the `Sync` trait [12]. It declares that a *reference* can safely be shared by multiple threads [12]. The relationship between the traits is that type `T` has the `Sync` trait *if and only if* the *reference* to the type, `&T`, has the `Send` trait [12]. Both traits are implemented automatically by the Rust compiler if deemed appropriate [11][12].

The `Clone` trait [6] declares a `clone` function to explicitly duplicate an object [6]. This can help avoid violations of the ownership rules discussed in Section 2.3.1. Example 2.13 below is Example 2.5 rewritten, solving the ownership violation by creating a duplicate that is separate from the original value, so ownership is *not* moved from `s` to `t`:

```
fn main() {
    let s = String::from("ownership");
    let t = s.clone();  // Create a duplicate.

    println!("{}", s);
}
```

Example 2.13: Cloning to prevent ownership violation.

The `Any` trait [4] is used to emulate *dynamic typing* in Rust [4]. Dynamic typing is achieved through the `Any` trait in conjunction with *trait objects* [27, §17.2]. Example 2.14 and Example 2.15 demonstrate dynamically typed collections using trait objects and the `Any` trait:

```
let mut v1: Vec<&u32> = vec![&1729];// Integer reference array.
v1.push(&1.618);                    // Compilation error!
```

Example 2.14: Array only allowing references to unsigned integer values (`&u32`).

```
let mut v2: Vec<&dyn Any> = vec![&1729]; // Trait object array.
v2.push(&1.618);                         // No error.
```

Example 2.15: Array allowing references to any type implementing `Any`.

Example 2.14 and Example 2.15 both use `Vec<T>`, a growable array [24], with generic type parameter for the stored values. Example 2.14 creates an array of unsigned integer references (`&u32`) in line 1. After that, the array can only store references to unsigned integers. Therefore, the compiler throws an error when line 2 attempts to insert a reference to a floating point number. This demonstrates that despite `Vec<T>` having a generic type parameter, it can only store values of one data type at a time.

Example 2.15, however, uses an array of trait objects with trait `Any`, denoted by `&dyn Any`. This means that it will hold references to values of types implementing the `Any` trait, which most types do [4]. Section 4.4 describes how this mechanism is used in the Rusty Junctions Library [26] to implement generic storage and functions of arbitrary parameter type.

This covers the features of the Rust programming language that heavily influence the design and implementation of the Rusty Junctions Library [26], described in Chapter 3 and Chapter 4, respectively. This also concludes the necessary background material. The next chapter lays out the design behind the Rusty Junctions Library [26].

Chapter 3

Design: Channels, Junctions and Controllers

With the previous chapter providing background material on join patterns and existing implementations, this chapter lays out the design of the Rusty Junctions Library [26] inspired by these.

Section 3.1 starts with an overview of the components that make up the Rusty Junctions Library [26]. Section 3.2 then provides insight into channels available in the library as well as their individual purpose. Section 3.3 follows with the design of join patterns. In Section 3.4, the *junction* is discussed as contribution of the Rusty Junctions Library [26] to group dependent join patterns. The chapter closes with Section 3.5 by exploring the *controller* that manages junction state in the background.

3.1 Overview

The Rusty Junctions Library [26] implements join patterns on a library level. Inspired by the Joins Concurrency Library [35], discussed in Section 2.2.2, it adds join patterns as an optional concurrency construct to Rust [1].

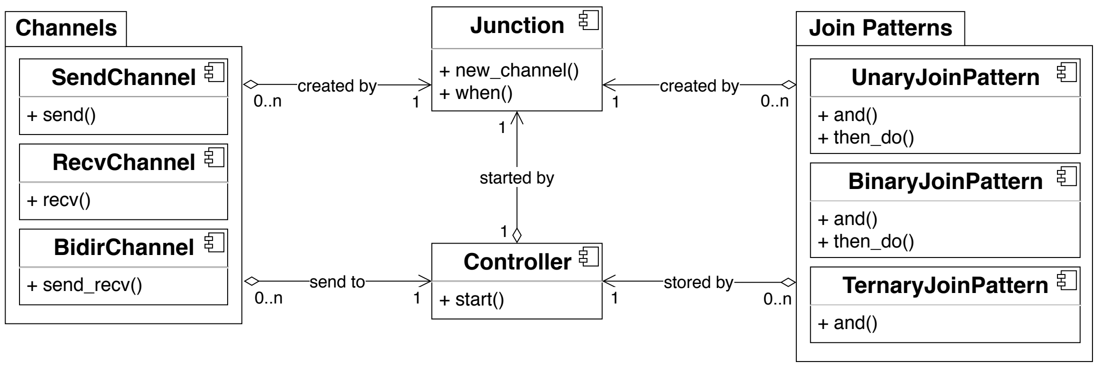

Figure 3.1: Overview of the components of the Rusty Junctions Library [26].

The Rusty Junctions Library [26] also implements generic channels similar to those in the Joins Concurrency Library [35]. These are used to construct join patterns *at runtime*, which is discussed in Section 4.3.

Furthermore, the Rusty Junctions Library [26] contributes a new unit for coordination, the *junction*, suggested by the project supervisor. A junction is a component that groups together dependent join patterns and the channels used for their construction.

The junction interacts with another component introduced by the Rusty Junctions Library [26], the *controller*. A controller handles messages sent by channels and stores join patterns. It also schedules the execution of join patterns based on the messages it receives. Together with the junction, it provides the local synchronisation and contention properties described in the Join Calculus [31].

Figure 3.1 summarises the design components of the Rusty Junctions Library. The next section provides more detail on the channels in the Rusty Junctions Library [26].

3.2 Channels

The Rusty Junctions Library [26] follows the Join Calculus [31] in using message passing to manage concurrency. For this, it provides three types of channels that can be synchronised in join patterns:

- `SendChannel`, an *asynchronous* channel to send values;
- `RecvChannel`, a *synchronous* channel to receive values;
- `BidirChannel`, a *synchronous* channel to both send and receive values.

The rule for channels in the Rusty Junctions Library [26] is that if they receive, they are synchronous. Otherwise, they are asynchronous.

Note that none of these channels are directly connected. The values a `SendChannel` sends are not directly received by a `RecvChannel`. Rather, the values sent are received by a controller, described in Section 3.5. A successful execution of a join pattern involving the channel may then send a value to a `RecvChannel`.

Figure 3.1 shows that each channel type is created by a junction. This links the channel to that particular junction. A channel can only ever be linked to a *single* junction, which ensures contention remaining local to that junction.

Figure 3.1 also shows that each channel implements a different method. For instance, the `SendChannel` implements a `send` method while the `RecvChannel` implements a `recv` method.

3.2.1 Asynchronous Sending with `SendChannel`

The `SendChannel` is designed as an asynchronous channel to send messages without blocking the current thread. They have a single generic type parameter that determines the type of messages sent, similarly to the Joins Concurrency Library [35]. The messages are sent using the `send` method, which takes the message as a parameter and does not return a value. Note that the parameter is *moved*, see Section 2.3.1.

Sending a message through a `SendChannel` *may* cause a join pattern which includes this channel to be executed. However, this execution *never* returns a result to this type of channel or notifies it of the execution.

This channel type parallels asynchronous channels in the Joins Concurrency Library [35], see Section 2.2.2. It also parallels functions declared with the `async` return type in Polyphonic C# [30], see Section 2.2.1.

The difference between a `SendChannel` and a function with `async` return type in Polyphonic C#, however, is that the `send` method of this channel always requires a value. Functions in Polyphonic C# are allowed to have an empty parameter list. This can be emulated in `SendChannel` by using Rust's `()` unit type [2] for the message type and passing `()` as the message.

3.2.2 Synchronous Receiving with `RecvChannel`

The `RecvChannel` is designed as a synchronous channel to receive results from a join pattern execution. If a join pattern is using a `RecvChannel`, it is automatically required to have a return value. The `RecvChannel` has a single generic type parameter that determines the type of value received and hence the return type of the `recv` method.

The result of a join pattern execution is sent to the `recv` method and returned by it. The `recv` method itself does not take any parameters. Calling the `recv` method sends a message to request a result which *may* cause a join pattern which includes the channel to be executed. However, the `recv` method is *guaranteed* to block the thread until such an execution has happened.

This channel type parallels synchronous channels in the Joins Concurrency Library [35], see Section 2.2.2. It also parallels functions declared with non-`async` return types in Polyphonic C# [30], see Section 2.2.1.

The difference between a `RecvChannel` and a function with non-`async` return type in Polyphonic C#, however, is that the `recv` method can *never* take a parameter and will *always* have a return value. A `void` return value of a Polyphonic C# function can be emulated by using Rust's `()` unit type [2] as the type of the channel.

3.2.3 Synchronous Sending and Receiving with `BidirChannel`

The last channel type provided by the Rusty Junctions Library [26] is a synchronous channel for sending *and* receiving messages, i.e. bidirectional communication. A `Bidir` channel has two generic type parameters, the first for the type of value sent, the second for the type of value received.

In general, a call to the `send_recv` method of a `BidirChannel` can be viewed as an appropriate call to the `send` method of a `SendChannel` and a subsequent call to the `recv` method of a `RecvChannel`. Therefore, what Section 3.2.1 and Section 3.2.2 describe holds for the `BidirChannel` and its method.

However, there is one crucial difference between a call to `send_recv` and a call to `send` followed by a call to `recv`. The `send_recv` call is *atomic*, i.e. treated as a single message. In a concurrent environment, there may be an arbitrary amount of messages sent between a `send` call and an in code following `recv` call. Therefore, using these two together cannot guarantee a relationship between the state of the system at the time of the `send` call and at the time of the `recv` call.

An example where such a guarantee is *required* is for swapping values in the storage cell in Example 2.1. With an arbitrary amount of messages between a call to update and a call to receive the value, there is no way of guaranteeing that the value received is actually the value updated. The `send_recv` method gives such a guarantee.

This channel type parallels a function with a single parameter and a non-`async` return value in Polyphonic C# [30]. The next section described the design of join patterns in the Rusty Junctions Library [26].

3.3 Join Patterns

Join patterns, as defined by the Join Calculus [31], see Definition 2.1 for reference, consist of a pattern of channels that may guard a process to be executed upon synchronisation of the channels in the pattern [31]. Therefore, an implementation requires:

- a mechanism to declare and store the pattern of channels *and*
- a mechanism to declare and store the processes that may be guarded by a join pattern.

Polyphonic C# uses function signatures for the pattern declaration and a single function body for the process declaration, see Section 2.2.1. The Joins Concurrency Library on the other hand uses API calls to declare a pattern, see Section 2.2.2, and a system of classes to represent and store them [35]. A similar approach is taken with the Rusty Junctions Library [26].

As Figure 3.1 shows, join patterns in the Rusty Junctions Library are split into various specialised types, for instance `UnaryJoinPattern` or `BinaryJoinPattern`, which are differentiated by the number of channels synchronised with the join pattern. If a join pattern has only one channel declared in its pattern, it is referred to as *unary* and represented by a `UnaryJoinPattern`. If a join pattern has two channels declared in its pattern, it is referred to as *binary* and represented with a `BinaryJoinPattern`. This nomenclature is extended to join patterns with three or more channels in their patterns.

Contrary to the Join Calculus [31], join patterns in the Rusty Junctions Library always guard a process that is executed upon synchronisations of the channels in the pattern, that is, if each of them has sent at least one message. The processes on the other hand are represented by Rust's *closures* [27, §13.1]. Since Rust allows for a closure with empty function body, which can be interpreted as an inert process **0** in the Join Calculus, see Definition 2.1, the restriction of always guarding a process does not fundamentally change the join pattern mechanic as used by the Join Calculus [31]. This approach to processes is inspired by the Joins Concurrency Library [35], see Section 2.2.2, as C#'s lambda expressions share similarities with Rust's closures.

Any of the three types of channels described in Section 3.2 may be used in the declaration of a join pattern. However, as Section 3.1 described, the restriction of allowing at most one synchronous channel per join pattern is imposed. Following the design of the channels as described in Section 3.2, this implies the following rules for join patterns in the Rusty Junctions Library [26]:

(i) A join pattern may consist of *one or more* `SendChannel`s; and
(ii) a join pattern can include *at most one* `RecvChannel` or `BidirChannel`, but *not both*.

These rules allow the Rusty Junctions Library [26] to uniquely determine the type of the closure attached to the join pattern as the guarded process. Section 4.3 describes in more detail how this design is realised using Rust's type system and how join patterns are validated by the compiler.

Finally, the API calls to construct a join pattern follow closely the example set by the Joins Concurrency Library [35], see Section 2.2.2. Example 3.1 showcases a simple construction of a join pattern associated with a junction instance `j` that uses two channels instances `ch_1` and `ch_2`:

```
j.when(&ch_1).and(&ch_2).then_do(|a,b| { /*...*/ })
```

Example 3.1: API calls to construct a join pattern.

Section 4.3 goes into the technical realisation of the design presented in Example 3.1. In the next section details the role of the junction in the Rusty Junctions Library [26].

3.4 Junctions

The *junction*, as suggested by the project supervisor, is designed as a unit for coordination that groups dependent join patterns together. One example would be the join patterns that make up the entire behaviour of a system, such as in the storage cell in Example 2.1. Furthermore, the junction is also designed to group together the channels that are used in its join patterns.

To provide the aforementioned grouping of channels and join patterns, junctions are designed to be the only way of creating new channels and constructing new join patterns, as can be seen in Figure 3.1. Any channel or join pattern created with a junction is *automatically* and *exclusively* associated with it. Channels or join patterns associated with a junction cannot be disassociated with it. The association of channels to junctions is designed in such a way that it is not possible to construct join patterns associated with one junction using channels associated with another. Section 4.2 and Section 4.3 describe the technical details of how channels are created and join patterns constructed using junctions, which also shows the realisation of the designed association.

The concept of a junction is comparable to classes in both Polyphonic C# [30] and the Joins Concurrency Library [35], described in Section 2.2.1 and Section 2.2.2, respectively. However, examining Example 2.1 and Example 2.2 written in Join Calculus, it appears that a grouping of channels and join patterns is somewhat implicit in the recursive function definitions, so a junction can be interpreted as the natural continuation of that implicit grouping.

Together with the controller described in the next section, junctions provide the local synchronisation and contention properties described in the Join Calculus [31]. The technical implementation of the features that junctions provide is actually located in controllers, the design of which is detailed next.

3.5 Controllers

As Figure 3.1 shows, every junction starts a single *controller*, which is designed to be the main component carrying the state of the junction, i.e. all messages that have been sent as well as the join patterns that have been constructed. Figure 3.1 also shows that channels actually send messages not to the junction they are associated with, but to the controller started by the junction they are associated with.

On the suggestion of the project supervisor, the controller serves two purposes:

(i) Separate the concerns of creating channels and join patterns and managing them; and
(ii) provide a single place in which the state necessary for a junction to operate is located.

Purpose (ii) is of particular importance for a concurrent environment. Many threads may have channels sending messages and join patterns can be ready to be executed at any time. Therefore, there needs to be a component managing all this, which necessarily cannot reside in any of the same threads as the junction or the channels. One reason is that the process of handling messages and executing join patterns may take up most of the computational time in a thread and another is that some threads may simply not live long enough. The solution to this used in the Rusty Junction library [26] is to have a controller started in a new thread by every new instance of a junction.

A more subtle but vital reason to have a controller run in a thread separate from both the junction and any of the channels associated with it, is the overhead of shared state. Sharing state, such as messages that have been sent or join patterns that have been constructed across multiple threads inevitably incurs additional synchronisation cost. This is not the case with the controller, as all state is kept in a single thread, so state changes, such as the addition of a new join pattern, can be handled as trivially as an insertion into a data structure. Section 4.5 will go into the details of the implementation of the controller and state handling.

It is worth pointing out that the controllers share some resemblance with *reflexive chemical abstract machines* or RCHAMs for the Join Calculus as described by Fournet and Gonthier [31], which also carry some form of state to support join patterns and their execution. However, the Rusty Junctions Library [26] is not built based on RCHAMs because it was designed more with the Joins Concurrency Library [35] as an existing library-level implementation in mind.

This concludes the chapter on the design of the components of the Rusty Junctions Library [26]. It describes the use of channels to send and receive values to execute join patterns in Section 3.2 and details the design of join patterns that synchronise said channels in Section 3.3. Section 3.4 provides insight into the central component of the Rusty Junctions Library [26], the junction, a unit for synchronisation that groups dependent channels and join patterns together. Finally, this section presents the controller as the managing component for join patterns to work. The next chapter details the actual implementation of the Rusty Junctions Library [26] as well as the technical difficulties encountered along the way and together with their solutions.

Chapter 4

Implementation: Bringing Rusty Junctions To Life

The previous chapter presented the design of the Rusty Junctions Library [26]. This chapter will focus on the technical details of the actual implementation of said design and the challenges that needed to be addressed for it. Version 1.35.0 of the Rust programming language is used for this implementation.

This chapter starts with an explanation of the message passing concurrency constructs used in the Rusty Junctions Library [26] in Section 4.1. After that, it describes how channels are created and join patterns constructed in Section 4.2 and Section 4.3, respectively. Then it lays out how dynamic typing for the state in the controller in Section 4.4 is implemented, with the state itself being described in Section 4.5. The algorithm to manage join pattern execution is elaborated on in Section 4.6. Finally, the chapter closes with a description of the `ControllerHandle` as a resource-managing construct in Section 4.7.

4.1 Message Passing with Packets

Section 3.2 presents various channels and describes how the Rusty Junctions Library [26] employs message passing concurrency in line with the Join Calculus [31]. However, message passing is in fact used for every aspect of the Rusty Junctions Library, from creating channels, to storing join patterns after they have been constructed, down to implementing the actual channels from Section 3.2. For this, the Rusty Junctions Library makes extensive use of existing channels in the Rust standard library: `Sender` [22] and `Receiver` [21].

Both `Sender` [22] and `Receiver` [21] are part of Rust's `mpsc` library, which implements *multiple-producer, single-consumer* communication primitives [20]. They are effectively two parts of a channel with a single generic type parameter. The `Sender` is the send-only end of the channel that implements the `Send` [11] and `Clone` [6] traits, see Section 2.3.3 and can thus be duplicated and passed to other threads. This is the multiple-producer part. The `Receiver` is the receive-only end of the channel which implements the `Send` [11] but *not* the `Clone` [6] trait. Therefore, a `Receiver` can be passed to other threads but never duplicated, so it is single-consumer.

On top of the restriction of being single-consumer, there is no standard way of multiplexing different `Receivers` together. This is why the Rusty Junctions Library does not directly use Rust's `Sender` and `Receivers` as channels but puts a layer of abstraction on top of them. Each channel described in Section 3.2 has a `Sender` to which the controller has the corresponding `Receiver`. This is how the sending depicted in Figure 3.1 from channels to controller is realised: the controller owns the single consumer of all of the channels' producers and simply loops through the messages.

However, there is one flaw with the aforementioned implementation: `Sender` and `Receiver` can only send and receive values of a single type [22] [21]. To avoid each channel in the Rusty Junctions Library having to have the same type or implementing a non-standard multiplexer over an arbitrary amount of `Receivers`, the Rusty Junctions Library implements the `Packet` type on suggestion of the project supervisor.

The `Packet` type is implemented as a generic payload that can be handled by the controller once received. The `Packet` type is implemented using Rust's `enums` [27, §6]. This allows each variant of a `Packet` to be listed and to be pattern matched against [27, §6.2]. Example 4.1 below presents a simplified version of the `Packet` type definition with only very low-level implementation details left out:

```
1 enum Packet {
2     Message,                  // Sent by channels.
3     NewChannelIdRequest,      // For channel creation.
4     AddJoinPatternRequest,    // To store join patterns.
5     ShutDownRequest,          // To shut down controller.
6 }
```

Example 4.1: Simplified definition of the `Packet` type.

Lines 2 to 5 in Example 4.1 are the variants of the `Packet` type. The variant relevant for channels sending messages to the controller is `Message` in line 2. There is further complexity with respect to dynamic typing hidden behind this variant, to which Section 4.4 is dedicated. However, the important point is that the `Message` variant allows every channel to have a `Sender` of type `Packet` and still have a unique type of value sent over it.

With the `Packet` type established, more tasks than channels sending messages can now be accomplished through message passing, described in the following sections:

- Line 3 in Example 4.1 defines a variant that is used by the junction, which also has a `Sender` of type `Packet`, when creating a new channel, see Section 4.2;
- line 4 defines a variant used by the junction to request for a new join pattern to be managed by its associated controller, see Section 4.3;
- line 5 defines a variant used to gracefully shut down the controller and the thread it is running in, see Section 4.7.

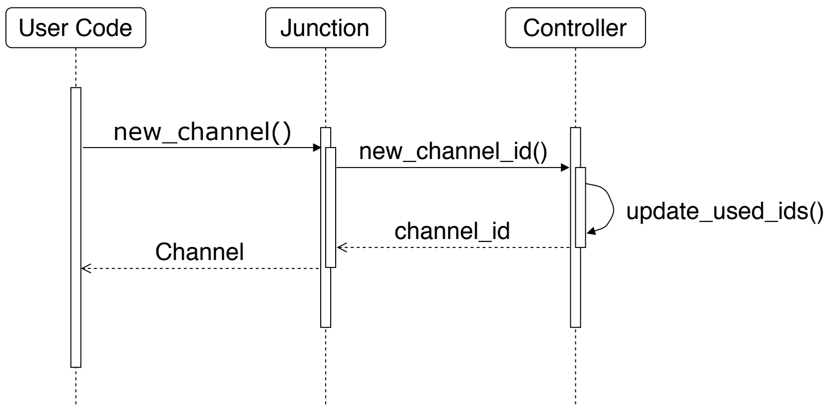

Figure 4.1: Simplified overview of the process of creating a new channel from user code request to controller state update.

4.2 Creating Channels

As Section 3.4 described, junctions are responsible for creating new channels. However, to explain the implementation of the process by which a junction creates any of the three channels described in Section 3.2, unique channel identification in the Rusty Junctions Library [26] has to be discussed.

For the join pattern execution algorithm implemented in the Rusty Junctions Library, described in Section 4.6, messages sent channels are required to uniquely identify the channel they have been sent by. The type of value a channel sends is, however, insufficient for this on its own, as there may be multiple channels sending messages of the same type. The `ChannelId` type solves this problem, implemented as a sufficiently large unsigned integer, which represents a channel ID *unique* at least on the level of a single junction, i.e. no two channels in one junction will have the same channel ID, but two channels in two different junctions may. Note further that the Rusty Junction library does not differentiate between the three channel types presented in Section 3.2 when it comes to `ChannelId`s [26].

To ensure that in each junction all `ChannelId`s are unique, state is added to the associated controller to keep track of the still available `ChannelId`s in a centralised place. For simplicity, the controller stores the latest given `ChannelId` and increments it every time a new one is needed.

If some user code requests a new channel of any of the three types described in Section 3.2 from the junction, the junction will send a request for a new `ChannelId` to its associated controller. This request is sent through a `NewChannelIdRequest` variant of the `Packet`, see Example 4.1. Along with the request, the junction will send one of Rust's `Senders` [22] which the controller will use to send back the requested `ChannelId` so that the junction can use the `Receiver` [21] to get it. See Section 4.1 for more information on Rust's `Sender` [22] and `Receiver` [21].

Figure 4.1 depicts a simplified version of the channel creation process. The two key steps now shown are:

- The use of specific functions to request either one of the three types of channels described in Section 3.2;
- the junction adding its own ID to the new channel to associate it with the junction.

Any junction has a globally unique ID itself, implemented with the `JunctionId` type as sequentially consistent unsigned integer using Rust's `AtomicUsize` [19]. By adding the junction's `JunctionId` to any newly created channel, the channels are associated with the junction as described in Section 3.4.

The use of `ChannelIds` and `JunctionIds` does not provide a unique but relatively simple solutions to their respective problems. Conceiving a way to use Rust's type system to give each channel a completely unique type for identification was attempted, but did not appear possible in the Rust version 1.35.0, used for the Rusty Junctions Library [26]. References to junctions in channels as a means of associating the two were also considered, but discarded to avoid potential lifetime issues, see Section 2.3.2.

After the description of how channels are created in the Rusty Junctions Library [26] and how they identify themselves using `ChannelIds`, the next section covers join pattern construction.

4.3 Constructing Join Patterns

The Rusty Junctions Library [26] implements the join pattern construction with a mechanism inspired by the Joins Concurrency Library [35] as described in Section 3.3. Example 3.1 shows how the construction starts with a junction calling the `when` method and providing a channel, then adding another channel to the pattern with `and` and finally providing a closure [27, §13.1] as the process to be run when the join pattern is executed. This section details how this construction and the constraints posed on join patterns in the Rusty Junctions Library described in Section 3.3 are, in fact, verified entirely by the Rust compiler through the use of Rust's type system.

Section 3.3 describes how join patterns in the Rusty Junctions Library are subdivided depending on the number of channels involved in them. The constraints on join patterns as restated from Section 3.3:

(i) A join pattern may consist of *one or more* `SendChannels`; and
(ii) a join pattern can include *at most one* `RecvChannel` or `BidirChannel`, but *not both*,

are implemented by further subdividing join patterns into more specific types depending on the type of channels involved in them. For this, one simplifying assumption is made: a `RecvChannel` or `BidirChannel` can only ever be added as the *last* channel in a join pattern. While this breaks with the Join Calculus definition of join patterns, see Definition 2.1, which does not require a certain ordering of the channels [31], the Rusty Junctions Library gains the following properties as a trade-off:

(i) Join patterns are *unique*, up to a reordering of the `SendChannels` and

(ii) the number of join pattern subtypes does not depend on the length of the join pattern.

Property (i) increases the readability and maintainability of code written with the Rusty Junctions Library [26] as the channel that will receive a return value from the join pattern execution will always be in the same place: at the end of the pattern. For an explanation of the use of property (ii), the join pattern types implemented in the Rusty Junctions Library [26] have to be explained first. However, note that channels being non-commutative within a join pattern does not decrease their expressiveness.

The construction of join patterns as described in Section 3.3 always starts with the pattern of channels involved and finishes with a call to the `then_do` method that adds a closure [27, §13.1], to be executed, to the join pattern. Therefore the Rusty Junctions Library [26] distinguishes between a `PartialPattern`, which is a pattern of channels that has not yet been associated with a closure [27, §13.1] and a `JoinPattern`, which has. Furthermore, it distinguishes:

- `SendPartialPattern` and `SendJoinPattern`, if the pattern of channels ends in a `SendChannel`,
- `RecvPartialPattern` and `RecvJoinPattern`, if the pattern of channels ends in a `RecvChannel`,
- `BidirPartialPattern` and `BidirJoinPattern`, if the pattern of channels ends in a `BidirChannel`.

Each of the above distinctions is well-defined as `RecvChannels` and `BidirChannels` can only appear last in a join pattern. Furthermore, each of the above types is implemented for any length of join patterns, i.e. unary, binary, and so on. It should now be apparent that without the constraint of `RecvChannels` and `BidirChannels` only appearing last, a length n join pattern may choose 1 place for a `RecvChannel` or `BidirChannel` and so distinguish between $\binom{n}{1} + \binom{n}{1} + \binom{n}{0} = 2n + 1$ different join pattern types. Note that the last binomial coefficient stems from the possibilities for join patterns with only `SendChannels`, ignoring reordering. Thanks to the constraint on `RecvChannel` and `BidirChannel`, this number is just 3.

With all of the join pattern types explained, Figure 4.2 represents the join pattern construction as a finite state machine. The transitions are the method calls to add either another channel or the closure [27, §13.1] at the end. Methods `when` and `and` add a `SendChannel`, their counterparts ending in `_recv` and `_bidir` add a `RecvChannel` or a `BidirChannel`, respectively. The finite state machine in Figure 4.2 shows that a `RecvChannel` or a `BidirChannel` can only be added once and exclusive of one another, validating the constraints described in Section 3.3 using entirely the Rust type system at compile time.

The validation of join patterns through the Rust type system goes one step further. All `PartialPatterns` carry type information of the types of values their channels send and in the case of `RecvChannel` and `BidirChannel` receive, all in order of declaration. By the time a call to `then_do` is made, the type of the closure [27, §13.1] that can be passed to it is fully determined and checked by the compiler.

One final constraint from Section 3.3 needs to be validated: only channels associated

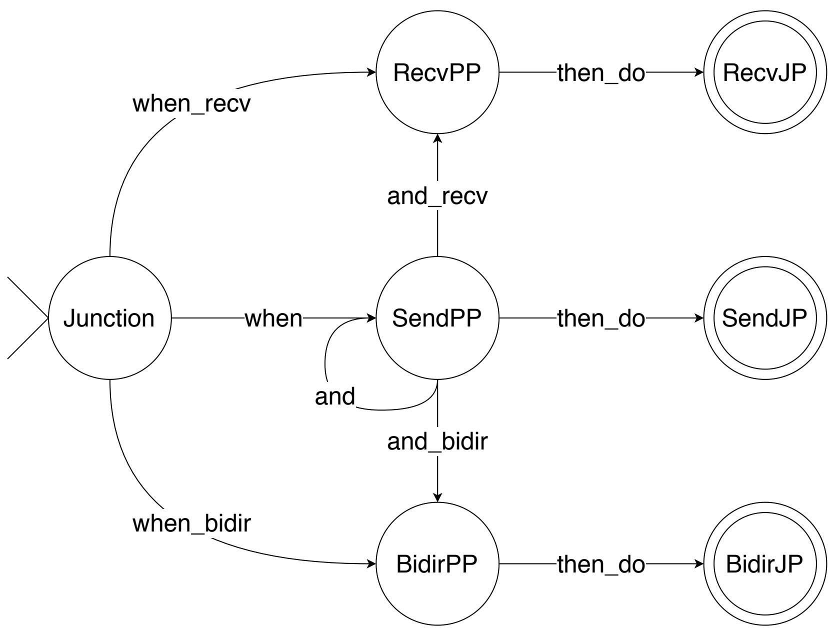

Figure 4.2: Finite state machine representing type transitions during join pattern construction. Start state is *Junction*, accepting states are *RecvJP*, *SendJP* and *BidirJP*, where *PP* stands for `PartialPattern`, *JP* stands for `JoinPattern`.

with the junction instance that started the join pattern construction can be used in the join pattern. As channels associate with a junction through the `JunctionId`, see Section 4.2, this has to be verified at runtime. If the `JunctionId` of the channel and junction do not match, the thread *panics* which terminates the program and provides error feedback [16]. A more graceful way of signalling this error is not supported by the current version 0.1.0 of the Rusty Junctions Library [26].

Once a join pattern has been constructed successfully, i.e. all constraints described in Section 3.3 have been verified, the closure [27, §13.1] associated with the join pattern is transformed, see section Section 4.4. After that, the join pattern and transformed closure are send to the controller through a `AddJoinPatternRequest` variant of the `Packet`, see Example 4.1, to be stored there. How this is done is the topic of the Section 4.5. However, first an explanation of dynamic typing in the Rusty Junctions Library [26] is given in the next section.

4.4 Dynamic Typing Through Trait Objects

Before an explanation of how state management is implemented in the controller in Section 4.5, this section discusses how storing values of an essentially arbitrary amount of types is implemented efficiently in the Rusty Junctions Library [26]. There are two aspects of the library where this is relevant:

- Messages sent by channels and
- closures [27, §13.1] associated with join patterns.

These two are in fact related. The parameter list and return type of the closures [27, §13.1] associated with join patterns are directly derived from the types of values the channels in the join pattern send and possibly receive, see Section 4.3.

For messages sent on channels, a `Message` type is implemented, which makes use of the trait object mechanics in Rust, see Section 2.3.3. Instead of using references, see Section 2.3.1, the `Message` type in the Rusty Junctions Library [26] uses Rust's `Box`, which is a pointer type to heap allocate values [5].

The `Message` type is implemented as a `Box` pointer [5] of a trait object [27, §17.2] that uses Rust's `Any` [4] and `Send` [11] traits, see Section 2.3.3 for details on these traits. This creates a level of indirection in the storage of the value through the heap, which, thanks to the traits used, allows a large range of values to be stored as a `Message` [4] *and* ensures that they can be safely sent to the controller running in its own thread [11], which was described in Section 3.5.

As for the closures [27, §13.1] associated with join patterns, with the help of the project supervisor, function transforms were developed that change the parameter and return type signature of any given closure into one accepting `Messages` as parameters and no return value. Example 4.2 provides a simplified definition of such a function transform:

```
1 fn transform<F, T>(f: F) -> Box<impl FnBoxClone>
2 where
3     F: Fn(T) -> (),
4 {
5     // Heap-allocate outer closure.
6     Box::new(|arg: Message| {
7         f(arg.cast::<T>());         // Cast Message at runtime.
8     })
9 }
```

Example 4.2: rust-book.]Simplified definition of a function transform that changes the type signature of a closure [27, §13.1].

The function transform in Example 4.2 accepts a closure [27, §13.1], henceforth referred to as inner closure, with a single parameter of generic type parameter `T` and no return value. The type signature of the inner closure is declared in line 3. It creates an outer closure in lines 6 to 8 that accepts a `Message` parameter and is stored on the heap through `Box` in line 6. When the outer closure is called with a `Message`, it calls the inner closure after casting the message parameter to the type that the inner closure expects. All of this is done at runtime, but because Rust is compiled, the exact types to cast to are determined at compile time and since they are based directly on the types expected by the inner closure, this approach is sound.

Should a closure have a return type, which is the case if the join pattern includes a `RecvChannel` or `BidirChannel`, Rust's `Sender` [22] and `Receiver` [21] are used to deliver the return value to the appropriate channel. The `Sender` that will send the return value of the inner closure is made a parameter of the outer closure, which then adds the extra step of sending the return value of the inner closure.

The return type `Box<impl FnBoxClone>` in line 1 of Example 4.2 essentially declares that the outer closure, that is returned, is a function that can be cloned, see the `Clone` trait explanation in Section 2.3.3. This is an implementation detail that Section 4.6 will explain.

With the ways in which the Rusty Junctions Library [26] uses trait objects [27, §17.2] to store values of almost arbitrary types in a statically typed language [34] presented, the next section moves on to explaining state management in the controller.

4.5 Handling State with Controllers

Section 3.5, describes that the controller component in the Rusty Junctions Library [26] is designed to manage state necessary for channels and join patterns to function. This section details the state that resides in a controller and how it is mutated to handle new messages from channels as well as join pattern execution.

The two core data collections in the controller are:

- A bag of messages that allows for previously sent messages from *any* channel, associated with the same junction as the controller, to be retrieved by `ChannelId` and
- a hash table that stores join patterns from the associated junction with a unique identifier.

The bag of messages is implemented with a `Bag` collection specifically developed for the Rusty Junctions Library [26]. Note that Rust's standard collections [7] to not support the required collection and relying on external and potentially unstable code is favourable. The `Bag` collection stores values associated with a key in a FIFO queue using Rust's `HashMap` [8] and `VecDeque` [10].

For the bag of messages in the controller, the values are of type `Message`, making use of dynamic typing as described in Section 4.4. The keys are of type `ChannelId`, representing the ID of the channel that sent the messages. This allows the controller to retrieve, for any channel in a join pattern, the message that has been waiting the longest to be consumed since `Bag` uses a FIFO queue. The `Bag` also implements functionalities to count how many messages for a given `ChannelId` have been received by the controller or simply check if any have been or not.

The hash table storing the join patterns is implemented with Rust's `HashMap` [8] and a new identifier type: `JoinPatternId`. The `JoinPatternId` has the same uniqueness properties as the `ChannelId` and is handled by the controller in the same way, see Section 4.2. The hash table simply has a for every join pattern unique `JoinPatternId` as its key and the actual join pattern as its value.

Note that there is only one hash table storing the join patterns. Section 4.3 described various types of join patterns depending on the types and number of channels involved. However, all these can be grouped into a single `JoinPattern` type using Rust's `enums`

[27, §6] in a way comparable to the `Packet` implementation in Section 4.1. The separate join pattern types are then simply variants of a `JoinPattern` `enum` type. Closures [27, §13.1] associated with the join patterns are also transformed as described in Section 4.4 to unify their type signature and allow for this single storage collection.

As Section 4.2 already described, on top of these core collections, the controller also stores the latest not yet given `ChannelId`. Since `JoinPatternId`s work in the same way, it also stores the latest not yet given `JoinPatternId`. Furthermore, the controller also has a hash table using Rust's `HashMap` [8] mapping `JoinPatternId`s to the last instant they have been executed. It also has an inverted index implemented for the library to store a mapping from `ChannelId`s to the `JoinPatternId`s of the join patterns they appear in. Both of these collections are described more in Section 4.6 and justify the introduction of a `JoinPatternId` to easily retrieve specific join patterns.

With all the state described, its mutation to enable channels and join patterns is simple. The controller continuously waits for `Packet`s to be sent to it. Depending on the variant of the `Packet`, see Example 4.1, it performs different tasks:

- `Message`: store the new message from the channel in the bag of messages and attempt to execute a join pattern, see Section 4.6 for the algorithm;
- `NewChannelIdRequest`: update the latest given `ChannelId` and send it as a reply;
- `AddJoinPatternRequest`: give the join pattern a new `JoinPatternId` and store it in core hash table for them. Also update the hash table storing the last instants the join patterns have been executed as well as the inverted index;
- `ShutDownRequest`: stop listening for `Packet`s, see Section 4.7.

This makes up all of the state and state mutations of the controller. The core of it is the bag of received messages from channels and the join pattern collection itself. It is now time to describe the algorithm that chooses which join pattern is to be executed in the following section.

4.6 Join Pattern Execution

The idea behind executing join patterns is simple: if each channel involved in a particular join pattern has sent *at least one* message, the join pattern is ready to be executed. This state is henceforth referred to the join pattern being `alive`. Note that one join pattern being executed may change which join patterns are alive depending on which messages are being used by the execution.

Observe that at any given moment, there may be more than one join pattern alive. A scheme to prioritise one of the alive join patterns to be executed is required. One such scheme could be based on the order of declaration of the join patterns. One implementation of Polyphonic C#, for instance, used textual ordering of the source code [29].

For the initial algorithm devised, declare such a scheme to prioritise certain alive join patterns explicitly is not chosen. This means the scheme is largely up to implementation specifics. Algorithm 1 provides pseudocode for this naive join pattern execution.

Algorithm 1 Basic Join Pattern Execution.

```
latestChannelId := ChannelId of the last received Message
joinPatterns := collection of declared join patterns
messages := collection of previously received Messages

 1: function ExecuteJP(latestChannelId, joinPatterns, messages)
 2:     relevantJPs ← {}
 3:     aliveJPs ← {}
 4:     for each j ∈ joinPatterns do
 5:         if j includes latestChannelId then
 6:             relevantJPs ← relevantJPs ∪ {j}
 7:     for each j ∈ relevantJPs do
 8:         aliveCandidates ← {}
 9:         for each channelId ∈ j do
10:             if messages has entry for channelId then
11:                 aliveCandidates ← aliveCandidates ∪ {j}
12:             else
13:                 aliveCandidates ← {}
14:                 continue
15:         aliveJPs ← aliveJPs ∪ aliveCandidates
16:     for each j ∈ aliveJPs do
17:         requiredMsgs ← {}
18:         for each channelId ∈ j do
19:             msg ← message for channelId from messages
20:             requiredMsgs ← requiredMsgs ∪ {msg}
21:         j.execute(requiredMsgs)
22:         continue
```

Algorithm 1 takes the `ChannelId` of the latest `Message` received by the controller and determines which join patterns include the channel that has sent this `Message`, henceforth named *relevant*. The inverted index from `ChannelId`s to `JoinPatternId`s described in Section 4.5 is used for this. For every `ChannelId` it holds a linked list of the `JoinPatternId`s that include the associated channel. The Rusty Junctions Library implements a specific `InvertedIndex` collection for this using Rust's `HashMap` [8] and `LinkedList` [9] for the same reason a `Bag` collection in Section 4.5 is implemented.

After Algorithm 1 has determined the relevant join patterns, it determines which of these are alive. Following this, Algorithm 1 chooses some alive join pattern, if there is any, gathers messages of all the channels in the join pattern and passes them to the join pattern to execute its associated closure [27, §13.1].

With the help of the project supervisor, issues in the execution of join patterns were identified that are not properly addressed in Algorithm 1. These issues and how they are addressed by revising Algorithm 1 into Algorithm 2 is discussed next.

4.6.1 Issue 1: Sharing Channels

Picture a setup as in Example 4.3, in which two join patterns are declared using three channels, `A`, `B` and `C`.

```
j.when(&A).and(&B).then_do(/*...*/);
j.when(&A).and(&C).then_do(/*...*/);
```

Example 4.3: Join patterns sharing channels in Rusty Junctions, where `j` is an instance of junction.

In Example 4.3, channel `A` is used in both join patterns. Now assume both channels `B` and `C` have sent *at least one* message, but channel `A` has sent *none*. If channel `A` now sends a *single* message, both join patterns become alive but only one can be executed. For the sake of argument, assume the join pattern using channels `A` and `B` is executed. By construction, there are no more messages from channel `A` after this and so none of the join patterns is alive.

Repeating this scenario, it becomes clear that it is possible to conceive of a situation in which the join pattern using channels `A` and `B` is always the one executed. It therefore blocks the other join pattern from ever running.

A simple scheme to prevent this issue from occurring is to store, for every join pattern, the last instant at which it was executed. Algorithm 2 shows in lines 19 and 20 how this instant can be used to fully order the alive join patterns and execute the one that has been waiting for execution the longest. This prevents this issue from occurring, since it would force the join patterns in Example 4.3 to execute in an alternating fashion, given the scenario described in this section.

Section 4.5 mentioned a hash table in the controller that stores for every join pattern the last instant it has been executed. To avoid system calls to the internal clock and on the suggestion of the project supervisor, a strictly monotonically increasing message counter is introduced to the controller state. The counter is incremented at the arrival of each new message that arrives. Since a new message arriving causes the controller to check and execute a join pattern if possible, the current message counter can be used as the instant of last execution. It is then possible to totally order join patterns by comparing the message counter value of their last execution since the message counter is strictly monotonically increasing and every new message will only ever cause one join pattern to be executed.

Notice that a system running join patterns using the Rusty Junctions Library [26] may run indefinitely. To avoid integer overflow of the message counter in such a case, a `Counter` type is implemented, used for the message counter that can be incremented *indefinitely*.

The `Counter` uses an internal array representation of its value and grows the array dynamically. Due to the lack of Rust standard library support for dynamically growing integers as well as to avoid dependencies on non-official libraries which may be unstable, an implemented specific to the Rusty Junctions Library [26] is provided.

A last point to make regarding the last execution instant solution to this issue is that it also solves a similar scenario shown in Example 4.4:

```
j.when(&A).and(&B).then_do(/*...*/);
j.when(&A).then_do(/*...*/);
```

Example 4.4: Join pattern using a proper subset of the channels of another in Rusty Junctions, where `j` is an instance of junction.

Example 4.4 includes two join patterns, one of which uses a proper subset of the other. If there now was one or more messages sent only by channel `B` and a single message is sent by channel `A`, then the scenario is comparable to the one in Example 4.3. In fact, it is a specialisation of said scenario. The actual issue in both scenarios is the existence of a channel that can cause two or more join patterns to become alive *simultaneously*. Since Algorithm 2 solves this issue in Example 4.3, it does so as well for the specialisation in Example 4.4.

4.6.2 Issue 2: Repeating Channels

Imagine the following setup of Example 4.5, in which a join pattern uses the same channel multiple times:

```
j.when(&A).and(&A).then_do(/*...*/);
```

Example 4.5: Join pattern using same channel multiple times in Rusty Junctions, where `j` is an instance of junction.

There is a subtle flaw in Algorithm 1 relating to the join pattern in Example 4.5. Assume that there have not been any messages sent on channel `A`. If channel `A` now sends a *single* message, the loop in lines 9 to 14 of Algorithm 1 will falsely declare the join pattern alive, even though two messages from channel `A` are required. Lines 18 to 20 in Algorithm 1 will then attempt and fail to collect enough messages for the join pattern to be executed.

The simple solution to this issue is given in Algorithm 2. Instead of checking whether a message is available for a given channel, Algorithm 2 checks if there is at least one message available for every occurrence of the channel in the join pattern.

Algorithm 2 is the final algorithm for join pattern execution in the Rusty Junctions Library [26]. What is left to be discussed is how line 26 of Algorithm 2 actually executes the closure [27, §13.1] associated with the join pattern.

4.6.3 Cloning Closures

Section 4.3 explained how closures [27, §13.1] are stored with join patterns so that they are executed when the join pattern is ready to be executed. This begs the question which thread should be responsible for executing the closure.

Algorithm 2 Revised Join Pattern Execution.

latestChannelId := `ChannelId` of the last received `Message`
joinPatterns := collection of declared join patterns
messages := collection of previously received `Messages`
N.B.: list assumed as ordered collection *with* duplicates, [ ] represents the empty list

```
1: function ExecuteJoinPattern(latestChannelId, joinPatterns, messages)
2:     $relevantJPs \leftarrow \{\}$
3:     $aliveJPs \leftarrow \{\}$
4:     for each $j \in joinPatterns$ do
5:         if $j$ includes $latestChannelId$ then
6:             $relevantJPs \leftarrow relevantJPs \cup \{j\}$
7:     for each $j \in relevantJPs$ do
8:         $aliveCandidates \leftarrow \{\}$
9:         for each $channelId \in j$ do
10:            $numOccurs \leftarrow$ number of occurrences of $channelId$ in $j$
11:            $numMsgsgets$ number of messages for $channelId$ in $messages$
12:            if $numOccurs \leq numMsgs$ then
13:                $aliveCandidates \leftarrow aliveCandidates \cup \{j\}$
14:            else
15:                $aliveCandidates \leftarrow \{\}$
16:                continue
17:        $aliveJPs \leftarrow aliveJPs \cup aliveCandidates$
18:    if $aliveJPs \neq \emptyset$ then
19:        $sortedJPs \leftarrow$ list of $aliveJPs$ sorted by least to most recently executed
20:        $selectedJP \leftarrow$ first entry of $sortedJPs$
21:        $requiredMessages \leftarrow [\,]$
22:        for each $channelId \in selectedJP$ do
23:            $numOccurs \leftarrow$ number of occurrences of $channelId$ in $selectedJP$
24:            $msgs \leftarrow$ list of $numOccurs$ messages for $channelId$ from $messages$
25:            $requiredMessages \leftarrow requiredMessages \cup msgs$
26:        $j.execute(requiredMessages)$
```

Section 3.5 described how the controller is designed to run in its own thread. It is also the controller that stores join patterns, see Section 4.5, and runs Algorithm 2 to determine which ones are to be executed. However, since the code within the closures associated with join patterns may be essentially arbitrary and hence take an indeterminate amount of time to finish it would be unwise to execute the join pattern closure in the controller's thread. Doing so would block the controller and prevent it from, for instance, handling new messages.

Once alternative to this would be to execute a join pattern's closure in a new thread. This would require the closure to be transferred to the new thread or to be shared with it through a reference. However, Section 2.3.2 has shown that using a reference would require the `'static` lifetime [27, §10.3], i.e. the reference would need to be valid for the entire duration of the programme. Since join patterns in the Rust Junctions library [26] can be dynamically added at any point in the programme, this cannot be guaranteed.

Note that a controller also cannot simply transfer the join pattern closure to a new thread or else a join pattern would only be executable once. Therefore, A trait [27, §10.2] to allow closures [27, §13.1] to be cloned is implemented, see Section 2.3.3 for cloning. This is an existing but non-standard approach in the Rust community and allows a duplicate of the closure associated with the join pattern to be transferred into another thread and executed while still being available for the future in the controller.

4.7 Managing Resources with `ControllerHandle`

Section 3.5 describes how a controller is started by a junction in a new thread. While this behaviour is easily implemented, it is less trivial how the controller thread should be handled as a resources. That is, if and when the junction is no longer in use, how does its controller and the controller's thread get shut down.

As a solution, a `ShutDownRequest` variant of the `Packet` is implemented that is only sent by the junction when it is dropped, see Section 2.3.1 for the conditions of this. Therefore, a controller can only work as long as its associated junction still has an owner.

While the aforementioned scheme of managing the controller and its thread ensures that no resources of the system are lost, it is not necessarily the desired behaviour. It may for instance be the case that a user-defined function declares a junction and some join patterns, then only returns the channels necessary for the desired behaviour to occur, not the junction. This is henceforth referred to as using a *private* junction. Example 2.1 in Join Calculus does exactly this. The current approach of the `ShutDownRequest` when the junction is dropped would not allow for this to happen, since the junction would be dropped as soon as the function returns.

The use of a junction is still important as Section 3.4 laid out, therefore a new component for the Rusty Junctions Library [26] is implemented: the `ControllerHandle`. The `ControllerHandle` is unique and created when the thread for the controller is started. It is kept as part of the junction and has a `stop` method which has taken over the ability to send the `ShutDownRequest` from the junction. The junction is implemented in a way that user code can take ownership of it from the junction. By doing so, the junction will no longer shut down its controller upon being dropped. However, it is then up to the user code to manage this resource.

User code may choose to take ownership of the `ControllerHandle` and ignore it. In this case, the controller and its thread will still continue to run, however, there is no longer a way to terminate it other than terminating the entire programme. User code may also choose to pass the `ControllerHandle` around until the appropriate place to shut down the controller has been reached and then manually call the `stop` function of the `ControllerHandle`.

This concludes the final section of the implementation chapter. It describes the technical realisation and challenges involved in providing a functioning version of the Rusty Junctions [26] design from Chapter 3. It gives details on how message passing is used to create channels and construct join patterns and how the latter is validated using Rusts type system. In Algorithm 2 It presents the algorithm contributed by the Rusty Junctions Library [26] to manage join pattern execution and describes Rust-specific implementation details such as dynamic typing and the `ControllerHandle`. Equipped with this, the next chapter showcases examples written in the Rusty Junctions Library [26] to present the final product in use.

Chapter 5

Examples: Programming in Rusty Junctions

With the Rusty Junctions Library [26] now established through the design described in Chapter 3 and the implementation in Chapter 4, this chapter shows some examples of it in action. It starts by revisiting the storage cell example from Section 2.1 and Section 2.2 and implements it using the Rusty Junctions Library [26]. After that, it provides a solution to the Santa Claus problem posed by Trono [36] which will highlight key features of the Rusty Junctions Library [26] and demonstrate the use of multiple junctions together.

5.1 Revisiting the Storage Cell

Different implementations of the storage cell are given throughout Section 2.1 and Section 2.2. To recap, the storage cell is meant to hold a value which can be updated or accessed at any point in the programme. It starts with Example 2.1 with a Join Calculus [31] implementation and then one in Polyphonic C# [30] in Example 2.3 and the Joins Concurrency Library [35] in Example 2.4.

This section provides basis for a comparison of join patterns in the Join Caclulus [31], implementations Polyphonic C# [30] and the Joins Concurrency Library [35] and finally, the Rusty Junctions Library [26]. For this purpose, an implementation of the storage cell in the Rusty Junctions Library is given in Example 5.1:

```rust
let cell = Junction::new();           // Declare Junction.

let get = cell.recv_channel::<i32>();  // Synchronous channel.
let put = cell.send_channel::<i32>();  // Asynchronous channels.
let val = cell.send_channel::<i32>();  //

let put_val = val.clone();
cell.when(&put).and(&val).then_do(move |new, _old| {
```

```
9       put_val.send(new).unwrap();
10  });
11
12  let get_val = val.clone();
13  cell.when(&val).and_recv(&get).then_do(move |v| {
14      get_val.send(v).unwrap();
15      v
16  });
17
18  val.send(1729).unwrap();                // Initial value.
```

Example 5.1: Storage cell written using the Rusty Junctions Library [26].

Example 5.1 is functionally the same as the previous implementations of the storage cell, for instance Example 2.1. Line 1 starts by declaring a new junction to group the channels and join patterns for the storage cell. Line 3 declares a synchronous `get` channel, a `RecvChannel` as we want to receive a value through it. Lines 4 and 5 declare asynchronous `SendChannels` as we do not need to block the current thread to update the storage cell value with `put` and or to carry the state with `val`.

Lines 7 to 10 in Example 5.1 declare the join pattern used to update a value if there is one. This works in the same way as, for instance, in Example 2.1 with one minor Rust-specific change. In the closure [27, §13.1] associated with the join pattern, we need to use the `val` channel to send a message with the new value. However, since the join pattern is sent to the controller once it is completely constructed, it is transferred to another thread as is everything in the closure associated with the join pattern. Therefore, `val` would become unavailable after the join pattern declaration in lines 8 to 10 due to Rust's ownership system, see Section 2.3.1. To avoid this problem Rusty Junctions implements the `Clone` trait [6] for every channel described in Section 3.2. This means that channels can be duplicated and a duplicate can be transferred to the controller's thread with the original still available in the current thread.

Lines 12 to 16 in Example 5.1 implement the other join pattern necessary for the storage cell again similarly to, for instance, Example 2.1. The noteworthy part about lines 12 to 16 is the use of the `and_recv` method and the placement of the `RecvChannel` as the last channel in the join pattern in line 13. The closure [27, §13.1] also only takes one parameter at the end of line 13 and does return a value in line 15. All of this is a result of the implementation described in Section 4.3.

Finally, line 18 in Example 5.1 sends an initial value for the storage cell just as the Join Calculus implementation in Example 2.1 did. The takeaway from Example 5.1 is that the Rusty Junctions Library [26] is capable of implementing the same join pattern mechanism as has been shown in Example 2.1 with the Join Calculus [31] itself.

The next example is significantly larger than the storage cell one. It shows how each aspect of the Rusty Junctions Library [26] comes together to solve the Santa Claus problem [36].

5.2 Holly Jolly Christmas

The *Santa Claus problem* is an exercise in concurrency that was originally posed by Trono [36] and can be stated as follows:

> Santa Claus has 9 reindeer and 10 elves in his shop at the North Pole. Normally Santa is taking a nap, the reindeer are all on holiday and the elves are working. However, if all 9 reindeer are back from holiday, they wake Santa up and he harnesses them to deliver toys. After delivering toys together, Santa unharnesses the 9 reindeer and they go back on holiday with Santa going back to sleep.
>
> Likewise, if a group of 3 elves joins together with a common problem, it is serious enough for Santa to be woken up and have it discussed with. So if a group of 3 elves has come together, they wake up Santa and he shows them into his office where they discuss the problem. During that time, any other group of elves needs to wait. Once the problem has been discussed, Santa shows the group of elves back out of his office, they go back to work and he goes back to sleep.
>
> Lastly, since the joy of children is Santa's utmost priority he will always prioritise the 9 reindeer once they are all back from holiday over a group of 3 elves wanting to discuss a problem [36].

This problem is of particular interest because Benton has shown that a solution using join patterns in Polyphonic C# [30] is possible [29]. In this section Benton's solution [29] is used and translated into the Rusty Junctions Library [26] to verify that it is capable of solving the Santa Claus problem [36] in the same way Polyphonic C# is and demonstrate various aspects of the Rusty Junctions Library in practice. The solution to this problem presented here also solves as an example of the interaction of many junctions as atomic units of concurrency management.

As the complete and commented solution would span multiple pages it is only paraphrased here with its crucial parts highlighted. See Example B.1 in Appendix B for the complete source code. Alternatively, a complete copy of the source code can be obtained in the examples folder in the official Rusty Junctions repository [25].

Each entity in this problem, i.e. Santa Claus, each of the 9 reindeer and each of the 10 elves, will be represented by a separate thread. Following Benton's approach, Santa Claus will use rendezvous to synchronise with the 9 reindeer during harnessing *and* unharnessing as well as a group of 3 elves while showing them in *and* out of his office. Conceptually, the rendezvous using message passing and join patterns works as follows:

(1) One thread initiates the rendezvous by sending out *n* free tokens in the form of a message carrying the value *n*, where *n* is the number of entities the thread wants to rendezvous. Following that, the thread blocks until all *n* token have been consumed by *n* entities.
(2) Threads ready to rendezvous send out a message that will synchronise with the

token message, each time decreasing the number of free tokens by one. If there are no free tokens, the thread blocks until there are.

(3) Once all *n* free tokens have been consumed, i.e. *n* messages from threads ready to rendezvous have been received the thread that sent the tokens unblocks.

An implementation of this behaviour using the Rusty Junctions Library [26] is provided in Example 5.2:

```
1  pub fn rendezvous() -> (
2      ControllerHandle, BidirChannel<u32, ()>, RecvChannel<()>
3  ) {
4      let mut j = Junction::new();
5
6      let accept_n = j.bidir_channel::<u32, ()>();
7      let token = j.send_channel::<u32>();
8      let wait = j.recv_channel::<()>();
9      let all_gone = j.send_channel::<()>();
10     let entry = j.recv_channel::<()>();
11
12     let token_clone = token.clone();
13     let wait_clone = wait.clone();
14     j.when_bidir(&accept_n).then_do(move |n| {
15         token_clone.send(n).unwrap();
16         wait_clone.recv().unwrap();
17     });
18
19     j.when(&all_gone).and_recv(&wait).then_do(|_| {});
20
21     let token_clone = token.clone();
22     let all_gone_clone = all_gone.clone();
23     j.when(&token).and_recv(&entry).then_do(move |n| {
24         if n == 1 {
25             all_gone_clone.send(()).unwrap();
26         } else {
27             token_clone.send(n - 1);
28         }
29     });
30
31     let controller_handle = j.controller_handle().unwrap();
32
33     (controller_handle, accept_n, entry)
34 }
```

Example 5.2: Rendezvous implementation with *private* junction written using the Rusty Junctiosn Library [26].

Like any join pattern declaration using the Rusty Junctions Library [26], the first step is to create a new junction, which is done in line 4 of Example 5.2. As Section 3.5

described, this causes a controller to be stated in a separate thread which will handle the messages being sent by channels associated to this junction, as well as executing the join patterns declared through it.

Next, line 6 in Example 5.2 creates a new channels that is used to implement the sending of the *n* free tokens and blocking until each has been consumed. Channel `accept_n` is used to send a 32-bit unsigned integer message which will be the number of requested entities to rendezvous, *n*. However, since `accept_n` should not only send a message but also block until the *n* entities have rendezvoused, it is declared as a `BidirChannel`, a synchronous sending and receiving channel, see Section 3.2.3.

The free tokens are implemented using a separate, asynchronous state channel named `token`, defined in line 7 of Example 5.2. Similar to the `val` channel in the storage cell example, Example 5.1, this channel sends messages that carry the state of how many free tokens are available. It is meant to only send out a 32-bit unsigned integer to represent said number and not block, so it is declared as a `SendChannel`, see Section 3.2.1.

The `accept_n` and `token` channels alone, however, cannot implement the desired behaviour of creating free tokens and blocking. As Section 4.6 described, the join pattern execution happens in a separate thread and by Section 3.2.3 a `BidirChannel` like `accept_n` will only block until said execution has finished. If sending a message on the `token` channel were all that was happening as a result of a message by the `accept_n` channel, the join pattern execution would terminate as soon as the free tokens are created, return and unblock the thread sending on `accept_n`. To actually wait for the free token to be consumed, Example 5.2 introduces an auxiliary channel, `wait`, in line 8 which is defined as a `RecvChannel`. As such, it will block any thread using its `recv` function until a join pattern with the `wait` channel in it has been executed. This is used to block the thread in which the join pattern, using the `accept_n` channel, is executed.

Putting the `accept_n`, `token` and `wait` channels together, lines 14 to 17 in Example 5.2 declare a join pattern that only uses the `accept_n` channel. Once a value has been sent through this channel it is extracted and a message on the `token` channel is sent with the number provided by the message from the `accept_n` channel, see line 15. This is the creation of the free token. Immediately following that in line 16 is a call to the `recv` method of the `wait` channel, which blocks the join pattern execution and thereby also the thread that sent a message using the `accept_n` channel. This `recv` call unblocks only when all free tokens have been consumed.

To unblock a call to the `recv` method of the `wait` channel, Example 5.2, based on Benton's solution [29], introduces another channel to signal that all free tokens have been consumed, the `SendChannel` `all_gone`, created in line 9. In line 19 a join pattern synchronises a message from the `all_gone` and `wait` channel with an empty closure that returns immediately. Therefore, an execution of the join pattern in line 19 unblocks a call to the `recv` method of the `wait` channel. An execution of this join pattern is what unblocks the call in line 16, terminates the enclosing join pattern execution and thereby unblocks the thread that used the `accept_n` channel.

To allow for free tokens to be consumed, line 10 in Example 5.2 defines the `entity` channel. A message from the `entity` channel synchronises with one from the `token` channel in the join pattern declared in lines 23 to 29. The `entity` channel is defined as a `RecvChannel` so that it will block until a free token is available following the conceptual behaviour of the rendezvous. The closure associated with this join pattern takes the number of free tokens, *n*, which is carried by the message from the `token` channel. The closure then checks if *n* is equal to 1, which signals that the message from the `entity` channel is the last message needed for the rendezvous to succeed. As a result, the closures sends a message on the `all_gone` channel in line 25 and does *not* resend a message on the `token` channel thereby signalling that all free tokens are consumed and no further messages from the `entry` channel should synchronise for this rendezvous. If *n* is not 1, then the state of free tokens is updated by resending a new value on the `token` channel in line 27.

With this, the behaviour of the rendezvous is implemented. There are two necessary remarks with respect to Example 5.2 that are a result of the use of Rust and the Rusty Junctions Library [26]:

(1) Lines 12, 13, 21 and 22 clone channels so that they can be used in the join pattern declarations directly following these lines. This is necessary due to Rust's ownership system [27, §4.1] alluded to in the explanation following Example 5.1.
(2) All channels in Example 5.2 with the exception of the `token` channel use Rust's unit type `()` [2] as either their send or receive value. Where this is done, it signals that the value is irrelevant and only the ability to block a thread or signal the occurrence of an event, such as all free tokens have been consumed, is important.

To avoid having to construct the necessary join patterns for a rendezvous over and over again, Example 5.2 makes use of *private* junctions as introduced in Section 4.7. This means that in line 4 a junction is created that is later dropped with its controller, all created channels and constructed join patterns preserved. The effect is that channels can still send messages, join patterns will still be executed but no new channels or join patterns can be added.

This behaviour is realised by taking ownership of the `ControllerHandle` from the junction in line 31 of Example 5.2 as described in Section 4.7. Together with channels `accept_n` and `entity`, the `ControllerHandle` is returned in line 33 to user code calling the `rendezvous` function. It is then up to the user code to shut down the controller and its thread which are enabling the rendezvous behaviour.

It is of particular importance to realise that *every* call to the `rendezvous` function creates an entirely separate junction. The implemented behaviour is always the same, but no two rendezvous will every interfere with each other providing the local synchronisation described in the Join Calculus [31].

While the rest of the Rusty Junctions solution to Santa Claus problem [36] does follow Benton's solution [29] in terms of channels and join patterns, it uses junctions to group together dependent join patterns within the implementation. In the Rusty Junctions solution, every group, the reindeer, the elves and Santa himself gets their own junction to handle synchronisation at various points. Example 5.3 starts with the definition of the junction representing the elves with the channels and join patterns necessary for them to queue into groups of 3:

```
1  let elves = Junction::new();
2
3  let elves_waiting = elves.send_channel::<u32>();
4  let elf_queue = elves.recv_channel::<()>();
5
6  let elves_ready_clone = elves_ready.clone();
7  let elves_waiting_clone = elves_waiting.clone();
8  elves.when(&elves_waiting)
9       .and_recv(&elf_queue).then_do(move |e|
10 {
11     if e == 2 {
12         elves_ready_clone.send(()).unwrap();
13     } else {
14         elves_waiting_clone.send(e + 1).unwrap();
15     }
16 });
```

Example 5.3: junction representing the elves with join pattern declaring behaviour.

Example 5.3 starts by defining a junction for the elves in line 1. It then defines a `SendChannel` in line 3 that will carry the state of how many elves are waiting to discuss their problem with Santa. This is again reminiscent of the `val` channel in Example 5.1. Line 4 then defines the `elf_queue` channel for an elf to add itself to the waiting elves. This channel is defined as a `RecvChannel` so that it blocks the thread until its message has been consumed and the state of waiting elves has been updated.

The actual update of the state is done through a join pattern defined in lines 8 to 16 in Example 5.3. The join pattern synchronises a message from the `elf_queue` channel signalling that an elf wants to join a group of elves to talk to Santa with a message from the state channel `elves_waiting`. The state is inspected, if enough elves are present, a message on the `elves_ready` channel is sent in line 12, otherwise the state is updated and resent with the `elves_waiting` channel. Observe that the `elves_ready` channel is associated not with the elves junction but with the junction that represents Santa as given in Example 5.4:

```
1 let santa = Junction::new();
2
3 let wait_to_be_woken = santa.recv_channel::<()>();
4
5 let reindeer_ready = santa.send_channel::<()>();
6 let reindeer_not_ready = santa.send_channel::<()>();
7 let clear_reindeer_not_ready = santa.recv_channel::<()>();
8
9 let elves_ready = santa.send_channel::<()>();
```

Example 5.4: junction representing Santa Claus.

The responsibility of handling the case when a group of 3 elves is ready lies with Santa, which is why the `elves_ready` channel is created by his junction. Recall that according to Section 3.4, Santa's junction would not be able to create join patterns using the `elves_ready` channel if it was not also created by Santa's junction. On the other hand, the responsibility of recognising that 3 elves are ready in a group lies within the elves themselves, which is why that behaviour is implemented in the elves' junction in Example 5.3. When ready, the elves simply signal to Santa that they are through Santa's `elves_ready` channel. This demonstrates how junctions serve as units for coordination that interact with each other by using each other's channels. It also serves as an example of how junctions allow for a separation of concerns.

The junction representing the reindeer is analogous to the one representing the elves in Example 5.3. Its full definition as well as the join patterns associated with it can be found in Appendix B in Example B.1.

The other channels in Santa's junction in Example 5.4 follow the solution provided by Benton [29]. The `wait_to_be_woken` channel in line 3 is a blocking `RecvChannel` that is used to simulate Santa napping. A message from this channel will synchronise with either a message from the `elves_ready` or the `reindeer_ready` channel which represents Santa being woken by either the 9 reindeer or a group of 3 elves being ready. The `reindeer_ready` channel in line 5 is analogous to the `elves_ready` channel but lets the reindeer junction signal that all 9 reindeer are back from holiday.

The `reindeer_not_ready` and `clear_reindeer_not_ready` channels in lines 6 and 7 of Example 5.4, respectively, are used to prioritise reindeer over elves. The join patterns that implement the described behaviour of Santa waking for either the reindeer or a group of elves *but* prioritising the reindeer are shown in Example 5.5:

```
santa
    .when(&elves_ready)
    .and(&reindeer_not_ready)
    .and_recv(&wait_to_be_woken)
    .then_do(move |_, _| { /* Discuss with elves. */ });

santa
    .when(&reindeer_ready)
    .and_recv(&wait_to_be_woken)
    .then_do(move |_| { /* Deliver with reindeer. */ });

santa
    .when(&reindeer_not_ready)
    .and_recv(&clear_reindeer_not_ready)
    .then_do(|_| {});
```

Example 5.5: Declaration of the join patterns implementing Santa's behaviour with closure bodies omitted, where `santa` is as in Example 5.4. See Example B.1 for the full definitions.

The join pattern in lines 1 to 5 of Example 5.5 declares that when the elves are ready and the reindeer are not ready, signalled by a message of the `reindeer_not_ready` channel, and Santa is waiting to be woken, then a discussion with the elves can happen. The join pattern in lines 12 to 15 will consume a `reindeer_not_ready` message only if there is a `clear_reindeer_not_ready`, which is always sent by the reindeer's junction upon finding all 9 reindeer ready. This, therefore, ensures that in case of both reindeer and a group of elves being ready at the same time, there will not be a `reindeer_not_ready` message around and thus the join pattern in lines 1 to 5, that would cause a discussion of Santa with the elves, cannot be executed and Santa *must* deliver toys with the reindeer instead. This also demonstrates a three-channel join pattern.

On the other hand, the join pattern in lines 7 to 10 only requires that the reindeer are ready and that Santa is waiting to be woken. There is no channel analogous to the `reindeer_not_ready` for elves. This asymmetry in channels is what ensures that Santa prioritises the reindeer over the elves.

The last part of the Rusty Junctions solution to the Santa Claus problem [36] that will be highlighted here is the use of the `ControllerHandle` returned by the `rendezvous` function defined in Example 5.2. It is used to gracefully shut down the rendezvous controller and free its resources. Example 5.6 shows the creation of a rendezvous for harnessing the reindeer and the shut down request sent to the associated controller after the main loop of the programme has finished and the rendezvous is no longer being used:

```
1 let (mut ch_3, harness_accept_n, harness_entry) =
2         rendezvous();
3
4 /* Main loop: Santa naps, is woken and naps again */
5
6 ch_3.stop(); // Shut down Controller.
```

Example 5.6: Demonstration of using `ControllerHandle` returned by `rendezvous` from Example 5.2 to gracefully free resources.

The returned `ControllerHandle` and the two channels returned by the `rendezvous` function from Example 5.2 are bound to the names in line 1 and 2 of Example 5.6. The `harness_accept_n` and `harness_entry` channels are used in the main loop, symbolically represented by the comment in line 4. After the main loop finishes the `ControllerHandle ch_3` uses its method `stop` to send the `ShutDownRequest`, see Section 4.7, to the controller and shut it and its thread down gracefully.

This concludes the exploration of a Rusty Junctions solution to the Santa Claus problem, based on the Polyphonic C# solution due to Benton [29]. The remaining code either follows Benton's solution directly or repeats concepts from the Rusty Junction Library [26] that have already been explored. See Example B.1 in Appendix B for the full solution or alternatively view it in the examples folder in the official Rusty Junctions repository [25].

In the course of this chapter, some important aspects and idioms of the Rusty Junctions Library [26] were demonstrated. Both the storage cell example, Example 5.1, as well as the solution to the Santa Claus problem demonstrate the use of the asynchronous `SendChannel` to carry state in a concurrent environment. The rendezvous implementation in Example 5.2 demonstrates the use of private junctions to set up a concurrency construct and only return the channels that are necessary to interact with, like in the Join Calculus, see Example 2.1. Lines 14 to 17 of Example 5.2 also presents an idiom for function execution in a separate thread through a join pattern using only a single `BidirChannel`. Most importantly, the solution to the Santa Claus problem presented in this section exemplifies the use of multiple junctions together, having them interact by using each other's channels while handling messages and join pattern execution completely separately and locally. The next chapter deals with the evaluation of the Rusty Junctions Library [26].

Chapter 6

Evaluation: A Qualitative Review

The conceptual design of the Rusty Junctions Library [26] is outlined in Chapter 3 with the details of its implementation in Chapter 4 and practical demonstrations of its using in Chapter 5. These three chapters present the Rusty Junctions Library from its design to its actual use but do not explore the properties and limitations of the library. This chapter presents a qualitative evaluation of both.

Section 6.1 opens with an exploration of the limitations that the current version 0.1.0 of the Rusty Junctions Library [26] exhibits. Section 6.2 explores the complexity, optimality and fairness of Algorithm 2 used to execute join patterns and Section 6.3 closes the chapter by presenting existing and partially experimental features of Rust that are unused but relevant in the context of the library and its implementation.

6.1 Limitations of the Implementation

Example 5.1 in Section 5.1 demonstrates that the Rusty Junctions Library [26] is capable of reproducing a simple join pattern construction that is possible in the Join Calculus as shown in Example 2.1. The solution to the Santa Claus problem [36] given in Section 5.2 extends this by showing that the Rusty Junctions Library [26] is further capable of reproducing more complex concurrent code that has been shown to work in another implementation of join patterns, namely Polyphonic C# [30][29]. Despite this, there are some limitations to the Rusty Junctions Library [26] which this section explores.

6.1.1 Number of Channels in Join Patterns

Example 5.5 demonstrates a join pattern constructed using the Rusty Junctions Library [26] which uses three channels. For the current version 0.1.0 of the Rusty Junctions Library, this is the highest number of channels possible in any join pattern [26]. While the solution to the Santa Claus problem in Section 5.2 shows that reasonably complex concurrency constructions can still be implemented despite this limitation, it is not present in the Join Calculus [31].

However, the design of the join patters in the Rusty Junctions Library [26], as described in Section 3.3, does not dictate that at most three channels can be used in join patterns and neither does the implementation described in Section 4.3. In fact, the implementation as presented in Section 4.3 can easily be extended to allow for an arbitraty number of channels in a join pattern as Figure 4.2 subtly demonstrates. The only requirement for this is to add new types for the join patterns with more than three channels.

A more flexible implementation of join patterns that would not a priori impose any restriction on the number of channels in a join pattern might be conceivable. However, the current implementation described in Section 4.3 uses Rust's static type system and compiler to validate join patterns automatically, which is a valuable property to have to ensure correctness.

6.1.2 Channel Types in Join Patterns

On top of the limitation on the number of channels in a join pattern, Section 3.3 presents a restriction on the type channels used in a join pattern, which is by design. Following the example set by Polyphonic C# [30], the Rusty Junctions Library [26] does require a join pattern to include *at most one* `RecvChannel` or `BidirChannel`, but *not both*, which is equivalent to the restrictions on non-`async` return types in Polyphonic C#.

Examining the formal definition of the Join Calculus [31], see Definition 2.1, it is again not the case that the above restriction, which effectively translates to at most one return to a channel or function call, is present. Using parallel composition $P \mid Q$ in the process that is guarded by a join pattern, multiple $\texttt{return}\ \widetilde{E}\ \texttt{to}\ f$ processes that return values to different function calls in the join pattern are possible [31]. The Join Calculus only requires that there be at least one function call in the join pattern to allow a return to a function call [31].

It is further the case that Rust's ownership system [27, §4.1], see Section 2.3.1, does not strictly prohibit the use of multiple `RecvChannels` or `BidiChannels` in a single join pattern. One interpretation of using multiple such channels would be to send the return value of the join pattern closure [27, §13.1] to each of them. As Section 4.6.3 stated, closures associated [27, §13.1] with join patterns are *cloneable*, see Section 2.3.3 for an explanation of the `Clone` trait and its behaviour. It follows from this that any value within the closure [27, §13.1] *must* implement the `Clone` trait and hence be cloneable, which includes the return value [6]. Therefore, the return value of the closure associated with the join pattern can be cloned and sent to multiple channels in multiple threads without causing ownership violations.

By the above argument, the restriction on the type of channels used in a join pattern as prescribed by Section 3.3 can be viewed as an oversight. It causes the types that need to be implemented to support join patterns in the Rusty Junctions Library [26] to be fewer, but scenarios where returning values to multiple channels is desirable are conceivable. Therefore, future versions of the Rusty Junctions Library [26] should remove this restriction.

6.1.3 Signalling Channels

Section 5.2 and in particular Example 5.2, which defined a rendezvous construction using the Rusty Junctions Library [26], shows that it is occasionally desirable for a channel not to have a type for the sent or received message. Rather, it is sufficient for a channel to simply send or receive *a* message such that synchronisation in join patterns can occur. This will henceforth be referred to as a channel *signalling* an event. For instance, the sole purpose of the `all_gone` channel in Example 5.2 is to signal that all free tokens are gone and no more. There is no meaningful value to be sent, the sending of the message carries the meaning.

There exists a solution in the Rusty Junctions Library [26] for a channel not sending or receiving a value but only signalling an event. It is to explicitly define the type of the channel to be Rust's unit type `()`, which has only the value `()` [2]. This, however, leads to the slightly awkward and verbose syntax for signalling that can be observed in Example 6.1:

```
all_gone.send(()) // SendChannel<()>.
all_gone.send()   // Signaling channel.
```

Example 6.1: Channel using Rust's unit type `()` compared to the potential syntax for a channel that is only signalling.

It may occur as a minor detail and as Example 5.2 shows is not an actual limitation, but having channels that only signal events rather than send or receive values could increase readability of code by making it less cluttered. With the limitation on the number of channels in a join pattern and the restriction on their types discussed, the next section evaluates the properties of Algorithm 2 that is used to execute join patterns.

6.2 Join Patter Execution

Section 4.6 presents Algorithm 2 as the algorithm behind the join pattern execution in the Rusty Junctions Library [26] managed in the controller as designed in Section 3.5. Chapter 5 follows this with a demonstration of the Rusty Junctions Library [26] in use that serves as practical evidence for the correctness of the implementation.

This section describes theoretical properties of Algorithm 2 and its implementation as described in Section 4.6. Section 6.2.1 is an exploration of its *time complexity* and *optimality* and Section 6.2.2 touches on its *fairness* with respect to a definition presented by Kwiatkowska [33].

6.2.1 Time Complexity and Optimality

For the purpose of describing the *time complexity*, let n be the number of join patterns stored in the controller, i.e. that are *available*, that is executing Algorithm 2. Further, let m be the maximum number of channels used in any of these join patterns.

The first loop in Algorithm 2, lines 4 to 6, determines which join patterns include the channel that sent the message, i.e. are *relevant*, causing Algorithm 2 to be executed. This would normally be linear in n, the number of join patterns, and also depend on the complexity of checking whether a certain channel is used in a particular join pattern. However, as Section 4.5 described, a controller has an inverted index that stores for each channel, using its `ChannelId`, which join patterns it is included in, using their `JoinPatternId`s. Therefore, the time complexity of lines 4 to 6 is $\mathrm{O}(1)$, since the required information is precomputed at the time of execution of the algorithm.

The next loop in Algorithm 2 is in lines 7 to 17 and determines which of the relevant join patterns are *alive*. It may happen that every join pattern stored in the controller is relevant, so the loop staring in line 7 may be executed for every join pattern available. The inner loop from lines 9 to 16 goes through every channel in a given join pattern to determine if said join pattern is alive, so overall the loop from lines 7 to 17 is linear in both the number of join patterns and the number of channels on the join patterns, so have time complexity $\mathrm{O}(nm)$.

In line 19 of Algorithm 2, all join patterns that have been found to be alive are sorted. Since it may well be that all available join patterns are relevant *and* alive, i.e. the number of join patterns to sort in the worst case may be n, the number of available join patterns. In the actual implementation of Algorithm 2 in the Rusty Junctions Library [26], the `sort_unstable` function of the `Vec` collection provided by the Rust standard library is used [26]. According to the documentation, this sorting has a $\mathrm{O}(n \log n)$ worst-case complexity [24, `sort_unstable`].

Lastly, once a join pattern has been chosen to be executed by Algorithm 2, i.e. is *selected*, the messages necessary for the execution are collected in the loop from lines 22 to 25. As this goes through every channel of the selected join pattern, which at most has m channels, this has worst-case time complexity of $\mathrm{O}(m)$.

Collecting the results, the implementation of Algorithm 2 has a worst-case time complexity of $\mathrm{O}(1) + \mathrm{O}(nm) + \mathrm{O}(n \log n) + \mathrm{O}(m)$, which using standard results can be simplified to $\mathrm{O}(n(m + \log n))$. In practice, as Section 6.1.1 alludes to, there is a limitation of at most three channels per join pattern, which would allow the worst-case time complexity to be simplified to $\mathrm{O}(n \log n)$. In general, however, the number of available join patterns, n, and the maximum number of channels in a given join pattern, m, are independent. Given a sufficient implementation, the maximum number of channels per join pattern, m, may also be arbitrary. Therefore, without additional information on the bound of m, the worst-case time complexity of $\mathrm{O}(n(m + \log n))$ cannot further be simplified.

While the worst-case time complexity of the implementation of Algorithm 2 cannot in general be simplified beyond $\mathrm{O}(n(m + \log n))$, where n is the number of available join patterns and *m* the maximum number of channels in these join patterns, it is simple to proof that Algorithm 2 is *suboptimal*. Here, optimality is taken to be with respect to the number of computational steps to select a join pattern for execution. To show that Algorithm 2 is suboptimal, observe the join patterns in Example 6.2:

```
j.when(&A).and(&C).and(&D).then_do( /*...*/ );
j.when(&B).and(&C).and(&D).then_do( /*...*/ );
```

Example 6.2: Join patterns which are handled sub-optimally by Algorithm 2, where `j` is an instance of a junction.

Example 6.2 is due to the project supervisor and demonstrates a collection of join patterns that, under certain circumstances, only requires to check if messages on a single channel have been sent, to possibly refute that either of them are alive. For this, assume that a message is being sent through channel `C`. Observe that to be alive either join pattern in Example 6.2 requires a message sent on channel `D`, so by checking if messages from channel `D` are available and finding that there are none, neither join pattern can be alive. It therefore only requires a single check to refute that either of the join patterns are alive.

However, Algorithm 2, does not inspect the channels in the join patterns in the way described above. The loop in lines 9 to 16 may in fact check for messages on each channel in an arbitrary fashion, so potentially check for messages sent on channel `A` first. If there are none, it may still be possible that the join pattern in line 2 of Example 6.2 is alive so one more check for messages sent on channel `B` is required. If there are none, this join pattern is not alive either but it took two checks rather than one to find them both not to be alive. Hence, Algorithm 2 is suboptimal.

There is another sense in which join pattern execution in the Rusty Junctions Library [26] is suboptimal, which is in its use of threads. As Section 4.6.3 describes, each join pattern is executed in a new thread, which is not strictly necessary and may have negative implications on the scalability of applications written using the join patterns as Turon and Russo demonstrate [37].

Besides the worst-case time complexity of the implementation of Algorithm 2 and its proven suboptimality, another important aspect of it is its *fairness* with respect to a definition presented by Kwiatkowska [33], which is discussed in the next section.

6.2.2 Fairness

Kwiatkowska presents a notion of *fairness* in the following general statement:

> *No component of a system which becomes possible sufficiently often should be delayed indefinitely.* [33, p.12]

Kwiatkowska goes on to explain that a specific fairness property is obtained by explicitly defining the *system component* and what *becoming possible* and *sufficiently often* mean [33, p.12]. In terms of Algorithm 2, the system component of interest can be thought of as the join patterns. Becoming possible can be thought of as a join pattern becoming alive and assuming sufficiently often to mean *infinitely often* results in the consideration of *strong fairness* of Algorithm 2 [33, p.13].

This section provides the outline of a proof that Algorithm 2 and its implementation in the Rusty Junctions Library [26] does obey strong fairness.

Consider in the following a controller that stores an arbitrary *but finite* amount of join patterns, which is reasonable in practice. As Section 4.5 and Section 4.6.1 describe, the controller stores the value of the message counter at the last time of a join patterns execution for each join pattern. The message counter is *strictly* monotonically increasing with every new message and uses the `Counter` type, which does never suffers of integer overflow and can be incremented *indefinitely*, see Section 4.6.1.

If a join pattern, j_0, is becoming alive *infinitely often*, as prescribed by strong fairness, there exists a *smallest* message counter value, c_0, where it becomes alive *for the first time*. For now, assume that it is *not* executed at message counter value c_0. This implies that this join pattern is waiting for execution longer than at least one other join pattern, j_1, which became alive at the same message counter value c_0 and *was* executed.

If j_0 and j_1 now become alive at the same message counter value again, call this one c_1 and note that $c_1 > c_0$, and are the only two to do so, then Algorithm 2 *must* choose j_0 for execution. This is possible as j_0 becomes alive infinitely often. It follows that j_0 was delayed $c_1 - c_0 < \infty$, i.e. not indefinitely and so strong fairness holds.

If j_0 and j_1 are not the only join patterns to be alive at c_1, the same argument can be repeated for larger and larger message counter values, each time *not* choosing j_0 for execution. Since there are finitely many join patterns, eventually j_0 must have waited the longest. Call the message counter value for this c_1^* and note that $c_1^* > c_0$. Therefore Algorithm 2 *must* choose it for execution. It follows that j_0 was delayed $c_1^* - c_0 < \infty$, i.e. not indefinitely and so strong fairness holds.

However, if j_0 and j_1 *never* become alive at the same message counter value again, then there exists a message counter value, c_2 where $c_2 > c_0$, where j_0 is alive and j_1 is not and so Algorithm 2 *cannot* choose j_1 over j_0 for execution and *must* choose j_0 if no other join pattern is alive at c_2. From this it follows that j_0 was delayed $c_2 - c_0 < \infty$, i.e. not indefinitely and so strong fairness holds again.

If there were other join patterns alive at c_2, the same argument about becoming alive at the same time again or never at the same time again can be repeated. Since there are finitely many join patterns by assumption, there must exist a finite message counter value, c_n where $c_n > c_0$, at which j_0 has waited the longest for execution out of all the alive join patterns and so Algorithm 2 *must* choose j_0 for execution. It again follows that j_0 was delayed $c_n - c_0 < \infty$, i.e. not indefinitely and so strong fairness holds.

Finally, if j_0 were executed at c_0, a finite message counter value, then it was trivially not delayed indefinitely. Additionally, if j_0 were the only join pattern stored in the controller, then j_0 has trivially waited the longest for execution out of all the alive join patterns and so Algorithm 2 *must* choose j_0 for execution *every time* it becomes alive, which implies it is never delayed. Thus, strong fairness holds in both cases.

It is worth pointing out that the development of the `Counter` type, which never suffers from integer overflow and can represent an infinite amount of integer values, was motivated by exactly this argument. If the message counter were implemented with a type that could only represent a finite amount of values there would exist the possibility that the difference between the message counter value of j_0 becoming alive and being executed for the first time is larger than the largest value that can be represented by said type.

The above argument should be able to convince the reader to consider Algorithm 2 and its implementation to exhibit the strong fairness property as described by Kwiatkowska [33]. The next section details features that exist in versions of the Rust programming language but remain unused in the current version 0.1.0 in the Rusty Junctions Library [26].

6.3 Unused Features

As Chapter 4 stated at the beginning, the current version 0.1.0 of the Rusty Junctions Library [26] is implemented using version 1.35.0 of the Rust programming language, which was the most recent version of the language at the start of the project but new versions with new features [3] have been published during the development of the Rusty Junctions Library [26]. This section reflects on the features that would have an impact on the API exposed by the Rusty Junctions Library and justifies why they are not used.

6.3.1 Futures and `async-await`

Rust's version 1.39.0 brought the addition of stable `async` and `await` syntax for suspended computation using futures [3]. Currently, as described in Section 3.2, the `RecvChannel` and `BidirChannel` are designed to block the thread once their `recv` or `send_recv` functions are called. With futures it would be possible to delay the point that the thread is blocked beyond the call of `recv` or `send_recv` as Example 6.3 demonstrates:

```
j.when(&val).and_recv(&get).then_do(async move || { /*...*/});

let v_future = get.recv(); // Does not block!

/* More code... */

let v = v_future.await;    // Now blocks and waits for result.
```

Example 6.3: Join pattern construction and receivng value using `async-await` syntax, where `j` is a junction and `val` and `get` are as in Example 5.1.

Example 6.3 assumes the same channel definitions as the storage cell implementation in Example 5.1 to allow for a syntax comparison.

The join pattern declared in line 1 of Example 6.3 uses the `async` keyword for the associated closure which marks it as returning a future. Following that, the call to the `recv` method of the `get` channel in line 3 returns a future and does *not* block. It is therefore possible to request a result from a join pattern using the `get` channel without blocking the thread immediately. The thread can go on to execute further code and at a convenient time using the `.await` syntax in line 7 on the future retrieved in line 3 finally blocking the thread to wait for an actual value. Note that Example 6.3 is only to be taken as a design proposal and not as a prototype for a working implementation.

The `async-await` syntax offering user code the ability to not immediately block the current thread could add more freedom to the Rusty Junctions Library [26] from a user's perspective. However, as this feature was released as stable only during the development of the Rusty Junctions Library, it appeared safer to stick to Rust features that were more proven.

There is also a question of how frequently user code would choose to postpone blocking a thread to later await a value. Since there is no extensive code base written using join patterns with the Rusty Junction Library [26], there is only a small amount of experience as to how frequently such a feature would be used in practice. Nevertheless, future versions of the library could consider integration of the `async-await` syntax with join patterns and channels.

6.3.2 The Function Call Operator

Section 6.1.3 already discusses one way in which the Rusty Junctions API may be more verbose than required with Example 6.1. Another way in which this may be the case, arises when comparing the Rusty Junctions API with Polyphonic C# [30]. The latter uses function calls to the functions declared in a chord to enable said chord to be fired [30]. The Rusty Junctions Library [26] on the other hand uses generic methods such as, for instance, `send` on `SendChannel`s to achieve a similar behaviour.

Rust actually offers function traits [27, §10.2] that would allow a channel to implement a function call operator so that sending a message through a channel can look like a genuine function call. The function traits are:

- `Fn`, for function calls that do not mutate state [13];
- `FnMut`, for function calls that may mutate state [14];
- `FnOnce`, for instance, in which multiple function calls may not be possible [15].

Example 6.4 presents a comparison of the current API for sending messages on channels in the Rusty Junctions Library [26] with how sending messages on channels could look like using the function call operator:

```
put.send(1729);       // Current API.
let v = get.recv(); //

put(1729);            // With function call operator.
```

```
let v = get();        //
```

Example 6.4: Comparison of Rusty Junctions API with and without implemented function call operator for channels, where `put` and `get` are as in Example 5.1.

Example 6.4 shows that using the function call operator with channels instead of methods such as `send` can reduce the amount of code needed to be written and increase readability. It would also be more in line with the Join Calculus itself, as the Join Calculus uses functions calls to enable join patterns as well [31]. Unfortunately, this feature of Rust is still experimental in all function traits at the time of writing [13][14][15]. It was therefore disregarded for version 0.1.0 of the Rusty Junctions Library [26] so that the library itself does not force experimental features into user code.

This concludes the qualitative evaluation of the Rusty Junctions Library. As section Section 6.1 discusses, there are some obvious limitations in the current implementation that are not based on the theoretical foundations in the Join Calculus, such as for instance the limitation on the types of channels in a join pattern. Section 6.2.1 on the other hand shows that the join pattern execution algorithm while being suboptimal may still exhibit interesting characteristics relating to fairness. Finally, this section demonstrates that there are Rust features that are not used for the current version of the Rusty Junctions Library [26] but are of interest for future development. The next chapter presents some closing remarks on this report and the project as a whole and outlines potential future work.

Chapter 7

Conclusions and Future Work

The Rusty Junctions Library [26] implements a novel concurrency construct named a *junction*, which uses message passing and join patterns based on the Join Calculus [31] to allow for thread-safe and declarative concurrent programming. Its design is heavily inspired by the Joins Concurrency Library, a previous implementation of join patterns in C# [35].

The contribution of junctions with an array of channels for various message passing tasks enriches the Rust programming language [1] with a new paradigm for concurrent programming. Seemingly complex synchronisation tasks such as the Santa Claus problem [36] are solvable using the Rusty Junctions Library by declaring the required behaviour. Using the Rusty Junctions Library no synchronisation has to be done manually which avoids potentially difficult to diagnose problems.

Join patterns in the Rusty Junctions Library [26] are designed around Rust's rich, static type system that offers memory safety through concepts such as ownership [34]. Rust's type system has undoubtedly had a great impact on the structure of the Rusty Junctions Library by requiring restrictions such as requiring cloneable values in join patterns bodies. However, Rust's type system is also leveraged by a series of implemented types that ensure the correctness of join patterns constructions at compile time as described in Section 4.3.

A large part of the contribution of the Rusty Junctions Library [26] is the *controller* construct that holds internal state for junctions and handles the scheduling of join pattern execution. The algorithm used for the scheduling, Algorithm 2, is a novel contribution specifically developed for the Rusty Junctions Library. While it is proven suboptimal in terms of steps to arrive at a scheduling decision, it exhibits fairness properties, for which a proof is outlined in Section 6.2.2.

This report demonstrates that the current version of the Rusty Junctions Library, version 0.1.0 [26], is a viable proof-of-concept for a join pattern implementation in the Rust programming language. This is done through examples as presented in Chapter 5. These show that despite the limitations that exist for this version of the library, explored in Section 6.1, a non-trivial synchronisation task such as the Santa Claus problem can be solved in the same way as a previous implementation of join patterns, Polyphonic

C# [30], managed to do [29].

The future work for the Rusty Junctions Library [26] entails both practical experimentation as well as improvements of the issues raised in Chapter 6. Turon and Russo have shown join patterns as implemented in the Joins Concurrency Library [35] to be a scalable solution to concurrency problems [37]. A similar approach investigation the throughput of systems, written using the Rusty Junctions Library [26], is to be undertaken, which is likely to raise issues that could lead to improvements of the design and implementation in order to make join patterns a viable tool in Rust's concurrency landscape.

Section 6.3 highlights that the Rust programming language is being actively developed which has led to new features that could benefit the API exposed by the Rusty Junctions Library [26]. As Section 6.2.1 has proven, there is also work to be done to optimise the join pattern execution, which is likely to benefit systems wanting to rely on junctions and join patterns as their concurrency programming model, making them a feasible approach in practice.

Bibliography

[1] Rust Language Website. https://www.rust-lang.org/. Accessed: 2020-03-05.

[2] Rust Primitive Type unit Documentation. https://doc.rust-lang.org/std/primitive.unit.html. Accessed: 2020-03-12.

[3] Rust Releases. https://github.com/rust-lang/rust/blob/master/RELEASES.md. Accessed: 2020-03-20.

[4] Rust std::any::Any Documentation. https://doc.rust-lang.org/std/any/trait.Any.html. Accessed: 2020-03-08.

[5] Rust std::boxed::Box Documentation. https://doc.rust-lang.org/std/boxed/struct.Box.html. Accessed: 2020-03-14.

[6] Rust std::clone::Clone Documentation. https://doc.rust-lang.org/std/clone/trait.Clone.html. Accessed: 2020-03-08.

[7] Rust std::collections Documentation. https://doc.rust-lang.org/std/collections/index.html. Accessed: 2020-03-14.

[8] Rust std::collections::HashMap Documentation. https://doc.rust-lang.org/std/collections/struct.HashMap.html. Accessed: 2020-03-14.

[9] Rust std::collections::LinkedList Documentation. https://doc.rust-lang.org/std/collections/struct.LinkedList.html. Accessed: 2020-03-14.

[10] Rust std::collections::VecDeque Documentation. https://doc.rust-lang.org/std/collections/struct.VecDeque.html. Accessed: 2020-03-14.

[11] Rust std::marker::Send Documentation. https://doc.rust-lang.org/std/marker/trait.Send.html. Accessed: 2020-03-08.

[12] Rust std::marker::Sync Documentation. https://doc.rust-lang.org/std/marker/trait.Sync.html. Accessed: 2020-03-08.

[13] Rust std::ops::Fn Documentation. https://doc.rust-lang.org/std/ops/trait.Fn.html. Accessed: 2020-03-20.

[14] Rust `std::ops::FnMut` Documentation. `https://doc.rust-lang.org/std/ops/trait.FnMut.html`. Accessed: 2020-03-20.

[15] Rust `std::ops::FnOnce` Documentation. `https://doc.rust-lang.org/std/ops/trait.FnOnce.html`. Accessed: 2020-03-20.

[16] Rust `std::panic` Documentation. `https://doc.rust-lang.org/std/macro.panic.html`. Accessed: 2020-03-14.

[17] Rust `std::println` Documentation. `https://doc.rust-lang.org/std/macro.println.html`. Accessed: 2020-03-05.

[18] Rust `std::string::String` Documentation. `https://doc.rust-lang.org/std/string/struct.String.html`. Accessed: 2020-03-05.

[19] Rust `std::sync::atomic::AtomicUsize` Documentation. `https://doc.rust-lang.org/std/sync/atomic/struct.AtomicUsize.html`. Accessed: 2020-03-13.

[20] Rust `std::sync::mpsc` Documentation. `https://doc.rust-lang.org/std/sync/mpsc/index.html`. Accessed: 2020-03-13.

[21] Rust `std::sync::mpsc::Receiver` Documentation. `https://doc.rust-lang.org/std/sync/mpsc/struct.Receiver.html`. Accessed: 2020-03-13.

[22] Rust `std::sync::mpsc::Sender` Documentation. `https://doc.rust-lang.org/std/sync/mpsc/struct.Sender.html`. Accessed: 2020-03-13.

[23] Rust `std::thread::spawn` Documentation. `https://doc.rust-lang.org/std/thread/fn.spawn.html`. Accessed: 2020-03-08.

[24] Rust `std::vec::Vec` Documentation. `https://doc.rust-lang.org/std/vec/struct.Vec.html`. Accessed: 2020-03-09.

[25] Rusty Junctions Library - Examples. `https://github.com/smueksch/rusty_junctions/tree/master/examples`. Accessed: 2020-03-16.

[26] Rusty Junctions Library Crate. `https://crates.io/crates/rusty_junctions/`. Accessed: 2020-03-08.

[27] The Rust Programming Language. `https://doc.rust-lang.org/book/title-page.html`. Accessed: 2020-03-05.

[28] Abhiram Balasubramanian, Marek S. Baranowski, Anton Burtsev, Aurojit Panda, Zvonimir Rakamarić, and Leonid Ryzhyk. System Programming in Rust: Beyond Safety. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, HotOS '17, page 156–161, New York, NY, USA, 2017. Association for Computing Machinery.

[29] Nick Benton. Jingle bells: Solving the santa claus problem in Polyphonic C#. *Unpublished manuscript, Mar*, 2003.

[30] Nick Benton, Luca Cardelli, and Cédric Fournet. Modern Concurrency Abstractions for C#. In *European Conference on Object-Oriented Programming*, pages 415–440. Springer, 2002.

[31] Cédric Fournet and Georges Gonthier. The join calculus: A language for distributed mobile programming. In *Applied Semantics, International Summer School, APPSEM 2000, Caminha, Portugal, September 9-15, 2000, Advanced Lectures*, page 268–332, Berlin, Heidelberg, 2000. Springer-Verlag.

[32] Cédric Fournet, Fabrice Le Fessant, Luc Maranget, and Alan Schmitt. Jocaml: A language for concurrent distributed and mobile programming. In *International School on Advanced Functional Programming*, pages 129–158. Springer, 2002.

[33] Marta Zofia Kwiatkowska. *Fairness for non-interleaving concurrency*. PhD thesis, University of Leicester Phd thesis, 1989.

[34] Nicholas D. Matsakis and Felix S. Klock. The rust language. In *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*, HILT '14, page 103–104, New York, NY, USA, 2014. Association for Computing Machinery.

[35] Claudio Russo. The Joins Concurrency Library. In *International Symposium on Practical Aspects of Declarative Languages*, pages 260–274. Springer, 2007.

[36] John A Trono. A new exercise in concurrency. *ACM SIGCSE Bulletin*, 26(3):8–10, 1994.

[37] Aaron J. Turon and Claudio V. Russo. Scalable join patterns. *SIGPLAN Not.*, 46(10):575–594, October 2011.

Appendices

Appendix A

Complete Join Calculus Definition

Definition A.1 provides a complete definition of the Join Calculus as described by Fournet and Gonthier [31]:

Definition A.1 Complete formal Join Calculus definition as given by Fournet and Gonthier [31].

P,Q,R	$::=$	processes
	$\texttt{let}\,x\ =\ E\ \texttt{in}\,P$	compute expression
	$\|\ \texttt{let}\,f(\widetilde{x})\ =\ E\ \texttt{in}\,P$	local recursive function definition
	$\|\ P \mid Q$	parallel composition
	$\|\ \mathbf{0}$	inert process
	$\|\ p\langle\widetilde{E}\rangle$	execute abstract process
	$\|\ \texttt{def}\ p\langle\widetilde{x}\rangle\ \triangleright\ P\ \texttt{in}\ Q$	process abstraction
	$\|\ \texttt{return}\ \widetilde{E}\ \texttt{to}\ f$	return value(s) to function call
	$\|\ \texttt{let}\,\widetilde{x}\ =\ E\ \texttt{in}\,P$	local recursive function definition
	$\|\ \texttt{def}\ f(\widetilde{x})\ \triangleright\ P\ \texttt{in}\ Q$	recursive function definition
	$\|\ \texttt{def}\ D\ \texttt{in}\ P$	process/function definition
E,F	$::=$	expressions
	x,y,f	variable
	$\|\ f(\widetilde{E})$	function call
	$\|\ \texttt{let}\,x\ =\ E\ \texttt{in}\,F$	local value definition
	$\|\ \texttt{let}\,f(\widetilde{x})\ =\ E\ \texttt{in}\,F$	local recursive function definition
	$\|\ \texttt{run}\ P$	spawn process
	$\|\ \texttt{let}\,\widetilde{x}\ =\ E\ \texttt{in}\,F$	local definition(s)
	$\|\ \texttt{def}\ f(\widetilde{x})\ \triangleright\ P\ \texttt{in}\ E$	recursive function definition
	$\|\ \texttt{def}\ D\ \texttt{in}\ E$	process/function definition
D	$::=$	definitions
	$J\ \triangleright\ P$	execution rule
	$\|\ D \wedge D'$	alternative definitions
	$\|\ \top$	empty definition
J	$::=$	join patterns
	$x\langle\widetilde{y}\rangle$	message send pattern
	$\|\ x(\widetilde{y})$	function call pattern
	$\|\ J \mid J'$	synchronisation

Appendix B

Full Santa Claus Solution

Example B.1 below provides the full source code of the solution to the Santa Claus problem [36] written using the Rusty Junctions Library [26]. A copy of the source code can also be optained in the examples folder of the official Rusty Junctions Library repository [25].

```rust
use rand::Rng;

use std::{thread, time::Duration};

use rusty_junctions::channels::{BidirChannel, RecvChannel};
use rusty_junctions::types::ControllerHandle;
use rusty_junctions::Junction;

fn main() {
    /*****************************
     * Elves Junction & Channels *
     *****************************/

    // Elf Junction.
    let elves = Junction::new();

    // Synchronous channel to signal that elf wants to queue up.
    let elf_queue = elves.recv_channel::<()>();

    // Asynchronous channel to carry the number of elves that
    // are queued up.
    let elves_waiting = elves.send_channel::<u32>();

    /********************************
     * Reindeer Junction & Channels *
     ********************************/

    // Reindeer Junction.
    let reindeer = Junction::new();

    // Synchronous channel to signal that reindeer is back from
    // holiday.
    let reindeer_back = reindeer.recv_channel::<()>();

    // Asynchronous channel to carry the number of reindeer
    // waiting in the stable.
    let reindeer_waiting = reindeer.send_channel::<u32>();

    /*****************************
     * Santa Junction & Channels *
     *****************************/

    // Santa's Junction.
    let santa = Junction::new();

    // Synchronous channel to wait to be woken by either
    // reindeer or elves.
    let wait_to_be_woken = santa.recv_channel::<()>();

    // Asynchronous channel to signal that enough reindeer
    // are ready.
    let reindeer_ready = santa.send_channel::<()>();

    // Asynchronous channel to signal that not enough reindeer
    // are ready.
    // Used for prioritisation.
    let reindeer_not_ready = santa.send_channel::<()>();

    // Synchronous channel to match and consume a
    // reindeer_not_ready message.
    // Used for prioritisation.
    let clear_reindeer_not_ready = santa.recv_channel::<()>();

    // Asynchronous channel to signal that enough elves are
    // ready.
    let elves_ready = santa.send_channel::<()>();

    // Rendezvous channels to let elves into room.
    let (mut ch_1, room_in_accept_n, room_in_entry) =
        rendezvous();

    // Rendezvous channels to let elves out of room.
    let (mut ch_2, room_out_accept_n, room_out_entry) =
        rendezvous();

    // Rendezvous channels to harness the reindeer.
    let (mut ch_3, harness_accept_n, harness_entry) =
        rendezvous();

    // Rendezvous channels to unharness the reindeer.
    let (mut ch_4, unharness_accept_n, unharness_entry) =
        rendezvous();

    /**********************
     * Elves Join Patterns *
     **********************/

    // Count up how many elves are waiting and possibly send
    // ready message.
    let elves_ready_clone = elves_ready.clone();
    let elves_waiting_clone = elves_waiting.clone();
    elves
        .when(&elves_waiting)
        .and_recv(&elf_queue)
        .then_do(move |e| {
            if e == 2 {
                // Last elf just queued.
                elves_ready_clone.send(()).unwrap();
                println!("<Elves>␣Group␣of␣3␣ready!");
            } else {
                elves_waiting_clone.send(e + 1).unwrap();
                println!("<Elves>␣{}␣waiting!", e + 1);
            }
        });

    /*************************
     * Reindeer Join Patterns *
     *************************/

    // Count up how many reindeer are waiting and possibly send
    // ready message.
    let reindeer_ready_clone = reindeer_ready.clone();
    let reindeer_waiting_clone = reindeer_waiting.clone();
    let clear_reindeer_not_ready_clone =
        clear_reindeer_not_ready.clone();
    reindeer
        .when(&reindeer_waiting)
        .and_recv(&reindeer_back)
        .then_do(move |r| {
            if r == 8 {
                // Last reindeer just came back.
                clear_reindeer_not_ready_clone.recv().unwrap();
                reindeer_ready_clone.send(()).unwrap();
                println!("<Reindeer>␣All␣9␣assembled!");
            } else {
                reindeer_waiting_clone.send(r + 1).unwrap();
                println!("<Reindeer>␣{}␣waiting!", r + 1);
            }
        });

    /***********************
     * Santa Join Patterns *
     ***********************/

    // Enough elves are ready so let's consult with them.
    let reindeer_not_ready_clone = reindeer_not_ready.clone();
    let elves_waiting_clone = elves_waiting.clone();
    santa
        .when(&elves_ready)
        .and(&reindeer_not_ready)
        .and_recv(&wait_to_be_woken)
        .then_do(move |_, _| {
            let mut rng = rand::thread_rng();

            // Reindeer will still not be ready so resend just
            // consumed message.
            reindeer_not_ready_clone.send(()).unwrap();

            // Show 3 elves into the office once all are ready.
            println!(
                "<Santa>␣Woken␣by␣elves,␣now␣showing␣them␣in!"
            );
            room_in_accept_n.send_recv(3).unwrap();
            println!("<Santa>␣Elf␣group␣shown␣in!");

            // Reset how many elves are waiting to allow others
            // to form a group.
            elves_waiting_clone.send(0).unwrap();

            // Consult with elves for 0 to 10 seconds, i.e.
            // pause thread.
            println!("<Santa>␣Now␣consulting␣with␣elves!");
            thread::sleep(
                Duration::from_secs(rng.gen_range(0, 10))
            );
            println!("<Santa>␣Consulted␣with␣elves!");

            // Done consulting with elves so show all 3 out once
            // all are ready.
            println!("<Santa>␣Now␣showing␣out␣elves!");
            room_out_accept_n.send_recv(3).unwrap();
            println!("<Santa>␣Elf␣group␣shown␣out!");
        });

    // Enough reindeer are ready so let’s deliver some presents.
    let reindeer_not_ready_clone = reindeer_not_ready.clone();
    let reindeer_waiting_clone = reindeer_waiting.clone();
    santa
        .when(&reindeer_ready)
        .and_recv(&wait_to_be_woken)
        .then_do(move |_| {
            let mut rng = rand::thread_rng();

            // Harness all 9 reindeer once they are all ready.
            println!(
                "<Santa>␣Woken␣by␣reindeer,␣now␣harnessing!"
            );
            harness_accept_n.send_recv(9).unwrap();
            println!("<Santa>␣Reindeer␣harnessed!");

            // Reindeer are harnessed, so they are no longer
            // ready.
            // Used for prioritisation.
            reindeer_not_ready_clone.send(()).unwrap();

            // Reset how many reindeer are waiting.
            reindeer_waiting_clone.send(0).unwrap();

            // Deliver toys with reindeer for 0 to 10 seconds,
            // i.e. pause thread.
            println!("<Santa>␣Now␣delivering␣toys!");
            thread::sleep(
                Duration::from_secs(rng.gen_range(0, 10))
            );
            println!("<Santa>␣Toys␣delivered!");

            // Done delivering toys, unharness all 9 reindeer
            // once all are ready.
            println!("<Santa>␣Now␣unharnessing␣reindeer!");
            unharness_accept_n.send_recv(9).unwrap();
            println!("<Santa>␣Reindeer␣unharnessed!");
        });

    // Clear the reindeer_not_ready message. Used for
    // prioritisation.
    santa
        .when(&reindeer_not_ready)
        .and_recv(&clear_reindeer_not_ready)
        .then_do(|_| {});

    /*******************************
     * Start North Pole Operations *
     *******************************/

    // Spawn in the 10 elves and send the initial number of
    // waiting ones.
    for i in 0..10 {
        new_elf(
            elf_queue.clone(),
            room_in_entry.clone(),
            room_out_entry.clone(),
        );
    }
    elves_waiting.send(0).unwrap();

    // Spawn in the 9 reindeer, send the initial number of
    // waiting ones and
    // send that they are not ready yet.
    for i in 0..9 {
        new_reindeer(
            reindeer_back.clone(),
            harness_entry.clone(),
            unharness_entry.clone(),
        );
    }
    reindeer_waiting.send(0).unwrap();
    reindeer_not_ready.send(()).unwrap();

    // Santa keeps napping until something comes up.
    println!("<North␣Pole>␣Starting␣operations!");
    while true {
        println!(
            "<Santa>␣Starting␣a␣nap,␣waiting␣to␣be␣woken..."
        );
        wait_to_be_woken.recv().unwrap();
        println!("<Santa>␣Woken␣from␣nap!");
    }

    // Clean up the controller resources in the background
    // manually at the end.
    ch_1.stop();
    ch_2.stop();
    ch_3.stop();
    ch_4.stop();
}

// Create a new elf in a new thread.
fn new_elf(
    queue: RecvChannel<()>,
    room_in_entry: RecvChannel<()>,
    room_out_entry: RecvChannel<()>,
) {
    println!("<North␣Pole>␣New␣elf␣hired!");
    thread::spawn(move || {
        // Random number generator for working and consulting
        // times.
        let mut rng = rand::thread_rng();

        while true {
            // Work for 0 to 10 seconds, i.e. pause thread.
            println!("<Elf>␣Going␣to␣work␣now!");
            thread::sleep(
                Duration::from_secs(rng.gen_range(0, 10))
            );
            println!("<Elf>␣Done␣working!");

            // Done working, so wait and try to join a group of
            // 3 elves.
            println!(
                "<Elf>␣Queuing␣now,␣waiting␣for␣more␣elves..."
            );
            queue.recv().unwrap();
            println!("<Elf>␣Found␣a␣group␣of␣elves!");

            // Found group of elves so sait for Santa to show
            // me in.
            println!(
                "<Elf>␣Waiting␣for␣Santa␣to␣show␣me␣in..."
            );
            room_in_entry.recv().unwrap();
            println!("<Elf>␣Santa␣is␣showing␣me␣in␣now!");

            // Consult with Santa for 0 to 10 seconds, i.e.
            // pause thread.
```

```
                println!("<Elf>␣Consulting␣with␣Santa␣now!");
                thread::sleep(
                    Duration::from_secs(rng.gen_range(0, 10))
                );
                println!("<Elf>␣Done␣consulting␣with␣Santa!");

                // Wait for Santa to show me out.
                println!(
                    "<Elf>␣Waiting␣for␣Santa␣to␣show␣me␣out..."
                );
                room_out_entry.recv().unwrap();
                println!("<Elf>␣Santa␣is␣showing␣me␣out␣now!");
            }
        });
}

// Create a new reindeer in a new thread.
fn new_reindeer(
    reindeer_back: RecvChannel<()>,
    harness_entry: RecvChannel<()>,
    unharness_entry: RecvChannel<()>,
) {
    println!("<North␣Pole>␣New␣reindeer␣hired!");
    thread::spawn(move || {
        // Random number generator for holiday and delivery
        // times.
        let mut rng = rand::thread_rng();

        while true {
            // Go on holiday for 0 to 10 seconds, i.e. pause
            // thread.
            println!("<Reindeer>␣Going␣on␣holiday␣now!");
            thread::sleep(
                Duration::from_secs(rng.gen_range(0, 10))
            );
            println!("<Reindeer>␣Back␣from␣holiday!");

            // Done with holiday so join a group of reindeer in
            // the stable.
            println!(
                "<Reindeer>␣Waiting␣now␣for␣enough␣reindeer..."
            );
            reindeer_back.recv().unwrap();
            println!(
                "<Reindeer>␣Reindeer␣now␣assembled␣in␣stable!"
            );

            // Enough reindeers in the stable so wait for Santa
            // to harness me.
            println!(
                "<Reindeer>␣Waiting␣for␣Santa␣to␣harness␣me..."
            );
            harness_entry.recv().unwrap();
            println!("<Reindeer>␣Santa␣is␣harnessing␣me␣now!");

            // Deliver toys with Santa for 0 to 10 seconds, i.e.
            // pause thread.
            println!(
                "<Reindeer>␣Delivering␣toys␣with␣Santa␣now!"
            );
            thread::sleep(
                Duration::from_secs(rng.gen_range(0, 10))
            );
            println!(
                "<Reindeer>␣Done␣delivering␣toys␣with␣Santa!"
            );

            // Done delivering toys with Santa so sait for him
            // to unharness me.
            println!(
                "<Reindeer>␣Waiting␣to␣be␣unharnessed..."
            );
            unharness_entry.recv().unwrap();
            println!(
                "<Reindeer>␣Santa␣is␣unharnessing␣me␣now!"
            );
        }
    });
}

// Set up a private Junction for a rendezvous and return the
// public channels.
pub fn rendezvous() -> (
    ControllerHandle, BidirChannel<u32, ()>, RecvChannel<()>
) {
    let mut j = Junction::new();

    // Asynchronous token channel to carry the state.
    let token = j.send_channel::<u32>();

    // Synchronous entry channel.
    let entry = j.recv_channel::<()>();
```

```rust

    // Synchronous channel to set up number of available tokens.
    let accept_n = j.bidir_channel::<u32, ()>();

    // Synchronous wait channel.
    let wait = j.recv_channel::<()>();

    // Asynchronous all_gone channel.
    let all_gone = j.send_channel::<()>();

    // Count down the arrivals.
    let token_clone = token.clone();
    let all_gone_clone = all_gone.clone();
    j.when(&token).and_recv(&entry).then_do(move |n| {
        if n == 1 {
            all_gone_clone.send(()).unwrap();
        } else {
            token_clone.send(n - 1);
        }
    });

    // Spawn n new token and wait for all entries to rendezvous.
    let token_clone = token.clone();
    let wait_clone = wait.clone();
    j.when_bidir(&accept_n).then_do(move |n| {
        token_clone.send(n).unwrap();
        wait_clone.recv().unwrap();
    });

    // Stop waiting once all tokens are gone.
    j.when(&all_gone).and_recv(&wait).then_do(|_| {});

    // Prevent Junction from stopping control thread after
    // return.
    let controller_handle = j.controller_handle().unwrap();

    // Return the necessary channels.
    (controller_handle, accept_n, entry)
}
```

Example B.1: Complete and commented implementation of a solution to the Santa Calus problem [36] written using the Rusty Junctions Library [26].