UNIVERSITY OF COPENHAGEN

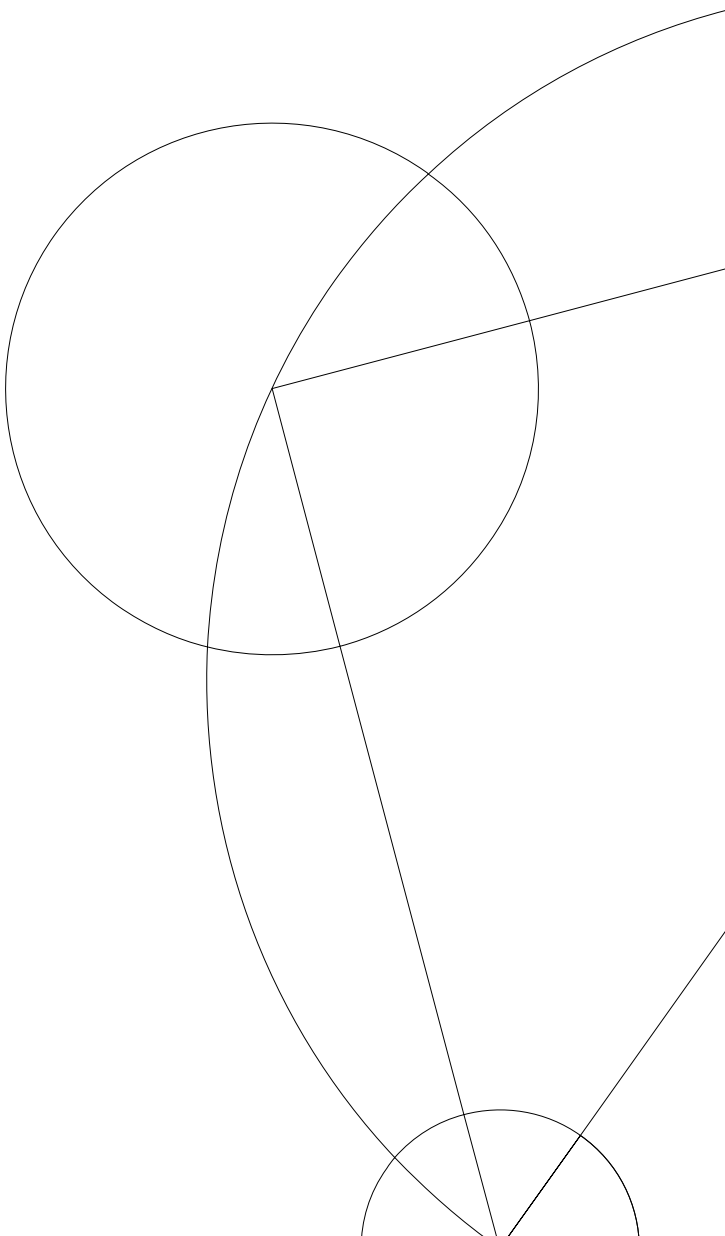
Practical Session Types in Rust

Developing safe communication protocols for real-world applications

Philip Munksgaard
pmunksgaard@gmail.com

Thomas Bracht Laumann Jespersen
ntl316@alumni.ku.dk

June 2015



Supervisor: Ken Friis Larsen

Abstract

We provide an implementation of session types in Rust and demonstrate how to incorporate session-typed channels in the Servo browser engine to guarantee protocol safety among concurrent, communicating threads. Session types provide a protocol abstraction, expanding on traditional typed communication channels, to ensure communication takes place according to a specified protocol.

Existing browser engines are over a decade old and unfit for modern computer architectures, where parallelism and energy efficiency are prioritized over raw power. To take advantage of modern computing platforms Mozilla Research is developing Servo, a next-generation rendering engine. Servo is implemented in Rust, a new systems programming language that provides memory-safety and simple concurrency mechanisms, while retaining the speed of C or C++, all of which is essential to Servo.

We present and discuss the overall design of Servo and its internal communication patterns and demonstrate how the design can lead to discrepancies in communication expectations and potentially result in errors. The communication patterns employed in Servo are defined in an ad-hoc manner without any imposed overall structure. The ad-hoc communication patterns make it difficult to reason about communication flow and it is therefore hard to make assertions concerning safety and deadlock-freedom.

To address these issues, we design and implement a session type library in Rust called `session-types`, and discuss practical use cases through examples and demonstrate how they may be used in a large-scale application. Specifically, we replace parts of Servo's internal communication infrastructure with session-typed channels, and demonstrate that the use of session types allows us to provide compile-time guarantees, without incurring any significant run-time penalties. We also discuss challenges and drawbacks we discovered when using session-typed communication in a large real-world application. Our work is publicly available as a fork of the Servo project and we are currently seeking to land these changes in Servo.

While developing `session-types`, we have also developed several examples and examined different use cases for `session-types`. Among these are a solution to the Santa Claus problem first described in [1], an implementation of the recursive arithmetic server described in [2], as well as many others. Along with our continual examination of the communication patterns in Servo, these examples and case studies have guided us towards a practical and usable library design that enforces a strict communication discipline, while still allowing many useful communication patterns.

We draw inspiration from implementations of session types in other languages, while taking advantage of some of the unique features of Rust, to provide a practical, usable library that provides static guarantees about communication correctness. Our results show how session-typed communication, while requiring more careful implementation by the user, helps to enforce sound communication patterns that are less error-prone, a property of particular importance to large-scale, massively concurrent applications. The implementation also serves to demonstrate how a library for session-typed channels can be directly implemented in a polymorphic language that supports affine types, without having to worry about aliasing.

Finally, we consider the limitations of our session type design, which relies on the affine types of Rust. In an attempt to address the biggest limitation—channel end-points may be voluntarily or involuntarily dropped at any time—we implement a compiler plugin for Rust called `humpty_dumpty`, which attempts to enable the user to track linearity of specially annotated types.

Our contributions are: a library implementation of session types that relies on affine types, the `humpty_dumpty` plugin for the Rust compiler, which enables the user to track linear types, a fork of the Servo browser engine which incorporates session-typed channels as part of the internal communication schemes, contributions to both the Rust and Servo projects, and a paper submission currently in review about our work to the 11th ACM SIGPLAN Workshop on Generic Programming (WGP 2015).[1]

[1] `http://www.wgp-sigplan.org/2015`

Contents

List of Figures

List of Tables

List of Listings

1 Introduction

Since the first web browser was created in 1990 a lot has changed in the online world. A multitude of browsers have seen the light of day, some for a short time, others are still around to this day.

The most well-known browsers—Internet Explorer, Firefox, Chrome, Safari and Opera—all have long development histories, and this is particurlarly true of their core rendering engines. IE's Trident was initially released as part of IE 4.0 in 1997, the Gecko engine used by Firefox was introduced in 1998, WebKit, which was pioneered by the Konqueror browser but also used in both Safari and Chrome, was introduced in 2000, and, lastly, Opera's Presto engine was introduced in 2003, although it is now discontinued in favour of WebKit [3, 4, 5, 6, 7]. For a graphical timeline of browser history see [8, 9].

All the major browsers use rendering engines whose initial development began at a time where uni-processor computers were the default, but the hardware has changed dramatically in the past decade. In particular, the number of cores and processing units in PCs have increased dramatically and multi-core processors have become ubiquitous. Smart phones and other mobile devices are common-place too, and they also favor multiple processors, because using multiple but less powerful processors have been found to contribute to improved battery usage, given proper utilization.

A substantial effort has been (and is still being) put towards developing new tools and methods to program concurrent software, but it is a daunting task to adapt existing software from single-threaded to multi-threaded execution, especially large software projects such as a rendering engine. Additionally, all the aforementioned rendering engines are implemented in C++, a language not initially designed with multi-threaded programming in mind. Instead, support for multi-threaded programming in C++ has been added later through libraries. The result is, that although you can use concurrency in C++, the language itself offers very little help, and it is easy to create data races and the like.

A browser is a complex piece of software that requires precise memory control and good overall performance, so at the time, choosing C++ made sense. However, manual memory management requires the programmer to keep track of all allocated memory and release it as appropriate. This is a well-known source of errors such as *buffer overruns*, *use-after-free* and *double-free*, which constitute potential security vulnerabilities

Mozilla has decided to develop a new browser engine called Servo, with the stated goal of utilising multi-core architectures to the fullest and eliminating security vulnerabilities due to memory corruption. The safety requirement could be achieved by using a garbage-collected language, such as Haskell or D, but, as noted earlier, the need for precise memory control does not permit the use of a garbage-collected language. Instead Mozilla has opted to implement Servo in Rust, a new systems programming language also developed by Mozilla.

Rust is a systems programming language that uses the LLVM back-end to produce highly optimized executables. Rust is *memory-safe* by default, i.e. it disallows any kind of code that may result in memory corruption. It also supports affine types and message-passing primitives, the latter of which are implemented in the standard library, but take advantage of Rust's sophisticated type system in order to provide security from data races. Section 2 gives an introduction to Rust.

The memory safety combined with easy-to-use concurrency primitives enables the developers to structure the rendering engine in a way that maximizes parallel execution. Some of the novel features of Servo include concurrent DOM traversal, and concurrent layout painting and JavaScript execution.

Rendering a single web page spawns a number of threads working in cohesion to render the result as fast as possible. The various tasks involved in rendering a web page lend themselves somewhat naturally to concurrent implementation, for example, fetching resources, parsing HTML and CSS, decoding images and caching fetched resources. Other tasks are not so

straightforward to parallelize, such as the interaction between the DOM and JavaScript execution. Additionally, an HTML parser must be re-entrant, because inlined JavaScript may alter the DOM in various ways.

One of the challenges in the development of Servo is orchestrating the large number of interacting threads. Traditional message-passing is simple to use, but can lead to errors caused by miscommunication and Servo has experienced various problems with concurrency control such as deadlocks or miscommunication resulting in crashes. Crashes caused by miscommunication has proved to be a recurring problem and the development team has expressed an interest in tackling this problem in a structured way.

The current structure in communication makes it hard to get an overview of the communication patterns and the flow of information is not obvious. In terms of process management, there is a lack of clear division of responsibility and as a result Servo's shutdown sequence has proved to be problematic. The shutdown sequence is elaborate, because there are different modes of operation depending on whether the entire application or just a part of it is being shut down.

This dissertation will study the communication patterns in Servo and outline our requirements for a solution to the problems that we encounter, and it will examine the possibilities for implementing session-typed channels in a polymorphic programming language that supports affine types. Our working hypothesis is:

> *Session types can be implemented directly in a polymorphic programming language that supports affine types. Additionally, we can port a large-scale application, Servo, to a communication scheme based on session types, providing additional safety guarantees without incurring significant performance overhead.*

The structure of the dissertation is as follows: Section 2 gives an introduction to the Rust programming language with a focus on the features of the language that we will be using. Section 3 describes the overall architecture of Servo and details its internal communication patterns, outlines problems in the current approach and establishes a list of properties we would like in a solution. Section 4 explains session types in detail and Section 5 presents our implementation of session types in Rust, namely the `session-types` library. We also study some example use cases and implement additional useful infrastructure that makes it easier to work with session-typed channels in a practical setting. Section 6 details our approach to introducing session-typed communication in Servo. Section 7 discusses limitations in `session-types` and presents a compiler plugin, `humpty_dumpty`, which helps enforce linear usage of session-typed channels. Section 8 addresses performance concerns, assesses the current implementation, discusses alternative solutions and related work, describes possible avenues for future work, lists our contributions and summarizes our work.

2 Rust, a Memory-Safe Language

This section gives an introduction to Rust. It is not meant to be a comprehensive tutorial, but rather an introduction focusing on the features that are unique to Rust and those features that will play a role in our work.

Rust is a systems programming language, initially developed independently by Graydon Hoare before being adopted by Mozilla and developed on a larger scale. The main goals of Rust are safety and speed. Safety in this context is understood to be *memory safety*.

Because it is aimed at being a systems programming language, Rust is designed to offer a high level of control over low level details like memory allocation, so the programmer can reason about how the program will behave at run-time. In particular, this means that there is no garbage collection. To achieve safety in the absence of garbage collection, Rust employs compile-time static analysis techniques and semantics to determine exactly when an allocated value goes out of scope such that it can be freed. Thus, the Rust compiler decides, at compile-time, where to place destructors and drop statements, freeing the programmer from that responsibility, while guaranteeing memory safety.

Rust also incorporates features typically associated with functional languages of the ML family, such as pattern matching, algebraic data types, strong static typing, higher-order functions and closures. Syntactically, Rust is close to C, employing brace-delineated blocks for structure and semi-colons to mark the end of statements.

```
1  fn fact(n: usize) -> usize {
2      if n > 1 {
3          n * fact(n - 1)
4      } else {
5          1
6      }
7  }
8
9  fn main() {
10     let n = 4;
11     println!("Hello world. {}! = {}", n, fact(n));
12 }
```

Listing 1: Hello World and the factorial function in Rust.

Listing 1 shows a small Hello World program, as well as a definition of the factorial function. The `usize` and `isize` (not shown here) types are pointer-sized integers, unsigned and signed respectively. Rust is expression-oriented: the last statement of a block is the value of the block if not terminated by a semi-colon, and all functions and statements have a return type.[2] Variable bindings are introduced with the `let` notation. Variable bindings are immutable by default; you can explicitly mark a binding as mutable by using `let mut`. `println!` is a macro, indicated by the exclamation mark. An interesting fact is that macros are expanded and type checked at compile-time, and they are powerful enough that `println!` can statically check that it receives the appropriate number of arguments.

The development of Rust is community-driven and publicly available on GitHub.[3] At the time of writing, there is no official language specification, but there is a reference document that informally describes the language constructs, the memory model and other parts of the language [10]. Work has been done to provide a formalization of the Rust, with formal semantics and soundness proofs for a collection of core operations [11]. The Rust compiler, `rustc`, serves as a reference implementation.

[2] When terminated by a semi-colon, the value of the block is the unit value `()`

[3] `http://github.com/rust-lang/rust`

Section 2.1 introduces the ownership model and the move/copy semantics which form the basis of Rust's memory management scheme, as well as a vital part of our implementation of session-typed channels. Section 2.2 explains how structs, Rust's compound data structures, work, while Section 2.3 describe Rust's trait system, which provides parametric polymorphism. Section 2.4 details some of the concurrency primitives implemented in the Rust standard library, and finally, Section 2.5 presents the `unsafe` keyword, which provides a way to temporarily circumvent Rust's safety mechanisms.

2.1 Ownership and Move Semantics

Ensuring memory safety and eliminating data races are two of Rust's main goals. To achieve this, Rust uses two concepts called *ownership* and *move semantics*. This section will explain these two concepts.

Generally, ownership is about resource management. A resource could be a file handle, a socket, or a myriad of other things, but we will explain ownership in terms of memory management. Basically, every piece of allocated memory is uniquely owned by a scope, and at the end of a scope, any owned memory allocations are automatically deallocated. This ensures that all allocated memory is deallocated exactly once and in a predictable fashion. The idea of letting the scope determine the exact deallocation point for memory allocations is known as *region-based memory management* and was developed in [12] and implemented in the MLKit and Cyclone programming languages, among others [13, 14].

Ownership may be transferred. For example, a function may pass a variable to another function and hand over the ownership of the variable's value. The compiler tracks the ownership transfer and the calling function may no longer refer to the transferred variable. In Rust terminology the value has been *moved* and this behaviour is called *move semantics*. Move semantics are the default behaviour in Rust. For example, the `process_list` function in Listing 2 specifies that the function takes ownership of the `list` argument, a vector of bytes.

```
1 fn process_list(list: Vec<u8>) {
2     // ... At the end of this function ‘list‘ is deallocated
3 }
4
5 fn borrow_list(list: &Vec<u8>) {
6     // ... ‘list‘ is borrowed until the end of this function (but not deallocated)
7 }
```

Listing 2: Examples of owning and borrowed function arguments.

Move semantics contrast with the traditional behaviour of programming languages which is to copy function arguments from caller to callee. In Rust, this is referred to as *copy semantics* and is sometimes a useful behaviour. Copy semantics can be opted into for a given type by implementing the `Copy` marker trait (see Section 2.3 for an introduction to the trait system). All primitive types in Rust are `Copy` and thus subject to copy semantics.

When a function does not wish to take ownership of an argument, it may specify that the value is borrowed. For instance, the `borrow_list` function in Listing 2 receives a reference to a vector of bytes for the duration of the function. Borrows, also called references, come in two forms:

- Immutable, denoted by `&`. There may be multiple active immutable references at any given point
- Mutable, denoted by `&mut`. There may only ever be one active mutable reference and no immutable references at the same time.

In either case, the owner of a variable may not mutate its value while there are active borrows, and the owner may not read from the variable while a `&mut` reference is active.

The rules governing references are semantically equivalent to read-write locks, allowing concurrent access for read operations, whereas write operations require exclusive access. The borrowing rules thus prevent memory-safety issues in a single-threaded context.

References exist for a certain duration before they are dropped. This duration is called their *lifetime* and Rust's lifetime system ensures that references do not outlive their referent, otherwise the reference could eventually be referring to deallocated memory, which would be an error.

As it turns out, Rust's move semantics and borrowing system are not only useful for avoiding a wide variety of common memory errors, they also naturally extend to support safe multi-threaded programming without data races: Because all data is owned by exactly one lifetime scope, the only way to share data between processes is through borrows, and since mutable borrows exclude any other attempts at accessing the data, no two processes can modify a given piece of data at the same time, and if there is a process that is currently modifying some data, other processes cannot read potentially inconsistent states. The result is that Rust's borrows and lifetime system, combined with `Send` and `Sync` bounds, form the building blocks for constructing safe concurrency primitives such as those included in the standard library.

2.2 Structures

Compound data types are defined as structs. Listing 3 shows the declaration of a `Rectangle` struct, a representation of a rectangle.

```
1  struct Rectangle {
2      height: f64,
3      width: f64
4  }
5
6  impl Rectangle {
7      fn new(h: f64, w: f64) -> Rectangle {
8          Rectangle {
9              height: h,
10             width: w
11         }
12     }
13     fn area(&self) -> f64 {
14         self.height * self.width
15     }
16 }
```

Listing 3: Example of a struct declaration and method implementation.

Methods are defined on structs in separate `impl` blocks. In Listing 3 we add two methods to the `Rectangle` struct, `new` and `area`. The `new` method is called a static method and is invoked as `Rectangle::new(h, w)`. It is a common idiom to define constructors like this, but it is not enforced by the language—`new` could have been called anything. The `area` method can only be invoked on instances of the `Rectangle` type, indicated by the `&self` argument. The explicit `self` argument is a familiar concept from languages such as Python, but in Rust the programmer must additionally specify how the `self` parameter is passed: either it is moved (`self`) or obtained by reference (`&self` or `&mut self`).

2.3 Traits

Before we can introduce the trait system, we will briefly go over Rust's support for generics. Generics implement parametric polymorphism, and allows functions to take and return values without knowing their type. For instance, Listing 4 shows a (rather silly) function that takes

```
fn optionify<T>(x: T) -> Option<T> {
    Some(x)
}
```

Listing 4: The function `optionify` takes a generic argument and wraps it in an `Option` container.

an `x` of any type (indicated by the generic type parameter `T`) and wraps it in an `Option` container. We refer the reader to [15] for a full introduction to generics in Rust.

Traits in Rust are primarily a method of placing bounds on generics, and serve much the same role as interfaces in Java or, perhaps more closely, typeclasses in Haskell. As such, the trait system is used to implement ad-hoc polymorphism.

```
trait HasArea {
    fn area(&self) -> f64;
}

impl HasArea for Rectangle {
    fn area(&self) -> f64 {
        self.height * self.width
    }
}

fn print_area<T: HasArea>(shape: T) {
    println!("The shape has an area of {}", shape.area());
}
```

Listing 5: Trait declaration, implementation and usage.

A trait is declared with the `trait` keyword, a name and a block containing method declarations, as demonstrated in Listing 5. The trait `HasArea` contains a single method declaration, `area`, with no body. Method declarations in traits can optionally provide a default implementation by simply providing a body, much like virtual methods in C++ or abstract class methods in Java. For example, the `Iterator` trait from Rust's standard library provides a host of different ways to iterate over data with default implementations, but requires the implementation of a `next` method to obtain the next item in the iterator.

A trait is implemented for a type in a separate `impl` block, reminiscent of method implementations, written as "`impl Trait for Type`". In Listing 5 we implement `HasArea` for `Rectangle` and provide a body for the `area` method.

A function can then specify a *trait bound* on generic parameters, requiring a generic argument to implement a given trait. In Listing 5 the `print_area` function only accepts types that implement the `HasArea` trait. This allows the function to invoke the `area` method irrespective of the specific argument type.

Multiple traits can also be specified in a single bound by separating the trait names with "+". For example `A: Debug + Clone` dictates that the type `A` must implement both the `Debug` and `Clone` traits. Where `A` is then used, all methods declared in both traits can be invoked.

Trait bounds may also be provided in *where clauses* as shown in Listing 6 as a rewritten version of `print_area`. At first this may seem like redundant syntax, but `where` clauses are more expressive than regular trait bounds. Specifically, they allow bounds where the left-

hand side is an arbitrary type—see Listing 7 for an example. The increased expressibility afforded by `where` clauses allows us to establish more complex relationships between types.

```rust
fn print_area<T>(s: T) where T: HasArea
{
    println!("Area: {}", s.area());
}
```

Listing 6: Using `where` clauses.

```rust
// This is invalid
fn foo<i32: Trait<T>>(x: i32) {
    /* body */
}

// But this is valid
fn foo<T>(x: i32) where i32: Trait<T> {
    /* body */
}
```

Listing 7: Example use of `where` not possible with regular trait bounds.

Since their inclusion, traits have found a wide variety of uses. Indeed, in Rust itself they are used for the majority of binary and unary operators (e.g. the `Add` trait), copy semantics (`Copy`), thread-safety (`Send` and `Sync`), and many other things.

2.3.1 Associated Types

An extension to Rust's trait system are *associated types*. It stems from the idea of type families and grouping multiple types together.

The canonical example from the RFC[4] and [15] is the definition of a `Graph` trait, which is generic over its nodes and edges. Consider Listing 8 which defines the `Graph` trait with associated functions `has_edges` and `edges`, and a function `dist`. It is evident that writing functions that take graphs as arguments quickly becomes cumbersome; even though `dist` is not using the edges of the graph, it must still specify a type parameter for edges to use `G`.

```rust
trait Graph<Node, Edge> {
    fn has_edge(&self, &Node, &Node) -> bool;
    fn edges(&self, &Node) -> Vec<Edge>;
}

fn dist<N, E, G: Graph<N, E>>(g: &G, start: &N, end: &N) -> u64 {
    // ...
}
```

Listing 8: A graph defined as a trait.

Alternatively, we may use associated types, as demonstrated in Listing 9. Associated types allow us to rewrite the `Graph` trait, such that the `Node` and `Edge` types depend on the implementation of the `Graph` trait. Now, to implement the `dist` function we only have to specify that `G` implements `Graph`, and we may freely refer to the associated types `Node` and `Edge`.

Since their inclusion, associated types have also shown to open up for more sophisticated implementations for the binary operators, defined as traits. For example, the addition operator "+" is defined by the trait declaration in Listing 10. The declaration of `Add` allows definitions for addition across types. For example, a representation of complex numbers in a `Complex` struct might conceiveably implement the following versions of `Add`:

```rust
impl Add<Complex> for Complex { type Output = Complex; ... }
impl Add<Complex> for u64 { type Output = Complex; ... }
impl Add<u64> for Complex { type Output = Complex; ... }
```

[4] https://github.com/rust-lang/rfcs/blob/master/text/0195-associated-items.md

```
trait Graph {
    type Node;
    type Edge;
    fn has_edge(&self, &Node, &Node) -> bool;
    fn edges(&self, &Node) -> Vec<Edge>;
}

fn dist<G: Graph>(g: G, start: &G::Node, end: &G::Node) -> u64 {
    // ...
}
```

Listing 9: A graph trait using associated types.

```
pub trait Add<RHS=Self> {
    /// The resulting type after applying the '+' operator
    type Output;

    /// The method for the '+' operator
    fn add(self, rhs: RHS) -> Self::Output;
}
```

Listing 10: The `Add` trait.

which allows the programmer to use the addition operator to not only add complex numbers to each other, but also add `u64`s to complex numbers. The trait system will infer at the call site which implementation to invoke.

2.4 Concurrency

Early versions of Rust had concurrency features built into the language, but it turned out that the requirements for building safe concurrency mechanisms could be fulfilled through the use of ownership, lifetimes, and two marker traits, `Sync` and `Send`, which specify what types can be safely shared between threads and exchanged between threads respectively.[5]

A type `T` is said to be `Sync` if `&T` can be used concurrently without introducing memory unsafety or data races. This is true of all the primitive types and most container types. Generally, types that are `Sync` adhere to *inherited mutability*. Inherited mutability means that the mutability of a variable binding for a struct value is inherited by the struct fields, i.e. the state of a struct value cannot be modified if the variable binding is not declared mutable (`mut` or `&mut`). By contrast, types that provide *interior mutability* can be mutated through immutable bindings. Such types are in general not thread-safe and therefore should not implement `Sync`.

For instance, consider Listing 11, which contains a declaration of a `Counter` struct and a method for said struct, `add1`. The `add1` method takes an immutable reference to a `Counter`, but you will notice that it actually increments the internal counter: it uses interior mutability, so it should not implement `Sync`.

A type `T` is `Send` if transferring ownership of the value into another thread cannot result in a data race. This is true for almost all data types except for the raw pointers types `*const T` and `*mut T`, which we will touch upon in Section 2.5

Usually, Rust will automatically implement `Sync` and `Send` and complain in cases where automatic implementations cannot be derived (but are required). The inference are relatively straight-forward, defined inductively for compound data types, i.e. a data type is `Send` if all

[5] For more information about how Rust's ownership model facilitates concurrency in a safe manner, see `http://blog.rust-lang.org/2015/04/10/Fearless-Concurrency.html`

```
struct Counter {
    x: RefCell<usize>,
}

impl Counter {
    fn add1(&self) {
        let mut ptr = self.x.borrow_mut();
        *ptr += 1;
    }
}
```

Listing 11: A struct which has interior mutability.

its constituent parts are `Send`. This rules out any type containing raw pointers.

In order to support concurrency, the Rust standard library provides mechanisms for both message passing and shared memory communication between threads, as well as a wide variety of synchronization primitives. Our work relies mostly on message passing, which is implemented through multi-user, single-consumer FIFO channels. Channels are created using the function:

```
fn channel<T: Send>() -> (Sender<T>, Receiver<T>)
```

The return value contains two endpoints—one endpoint is used for sending values, the other for receiving them. This separation of the end-points is a consequence of Rust's ownership model, as the threads can be allowed to own their half of the channel. It also serves to enforce the direction on the channel (they are uni-directional) and supports the design choice that `Sender`s can be cloned, but `Receiver`s cannot. This eliminates any potential ambiguity of who will actually receive a value—the design ensures that for a given `Sender` there is a unique `Receiver`.

The net result of the `Sync` and `Send` abstractions is that the choice of concurrency primitives is not dictated by the language, but rather enables the user to implement and explore different paradigms.

2.5 Unsafe Code

Thanks to static analysis and borrow checking, the majority of Rust code is memory safe. However, sometimes it is necessary to circumvent Rust's safety mechanisms in order to express safe programs that cannot be statically verified. For purposes like this, Rust provides the **`unsafe`** keyword (see Listing 12), which allows the programmer to mark a block, function, method implementation, or trait as unsafe. An example is the `transmute` function that we will describe in more detail later.

```
unsafe fn read<T>(src: *const T) -> T {
    // all code here may is considered unsafe
}

// Calling an unsafe function
fn main() {
    let n: *const u32 = &42;
    let m = unsafe {
        read(n)
    };
}
```

Listing 12: Unsafe function declaration.

Calling an unsafe function can only be done from another unsafe function or from inside an unsafe block. Using an unsafe block is demonstrated in Listing 12, where we also see the declaration of a raw pointer type, the `*const T` (the mutable variety is `*mut T`). Raw pointers are exempted from all of the rules imposed on references:

- They are allowed to freely alias.
- They can be null and are not guaranteed to point to valid memory.
- They require manual memory management, as there is no automatic clean-up.
- They do not move ownership.
- They do not have lifetimes.

In short, raw pointers are directly translatable to pointers in C, and as a result they are considered unsafe to dereference.

It may be tempting to dismiss the safety efforts implemented in Rust if the language then provides a way to disable all the safety mechanisms, but **unsafe** exists because it allows some common use cases that are provably safe, but not verifiable by the borrow checker. An example of this is the doubly-linked list data structure. Providing pointers in both directions in a list requires the pointers to alias, but this is prohibited for mutable references. For shared references, the borrow checker cannot infer appropriate lifetimes for the references. Hence, raw pointers are required to build a doubly-linked list. In short, unsafe code should only be used when Rust cannot statically verify a piece of code that we know to be safe. The programmer may temporarily break memory consistency in an **unsafe** block or function, but it is her responsibility to ensure that consistency is restored at the end of the block.

The **unsafe** keyword also serves as a visual cue to programmers that potentially memory-unsafe operations take place, and applications that experience problems with memory management, such as use-after-free or dangling pointers, should be able to locate their errors in **unsafe** blocks. Being able to isolate and label potentially memory-unsafe operations is a huge advantage when dealing with memory-safety issues.

There are cases where unsafe operations are preferable, because they allow an implementation that is faster than the verifiably safe equivalent, and many of the abstractions provided in Rust's standard library use unsafe operations to construct safe interfaces.

transmute In the later chapters we will require the ability to interpret one type as another and Rust provides a built-in function `transmute` to perform that operation. Its declaration is:

```
pub unsafe extern "rust-intrinsic" fn transmute<T, U>(e: T) -> U;
```

The `transmute` function reinterprets a type `T` as some other type `U` with the restriction that the types must have the same size in memory. This operation is not safe in general, so the function is marked as **unsafe**. As example usage of `transmute` from [15] is shown in Listing 13 showing that we can interpret an array of four bytes as a 32-bit number. In Rust, an array has a fixed size that is part of the type and therefore the Rust compiler can verify that the sizes are the same.

```
1 use std::mem;
2 unsafe {
3     let array: [u8; 4] = [0u8, 0u8, 0u8, 0u8];
4
5     let number = mem::transmute::<[u8; 4], u32>(array);
6 }
```

Listing 13: Interpreting an array of four bytes as a 32-bit integer.

3 Servo, a Concurrent Browser

This section describes the architecture and internal communication patterns of Servo. It highlights some problems in the communication structure and discusses how they may be fixed. Section 3.1 presents an overview of the architecture of Servo and Section 3.2 explains its communication patterns in greater detail. We then go on to discuss how the communication strategy, although simple to implement, is difficult to maintain in an evolving piece of software. A challenging part of the communication strategy is orchestrating shutdown, and in Section 3.3 we discuss in detail how this is handled in Servo. Finally, Section 3.4 outlines our requirements for a solution to the challenges that we identify.

3.1 Servo and its Architecture

This section describes the overall workflow in Servo and the different tasks involved in rendering a web page.

When an HTML document is loaded, it is parsed and turned into a DOM tree. Associated styling information, through inlined or linked CSS documents, is also loaded and a styled tree, called a *flow tree*, is computed with styling information. The flow tree is then processed to produce a set of *display list* items, which represent the actual graphical elements in their final on-screen position. Once all the elements to appear have been computed, they are rendered onto *layers*, that are either a set of memory buffers or graphical surfaces. These layers are then composited together to produce the final image for presentation. Throughout the different passes of rendering a page, JavaScript code may execute at any time, and these scripts may modify the DOM tree, which requires reruns of the layout and painting task.

The design of Servo is focused on taking advantage of opportunities to parallellize as many tasks as possible. One of the unique achievements in Servo is the concurrent execution of the rendering and compositing phase, where traditionally these phases are implemented sequentially, because script execution may interact with the DOM in non-obvious ways.

In Servo, there are a number of different tasks involved:

Compositor Handles compositing and relaying events from the window system to the constellation, while receiving messages from the constellation and paint task.

Constellation Conceptually, the constellation corresponds to a single tab or window in a browser. It manages pipelines for rendering and handles events from its script tasks and the compositor.

ScriptTask Owns the DOM in memory, runs JavaScript and spawns parsing tasks.

LayoutTask Performs layout on the DOM, builds display lists and sends these to be painted. The display list is a sequence of high-level drawing commands that can be sent to the paint task.

PaintTask Handles all painting. It handles three-way communication with the compositor, constellation and layout task.

The constellation constructs a new pipeline for each page load. A pipeline in Servo comprises a script task, a layout task and a paint task, so when we talk about a "pipeline" we are talking about this particular grouping. Pipelines are organized in a tree to handle iframes (called a *frame tree*). All pages have a top-level frame and iframes can be arbitrarily nested inside the frame. Modern websites use iframes not only for content, but for ads, trackers and various social media buttons.

On receiving a new URL to load, the constellation creates a new pipeline with associated script, layout and paint tasks. The actual fetching of resources and parsing is then delegated to the script task. Loading a page in a script task consists of two steps: The first step is initiating a network request to fetch the named resources. This is implemented as a

non-blocking procedure and while it is underway, the script task is free to handle incoming events from other sources. It keeps track of incomplete loads to defer certain actions until their completion. The second step is parsing, which is instantiated as soon as the loading completes.

Although the script task owns the DOM structure, the layout task performs layout on the DOM at the request of the script task. The paint task communicates primarily with the layout task and compositor performing painting on request.

The above-mentioned tasks are central to the rendering pipeline in Servo. There are other tasks assisting these central tasks mostly in terms of resource management; the resource task, image cache task, font cache task, storage task and time and memory profiling tasks (plus some others) are all "global" tasks. They are instantiated by the constellation and live until the constellation shuts down, and can be viewed as servers in a more traditional client-server setting. All the pipeline tasks (script, layout and paint) hold references to some or all of these tasks and query them for various bits of information.

It is not uncommon in Servo for anonymous worker threads to perform some work asynchronously. As an example, we will look at the `start_page_load` function (Listing 14) called by the script task on initialization and when new pages are loaded. Instead of blocking on a `Load` message, it defers the workload to another thread, by a call to `spawn_named`. The script task is then free to respond to other requests while the resource task handles the `Load` message. On completion, the worker thread passes the result returned by the resource task to the script task by another message, `PageFetchComplete`.

```
1  fn start_page_load(&self, incomplete: InProgressLoad, mut load_data: LoadData) {
2      // Channels
3      let script_chan = self.chan.clone();
4      let resource_task = self.resource_task.clone();
5
6      spawn_named(format!("fetch for {:?}", load_data.url.serialize()), move || {
7          let (input_chan, input_port) = channel();
8          resource_task.send(ControlMsg::Load(NetLoadData {
9              // Fields with values for NetLoadData
10             consumer: input_chan,
11         })).unwrap();
12
13         let load_response = input_port.recv().unwrap();
14         script_chan.send(ScriptMsg::PageFetchComplete(id, subpage, load_response))
15            .unwrap();
16     });
17     self.incomplete_loads.borrow_mut().push(incomplete);
18 }
```

Listing 14: Initiating a page load.

3.2 Communication Patterns

This section describes the communication patterns employed in Servo today and attempts to describe how the current method does not prevent common communication problems.

The canonical method employed in Servo to structure communication is simple: A task exposes an API through an enum type, describing the different kinds of messages it can receive. A simple example is the `ResourceTask` defined in Listing 15. The `ControlMsg` enum type defines four variants, each corresponding to a message.

An interesting variant of `ControlMsg` is the second field of `GetCookiesForUrl` of type `Sender<Option<String>>`. It showcases another common communication pattern where the requester passes in a channel

```
type ResourceTask = Sender<ControlMsg>;

enum ControlMsg {
    Load(LoadData),
    SetCookiesForUrl(Url, String, CookieSource),
    GetCookiesForUrl(Url, Sender<Option<String>>, CookieSource),
    Exit
}
```

Listing 15: The `ResourceTask` and its message API.

```
loop {
    match self.from_client.recv().unwrap() {
        ControlMsg::Load(load_data) => {
            // load data ...
        }
        ControlMsg::SetCookiesForUrl(request, cookie_list, source) => {
            // set cookie ...
        }
        ControlMsg::GetCookiesForUrl(url, consumer, source) => {
            // get cookie ...
            consumer.send(maybe_cookie);
        }
        ControlMsg::Exit => {
            break
        }
    }
}
```

Listing 16: Handling control messages in `ResourceManager`.

on which to send a response.

The `ResourceManager`, when spawned, loops indefinitely receiving control messages. See Listing 16. This pattern, although readable, suffers a simple flaw: What if a thread sent it `ControlMsg::Exit` before other clients were finished? Any thread holding a `ResourceTask` handle may access this API, and there is no way to keep track of the number of clients still wishing to communicate with the `ResourceManager`. In particular, there is no visibly assigned responsibility for sending `ResourceManager` the shutdown signal, meaning any thread could potentially assume that responsibility.

The resource manager (or any receiving thread) cannot control who sends a given message nor control who may send it messages. Although this is typically desirable to reduce the coupling between components in a system, it would be disastrous if any thread other than the constellation sent an `Exit` message to the resource manager. The problem here is that there are certain implicit assumptions concerning who sends which type of messages. It should only be the constellation that shuts down the resource manager, but this restriction is not reflected in the resource manager's API. If a developer is not aware of such assumptions or restrictions, she can only rely on documentation to ensure she follows the protocol correctly. In other words, the responsibility for behaving correctly with respect to an implicit protocol is left in the hands of the developer and this discipline becomes increasingly difficult to maintain as a project and its communication APIs grow.

The internal communication protocols have evolved gradually as Servo has grown. If a new type of message was required from one task to another, it was added and handled appropriately in other places where it required handling. As the number of inter-communicating tasks have grown, these communication schemes become difficult to maintain and extend, a

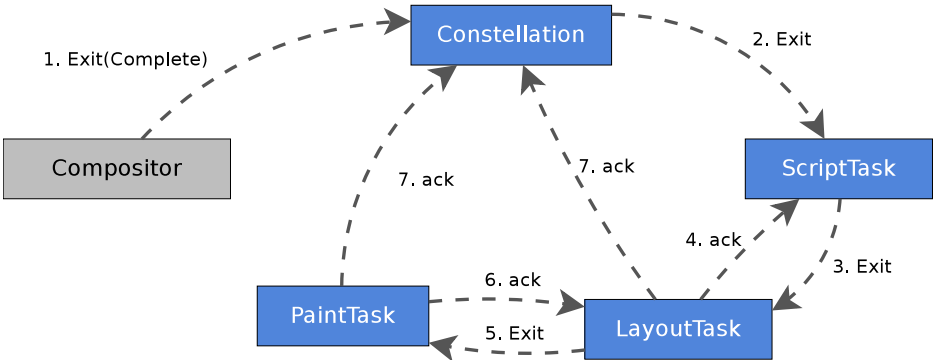


Figure 1: The shutdown sequence for a pipeline for a complete shutdown. `ack` messages are "acknowledgment" messages of type `()`.

condition that has been referred to as "protocol soup".

3.3 The Shutdown Sequence

This section describes how Servo handles shutting down single pipelines and complete shutdowns. The shutdown sequence is important, because deterministically shutting down a host of intercommunicating processes is no simple task, and it has proved to be problematic in Servo.

Throughout Servo's lifetime pipelines are created and shut down. Pipelines are created by the constellation to handle new page loads and form a chain that reflects the current browsing history. Navigating backwards (usually done by pressing the backspace key) does not create a new pipeline, but instead fetches an existing pipeline. Pipelines are shut down when they become part of a branch in the history no longer accessible by backwards and forwards navigation.

From the point of view of a pipeline, there are two shutdown "modes", defined by the `PipelineExitType` enum type:

- `PipelineOnly`: Only this pipeline is being shut down. The only tasks to be shut down are its associated script, layout and paint tasks. As this action was initiated by the constellation, the compositor is notified of exited pipelines (`PaintTaskExited`).
- `Complete`: The entire application is shutting down. This action originates either from the compositor or the script task. The script task routes this message via the compositor, so it appears to originate from the compositor. In this case the compositor does not need to be notified of exited pipelines and it only waits for confirmation that the constellation has exited.

The exit mode has implications for the behaviour of the tasks being shut down. In a pipeline-only shutdown the paint task will wait for any layer buffers the compositor is currently borrowing to be returned to it before terminating. If the paint task did not wait, the buffers would leak. During a complete shutdown the paint task will not wait, because all resources used by Servo (including would-leak buffers) will be collected by the operating system when the application exits.

In a complete shutdown, all pipelines are requested to shut down (by the constellation) and the flow of `Exit` messages follows the pattern presented in Figure 1 (for a single pipeline). Each task in the chain awaits acknowledgment from the next task in the chain before terminating. The script task has the responsibility for shutting down the layout task and the layout task in turn shuts down the paint task. The pipeline awaits confirmation of shutdown from both the layout and paint task on a separate port.

```
/// Enters a quiescent state in which no new messages except for
/// `layout_interface::Msg::ReapLayoutData` will be processed until an `ExitNow`
/// is received. A pong is immediately sent on the given response channel.
fn prepare_to_exit<'a>(&'a self, response_chan: Sender<()>) {
    response_chan.send(()).unwrap();
    loop {
        match self.port.recv().unwrap() {
            Msg::ReapLayoutData(dead_layout_data) => {
                // ... Handle dead layout data
            }
            Msg::ExitNow(exit_type) => {
                debug!("layout task is exiting...");
                self.exit_now(exit_type);
                break
            }
            _ => {
                panic!("layout: message that wasn't `ExitNow` received after \
                        `PrepareToExit`")
            }
        }
    }
}

/// Shuts down the layout task now. If there are any DOM nodes left, layout will now
/// (safely) crash.
fn exit_now<'a>(&'a self, exit_type: PipelineExitType) {
    let (response_chan, response_port) = channel();

    // ... omitted shutdown of `possibly_locked_rw_data`

    self.paint_chan.send(PaintMsg::Exit(Some(response_chan), exit_type));
    response_port.recv().unwrap()
}
```

Listing 17: The two-step shutdown logic for the layout task.

The normal shutdown sequence for the layout task is two-step: It first receives a `PrepareToExit` message and enters a state in which it only accepts two types of messages: `ReapLayoutData` and `ExitNow`. This stage exists to ensure no memory is leaked. The corresponding methods handling these messages are shown in Listing 17. There are a few observations to be made here: Firstly, the layout task does not request the paint task to exit before `ExitNow` is received, so at the time the paint task receives the exit message, the layout is ready to exit. Secondly, whilst waiting to receive the `ExitNow`, the paint task will panic if any other message type than `ReapDataLayout` or `ExitNow` is received. This behaviour makes the layout task brittle, because there is no explicit mechanism in place to ensure this does not happen. It is the responsibility of the developer to ensure this scenario does not occur, and in a concurrent, non-deterministic context it can be a difficult promise to keep. From the viewpoint of tasks wishing to interact with the layout, it would appear as if the layout task may behave differently over time when receiving the same type of message. Ideally, its protocol should have changed and the interacting parties been notified.

3.3.1 An Example of Communication Inconsistency

There is an alternate shutdown scheme for a single pipeline. In case a task in a pipeline has crashed and the pipeline must be forcibly closed, the constellation can invoke `force_exit` on

the pipeline. An annotated version of the `force_exit` method is shown in Listing 18.

```
pub fn force_exit(&self) {
    // Send 'ExitPipeline' to script task
    let ScriptControlChan(ref script_channel) = self.script_chan;
    let _ = script_channel.send(
        ConstellationControlMsg::ExitPipeline(self.id,
                                              PipelineExitType::PipelineOnly)).unwrap();

    // Send 'Exit' to paint task
    // 'None' indicates the pipeline does not want confirmation of shut down
    let _ = self.paint_chan.send(PaintMsg::Exit(None, PipelineExitType::PipelineOnly));

    // Send 'ExitNowMsg' to layout task
    let LayoutControlChan(ref layout_channel) = self.layout_chan;
    let _ = layout_channel.send(
        LayoutControlMsg::ExitNowMsg(PipelineExitType::PipelineOnly)).unwrap();
}
```

Listing 18: The `force_exit` method on pipelines.

In this forced exit, the pipeline sends exit messages to all tasks, instead of proceeding in the circular fashion shown in Figure 1. This is necessary, because any of its tasks could have crashed and it is not known which. In particular, the layout task is asked to exit immediately, skipping the intermediate data reaping stage, i.e. skipping `prepare_to_exit` and going straight to `exit_now`. Regardless, the layout task will ask the paint task to exit—as will the pipeline. This following communication sequence will thus lead to a crash:

1. `pipeline.force_exit()` is invoked. Pipeline sends exit messages to script, layout and paint tasks
2. Layout task receives `ExitNow` message from pipeline
3. Paint task receives pipeline-only exit message from pipeline
4. Paint task has zero outstanding buffers and exits immediately
5. Layout task sends exit message to paint task and awaits response

The layout task will crash at this point, because the paint task already exited. Subtle differences in behaviour and timing can avoid this crash: If the paint task waits for more buffers, it can still receive another `Exit` message and thus respond to the layout task. The paint task may also happen to process the message from the layout task before the message from the pipeline, avoiding the crash situation. We note that we have not observed a crash caused by this scenario, but it is evident that this scenario is avoided by chance and not by careful design.

3.4 Finding a Solution

The problem we would like to address is the problem of process crashes caused by miscommunication. The tasks running in Servo are not expected to crash, and we should make efforts to prevent them from crashing. Our goal is to reduce the risk of crashes by making the communication schemes safer.

We list the following requirements for a solution. It should:

1. Be possible to "specify" the communication schemes in advance to specify what interaction patterns are allowed.
2. Provide static guarantees that interaction patterns cannot be violated.
3. Be possible to implement in Rust, as a library.

4. Be usable in Servo

We prefer static guarantees in item 2 over run-time checks for performance reasons. Servo's internal communication should be fast and not overly burdened. To implement changes in Servo, it should be possible to provide a solution as a Rust library (a Rust library is called a *crate*).

Considering our requirements, the nearest candidate for a solution is the *session type* theory. Session types enhance channel-based communication with a form of protocol specification and through its type discipline ensures that only compatible communication patterns can occur.

Session types are intriguing, because the definition lends itself to embeddings in other programming languages, but the requirement of linear handling of sessions requires special attention in most programming languages, because this restriction is not straight-forward to encode. We introduce the session type theory in detail in the next section.

4 Session Types

This chapter properly describes the session type theory which underpins the rest of this dissertation. We provide an overview of the session type language $\mathcal{L}$ and its type system.

Session types were first introduced in [16] and provided a new method for structuring sequences of reciprocal interactions in a type-safe manner. Concurrency primitives based on message-passing have been studied greatly and implemented in a host of programming languages, but typically each interaction is considered distinct and unrelated to other interactions. The session type theory introduced a basic structuring concept called a *session*. A session has an associated (implicit) channel over which all interactions take place and the interactions via a session are modelled by a type—called a *session type*. The typing system then ensures that two processes only communicate via a session if their session types are compatible. Session types allow the programmer to specify complex communication protocols without worrying about run-time errors caused by violations of the protocol.

As a concept session types are suitable for embedding in other programming languages, but were initially developed in its own language $\mathcal{L}$ to clearer showcase the novel features of the system in a minimal language. Since their introduction, session types have been studied widely and re-formulated in other formal frameworks, in particular the π-calculus [2, 17], implemented in programming languages, both as libraries and language extensions [18, 19].

Section 4.1 gives a general introduction to the Honda-Vasconcelos-Kubo session typing system, and describes the language $\mathcal{L}$ in which it is implemented, while Section 4.2 introduces the typing system used in $\mathcal{L}$. Section 4.3 describes the implementation of session types in Haskell first published in [18], which forms part of the inspiration for our implementation of session types in Rust. Finally, Section 4.4 describes SessionJ, an implementation of session types for Java, which has also served as a source of inspiration for our library.

4.1 The Honda-Vasconcelos-Kubo Session Typing System

To initialize a session in $\mathcal{L}$ the primitives `request` and `accept` must be used. Given a port a, which we can think of as a port in the traditional message-passing way, "`request` $a(k)$ `in` P" requests a session to be used in the process P, binding it to the name k. "`accept` $a(k)$ `in` P" accepts incoming session requests bound to the name k and proceeds with the process P. The syntax of $\mathcal{L}$ is shown in Figure 2.

A session conceptually contains an implicit channel through which all interaction takes place. The basic interactions are send and receive, denoted by $![\tilde{e}]; P$ and $?[\tilde{e}]$ `in` P respectively. These operations are equivalent to send and receive in other message-based concurrency systems. Session interactions also include *labelled branching* and *branch selection*. Branch selection is denoted $k \triangleright l; P$ and selects the branch labelled l offered by the opposing communicating party and continues with the process P. The converse of branch selection is labelled branching in which the process offers a choice of processes to continue with. It is denoted $k \triangleleft \{l_1 : P_1, \ldots, l_n : P_n\}$ and waits for the other process to select a branch by one of the labels l_i, and then continues with the process P_i.

Session channels are not first class values in $\mathcal{L}$, meaning that we cannot send session channels over session channels, so *session delegation* is handled explicitly. Session delegation uses the `throw` and `catch` keywords to pass a session channel from one process to another in a type-safe manner. We will not discuss session delegation in great detail, as our implementation can implicitly handle session delegation without special support.

Apart from session interaction $\mathcal{L}$ also includes the basic types integers and booleans, and the basic control flow structures for conditional branching and recursion. Recursion is denoted by `def` D `in` P, where D can occur in P. $P \mid Q$ denotes parallel composition and `inact` is the inactive process. Sequencing is not explicitly included as each communication primitive contains a notion of a "next" action. The exceptions to this are the conditional

$$
\begin{array}{rll}
P ::= & \texttt{request}\ a(k)\ \texttt{in}\ P & \text{session request} \\
| & \texttt{accept}\ a(k)\ \texttt{in}\ P & \text{session acceptance} \\
| & k\,![\tilde{e}]; P & \text{data sending} \\
| & k\,?(\tilde{e})\ \texttt{in}\ P & \text{data receiving} \\
| & k \rhd l; P & \text{label selection} \\
| & k \lhd \{l_1 : P_1, \ldots, l_n : P_n\} & \text{label branching} \\
| & \texttt{if}\ e\ \texttt{then}\ P\ \texttt{else}\ Q & \text{conditional branch} \\
| & P \mid Q & \text{parallel composition} \\
| & \texttt{inact} & \text{inaction} \\
| & \texttt{def}\ D\ \texttt{in}\ P & \text{recursion} \\
| & \texttt{throw}\ k[k']; P & \text{channel sending} \\
| & \texttt{catch}\ k(k'); P & \text{channel receiving} \\
| & (\nu u)P & \text{name/channel hiding} \\
| & X[\tilde{e}\tilde{k}] & \text{process variables} \\
e ::= & c & \text{constant} \\
| & e + e' \mid e - e' \mid e \times e' \mid \texttt{not}(e) \mid \ldots & \text{operators} \\
D ::= & X_1(\tilde{x}_1\tilde{k}_1) = P_1\ \texttt{and}\ \cdots\ \texttt{and}\ X_n(\tilde{x}_n\tilde{k}_n) = P_n & \text{declaration for recursion}
\end{array}
$$

Figure 2: Syntax of $\mathcal{L}$.

branch, parallel composition, process variable and `inact` constructs. The next action of a conditional branch is either P or Q, and the next action of a process variable is the substituted process. Parellel composition and `inact` do not have a notion of a next action.

The *name hiding* primitive cannot be used explicitly in programs, but is necessary for the operational semantics to denote the establishment of a shared, hidden channel between two processes:

$$\texttt{accept}\ a(k)\ \texttt{in}\ P \mid \texttt{request}\ a(k)\ \texttt{in}\ Q \longrightarrow (\nu k)(P \mid Q) \qquad \text{[LINK]}$$

We will not state all the operational semantics rules, but refer the reader to [20]. This concludes our introduction of the syntax and we give an example program fragment of an arithmetic server:

$$\begin{aligned}\textsc{Arith} = \texttt{accept}\ a(k)\ \texttt{in}\ k \lhd \{&\texttt{add} : k\,?[a]\ \texttt{in}\ k\,?[b]\ \texttt{in}\ k\,![a+b]; \texttt{inact}, \\ &\texttt{neg} : k\,?[a]\ \texttt{in}\ k\,![-a]; \texttt{inact}\}\end{aligned}$$

The server accepts a session and offers a choice between two operations, `add` and `neg`. The first operation receives two numbers, a and b, and sends back their sum, and the second receives one number and sends back its negation. At the end of either branch the session terminates and no more progress can be made.

We can construct a client to interact with the arithmetic server and write a full program as follows:

$$\textsc{Arith} \mid \texttt{request}\ a(k)\ \texttt{in}\ k \rhd \texttt{add}; k\,![\texttt{2}]; k\,![\texttt{5}]; k\,?[r]\ \texttt{in}\ P$$

Note that the communication patterns of the client and Arith are compatible. The client requests a session k through the shared port a, selects the `add` branch and sends the numbers

2 and 5. It then obtains the answer (hopefully 7) bound to the name r and continues with the process P. As the session has been terminated, the process P cannot contain any interactions over the session channel k.

Interacting with the arithmetic server is unergonomic if the client wants to perform multiple additions or negations, because the client would have to repeatedly request new sessions. We can improve the arithmetic server to recurse after an operation and provide a third branch to let the client decide when to terminate the session:

$$\begin{aligned}\textsc{Arith'} = \texttt{accept}\ a(k)\ \texttt{in def}\ \textsc{Ops}(k) = k \lhd \{\ & \texttt{add} : k\,?[a]\ \texttt{in}\ k\,?[b]\ \texttt{in}\ k\,![a+b]; \textsc{Ops}[k] \\ & \texttt{neg} : k\,?[a]\ \texttt{in}\ k\,![-a]; \textsc{Ops}[k] \\ & \texttt{quit} : \texttt{inact}\} \\ \texttt{in}\ \textsc{Ops}[k]\end{aligned}$$

One of the primary concerns for any implementation of session types is aliasing. Aliasing a session can lead to communication inconsistencies. For example, consider the following program:

$$\begin{aligned}&\texttt{request}\ a(k)\ \texttt{in}\ k\,![42]; \texttt{inact} \\ \mid\ &\texttt{accept}\ a(k)\ \texttt{in request}\ b(k')\ \texttt{in throw}\ k'[k]; k\,?[x]; \texttt{inact} \\ \mid\ &\texttt{accept}\ b(k')\ \texttt{in catch}\ k'[k]; k\,?[x]; \texttt{inact}\end{aligned}$$

This program shares the session k between the second and third process, and both processes execute $k\,?[x]$, i.e. attempt to receive a value over k. The first process only sends one value and thus the protocol associated with k can only allow one interaction. The result is that this program cannot complete successfully because the communication is inconsistent, and the type system will reject it. To address this, any implementation of session types should consider session channel as linear types, or at the very least affine types [18]. This prevents aliasing and thus communication inconsistency. As we saw in Section 2.1 Rust's move semantics implement affine types and we will see in Section 5 how move semantics directly address this requirement. Embeddings in other programming languages must explicitly tackle this issue by other restrictions, but in Rust the required language features are already in place.

For the remainder of this dissertation we will primarily describe session types by the type system rather than stating programs written in $\mathcal{L}$.

4.2 Type System

This section introduces the typing system for session type channels in $\mathcal{L}$. Its novel features are the treatment of session channels and the ability to verify the compatibility of process interaction beyond singular message passing. The typing discipline ensures that only type safe interaction patterns are allowed.

The grammar for session-typed channels is shown in Figure 3. Type variables are ranged over by t, t', ..., $\mathcal{S}$ is the set of *sorts* ranged over by S, S', ... and $\mathcal{T}$ is the set of types ranged over by α, β, ... $\tilde{S}$ is a vector of S.

The type $?[\tilde{S}]; \alpha$ represents receiving values of sorts $\tilde{S}$, and continuing with the type α. $![\tilde{S}]; \alpha$ is the converse, or dual, of $?[\tilde{S}]; \alpha$ and represents sending values of sorts $\tilde{S}$ followed by the actions prescribed by the type α. Labelled branching is typed as $\&\{l_1 : \alpha_1, \ldots, l_n : \alpha_n\}$ which waits with n options, labelled l_1 through l_n, and continues with type α_i if the branch l_i is selected. The converse of labelled branching is branch selection and its type is $\oplus\{l_1 : \alpha_1, \ldots, l_n : \alpha_n\}$, which indicates an active choice of a branch l_i is made and the process continues with the type α_i. The type of `inact` is `end` and it terminates the sequential composition. The type $\mu t.\alpha$ is a recursive behaviour which performs α and upon encountering

$$
\begin{aligned}
S &::= \mathsf{nat} \mid \mathsf{bool} \mid \langle \alpha, \overline{\alpha} \rangle \\
\alpha &::= ?[\tilde{S}]; \alpha \mid ?[\alpha]; \beta \mid \&\{l_1 : \alpha_1, \ldots, l_n : \alpha_n\} \mid \mathsf{end} \mid \bot \\
&\quad \mid ![\tilde{S}]; \alpha \mid ![\alpha]; \beta \mid \oplus\{l_1 : \alpha_1, \ldots, l_n : \alpha_n\} \mid t \mid \mu t.\alpha
\end{aligned}
$$

Figure 3: The syntax of types in $\mathcal{L}$.

$$
\begin{aligned}
\overline{![\tilde{S}]; \alpha} &= ?[\tilde{S}]; \overline{\alpha} & \qquad \overline{\oplus\{l_i : \alpha_i\}_{i \in I}} &= \&\{l_i : \overline{\alpha_i}\}_{i \in I} \\
\overline{?[\tilde{S}]; \alpha} &= ![\tilde{S}]; \overline{\alpha} & \qquad \overline{\&\{l_i : \alpha_i\}_{i \in I}} &= \oplus\{l_i : \overline{\alpha_i}\}_{i \in I} \\
\overline{![\alpha]; \beta} &= ?[\alpha]; \overline{\beta} & \qquad \overline{\mathsf{end}} &= \mathsf{end} \\
\overline{?[\alpha]; \beta} &= ![\alpha]; \overline{\beta} & \qquad \overline{\mu t.\alpha} &= \mu t.\overline{\alpha}
\end{aligned}
$$

Figure 4: The co-type (or dual) of a type α is denoted $\overline{\alpha}$.

t, recurses to perform α again. Recursive types are assumed to be *contractive*, meaning that types do not contain a subexpression of the form $\mu t.\mu t_1 \cdots \mu t_n.t$, and the type system takes an *equi-recursive* approach, considering $\mu t.\alpha$ and its expansion $\alpha[\mu t.\alpha/t]$ equivalent. The types $![\alpha]; \beta$ and $?[\alpha]; \beta$ are the types for the session delegation actions, i.e. sending (`throw`) and receiving (`catch`) session channels respectively.

Two types α and β are considered equivalent if we can go from one to the other via recursive expansion and/or reordering branches. For example, the types $\oplus\{l_1 : \alpha_1, l_2 : \alpha_2\}$ and $\oplus\{l_2 : \alpha_2, l_1 : \alpha_1\}$ are equivalent, and $\mu t.[\tilde{e}]; t$ and $[\tilde{e}]; \mu t.[\tilde{e}]; t$ are equivalent. Formally, equivalence is defined in terms of a largest fix point of a monotone function, but we omit the full definition and refer the reader to Definition 2.2 of [20].

We state the type of the arithmetic server as follows:

$$\&\{\mathsf{add} : ?[\mathsf{nat}]; ?[\mathsf{nat}]; ![\mathsf{nat}]; \mathsf{end}, \mathsf{neg} : ?[\mathsf{nat}]; ![\mathsf{nat}]; \mathsf{end}\},$$

which is a straightforward transcription of the program text. Similarly, the recursive arithmetic server has the type:

$$\mu t.\&\{\mathsf{add} : ?[\mathsf{nat}]; ?[\mathsf{nat}]; ![\mathsf{nat}]; t, \mathsf{neg} : ?[\mathsf{nat}]; ![\mathsf{nat}]; t, \mathsf{quit} : \mathsf{end}\}.$$

An important concept in the typing discipline is the concept of a *dual* or *co-type* (we will use the former term). The dual of a type α exchanges session interactions with their logical counterpart. Specifically we exchange ! and ?, and & and $\oplus$. Computing the dual $\overline{\alpha}$ is defined inductively in Figure 4.

Two processes interacting in a session are only compatible if their types are dual. We can thus verify that our example client from earlier is compatible with the ARITH program with the type:

$$\oplus\{\mathsf{add} : ![\mathsf{nat}]; ![\mathsf{nat}]; ?[\mathsf{nat}]; \mathsf{end}, \mathsf{neg} : ![\mathsf{nat}]; ?[\mathsf{nat}]; \mathsf{end}\}$$

The typing rules for $\oplus$ allows it to be compatible with any $\&\{\cdots\}$ construction as long as it contains a branch labelled `add` with the subsequent dual type matching $![\mathsf{nat}]; ![\mathsf{nat}]; ?[\mathsf{nat}]; \mathsf{end}$. This is evident in the typing rule for branch selection:

$$\frac{\Theta; \Gamma \vdash P \rhd \Delta \cdot k : \alpha_j}{\Theta; \Gamma \vdash k \lhd l_j; P \rhd \Delta \cdot k : \oplus\{l_1 : \alpha_1, \ldots, l_n : \alpha_n\}} \, (1 \leq j \leq n)$$

The arithmetic server could contain many more branches and the same client program would still type-check, but its type would be different.

As it may seem strange that the type of the connecting client may depend on branches the client will never use, a notion of sub-typing for session has been developed in the π-calculus [17].

This concludes our introduction to $\mathcal{L}$. For a further introduction to $\mathcal{L}$ and the Honda-Vasconcelos-Kubo session typing system, we refer to [20], which includes proper type equivalence definitions, lists the typing judgements, and proves that the system preserves *type safety*. Informally stated for session types, type safety means that in a well-typed program, a process cannot send a value of a type not expected by the receiving party.

4.3 Session Types in Haskell

This section describes the implementation of session types in Haskell published in [18]. There are other implementations of session types in Haskell [21, 22], but this one is noteworthy because it is relatively straightforward to use (thanks to the `ixdo` preprocessor, Haskell's `do`-notation, and Haskell's powerful type inference), and its formulation of session types readily translate into other polymorphic, typed languages, such as Java [18]. For simplicity, we will focus on the implementation of a single implicit channel: The paper also extends the implementation to work with multiple channels in a single session.

The session types constructs are implemented by the data types shown in Listing 19. Recursion is achieved through the `Rec` and `Var` constructs and Peano numbers constructed by

```
1  data (:!:) a r  ;; send
2  data (:?:) a r  ;; receive
3  data Eps        ;; eps
4  data (:+:) r s  ;; choose
5  data (:&:) r s  ;; offer
6  data Z          ;; zero
7  data S n        ;; successor
8  data Rec r      ;; enter a recursive scope
9  data Var v      ;; recurse into scope ‘v‘
```

Listing 19: Haskell data types for encoding session types.

the use of `Z` and `S`. The branching constructs, `:+:` and `:&:`, only have two branches; unlike traditional session type systems, the implementation does not support labelled branching, but relies on binary branching.

Using these constructs, the type of the recursive arithmetic server from Section 4.2 is written as follows:

```
Rec ((Int :?: Int :?: Int :!: Var Z) :&:
     ((Int :?: Int :!: Var Z) :&: Eps))
```

However, this is not equivalent to the session type given for the recursive arithmetic server given in Section 4.2: The order of the branches in the offer type (`:&:`) matter and "`a :&: b`" and "`b :&: a`" are in general not equal.

Listing 20 shows the definition of a `Session` and a collection of the functions for interacting with a session. A `Session` represents a transition from a state `s` to a new state `s'` yielding a value of type `a`, where both `s` and `s'` are capabilities. A capability, represented by `Cap`, includes a protocol stack, called the environment, and the current protocol. As an example, `send` returns a `Session` that sends a value `x` and then continues with the protocol `r`. Additionally, the functions `close`, `send`, `offer`, `enter`, `zero` and `suc` are defined, as well as `recv`, `sel1`, and `sel2`, which have been omitted for brevity.

```haskell
newtype Session s s' a = Session { unSession :: UChan -> IO a }
data Cap e r

close :: Session (Cap e Eps) () ()
close = Session (\_ -> return ())

send :: a -> Session (Cap e (a :!: r)) (Cap e r) ()
send x = Session (`unsafeWriteUChan` x)

-- `recv` is analogous

offer :: Session (Cap e r) u a ->
         Session (Cap e s) u a ->
         Session (Cap e (r :&: s)) u a
offer (Session m1) (Session m2)
    = Session (\c -> do b <- unsafeReadUChan c
                        if b then m1 c else m2 c)

-- `sel1` and `sel2` send `true` and `false`, respectively

enter :: Session (Cap e (Rec r)) (Cap (r, e) r) ()
enter = Session (\_ -> return ())

zero :: Session (Cap (r, e) (Var Z)) (Cap (r, e) r) ()
zero = Session (\_ -> return ())

suc :: Session (Cap (r, e) (Var (S v))) (Cap e (Var v)) ()
suc = Session (\_ -> return ())
```

Listing 20: Haskell implementations of the `Session` and `Cap` types, as well as `close`, `send`, `offer`, `enter`, `zero` and `suc`.

The `zero`, `suc`, and `enter` functions are of particular interest. Like `close`, they are no-ops, but they perform an important function, which is to facilitiate recursion by manipulating the protocol environment stack. The function `enter` pushes the current protocol onto the protocol environment stack, `suc` pops a protocol from the stack, allowing one to access the underlying ones with `zero`, which enters the protocol currently at the top of the stack.

```haskell
class IxMonad m where
    (>>>=) :: m i j a -> (a -> m j k b) -> m i k b
    (>>>) :: m i j a -> m j k b -> m i k b
    m >>> k = m >>>= const k
    ret :: a -> m i i a

instance IxMonad Session where
    ret a = Session (\_ -> return a)
    m >>>=  k = Session (\c -> do a <- unSession m c
                                  unSession (k a) c)
```

Listing 21: The definition of the Indexed Monad.

Session computations can be composed, but instead of a regular monad, `Session` is an instance of an indexed monad. Indexed monads help capture the restriction that sessions should only be composable when the end state of the first session is equal to the start state of the second session, and they can be used to enforce the affine use of channels. The definition and implementation is shown in Listing 21. Additionally, an `ixdo` notation is implemented

```haskell
class Dual r s | r -> s, s -> r

instance Dual r s => Dual (a :!: r) (a :?: s)
-- And so on
```

Listing 22: Haskell definition of `Dual` plus its instance implementation for `:!:`.

```haskell
newtype Rendezvous r = Rendezvous (TChan UChan)

newRendezvous :: IO (Rendezvous r)
newRendezvous = liftM Rendezvous newChan

accept :: Rendezvous r ->
          Session (Cap () r) () a -> IO a
accept (Rendezvous c) (Session f) = do
  nc <- newUChan
  writeChan c nc
  f nc

request :: Dual r r' => Rendezvous r ->
           Session (Cap () r') () a -> IO a
request (Rendezvous c) (Session f)
        = readChan c >>= f
```

Listing 23: Haskell implementations of `Rendezvous`, `newRendezvous`, `accept`, and `request`.

by a preprocessor, which works like the standard `do`, but for indexed monads.

Ensuring protocol compatibility is handled by the `Dual` type class shown in Listing 22, where we are also showing the implementation of `Dual` for `:!:`. The remaining instances are analogous. Connecting end-points is handled by the `accept` and `request` functions, shown in Listing 23. A `Rendezvous` is used to initialize the session between two functions using `accept` and `request`, and as we can see, duality is enforced by the constraint in `request` that the two protocols `r` and `r'` must be dual.

We show an implementation of a server satisfying the type for the recursive arithmetic server defined above:

```haskell
server = enter >>> loop where
    loop = offer (ixdo
                  n <- recv
                  m <- recv
                  send (n + m)
                  zero
                  loop)
                 (offer (ixdo
                         n <- recv
                         send (-n)
                         zero
                         loop)
                        close)
```

The example demonstrates how the use of the `ixdo` notation allows the code to be succinct, and Haskell's type inference is powerful enough that we do not have to write the types explicitly. Note that handling a chain of branches quickly becomes cumbersome, because there is little to alleviate the right-ward drift.

In addition to introducing and demonstrating their session types library, [18] also formalizes their system and prove that the type system prevents threads from receiving unexpected

```
protocol placeOrder {
  begin.
  ![
    !<String>.
    ?(Double)
   ]*.
  !{
    ACCEPT: !<Address>.?(Date),
    REJECT:
  }
}
```

Listing 24: Protocol declaration in SessionJ.

types, despite the reliance on unsafe, untyped channels.

This concludes our brief overview of the implementation of session types in Haskell described in [18].

4.4 Session Types in Object-Oriented Programming

We also looked at implementations of session types in object-oriented programming languages, and this section describes one such effort, SessionJ, that brings session types to Java. Aside from the core session-typed channels, SessionJ features asynchronous message passing, session delegation, session subtyping and failure handling [19].

The main focus of SessionJ is to bring session types to distributed computing, and to achieve this, the underlying communication primitives are based on sockets and the system features both static and dynamic session type checking. The authors demonstrate that the run-time overhead is low.

Session types, called protocols, are declared in a `protocol` block in which a special session types DSL is accepted. The SessionJ compiler then translates the protocol to corresponding Java constructs and performs a dedicated type-checking phase where it checks that channel values are used linearly. The SessionJ frontend produces a valid Java program that can be compiled with the normal Java compiler. An example protocol declaration is shown in Listing 24. The send and receive operations are denoted by `!<type>` and `?(type)` respectively. Multi-way labelled branching is also supported by the `!{..}` and `?{..}` constructs and the branch labels are translated to run-time values that can be used to select a branch. The branching constructs can also be thought of as a distributed switch-statement.

The notable difference between the session type theory, the Haskell version and SessionJ are the constructs for iteration. In both the session type theory and the Haskell implementation, iteration is implemented by recursion. In SessionJ iteration is implemented by the mutually dual constructs `![..]*` and `?[..]*`, where `[..]*` expresses that the contained interactions may be iterated zero or more times. The construct `![..]*` controls the iteration and `?[..]*` follows the decision, and they can together be thought of as a distributed while-loop.

Channel end-points, called *session sockets*, are run-time entities over which all session-typed communication takes place. Session sockets are instantiated with a pre-defined protocol, and all interactions are implemented as methods on a session socket, i.e. the method `send` and `recv` implement `!` and `?` respectively, `outbranch` implements branch selection, `inbranch` implements label branching, and `outwhile` and `inwhile` implement active and passive iteration.

Before a session can be initiated, the two communicating parties exchange protocols and verify that their protocols are compatible. Only then is the session initiated and session sockets obtained. Because protocol compatibility is checked at run-time, protocol mismatches can occur, which produces an error. To handle a session failure situation, SessionJ provides an extended try-catch mechanism.

SessionJ is an advanced implementation of session types in terms of features and robustness. Its main drawback is that it is a language extension requiring its own special compiler, but this design choice also enables features that would otherwise not be possible in regular Java. The syntax extensions mesh well with the syntax of Java and handling session-typed communication is not a great burden for the programmer.

5 Session Types in Rust

This chapter describes our implementation of a session type library in Rust, that we will to use to solve the communication problems in Servo described in Section 3.2. Specifically, we want to be able to translate the current communication schemes in Servo, which rely on enum types and single-typed channels, to a scheme which relies on session-typed channels for inter-process communication. The library should be able to decide at compile-time, whether or not a given program adheres to the protocols specified, and if that is the case, the program should not fail due to communication

errors. Section 5.1 describes the overall design of `session-types`, Section 5.2 describes the implementation in detail. Section 5.3 argues that our implementation is safe. Section 5.4 gives examples and motivates useful extensions to the `session-types` library. Section 5.5 outlines an alternative implementation of session types that we worked on initially. Finally, Section 5.6 discusses alternative approaches to the design of `session-types`, while Section 5.7 gives an short evaluation of our library.

5.1 Design

We have modelled our library after the Haskell implementation in [18]. This design has the benefit that it can be expressed entirely in Rust itself, without any needs for compiler plug-ins, preprocessors or other language extensions. It simply requires a parametric polymorphic, typed language, which Rust happens to be. Also in terms of constructs exposed to the user, we decided to follow the Haskell implementation closely; we provide the `Send` (!), `Recv` (?), `Offer` (&), and `Choose` ($\oplus$) constructs. Unlike the original formulation of session types, we are not using labels but binary choice to branch between different paths in our protocol. The two methods are equally powerful (labelled branches allow the user to choose between arbitrary number of branches, but we can emulate an arbitrary number of choices by chaining binary branches), but they are not equivalent (we can freely reorder labels in a &; for our constructs, the order of branches matters). Although labelled branching seems inherently more descriptive and user-friendly, we have chosen to use binary branches to keep the implementation simple.[6] Additionally, to provide recursion, we rely on the `Rec` and `Var` type constructs, which together with Peano numbers (`S` and `Z`) allows one to express recursion.

For example:

$$\mu t.\&\{\texttt{fst} : ![\texttt{nat}]; t,\\ \texttt{snd} : ?[\texttt{nat}]; \texttt{end}\}$$

becomes

```
Rec<Offer<Send<u8, Var<Z>>,
          Recv<u8, Eps>>>
```

Note that, like recursion in traditional session types, the recursive construct `Rec` does not have any notion of a "next" action [20]. We considered an alternative construct for iteration that included an explicit "next action" inspired by the iteration constructs, `![..]*` and `?[..]*`, from SessionJ [19]. This approach is discussed in detail in Section 5.6.

5.2 Implementation

This section describes the implementation of the Rust library `session-types`. We will first describe how the different session types are encoded, then how untyped channels are implemented, and finally how the functions for using the channels are implemented, using the untyped channels in a safe manner.

[6] See Section 5.4.2 for a description of an attempt to use macros to ease the use of binary branches.

```rust
/// End of communication session (epsilon)
#[allow(missing_copy_implementations)]
pub struct Eps;

/// Receive `A`, then `P`
pub struct Recv<A, P> ( PhantomData<(A, P)> );

/// Send `A`, then `P`
pub struct Send<A, P> ( PhantomData<(A, P)> );

/// Active choice between `P` and `Q`
pub struct Choose<P, Q> ( PhantomData<(P, Q)> );

/// Passive choice (offer) between `P` and `Q`
pub struct Offer<P, Q> ( PhantomData<(P, Q)> );

/// Enter a recursive environment
pub struct Rec<P> ( PhantomData<P> );

/// Recurse. `N` indicates how many layers of the recursive environment we
/// recurse out of (encoded as Peano numbers)
pub struct Var<N> ( PhantomData<N> );

/// Peano numbers: Zero
#[allow(missing_copy_implementations)]
pub struct Z;

/// Peano numbers: Succ
pub struct S<N> ( PhantomData<N> );
```

Listing 25: Rust structs used to represent session types.

The code in Listing 25 shows the encoding of session types in Rust's type system. `Eps` is the empty protocol, and indicates that communication between the two processes has run to completion and the channel can be closed. The structs `Send<A, P>` and `Recv<A, P>` denote sending and receiving messages of type `A`, before continuing with the protocol `P`. `Choose<P, Q>` and `Offer<P, Q>` represent active and passive choice respectively, between protocols `P` and `Q`. `Rec` is used to enter a recursive environment, pushing the current protocol onto a protocol stack, and the structs `Z` and `S<N>` are used as arguments to `Var<V>`, indicating how many protocols to pop from the stack before recursing. For example, `Var<S<S<Z>>>` indicates that we are to pop two protocols from the stack before recursing into the third one. We shall see how this works in detail later. The `PhantomData` field in the structs with generic type parameters is required to mark phantom types. Rust does not permit unused type parameters and `PhantomData` is provided to explicitly mark phantom types.

We encode duality in terms of a trait with an associated type. Listing 26 shows the declaration of the trait `HasDual` and how it is implemented for the various session type constructs. The `HasDual` trait has a single entry: an assocaited type called `Dual` which represents the dual type for the implementing type. For example, the `Dual` type of `Send<A, P>` is `Recv<A, P::Dual>`, where `P` must also implement `HasDual`. The use of associated types allows us to elegantly express our requirements for duality. Additionally, the trait and the implementations are marked as `unsafe` to indicate that users should not implement `HasDual` themselves. Doing so potentially violates the session type guarantees that form the basis of our library.

An earlier implementation, which we used for a few months, used tupled type parameters instead of associated types (see Listing 27). This was directly inspired the Haskell imple-

```rust
pub unsafe trait HasDual {
    type Dual;
}

unsafe impl HasDual for Eps {
    type Dual = Eps;
}

unsafe impl <A, P: HasDual> HasDual for Send<A, P> {
    type Dual = Recv<A, P::Dual>;
}

unsafe impl <A, P: HasDual> HasDual for Recv<A, P> {
    type Dual = Send<A, P::Dual>;
}

unsafe impl <P: HasDual, Q: HasDual> HasDual for Choose<P, Q> {
    type Dual = Offer<P::Dual, Q::Dual>;
}

unsafe impl <P: HasDual, Q: HasDual> HasDual for Offer<P, Q> {
    type Dual = Choose<P::Dual, Q::Dual>;
}

unsafe impl HasDual for Var<Z> {
    type Dual = Var<Z>;
}

unsafe impl <N> HasDual for Var<S<N>> {
    type Dual = Var<S<N>>;
}

unsafe impl <P: HasDual> HasDual for Rec<P> {
    type Dual = Rec<P::Dual>;
}
```

Listing 26: The `HasDual` trait and impls.

mentation discussed earlier, but changes to the coherence inference scheme in Rust made it impossible to infer the appropriate types. Our current encoding using associated types much clearer conveys the meaning of duality: For a given type, there exists a uniquely defined dual type that can be associated with this type.

To demonstrate how the `HasDual` trait is used we implement corresponding versions of the session initiation functions `accept` and `request` from [23]. `accept` can be viewed as the server-side function, whose responsibility is to initiate a session between the caller and a process requesting a connection. The code for `accept` is shown in Listing 28. `accept` produces two end-points for a session-typed channel, passes one of the end-points over a provided channel and returns the other. The types are specified as follows: `accept` requires a generic argument `P` implementing the trait `HasDual`. The type of the channel passed over the channel is then `Chan<(), P::Dual>` and the returned channel has type `Chan<(), P>`. Thus duality is enforced at session initiation. The `request` function only receives a session-typed channel on a provided `Receiver` and does not have to worry about duality.

Interacting with a channel and taking steps in the protocol is implemented as methods matching on the types defined Listing 25. Listing 29 shows the implementations of methods `close`, `send`, and `recv`. Each method is specialized, such that it can only be called on channels

```rust
/// Indicates that two protocols are dual
pub unsafe trait Dual: PhantomFn<Self> {}

unsafe impl <A, P, Q> Dual for (Send<A, P>, Recv<A, Q>)
    where (P, Q): Dual {}
```

Listing 27: Our previous approach for handling duality using tupled type parameters.

```rust
pub fn accept<P: HasDual>(tx: Sender<Chan<(), P::Dual>>) -> Option<Chan<(), P>> {
    let (tx1, rx1) = channel();
    let (tx2, rx2) = channel();

    let c1 = Chan(tx1, rx2, PhantomData);
    let c2 = Chan(tx2, rx1, PhantomData);

    match tx.send(c1) {
        Ok(_) => Some(c2),
        _ => None
    }
}
```

Listing 28: The `accept` function to initiate a session.

with the appropriate protocol. For example, `close` can only be called on a channel where communication has run to completion, i.e. its protocol is `Eps`, regardless of what the environment is. Similarly, we can only call `send` on a channel where the next step in the protocol is `Send<A, P>`, returning a channel with a session of type `P`.

On calling `recv` on a channel of type `Chan<E, Recv<A, P>>`, we read a value of type `A` using the `unsafe_read_chan` function, and then transmute the channel into a `Chan<E, P>`, indicating that we have received a value of type `A` and are now ready to proceed with the protocol indicated by `P`. Using `transmute` is safe only because we have actually read a value from the channel.

Overall the operations available on channels in different stages of a protocol may be described by the following "steps":

```
Chan<E, Send<A, P>> ⟶ Chan<E, P>                          send()
Chan<E, Recv<A, P>> ⟶ (Chan<E, P>, A)                     recv()
Chan<E, Choose<P, Q>> ⟶ Chan<E, P>                        sel1()
Chan<E, Choose<P, Q>> ⟶ Chan<E, Q>                        sel2()
Chan<E, Offer<P, Q>> ⟶ Result<Chan<E, P>, Chan<E, Q>>     offer()
Chan<E, Rec<P>> ⟶ Chan<(P, E), P>                         enter()
Chan<(P, E), Var<Z>> ⟶ Chan<(P, E), P>                    zero()
Chan<(P, E), Var<S<N>>> ⟶ Chan<E, Var<N>>                 succ()
Chan<E, Eps> ⟶ ()                                         close()
```

For example, the method `offer` is *only* available when the next step in the protocol is an `Offer` type. Furthermore, it is the only method available on a channel of that type, and it can only transform the channel into either a `Chan<E, P>` or `Chan<E, Q>` depending on the choice communicated by the other party.

`recv` could also have been written using only safe code, as shown in Listing 30. We note that our use of `transmute` is safe, because we are converting phantom types with no runtime

```rust
impl<E> Chan<E, Eps> {
    /// Close a channel. Should always be used at the end of your program.
    pub fn close(self) {
        // Consume `c`
    }
}

impl<E, P, A: marker::Send + 'static> Chan<E, Send<A, P>> {
    /// Send a value of type `A` over the channel. Returns a channel with
    /// protocol `P`
    pub fn send(self, v: A) -> Chan<E, P> {
        unsafe_write_chan(&self, v);
        unsafe { transmute(self) }
    }
}

impl<E, P, A: marker::Send + 'static> Chan<E, Recv<A, P>> {
    /// Receives a value of type `A` from the channel. Returns a tuple
    /// containing the resulting channel and the received value.
    pub fn recv(self) -> (Chan<E, P>, A) {
        let v = unsafe_read_chan(&self);
        (unsafe { transmute(self) }, v)
    }
}
```

Listing 29: Rust implementations of `close`, `send`, and `recv`.

```rust
impl<E, P, A: marker::Send + 'static> Chan<E, Recv<A, P>> {
    /// Receives a value of type `A` from the channel. Returns a tuple
    /// containing the resulting channel and the received value.
    pub fn recv(self) -> (Chan<E, P>, A) {
        let v = unsafe_read_chan(&self);
        let Chan(tx, rx, _) = self;
        (Chan(tx, rx, PhantomData), v)
    }
}
```

Listing 30: An alternative Rust implementation of `recv`.

representation, and not re-interpreting an integer as an exotic data structure. Transmuting the channel type also conveys the idea that we are operating on the same channel.

```rust
/// A session-typed channel. `P` is the protocol and `E` is the environment,
/// containing potential recursion targets
pub struct Chan<E, P> (Sender<Box<u8>>, Receiver<Box<u8>>, PhantomData<(E, P)>);
```

Listing 31: The Rust implementation of a session channel.

Listing 31 shows the definition of `Chan`, a session-typed channel. In short, a channel is a type that takes two type parameters as arguments: an environment `E` and a protocol `P`, both marked as phantom types. The channel contains both a `Sender` and a `Receiver`, which facilitate the communication between the two processes.

Listing 32 shows the code for `unsafe_read_chan` and `unsafe_write_chan`, which are used by `recv` and `send`. It might seem strange at first that we have to box the value we are sending, but it is required to make sure that all the values we send have the same size. If a `Chan` contained a `Sender<u8>`, we could only transmute it to a `Sender` of the same size, e.g. a `Sender<i8>`, but

```rust
fn unsafe_write_chan<A: marker::Send + 'static, E, P>
    (&Chan(ref tx, _, _): &Chan<E, P>, x: A)
{
    let tx: &Sender<Box<A>> = unsafe { transmute(tx) };
    tx.send(Box::new(x)).unwrap();
}

fn unsafe_read_chan<A: marker::Send + 'static, E, P>
    (&Chan(_, ref rx, _): &Chan<E, P>) -> A
{
    let rx: &Receiver<Box<A>> = unsafe { transmute(rx) };
    *rx.recv().unwrap()
}
```

Listing 32: Implementations of `unsafe_write_chan` and `unsafe_read_chan`.

not a `Sender<u64>`. A `Box` is an owned unique pointer to a piece of memory on the heap, and boxes always have the same size, even though the piece of memory it points to might be of varying sizes. To send a value, we therefore box it first (copy it to the heap), and, consequently, to receive a value, we must then unbox it (copy it from the heap). The choice of `u8` in `Sender<Box<u8>>` is an arbitrary one and does not have any effect on the program.

One will also note that we are using the `unwrap` method on the result of sending and receiving. As a result, `unsafe_write_chan` will panic if the receiver has been disconnected. However, the whole point of our session type library is to guarantee that two processes have compatible communication patterns, and so we assume that this will never happen. We will address this in greater detail in Section 5.3.

```rust
impl<E, P, Q> Chan<E, Offer<P, Q>> {
    /// Passive choice. This allows the other end of the channel to select one
    /// of two options for continuing the protocol: either `P` or `Q`.
    pub fn offer(self) -> Result<Chan<E, P>, Chan<E, Q>> {
        let b = unsafe_read_chan(&self);
        if b {
            Ok(unsafe { transmute(self) })
        } else {
            Err(unsafe { transmute(self) })
        }
    }
}

impl<E, P, Q> Chan<E, Choose<P, Q>> {
    /// Perform an active choice, selecting protocol `P`.
    pub fn sel1(self) -> Chan<E, P> {
        unsafe_write_chan(&self, true);
        unsafe { transmute(self) }
    }

    /// Perform an active choice, selecting protocol `Q`.
    pub fn sel2(self) -> Chan<E, Q> {
        unsafe_write_chan(&self, false);
        unsafe { transmute(self) }
    }
}
```

Listing 33: Rust implementations of `offer`, `sel1`, and `sel2`.

Listing 33 shows the code for `offer`, `sel1`, and `sel2`. `sel1` and `sel2` perform the active choice and communicate their choice by sending a boolean, indicating if protocol `P` or `Q` is to be used. `offer` then receives that boolean; returning the channel in a `Ok` or `Err` wrapper. Note that `Ok` and `Err` do not carry their usual Rust meaning here, i.e. returning an `Err` does not imply that an error has occured. Instead we can think of `Result` as binary choice, similar to the `Either` type in Haskell (`Left` and `Right`). Indeed, we could have implemented our own `Either` type for this, but it is unnecessary and there is a lot of infrastructure in place for `Result` already that users might want to take advantage of.

At this point it should be noted that the session types in `session-types` are not equivalent to traditional session types, even though we can express programs that have the same semantic meaning as any program written in $\mathcal{L}$. For example, consider the type of the arithmetic server from Section 4.2:

$$\&\{\texttt{add} : ?[\texttt{nat}]; ?[\texttt{nat}]; ![\texttt{nat}]; \texttt{end}, \texttt{neg} : ?[\texttt{nat}]; ![\texttt{nat}]; \texttt{end}\}$$

which we can express in Rust as:

```
Offer<Recv<usize, Recv<usize, Send<usize, Eps>>>,
      Recv<usize, Send<usize, Eps>>>
```

but the following translation is equally valid:

```
Offer<Recv<usize, Send<usize, Eps>>,
      Recv<usize, Recv<usize, Send<usize, Eps>>>>
```

However, the two types are not equivalent: They are not equal in Rust's type system and the server implementing the two different types would look differently, just like a client talking to the first server would not be able to talk to the second one. In $\mathcal{L}$, by contrast, changing the order of the arguments to a & or a $\oplus$ does not change the set of valid implementations, as the types would still be equivalent. This does not mean that the session types in `session-types` are less expressive, just that order matters, and the user cannot shuffle arguments to `Choose` and `Offer` around as they please, without changing their implementations.

Similarly, the Rust type system does not allow us to encode the equi-recursive view of session types that $\mathcal{L}$ has. The result is that a type such as

```
Rec<Send<T, Var<Z>>>
```

is not considered equivalent to

```
Send<T, Rec<Send<T, Var<Z>>>>
```

for some sendable type `T`, but the second type corresponds exactly to a single expansion of the `Rec`. Again, we do not lose expressiveness, we just lose a bit of flexibility.

Listing 34 shows the code for the methods `enter`, `zero`, and `succ`, which are used to facilitate recursion. As we can see, the implementations themselves are not particularly interesting, but the type machinery is. Inspired by the Haskell implementation described in 4.3 we use de Bruijn indices to refer to the different recursion scopes. Entering a recursive scope, by using `enter`, pushes the current protocol to the environment stack so that we can later retrieve it again using `succ` which pops scopes from the stack and `zero` which enters the recursion scope at the top of the environment stack.

Finally, to create session channels, we provide the function `session_channel`, as shown in Listing 35. The interesting part here is the types of the returned channels. The type parameter P is a session type, and we require that it implements `HasDual` to enforce compatibility at session initialization.

This concludes our description of our Rust implementation of session types. Listing 36 shows a small example client-server program, where the client sends a `u64` and then exits. The server receives the number, closes its channel and prints the received number.

```rust
impl<E, P> Chan<E, Rec<P>> {
    /// Enter a recursive environment, putting the current protocol on the
    /// top of the environment stack.
    pub fn enter(self) -> Chan<(P, E), P> {
        unsafe { transmute(self) }
    }
}

impl<E, P> Chan<(P, E), Var<Z>> {
    /// Recurse to the environment on the top of the environment stack.
    pub fn zero(self) -> Chan<(P, E), P> {
        unsafe { transmute(self) }
    }
}

impl<E, P, N> Chan<(P, E), Var<S<N>>> {
    /// Pop the top environment from the environment stack.
    pub fn succ(self) -> Chan<E, Var<N>> {
        unsafe { transmute(self) }
    }
}
```

Listing 34: Rust implementations of `enter`, `zero`, and `succ`.

```rust
/// Returns two session channels
pub fn session_channel<P: HasDual>() -> (Chan<(), P>, Chan<(), P::Dual>) {
    let (tx1, rx1) = channel();
    let (tx2, rx2) = channel();

    let c1 = Chan(tx1, rx2, PhantomData);
    let c2 = Chan(tx2, rx1, PhantomData);

    (c1, c2)
}
```

Listing 35: Rust implementations of `session_channel`.

5.3 Safety

In this section we will argue why and to what extent `session-types` guarantees safe communication patterns.

What do we mean when we talk about safe communication patterns? Of course, we cannot in general guarantee that a program will never fail, or that it will halt at some point. What we are trying to achieve is a guarantee that two processes are *compatible* "*in the sense that an interaction between the two will not terminate prematurely because of a mismatch in the expectations of one of the partners*" [24]. For example, if one side expects to receive a string, the other side will never send a value of any other type, nor will it expect any other action to take place, like recursion or choice. It may be that the other side fails to ever take action because of exceptional behavior (panics or infinite loops, for example), but if an action is ever taken (and the user is only using safe code), we guarantee that it is the expected action.

We will not give a formal proof for our claims, partly because Rust has no formal specification, but we will argue for our claims in terms of Rust's informal semantics. We will focus on our library-provided methods and functions, treating `session-types` as an embedded DSL for dealing with sessions. Later we will discuss how certain Rust functions, like `drop`, affects our claims.

```rust
fn client(n: u64, c: Chan<(), Send<u64, Eps>>) {
    c.send(n).close()
}

fn server(c: Chan<(), Recv<u64, Eps>>) {
    let (c, n) = c.recv();
    c.close();
    println!("Received {}", n);
}

fn main() {
    let n = 42;
    let (c1, c2) = session_channel();

    let s1 = scoped(move || client(n, c1));
    let s2 = scoped(move || server(c2));

    s1.join();
    s2.join();
}
```

Listing 36: Demonstrating `send`, `recv` and `session_channel`.

First of all, we wish to argue that the only way the user can send any data through a session channel, is by using either `send`, `sel1`, or `sel2`. The underlying `Sender` is encapsulated in the `Chan`, and is not publicly accessible, meaning that it can only be accessed from within `session-types` itself. It is crucial for our safety claims that our API does not expose references to the inner fields of a `Chan` to the user. From the function definitions above, we can see that the only place where the `Sender` is accessed is in `unsafe_write_chan`, which is not publicly accessible either. That function, in turn, is only called from the public functions `send`, `sel1`, and `sel2`. Hence, we can see that all communications sent through a session channel must go through either `send`, `sel1`, or `sel2`.

Furthermore, it is clear that, given a channel c of type `Chan<E, Send<A, P>>`, `send` will only ever send a value of type `A` before returning a channel of type `Chan<E, P>`. Similarly, given a channel c of type `Chan<E, Choose<P, Q>>`, `sel1` and `sel2` will only ever send a value of type `bool`, whereafter they will return channels of type `Chan<E, P>` and `Chan<E, Q>`, respectively.

From this, we can gather than it is only possible to send values through a channel via `send`, `sel1`, and `sel2`, and they are well-behaved, in the sense that they send the types that we expect them to, and they progress the session type of the session. We can use an analogous argument to show that the only way to receieve values through a channel is via `recv` and `offer`, that they are well-behaved, and that they progress the session type.

By inspecting the rest of the functions given above, we can also see that `send` is the only function that takes a `Chan<E, Send<A, P>>`, for some E, A, and P as an argument, and the only way to use the channel is therefore to progress it by calling `send`. Analogously, `Chan<E, Recv<A, P>>` can only be used by `recv`, `Chan<E, Offer<P, Q>>` by `offer`, and `Chan<E, Choose<P, Q>>` by either `sel1` or `sel2`.

Now, a session always consists of exactly two endpoints that we call channels; the only way we can get a session-typed channel is through the function `session_channel`, which returns two channels. Additionally, through the use of the `HasDual` trait, `session_channel` ensures that the types the channels are dual. Because the dual of a session type is uniquely defined, there is only one possible dual session type s' for any given session type s, and the `HasDual` trait reflects this. Thus—assuming that channels are synchronous—whenever session s is in the state `Chan<E, Send<A, P>>`, session s' will be in the state `Chan<E, Recv<A, P::Dual>>` and

so on. The channels we use in `session-types` are not synchronous, but as [24] has shown, this is not a problem, session s' will merely arrive at the state `Chan<E, Recv<A, P::Dual>>` at a later time.

We also have take aliasing into account, as discussed in Section 4. To address this, we can leverage Rust's move semantics by ensuring that `Chan` does not implement neither `Clone` nor `Copy`. For example, consider the following piece of code:

```
let (c1, c2) = session_channel::<Send<u8, Eps>>();
let c3 = c1;
c1.send(42u8).close();
c3.send(43u8).close();
let (c2, n) = c2.recv();
c2.close();
```

If `Chan` values could be cloned, this would be a valid program, that violates our session type guarantees. `c1` has the type `Chan<(), Send<u8, Eps>>`, so it should only be possible to perform one send operation on it. However, the underlying `Sender` would be sending two values, violating the protocol. Because of move semantics the assignment `let c3 = c1` moves the value `c1` and prevents further use of it.

We have now stated three things: We can only send and receive values through a channel using the provided methods, `send`, `recv`, `offer`, `sel1`, and `sel2`; when sending or receiving anything through a channel, it will have the type indicated by the type parameters to the channel, i.e. sending on a channel with protocol `Send<A, P>` for some `A` and `P`, the value we are sending will have type `A`; and any time we have a session channel endpoint s with type `P`, there will be exactly one corresponding dual endpoint s' with type `P::Dual`. This means that if a session channel is used, it will be used correctly, and the program will be well-behaved. We say that our session types are *type safe* (or *sound*).

However, we cannot claim that our session-typed channels will never fail, as a program may drop a channel at any time. It is therefore up to the user to ensure that all session-typed channels are closed properly with `close`.

5.4 Examples and Extensions

To illustrate how to use `session-types`, and to exhibit some use cases where we found a need for some additional tooling, we now show and discuss some examples that use `session-types`, as well as some of the extra mechanics that we have introduced. Section 5.4.1 shows how we have implemented a selection mechanism for choosing among multiple receiving channels. Section 5.4.2 describes an attempt to solve the problem of handling many `Offer` branches. Section 5.4.3 demonstrates how we can safely handle an unknown number of clients with session types. Finally, Section 5.4.4 shows how to solve an interesting concurrency problem, called the Santa Claus problem, using session types.

In addition to the demonstrations and showcases we will discuss below, we have also implemented a variety of other examples using session types which we wont cover in depth. The examples come from a variety of other papers, and we have implemented our own versions using `session-types`. These examples include: the reentrant polygon clipping algorithm from [25], which has also been implemented in [18]; the ATM example from [23], both in the normal version and in a version that uses delegation to handle the contact between the user and the bank; the POP3 server from [2]; a variety of echo servers and examples that demonstrate the basic functionality of our library; and the ticket ordering system from [19].

5.4.1 Selecting Over Multiple Channels

It is a common use-case for a process to listen on multiple channels. Traditional CSP provides a choice operator allowing a process to act upon a selection of communications [26], and

is implemented in a number of programming languages, notably in occam [27] as the `ALT` construct, in Alef [28] as the `alt` statement and in Go [29] as the `select` statement. The Rust standard library also provides a `Select` structure for listening on multiple channels. Contrary to other programming languages, in Rust the provided concurrency mechanisms are implemented in the standard library rather than provided as a language feature.

In our implementation of the Santa Claus problem (see Section 5.4.4) we discovered a need for a selection construct, which led to the design of a structure a structure similar to `Select` called `ChanSelect`.

We cannot directly use the `Select` structure for the reason that we cannot safely expose the innards of our `Chan` structure (as mentioned earlier). Instead we provide an API similar to that of `Select`, but with minor differences to ensure safety.

The declaration of `ChanSelect` is shown in Listing 37. Channels are added to the structure

```
struct ChanSelect<'a, T> {
    chans: Vec<(&'a, Chan<(), ()>, T>)
}
```

Listing 37: The `ChanSelect` structure.

and borrowed by the structure until its destruction. This is to prevent channels from being consumed while they are referenced in the structure. The regular `Select` does not implement such a limitation. The type parameter `T` indicates a value to return for the selected channel. This becomes useful later when we put session-typed channels to use in Servo (see Section 6).

Channels are added to a `ChanSelect` structure by means of an `add` method, which is shown in Listing 38. We restrict ourselves to accept only channels whose next action in their protocol is `Recv`. Internally, the protocols of the added channels are ignored. This is necessary to be able to add channels with different subsequent protocols to the same `ChanSelect`. The `transmute` function is used to coerce the protocol phantom type to `()`. We argue that discarding the protocol internally is safe, because the structure can only accept channels whose next action is `Recv` (by the declaration of `add`).

```
impl<'a, T> ChanSelect<'a, T> {
    fn add<E, P, A: marker::Send>(&mut self,
                                  chan: &'a Chan<E, Recv<A, P>>,
                                  ret: T)
    {
        self.chans.push((unsafe { transmute(chan) }, ret));
    }
}
```

Listing 38: Adding a channel to the selection structure.

The borrowed references to channels are stored and their lifetimes are tied to the lifetime of the `ChanSelect`. The `ChanSelect` must prevent the owner of a channel from moving the value while it is added, and binding the lifetime of the borrows to the lifetime of the structure achieves exactly that.

To find the next channel ready to receive, the `wait` method is invoked. The declaration of `wait` is:

```
fn wait(self) -> T
```

Calling this method consumes the `ChanSelect` structure and returns the value of type `T` associated with the selected channel. Once a channel is selected, we want to end all borrowed references to it to allow it to be consumed. Because the lifetime of the borrow is tied to

the lifetime of the `ChanSelect` structure, we must drop the whole structure (thus ending all borrowed references). It is for this reason that `wait` takes a owned `self` parameter. Arguably a mutably borrowed reference (`&mut self`) suffices for `wait`, but declaring it to be consuming better conveys the intent of the method. When calling `wait`, internally a `Select` structure is constructed with all the `Receivers` contained in the borrowed channels. Each `Receiver` added to `Select` receives an ID, which is then mapped to the channel's associated return value.

The Rust standard library also provides a convenient `select!` macro that allows for a more ergonomic use of `Select` without explicitly having to construct a `Select` and add `Receivers`. We provide a similar macro called `chan_select!` that internally constructs a `ChanSelect` adding all the referenced channels:

```
chan_select! {
    (c, n) = c1.recv() {
        println!("Received integer: {}", n);
        c.close();
    }
    (c, s) = c2.recv() {
        println!("Received string: {}", s);
        c.close();
    }
}
```

where `c1` and `c2` are `Chan` types with the protocols `Recv<u8, Eps>` and `Recv<String, Eps>` respectively. Inside the macro the referenced channels must be rebound to new names, because the macro expands to a call to `recv` on the selected channel, thereby moving the value.

As a final note on `ChanSelect` we expanded it to also support channels with protocols whose next action are `Offer`. Internally the first action of the `offer` method is to receive a boolean indicating the branch to be chosen, so the interface naturally extends to include channels that can wait to receive the selected branch.

Introducing a structure such as `ChanSelect` has the potential to introduce memory safety "holes", i.e. ways to use the API to cause memory errors. Working on the design, we focused heavily on ensuring the safety of the API and we have argued informally that it is.

5.4.2 Selecting Among Multiple Branches

During our design process, we implemented a version of the arithmetic server example from introduced in Section 4. The server initially offered two branches, one for an addition operation and one for a negation operation. We extended the arithmetic server with two new operations: `sqrt` and `eval`. The former receives a floating point number and sends back its square root, and the latter receives a function and an argument for the function and returns the result of evaluating the function with the argument. The session type for the extended arithmetic server looks as follows:

```
type Srv =
    Offer<Eps,                                                    // close
    Offer<Recv<i64, Recv<i64, Send<i64, Var<Z>>>>,                // add
    Offer<Recv<i64, Send<i64, Var<Z>>>,                           // neg
    Offer<Recv<f64, Choose<Send<f64, Var<Z>>, Var<Z>>>,           // sqrt
          Recv<fn(i64) -> bool, Recv<i64, Send<bool, Var<Z>>>>>>>>; // eval
```

If we use `match` statements to handle the `Srv` protocol the result can look as demonstrated in Listing 39. To address the right-wards drift and unclutter the code, we provide an `offer!` macro that can handle a list of `Offers`. As a convention we only handle chains where the next `Offer` is in the second branch.

Besides handling right-wards drift, the syntax in `offer!` also allows the programmer to label the different branches. The labels have no semantic meaning, but can be used to provide

```rust
// c: Chan<(Srv, ()), Srv>
match c.offer() {
    Ok(c)  => {
        // close
    }
    Err(c) => match c.offer() {
        Ok(c)  => {
            // add
        }
        Err(c) => match c.offer() {
            Ok(c)  => {
                // neg
            }
            Err(c) => match c.offer() {
                Ok(c)  => {
                    // sqrt
                }
                Err(c) => {
                    // eval
                }
            }
        }
    }
}
```

Listing 39: Handling the `Srv` protocol.

```rust
// c: Chan<(Srv, ()), Srv>
offer! { c,
    Close => {
        // close
    },
    Add => {
        // add
    },
    Negate => {
        // neg
    },
    Sqrt => {
        // sqrt
    },
    Eval => {
        // eval
    }
}
```

Listing 40: Using the `offer!` macro.

the reader with a name for the branch. The expansion of the macro is tightly bound to the structure of the session type, so the branches cannot be swapped around freely, they must be handled in the order they appear in the type. Listing 40 shows how the arithmetic server can use the `offer!` macro give an overview of its branches.

We have found the `offer!` macro useful in providing an overview of many branches, not only does it resemble a match statements, it also allows meaningful labels to be added.

5.4.3 An Unknown Number of Clients

First, we would like to examine how one might write a server that waits for an unknown number of clients to establish a connection, then receives a number and adds 42 to that number and returns the sum, unless the sum is above 255, in which case the addition overflows and the server indicates thus.

In `session-types`, the protocol for the clients, and the code for the client- and server-handler could be written like in Listing 41. However, we run into problems when we try to connect the two. We cannot create a `Chan` type and share it between clients, because `Chan`s cannot be copied or cloned. It is conceivable that we could put such a channel in some sort of shared reference counted structure, but would not know when we are done using it, and we might not be able to properly close it. Instead, we can let the clients create the session-typed channels themselves and have a `Sender<Chan<(), Server>>` that allows them to send one endpoint to the server. Since `Sender`s can be cloned, we are free to give one to each of the clients, while the server owns the `Receiver`. In code, it looks like Listing 42.

Note, that we are using a bit of Rust magic to make this happen: Most notably, `Sender`s are reference counted, and once there are no more references to a `Sender`, the corresponding `Receiver` will `Err`, which we use to stop the loop in `server`. This only works, however, because of the explicit `drop` in the main function; without it, `server` would hang.

In conclusion, we can say that this is a useful pattern, and it turns out that we can express

```rust
type Server = Recv<u8, Choose<Send<u8, Eps>, Eps>>;
type Client = Send<u8, Offer<Recv<u8, Eps>, Eps>>;

fn server_handler(c: Chan<(), Server>) {
    let (c, n) = c.recv();
    match n.checked_add(42) {
        Some(n) => c.sel1().send(n).close(),
        None => c.sel2().close(),
    }
}

fn client_handler(c: Chan<(), Client>) {
    let n = random();
    match c.send(n).offer() {
        Ok(c) => {
            let (c, n2) = c.recv();
            c.close();
            println!("{} + 42 = {}", n, n2);
        },
        Err(c) => {
            c.close();
            println!("{} + 42 is an overflow :(", n);
        }
    }
}
```

Listing 41: The `Server` and `Client` types, as well as the implementation of the `client` and `handler` function.

it in a straightforward manner using our session type constructs and Rust's standard library functions.

5.4.4 The Santa Claus Problem

The Santa Claus problem appeared is a concurrency exercise that first appeared in [1]. Santa Claus sleeps in his hut in the North Pole and can be awakened either by his elves when they need assistance with their toy-making or by his nine reindeer back from holiday and ready to deliver presents. To avoid disturbing Santa all the time the elves must come in groups of three, so the first elf in need of assistance will wait for two others to join him, before waking up Santa. While a group of elves are talking to Santa, a new group may form outside his door, but they cannot go in until the first group has exited. All nine reindeer must be back before waking up Santa, and because the reindeer are antsy and want to return on holiday in the tropics they must take precedence if a group of elves is ready at the same time.

We implement a solution of the Santa Claus problem using our session types library. The original solution uses semaphores and mutexes, but our approach is slightly different, because we want to showcase how session types can be used to structure the communication among the different parties.

An elf works and interacts with Santa using a single channel. When in need of assistance the elf sends a unit value on his channel to signal that he is ready and blocks immediately for a confirmation that Santa is ready to talk. Talking to Santa is represented as a send (the elf asks a question) followed by a receive (Santa gives an answer). This protocol, `Elf`, is written as:

```rust
type Elf = Rec<Send<(), Recv<(), Send<(), Recv<(), Var<Z>>>>>>;
```

The `elf` function shown in Listing 43 implements an elf that works for a random amount of

```rust
fn server(rx: Receiver<Chan<(), Server>>) {
    loop {
        match rx.recv() {
            Ok(c) => {
                spawn(move || server_handler(c));
            },
            Err(_) => break,
        }
    }
}

let (tx, rx) = channel();
let n: u8 = random();

for _ in 0..n {
   let tmp = tx.clone();
   spawn(move || {
       let (c1, c2) = session_channel();
       tmp.send(c1).unwrap();
       client_handler(c2);
   });
}
drop(tx);

server_handler(rx);
```

Listing 42: Using a `Sender` to connect an arbitrary number of sessions to a server.

time before being in need of help. At that point, the elf will use his channel to get help from Santa.

The elf must not communicate directly with Santa until the right time, so we implement a secretary handling the waiting elves. The secretary, Edna, receives the elves one by one in the waiting room, and groups them in threes to go wake up Santa. Edna holds the channels to all elves and receives their requests for help. When three elves have requested help, she delegates the elf channels to Santa who completes the elves' requests and returns the channels to Edna (i.e. shows the elves out the door). See Listing 44.

To select over the list of elf channels, Edna uses a function called `hselect`. `hselect` is short for "homogenenous" select and it receives a list of channels, all of which have the same protocol `Recv<A, P>` and picks out from the list the channel that is ready to communicate. It returns a tuple containing the selected channel and the rest of the list. `hselect` is included in the session types library and provides a convenient way to select over a list of channels of the same type.

The session delegation is transparent for the elves, they do not need to communicate among each other, because Edna ensures that elves do not progress until enough have requested help. The reindeer are handled in a similar fashion, through another secretary, Robin, who waits for all reindeer to return before Santa is notified.

The final actor in this system is Santa Claus. Santa has two channels: one on which he receives elves and another on which he receives reindeer. The implementation is shown in Listing 45. The communication between Santa Claus and his secretaries is modelled by the protocol:

```
Rec<Recv<Vec<SantaElf>, Send<Vec<DoneChan>, Var<Z>>>>
```

where `SantaElf` and `DoneChan` are the following declarations:

```rust
fn elf(id: usize, c: Chan<(), Elf>) {
    let mut c = c.enter();
    let mut g = thread_rng();
    let range = Range::new(0, 1000);
    loop {
        sleep_ms(range.ind_sample(&mut g));
        c = {
            let c = c.send(());          // We want to talk to santa
            let (c, _) = c.recv();       // Santa is ready to talk to us
            println!("Elf {} talking to santa", id);
            let c = c.send(());          // We talk to santa
            let (c, _) = c.recv();       // Santa talks to us
            c.zero()
        }
    }
}
```

Listing 43: A working elf.

```rust
type SantaElf = Chan<(EdnaElf, ()), Send<(), Recv<(), Send<(), Var<Z>>>>>
type DoneChan = Chan<(EdnaElf, ()), Var<Z>>
```

i.e. Santa waits to receive a vector of elf channels, and then returns a vector of channels that are ready to recurse. The `SantaElf` channel type has exactly the protocol corresponding to the sequence: Santa acknowledges elf, elf asks a question and finally Santa gives an answer. The `DoneChan` type reflects the state of the received channels after these interactions have been carried out.

The precedence requirement is handled implicitly by `chan_select!`, because it evaluates its arguments in order and picks the first that is ready to interact. Putting `reindeer` first then ensures that if both reindeer and elves are ready the reindeer will be selected. This could potentially lead to starvation of the elves, if the reindeer were always ready to deliver gifts, but the threshold for elves is a third of the threshold for reindeer, so this is unlikely to occur (preventing starvation of the elves is not a requirement either).

```rust
fn edna(mut elves: Vec<Chan<(EdnaElf, ()), EdnaElf>>,
        santa: Chan<(), Rec<Send<Vec<SantaElf>>,
        Recv<Vec<DoneChan>>, Var<Z>>>>>) {
    let mut santa = santa.enter();
    loop {
        let mut queue = Vec::new();
        while queue.len() < ELF_THRESHOLD {
            let (elf, tmp) = hselect(elves);   // Wait for elves
            let (elf, _) = elf.recv();
            queue.push(elf);                   // Add to wait queue
            elves = tmp;
        }
        santa = {
            let santa = santa.send(queue);     // Send queue to santa

            let (santa, queue) = santa.recv(); // Receive queue from santa

            for elf in queue.into_iter() {     // Push queue to elves
                elves.push(elf.zero());
            }
            santa.zero()
        }
    }
}
```

Listing 44: Edna handling elves.

```rust
fn santa(elves: Chan<(), Rec<Recv<Vec<SantaElf>>, Send<Vec<DoneChan>>, Var<Z>>>>>,
         reindeer: Chan<(), Rec<Recv<Vec<SantaElf>>, Send<Vec<DoneChan>>, Var<Z>>>>>) {
    let mut elves = elves.enter();
    let mut reindeer = reindeer.enter();

    loop {
        chan_select! {
            (c, v) = reindeer.recv() => {              // Receive all reindeer
                let v = v.map_in_place(|r| {           // Get ready to deliver gifts
                    let (r, _) = r.send(()).recv();
                    r
                });
                let v = v.map_in_place(|r| r.send(())); // Deliver gifts
                println!("Done delivering gifts.\n");
                reindeer = c.send(v).zero();           // Return to Robin
            },
            (c, v) = elves.recv() => {                 // Receive elves
                let v = v.map_in_place(|r| {           // Receive questions
                    let (r, _) = r.send(()).recv();
                    r
                });
                let v = v.map_in_place(|r| r.send(())); // Answer them
                println!("Done talking.\n");
                elves = c.send(v).zero();              // Return to Edna
            }
        }
    }
}
```

Listing 45: Santa Claus handling elves and reindeer.

5.5 Monadic Session Types

Before we settled on the current implementation of `session-types`, we considered various approaches, the most prominent one being an almost literal translation of the Haskell implementation described in Section 4.3. For the sake of completeness, this section describes the implementation and why we chose not to keep it. The code in this section does not work with newer versions of Rust, because it has not been updated since rust-1.0.0-alpha. The code would need to be rewritten to work with the newest version of Rust, but most of it should make sense.

```
pub struct Session<S, S_, A: marker::Send> (
    Box<for <'a>Invoke<&'a UChan, A>+marker::Send>
);
pub struct Cap<E,R>;

pub fn send<A: Send, E, R>(x: A) -> Session<Cap<E,Send<A,R>>, Cap<E,R>, ()> {
    Session( box move |: c: &UChan| unsafe_write_chan(c,x))
}

impl<A: Send, S, T> Session<S, T, A> {
    pub fn ret(a: A) -> Session<S, T, A> {
        Session (box move |: _:&UChan| a )
    }

    pub fn bind<B: Send, U>(self, k: Box<Invoke<A,Session<T, U, B>>+Send> )
                            -> Session<S, U, B>
    {
        Session (box move |: c: &UChan| {
            let Session(m) = self;
            let a = m.invoke(c);
            let Session(m_) = k.invoke(a);
            m_.invoke(c)
        })
    }

    pub fn then<B: Send, U>(self, next: Session<T, U, B>)
                            -> Session<S, U, B>
    {
      // snip, similar to `bind`
    }
}
```

Listing 46: Sample functions and declarations (`Session`, `Cap`, `send`, `ret`, and `bind`) from the monadic Rust implementation of session types.

The data types for describing session-typed channels and duality in this implementation of session types are identical to the the ones in the final implementation. However, instead of a `Chan` type for explicit channels, the monadic implementation of session types relies on a `Session` monad that carries an implicit channel. The definition of `Session` and `Cap`, which encapsulates the environment and protocol, can be seen in Listing 46. They are direct translations of the Haskell data types. Now, because channels are implicit, the session type operations like `send` (also shown in Listing 46) need to return a closure enclosed in a `Session`. Finally, to thread all of the sessions together we use `bind`, which is a translation of the `>>>=` function.

Using all of this, the (non-recursive) arithmetic server might be implemented as shown in Listing 47. The code is more or less a straightforward adaptation of the Haskell code.

```
fn server() -> Session<Cap<(), Offer<Recv<int, Recv<int, Send<int, Eps>>>,
                                     Recv<int, Send<int, Eps>>>>, (), ()> {
    offer(recv()
          .bind(box move |: a: int| recv().bind(box move |:b: int| send(a + b)))
          .then(close())
          ,
          recv()
          .bind(box move |: a: int| send(-a))
          .then(close()))
}
```

Listing 47: An implementation of the arithmetic server using the monadic implementation of session types in Rust.

However, it is neither idiomatic Rust nor is it ergonomic to work with. When it comes to working with functional values and monads, Rust's notation is not as terse as that of Haskell. On the contrary, wrapping everything in closures feels very forced, and there is nothing like an `ixdo` preprocessor to help us with `bind`s.[7] Additionally, Rust does not currently optimize recursive closures well, which means that there is a high risk of running into stack overflows.

All this considered, we decided to explore other ways to implement session-typed channels in Rust. Rust's affine types allows us to express the same requirements as the approach using indexed monads, and the result is the implementation decribed in Section 5.2

This concludes the brief overview of our implementation of monadic session types in Rust. In the end, we deemed the monadic approach unfriendly to work with, and instead reached the current implementation. The next section will discuss some alternative implementation strategies and library designs that we never had time to experiment with.

5.6 Alternative Designs

Aside from the monadic design approach, we considered other ways of designing the session types implementation that we will briefly outline in this section.

Syntax extensions Through compiler plugins we could potentially embed support for session types as syntax extensions to Rust. It would offer greater flexibility in the choice of syntax, but would wrap all session interactions in macro invocations. Leveraging syntax extensions could possibly be used to implement multi-way branching with labels as well. Embedding special syntax in Rust is fine for small uses and is a common practice in Rust projects, but embedding an entire DSL that is supposed to mix with other statements is neither idiomatic nor ergonomic.

We also experimented with smaller syntax extensions to make protocol declaration as type aliases more ergonomic. Writing big type expressions is error-prone when trying to keep track of matching `<` and `>`, but we found ourselves getting used to writing these types and consider a declaration macro a useful feature, but not a necessity.

Labelled, multi-way branching We considered different ways to bring labelled, multi-way branching into our library, but found no satisfactory way to do it. The only way we could see to implement labels without turning them into run-time values would be through syntax extensions, but this approach has already been addressed above.

An alternative approach we considered was to introduce more structs to reflect different numbers of branches. For example we would have `Choose3`, `Offer3`, `Choose4`, `Offer4` and so on,

[7] Though we could potentially write one ourselves. Rust's plugin system is quite powerful, as we shall see in Section 7

but that obviously puts an upper limit on the number of branches we would be able to offer. In the end we decided to stick to binary branches.

Iteration structs We experimented with and implemented iteration constructs inspired by the constructs provided in SessionJ [19]. We defined the structs

```
pub struct IterIn<P, N> (PhantomData<(P, N)>);
pub struct IterOut<P, N> (PhantomData<(P, N)>);
```

where `IterIn` is the passive iteration and `IterOut` the active. The first phantom type parameter `P` is the body of the iteration, the `N` parameter is action to execute once iteration terminates. Like the `offer` method, a `Chan` whose next action is `IterIn` implements a method `in_while` that returns a `Result` indicating whether to iterate or not. It performs the following transformation:

$$\texttt{Chan<E, IterIn<P, N>>} \longrightarrow \texttt{Result<Chan<(IterIn<P, N>, E), P>, Chan<E, N>>}$$

Either the `IterIn` type is pushed on the stack and the body is executed, or the `IterIn` is removed from the type and `N` is executed. `Chan` types whose next actions are `IterOut` implement `out_while` that iterate and `exit_while` that ends iteration and continues with the next action.

To mark the end of an iterating construct a special `IterEps` struct is used, that serves the same function as `Var<Z>`. Once `IterEps` is reached, the user must call an `iter` method to replace the `IterEps` with the protocol on top of the stack. Its transformation is as follows:

$$\texttt{Chan<(P, E), IterEps>} \longrightarrow \texttt{Chan<E, P>}$$

By construction, the only possibilites for `P` in a typable program are `IterIn` and `IterOut`. Note that `IterEps` can only be used with a non-empty protocol stack, so typable programs can only have `IterEps` nested in either `IterIn` or `IterOut`. It is therefore not possible to express protocols that do not have the option of terminating.

We debated the inclusion of the iteration constructs in favor of the recursive constructs, but decided in the end to leave them out. The recursive constructs are more expressive and also keeps our session types DSL in line with the Haskell version.

5.7 Evaluation

The `session-types` library provides a pure Rust implementation of session types that relies on the polymorphic type system and affine types. The implementation is clean and simple and we have found that once you get accustomed to using affine types, writing correct programs become a breeze. The affine type system cannot give all the guarantees that we require, because we cannot prevent channel values from being dropped prematurely. To achieve this property linear types are required. In the meantime, we can urge users to be diligent in explicitly calling the `close` method on all channels, thus making sure that protocols are run to completion.

6 Session Types in Servo

This section presents our work in translating Servo's internal communication patterns from enum based messaging to session-typed channels. We refer back to Section 3 for an explanation of roles of the different processes in Servo.

In Section 6.1 we discuss the paint task and its interacting processes in detail and present a translation of their communication schemes from regular channels to session-typed channels. The paint task is an interesting case: It is an integral part of the pipeline, but it primarily handles requests from the compositor. Specifically, we discuss how to handle the non-trivial case of shutting down a paint task. In Section 6.2 we discuss some of the challenges we encountered in Servo, and in Section 6.3, we move on to another example that demonstrates how a classic client-server setup can be re-interpreted in the session types framework. Finally, we discuss our experience with porting Servo to `session-types` in Section 6.4.

6.1 The `PaintTask`

The paint task listens on a port receiving messages of the type shown in Listing 48. For example, receiving a `Paint` message, the task performs the following actions:

- If we do not have permission to paint, send a `PainterReady` message to the constellation and notify the constellation of discarded paint messages
- Otherwise, signal to the compositor that we are painting and handle all the requests submitted with the `Paint` message.
- When finished, signal the compositor that we are idle.

```
1 pub enum Msg {
2     PaintInit(Arc<StackingContext>),
3     Paint(Vec<PaintRequest>, FrameTreeId),
4     UnusedBuffer(Vec<Box<LayerBuffer>>),
5     PaintPermissionGranted,
6     PaintPermissionRevoked,
7     Exit(Option<Sender<()>>, PipelineExitType),
8 }
```

Listing 48: The paint task `Msg` enum type.

Only three tasks communicate directly with the paint task: The compositor, the constellation and the layout task. To model the communication scheme with the paint task, we begin by looking at which variants of the enum type are used by which tasks. The result of this is shown in Figure 5, from which we see that the enum variants can be disjointly grouped—that is, all except for `Exit`.

Both the constellation and layout task may send the `Exit` message to the paint task, which in itself already suggests inconsistencies. We saw in Section 3.3 that the layout task is responsible for shutting down the paint task under normal circumstances, but in failure cases (forced exits) the pipeline handles shutdown. To avoid spreading the responsibility among tasks, we will have the pipeline always handle shutdown. The paint task is not shared with other pipelines, so this strategy should not present any problems.

During initialization of a new pipeline, both the pipeline and its layout task receive copies of the paint task channel. The compositor, however, does not. The compositor receives a channel to a paint task through a `SetFrameTree` message. This message contains references to a new frame tree and for each frame a view of its associated pipeline, called a `CompositionPipeline`. A `CompositionPipeline` contains a clone of the script and paint channels,

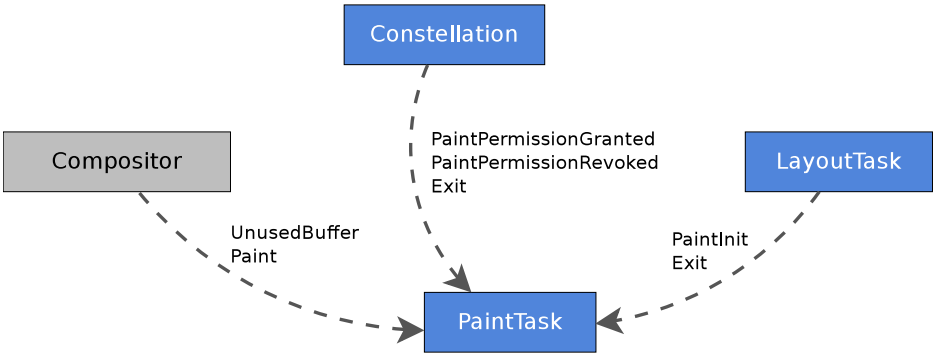


Figure 5: Communication with `PaintTask`.

allowing the compositor to communicate directly with these tasks. The compositor stores these `CompositionPipeline`s internally for later use.

The paint channel may be cloned multiple times. Whenever a pipeline in the history is revisited or an iframe is initialized, its script and paint channels are cloned to be passed to the compositor. The compositor does not keep track of whether or not it has already received channels to a given pipeline, but overwrites any previous values. This is not a problem for regular channels, but it is a problem for us, because it means that a stored channel will not properly close its connection (it is just dropped). We want to obey the principle that all protocols are run to completion and connections are closed properly, so we cannot allow the compositor to just overwrite a stored channel.

We start by translating each of the variants of the `Msg` to corresponding session type declarations:

```
type UnusedBuffer = Recv<Vec<LayerBuffer>, Var<Z>>
type Paint = Recv<(Vec<PaintRequest>, FrameTreeId), Var<Z>>
type PaintPermissionGranted = Var<Z>
type PaintPermissionRevoked = Var<Z>
type PaintInit = Recv<Arc<StackingContext>, Var<Z>>
```

We will discuss how `Exit` is handled later. Note how the permission control does not send or receive any values, rather the selection of a branch in the protocol is enough to encode the same information. All these protocols end with `Var<Z>`, because they are expected to be nested within a `Rec` construct.

We declare a protocol for each of the tasks communicating with the paint task. They are shown in Listing 49 from the point of view of the paint task.

```
1  type Compositor = Offer<UnusedBuffer,
2                    Offer<Paint,
3                          Eps>>;
4
5  type Pipeline = Offer<PaintPermissionGranted,
6                  Offer<PaintPermissionRevoked,
7                        Exit>>; // To be defined later
8
9  type Layout = Offer<PaintInit,
10                     Eps>;
```

Listing 49: Three protocols for communicating with the paint task.

To address the issue of channel cloning we could keep track of the open sessions and have the paint task handle multiple connections to the compositor. Instead of overwriting stored channels, the compositor would first close any existing connections, and then store the newly

received channels. This approach requires the paint task to be able to receive new compositor connections at any point and be able to select over multiple channels. Alternatively, we can keep track of whether or not a connection between the compositor and a paint task has already been established. This second approach is desirable, because it simplifies implementation in the paint task and avoids excessive copying of data.

Recall that pipelines are organized in a frame tree to handle iframes: The compositor keeps a reference to the root of the pipeline tree in `root_pipeline`. However, the introduction of a `Chan` type in `CompositionPipeline` prevents cloning, so the `root_pipeline` cannot be a direct clone of root. But usage of the `CompositionPipeline` is internal to the compositor, and it does not lend out references or make clones to pass to other threads. This means we can store `CompositionPipelines` in reference-counted pointers with `Rc`. The type of `root_pipeline` then becomes `Rc<CompositionPipeline>`. The `Rc` type is not `Sync`, so we cannot risk memory unsafety with this.

We still need to handle the initialization of the session between the compositor and the paint task. This session is established on demand, so we will change the protocol between the pipeline and paint task to include the following branch:

```
type CompositorChan = Recv<Chan<(), Compositor>, Var<Z>>
```

The `Pipeline` protocol is then modified to the version shown in Listing 50, where we still need to define `Exit`. This allows the pipeline's `to_sendable` method to create a new session and pass one end-point to the paint task. `CompositorChan` demonstrates how session delegation is effortlessly supported in our session types system, as channels are first-class values.

```
1 type Pipeline = Offer<PaintPermissionGranted,
2                 Offer<PaintPermissionRevoked,
3                 Offer<CompositorChan,
4                       Exit>>>;
```

Listing 50: Extending the pipeline protocol.

There is one drawback to the `Pipeline` protocol: It does not reflect the requirement that at most one compositor channel should be transmitted. This could be addressed in different ways. The paint task could be initialized with a compositor session, but this requires the pipeline to keep track of the other end of the session until it can hand it off to the compositor. Preferably, it should be possible to execute the `CompositorChan` at most once. Listing 51 shows an enhanced version of the `Pipeline` protocol that only allows the paint task to receive a compositor channel once. The `Rec` pushes the `PipelineInner` on the protocol stack, and the branches of `PipelineInner` do not pop any items off this stack, so the outer protocol becomes inaccessible, because there is no interaction trace that could execute the third branch more than once. We are explicitly encoding state in the interaction between the pipeline and paint task.

```
1 type Pipeline = Offer<PaintPermissionGranted,
2                 Offer<PaintPermissionRevoked,
3                 Offer<Recv<Chan<(), Compositor>, Rec<PipelineInner>>,
4                       Exit>>>;
5
6 type PipelineInner = Offer<PaintPermissionGranted,
7                      Offer<PaintPermissionRevoked,
8                            Exit>>;
```

Listing 51: Encoding a restriction on the number of times the `CompositorChan` branch can be executed.

```
fn run(&mut self,
       pipeline_chan: Chan<(), Rec<Pipeline>>,
       layout_chan: Chan<(), Rec<Layout>>) {

    let mut pipeline_chan = Some(pipeline_chan.enter());
    let mut layout_chan = Some(layout_chan.enter());
    let mut compositor_chan = None;

    enum ChanToRead {
        Pipeline,
        Layout,
        Compositor
    }

    while pipeline_chan.is_some()
            || layout_chan.is_some()
            || compositor_chan.is_some() {
        let chan_to_read = {
            let mut sel = ChanSelect::new();
            if let Some(ref pipeline_chan) = pipeline_chan {
                sel.add_offer_ret(&pipeline_chan, ChanToRead::Pipeline);
            }
            if let Some(ref layout_chan) = layout_chan {
                sel.add_offer_ret(&layout_chan, ChanToRead::Layout);
            }
            if let Some(ref compositor_chan) = compositor_chan {
                sel.add_offer_ret(&compositor_chan, ChanToRead::Compositor);
            }
            sel.wait()
        };
        match chan_to_read {
            ChanToRead::Pipeline   => { /* handle message from pipeline */ }
            ChanToRead::Layout     => { /* handle message from layout */ }
            ChanToRead::Compositor => { /* handle message from compositor */ }
        }
    }
}
```

Listing 52: The paint task's `run` method.

Although we can encode this restriction and it leads to a more accurate representation of the intended interaction patterns, we opted for the first version, because it is simpler. The enhanced version of the protocol requires a greater book-keeping effort on the part of the programmer and it may unnecessarily burden the programmer rather than assist her.

A skeleton implementation of the paint task's `run` method is shown in Listing 52. The idea is that all the tasks communicating with the paint task eventually close the connection. So the paint task stores its connections in `Option` types and only exits when all of them are `None`. Initially the paint task receives a connection to the pipeline and layout task, but not to the compositor. Handling the reception of a compositor channel and assigning it to the `compositor_chan` variable must be dealt with explicitly by the programmer.

We still need to consider the shutdown sequence and fill in the unspecified `Exit` branch. As discussed in Section 3.3 the shutdown sequence is complex and depends on whether it is a single pipeline being shut down or the entire application. In the latter case, we need to ensure the compositor closes all its connections to different pipelines. We do not need to assign responsibility for shutting down the paint task to any task in particular, because,

as Listing 52 demonstrates, the paint task will wait for *all* its connections to be closed, before terminating.

A single-pipeline shutdown is more intricate. As the application will continue to run, it is important that resources used by the pipeline being shut down are properly cleaned up, so the paint task wants to collect all its buffers from the compositor before exiting. The pipeline must therefore wait for the paint task to report that it is ready to shut down before exiting itself. We can define `Exit` as:

```
type Exit = Offer<Eps, Send<(), Eps>>
```

The `Offer` encodes the difference between complete and pipeline-only shutdown. In the former case, the connection is merely closed. In the latter however, the paint task must send an acknowledgment, encoded as `()`, before closing the channel. This construction allows the pipeline to wait for the paint task at the correct time and ensures the pipeline does not attempt to engage any further with the paint task.

When the pipeline requests a pipeline-only exit, the paint task then stores a channel value with the following type:

```
Chan<(Pipeline, ()), Send<(), Eps>>
```

and delays the acknowledgment until it has received all outstanding buffers. This is demonstrated in Listing 53. The `handle_compositor` method is called when the `run` method has determined the compositor is ready to communicate. Apart from the compositor channel to interact with, it receives also a `pipeline_exit_chan` which may be `None`. Following the `Compositor` protocol, three choices are offered, the first being to receive unused buffers. Upon receiving these unused buffers an internal counter is decremented and if the counter reaches zero, we send an ack on the `pipelin_exit_chan` if provided. The connection to compositor must also be shut down, but this must happen in a separate interaction once the compositor is notified of the pipeline's exit.

6.2 Challenges in Servo

To consume a session-typed channel, we must own the channel. It is not enough to hold a reference like `&` or `&mut`. This is exactly the property we want to have, because it directly (and elegantly) addresses the aliasing problem. But this requirement does present some challenges. Specifically, storing channels in structs for later consumption is not particularly straightforward, because the usage locations may only be borrowed, in which case we cannot consume the channel.

We considered several solutions, but the one we ended up with, as suggested by Lars Bergstrom, is to wrap the `Chan` value in an `Option` type and use its `take` method. It has the following declaration:

```
fn take(&mut self) -> Option<T>
```

This function returns the contents of the `Option`, leaving `None` in its place. From the declaration we see that this requires a `&mut` context, so this approach enables us to consume the contained value in a mutably borrowed context Putting a value back into the `Option` is just assigning a value of type `Some(T)`. This method of allowing consumption of mutably borrowed values using the `take` function is also colloquially referred to as the "`Option` dance".

Next we need to deal with immutable borrows. In Servo, the majority of contexts in which `Senders` and `Receivers` are used, are immutably borrowed contexts, i.e. in methods declared with `&self`. To address this issue we need to investigate the structs in which the channels (in particular `Senders`) are stored. If the struct is not shared across threads we can wrap it in a `RefCell`, which, in an immutable context, will allow us to mutably borrow its content. The "extended `Option` dance" then becomes:

```
fn handle_compositor(&mut self,
                     compositor_chan: Chan<(Compositor, ()), Compositor>,
                     pipeline_exit_chan: Option<Chan<(Pipeline, ()), Send<(), Eps>>>)
                     -> (Option<Chan<(Compositor, ()), Compositor>>,
                         Option<Chan<(Pipeline, ()), Send<(), Eps>>>) {
    offer! {
        compositor_chan,
        UnusedBuffer => {
            let (c, unused_buffers) = compositor_chan.recv();
            debug!("PaintTask: Received {} unused buffers", unused_buffers.len());
            self.used_buffer_count -= unused_buffers.len();

            for buffer in unused_buffers.into_iter().rev() {
                self.buffer_map.insert(native_graphics_context!(self), buffer);
            }

            if self.used_buffer_count == 0 {
                debug!("PaintTask: Received all loaned buffers.");
                pipeline_exit_chan.map(|c| c.send(()).close());
                (Some(c.zero()), None)
            } else {
                (Some(c.zero()), pipeline_exit_chan)
            }
        },
        Paint => { /* Omitted */ },
        Close => {
            compositor_chan.close();
            (None, None)
        }
    }
}
```

Listing 53: Sending ack to the pipeline at the correct time.

```
fn foo(&self) {
    // stored_channel: RefCell<Option<Chan<E, P>>>
    let mut chan_ref = self.stored_channel.borrow_mut();
    let chan = chan_ref.take().unwrap();

    // new_chan is result after interacting with chan
    *chan_ref = Some(new_chan);
}
```

This extended trick to allow consumption of a stored channel makes use of the interior mutability provided by `RefCell` and the `take` method on `Option`. The underlying assumption is that these channel cells are not shared across thread boundaries, i.e. these cells should not be considered `Sync`. Fortunately, `RefCell` is not `Sync`, so we are not introducing any memory unsafety.

6.3 The `StorageTask`

We also experimented with replacing the communication of another task, the storage task. The storage task works as a key-value store, and clients may store and request bits of information from the task. In this regard the storage task resembles a typical server, acting in a request-response pattern.

The communication API of the storage task is shown in Listing 54. The `StorageTaskMsg`

```
pub enum StorageTaskMsg {
    Length(Sender<u32>, Url),
    Key(Sender<Option<DOMString>>, Url, u32),
    GetItem(Sender<Option<DOMString>>, Url, DOMString),
    SetItem(Sender<bool>, Url, DOMString, DOMString),
    RemoveItem(Sender<bool>, Url, DOMString),
    Clear(Sender<bool>, Url),
    Exit
}
```

Listing 54: The storage task protocol.

enum includes variants for getting and setting key-value pairs and querying the size of the stored data. All the variants, except `Exit`, include a `Sender` on which the storage task can provide a reply. On receiving `Exit` the storage task acts like the resource manager presented in Section 3.2 and exits immediately. This strategy is obviously brittle, because client connections may be suddenly cut when the storage task shuts down and clients may crash as a result.

The translation to session types can be done piece-wise, by first declaring the following protocols fragments:

```
type Length     = Recv<Url, Send<u32, Var<Z>>>;
type Key        = Recv<(Url, u32), Send<Option<String>, Var<Z>>>;
type GetItem    = Recv<(Url, String), Send<Option<String>, Var<Z>>>;
type SetItem    = Recv<(Url, String, String), Send<bool, Var<Z>>>;
type RemoveItem = Recv<(Url, String), Send<bool, Var<Z>>>;
type Clear      = Recv<Url, Send<bool, Var<Z>>>;
```

These fragments all receive a value, return a response and then recurse. We combine them in an `Offer` chain as follows:

```
type StorageSrv = Offer<Length,
                  Offer<Key,
                  Offer<GetItem,
                  Offer<SetItem,
                  Offer<RemoveItem,
                  Offer<Clear
                        Eps>>>>>>;
```

The `StorageSrv` type represents the communication scheme available for clients. A client can store and retrieve information and close the connection. Therefore, selecting the `Eps` branch does not tell the storage task to shut down, it merely closes the connection. To handle shutdown, the constellation has its own connection to the storage task through which it can request shutdown. This connection can be described succinctly as the protocol:

```
type ConstellationToStorage = Send<(), Eps>;
```

This "one-shot" protocol allows the storage task to select on all its connections and receive an exit message at any time. By splitting the connections, the storage task can be allowed to wait for other clients to close their connections before exiting.

We must also handle the addition of new clients. This could be done by adding another branch to the constellation connection to allow the constellation to send new channel endpoints. We rewrite the `ConstellationToStorage` protocol as follows:

```
type ConstellationToStorage = Choose<Send<Chan<(), Rec<StorageSrv>>, Var<Z>>,
                                     Eps>;
```

The first branch delegates a new client channel and the second branch is the shutdown signal. It is thus the responsibility of the constellation to initiate new sessions between the storage task and its clients.

We implemented an example storage task to experiment with different approaches to the communication scheme (outside Servo). In this example the storage task spawned a new thread per incoming client, and the underlying hash map was then wrapped in an `Arc<Mutex<..>>` to allow the client-handling threads to share the data. In effect we pushed the synchronization point from the storage task into a mutex guarding the data structure. This approach gives rise to even more threads in a system that already spawns many threads, and may for this reason not be desirable. An alternative strategy would be to have the storage task select over all its connections using the `ChanSelect` structure that we introduced in Section 5.4.1.

6.4 Experience

Our experience with introducing session types in Servo is that restructuring communication has far-reaching implications and is by no means trivial. Session-typed communication requires forethought and thorough planning, but once the protocols and channels are established, it seems impossible to use a channel incorrectly, because any incorrect use will not type-check.

In this chapter we have demonstrated that session-typed communication in larger applications offers fine-grained control and forces the programmer to consider all possible communication patterns, but the restriction to binary communication with non-shareable end-points can be unergonomic and requires a greater effort of the programmer to implement. We experimented with ways to share a `Chan` type across threads in a safe manner, but never arrived at a satisfactory solution (like a distributed `Option` dance). We also felt that this was not the right approach for `session-types`.

We have also attempted to enforce the principle that all connections are closed, but in reality threads crash for various reasons, and our communication schemes do not handle this issue.

7 Linear Types in Rust

Using session-typed channels gives the user certain safety guarantees regarding inter-process communication, but `session-types` has certain limitations because Rust's type system is affine and not linear. As an example, consider the code in Listing 55 that compiles without warning, but will fail when actually run: the client never sends anything to the server, but the server (quite reasonably) expects a number and panics when no such number arrives. Our implementation of `recv` assumes correct behavior from the other endpoint, but `client` never uses its `Chan`, so the underlying `Sender` is dropped and `recv` panics as a result thereof. Rust allows the user to drop a value at any time, either implicitly by not referring to it any more (or placing a wild card _ instead of a name in a pattern, as in Listing 55), or explicitly by using the `drop` function. This is the reason why `session-types` can guarantee that *if* something is sent, it is correctly interpreted, but cannot guarantee that anything is actually sent.

```
1  fn server(c: Chan<(), Recv<u8, Eps>>) {
2      let (c, _) = c.recv();
3      c.close();
4  }
5
6  fn client(_: Chan<(), Send<u8, Eps>>) {}
7
8  fn main() {
9      let (c1, c2) = session_channel();
10     spawn(|| client(c1));
11     server(c2);
12 }
```

Listing 55: A program using `session-types` that compiles, but panics when run.

To alleviate this problem, we would have to enforce that only channels of type `Chan<E, Eps>`, for some environment `E`, were ever dropped, preferably by calling the `close` method on them. Ideally, we would like to treat `Chan` types as linear types. Support for linear types in Rust has been discussed[8] and may appear in the future, but at the time of writing, the proposal has been postponed.

This problem is not unique to `session-types`. Indeed, the channels in the Haskell implementation of session types described in Section 4.3 are not linear—you can always raise an exception—and neither are the channels in SessionJ.

Although it is usually quite easy to keep track of your channels in smaller applications, it becomes progressively harder the more complex your application is. For instance, Servo consists of more than a hundred thousand lines of Rust code split over almost two thousand files, so it can be hard to make sure that no session channels are ever prematurely dropped. Of course, the same is true for Rust's own message passing channels, but they were purposely designed to be safely drop-able, whereas `Chan` types are not.

To address this problem, we decided to implement a Rust plugin, `humpty_dumpty`, to track linearity of specific types and report linearity errors to the user.

The next section (Section 7.1) will describe the design and implementation of `humpty_dumpty`. Section 7.2 will discuss the limitations present in the current implementation of `humpty_dumpty`. Some can be fixed, but it is unlikely that we will be able to solve the linearity issue in the general case. Lastly, Section 7.3 briefly evaluates `humpty_dumpty` and our experience writing it.

[8] Most recently in `https://github.com/rust-lang/rfcs/issues/814`

7.1 Design and Implementation of `humpty_dumpty`

This section describes the design and implementation of a compiler plugin that tracks linear usage of annotated types. The Rust compiler exposes an advanced plugin infrastructure, that allows user-provided libraries to provide syntax extension and lint checks. Syntax extensions allow the programmer to extend the abstract syntax tree (AST), and lint checks allow custom project-specific checks to be implemented by traversing the fully typed AST. Servo showcases advanced uses of both these methods, for example building perfect hash maps at compile-time, auto-generating GC trace hooks and implementing safety checks for the interface with the SpiderMonkey garbage collector [30].

The goal with `humpty_dumpty` is to provide a lint that tracks usage of specially annotated types throughout the user's programs and warns the user on incorrect usage. For instance, in Listing 55, we would like `humpty_dumpty` to return an error stating that the `client` function does not correctly handle the `Chan` type linearly.

For the remainder of this chapter we will consider a type `Foo` that we wish to protect against being incorrectly dropped. The only way the user is allowed to get rid of a `Foo` value, is through the `close` function. We assume that there are no references to, or compound data structures containing, values of type `Foo`. We also assume that all called functions and methods are local to the current crate and that there are no generic functions, closures, or labelled loops.

For `humpty_dumpty` to correctly assert that no violations of linearity take place, the plugin should check all function definitions in the user's program to verify that all `Foo` values are correctly handled. For instance, if a `let` declaration introduces a new variable binding of the type `Foo` that variable should be tracked through the remainder of the program. To keep track of variables of interest, we use a hash map containing a reference to the original place of declaration for each value.

When checking a function, we first figure out which, if any, of the function arguments have type `Foo` and add those to the hash map. We then walk the body of the function, removing and adding variables to the hash map as necessary. Specifically, every time a protected value is consumed—by passing it to a function for instance—we remove it from the hash map, and every time a protected value is returned from a function, or otherwise brought into existence, we add it to the hash map. If any protected values are returned at the end of the function, we also remove those from the hash map. If the hash map is empty once we have traversed the entire function, we declare that this function upholds linearity for the protected type `Foo`. Specifically, if the hash map is empty when the entire function has been traversed, then all protected values have either been consumed by passing them on to other (supposedly correct) functions or returned. We also have to take into account Rust's control flow operations. Here, we will focus on `match` and `loop` statements, because we can regard these as generalizations of the remaining control flow structures (`for`, `if`, and `while`) in Rust.

First, we will show how we handle `match` statements. Consider the code in Listing 56. The two branch arms of the `match` statement are not equivalent in terms of linearity in the

```
1  let x = Foo;
2
3  match some_predicate {
4      true => { close(x); }
5      false => { }
6  }
```

Listing 56: A non-linear usage of the `match` statement.

`x` variable: The `true` branch consumes `x`, the `false` branch does not. Therefore, the match statement is not linear, and we want to catch cases like this.

The arms of the match statement are individual blocks and they can therefore not introduce new bindings that leave their block. They can mutate and consume values from, and return values to, the outside scope, but otherwise a match block cannot modify the enclosing scope. New linear values can be created, but they must be correctly consumed before the end of the block, otherwise they will be implicitly dropped. Finally, if one branch of a `match` statement consumes a protected value from the enclosing scope, all other branches must consume the protected value as well, otherwise the linearity state at the end of the `match` statement would be inconsistent. Therefore, whenever we reach a `match` statement, we make separate traversals of each branch. If, at the end, the hash maps for the branches are identical, they are linearly equivalent with respect to `Foo`, and if not, we can notify the user that the match arms are not linear. Checking that `match` arms are mutually equivalent with respect to linear variables is not sufficient though. We must also ensure that the hash map produced by each arm is a subset of the initial hash maps at the beginning of the `match` statement.

Our code also has to take into account the argument that is being matched, as well as the patterns in the different arms of the `match` statement. The argument is consumed like the arguments to a function and any protected values in the patterns are added to the hash map.

We must also consider how the `return` statement influences our analysis. With no enclosing control structures, `return` marks the end of a function, and we check that the hash map is empty and do not process any following statements. However, inside a `match` statement, some or all of the arms may call `return`, which complicates our analysis. Consider Listing 57

```
let x = Foo;

match some_predicate {
    true => { some_fn(x); return; }
    false => { }
}
```

Listing 57: An example showing usage of `return` in a `match` statement.

in which the `true` branch contains a `return` statement. Clearly, it is okay to return here (assuming `x` is the only protected value), but it does not make sense to compare the hash map of `true` branch to the hash map of the `false` branch, so we cannot tell if the entire match expression is valid or not. Instead, we keep track of an additional boolean entry to indicate whether we have encountered a `return` statement or not. Whenever we encounter a `return` statement, we set that boolean to true, and treat the expression just like the end of a function by asserting that the hash map is empty. Now, whenever we have traversed the branches of a `match` statement, we check for returning branches. If all branches are returning, then the whole `match` statement is returning and we do not have to worry about linearity, because the individual branches have already been checked. Otherwise we simply ignore the returning branches and proceed as described previously with the remaining non-returning branches.

We also have to handle `loop` statements. For simplicity, we will disregard labelled loops and labelled breaks. As an example, we will consider the code in Listing 58. and Listing 59. Clearly, Listing 58 is an example of correct linear usage in a loop: if `some_predicate()` is true, we consume `x` and break the loop, and otherwise `x` is not touched. However, although Listing 59 is similar, it is not correct: both branches of the `if` statement break the loop, but only one consumes `x`. For `humpty_dumpty` to correctly check loops, there are two primary areas of concern.

First of all, if we reach the end of the loop, the hash map should be identical to the hash map at the beginning of the loop. Like in the `match` statement we can mutate external protected values and create new local ones, but the local protected values have to be consumed before the end of the scope. Secondly, any time we encounter a `break` statement, we have

```
let x = Foo;
loop {
    if some_predicate() {
        close(x);
        break;
    } else {
        do_work();
    }
}
```

Listing 58: An example of a loop statement with protected values.

```
let x = Foo;
loop {
    if some_predicate() {
        close(x);
        break;
    } else {
        break;
    }
}
```

Listing 59: A non-linear loop.

to make sure that the current hash map contains a subset of the entries of the hash map at the beginning of the loop, otherwise there would be unconsumed protected values when exiting the loop. We also have to make sure that all `break` statements result in the exact same output hash map, otherwise there will be inconsistencies in the linearity of the loop. In essence, `break` statements in loops are analogous to the end of a match block with regards to linearity.

Furthermore, we need to consider the `continue` statement. However, it is similar to handling the end of a `loop` block: we must ensure that the current hash map is the same as the hash map at the beginning of the block.

Now that we have given a rough description of how `humpty_dumpty` works, we will discuss some of the assumptions we made at the beginning of this chapter: we assumed that there were no references (borrows) to protected values and that protected values did not appear in any structs or other data types (we had other assumptions, but those will be dealt with in Section 7.2).

First, we cannot completely ignore references: they are essential to many Rust programs and many programs would be impossible or very hard to express without them. Luckily, a simple reference like `&x`, where `x` is a `Foo`, is easily allowed: we simply ignore it and let the borrow checker do its job. However, it is also possible to write something like `let y = &SomeStruct { val: x }`, which actually consumes `x` but only returns a reference to a `SomeStruct`. When encountering anything that is being borrowed, we can only ignore it, if it is of the simple form described above. Otherwise, we have to traverse the whole type and figure out if it contains any protected values.

Similarly, we have to take data structures containing `Foo`s into account. For example, one can imagine the usefulness of a `Vec<Foo>` or an `Option<Foo>`. The way we handle these, is to track them as if they were regular `Foo` types. That is, any time we come upon a variable of some type `T`, we traverse that type to determine if it contains any occurrences of `Foo`. If that is the case, we track it through the remainder of the program. This approach covers simple cases, but not all of them. We discuss the limitations of `humpty_dumpty` in the next section.

This concludes the description of the design and implementation of `humpty_dumpty`. The code has been developed in collaboration with Manish Goregaokar, is available on GitHub,[9] and is published on `crates.io`.[10]

7.2 Limitations in `humpty_dumpty`

Although `humpty_dumpty` is able to detect a wide variety of linearity errors in a given program, as showcased by the many tests and examples also included with the code, it is by no means perfect. Rust is a complex language and there are limitations to what `humpty_dumpty` can do.

[9] `https://github.com/Manishearth/humpty_dumpty`
[10] `https://crates.io/crates/humpty_dumpty`

Some are inherent to Rust, and are not likely to be alleviated, while others are shortcomings in `humpty_dumpty`, which can and should be fixed. The biggest problems are closures, generic functions and external crates.

```
let x = Foo;
let cl = || { close(x); };
```

Listing 60: A closure that captures and correctly closes a protected value.

```
fn dropper<T>(_: T) {}

let x = Foo;
foo(x);
```

Listing 61: A generic function, `dropper`, which can currently be used to drop protected values without complaints from `humpty_dumpty`.

First of all, closures pose an interesting problem. Listing 60 shows a closure, `cl`, that captures a protected value, `x`, and correctly closes it using `close`. However, the code is only correct if `cl` is actually run, otherwise `x` is implicitly dropped when the closure runs out of scope. To address this, we need to make sure that `cl` is eventually invoked, which requires that `cl` is treated as a linear type. But `cl` is of type `FnOnce<()>` which does not tell `humpty_dumpty` that the value should be tracked. Therefore, tracking `cl` is not straightforward, and would require rewriting much of `humpty_dumpty`'s code.

Secondly, generic functions allow users to fool `humpty_dumpty` and unsafely drop protected values. The function `dropper` in Listing 61 is an example of such a function. It takes a value of generic type as an argument and promptly drops it. Because `humpty_dumpty` checks functions independently of each other, it does not know that `T` is actually instantiated to `Foo`, and it therefore happily—but wrongly—allows its argument to be dropped. We could identify at the call site of a generic function the types of its arguments, but the provided AST only has the generic version available (i.e. before monomorphization), so checking these require special treatment.

Finally, `humpty_dumpty` cannot traverse functions defined in external crates. The only information we have regarding functions from external crates is their signature and any attributes that they might have. Whenever we pass a protected value to an external function, we therefore have no way of knowing whether or not it will be correctly handled. Even though some external function has the signature `ext(Foo) -> Foo`, there is no way to guarantee that the `Foo` we get back is the same `Foo` we had to begin with. The first `Foo` might have been incorrectly dropped and a new one returned, or perhaps the external function does not even return a value of the same type, and we cannot tell if it correctly closes the `Foo` we passed. We can hope that the creator of the external library also uses `humpty_dumpty` to make sure that protected values are treated correctly, but even then, many functions from external crates that we would use are probably generic functions.

Lastly, we have ignored the issue of labelled loops: loops can be labelled, and both `break` and `continue` can specify a label, indicating which of the enclosing loops to break or continue from. Handling labelled loops would require keeping track of all enclosed loops instead of just the closest one. Doing so would require reworking the existing code, but it appears to be much more manageable to fix than either closures and generic functions. Unfortunately, we have not had the time to do so, so it will have to remain as future work.

7.3 Evaluating `humpty_dumpty`

The fundamental problem we wanted to address with `humpty_dumpty` is the following: For an annotated type `T`, ensure that no values of type `T` are ever implicitly dropped. It is of relevance to our session types implementation, because our `Chan` types are explicit values and can be unintentionally dropped. This contrasts with the implicit channels in [18] that cannot be unintentionally dropped, because they are enclosed in a monad.

The standard approach in Rust is to use the `#[must_use]` attribute, but it does not guarantee that certain types and return values are indeed used. For example, a common idiom when using the `Result` type (which is marked as `#[must_use]`) is to silence the warnings by a binding to the wildcard identifier. A statement like:

```
Ok::<u8, bool>(42);
```

will elicit a warning, because `Ok` is `#[must_use]`, but it can be silenced by the following binding:

```
let _ = Ok::<u8, bool>(42);
```

We concluded `#[must_use]` was not satisfactory for our purposes, as there were too many error cases that it would not catch, and set out to create a compiler plugin that traces linear usage of specifically annotated types. Although `humpty_dumpty` handles a collection of non-trivial cases, it does not handle them all, and it has shown that compiler plugins for Rust have limitations that put restrictions on what we can achieve.

We conclude that `humpty_dumpty` does not fully satisfy the goals we started out with. We have explored how to transform a programming language with affine types into one with linear type capabilities and found that true linear types require fundamental changes to the language. We expect that many of the shortcomings of `humpty_dumpty` can be addressed, but it will never be a true substitute for linear types in Rust.

8 Evaluation

This section summarizes the results of our work, compares it with other works in similar areas, and gives suggestions for future work that might be interesting. Section 8.2 discusses some related work, most notably OTP supervisors (of Erlang fame), which offer an alternative way of handling communication and communication failues. Section 8.3 lays out our thoughts and ideas for relevant and interesting improvements and extensions to our work. Section 8.4 summarizes our work and presents our conclusion.

8.1 Performance

One of the success criterias for Servo is performance. It has to be as least as fast as other browsers. Consequently, we cannot allow our internal communication to take up too much time.

Internally, the `Chan` types use the channel abstraction provided in Rust's standard library, so the performance of the internal send and receive operations should match their performance.

We have identified three potential causes for overhead in our implementation. First of all, we have to consider boxing and unboxing values when they are transmitted. Because our implementation of untyped channels currently relies on boxing the values and sending a transmuted reference to that boxed value over a `Sender` and `Receiver`, there is a direct overhead here. Secondly, branch selection incurs a direct overhead: each branch requires a boolean to be sent. Finally, the wrapper functions `unsafe_write_chan` and `unsafe_read_chan` can incur additional function call overhead.

The wrapper functions are in all likelihood not a problem, because the Rust compiler uses the LLVM compiler infrastructure, which is sophisticated enough to identify thin wrappers with the potential for inlining. Furthermore, we can annotate these functions with `#[inline]`, which hints to the compiler that these functions should be inlined. The `unsafe_write_chan` and `unsafe_read_chan` are both two-line functions (where one line effectively is a no-op) and are used in just two places each, so the overhead from inlining should be small.

Branch selection is an overhead inherent in the design of the library. In general, to select among n branches, we are must transmit at least $\log n$ booleans. Additionally, if the user chooses to use our `offer!` macro to ease legibility, the number of transmissions becomes linear. This overhead can only be addressed by redesigning the branching constructs, for example by implementing k-ary branching.

Boxing and unboxing of values is interesting, because boxing a value before we send it is not inherently slower than not boxing it. Indeed, for large data structures boxing is typically desired, because it avoids copying large chunks of memory. For small values, however, boxing will typically be inefficient. We decided to investigate the overhead of boxing for differently sized data types to get an insight into the trade-off. This is discussed in Section 8.1.1

We box values, because the standard Rust channels do not allow transmitting differently sized values over the same channel. But if we were to implement our own transmission mechanism that could handle values of different sizes, we would likely be able to get rid of the requirement for boxing.

The benefits of addressing the boxing overhead are multiple: Changing the underlying transmission mechanism does not imply a change in the library API, so addressing this issue would be transparent to the user. It also affects the cost of branching by lowering the cost of each transmission.

Section 8.1.1 investigates the cost of boxing values before they are sent over a channel, while Section 8.1.2 examines introducing session-typed channels in Servo's internal communication affects its performance. In both cases, we perform a variety of benchmarks, all of

	Unboxed		Boxed	
Bytes	μ	σ	μ	σ
8	28.218	1.8057	35.769	2.1372
64	34.264	1.7006	35.141	2.1226
512	56.809	2.5561	47.398	1.6484

Table 1: Numbers obtained from microbenchmark. All numbers are in microseconds (μs).

which were carried out on an Intel Core i7-4900MQ with eight cores each at 2.8GHz, and 16GB of RAM.

8.1.1 The Cost of Boxing

We decided to investigate the cost of boxing. To do this, we considered using Rust's built-in mechanisms for benchmarking, but by default only the median and the range are reported and we found that outliers were not dealt with. Instead we opted to use an implementation of the criterion benchmarking tool for Rust, `criterion.rs`.[11] The benchmarks were run using values of sizes 8 bytes (the size of an `f64`), 64 bytes and 512 bytes. For each of the sizes, 100 tests with transmitting boxed and unboxed values were run and in each test the value was transmitted 100 times. The results are reported in Table 1.

With a difference of 23.6%, we see that the overhead from boxing is large for 8-byte values. For 64-byte values, the time for unboxed values increase to roughly same time as for the boxed values (2.53% difference), and the time for transmitting boxed values does not change significantly with the size of the data. Jumping to 512 byte values, we see an increase in times for both boxed and unboxed values, but transmitting boxed values is now 18.06% faster than unboxed values.

This analysis suggests there is a potential performance gain to be had by specialising code to not box small values, while keeping large values boxed.

8.1.2 Performance in Servo

We also wanted to get an insight into how the changes introduced in Servo affected its overall performance. Internally the Servo team has a collection of static sites used for small benchmarks, and we were granted access to this repository.

Servo has built-in time and memory profilers that the user can hook into and request profiling output. But the output is specifically focused on certain computation-intensive tasks that do not necessarily involve communication, and we are interested in seeing how the system performs as a whole, i.e. we wish to measure the time taken from initial page load until the compositor sits idle.

Our strategy is simple: We instrumented Servo to initiate a shutdown if the script task sat idle for a long enough time. By trial and error we estimated the average page load to be under three seconds, so this is the cap we chose. At start-up the current time is stored and on shutdown, the time taken from start-up until shutdown is reported. For each site selected from the collection of static sites, we ran Servo with the given site 100 times and collected the output duration. The results are displayed in Table 2.

First, we should note that the times reported measures how long it took before the script task had been sitting idle for three seconds. The measurements also include the overhead of actually starting and stopping the Servo process. Therefore, the times reported are not indicative of how long the user will actually perceive that the load took. For example, the

[11] `https://github.com/japaric/criterion.rs`

	master		session-types		
Website	μ	σ	μ	σ	Diff.
Reddit	4492.27	96.93	4482.11	48.54	−10.16
Reddit (no JS)	4208.98	11.86	4209.51	11.60	0.53
Ars Technica	3080.41	11.48	3077.09	9.95	−3.32
Wikipedia	5127.47	10.55	5117.18	7.03	−10.29
YouTube	3154.75	14.19	3150.66	12.55	−4.09

Table 2: Duration of time from start-up until shutdown (including idle time) with standard deviation and difference of mean values. The difference is given as $\mu_{\texttt{session-types}} - \mu_{\texttt{master}}$. All numbers are in milliseconds (ms).

Wikipedia page took over five seconds to load, on average, which, even after subtracting the three second timeout is not indicative of the general user experience, but must instead be a result of a background script firing events after rendering is complete. Indeed, anyone opening a locally stored Wikipedia homepage in Servo, with the original version or ours, will see that it loads almost instantly. However, we are comparing two branches of Servo where the only difference is whether or not session-typed channels are used, so we can still draw some useful observations from the test.

Overall we do not see big differences in the reported mean values—less than 1% in all cases. We note that except for the Reddit (no JS) site, our version with session types is a little faster, but it is all within the standard deviation and there too many potential sources for noise in these measurements to attribute this to our changes. This data indicates that session-typed communication does not impose much, if any, performance overhead in a large application such as Servo.

8.2 Related Work

OTP Supervisors The open telecom platform (OTP) is a framework for building massively scalable soft real-time systems. It comprises the Erlang programming language, an application server, a number of auxiliary tools and several libraries [31].

One of the design principles in OTP, is the supervisor behaviour in which there are worker processes and supervisor processes. In case of a worker crash, its supervisor can try to restart it according to a prespecified strategy. Supervisors are organised in a tree structure to establish a hierarchy [32].

We considered implementing supervisor trees in Rust and using them in Servo, but decided against it for a number of reasons. Supervisor trees in Servo would be small, because Servo consists of an array of independent processes that all intercommunicate. To handle these processes, one supervisor could be implemented to monitor all of these processes, but this still does not prevent one process from crashing because of miscommunication from another process.

Supervisors are aimed at handling the events where processes fail, but we are interested in preventing these failures. Ensuring communication consistency is not the domain of supervisor trees.

The Join Calculus The join calculus is a process calculus developed for distributed programming, and for this reason the core calculus only provides asynchronous communication mechanisms. The core idea is to decouple transmission from synchronization to allow synchronization issues to be resolved locally [33]. Synchronous communication mechanisms, such as rendez-vous points, can be implemented via the asynchronous mechanisms.

The join calculus introduces *join patterns* that facilites matching against messages on multiple channels simultaneously (in effect using pattern matching as a synchronization mechanism), and the construct is powerful enough to encode standard concurrency primitives such as the actor model, synchronous message passing and rendez-vous. A common idiom in using join patterns is to encode state of a concurrent object in private messages, and join patterns can be used to implement data structures such as shared variables and bounded buffers.

Encoding the state of an object or process in message passing has a certain appeal (we have mentioned how session types can be thought of as encoding the state of a process). This approach greatly simplifies the implementation of common concurrency communication mechanisms. The following example code implements a read-write lock:

```
let rwlock () =
  def shared() & idle()       = reply to shared & s(1)
   or shared() & s(n)         = reply to shared & s(n+1)
   or release_shared() & s(n) = if n = 1 then idle() else s(n-1)
   or exclusive() & idle()    = reply to exclusive
   or release_exclusive()     = idle()
  in spawn idle();
  shared, release_shared, exclusive, release_exclusive
;;
```

The `def-or` is a *join-definition*, and each line is a reaction rule that comprises a join pattern and a process body. A join pattern is a list of channel names with formal arguments (for example "`shared() & idle()`" above). The process body of a reaction rule is executed only when there are messages present on all channels in the join pattern. For example, sending a message on `shared` can match either the first or second reaction rule of `rwlock`. If there is a message on `idle`, the process body of `shared() & idle()` is executed. The `reply to shared` makes calls to `shared` synchronous, i.e. a call to `shared` blocks the process until one of the two join patterns are matched. The channel names `idle` and `s` encode the internal state (they are not made available to the user). On initialization a message is ready on `idle`, and this enables either the rule `shared() & idle()` or `exclusive() & idle()`. The `s` channel carries as a message the number of readers currently holding a shared lock, and calls to `release_shared` match with `s(n)` and either decrements the number of holders or calls `idle`.

A number of implementations of the join calculus have been made, primarily as extensions to existing programming languages, notably JoCaml and Polyphonic C♯ [34, 35], and an elegant solution to the Santa Claus problem discussed in Section 5.4.4 has been implemented in Polyphonic C♯ [36].

The Rust approach to concurrency is to specify the requirements for data to be sent and shared (through the `Send` and `Sync` traits), and let the user pick his or her concurrency model. For this reason, it would be an interesting project to see if the join calculus could be embedded in Rust, and furthermore see how its declarative approach to synchronization could improve the communication schemes in Servo.

8.3 Future Work

8.3.1 Improving the Type System

The type system of `session-types` is not complete in the sense that two programs that express the same patterns of interactions (i.e. their interaction traces are equal) do not necessarily have the same type. In this respect we diverge from the session types theory that defines an equivalence relation on types, but we do not support this notion of equivalence. This can be observed in the cases of branching and recursion.

We have discussed these limitations in Section 5.2, but have not attempted to provide a solution here, because we did not feel it was within the scope of this dissertation.

8.3.2 Improvements in `session-types`

Based on our discussion in Section 8.1.1 we are confident that we can improve the performance of `session-types` by creating our own message passing mechanism instead of boxing and unboxing values. Ideally, we could switch on the size of a value to decide whether or not to box it. From our experience with Servo it does not appear common to transmit large values, so removing boxing for small values would quite likely be an improvement.

Furthermore, we could probably improve the memory allocation strategy by inferring the size of the largest value that can be transmitted in a given protocol. The Rust standard library provides a `size_of` function that at compile-time determines the size of an indicated type. We could then exploit the tree-like structure of the session types DSL to infer the size of the biggest type in the protocol (this obviously does not promise that a value of this type will be transmitted, only that it could be). It would then suffice to allocate memory slots of that size for any channel with that protocol.

8.3.3 Rewriting Servo

We have only rewritten a fraction of Servo and we need to carry on the work of porting the process communication to session-typed channels. We are collaborating with Lars Bergstrom of Mozilla Research to land our changes in the Servo mainline.

As we have demonstrated in this dissertation, replacing the communication schemes requires manual work, and it requires a substantial amount of research to understand and translate the implicit protocols already present. In many cases, no direct translation is possible, simply because a certain communication scheme may not be feasible using session types (like repeatedly cloning `Sender` values), and the developer will have to come up with alternative schemes. Other cases may have several reasonable translations depending on how the original is expected to work, and the developer will have to decide which one is correct.

8.3.4 Improving `humpty_dumpty`

Of the three issues described in Section 7.2: closures, generic functions, and external crates, the latter is probably the hardest to fix. Essentially, `humpty_dumpty` is limited by what a plugin can do: it cannot access function definitions in external crates. Possible solutions include banning (or at least warning on) all calls to functions in external crates that moves protected values, except, perhaps, for a list of manually verified functions. For instance, it might be useful to be able to put `Foo`s in a vector (`Vec`, defined in `std::vec`) with `push`, and take them out again using `pull`. Another solution is to allow calls to external functions, but track all values returned from them, regardless of their types, and make sure that they are dropped in some specific manner. Neither solves the problem in general, but it might be worth experimenting with both solutions to see which one works best for our purposes.

In contrast, handling closures and generic functions is not a insurmountable problem, but it will still require substantial effort to do right. We need to do additional tracking of functional values as well as custom traversals of function definitions with type parameters substituted.

8.4 Conclusion

We have demonstrated that session types can be implemented directly in a programming language with affine types, and we provide an implementation of session types as a Rust library, `session-types`, along with an informal argument for type safety. Emphasizing usability in real-world applications has led us to expand the core session types primitives with the macros `chan_select!` and `offer!`.

We have ported parts of Servo's internal communication patterns from simple message-passing channels to session-typed channels, and demonstrated that it is feasible to introduce session types in a concurrent, real-world application to prevent errors arising from miscommunication. Through benchmarks we have shown that the use of session types in Servo carries negligible performance overhead.

We have explored the possibility of enforcing linearity of types in Rust, and developed the `humpty_dumpty` compiler plugin as a result. Although we can solve a host of cases, the limitations of the Rust compiler's plugin infrastructure do not allow us to solve the general case.

Finally, we have submitted a paper about our work to the 11th ACM SIGPLAN Workshop on Generic Programming (WGP 2015) that is currently in review.

While working on this project, we have actively participated in both the Rust and Servo developer communities by submitting bug reports, feature requests and minor fixes. To test `humpty_dumpty`, we extracted a tool from the Rust compiler, `compiletest`, into a stand-alone crate that we published on `crates.io`. To our surprise, the `compiletest` utility has turned out be a popular tool among other developers, with over 400 downloads at the time of writing.

We hope to be able to continue to develop our `session-types` library, because there are still many interesting venues for improvement. For example, we would like to formalize and prove additional properties about our session type system. Other areas include performance improvements and extending the capabilities of `humpty_dumpty`.

With our work, we believe to have shown that session types can be used to specify and verify the communication patterns of concurrent processes, without sacrificing too much expressiveness or performance. We hope that our work will help pave the way for a wider adoption of session types in production-level software.

Acknowledgments

We would like to thank Lars Bergstrom of Mozilla and Manish Goregaokar of IIT Bombay for their help and support. Lars has been enormously helpful in understanding the communication patterns in Servo and the problems that Servo had. He has been very supportive from the beginnning of this project, and there would probably not have been a project without him. Manish has been immensely helpful in understanding many of the inner workings of the Rust compiler, and has contributed with many suggestions for improving our work on `session-types`. Additionally, his help has been crucial for our work with Rust's compiler plugins: `humpty_dumpty` would not be anywhere near where it is now, if not for him. Their support has been greatly appreciated.

We would also like to thank our supervisor, Ken Friis Larsen, who has been very encouraging through the whole process, and we are grateful for having the opportunity to submit a joint paper about our work to the WGP 2015 with him.

Finally, we would like to extend our thanks to the people of the `#rust-internals` and `#servo` channels on `irc.mozilla.org`, who have always been helpful in answering our questions, the great community surrounding Rust and Servo for supporting this language and the browser engine of tomorrow, as well as everyone else who have, directly or indirectly, helped us.

References

[1] John A. Trono. A new exercise in concurrency. *SIGCSE Bulletin*, 1994.

[2] Simon Gay and Malcolm Hole. Types and subtypes for client-server interactions. In S.Doaitse Swierstra, editor, *Programming Languages and Systems*, volume 1576 of *Lecture Notes in Computer Science*, pages 74–90. Springer Berlin Heidelberg, 1999.

[3] The Microsoft Internet Explorer web browser. `https://en.wikipedia.org/w/index.php?title=Internet_Explorer&oldid=651130028`.

[4] Mozilla Firefox. `https://en.wikipedia.org/w/index.php?title=Firefox&oldid=651485968`.

[5] Safari (web browser). `https://en.wikipedia.org/w/index.php?title=Safari_%28web_browser%29&oldid=651567276`.

[6] Opera (web browser). `https://en.wikipedia.org/w/index.php?title=Opera_%28web_browser%29&oldid=651331466`.

[7] Google Chrome. `https://en.wikipedia.org/w/index.php?title=Google_Chrome&oldid=651567171`.

[8] Timeline of web browsers. `https://en.wikipedia.org/w/index.php?title=Timeline_of_web_browsers&oldid=650932481`.

[9] Browser History Timeline. `http://meyerweb.com/eric/browsers/timeline-structured.html`.

[10] The Rust Reference. `http://doc.rust-lang.org/reference.html`.

[11] Eric Reed. Patina: A formalization of the rust programming language. `ftp://ftp.cs.washington.edu/tr/2015/03/UW-CSE-15-03-02.pdf`, February 2015.

[12] Mads Tofte and Jean-Pierre Talpin. Implementation of the typed call-by-value λ-calculus using a stack of regions. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '94, pages 188–201, New York, NY, USA, 1994. ACM.

[13] Mads Tofte, Lars Birkedal, Martin Elsman, Niels Hallenberg, Tommy Højfeld Olesen, and Peter Sestoft. Programming with regions in the MLKit (revised for version 4.3.0). Technical report, IT University of Copenhagen, Denmark, January 2006.

[14] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based memory management in Cyclone. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, pages 282–293, New York, NY, USA, 2002. ACM.

[15] The Rust programming language. `http://doc.rust-lang.org/book/`.

[16] Kaku Takeuchi, Kohei Honda, and Makoto Kubo. An Interaction-based Language and its Typing System. In *PARLE '94: Proceedings of the 6th International PARLE Conference on Parallel Architectures and Languages Europe*, pages 398–413, London, UK, 1994. Springer-Verlag.

[17] Simon Gay and Malcolm Hole. Subtyping for session types in the pi calculus. *Acta Informatica*, 42(2-3):191–225, 2005.

[18] Riccardo Pucella and Jesse A. Tov. Haskell session types with (almost) no class. *SIGPLAN Not.*, 44(2):25–36, September 2008.

[19] Raymond Hu, Nobuko Yoshida, and Kohei Honda. Session-based distributed programming in java. In Jan Vitek, editor, *ECOOP 2008 – Object-Oriented Programming*, volume 5142 of *Lecture Notes in Computer Science*, pages 516–541. Springer Berlin Heidelberg, 2008.

[20] Nobuko Yoshida and Vasco T. Vasconcelos. Language primitives and type discipline for structured communication-based programming revisited: Two systems for higher-order session communication. *Electronic Notes in Theoretical Computer Science*, 171(4):73 – 93, 2007. Proceedings of the First International Workshop on Security and Rewriting Techniques (SecReT 2006).

[21] Matthias Neubauer and Peter Thiemann. An implementation of session types. In *Practical Aspects of Declarative Languages*, pages 56–70. Springer, 2004.

[22] Matthew Sackman and Susan Eisenbach. Session types in haskell. 2008.

[23] Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In *Proceedings of the 7th European Symposium on Programming: Programming Languages and Systems*, ESOP '98, pages 122–138, London, UK, UK, 1998. Springer-Verlag.

[24] Simon J Gay and Vasco T Vasconcelos. Linear type theory for asynchronous session types. *Journal of Functional Programming*, 20(01):19–50, 2010.

[25] Ivan E Sutherland and Gary W Hodgman. Reentrant polygon clipping. *Communications of the ACM*, 17(1):32–42, 1974.

[26] C. A. R. Hoare. Communicating sequential processes, 1985.

[27] Occam manual. `http://www.eg.bucknell.edu/~cs366/occam.pdf`.

[28] Alef language reference manual. `https://swtch.com/~rsc/thread/alef.pdf`.

[29] The Go programming language language specification. `http://golang.org/ref/spec`.

[30] B. Anderson, L. Bergstrom, D. Herman, J. Matthews, K. McAllister, M. Goregaokar, J. Moffitt, and S. Sapin. Experience Report: Developing the Servo Web Browser Engine using Rust. *ArXiv e-prints*, May 2015.

[31] Seved Torstendahl. Open telecom platform.

[32] Who supervises the supervisors? learn you some erlang for great good! `http://learnyousomeerlang.com/supervisors#from-bad-to-good`.

[33] Cédric Fournet and Georges Gonthier. The join calculus: a language for distributed mobile programming. In *Applied Semantics*, pages 268–332. Springer, 2002.

[34] Sylvain Conchon and Fabrice Le Fessant. Jocaml: Mobile agents for objective-caml. In *Agent systems and applications, 1999 and third international symposium on mobile agents. Proceedings. First international symposium on*, pages 22–29. IEEE, 1999.

[35] Nick Benton, Luca Cardelli, and Cédric Fournet. Modern Concurrency Abstractions for C♯, 2002.

[36] Nick Benton. Jingle bells: Solving the Santa Claus Problem in Polyphonic C♯. *Unpublished manuscript, Mar*, 2003.

A The **session-types** Library

```
//! session_types
//!
//! This is an implementation of *session types* in Rust.
//!
//! The channels in Rusts standard library are useful for a great many things,
//! but they're restricted to a single type. Session types allows one to use a
//! single channel for transferring values of different types, depending on the
//! context in which it is used. Specifically, a session typed channel always
//! carry a *protocol*, which dictates how communication is to take place.
//!
//! For example, imagine that two threads, `A` and `B` want to communicate with
//! the following pattern:
//!
//!  1. `A` sends an integer to `B`.
//!  2. `B` sends a boolean to `A` depending on the integer received.
//!
//! With session types, this could be done by sharing a single channel. From
//! `A`'s point of view, it would have the type `int ! (bool ? eps)` where `t ! r`
//! is the protocol "send something of type `t` then proceed with
//! protocol `r`", the protocol `t ? r` is "receive something of type `t` then proceed
//! with protocol `r`, and `eps` is a special marker indicating the end of a
//! communication session.
//!
//! Our session type library allows the user to create channels that adhere to a
//! specified protocol. For example, a channel like the above would have the type
//! `Chan<(), Send<i64, Recv<bool, Eps>>>`, and the full program could look like this:
//!
//! ```
//! extern crate session_types;
//! use session_types::*;
//!
//! type Server = Recv<i64, Send<bool, Eps>>;
//! type Client = Send<i64, Recv<bool, Eps>>;
//!
//! fn srv(c: Chan<(), Server>) {
//!     let (c, n) = c.recv();
//!     if n % 2 == 0 {
//!         c.send(true).close()
//!     } else {
//!         c.send(false).close()
//!     }
//! }
//!
//! fn cli(c: Chan<(), Client>) {
//!     let n = 42;
//!     let c = c.send(n);
//!     let (c, b) = c.recv();
//!
//!     if b {
//!         println!("{} is even", n);
//!     } else {
//!         println!("{} is odd", n);
//!     }
//!
//!     c.close();
//! }
```

```
57  //!
58  //! fn main() {
59  //!     connect(srv, cli);
60  //! }
61  //! ```
62
63  #![feature(std_misc)]
64
65  use std::marker;
66  use std::thread::spawn;
67  use std::mem::transmute;
68  use std::sync::mpsc::{Sender, Receiver, channel, Select};
69  use std::collections::HashMap;
70  use std::marker::PhantomData;
71
72  /// A session typed channel. `P` is the protocol and `E` is the environment,
73  /// containing potential recursion targets
74  #[must_use]
75  pub struct Chan<E, P> (Sender<Box<u8>>, Receiver<Box<u8>>, PhantomData<(E, P)>);
76
77  fn unsafe_write_chan<A: marker::Send + 'static, E, P>
78      (&Chan(ref tx, _, _): &Chan<E, P>, x: A)
79  {
80      let tx: &Sender<Box<A>> = unsafe { transmute(tx) };
81      tx.send(Box::new(x)).unwrap();
82  }
83
84  fn unsafe_read_chan<A: marker::Send + 'static, E, P>
85      (&Chan(_, ref rx, _): &Chan<E, P>) -> A
86  {
87      let rx: &Receiver<Box<A>> = unsafe { transmute(rx) };
88      *rx.recv().unwrap()
89  }
90
91  /// Peano numbers: Zero
92  #[allow(missing_copy_implementations)]
93  pub struct Z;
94
95  /// Peano numbers: Increment
96  pub struct S<N> ( PhantomData<N> );
97
98  /// End of communication session (epsilon)
99  #[allow(missing_copy_implementations)]
100 pub struct Eps;
101
102 /// Receive `A`, then `P`
103 pub struct Recv<A, P> ( PhantomData<(A, P)> );
104
105 /// Send `A`, then `P`
106 pub struct Send<A, P> ( PhantomData<(A, P)> );
107
108 /// Active choice between `P` and `Q`
109 pub struct Choose<P, Q> ( PhantomData<(P, Q)> );
110
111 /// Passive choice (offer) between `P` and `Q`
112 pub struct Offer<P, Q> ( PhantomData<(P, Q)> );
113
114 /// Enter a recursive environment
```

```rust
pub struct Rec<P> ( PhantomData<P> );

/// Recurse. N indicates how many layers of the recursive environment we recurse
/// out of.
pub struct Var<N> ( PhantomData<N> );

pub unsafe trait HasDual {
    type Dual;
}

unsafe impl HasDual for Eps {
    type Dual = Eps;
}

unsafe impl <A, P: HasDual> HasDual for Send<A, P> {
    type Dual = Recv<A, P::Dual>;
}

unsafe impl <A, P: HasDual> HasDual for Recv<A, P> {
    type Dual = Send<A, P::Dual>;
}

unsafe impl <P: HasDual, Q: HasDual> HasDual for Choose<P, Q> {
    type Dual = Offer<P::Dual, Q::Dual>;
}

unsafe impl <P: HasDual, Q: HasDual> HasDual for Offer<P, Q> {
    type Dual = Choose<P::Dual, Q::Dual>;
}

unsafe impl HasDual for Var<Z> {
    type Dual = Var<Z>;
}

unsafe impl <N> HasDual for Var<S<N>> {
    type Dual = Var<S<N>>;
}

unsafe impl <P: HasDual> HasDual for Rec<P> {
    type Dual = Rec<P::Dual>;
}

impl<E> Chan<E, Eps> {
    /// Close a channel. Should always be used at the end of your program.
    pub fn close(self) {
        // Consume `c`
    }
}

impl<E, P, A: marker::Send + 'static> Chan<E, Send<A, P>> {
    /// Send a value of type `A` over the channel. Returns a channel with
    /// protocol `P`
    #[must_use]
    pub fn send(self, v: A) -> Chan<E, P> {
        unsafe_write_chan(&self, v);
        unsafe { transmute(self) }
    }
}
```

```rust

impl<E, P, A: marker::Send + 'static> Chan<E, Recv<A, P>> {
    /// Receives a value of type `A` from the channel. Returns a tuple
    /// containing the resulting channel and the received value.
    #[must_use]
    pub fn recv(self) -> (Chan<E, P>, A) {
        let v = unsafe_read_chan(&self);
        (unsafe { transmute(self) }, v)
    }
}

impl<E, P, Q> Chan<E, Choose<P, Q>> {
    /// Perform an active choice, selecting protocol `P`.
    #[must_use]
    pub fn sel1(self) -> Chan<E, P> {
        unsafe_write_chan(&self, true);
        unsafe { transmute(self) }
    }

    /// Perform an active choice, selecting protocol `Q`.
    #[must_use]
    pub fn sel2(self) -> Chan<E, Q> {
        unsafe_write_chan(&self, false);
        unsafe { transmute(self) }
    }
}

/// Convenience function. This is identical to `.sel2()`
impl<Z, A, B> Chan<Z, Choose<A, B>> {
    #[must_use]
    pub fn skip(self) -> Chan<Z, B> {
        self.sel2()
    }
}

/// Convenience function. This is identical to `.sel2().sel2()`
impl<Z, A, B, C> Chan<Z, Choose<A, Choose<B, C>>> {
    #[must_use]
    pub fn skip2(self) -> Chan<Z, C> {
        self.sel2().sel2()
    }
}

/// Convenience function. This is identical to `.sel2().sel2().sel2()`
impl<Z, A, B, C, D> Chan<Z, Choose<A, Choose<B, Choose<C, D>>>> {
    #[must_use]
    pub fn skip3(self) -> Chan<Z, D> {
        self.sel2().sel2().sel2()
    }
}

/// Convenience function. This is identical to `.sel2().sel2().sel2().sel2()`
impl<Z, A, B, C, D, E> Chan<Z, Choose<A, Choose<B, Choose<C, Choose<D, E>>>>> {
    #[must_use]
    pub fn skip4(self) -> Chan<Z, E> {
        self.sel2().sel2().sel2().sel2()
    }
}
```

```
231
232 /// Convenience function. This is identical to `.sel2().sel2().sel2().sel2().sel2()`
233 impl<Z, A, B, C, D, E, F> Chan<Z, Choose<A, Choose<B, Choose<C, Choose<D,
234                           Choose<E, F>>>>>> {
235     #[must_use]
236     pub fn skip5(self) -> Chan<Z, F> {
237         self.sel2().sel2().sel2().sel2().sel2()
238     }
239 }
240
241 /// Convenience function.
242 impl<Z, A, B, C, D, E, F, G> Chan<Z, Choose<A, Choose<B, Choose<C, Choose<D,
243                              Choose<E, Choose<F, G>>>>>>> {
244     #[must_use]
245     pub fn skip6(self) -> Chan<Z, G> {
246         self.sel2().sel2().sel2().sel2().sel2().sel2()
247     }
248 }
249
250 /// Convenience function.
251 impl<Z, A, B, C, D, E, F, G, H> Chan<Z, Choose<A, Choose<B, Choose<C, Choose<D,
252                                        Choose<E, Choose<F, Choose<G, H>>>>>>>> {
253     #[must_use]
254     pub fn skip7(self) -> Chan<Z, H> {
255         self.sel2().sel2().sel2().sel2().sel2().sel2().sel2()
256     }
257 }
258
259 impl<E, P, Q> Chan<E, Offer<P, Q>> {
260     /// Passive choice. This allows the other end of the channel to select one
261     /// of two options for continuing the protocol: either `P` or `Q`.
262     #[must_use]
263     pub fn offer(self) -> Result<Chan<E, P>, Chan<E, Q>> {
264         let b = unsafe_read_chan(&self);
265         if b {
266             Ok(unsafe { transmute(self) })
267         } else {
268             Err(unsafe { transmute(self) })
269         }
270     }
271 }
272
273 impl<E, P> Chan<E, Rec<P>> {
274     /// Enter a recursive environment, putting the current environment on the
275     /// top of the environment stack.
276     #[must_use]
277     pub fn enter(self) -> Chan<(P, E), P> {
278         unsafe { transmute(self) }
279     }
280 }
281
282 impl<E, P> Chan<(P, E), Var<Z>> {
283     /// Recurse to the environment on the top of the environment stack.
284     #[must_use]
285     pub fn zero(self) -> Chan<(P, E), P> {
286         unsafe { transmute(self) }
287     }
288 }
```

```
289
290 impl<E, P, N> Chan<(P, E), Var<S<N>>> {
291     /// Pop the top environment from the environment stack.
292     #[must_use]
293     pub fn succ(self) -> Chan<E, Var<N>> {
294         unsafe { transmute(self) }
295     }
296 }
297
298 /// Homogeneous select. We have a vector of channels, all obeying the same
299 /// protocol (and in the exact same point of the protocol), wait for one of them
300 /// to receive. Removes the receiving channel from the vector and returns both
301 /// the channel and the new vector.
302 #[must_use]
303 pub fn hselect<E, P, A>(mut chans: Vec<Chan<E, Recv<A, P>>>)
304                         -> (Chan<E, Recv<A, P>>, Vec<Chan<E, Recv<A, P>>>)
305 {
306     let i = iselect(&chans);
307     let c = chans.remove(i);
308     (c, chans)
309 }
310
311 /// An alternative version of homogeneous select, returning the index of the Chan
312 /// that is ready to receive.
313 pub fn iselect<E, P, A>(chans: &Vec<Chan<E, Recv<A, P>>>) -> usize {
314     let mut map = HashMap::new();
315
316     let id = {
317         let sel = Select::new();
318         let mut handles = Vec::with_capacity(chans.len()); // collect all the handles
319
320         for (i, chan) in chans.iter().enumerate() {
321             let &Chan(_, ref rx, _) = chan;
322             let handle = sel.handle(rx);
323             map.insert(handle.id(), i);
324             handles.push(handle);
325         }
326
327         for handle in handles.iter_mut() { // Add
328             unsafe { handle.add(); }
329         }
330
331         let id = sel.wait();
332
333         for handle in handles.iter_mut() { // Clean up
334             unsafe { handle.remove(); }
335         }
336
337         id
338     };
339     map.remove(&id).unwrap()
340 }
341
342 /// Heterogeneous selection structure for channels
343 ///
344 /// This builds a structure of channels that we wish to select over. This is
345 /// structured in a way such that the channels selected over cannot be
346 /// interacted with (consumed) as long as the borrowing ChanSelect object
```

```
347 /// exists. This is necessary to ensure memory safety.
348 ///
349 /// The type parameter T is a return type, ie we store a value of some type T
350 /// that is returned in case its associated channels is selected on `wait()`
351 pub struct ChanSelect<'c, T> {
352     chans: Vec<(&'c Chan<(), ()>, T)>,
353 }
354
355
356 impl<'c, T> ChanSelect<'c, T> {
357     pub fn new() -> ChanSelect<'c, T> {
358         ChanSelect {
359             chans: Vec::new()
360         }
361     }
362
363     /// Add a channel whose next step is `Recv`
364     ///
365     /// Once a channel has been added it cannot be interacted with as long as it
366     /// is borrowed here (by virtue of borrow checking and lifetimes).
367     pub fn add_recv_ret<E, P, A: marker::Send>(&mut self,
368                                                chan: &'c Chan<E, Recv<A, P>>,
369                                                ret: T)
370     {
371         self.chans.push((unsafe { transmute(chan) }, ret));
372     }
373
374     pub fn add_offer_ret<E, P, Q>(&mut self,
375                                   chan: &'c Chan<E, Offer<P, Q>>,
376                                   ret: T)
377     {
378         self.chans.push((unsafe { transmute(chan) }, ret));
379     }
380
381     /// Find a Receiver (and hence a Chan) that is ready to receive.
382     ///
383     /// This method consumes the ChanSelect, freeing up the borrowed Receivers
384     /// to be consumed.
385     pub fn wait(self) -> T {
386         let sel = Select::new();
387         let mut handles = Vec::with_capacity(self.chans.len());
388         let mut map = HashMap::new();
389
390         for (chan, ret) in self.chans.into_iter() {
391             let &Chan(_, ref rx, _) = chan;
392             let h = sel.handle(rx);
393             let id = h.id();
394             map.insert(id, ret);
395             handles.push(h);
396         }
397
398         for handle in handles.iter_mut() {
399             unsafe { handle.add(); }
400         }
401
402         let id = sel.wait();
403
404         for handle in handles.iter_mut() {
```

```
405             unsafe { handle.remove(); }
406         }
407         map.remove(&id).unwrap()
408     }
409
410     /// How many channels are there in the structure?
411     pub fn len(&self) -> usize {
412         self.chans.len()
413     }
414 }
415
416 /// Default use of ChanSelect works with usize and returns the index
417 /// of the selected channel. This is also the implementation used by
418 /// the `chan_select!` macro.
419 impl<'c> ChanSelect<'c, usize> {
420     pub fn add_recv<E, P, A: marker::Send>(&mut self,
421                                            c: &'c Chan<E, Recv<A, P>>)
422     {
423         let index = self.chans.len();
424         self.add_recv_ret(c, index);
425     }
426
427     pub fn add_offer<E, P, Q>(&mut self,
428                               c: &'c Chan<E, Offer<P, Q>>)
429     {
430         let index = self.chans.len();
431         self.add_offer_ret(c, index);
432     }
433 }
434
435 /// Sets up an session typed communication channel. Should be paired with
436 /// `request` for the corresponding client.
437 #[must_use]
438 pub fn accept<P: HasDual>(tx: Sender<Chan<(), P::Dual>>) -> Option<Chan<(), P>> {
439     borrow_accept(&tx)
440 }
441
442 #[must_use]
443 pub fn borrow_accept<P: HasDual>(tx: &Sender<Chan<(), P::Dual>>)
444                                  -> Option<Chan<(), P>> {
445     let (c2, c1) = session_channel();
446
447     match tx.send(c1) {
448         Ok(_) => Some(c2),
449         _ => None
450     }
451 }
452
453 /// Sets up an session typed communication channel. Should be paired with
454 /// `accept` for the corresponding server.
455 #[must_use]
456 pub fn request<P: HasDual>(rx: Receiver<Chan<(), P>>) -> Option<Chan<(), P>> {
457     borrow_request(&rx)
458 }
459
460 #[must_use]
461 pub fn borrow_request<P: HasDual>(rx: &Receiver<Chan<(), P>>) -> Option<Chan<(), P>> {
462     match rx.recv() {
```

```
463         Ok(c) => Some(c),
464         _ => None
465     }
466 }
467 
468 /// Returns two session channels
469 #[must_use]
470 pub fn session_channel<P: HasDual>() -> (Chan<(), P>, Chan<(), P::Dual>) {
471     let (tx1, rx1) = channel();
472     let (tx2, rx2) = channel();
473 
474     let c1 = Chan(tx1, rx2, PhantomData);
475     let c2 = Chan(tx2, rx1, PhantomData);
476 
477     (c1, c2)
478 }
479 
480 /// Connect two functions using a session typed channel.
481 pub fn connect<F1, F2, P>(srv: F1, cli: F2)
482     where F1: Fn(Chan<(), P>) + marker::Send + 'static,
483           F2: Fn(Chan<(), P::Dual>) + marker::Send,
484           P: HasDual + marker::Send + 'static,
485           <P as HasDual>::Dual: HasDual + marker::Send + 'static
486 {
487     let (tx, rx) = channel();
488     let t = spawn(move || srv(accept::<P>(tx).unwrap()));
489     cli(request::<P::Dual>(rx).unwrap());
490     t.join().unwrap();
491 }
492 
493 /// This macro is convenient for server-like protocols of the form:
494 ///
495 /// `Offer<A, Offer<B, Offer<C, ... >>>`
496 ///
497 /// # Examples
498 ///
499 /// Assume we have a protocol `Offer<Recv<u64, Eps>, Offer<Recv<String, Eps>,Eps>>>`
500 /// we can use the `offer!` macro as follows:
501 ///
502 /// ```rust
503 /// #[macro_use] extern crate session_types;
504 /// use session_types::*;
505 /// use std::thread::spawn;
506 ///
507 /// fn srv(c: Chan<(), Offer<Recv<u64, Eps>, Offer<Recv<String, Eps>, Eps>>>) {
508 ///     offer! { c,
509 ///         Number => {
510 ///             let (c, n) = c.recv();
511 ///             assert_eq!(42, n);
512 ///             c.close();
513 ///         },
514 ///         String => {
515 ///             c.recv().0.close();
516 ///         },
517 ///         Quit => {
518 ///             c.close();
519 ///         }
520 ///     }
```

```
521 /// }
522 ///
523 /// fn cli(c: Chan<(), Choose<Send<u64, Eps>, Choose<Send<String, Eps>, Eps>>>) {
524 ///     c.sel1().send(42).close();
525 /// }
526 ///
527 /// fn main() {
528 ///     let (s, c) = session_channel();
529 ///     spawn(move|| cli(c));
530 ///     srv(s);
531 /// }
532 /// ```
533 ///
534 /// The identifiers on the left-hand side of the arrows have no semantic
535 /// meaning, they only provide a meaningful name for the reader.
536 #[macro_export]
537 macro_rules! offer {
538     (
539         $id:ident, $branch:ident => $code:expr, $($t:tt)+
540     ) => (
541         match $id.offer() {
542             Ok($id) => $code,
543             Err($id) => offer!{ $id, $($t)+ }
544         }
545     );
546     (
547         $id:ident, $branch:ident => $code:expr
548     ) => (
549         $code
550     )
551 }
552
553 /// This macro plays the same role as the `select!` macro does for `Receiver`s.
554 ///
555 /// It also supports a second form with `Offer`s (see the example below).
556 ///
557 /// # Examples
558 ///
559 /// ```rust
560 /// #[macro_use] extern crate session_types;
561 /// use session_types::*;
562 /// use std::thread::spawn;
563 ///
564 /// fn send_str(c: Chan<(), Send<String, Eps>>) {
565 ///     c.send("Hello, World!".to_string()).close();
566 /// }
567 ///
568 /// fn send_usize(c: Chan<(), Send<usize, Eps>>) {
569 ///     c.send(42).close();
570 /// }
571 ///
572 /// fn main() {
573 ///     let (tcs, rcs) = session_channel();
574 ///     let (tcu, rcu) = session_channel();
575 ///
576 ///     // Spawn threads
577 ///     spawn(move|| send_str(tcs));
578 ///     spawn(move|| send_usize(tcu));
```

```
///
///     loop {
///         chan_select! {
///             (c, s) = rcs.recv() => {
///                 assert_eq!("Hello, World!".to_string(), s);
///                 c.close();
///                 break
///             },
///             (c, i) = rcu.recv() => {
///                 assert_eq!(42, i);
///                 c.close();
///                 break
///             }
///         }
///     }
/// }
/// ```
///
/// ```rust
/// #![feature(rand)]
/// #[macro_use]
/// extern crate session_types;
/// extern crate rand;
///
/// use std::thread::spawn;
/// use session_types::*;
///
/// type Igo = Choose<Send<String, Eps>, Send<u64, Eps>>;
/// type Ugo = Offer<Recv<String, Eps>, Recv<u64, Eps>>;
///
/// fn srv(chan_one: Chan<(), Ugo>, chan_two: Chan<(), Ugo>) {
///     let _ign;
///     chan_select! {
///         _ign = chan_one.offer() => {
///             String => {
///                 let (c, s) = chan_one.recv();
///                 assert_eq!("Hello, World!".to_string(), s);
///                 c.close();
///             },
///             Number => {
///                 unreachable!()
///             }
///         },
///         _ign = chan_two.offer() => {
///             String => {
///                 unreachable!()
///             },
///             Number => {
///                 unreachable!()
///             }
///         }
///     }
/// }
///
/// fn cli(c: Chan<(), Igo>) {
///     c.sel1().send("Hello, World!".to_string()).close();
/// }
///
```

```
637 /// fn main() {
638 ///     let (ca1, ca2) = session_channel();
639 ///     let (cb1, cb2) = session_channel();
640 ///
641 ///     spawn(move|| cli(ca2));
642 ///
643 ///     srv(ca1, cb1);
644 /// }
645 /// ```
646
647
648 #[macro_export]
649 macro_rules! chan_select {
650     (
651         $(($c:ident, $name:pat) = $rx:ident.recv() => $code:expr),+
652     ) => ({
653         let index = {
654             let mut sel = $crate::ChanSelect::new();
655             $( sel.add_recv(&$rx); )+
656             sel.wait()
657         };
658         let mut i = 0;
659         $( if index == { i += 1; i - 1 } { let ($c, $name) = $rx.recv(); $code }
660            else )+
661         { unreachable!() }
662     });
663
664     (
665         $($res:ident = $rx:ident.offer() => { $($t:tt)+ }),+
666     ) => ({
667         let index = {
668             let mut sel = $crate::ChanSelect::new();
669             $( sel.add_offer(&$rx); )+
670             sel.wait()
671         };
672         let mut i = 0;
673         $( if index == { i += 1; i - 1 } { $res = offer!{ $rx, $($t)+ } } else )+
674         { unreachable!() }
675     })
676 }
```