Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Master's Thesis Nr. 146

Systems Group, Department of Computer Science, ETH Zurich

OS Development in Rust

by

Claudio Foellmi

Supervised by

Timothy Roscoe
David Cock

November 2015 – May 2016

Informatik
Computer Science

Abstract

Different programming languages are suited for different needs. Rust is a new programming language that has received a lot of attention for its potential in the realm of systems programming. In this work, we take a closer look at Rust in the context of systems code. We compare it to other languages that have been used in the field, most notably C. We port Rust to Barrelfish, a research operating system, and show how it can be used to write both regular applications and well-integrated system services.

Contents

List of listings

1 Introduction

At the core of any computer system, there is code that manages and controls the hardware resources. This code is special, as it provides clean abstractions that the rest of the system can then use. To implement these abstractions, this code may need to do many things that a normal program would not, such as constructing objects from raw memory, writing to specific registers or executing specialized machine instructions.

Bugs in such code cannot only crash a program, but possibly take other parts of the system down with it, as the abstractions that other code relies on break down.

This suggests that systems code could benefit greatly from modern programming languages with strong safety guarantees and debugging facilities. But many of these languages come at a great cost, as they restrict the low-level machine access that systems code needs. Today, much systems code is still written in C, a language that is notorious for its many subtle opportunities for writing faulty code, and has been criticized for decades[8].

In this thesis, we take a closer look at Rust, a new programming language that tries to combine high-level guarantees and expressibility with the ability to opt in to low-level access where necessary. In section 2 we compare the features of Rust to other languages used for systems code, most notably C. In section 3 we report how we ported Rust to Barrelfish, a research operating system that makes heavy use of systems code in libraries. In section 4 we document how we integrated Rust into the Barrelfish ecosystem. In section 5 we discuss what applications we were able to run.

2 Background

2.1 System Programming Languages

There are many programming languages, and new ones are being developed all the time. While there is no exact definition of the term *systems code*, it is generally recognized to include operating systems, language runtime systems, compilers and databases. From the end user's perspective, systems code is simply all code that is not application code. All time spent executing systems code is therefore a form of overhead.

One of the most important requirements for writing operating systems is the ability to interface with different ABIs[1]. Interrupt handlers, system calls and context switching are all conceptually similar to functions, but do not follow the regular calling conventions or stack layout.

Hardware interfaces, such as page tables or memory-mapped device registers, are defined in terms of a bitwise representation in memory. To properly access this memory, we need to be able to manipulate individual bits, such as setting a flag in a page table entry. Buffers allocated to interact with hardware have to satisfy alignment constraints, making the ability to control alignment a matter of correctness rather than just optimization[39]. Controlling memory-mapped devices requires the ability to read and write to the device's addresses in a way that will not be reordered, cached, or otherwise optimized away by the compiler. Because the hardware interfaces are all specified in the same low-level model, systems programmers are used to thinking at that level. Having data types and operations that represent operations at that level is therefore advantageous.

In systems code, performance matters. A network stack that unnecessarily copies packets because of language constraints will perform much worse than one that can add or remove headers without copying the payload. Home users' machines are currently much more powerful than necessary, with plenty of cycles to spare. But systems code also runs in data centers, where a decrease in performance directly corresponds to an increase in cost.

As Kernighan argued in a critique of Pascal[22], one of the most practically useful features of a systems language is the ability to escape it. Being able to access a layer below the abstractions normally provided by the language in a consistent way allows the programmer to implement functionality not expressible in the language itself. In principle, any language with the ability to interact with lower-level code can be used to write operating systems. It is therefore not surprising that there have been operating systems written in many different languages[37].

[1]Application binary interface: the specifications for the interaction of code at the machine level, such as stack frame layout and calling conventions

2.1.1 Assembly

In the early days of computing, systems code was written in assembly — as was much of the application code. As compilers and language designs improved, the realm of assembly shrank down to small snippets of code that simply could not be expressed in other languages.

The main benefit of programming in assembly is the unparalleled level of control available. The code directly corresponds to machine instructions, and can use the entire range of hardware operations. The programmer manages all registers and memory accesses, allowing them to implement any ABI. But assembly code is inherently non-portable, as it is always written for a specific hardware architecture. Assembly code is hard to read and maintain, as even very simple programs require many lines of code. Memory locations can be referenced symbolically through labels, but all computations have to be done on the raw representation of data.

While there are few projects entirely written in assembly nowadays, small snippets of code are still used whenever the programmer needs access to specialized machine instructions, such as for reading performance counters or saving a stack frame. A notable example of a modern operating system written in assembly is the original L4 microkernel[27]. It was designed to provide the fastest possible inter-process-communication (IPC) mechanism, using hand-crafted assembly code to drastically reduce the number of costly cache misses. Its success demonstrated that the poor performance of previous microkernels such as Mach was not an inherent problem of the microkernel approach[20].

2.1.2 Managed Environments

On the other end of the spectrum are programming languages with managed environments. The benefit of using very abstract languages, such as Lisp or Haskell, is their expressiveness. The ability to write algorithms in fewer lines of code leads to a system that is easier to inspect and maintain. Strong type systems force the programmer to be explicit about their design choices.

Examples of kernels written almost entirely in Haskell are House and Osker [18, 12]. These represent low-level operations using a monad, similar to the IO monad commonly used in functional contexts.

However, the problem domain of operating systems consists not so much of inherently complex algorithms, but rather of many small decisions that are expected to be made often and quickly. Any additional time that a scheduler uses to decide what task to run next is time that could have been spent doing actual work. The representation of state in purely functional languages has traditionally been rather complex, yet the main task of operating systems is that of machine state management.

Because these languages require runtime support such as garbage collection, the design space of the resulting operating system is inherently limited. Any language with garbage collection will have variance in its execution time.

In the 1970s and '80s, the *Lisp Machine* was an attempt to build language support for Lisp into the hardware itself[17]. Tagged memory and dedicated microcode was used to speed up common operations by resolving dynamic types in hardware.

While functional languages can model the behaviour of a system, they are not suitable for the implementation if either resource constraints or performance are important. The seL4 kernel, the first formally verified operating system, used Haskell to model the kernel design[23, 29]. With the help of a hardware simulator, this model could be executed to run applications on the prototyped kernel design. The finalized design was then implemented using C. This dual-layered approach to system design is a great example of how a very abstract language can be useful even for systems with requirements the language cannot provide. The seL4 kernel is designed to never allocate heap memory. Haskell is able to express that constraint in the model, using its type system, but it is unable to implement it in its own simulations — the language uses heap allocation for all objects[11].

A different approach to using a managed environment is that of languages like Oberon[41]. Oberon belongs to a whole family of conceptually similar languages that are successors to Pascal. Its main design goal is simplicity: a single programmer can comprehend and implement the entire system, including the file system, interrupt handlers, a graphical user interface and even the associated compiler. The price for this simplicity is incompatibility: the custom file system and object file formats, which are tailored to the modular approach of the language, are not compatible with other, more widely used formats.

Oberon applications are structured as a collection of interdependent modules that are all run in a single process. When loading a new module, the runtime system first recursively loads all its dependencies. As all modules are executed within the same context, modern process separation techniques like virtual address spaces are not available. Therefore these systems are better suited for the development of special-purpose devices rather than general-purpose operating systems.

Low level operations, such as raw memory access, are encapsulated in a special *system* module, which is the only code not itself written in Oberon.

This approach of making an operating system out of the language runtime system, blurring the line between the two, is actually quite common. Many systems have been developed together with the language used to implement them[31, 9, 19, 16].

2.1.3 C

C is the *lingua franca* of systems programming. It was first developed in the early 1970s to rewrite the Unix operating system, which was up to that point written in assembly[34]. C was heavily influenced by BCPL[33], but added a static type system and support for byte-addressable memory. From the beginning, C was designed to be easily translatable to machine code. The operators on primitive types correspond directly to machine instructions on registers.

It should be noted that the dialect of C that is used for systems programming is actually quite different from the portable C specified in the ISO standards[15]. The standards mostly concern *strictly conforming* programs, and define the language in terms of an abstract machine. But much of systems programming concerns code that is inherently non-portable. This code is described by the standard as *conforming*, and relies on knowledge about the target machine, such as the representation of data types. For example, while the standard does not guarantee that signed integer types follow two's complement, it has been the norm for multiple decades, and in practice systems programmers can rely on that assumption.

The appeal of C for systems programming comes from the direct mapping of language constructs to machine constructs. The language has deliberately been kept small in scope. Interfacing with external services, such as for reading files, is done through libraries rather than keywords or built in types. Systems programmers are used to reasoning on the machine level, as this is how interfaces to the hardware are defined.

From the beginning, C has always been a language that is easy to escape[34]. Compilers have never been very strict about type safety, and much systems code uses the integer types just as a way to operate on the memory representation. The focus on pragmatism and hackability is what allows C to be used in many ways as a more high-level assembler. For embedded systems, the minimalism of C is crucial.

There are C compilers for basically any hardware and a variety of output formats, supporting many ABIs and interfacing with other languages in very transparent ways. Not only are many widely used operating systems (such as various Unix derivatives) written in C, but their officially-supported interfaces are C libraries, making it their primary language.

2.1.3.1 Memory Model

The memory model of C is based on statically typed objects and pointers. An object is defined entirely by its static type and representation in memory. In contrast to languages like Java, C pointers are value types themselves, and can be used in computations.

The C standard only makes guarantees about pointer arithmetic within objects, such as addressing individual elements of an array. Moreover, the representation of fundamental data types in memory is only guaranteed to follow one of several allowed formats, so any *strictly conforming* C code cannot use bitwise operations for arithmetic computations. In systems C code, pointers can be constructed by any computation, as they directly correspond to the addresses used in machine code. The representation of data types is part of the machine's architecture specification, and systems code makes liberal use of the knowledge of how a given data structure will be laid out in memory.

The language itself provides only two lifetimes for objects: Global and static variables are valid for the entire duration of a program's execution. Local variables are tied to the encompassing scope, and stored on the stack. When a function returns, its local variables can no longer be safely used, as the underlying memory may be used by another stack frame (see Listing 1). It is the programmer's responsibility to ensure that pointers do not outlive the objects they point to.

```c
// This function is dangerous: the pointer it returns
// has already been invalidated, so it cannot be safely
// used by the caller.
int *foo() {
    int a = 4;
    return &a;
}
```

Listing 1: An example of a dangling pointer.

Objects that do not fit within these two models can be stored in dynamically-allocated memory on the heap. Dynamic allocation is not part of the language core (i.e. it does not use special keywords or data types), but implemented in libraries, such as the functions *malloc* and *free* (Listing 2). This makes it easy to use custom allocators, which is valuable to systems code.

```c
// Here, the object will outlive the function.
// It is the caller's responsibility to call 'free' on the
// pointer when the object's lifetime ends.
int *foo(){
    int *a = malloc(sizeof(int));
    if (a) *a = 4;
    return a;
}
```

Listing 2: Dynamic memory allocation in C.

2.1.3.2 Undefined Behaviour

The C language definition contains many instances of *undefined behaviour*, such as reading from uninitialized memory or a signed integer overflowing [2]. A program execution that exhibits undefined behaviour is considered invalid, and there are no restrictions on its side effects[3]. This is different from *implementation defined behaviour*, where the compiler writer is allowed to choose the effect of a statement, but must consistently translate all instances of that statement the same way.

When optimizing C code, the compiler is allowed to assume that no undefined behaviour will ever happen. Any execution path containing undefined behaviour can therefore be ignored when generating code, which allows it to be optimized away. This is a very powerful concept — by leaving out edge cases from the specification of *strictly conforming* C, the standard can define a portable language core, while also allowing compilers to directly map C operators to the corresponding machine instructions on many different targets.

Any bug in the program that actually leads to undefined behaviour has enormous destructive potential. Because the compiler does not have to generate *any* code handling the path that was assumed to be impossible, the program might just continue the normal execution path, but with violated assumptions. Such a program may not just crash, but corrupt any data it can access along the way. Ironically, a common issue with undefined behaviour is the deletion of security checks[40]. Not only are there many ways to introduce subtle bugs in C code, but they may not even manifest until a new compiler version translates the code differently.

The biggest problem of undefined behaviour is that it is very difficult to warn the programmer about it. While some simple cases can easily be caught — like a function that provably always reads uninitialized memory — after just a few simple transformations by the compiler, any ordinary C program may contain many places with potential for undefined behaviour[24]. If the optimizer were to warn the programmer of every assumption it is allowed to infer, it would likely have to spew out so many warnings about ultimately harmless cases as to be unhelpful. Many of these would even be about code in system header files and transformed code that does not look anything like the original input, so the programmer would not be able to use that feedback effectively.

[2] A formalization effort counted 221 separate instances of undefined behaviour in the C standard, over half which are only detectable at runtime[13].

[3] In principle, undefined behaviour is therefore allowed to delete all the user's files, crash or just continue execution as if nothing happened.

2.2 Rust

Rust is a new programming language designed to provide strong safety guarantees by default, but with the option to access lower-level semantics where necessary. It incorporates many high-level concepts from other languages, such as generic functions, algebraic types and closures, but emphasizes easy ABI compatibility with C. Rust guarantees the absence of dangling pointers and data races by strictly enforcing rules that have been recommended as *best practices* in other languages such as C++. The prospect of a systems programming language with strong safety guarantees lead to Rust getting much attention long before its first stable release in 2015. A fundamental design principle of Rust is that of the *zero-cost abstraction*. Whenever possible, abstractions make use of compile-time knowledge, rather than make decisions at runtime.

2.2.1 Memory Model

The Rust memory model can be seen as a stricter version of the one used by C. Like C, variables can live for the entire duration of the program (static), or be limited to a scope (local). Unlike C, dynamically allocated memory is not explicitly managed by the programmer, but implicit in the use of container objects.

In Rust, lifetimes are part of the type system and statically checked by the compiler. This prevents all use-after-free bugs[32]: the compiler requires all code to be written so that objects provably outlive any references to them.

The main mechanism of memory management is ownership. All memory is *owned* by the variable that is bound to it. When the variable goes out of scope, the owned memory is deallocated automatically. If a variable's value is assigned to another variable, ownership is moved: the old variable cannot be used anymore, and the memory will be deallocated when the new variable goes out of scope instead. Listing 3 shows a simple example of an object's ownership being moved.

```rust
{
    // bind a value to a variable
    let x = Foo::new();
    // move ownership to a different variable
    let y = x;
    // illegal: x is no longer bound,
    // because the value has been moved to y
    x.foo();
}
// x and y leave scope
// (only y is still bound)
// the owned memory is freed up
```

Listing 3: Movement of ownership.

Types can implement the special 'Copy' trait, which marks them as being safe to copy: this replaces moves with value copying, as shown in listing 4.

```
// integer types implement 'Copy', so ownership is not moved
{
    let x = 4;
    let y = x;
    // x can still be used,
    // because the value has been copied rather than moved
    println!("{}", x);
}
```

Listing 4: Copy semantics.

To use a variable's value without moving ownership, functions can take arguments by reference. This pattern, as shown in listing 5 is called *borrowing*.

```
// function moving ownership back after use
fn foo_moving(x: Foo) -> Foo {
    // do something with x
    // return x to move ownership back to caller
    x
}

//function doing the same operations without moving
fn foo_borrowing(&x: Foo) {
    // do something with x
}
```

Listing 5: Borrowing arguments instead of moving both ways.

At any point, there may be either one mutable reference or any number of immutable references to a value, and the compiler statically checks that the value outlives all references to it. For many common cases, this is trivial because of the LIFO nature of function calls. A function that takes a reference will return before the caller returns, so any memory owned by the caller outlive the references given to the callee. The restriction to at most one mutable reference at any time is Rust's way to deal with the problem of aliasing. Because multiple mutable references never refer to the same object, the generated code does not need to handle that case.

Like in C, objects in Rust are allocated on the stack by default. Dynamically allocated memory is managed through container objects. Instead of the explicit allocation and deallocation of *malloc/free* in C, Rust uses container types like *Box* that abstract pointers to heap-allocated memory. When the Box is deallocated, it also automatically reclaims the memory on the heap (listing 6).

While the ownership and borrowing rules prevent use-after-free bugs, they are very restrictive, especially with regards to data structure layouts and optimized algorithms. As a simple example, a doubly-linked list is a very basic

```
{
    // allocate a new Foo object on the heap
    let x = Box::new(Foo::new());
    // do something with x
}
// x goes out of scope -> owned memory is deallocated
// when Box is deallocated, it frees up the heap memory as well
```

Listing 6: Boxed values.

data structure that requires two mutable references to the same node, one for each direction. In safe Rust, this is strictly forbidden. To allow programmers to still implement such data structures, Rust also has raw pointers, which behave more like their C counterparts. Listing 7 shows the two struct layouts, only one of which will work. Raw pointers can point to invalid addresses, be

```
// naive layout of a doubly linked list
// this will not work, as there can only be one
// mutable reference to any object at a time
struct Node {
    next: &mut Node,
    previous: &mut Node,
    data: Foo,
}

// same layout using raw pointers instead of references
// this will work, but requires unsafe code
struct Node_raw {
    next: *mut Node_raw,
    previous: *mut Node_raw,
    data: Foo,
}
```

Listing 7: Raw pointers allow mutable aliasing.

used for pointer arithmetic, and are allowed to alias other pointers, so they are not safe to use. To separate this unsafe code from the rest, there is a special *unsafe* keyword, that allows a block to violate the memory safety rules. This allows programmers to write data structures that use unsafe code internally, but provide a safe interface to the user[4].

The single-owner-restriction allows Rust to enforce memory safety without a garbage collector. Objects have a statically known lifetime with a very clear end, at which point they can be deallocated safely. The only way to get a dangling pointer is through code that has to be explicitly marked as *unsafe*, narrowing down the search space when debugging.

[4] It is the responsibility of the programmer to ensure that unsafe blocks comply with the invariants of the language[4].

2.2.2 C Compatibility

Rust provides compatibility with C through a well-integrated foreign-function interface (FFI). While the default ABI is different from C, each function can have its ABI made explicit through the *extern* keyword. This allows for both the declaration of external C function prototypes and the definition of Rust functions that can be called from C. Most C types can trivially be expressed as Rust types, and Rust structs and enumerations can be declared to use the C representation, so moving data across the FFI does not require conversions[5]. Because C functions do not enforce the memory safety rules of Rust, calling a C function is considered unsafe and can only be done in an unsafe block.

As all Rust functions have their symbol names mangled by the compiler (to implement namespacing), in order to call a Rust function from C code, one either needs to provide the C code with a function pointer, or deactivate the name mangling for that function to give it a fixed symbol name, as shown in listing 8.

```
// The attribute ensures that the object file
// contains a symbol 'foo',
// so it can be linked to from other languages.
// The 'extern' keyword forces this function to follow the C ABI.
#[no_mangle]
extern pub fn foo(i: u64) -> u64 {
        i * 7
}
```

```
// the corresponding function prototype in C code that uses it
extern uint64_t foo(i: uint64_t);
```

Listing 8: A simple extern function callable from C.

As we will discuss in section 4.1, the interactions between C and Rust are not always that straightforward. This is mostly because many C libraries have their own constraints on object lifetimes and the proper ways of managing memory.

[5] The exception are unions, as Rust uses tagged unions to represent sum types. In isolation, C unions can be dealt with using the transmutation functions of Rust, but there currently is no clean way to represent unions embedded in structs. We expect official support for untagged unions to be implemented soon after the publication of our work, as a corresponding language extension has already been approved[3].

2.2.3 Type System

Rust is statically typed, but much like in C, a value is entirely defined by its type and representation in memory. Objects in Rust can implement *traits*, which are similar to Haskell typeclasses. A single type can implement any number of traits, with some like *Default* or *Copy* having special semantics built into the language. This many-to-many relationship between traits and implementing types allows programmers to express interface inheritance. Notably, traits are not types themselves. One can therefore neither declare an object of type *Default*, nor define a new type to be an alias of *Default*. However, one can declare an object to be of type &*Default*, meaning "reference to an object implementing the *Default* trait". This is called a *trait object*, and is the idiomatic way to achieve dynamic dispatch in Rust. The compiler will implicitly generate a set of Vtables[6] for this trait, which is used to find the correct methods at runtime.

2.2.4 Macros

Rust code can be transformed at compile time using either macros or compiler plugins. Macros are declared as a list of patterns and expansions. When a macro is used, the patterns are matched against the arguments given, and the corresponding expansions used as substitutions for the invocation.

```
//simple macro with only a single pattern
macro_rules! max {
    ($a:expr, $b:expr) => (if $a > $b {$a} else {$b});
}

max!(4, 7); // evaluates to 7
max!(4.2, 7.2); // evaluates to 7.2
max!(4, 7.2); // compile time error: mismatched types
```

Listing 9: A macro definition and usage.

Listing 9 shows a simple macro definition consisting of a single pattern. It transforms two comma-separated expressions a and b into a new expression that computes the maximum of the two.

In contrast to C, Rust Macros are applied on the abstract syntax tree (AST) rather than on the source code, and can therefore not be used to construct new tokens from parts. Macros are only allowed to transform AST subtrees into other AST subtrees, but they support recursion and have their own naming scope, making them very similar to functions and much less error-prone than raw textual substitution.

The Rust standard library uses macros in very prominent places, such as for printing text and initializing vectors. This is because macros allow for a variable number of arguments of any combination of types, whereas functions are limited to a fixed number of arguments of predefined types.

[6] *Virtual method table*, a table of function pointers. Every type implementing the trait will have its own copy, filled in with function pointers to its own implementation of the trait's methods.

Compiler plugins are more powerful, as they can change even the parser (e.g. by introducing new keywords), but they do not have a stable API yet and are harder to write. It is unclear if compiler plugins will ever have a stable API, since they interact with compiler internals in ways that are difficult to abstract away.

In the current design of Rust, macros do not follow the same namespacing as the rest of the code, because they are not actually *items* and therefore resolved in a separate compilation step before name resolution. As we will discuss in section 4.2, this special treatment of macros is actually a hindrance to writing modular code. There are efforts to solve this problem in future releases[1], which will make it much easier to write and use macro libraries. But until then, we generally recommend against the use of macros as APIs wherever possible.

2.2.5 Error Handling

Rust has two mechanisms for dealing with errors: return values and stack unwinding. The standard library contains a generic *Result* sum type, that is used throughout the library as the return type of functions that can fail. A *Result<T, E>* value can either be *Ok(t)* or *Err(e)*, where t is of type T and e of type E. This is essentially the type safe equivalent of the return codes commonly used in C. Much like Haskell's *Maybe* monad, it enforces a clear distinction between the result of the function and the computed value (in case of success), by making them different types.

For more unexpected failures, there is also an exception handling system based on stack unwinding. When an exception is raised, such as through the 'panic!' macro or assertion failure, a panic handler aborts the current operation in a memory-safe way. The stack is unwound one stack frame at a time, by executing special code that releases any resources that the aborted functions owned. This code (called a *landing pad*) is generated by the compiler for each function.

This stack unwinding mechanism is similar to the exception handling of C++ and common C extensions, but in Rust it is not intended as the default mechanism for handling errors. There are no objects to represent the context of an exception, and while there is a mechanism for catching exceptions in a user's code, it is a new addition to the language and highly discouraged.

The stack unwinding system is a remnant of earlier Rust designs, when there were execution contexts much smaller than native threads. Even though the stack unwinding provides memory safety by correctly reclaiming only out-of-scope objects, writing code that is *exception safe* is far from trivial, as listing 10 shows. Exception safety is the property that all the logical invariants of a program hold after an exception has been handled. In general, the only way to reliably write exception safe code is by structuring all functions as transactions, so they can be aborted cleanly.

```
// Example of a function that is not exception safe.
// If 'add' panics, the accounts will be in an inconsistent state,
// even though memory safety has not been violated
fn transfer_money(&sender: Account, &receiver: Account,
                  amount: u64) -> Bool {
    if sender.can_afford(amount) {
        sender.remove(amount);
        receiver.add(amount);
        true
    } else {
        false
    }
}
```

Listing 10: Exception-unsafe code.

The course-grained nature of Rust's exception handling severely limits its usefulness. To properly handle an exception, the programmer typically needs to know what failed, but finding all the possible points of failure in a given block of code is actually very hard. Because of the difficulty of handling exceptions correctly, and the complexity associated with having implicit exit points all over the code, the future of the entire exception mechanism is a regular topic for discussion in the Rust community[2]. One important plan is a compiler option to turn all exceptions into simple aborts, shutting down the whole program instead of trying to recover. This would be especially interesting for systems code, as we will discuss in section 4.1.

Writing exception-safe Rust code is in many ways similar to writing memory-safe C code: There are many ways to get it wrong, some of them very subtle, and the compiler is not much help. For failure states that the systems programmer anticipated, using return values works very well. This way, not only can errors be handled by the code that called the failing function, but it allows us to reserve panics for truly unrecoverable problems. And in that case we do not actually need the stack unwinding and object deallocation that Rust uses by default.

2.2.6 Compiler

There is currently only one Rust compiler, called *rustc*. It is the reference implementation, as there is no formalized language standard at this point. Rustc uses LLVM[25] as its backend, which allows it to be used for many target machines.

The compilation unit of Rust is called a *crate*. A crate is either a library or an application, and consists of a hierarchy of modules organized in a tree structure. At the top level of that hierarchy is the crate root, which is also the root of the namespace. Namespaces are hierarchical and follow the same layout as the modules, but names can be reexported from any module. This is in contrast to languages like C++, where namespaces are orthogonal to source layout and can be spread across any number of locations.

By default, all symbol names are mangled, to prevent conflicts between namespaces. This also allows a single application to link to multiple different versions of a dependency.

Rust is still a very young language, and developing at a rapid pace. Version 1.0 was released on May 15, 2015. Minor releases are made every six weeks, each expanding the set of stable library APIs. When we started this work, we used the then cutting-edge version 1.5, but despite updating to version 1.7 in between, we will already be out of date again by the time of publication.

During its very early development, Rust went through several major design changes. There used to be a runtime system, multiple levels of threading, several different kinds of pointers, optional garbage collection and a notion of purity. Much of the literature on Rust therefore concerns versions of the language that are quite different from the current one. But even when it still contained a garbage collector, it was already interesting to many systems programmers. Since the release of version 1.0, changes have been much smaller in scope, and mostly related to the finalization of API designs.

2.2.7 Related Work

Even though Rust is still a very young language, it has already attracted much attention from the systems programming community. Rust has already been used to implement an embedded operating system called Tock[26]. A very important insight from that effort was that the strict rules on sharing state between threads do not match well with the traditional design of embedded systems. Embedded systems are typically very resource-constrained, both in terms of memory and processing power. Their hardware usually does not provide isolation mechanisms such as virtual memory. As a consequence, many embedded operating systems are designed to run all applications in the same context, using lightweight threads, often implemented with cooperative scheduling. Any isolation between applications is restricted to the source-code level. In such a system, much information is shared between subsystems. But Rust places heavy restrictions on mutable shared state, because its rules are designed to prevent race conditions in a more general threading model. The authors propose an extension to the type system that would allow multiple mutable references to the same object within a thread, but forbid the sharing of those references across execution contexts.

Shortly before the first stable release of Rust, it was used in a student project[28] to implement a Unix-like operating system called *Reenix*. In that effort, the restrictions on shared state were also highlighted as a problem. The greatest advantage in expressiveness came from the use of traits and algebraic data types. As Reenix is a monolithic operating system, with drivers and buffer caching in kernel space, the heap allocation mechanism was identified as a serious problem. All heap-allocated object types of the standard library such as *Box* and *Vec* use an allocation mechanism that is assumed not to fail. In user space, this assumption is not all that unrealistic, since page swapping, copy-on-write and on-demand paging can give the illusion of an abundance of available memory. The point of failure, where the available physical memory has actually run out, may only be discovered long after the system call used to allocate it. In kernel space however, there is little memory that can safely be swapped out to persistent storage, and there is no system further down in the chain to delegate dealing with allocation failures to. Using the standard container types inside the kernel is therefore dangerous. The proposed solution was to make the possibility of allocation failures explicit. Changing the various container types to cope with failures requires serious changes to the way Rust code is written, but it may be feasible to simply extend the current implementations to provide an additional constructor, which returns an error if an allocation fails.

In 2015, Rust was ported to the distributed research operating system *DIOS*[38]. This work was done shortly before the first stable release of Rust, at a time of great changes to numerous APIs, making the port more difficult. The performance of IO-bound programs in Rust was observed to be comparable to that of C code using the DIOS APIs directly.

In 2013, LLVM released a compiler back-end for PTX, a virtual instruction set for GPUs [7]. Holk et al used this back-end to program GPU algorithms in Rust, by adding extensions for GPU-specific operations[21]. One of the challenges they faced was the conflicting semantics of LLVM address spaces between PTX and Rust. This was when Rust still had optional garbage collection, and used address spaces to separate collectable regions from other memory.

[7]Graphics processing unit. A specialized processor for high-throughput parallel data processing.

2.3 Barrelfish

Barrelfish is a research operating system developed at ETH Zurich. It explores a machine model that treats the set of processors similarly to a network of machines[7]. Processes communicate using message passing through clear interfaces instead of implicitly shared state. In Barrelfish, much of the functionality traditionally associated with operating systems is implemented in user space libraries and services, rather than kernel space.

Each processor runs its own independent copy of the kernel, which is called the *CPU driver* as it only handles bootstrapping, process scheduling, capability management and interrupt forwarding, all of which require operations exclusive to kernel space. Memory allocators, device drivers, file systems and even network stacks all run in user space, either as libraries that applications can link to, or as services that communicate with client processes.

The scheduling of threads is done by the applications themselves, using scheduler activations[5]. Applications are responsible for all their own resources, including the memory used to represent them in the kernel, using a capability system modeled after seL4[23].

Barrelfish is written almost entirely in C, but also uses several domain specific languages (DSL). These languages compile to C stubs, which are then called from the C code. The main purpose of these DSLs is to translate declarative specifications, such as the memory layout of a device, into procedural code. These DSLs are a great way to combine expressibility with efficiency, but they are by their very nature restricted in scope and require a lot of initial work to develop.

We were interested to find out if Rust could provide benefits similar to those of the DSLs, but to a larger fraction of the codebase. The zero-cost abstractions of Rust are a different way to achieve expressibility without sacrificing efficiency. We were curious to see how well Rust was suited to replacing system services written in a combination of C and DSLs.

There have been previous projects to integrate other languages into Barrelfish. Most notably Go[30], which was used to explore how the different message passing designs of the language and the operating system interact. In December 2014, Zaheer Chothia implemented a proof-of-concept port of Rust to Barrelfish. The insights into the structure of Rust gained in that project were a great help in our own porting efforts.

3 Porting Rust to Barrelfish

In this section, we describe the work that was necessary to compile Rust programs for Barrelfish. The Rust compiler itself did not require any changes, but merely the definition of a new target (*x86_64-pc-barrelfish*).

3.1 Standard Libraries

By default, all Rust code links against *libstd*, the standard Rust library. It provides definitions for important types like *Result* and *Option*, standard traits like *Copy* and *Iterator*, container types like *Vec*, methods for string manipulation and bindings for system-specific functionality like files and sockets.

As most Rust code depends on this library[8], porting it is essential in the effort to support a new platform.

Many of the definitions it exports are not actually implemented in libstd itself. Instead, it merely reexports definitions from other libraries in a unified, well-structured namespace. The most important of these libraries that libstd depends on is *libcore*, which contains the definitions for all the primitive types (e.g. *u*64), as well as many other essential types such as *Option*. As all Rust code ultimately depends on libcore[9], it is designed to be entirely portable. Porting it to Barrelfish did not require a single line to be changed.

The various container types are all part of *libcollections*, which also did not require any changes. This is because the target-specific aspect of containers — heap allocation — is the subject of its own library. All heap allocations in Rust are internally based on calling a C library function that provides aligned allocations, such as *memalign*. Because the standard C library used in Barrelfish does not yet provide such a function, we use a different memory allocation library.

Commonly used C bindings are collected in a library called *liblibc*. These include correctly-sized aliases for the various C integer types, but also system-specific definitions such as the socket interface of Unix-like systems. For Barrelfish, we decided to gather system-specific C bindings in its separate library, which will be the topic of the next section. This left the Barrelfish port of *liblibc* small, mainly consisting of mappings of symbol names for C library functions.

Many of the modules implemented in libstd itself are concerned with system-specific functionality. Some of the features libstd expects from the host system, like a Unix timestamp counter or pipes, are not available on Barrelfish. Applications that rely on such system-specific functionality will therefore require additional work in order to be ported to Barrelfish — either modifying the application or adding the necessary features to Barrelfish.

One of the biggest challenges with porting libstd is its sheer scope. Much of the functionality it wants to expose is actually not part of the main Barrelfish library (*libbarrelfish*). The network stack and file system for example are their own separate libraries, and not linked with by default. To reduce the scope of our porting effort, we decided to exclude the *std::net* module, which is easily

[8]It is possible to disable the automatic inclusion of libstd into every crate, and directly use libcore instead. But until recently, this was not possible on the stable release build of Rust.

[9]The attribute that is used to opt-out of importing libstd implicitly imports libcore instead.

decoupled from the rest of libstd. As Barrelfish does not use dynamically linked libraries yet, we did not port the corresponding module.

Porting the *std::sys::** modules largely consisted of adapting the *mutex* and *thread* implementations to use Barrelfish-specific memory representations and operations. One assumption that libstd makes is that all threads share their address space. The communication modules in *std::sync* all assume that communication channels can be implemented by atomically accessing shared buffers that are allocated somewhere on the heap. In Barrelfish, this assumption does not hold in the general case, but is valid if all threads are running on the same core, which is the default. The difference between multicore applications and multithreaded applications is very explicit on Barrelfish, and the *std::thread* interface implicitly relies on guarantees only the latter provides.

While the higher-level abstractions of libstd are written entirely in Rust, much of the system-specific code relies on C libraries. In Barrelfish, we use a different set of libraries to what Unix-like systems use. To ensure all necessary symbols are defined, we link Rust programs on Barrelfish against the following libraries:

libbarrelfish for many definitions related to threading and mutexes

libdmalloc for memalign

libnewlib as the C standard library

libcxx for the stack unwinding routines to handle panics

libvfs for file descriptor support

3.2 Liblibbarrelfish

In Barrelfish, much of the functionality often associated with operating systems is provided through user space libraries instead. The most important one is *libbarrelfish*, which contains code to do system calls, use capabilities, spawn new processes, schedule threads, and interact with the service processes (like the memory server or the monitor). Naturally, we want to be able to do these things from Rust code as well. Notably, *libstd* requires interfaces to manage threads and locks.

Using a C function from Rust code is as simple as declaring it in an *extern* block, which can be done at any point where you can declare functions. But declaring the C functions in every piece of code that use them is error-prone, especially in a system with a large code base. To interface with libbarrelfish from Rust more cleanly, we therefore wrote a simple wrapper library called *liblibbarrelfish*[10]. It contains common type and function definitions for interacting with libbarrelfish, as well as a few functions written in Rust.

One problem we faced when writing this library is the structure of the Barrelfish project. Much of the functionality is in header files, and makes heavy use of inline functions. Inline functions are very useful in C, as they are a way to combine the inlining of macros with the scoping and syntax of functions. However, they are bound to the language in a way that regular functions are not.

There exist automatic tools for generating Rust code from C header files. But these tools are limited by the fact that Rust is much stricter than C, especially with regard to the type system. In C, constants are often written using the preprocessor, which simply performs a textual substitution. In Rust, constants are items just like variables, and therefore require a type in their declaration. In C, enumerations do not have to be exhaustive — they are essentially just numeric types after all. In Rust, enumeration definitions must be exhaustive. Assigning a value outside the defined range to an enumeration is undefined behaviour, and prevented by the compiler in safe code. Converting idiomatic C code into idiomatic Rust code is therefore not trivial.

On the other hand, writing the adapters by hand allows us to use the stronger semantics of the Rust type system. We can use the fact that Rust enumerations are range-checked at compile time, for cases where we know that the declaration in C is complete. Types that we treat as black boxes[11], such as threads, we represent as empty enumerations. This prevents them from being instantiated, but ensures that pointers to them are different types, giving us the ability to typecheck function calls.

One of the most important types in *libbarrelfish* does not get stronger guarantees. Access control and resource management in Barrelfish is implemented through capabilities, which are implemented in a similar way to seL4[14]. A capability may represent a region of physical memory, a device, a communication endpoint or a privileged operation. The actual capabilities are stored in slots only accessible to the kernel, but they can be referenced in user space. Capabilities can be used through the *invoke* system call, by specifying their slot address and the operation to perform. In principle, the Rust type system is a great fit for capabilities. The various legal operations can be represented

[10]Named after *liblibc*, the wrapper around *libc*.

[11]Meaning we do not allocate them nor access their content in Rust.

by traits, with each capability type implementing the appropriate traits. But although that approach works for the capabilities themselves, capability references are slightly different and cannot be represented the same way. A capability reference only refers to a slot, which may change its contents completely, and do so without the user space code's knowledge. A local copy of a capability owned by another process may be revoked at any time, at which point the slot becomes empty. Any user space representation that includes type information is therefore liable to get out of sync with the actual contents of a slot. One naive way to repair an out-of-sync reference would be to have the kernel return the actual type of a capability if it is invoked illegally[12]. But because the invocation is a method on an object, we can not change the type of the invoked capability reference directly. Instead, we have to return a new capability reference to the user, who then needs to replace their old version with this up-to-date one[13]. These complications all arise from the attempt to model different kinds of capability references as different types. For our port, we instead kept the model we use in C, where all capability references are a single type. Invocations can fail for many different reasons, and providing the wrong type of capability is simply another one.

The system call interface is a great example of the usefulness of recursive macros. The actual system call is implemented as a function, using inline assembly to control the contents of all registers, as shown in listing 11. A system call can involve a command identifier and up to twelve register-sized arguments, so that has to be the signature of the syscall function. In practice however, most system calls only require very few arguments, and many of them are even smaller than a register. To make the use of system calls cleaner, we wrote a simple macro that recursively pads out the list of arguments, shown in listing 12. This is typical of the way macros are used in Rust. While functions must have a fixed number of arguments of predefined types, macros can be defined to take any number of arguments of any type.

[12]Another way would be to add a *synchronize* method to all capability reference types, but this runs into the same problems.

[13] Even worse, because the different kinds of capability reference are different types, we can not return them by value, but have to use trait objects instead.

```
/// perform a x86_64 syscall
#[inline]
pub fn syscall(num: Syscall, arg1: usize, arg2: usize,
               arg3: usize, arg4: usize, arg5: usize,
                           arg6: usize, arg7: usize, arg8: usize,
                           arg9: usize, arg10: usize, arg11: usize,
                           arg12: usize) -> Result<usize, Errval> {
    unsafe {
        let result: usize;
        let errval: usize;
        asm!("pushq %rbp; movq %rcx, %rbp; syscall; popq %rbp"
            : "={rax}"(errval), "={rdx}"(result)
            : "{rdi}"(num as u64), "{rsi}"(arg1), "{rdx}"(arg2),
                        "{r10}"(arg3), "{r8}"(arg4), "{r9}"(arg5),
                        "{r12}"(arg6), "{r13}"(arg7), "{r14}"(arg8),
                        "{r15}"(arg9), "{rax}"(arg10),
            "{rcx}"(arg11), "{rbx}"(arg12)
            : "r11"
            : "volatile"
        );
        if err_is_ok(errval) {
            Ok(result)
        } else {
            Err(errval as Errval)
        }
    }
}
```

Listing 11: A system call in Rust using inline assembly.

```
macro_rules! syscall {
    //base case: too many arguments
    ($num:expr, $a1:expr, $a2:expr, $a3:expr, $a4:expr,
        $a5:expr, $a6:expr, $a7:expr, $a8:expr, $a9:expr,
        $a10:expr, $a11:expr, $a12:expr, $($a13:expr)+)
    => (panic!{"illegal: syscall with more than 12 arguments"});

    //base case: all arguments
    ($num:expr, $a1:expr, $a2:expr, $a3:expr, $a4:expr,
        $a5:expr, $a6:expr, $a7:expr, $a8:expr, $a9:expr,
        $a10:expr, $a11:expr, $a12:expr)
    => ($crate::syscall::syscall($num, $a1, $a2, $a3,
            $a4, $a5, $a6, $a7, $a8, $a9, $a10, $a11, $a12));

    //recursively insert a 0 argument
    ($($arg:expr),+) => (syscall!{$($arg),+ , 0});
}
```

Listing 12: A recursive macro to use system calls with any number of arguments.

3.3 Example Program

The port of libstd is enough to run regular Rust programs on Barrelfish. An example of such a program is the fibonacci program shown in listing 13, which makes use of command-line arguments, formatted output and multithreading. As can be seen from this example, Rust allows us to write very compact code, leading to a smaller set of lines to debug.

```rust
use std::thread;
use std::env;

fn main() {
    let n = match env::args().last().unwrap_or_default().parse() {
        Ok(i) => i,
        _     => 4
    };

    println!("fibonnacci({}) = {}", n, fibonacci_threaded(n));
}

// naive computation of a fibonacci number using threads
fn fibonacci_threaded (n: u32) -> u32 {
    if n < 3 {
        return 1;
    }
    println!("spawning threads to compute fibonnacci({})", n);
    let left = thread::spawn(move || fibonacci_threaded(n-1));
    let right = thread::spawn(move || fibonacci_threaded(n-2));
    let leftres = left.join();
    let rightres = right.join();
    match (leftres, rightres) {
        (Ok(l), Ok(r))  => return l + r,
        _               => panic!("fibonacci({}) failed", n),
    }
}
```

Listing 13: A simple program using input, output and multithreading.

4 Integrating Rust into Barrelfish

While *libstd* and *liblibbarrelfish* are enough to run regular Rust programs on Barrelfish, they do not suffice to write system services or device drivers. Barrelfish programs make heavy use of subsystems that cannot simply be wrapped by *liblibbarrelfish*. Most notably, communication and device access are implemented using domain-specific languages called *Flounder* and *Mackerel*, each with their own toolchains. To truly interface with the rest of Barrelfish, we need to be able to use these languages from our Rust code.

4.1 Flounder

Flounder is the language for specifying message interfaces in Barrelfish[6]. Because message passing and RPC are the primary forms of interprocess communication, they are absolutely essential to Barrelfish, and their implementations are closely coupled with other fundamental aspects like scheduling and the capability system. Any new interface to this infrastructure therefore has to be binary-compatible with the underlying implementation that does all the heavy lifting (such as choosing transport layers[14] and triggering events).

A simple example of a Flounder interface is xmplmsg, as shown in listing 14. This short piece of code defines a new message passing interface called *xmplmsg*, that consists of two messages. One for transmitting a pair of integers, the other transmitting a C-style string.

```
interface xmplmsg "Example message interface" {
        message msg_ints(int i, int j);
        message msg_string(string s);
};
```

Listing 14: A Flounder message interface definition.

The Flounder DSL compiler generates multiple C header files from this definition, which contain functions for sending and receiving such messages (using CPU registers or shared memory to transmit the message payload). Conceptually, all messages are transmitted over channels, with each interface using a separate channel. To create a channel, one process publicly advertises itself in the system's nameserver. Another process then binds to that advertised interface reference. Once a channel is created, each process can register a set of handler functions that should be called to process incoming messages. Sending and receiving both happen asynchronously. The sender can optionally specify a callback that should be executed once the message has been sent.

[14] The two main transport layers of Flounder are local message passing (LMP) and user level message passing (UMP). LMP uses a system call to transport the message through the kernel, and can only be used to communicate with processes on the same core. UMP uses shared memory to transport the message entirely in user space, and can be used across cores.

4.1.1 Design

To be as compatible as possible, we designed our Rust interface to reuse the generated C code for sending and receiving, and simply wrap the existing channels in an adapter that provides a more idiomatic Rust interface to the end user. This allows us to use regular Rust functions as message handlers, and method calls for sending. In the underlying C code, each channel contains two virtual method tables (vtables)[15], one for sending messages and one for receiving. The table for sending is filled in by the underlying transport layer when the channel is created. The table for receiving is meant to be filled in by the user.

In Rust, we represent Flounder channels by a *Channel* object, and operation on a channel as methods on that object. Sending a message is done by calling the corresponding *send* method. The body of this method simply consists of a lookup in the vtable, and a call to that function pointer. Providing a callback to be executed after the message has been sent turned out to be much more complicated than we originally expected. The Flounder internals require any dynamically allocated memory to be kept alive until the send is complete and the callback is executed. This is clearly a specification of a lifetime, but we cannot express it in Rust because the actual time of deallocation is not statically known and cannot be tied to the sending method. This means that we cannot expose the asynchronous interface in a memory-safe way. The user has to make sure that the referenced variables outlive the sending process without the compiler's help.

To give the end user a memory-safe alternative for sending messages, we also define a blocking send method for every message. It first consists of a call to the nonblocking version, providing a callback that directly sets a variable on the stack. It then waits until this variable has been set. While this design is very simple, it relies on unsafe blocks, so it is better suited to be implemented in the library rather than in user code.

On the receiving side, we want to be able to use regular Rust functions as message handlers. By default, Rust functions do not use the same ABI as C functions. This difference is what allows the Rust compiler to optimize expressions that would naively involve temporary copies, such as the common pattern shown in listing 4.1.1. To be directly callable from C, functions can be

```rust
// The inner function call can be rewritten by the
// compiler to return its value by reference, saving us
// from copying memory around just to put it on the heap.
let b = Box::new(std::time::SystemTime::now());
```

declared with the *extern* keyword. This ABI change is part of the type, so it is possible to have a function that only accepts C-callable functions as its argument, as in listing 15. In theory, this is enough to write message handlers and have them be called directly from the underlying transport layer. However, writing such a message handler correctly requires knowledge of the intricacies of both Flounder and the Rust foreign-function interface. To help users, we introduced a small function that sits between the C code that transports the

[15] A table of function pointers, each index corresponding to a message.

```
fn call_fn(f: extern fn()) {
        // extern functions can be used from Rust
    // just like any other function
        f();
}
```

Listing 15: A function that only accepts functions callable from C.

message and the Rust function that processes it. In Flounder, dynamically-sized payloads (such as strings) are transmitted using a base pointer and a length. The backing memory is allocated by the transport code using *malloc*, and the programmer is expected to call *free* once they are finished with it. This is in contrast to the way memory ownership is handled in Rust. Our helper functions do the conversion to vectors and Rust strings, which includes allocating new memory for the idiomatic data type and freeing up the received pointer. This requires copying the data, because all container types manage their own memory and will deallocate it using their own method, which may be different than just calling *free*.

Another big constraint holds for any Rust function called from C code: any panics[16] must be caught before the automatic stack unwinding reaches the C code. This is because the C functions do not have any unwind information associated with them, so the panic-handling code does not know how to clean them up, and aborts the program instead. The officially recommended way to catch panics is to spawn a new thread that does all the work. Because this thread does not contain any C stack frames, it can exit cleanly even in the case of a panic. However, spawning a whole thread for every single message we receive is clearly not appropriate. While threads in Barrelfish are cheap compared to other systems (as they are managed completely in user space), Rust's panic handling is supposed to be a zero-cost abstraction, so we should only have to pay for it when we actually use it. Luckily, a general panic-catching function has recently been added to *libstd*, with the specific intent to let callbacks from C catch panics within their own thread. If we were to directly call the user-written handler from C, it would be the user's responsibility to explicitly catch panics in all message handlers they write. By moving this step into the helper function, we can run arbitrary message handlers.

[16] See section 2.2.5 for an overview of Rust's panic mechanism.

4.1.2 Implementation

We added a new backend to the Flounder compiler. For each Flounder interface, it generates a small Rust library that defines a *Channel* type with methods for sending and receiving the corresponding messages. The Flounder compiler is written in Haskell, and already contains several backends, used to generate C code for the various transport layers.

One oddity of the internals of Flounder is the way it handles dynamically sized arrays. While they are represented very straightforwardly as a base pointer and a length variable, they are not considered a type by the language. Flounder's type system contains both a set of built in primitive types (such as bytes or pointers), and a set of higher-order types (such as arrays or structs), but dynamic arrays are part of the message specifications instead. While this does have the nice invariant that all types can be represented by single variables, it moves much of the logic dealing with arrays into the code that is dealing with messages — which is already the most complicated part.

In order to insert our helper functions between the C code and the user-supplied message handlers, we introduced a third vtable, which holds the actual message handlers. The receiving vtable used by the C code is then filled in with our helper functions. When a message arrives, the helper function moves any dynamically allocated memory into the corresponding Rust container type, looks up the user-supplied handler function in the vtable, and then executes it in a context that will catch panics before they propagate into C code. For users who do not want to pay this overhead, the definition of the *Channel* type is deliberately exposed, so they can register their own raw message handlers directly in the vtable used by C[17]. We expect that any safe wrapper around an asynchronous C library will need to implement similar mechanisms for catching panics and managing memory.

[17] Measurements on hardware suggest this overhead to increase the round-trip time of each message by about 10%.

Based on these experiences, we suggest the following language extensions to Rust:

nonpropagating functions Similar to how the *extern* keyword changes the ABI of a function and is reflected at the type level, the same approach can be taken to change the panic-handling properties of functions as well. Being able to specify that a function will not propagate panics downward would allow system programmers to write callbacks for C libraries without having to always explicitly catch panics in the function body. Moreover, it would allow us to specify that a function accepts a non-panicking callback at the type level. The *extern* keyword can already take a parameter to specify the ABI, so this extension would not require any new syntax. The way stack unwinding works, each function already has an entry in a table that specifies what action to take when unwinding into it. Expressing that choice at the type level is therefore natural.

malloc container The various Rust container types all reclaim the memory they represent when they are destroyed. The mechanism by which they are allocated and reclaimed is not specified, so transmuting a pointer into a Box is only allowed if the pointer was originally allocated by the Box allocator. When dealing with C libraries, one often receives ownership of memory that was allocated by the library, but needs to be freed by the user. Having a container type to abstract this situation would allow us to directly use that memory from Rust, without having to worry about either use-after-free bugs or memory leaks. The container would need to use the same representation as the pointer itself (which is already how boxes work), so it could be used in the declarations of C functions. As its destructor, it would simply call free (or possibly another reclamation function, depending on the API of the C library) on the pointer.

Both of these extensions are compatible with the greater Rust design, as they are merely extending existing mechanisms to cover more use cases. But while they are modest in scope, they would greatly simplify the efficient interfacing between complex C libraries and Rust.

4.2 Mackerel

Managing devices is a very important and common task in systems code, both in user space and kernel space. Yet it has traditionally received very little attention and support from language designers and compilers. So much so that one of the major advantages of C over other languages is not that it makes device access easy, but just that it allows it to be implemented at all without resorting to assembly. But directly writing the C code to deal cleanly with memory-mapped device registers is both tedious and error prone.

Mackerel is the device definition language of Barrelfish. It is used to generate code for accessing any field of a device in a way that respects special semantics like read-only, or reserved fields. The devices are specified using a syntax that is similar to that used in reference manuals[36].

```
device foo (addr base) "example device with only two registers" {

    register first addr(base, 0x0) "a wild mixture of fields" {
        a 16 ro  "a read-only field";
        b 4  rw  "a read-write field";
        c 4  rw  "another read-write field";
        _ 3  mbz "a few bits that must always be zero";
        d 5  wo  "a write-only field";
    };

    register second add(base, 0x40) "a different size and offset" {
        e 1  ro  "a single read-only bit";
        f 7  rw  "most of a byte";
    };
}
```

Listing 16: A simple example of a Mackerel device specification.

Listing 16 shows a simplified device, consisting of two registers of different sizes. From such descriptions, the Flounder compiler generates a header file containing a set of inline functions. For each register and each field, Flounder generates a function for reading, a function for writing as well as other functions more useful for debugging purposes than driving devices. In a device driver, the programmer can then include that header file and use these functions to read and write individual register fields without having to worry about accidentally overwriting other parts of the register. Having the details of the memory access and bit extraction abstracted in a single function call leads to much cleaner and easier to understand driver code.

It is important to note here that the semantics of Mackerel are actually much more complicated than they may seem at first glance. Mackerel is probably the most complex of all the Barrelfish DSLs, simply because the problem domain is itself rather messy. There is no standard for device register semantics, so manufacturers have used all possible combinations of write-only, read-only, write-to-clear, preserve-to-write and other constraints.[18]

[18]There even are cases of ”write back with two bytes swapped”.

For each field, we need to consider the following values:

- the offset of the register
- the width of the register
- which bits belong to the field
- how each other bit should be written:
    - constant 0
    - constant 1
    - preserve value by reading
    - preserve value without reading

All of this information is necessary to properly write to a field without overwriting the other fields of the register. In order to preserve values without reading them, we need to keep track of previously written values. Each bit in a device register may have different access semantics from the rest. In the worst case, a single write has to be preceded with a read from the register and a read from a shadow register, in order to preserve the values we are trying not to overwrite. In our example device *foo* of listing 16, this is the case for the fields b and c. In order to write to b while preserving the other bits in the register, we first need to read field c, so we know what to write at that place. To also preserve d, we cannot read the previous value from the register itself, since d is not a readable field. Instead we have to remember what was last written to d, which can be done by keeping a shadow copy of that field in regular memory, which we update every time we write to the register.

Because device access is not tied to any other aspect of the Barrelfish ecosystem, we had much more freedom in our design of the Mackerel Rust backend than we had with Flounder, where compatibility with existing code was crucial. We only had to be compatible with the specification syntax and semantics (and preferably the compiler frontend), but had no restrictions on the generated code. We were therefore a bit surprised to find that Rust itself considerably restricted our design space.

For Rust, we would have liked to make use of language features to model the device more naturally than just as a collection of functions. After all, the prospect of high-level language features with low-level access on demand was one of our big motivations for using Rust in systems code in the first place. The natural way to model a device would be a data structure that syntactically shared the same logical layout – a device consisting of registers, which in turn consist of fields of varying sizes. Logically, each device is a tree, with the register fields as the leaves. But expressing this tree in an efficient way is surprisingly hard. Nesting structs would give us the very natural *device.register.field* syntax, but not the semantics we would want. The inner fields cannot reference the parent struct, but it is the topmost layer that will actually contain important information. The layout of a given device is constant: the offset of fields within a register and of the registers within the device are all relative to the base address of the memory-mapped device, which is the only value not known at compile time. We want a call like *device.register.field* to use *device.base*, but the object *device.register.field* does not have any way to reference its parent. Adding a copy of the base address or a reference to the root into every layer of the tree is extremely inefficient[19]. Representing the device by a data structure that shares its tree-like layout is currently not possible within the language.

An alternative design we considered is that of a single function for reading all fields, taking a field identifier as its argument. This allows us to use a nice *device.read(field)* syntax. In the function body, the field identifier would be converted into the set of associated constants (offset into device, access width, bitmasks and shift amounts) through pattern matching, followed by the actual memory access. Whenever this function is called with a constant field identifier, which is by far the most common use case, the call would then be inlined, and the pattern matching resolved at compile time. This approach also allows us to have a very human-readable representation of the device layout in the definition of the field identifiers, separate from any implementation details. However, we run into the problem of what the return type of such a *read* function should be. Register fields vary greatly in size, from single bits to multiple bytes, including everything in between. To be generic over all registers, this *read* function would have to return the widest possible type. But this just moves the burden to the user, requiring explicit casts in every call. In C, having functions use bigger types than the calling code is fine (and very common in systems code), but in Rust the casting is not optional, so the discrepancies between the types is very visible.

With the insight that the return type really is just another constant property of a field, we can do what is very common in Rust when the limitations of functions are too restrictive. A macro has many of the properties of functions, such as scoped names and recursive calls, but is not bound to evaluate to a single type. By replacing the single *read* function from before with a *read* macro, we can have it resolve to an expression with the appropriate type for each field. We can even use the token matching rules for macros to choose the calling syntax. For example, we could make all invocations look like *read!(device =>field)*. As it turns out, while this approach solves the issues we had with the function call design, it also introduces its own problem. Macros are not language items like variables or functions, so they are not part of the namespacing system. All macro names are global to a crate, and any macros imported from external sources must be imported at the crate root[20]. This is in strong contrast to the overall design of Rust, which emphasizes modular code structuring. Using macros for Mackerel would prevent us from having the *read* macro be named something simple (like *read*), because a driver managing two devices would see a conflict in the two definitions it imported.

[19]As an example, the comparatively simple pc16550d UART device consists of 39 fields across 13 registers. Storing a redundant address for each of them — either the base of the device or the root of the nested struct — will therefore require 52 times more memory than necessary

[20]The base of the module hierarchy of a project. In applications, this is typically where *main* is defined.

In light of these serious problems with macros, we decided to fall back to using the same approach as our C code after all. We wrote a new Mackerel backend that generates one Rust library per device, consisting of a set of accessor methods for the various hardware registers. While this works just fine, it was disappointing to have to resort to this old approach. However, once the cross-library macro support has improved, this design can easily be extended to also support that model.

It may seem strange that after so many years, there still is no systems language with good built-in support for memory-mapped devices. But it should be remembered that it is actually a complex problem. Partial support, such as a *volatile* modifier or a bitfield type simply is not enough to express the actual semantics we are trying to achieve. A device definition in Mackerel closely resembles the way devices are described in technical reference manuals and data sheets. All the complexity of the Mackerel language comes directly from the fact that real devices are complex. But Mackerel also shows the power of abstraction for very low-level operations. Having the low-level details of memory accesses abstracted away makes the actually interesting code (namely the device driver) easier to read. Adding native support for so much semantic nuance into a general-purpose language is not very attractive to compiler writers — and understandably so, considering that it is not of much use outside the field of device drivers. But having more fine-grained control over the memory accesses to a data structure would simplify the work of many systems programmers.

4.3 Other DSLs

Barrelfish also has domain specific languages for other use cases, such as specifying the structure of the capability system and error definitions. These are not as important for our purposes, and we deemed them not necessary to port to Rust at this point. The exact structure of the capability type system, defined using the *Hamlet* language, is very important to the kernel, but in user space we mostly just need the ability to invoke them through cabability references. For this use case, the mappings of the various invocation commands to their representation in system calls is much more important than the size or layout of the capabilities themselves.

Similarly, the error codes defined using the *Fugu* language are ubiquitous in libbarrelfish, but for our purposes we really only need to differentiate between success and failure. In Rust, the distinction between success and failure is normally expressed using the *Result* type. This can hold either a success value, or an error value, and these can be of different types. Conveniently, the Barrelfish error value type is a valid choice for the error type in a *Result*, so errors returned by libbarrelfish functions can easily be forwarded to the user in Rust code.

Both *Hamlet* and *Fugu* are implemented using a language for specifying languages called *FiletOFish*[10] (itself a DSL), which makes heavy use of the assumption that the intended target language is C. While adding a Rust backend to the languages implemented in pure Haskell was straightforward, adding Rust to FiletOFish would probably be more work than writing entirely new compilers for the DSLs we want to use. Since for both languages, we are only really interested in the enumerations used by the C libraries, we can much more easily extract those from the generated C code than from the actual DSLs[21].

[21]Since we want to be compatible with the libraries we link to, writing Rust backends for these DSLs would not actually be useful if we ended up with different representations for the enumerations.

4.4 Build system integration

4.4.1 Compiler Versions

The Rust compiler comes in three flavours: *stable*, *beta* and *nightly*. The beta and stable releases only allow the use of stable features [22]. These are the parts of the standard library that have been deemed complete, and are very unlikely to change between releases. Unstable features are only enabled on the nightly builds, and additionally require an explicit opt-in using the *feature* attribute in the code.

Since we are recompiling the standard library, we require unstable features. We therefore have no choice but to use a nightly build. While the stable branch is released in clearly defined steps, there is no such thing as "the corresponding nightly build". The stable release largely consists of the state of the nightly build of a few days before, but may contain backported fixes that entered the nightly build in the intervening days. This is quite inconvenient for our purposes, as we cannot just use "nightly 1.7", but rather "nightly a few days before stable 1.7 was released". To alleviate this problem, we used a tool called *multirust*[23], which allowed us to manage multiple Rust installations in parallel (so we can have both a stable and a nightly compiler on the same machine). It also allowed us to specify the exact version of the nightly build to be used.

Some of the features we use for our own libraries, such as inline assembly, will probably be stabilized in the near future. Their API has not actually changed between versions in a long time. Others features, like linking to *libcore* directly without *libstd*, started out as unstable but have already been stabilized during the writing of this thesis. But recompiling *libstd* and its dependencies will always depend on unstable interfaces such as compiler builtins[24]. This means that systems projects that do not use the standard library, such as OS kernels, will be feasible on the stable release in the future, but porting the whole language to a new system will always require a nightly build.

4.4.2 Hake and Cargo

In Barrelfish, the rules for building all programs are managed by a custom build system called Hake[35]. Its core purpose is to transform a set of Hake rule definitions into *make* recipes. Each directory in the Barrelfish source tree may contain a Hakefile, consisting of a Haskell expression that evaluates to a set of Hake rules. Hake finds all Hakefiles in the source tree, and evaluates them in the context of a set of built-in rules, generating as its output one big Makefile containing all recipes.

Rust uses its own custom build system called *Cargo*, which is used for managing dependencies, compiling projects and even executing them. At the time of writing, Cargo is still in its infancy, and cannot be integrated into other build systems easily. By invoking the Rust compiler directly instead, we get

[22]The term "feature" is used here in a very broad sense. It might simply be a type definition like Box, or a language mechanism like extensions to the pattern matching syntax.

[23] `http://github.com/brson/multirust`

[24]And — ironically — the stability attribute used to mark APIs as stable, which is itself an unstable feature.

full access to all its options, including partial compilation with a separate link step.

In the Rust ecosystem, everything is expected to be versioned by default. Crates can specify the exact versions of their dependencies, and Rust object files can only be linked against the exact versions of the libraries that were used during the compilation step. Through the transitive nature of dependencies, a single application may end up being linked against two different versions of the same library, with name mangling ensuring that the two copies will not conflict with each other. Barrelfish on the other hand currently does not use versioning. Building two different versions of the same library and linking against both is not supported by the default Hake configurations. However, it can be implemented on top of Hake by adding subdirectories for each version and then linking against each explicitly.

In order to compile Rust programs for Barrelfish, we extended Hake. We added additional fields to the *Args*[25] type, so that we can specify source files, compiler options and library dependencies of Rust programs in the same way as we do for C. We added basic rules for compiling individual files, and extended the commonly used *application* rule to support Rust files. We also added rules for generating and linking against the Rust libraries from our new Mackerel and Flounder backends.

4.4.3 Dependency Files

One of the difficulties of integrating Rust into another build system is the generation of dependency files. These files contain simple *make* rules, listing the dependencies between compiler output and source files. They are used to find out if a given object file needs to be recompiled. These dependency files are very important when compiling Rust. To compile a crate, only the root source file is given to the compiler as an argument. The compiler then finds all module files referenced in the code automatically.

The Rust compiler does have an option to generate a dependency file, but in stark contrast to other compilers, it will always generate rules for the current invocation. To generate rules for an object file, we have to actually generate that object file in the same compiler invocation that we generate the dependency file. While this is fine for capturing the intent of dependency files — finding out if a recompile is necessary — it does not integrate well into Makefiles.

A Makefile can include other files that contain rules, such as these dependency files. If a file to include does not exist, *make* will try to generate it if it knows the appropriate recipe. For C files, this means that before the actual target is built, all included dependency files are read — and, if necessary, built. For Rust code, the only possible recipe for generating a dependency file is to compile the corresponding crate. This leads to the unfortunate situation that the first *make* invocation will start by compiling every Rust program in the entire source tree.

Our solution to this problem consists of two parts. All Rust dependency files are included using the weak include directive instead of the regular include. This way *make* will still try to generate them, but skip them if there is

[25] A list of all files, libraries and options required to compile an application.

no recipe instead of aborting. Additionally, the recipes that build the dependency files do not list them as part of their output, essentially making them a byproduct of compiling Rust code. This way, crates will be recompiled when any of their source files change, but the dependency files will only exist if the crate has been compiled before. This solution required changes to the core of Hake. We added a new *HRule* that translates into the weak include command. Additionally, the rules for compiling Rust code had to be adapted to use the compiler flag to emit dependency files, and do so without treating that dependency file as an output of that rule.

All in all, the problems with integrating Rust into existing build systems all seem to come from the fact that both rustc and cargo are still very young. As more people integrate Rust into existing projects, these tools will presumably be extended to provide the functionality they are currently lacking, especially since it is largely a matter of adding more compiler flags for simple but niche use cases.

4.5 Writing a System Service

In order to evaluate the usefulness of Rust in systems programming, we wrote a system service. In Barrelfish, much of the functionality that is traditionally associated with kernels is implemented in user space programs. This notably includes device drivers. If a program wants to print text on the terminal, or request user input, it does so by connecting to the terminal server. This server contains a device driver that manages the serial interface using Mackerel bindings. For ease of use, there is a helper library that manages the multiple Flounder interfaces that the terminal server provides. We wrote a drop-in replacement for the existing terminal server using Rust. In doing so, as well as from writing other Rust programs, we noticed practical differences between C and Rust that we expect to be relevant for many systems programming projects.

Idiomatic Rust code generally encapsulates every resource in an object. The lifetime of the object represents the duration of access to that resource[26]. This is very useful for resources that can actually exist in multiple instances, such as buffers of memory or files. However, in device drivers and system services, we often deal with resources that are unique. The C implementation of the terminal server uses static variables to represent the device and the state of the active client. In Rust, there are static variables, but mutable shared state is heavily discouraged by the language. The purpose of these limitations on mutable shared state is to facilitate multithreaded programming, but they also affect single-threaded applications.

In C, a very common idiom is that of the *initializer function*, a function that needs to be called before the system can be used. In Rust, uninitialized variables are strictly forbidden. Every declaration also requires an initial assignment, and static variables need to be initialized with constant expressions evaluated at compile time. [27]

These factors combined make it surprisingly difficult to adapt a simple C program into equivalent Rust code. It seemed to be much easier to rewrite the Rust version from scratch, without following the code layout of the C version.

[26]This pattern is also commonly known as RAII: resource acquisition is initialization.

[27]While there is an unsafe function to leave a value uninitialized, it is not a constant expression, so it cannot be used for the pattern used in C.

5 Evaluation

We successfully ported Rust to the Barrelfish operating system. Our port has already been used by another student project to run *timely dataflow*[28] on Barrelfish. Timely dataflow is a stream processing framework written entirely in Rust. It has been ported to run on Barrelfish, and is using our Flounder bindings to communicate between workers in a multiprocessor environment.

We implemented a drop-in replacement for the serial driver and terminal server using our Mackerel bindings to control the hardware.

Integrating Rust into an existing C project turned out not to be as straightforward as the FFI might make it look. Because Rust does not parse C header files, the interface descriptions therein have to be duplicated in Rust, which provides new opportunities for errors. While calling C functions from Rust really is as simple as any other function call, C libraries may have APIs that are not compatible with Rust's memory model.

Rust applications are significantly bigger than equivalent C programs. Our terminal server is 18MB big, whereas the C terminal server it replaces only takes up 6.5MB[29]. Part of that difference can be explained by the fact that we link our Rust programs against more libraries. All Barrelfish applications are linked against libbarrelfish, which in turn depends on libc. C programs can use any libc functions used internally by libbarrelfish, without adding to the set of linked-to functions. Rust programs using equivalent functions provided by libstd will end up linking to both the C and the Rust version. But this explanation can not account for a factor of 3 in code size. Inspecting the object code produced by the Rust compiler, we were able to see two trends:

padded functions Many functions contain conspicuous blocks of nop-instructions. This is because the compiler puts the cleanup code for the stack unwinding (the *landing pad*) inside the corresponding function body. The nop-instructions serve to align the landing pad to a 16-byte boundary.

inline functions Rust handles inlining differently from C. Functions to be inlined are marked with the *inline* attribute, but are still regular items. It is therefore possible to create a function pointer to an inline function, for which case the compiler needs to also emit a non-inlined version of said function. Arithmetic operations for the various numeric types are implemented as inline functions in libcore, and our binaries seem to contain non-inlined versions for all of them. A quick search reveals over 1000 uninlined functions of the *num* namespace alone, including such gems as *num::i32.Zero::zero*. Presumably, the compiler is unable to prove that these functions will never be used, most likely because of trait objects.

This shows that so-called *zero-cost abstractions* are not entirely free. Having padded functions decreases the locality of the code, leading to more misses in the instruction cache. The uninlined functions should not incur any performance penalties, provided that they are actually unused. They do however increase the size of the loaded binary in memory considerably, which may

[28] `http://github.com/frankmcsherry/timely-dataflow`

[29] Smaller example programs show very similar sizes — even an empty main function still leads to 18MB binaries.

be problematic in very resource-constrained embedded systems. It is unclear if future versions of the Rust compiler will be able to remove the uninlined functions, as having primitive types implement traits appears to be a deeply-ingrained design choice[30].

We see one of the biggest dangers to Rust's future as a systems language in the difficulty to adapt it. After all, the strictness of the language serves a very practical purpose: that it should not be possible to write code that violates memory safety, or exhibits either data races or undefined behaviour, without explicitly using unsafe blocks. This fundamentally limits the design space of language extensions. The restrictions on shared state to prevent data races are especially problematic, as they universally apply to all code, even code that the systems programmer knows will never be executed in a concurrent setting.

[30] Just the signed 32-bit integer type *i32* alone implements 80 traits.

6 Conclusion

In this thesis, we investigated the suitability of Rust for systems programming.

We ported the standard libraries to the Barrelfish operating system, and were able to run Rust applications, such as *timely dataflow* on Barrelfish. We integrated Rust into the Barrelfish ecosystem, by extending the Hake, Flounder and Mackerel languages. This allowed us to use Rust in very Barrelfish-specific applications, such as system services and device drivers.

We identified several shortcomings in the current design of Rust:

panic handling The panic handling and recovery mechanism is not very useful for systems code, as any anticipated errors are better served with the *Result* type. But there is currently no way to opt out of the automatic stack unwinding, so we must take care to catch all panics at the foreign function interface.

limited containers The various container types that manage heap memory are not compatible with pointers to memory allocated by C code. In order to make use of the automatic memory management of Rust, we therefore have to copy any heap memory we receive from C code into new containers.

macro name resolution Because macros are not items, and resolved before the compiler resolves item names, all macros are part of a single namespace, independent of a project's structure. This leads to macros needing special treatment from programmers, and severely limits the reusability this otherwise very useful system.

References

[1] Rust RFC 1561: Macro naming and modularisation, . URL `https://github.com/rust-lang/rfcs/pull/1561`. Accessed: 2016-04-30.

[2] Rust RFC 1328: Allow a custom panic handler, . URL `https://github.com/rust-lang/rfcs/pull/1328`. Accessed: 2016-04-30.

[3] Rust RFC 1444: Union, . URL `https://github.com/rust-lang/rfcs/pull/1444`. Accessed: 2016-04-30.

[4] The Rustonomicon: The Dark Arts of Advanced and Unsafe Rust Programming. URL `http://doc.rust-lang.org/nightly/nomicon`. Accessed: 2016-04-30.

[5] Thomas E Anderson, Brian N Bershad, Edward D Lazowska, and Henry M Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. *ACM Transactions on Computer Systems (TOCS)*, 10(1):53–79, 1992.

[6] Andrew Baumann. Barrelfish Technical Note 011: Inter-Dispatcher Communication in Barrelfish. Technical report, ETH Zurich, 2011.

[7] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The multikernel: a new OS architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 29–44. ACM, 2009.

[8] Eric A Brewer, Jeremy Condit, Bill McCloskey, and Feng Zhou. Thirty Years Is Long Enough: Getting Beyond C. In *HotOS*, 2005.

[9] BL Clark and JJ Horning. Reflections on a language designed to write an operating system. In *ACM SIGPLAN Notices*, volume 8, pages 52–56. ACM, 1973.

[10] Pierre-Evariste Dagand, Andrew Baumann, and Timothy Roscoe. Filet-o-Fish: practical and dependable domain-specific languages for OS development. *ACM SIGOPS Operating Systems Review*, 43(4):35–39, 2010.

[11] Philip Derrin, Kevin Elphinstone, Gerwin Klein, David Cock, and Manuel MT Chakravarty. Running the manual: An approach to high-assurance microkernel development. In *Proceedings of the 2006 ACM SIGPLAN workshop on Haskell*, pages 60–71. ACM, 2006.

[12] Iavor S Diatchki, Thomas Hallgren, Mark P Jones, Rebekah Leslie, and Andrew Tolmach. Writing systems software in a functional language: an experience report. In *Proceedings of the 4th workshop on Programming languages and operating systems*, page 1. ACM, 2007.

[13] Chucky M Ellison and Grigore Rosu. Defining the undefinedness of C. 2012.

[14] Kevin Elphinstone and Gernot Heiser. From L3 to seL4 what have we learnt in 20 years of L4 microkernels? In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 133–150. ACM, 2013.

[15] M Anton Ertl. What every compiler writer should know about programmers or "Optimization" based on undefined behaviour hurts performance. In *Kolloquium Programmiersprachen und Grundlagen der Programmierung (KPS 2015)*, 2015.

[16] Matthew Flatt, Robert Bruce Findler, Shriram Krishnamurthi, and Matthias Felleisen. Programming languages as operating systems (or revenge of the son of the Lisp machine). In *ACM SIGPLAN Notices*, volume 34, pages 138–147. ACM, 1999.

[17] Richard D Greenblatt, Thomas F Knight, John T Holloway, and David A Moon. A Lisp machine. In *ACM SIGIR Forum*, volume 15, pages 137–138. ACM, 1980.

[18] Thomas Hallgren, Mark P Jones, Rebekah Leslie, and Andrew Tolmach. A principled approach to operating system construction in Haskell. In *ACM SIGPLAN Notices*, volume 40, pages 116–128. ACM, 2005.

[19] Per Brinch Hansen. *Joyce—a programming language for distributed systems*. Springer, 1987.

[20] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Jean Wolter, and Sebastian Schönberg. The Performance of μ-kernel-based Systems. *SIGOPS Oper. Syst. Rev.*, 31(5):66–77, October 1997. ISSN 0163-5980. doi: 10.1145/269005.266660. URL `http://doi.acm.org/10.1145/269005.266660`.

[21] Eric Holk, Milinda Pathirage, Anamika Chauhan, Andrew Lumsdaine, and Nicholas D Matsakis. Gpu programming in Rust: Implementing high-level abstractions in a systems-level language. In *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International*, pages 315–324. IEEE, 2013.

[22] Brian W Kernighan. *Why Pascal is not my favorite programming language*. Bell Laboratories. Computing Science, 1981.

[23] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 207–220. ACM, 2009.

[24] Chris Lattner. What Every C Programmer Should Know About Undefined Behavior, 2011. URL `http://blog.llvm.org/2011/05/what-every-c-programmer-should-know.html`. Accessed: 2016-04-30.

[25] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86. IEEE, 2004.

[26] Amit Levy, Michael P Andersen, Bradford Campbell, David Culler, Prabal Dutta, Branden Ghena, Philip Levis, and Pat Pannuto. Ownership is theft: experiences building an embedded OS in Rust. In *Proceedings of the 8th Workshop on Programming Languages and Operating Systems*, pages 21–26. ACM, 2015.

[27] Jochen Liedtke. Improving IPC by kernel design. In *ACM SIGOPS Operating Systems Review*, volume 27, pages 175–188. ACM, 1994.

[28] Alex Light. Reenix: Implementing a Unix-Like Operating System in Rust. 2015.

[29] TS Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, Timothy Bourke, Sean Seefried, Carmen Lewis, Xin Gao, and Gary Klein. seL4: from general purpose to a proof of information flow enforcement. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 415–429. IEEE, 2013.

[30] Martynas Pumputis. Message Passing for Programming Languages and Operating Systems. Master's thesis, ETH Zurich, 2015.

[31] David D Redell, Yogen K Dalal, Thomas R Horsley, Hugh C Lauer, William C Lynch, Paul R McJones, Hal G Murray, and Stephen C Purcell. Pilot: An operating system for a personal computer. *Communications of the ACM*, 23(2):81–92, 1980.

[32] Eric Reed. Patina: A formalization of the Rust programming language. *University of Washington, Department of Computer Science and Engineering, Tech. Rep. UW-CSE-15-03-02*, 2015.

[33] Martin Richards. BCPL: A tool for compiler writing and system programming. In *Proceedings of the May 14-16, 1969, spring joint computer conference*, pages 557–566. ACM, 1969.

[34] Dennis M Ritchie. The development of the c language. *ACM SIGPLAN Notices*, 28(3):201–208, 1993.

[35] Timothy Roscoe. Barrelfish Technical Note 003: Hake. Technical report, ETH Zurich, 2010.

[36] Timothy Roscoe. Barrelfish Technical Note 002: Mackerel User Guide. Technical report, ETH Zurich, 2013.

[37] Jean E Sammet. Brief survey of languages used for systems implementation. *ACM SIGPLAN Notices*, 6(9):1–19, 1971.

[38] Andrew Scull. Hephaestus: a rust runtime for a distributed operating system. 2015.

[39] Jonathan Shapiro. Programming language challenges in systems codes: why systems programmers still use C, and what to do about it. In *Proceedings of the 3rd workshop on Programming languages and operating systems: linguistic support for modern operating systems*, page 9. ACM, 2006.

[40] Xi Wang, Haogang Chen, Alvin Cheung, Zhihao Jia, Nickolai Zeldovich, and M. Frans Kaashoek. Undefined Behavior: What Happened to My Code? In *Proceedings of the Asia-Pacific Workshop on Systems*, APSYS '12, pages 9:1–9:7, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1669-9. doi: 10.1145/2349896.2349905. URL `http://hdl.handle.net/1721.1/86949`.

[41] Niklaus Wirth and Jürg Gutknecht. The Oberon System. *Software: Practice and Experience*, 19(9):857–893, 1989.

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

OS Development in Rust

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Foellmi	Claudio

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Hüttwilen 1.5.2016	[illegible]

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.