Transactional Memory in Rust

Contents

Acknowledgments

I would like to thank my thesis advisor Eddie Kohler for his excellent guidance, unwavering support, and infinite capacity to put up with my nonsense. And also for making measure performance.

1

Introduction

In this thesis I present `sto-rs`, a port of the STO[15] library from C++ to Rust. STO (Software Transactional Objects) is a new design for a software transactional memory system by Herman et al. STO enables higher performance than previous systems by allowing datatype authors to use any concurrency control mechanism within their data structures and eliminate transaction conflicts caused by logically commutable operations on the data structures.

To use STO, dataytpe authors implement a small set of methods on STO-aware data structures for use by the STO commit protocol.

In porting STO to Rust, I show that many of the mistakes users could make when programming with STO in C++ can be statically prevented using Rust's unique type system. I also show that Rust's type system is flexible enough to express STO's internal data structures and object relationships with only a small amount of well encapsulated unsafe Rust, which guarantees to users that the core system is free of memory errors.

In the rest of the introduction I first describe what transactional memory is and why it useful. I then introduce the parts of Rust's type system that are pertinent to this thesis. I finish the introduction with an outline of the rest of the thesis.

1.1 Transactional Memory

Transactional memory is a concurrency primitive with many advantages over the more traditional and mainstream mutual exclusion lock. Locks are hard to reason correctly about because they do not compose; two threads that run correctly on their own may deadlock when run in parallel. To avoid deadlocks, programmers need to ensure that locks are acquired in the same order everywhere, but this can further complicate algorithms and data structures that are complicated to begin with. As a result, programmers can easily write incorrect parallel code, but it is notoriously difficult to write and maintain correct parallel

code using locks[28]. In contrast, transactional memory is a composable concurrency primitive that makes deadlocking impossible and removes the need to reason about lock ordering.

Using transactional memory, programmers place sections of code that need to be executed atomically into transactions. An entire transaction is either committed atomically, so that its operations do not appear to interleave with operations from other transactions, or aborted. Aborted transactions have no effect on the state of the system, so it is safe to retry them any number of times. Just as with locking, multiple threads trying to access the same data in a transactional memory system can reduce performance due to high contention, but a transactional memory system can always make forward progress by choosing some transaction to commit, so it is not possible for a transactional memory system to deadlock.

Although lock-based systems also have no deadlocks when they are constructed properly, it is much harder to reason about and construct a correct system using locks than it is using transactional memory. Locks protect specific memory locations and must be held for the duration of any operations that depend on or write those locations. This is very simple when there is a single lock that protects all shared data, but such global locks can come with unacceptable performance hits. To increase performance, programmers can use fine-grain locking, in which there are more locks that each protect smaller amounts of memory, but this requires algorithms to correctly acquire and release more locks in the correct global

order. When Dijkstra introduced locking in 1968[6], he called deadlocks "deadly embraces" and mentioned that a proof by induction was necessary to prove the absence of deadlocks in a system. Transactional memory removes this burden from the programmer.

The advantages of transactional memory over locks are not just theoretical. In 2011 Pankratius and Adl-Tabatai ran an experiment at Intel in which six teams each wrote an application to satisfy the same requirements over fifteen weeks[22]. Half of the teams used transactional memory and the other half used traditional locking. During the experiment, the teams using transactional memory spent less time debugging for correctness and produced more easily understandable code, but also spent more time debugging performance issues.

Another study was published by Rossbach, Hofmann, and Witchel in 2010 in which 237 students were surveyed after completing a programming assignment using course-grained locks, fine-grained locks, and a transactional memory library[25]. The students found programming with the transactional memory library to be only slightly easier than with fine-grained locks, but their error rates were also far lower with transactional memory: 10% had errors with transactional memory while 70% had errors with locks. The authors blame the perceived difficulty of using transactional memory on the heavyweight syntax of the Java libraries they were using, so altogether these are very promising results for the potential of transactional memory to make parallel programming easier and more productive.

When Herlihy and Moss first introduced transactional memory in 1993, they envisioned

it as a hardware feature implemented in the processor's cache coherency protocol and exposed to users via new hardware instructions[13]. Indeed, recent Intel processors support Intel's Transactional Synchronization Extensions (TSX)[17], but hardware implementations have practical issues that make them unfit for language- or library-level primitives. First, hardware implementations are not scalable; if a transaction reads or writes too many memory locations, the transaction will be aborted even if there are no conflicts with other cores. Intel's specification also allows transactions to be aborted under other circumstances, such as the execution of particular instructions. These limitations mean that hardware transactional memory does not have the property that some transaction can always be committed, so it cannot provide the forward progress guarantees that make transactional memory so attractive. Although hardware transactional memory instructions could be emitted by compilers that reason fully about how they are used, they are not suitable as a synchronization primitive for most users of high-level languages.

Due to the limitations of traditional hardware transactional memory, researchers have proposed a spectrum of alternative systems, from new architectural implementations that store data structures for unbounded transactions in memory[1], to systems that use a hardware interface but depend on software to overcome hardware limitations[23], to hybrid transactional memory systems that use hardware transactions if available but otherwise fall back on software solutions[3]. On the other end of the spectrum is purely software transactional

memory, which was introduced in 1995 by Shavit and Touitou[27] to make transactional memory available for programmers without any hardware support at all. STO, the system implemented in this thesis, and other related software transactional memory systems are described more fully in chapter 2.

1.2 Rust

Rust[21,26] is a new systems programming language with a focus on both safety and performance. Rust achieves memory safety without the use of a garbage collector by incorporating object and reference lifetimes into its type system. Each object in Rust is uniquely owned by a variable binding or another object, and objects are automatically destroyed when they not owned by any other object and they go out of scope. Rust also allows for taking references to objects and statically ensures that an object cannot be moved or destroyed while there are outstanding references to it.

References in Rust come in two varieties: mutable and immutable. Taking a reference to an object is called *borrowing* the object. There can be only one mutable reference to an object at a time or any number of immutable references. At no point can a program have multiple mutable references to an object or both mutable and immutable references to an object simultaneously. This restriction may seem onerous, especially in a single-threaded context, but it brings important safety benefits. For example, it is impossible in Rust for

an iterator into a collection to be invalidated by a modification of that collection, since the existence of the iterator's reference into the collection precludes the existence of a mutable reference to the collection outside of the iterator's control. It is impossible even for the owner of an object to modify it while there exists a reference to the object.

Each reference type in Rust is parameterized by a *lifetime*, which allows the compiler to ensure that references do not outlive its referents. Often this lifetime can be inferred from context and can be syntactically elided, but in other cases the lifetime must be explicitly provided as part of a reference type or as a type parameter for a struct or function. Lifetimes can have subtype relationships; `'a: 'b` means that lifetime `'a` contains lifetime `'b`, and therefore can be used wherever a lifetime of `'b` is expected. As we will see in chapter 3, lifetimes are very important for ensuring memory safety in `sto-rs`.

Sometimes it is necessary to allow mutation of data through an immutable reference. This is called *interior mutability* and is made possible in Rust by the `UnsafeCell<T>` type, which is a zero-cost wrapper around an object of type `T`. `UnsafeCell<T>` has a single function, `get`, which takes an immutable reference to the `UnsafeCell<T>` and returns a mutable raw pointer to its underlying data. Since raw pointers in Rust do not contain any lifetime information, it is up to the programmer to ensure that they point to valid, live data. To signal that such an invariant must be met by the programmer, the act of dereferencing a raw pointer must be contained inside an `unsafe` block. `sto-rs` makes extensive but well

encapsulated use of `UnsafeCell<T>` for reasons explained in chapter 3.

In addition to enforcing memory safety via ownership and reference lifetimes, Rust also enforces the absence of data races in multithreaded code. Normally Rust's referencing rules would be sufficient to ensure the absence of data races, but the presence of interior mutability makes these rules insufficient. Rust solves this problem by using its trait system. A trait is like an interface; it declares the functions that must be defined for any type implementing the trait. But traits can also be used as type bounds for generics. For instance, a function declared

```
fn foo<T: Bar + Baz>(x: T)
```

can take as its argument any type `T` that implements both the `Bar` and `Baz` traits. The Rust standard library defines two marker traits, i.e. traits that contain no functions, called `Send` and `Sync` for the purpose of ensuring thread safety.

`Send` is implemented for any type that is safe to move from one thread to another, and `Sync` is implemented for any type that is safe to share between threads. Since these traits are so fundamental to the language, they are automatically implemented for any type composed of `Send` and `Sync` types. However, types containing `UnsafeCell<T>` do not automatically implement `Send` or `Sync` because the compiler cannot know whether the programmer implemented them in a thread-safe way. `sto-rs` implements `Send` explicitly for its most fundamental datatypes, and section 3.2 shows that it does indeed prevent data races

as the language requires.

The rest of the thesis is organized as follows: chapter 2 describes related work in transactional memory and mechanisms for parallelism in Rust, section 3.1 describes the STO protocol and its Rust implementation in detail, section 3.2 describes the implementation of STO data structures in Rust, using `TVal` and `TArray` as representative examples, and chapter 4 concludes.

2

Related Work

Transactional memory has been a popular topic of research since its introduction, and this thesis builds on a large amount of existing work in transactional memory and other concurrency mechanisms. This chapter discusses three large areas of related work: general software transactional memory systems, systems that improve transactional performance or ease of use by taking advantage of datatype-specific information, and other mechanisms for con-

trolling parallelism in Rust.

2.1 Software Transactional Memory Systems

Software transactional memory systems take many forms, from software libraries, to syntax extensions with new runtime systems, to binary rewriting, and there have been many efforts to create transactional memory systems for specific languages. For example, Harris and Fraser augmented the Java syntax and runtime to incorporate transactional memory directly into the language[10], the Clojure language has built-in support for transactional memory[2], and Intel's C++ compiler has support for transactional memory intrinsics[17]. Each of these efforts tries to reduce the friction of using software transactional memory by including it explicitly in the language design. In contrast, `sto-rs` is a software library and the Rust compiler has no special knowledge of how it works.

Besides ergonomics, the advantage of having compiler support for transactional memory is that the transactional memory system can integrate with the language runtime to provide capabilities that a library could not provide. For example, Harris and Fraser's work in Java allows for transactional accesses to any object without having to modify or wrap the objects in any way. A library could not provide this capability because it would have no way to intercept direct field accesses to objects. In their survey of software transactional memory systems for Scala, Goodman et al. show that transactional memory libraries have

a large number of usability issues, but that their advantage is that they can be used with unmodified runtimes and toolchains and therefore can be seamlessly introduced to existing codebases[8].

In Rust, it might be possible to use procedural macros and compiler plugins to implement more invasive transactional memory systems, but this work goes in a different direction and uses Rust's type system to address some of the usability issues present in the original STO library. This use of a type system to ensure transactional correctness is more similar in spirit to Harris, Marlow, Jones, and Herlihy's work integrating software transactional memory into Haskell[11]. In both this thesis and the Haskell work, values are wrapped in transactional types before they can be accessed by transactions. To prevent non-transactional object accesses from occurring during transactions, the Haskell work defines an `STM` monad, and only allows pure computations and operations on that monad to occur inside a transaction. This thesis also uses transactional wrapper types, and uses Rust's type system to provide similar guarantees, but cannot provide the full transactional safety of the Haskell system. For example, `sto-rs` cannot prevent the programmer from doing I/O from within a transaction.

Other work in software transactional memory has focused on improving the design of the transaction system itself and is independent of implementation language. One example that is especially salient for this thesis is Dice, Shalev, and Shavit's Transactional Lock-

ing II (TL2)[4]. TL2's transactional commit protocol was the first to provide *opacity*[20,9], the property that an inconsistent program state will never be observed, even in a transaction that will eventually be aborted. In a system like `sto-rs` in which it is possible for non-transactional accesses to occur inside a transaction, it is important to provide opacity to prevent inconsistent program state from leaking out of aborted transactions. The structure of the STO commit protocol, which this work uses unchanged, is also heavily inspired by the TL2 protocol.

2.2 Type-aware Transactions

STO is part of another line of research in transactional memory that takes advantage of data structure semantics to make transactional memory either easier to use or faster. This line of research began in 1988 when Weihl used the commutativity of operations on data structures to define abstract locks that allowed commutable operations to happen simultaneously[29], an idea that was formalized and extended by Kulkarni et al. in 2011[19]. The STO protocol allows such locking schemes to be used transactionally, and although none of the datatypes implemented as part of this work use locking internally, datatype authors writing new STO-aware data structures would be able to use Kulkarni's methods to trade off between parallelism and lock overhead.

Another method of exploiting datatype semantics is *transactional boosting*, introduced by

Herlihy and Koskinen in 2008[12]. Transactional boosting is a method for taking any linearizable[14] datatype with reversible operations and a commutativity specification and making it usable transactionally. Transactional boosting acquires commutativity-based locks and applies operations during the transaction. In the event of an abort, the inverses of all modifications are applied to roll back the transaction. This work is complementary to `sto-rs` because it offers a way to mechanically create STO-aware data structures from pre-existing parallel data structures, which could make integration of `sto-rs` into existing codebases much easier. It would be interesting to see transactional boosting modified for use with `sto-rs` and implemented in a Rust procedural macro.

Recent work by Dickerson, Gazillo, Koskinen, and Herlihy introduces a framework called Proust for lowering commutativity-aware transaction conflict rules to simple memory reads and writes[5]. These reads and writes produce the desired transaction conflicts in any transactional memory system without the system itself needing to be aware of datatype semantics. Proust has the same goal as STO: to increase transactional performance by taking datatype commutativity rules into account while allowing programmers flexibility in datatype design, but Proust wrappers are complicated by the fact that Proust tries to also generalize over the underlying transactional memory system. This thesis is specifically concerned with implementing the STO protocol in Rust, and since there is no widely used transactional memory system in Rust already, it would not have made sense to use Proust as

a starting point for an investigation of transactional memory in Rust.

Of course, STO itself is another entry in this line of work, and is highly relevant to this thesis because `sto-rs` is a reimplementation of the STO library. However, further details of how STO works will be left to chapter 3, where the Rust implementation of transactions and the STO commit protocol is described in detail.

2.3 Rust Parallelism

As this thesis is an exploration of a new concurrency control mechanism in Rust, it seems appropriate to survey existing concurrency control mechanisms as well. Many libraries have been written to make concurrency easy and safe in Rust. These libraries are not the result of traditional academic work, but it is important to have a sense of them to appreciate how transactional memory fits into the Rust landscape.

First, there is the traditional `Mutex<T>` type found in the Rust standard library. Like `std::mutex` in C++, Rust's `Mutex<T>` returns a lock guard that uses the RAII (resource acquisition is initialization) pattern to automatically unlock the lock when the guard goes out of scope. Unlike with C++'s lock, however, it is impossible to access the data protected by Rust's lock without first acquiring the lock if it is shared between threads.

`Mutex<T>` can enforce this invariant because it owns the data of type `T` that it protects, and the only way to get a reference to that data given an immutable reference to the `Mu-`

`tex<T>` is by acquiring the lock. Even then, the reference to the inner data is owned by the lock guard, ensuring that the data can only be accessed as long as the lock is held. A reference to the `Mutex<T>`'s inner data can be had without first acquiring the lock if the caller has a mutable reference to the `Mutex<T>`, since it is only possible to have such a mutable reference if the `Mutex<T>` is not shared between threads. The tradeoff for this increased safety is that Rust's lock is less general than C++'s lock. While a C++ mutex can be used to protect arbitrary data, even data physically split across multiple other data structures, Rust's mutex can only be used to protect data that it directly owns.

Just like `Mutex<T>`, `sto-rs` uses wrapper types to control access to inner data, although the invariant that `sto-rs` enforces is that non-transactional modifications to data may only happen when the data is not shared among threads and that transactional modifications of data may only happen inside transactions.

Another synchronization primitive in the Rust standard library is the multiple producer, single consumer channel. Channels allow for passing ownership of objects from one thread to another, and come in both synchronous and asynchronous varieties. On their own, channels solve the problem of preventing data races by not sharing objects at all. Similarly, the Rust library Rayon[24] provides easy parallelism by using parallel iterators to automatically send objects to worker threads and collect the results back on the original thread.

These message passing mechanisms have a long history that began with Hoare's Commu-

nicating Sequential Processes (CSP) language[16]. Google's Go language is a modern mainstream descendent of CSP, and its goroutines (lightweight user-level threads) and channels are a large selling point because they provide for easy and safe parallelism[7]. However, Rust is not as opinionated as Go about whether it is better to use channels or shared data for parallelism, and allows for sending reference counted references to shared data along channels. Rust's channels are therefore complementary to `sto-rs` because they can be used to share references to transactional data structures between threads, while `sto-rs` then allows transactional computations to be done over those data structures.

Internally, each of these Rust synchronization mechanisms is implemented with `unsafe` Rust code, which can do things like dereference raw pointers and create uninitialized memory. However, each mechanism hides these unsafe implementation details beneath a safe interface that is a contract to the programmer that the mechanism will not break Rust's memory safety or introduce data races. Recent work by Jung et al. formally proves the correctness of many Rust mechanisms depending on `unsafe` code, both in the standard library and in the wider Rust ecosystem[18]. Although this thesis does not include a formal proof of correctness for `sto-rs`, it does aim to justify all uses of `unsafe` code and convince the reader that Rust's safety guarantees are upheld.

3

Implementation

In this chapter I describe the implementation of `sto-rs`. Section 3.1 describes STO's transaction object and the interface it exposes to STO-aware data structures, the interface those data structures must support to work with the STO commit protocol, and the way users actually run transactions. Section 3.2 discusses the construction of STO-aware data structures that statically disallow non-transactional modifications while they are shared between

threads, using a simple wrapper type as an example.

3.1 STO Protocol

At a high level, STO works by providing a transaction object to the code running inside a transaction and by requiring transactional data structures to support a set of callbacks. The transaction object exposes an interface that STO-aware data structures use to record reads and writes to their logical segments. Transactions and logical segments of data structures are associated with version numbers, which monotonically increase as transactions are committed. The version numbers are used to detect both read-write and write-write conflicts between transactions. At the end of a transaction, if it has not already been aborted, STO uses the data structure callbacks to lock modified logical segments, verify that no concurrent transactions have invalidated any reads, install updates, and perform any necessary cleanup.

In addition to the basic version number checking, STO supports evaluating arbitrary predicates to detect conflicts. This extra expressiveness can be used to allow more complicated commutative operations to be committed simultaneously and can improve throughput in highly contentious workloads. However, since this is an advanced feature of STO, it is not included in this work.

The type of the transaction object and the methods it provides for use by datatype au-

```
struct TItem<'a> {
    obj: &'a TObject,
    read: Option<u64>,
    write: Option<Box<Any>>,
}
struct Transaction<'a> {
    tracking_set: UnsafeCell<HashMap<(usize, u64), TItem<'a>>>,
    version: Version,
    err: UnsafeCell<Option<Error>>,
}
impl<'a> Transaction<'a> {
    pub fn get_write<T: TContents>(
        &self, obj: &'a TObject, key: u64) -> Option<T> {...}
    pub fn log_read(
        &self, obj: &'a TObject, key: u64, version: u64) {...}
    pub fn log_write<T: 'static>(
        &self, obj: &'a TObject, key: u64, val: Box<Any>) {...}
    pub fn abort(&self, err: Error) {...}
    ...
}
```

Listing 1: The types of the transaction transaction tracking set entries and transaction objects.

thors is given in listing 1. Whenever a user performs a writing operation on a data structure, the data structure is responsible for calling `log_write` to ensure the write is reflected in the transaction's tracking set. Reading operations first check the tracking set, using the address of the data structure as a `usize` integer and the logical segment identifier as the key, to see if the current transaction has previously written to the given logical segment of the given transactional object. If there was no previous write, the read operation does the lookup in the data structure and calls `log_read` to record the version number of the data it read. This version number record is later used during the transaction commit to ensure that the

read has not been invalidated.

The tracking set entries need to contain references to the data structures they correspond to, so `TItem<'a>` contains a reference to some type that implements the `TObject` trait (shown in listing 2) that all STO-aware data structures implement. It would suffice to store a raw pointer to the data structure or to simply cast the `usize` used in the tracking set key back into a pointer at commit time, but using a reference has important benefits. Since `TItem<'a>` contains a reference, it must be parameterized by the reference's lifetime, `'a`, and since `Transaction<'a>` contains objects of type `TItem<'a>`, it also must be parameterized with a lifetime. This lifetime parameterization of the transaction object means that transactional data structures accessed in a transaction must outlive that transaction's `Transaction<'a>` object; trying to create or destroy transactional data inside a transaction will prevent the program from compiling. Statically checking for these errors, which would lead to use after free bugs, is something that `sto-rs` does that the original C++ implementation cannot do.

Another point of difference between C++ STO and `sto-rs` is the way users run transactions. In C++, STO provides macros to mark the beginning and end of transactional blocks. Those macros are lowered to become a do-while loop that retries the transactional code as long as it aborts by throwing an exception and a retry condition supplied by the user evaluates to `true`. In the Rust implementation, users instead call the `Transaction::run`

```
trait TObject {
    fn lock(&self, key: u64) -> bool;
    fn version(&self, key: u64) -> u64;
    fn install(&self, key: u64, val: &Any, new_version: u64) -> bool;
    fn unlock(&self, key: u64);
    fn cleanup(&self, key: u64);
}
```

Listing 2: The TObject trait that every STO-aware data structure must implement. STO uses these callbacks to communicate with the data structure during transaction commit. The `key` argument identifies a logical segment of the data structure.

```
impl<'a> Transaction<'a> {
    pub fn run<F, G, R>(trans: F, retry: G) -> Result<R, Error>
    where
        F: Fn(&Self) -> Result<R, Error>,
        G: Fn() -> bool,
    { ... }
    ...
}
```

Listing 3: The signature of `Transaction::run`, which is the function users call to run a transaction.

function shown in listing 3. This function takes the transactional code as a closure that takes a reference to a `Transaction<'a>` and returns a `Result<R, Error>` for some user-specified return type `R`.

`Result<T, E>` is a standard Rust sum type of two variants, `Ok(T)` and `Err(E)`, that is used as the return type of fallible operations. Rust provides syntax sugar so that `Result` can be used in a way that emulates exceptions in other languages, but without any stack unwinding. `Error` is a sum type defined in `sto-rs` that has different variants for each different reason a transaction could be aborted. Overall, a return type of `Result<R, Error>`

```rust
trait TContents: Clone + Send + 'static {}
impl<T: Clone + Send + 'static> TContents for T {}
```

Listing 4: A marker trait implemented for any type that is safe to store within a transactional data structure.

for transactions lets users optionally return the result of some transactional computation and also provides a mechanism for the user to determine whether a transaction was committed.

The choice of closure trait to use in the signature of `Transaction::run` has large implications for transactional safety in `sto-rs`. Rust closures can implement three different traits: `FnOnce`, `FnMut`, and `Fn`. Each trait corresponds to how the closure captures. In particular, closures implementing `Fn` are only allowed to have immutable references to their environment. By using `Fn` closures in `Transaction::run`, the ability of transactions to non-transactionally modify their environment is greatly reduced. In `sto-rs` the only way to perform a non-transactional modification of program state from inside a transaction is to use interior mutability, which is relatively rare. This property lets `sto-rs` statically catch many user mistakes that would be allowed in C++ STO.

3.2 Data Structures

In addition to reducing the ability of transactions to non-transactionally modify their environment, it would also be nice if data structures in `sto-rs` could prevent non-transactional accesses to themselves while they were shared between threads in much the same way Rust's

```
struct Val<T: TContents> {
    val: UnsafeCell<T>,
    version: Version,
}
impl<T: TContents> Val<T> {
    pub fn new(val: T) -> Self {...}
    pub fn get(&self) -> T {...}
    pub fn set(&mut self, val: T) {...}
}
unsafe impl<T: TContents> Send for Val<T> {}
```

Listing 5: A simple transactional wrapper type that can hold. It is explicitly marked safe to move to other threads.

`Mutex<T>` prevents access to its data when it is shared between threads. This section shows how to achieve this property, using a simple data wrapper type as an example.

Listing 5 gives the definition of `Val<T>`, the simplest possible STO-aware type. Like `Mutex<T>`, `Val<T>` simply wraps an object of type `T`. The data type `T` is bound by a marker trait given in listing 4 that is defined for any type that can be cloned, can be safely sent to other threads, and does not contain references to any non-static data. The requirement that the inner data be cloneable is because transactionally read data needs to be copied out of the `Val<T>`; having a reference to the data inside the `Val<T>` could lead to undetectable opacity violations. The other two conditions are necessary for it to be safe to share the inner data between threads at all.

`Val<T>` presents a simple non-transactional interface that allows for reading and writing the value it contains, and it is legal to send a `Val<T>` to another thread, but not to share a reference to `Val<T>` between threads, as it has no synchronization. In order to use a

```rust
struct SharedVal<T: TContents> {
    inner: Arc<Val<T>>,
}
impl<T: TContents> SharedVal<T> {
    pub fn new(val: Val<T>) -> Self {...}
    pub fn get<'a>(
        &'a self, transaction: &Transaction<'a>) -> Result<T, Error> {...}
    pub fn set<'a>(
        &'a self, transaction: &Transaction<'a>, val: T) {...}
    pub fn try_unwrap(self) -> Result<Val<T>, SharedVal<T>> {...}
}
unsafe impl<T: TContents> Send for SharedVal<T> {}
```

Listing 6: An atomically reference counted `Val<T>` that can only be accessed transactionally

`Val<T>` transactionally, it is necessary to create a `SharedVal<T>`, shown in listing 6.

A `SharedVal<T>` is simply an atomically reference counted `Val<T>`, but the act of creating a `SharedVal<T>` consumes the `Val<T>`, making it impossible to either read or write the inner value without going through the `SharedVal<T>` transactional interface. The reason a new type is needed here instead of just using an atomically reference counted `Val<T>`, i.e. a `Arc<Val<T>>`, is that `Arc<Val<T>>` would allow for non-transactional reads of the inner value. `SharedVal<T>` needs to explicitly implement `Send` because `Arc<T>` only implements `Send` for `T` that implement `Sync`. It is safe to implement `Send` for `SharedVal<T>` because `SharedVal<T>` prevents data races on its inner data by only offering a transactional interface to access it.

Although `SharedVal<T>` fully prevents non-transactional access to its inner data, it is not a completely satisfying solution. `SharedVal<T>` has the same methods as the original

```rust
struct TVal<'t, 'a: 't, T: TContents> {
    transaction: &'t Transaction<'a>,
    inner: &'a SharedVal<T>,
}
impl<'t, 'a: 't, T: TContents> TVal<'t, 'a, T> {
    pub fn new(val: &'a SharedVal<T>, trans: &'t Transaction<'a>) -> Self {...}
    pub fn get(&self) -> Result<T, Error> {...}
    pub fn set(&self, val: T) {...}
}
```

Listing 7: A wrapper for a `SharedVal` with methods arguments that match those of `Val`.

`Val<T>`, but they each take an extra transaction argument. At its best, this extra argument is only a small ergonomic hiccup, but at its worst it prevents the shared version of the original data structure from implementing traits that are critical for its normal use, such as the `Index` trait that provides the subscript operator for array-like structures. To fix this usability issue, one final type, `TVal` is introduced.

`TVal`, shown in listing 7, is simply a wrapper around a reference to a `SharedVal<T>` and a `Transaction<'a>`. It is parameterized by two lifetimes: `'t`, the lifetime of the reference to the transaction, and `'a`, the lifetime of the reference to the `SharedVal<T>` and the lifetime for which all transactionally accessed structures must be alive. Thankfully, the Rust compiler infers all the necessary lifetime parameters when a `TVal` is created, so all these lifetimes are not an ergonomic issue for the end user. `TVal` has all the same methods as a `SharedVal<T>`, except instead of taking the transaction object as an argument, the `TVal` simply uses its internal transaction reference. Due to the `'t` lifetime parameter, it is impos-

sible for a `TVal` to outlive a transaction, so its inner transaction reference is always valid and a reference to the correct transaction object. Because this third level of type does not take a transaction argument to its methods, it can provide transactional implementations of any traits the original data structure implemented.

Although this example dealt with a very simple data structure, the same patterns can be usefully applied to any other data structure. Rust's transferable ownership allows datatype authors to guarantee that any non-transactional accesses occur when a data structure is not shared between threads and that every access to a shared data structure is transactional. This pattern was also used successfully in the implementation of a transactional array type that was composed of `Val<T>`s. Since all of the array data was already inside `Val<T>` cells, the array implementation did not need to implement `TObject` or interface with a transaction object itself. However, by defining `Array<T>`, `SharedArray<T>`, and `TArray` types composed of `Val<T>`, `SharedVal<T>`, and `TVal` cells, it was easy to get the same transactional safety guarantees.

Although I was not able to measure the performance of the final implementation due to time constraints, earlier benchmarks suggested that `sto-rs` performed within five times as fast as the original implementation. It is likely that with further tuning this difference could be dramatically decreased.

4

Conclusion

In this thesis I showed that Rust's type system could be used to increase the transactional safety of STO and catch many mistakes at compile time that could not be caught in the original C++ implementation. By using Rust's lifetime parameters in the type of the transaction object, I was able to ensure that transactional data structures could not be created or destroyed inside of transactions, which would otherwise be a source of subtle bugs that a

programmer may not be aware of. By using Rust's `Fn` closure type in the signature of the transaction running function, I was able to substantially reduce the number of ways a programmer could accidentally modify program state non-transactionally from inside a transaction. And by using Rust's transferable ownership, I was able to construct wrapper types for transactional data structures to ensure that shared data is only accessible transactionally. None of these improvements in safety would have been possible in C++.

The attractive safety properties of `sto-rs` suggest that transactional memory could be a useful addition to the Rust ecosystem. Transactional memory's guarantee of the absence of deadlocks complements Rust's guarantee of the absence of data races, and is a step toward making parallelism in Rust even easier to use.

References

[1] Ananian, C. S., Asanovic, K., Kuszmaul, B. C., Leiserson, C. E., and Lie, S. Unbounded transactional memory. *IEEE Micro 26*, 1 (Jan 2006), 59–69.

[2] Clojure - refs and transactions. https://clojure.org/reference/refs. Accessed: March 29, 2018.

[3] Damron, P., Fedorova, A., Lev, Y., Luchangco, V., Moir, M., and Nussbaum, D. Hybrid transactional memory. *SIGPLAN Not. 41*, 11 (Oct. 2006), 336–346.

[4] Dice, D., Shalev, O., and Shavit, N. Transactional locking ii. In *Proceedings of the 20th International Conference on Distributed Computing* (Berlin, Heidelberg, 2006), DISC'06, Springer-Verlag, pp. 194–208.

[5] Dickerson, T. D., Gazzillo, P., Koskinen, E., and Herlihy, M. Proust: A design space for highly-concurrent transactional data structures. *CoRR abs/1702.04866* (2017).

[6] Dijkstra, E. W. The structure of the "THE"-multiprogramming system. *Commun. ACM 11*, 5 (May 1968), 341–346.

[7] The Go programming language. https://golang.org. Accessed: March 16, 2018.

[8] Goodman, D., Khan, B., Khan, S., Luján, M., and Watson, I. Software transactional memories for scala. *J. Parallel Distrib. Comput. 73*, 2 (Feb. 2013), 150–163.

[9] Guerraoui, R., and Kapalka, M. On the correctness of transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New York, NY, USA, 2008), PPoPP '08, ACM, pp. 175–184.

[10] Harris, T., and Fraser, K. Language support for lightweight transactions. *SIGPLAN Not. 49*, 4 (July 2014), 64–78.

[11] Harris, T., Marlow, S., Peyton-Jones, S., and Herlihy, M. Composable memory transactions. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New York, NY, USA, 2005), PPoPP '05, ACM, pp. 48–60.

[12] Herlihy, M., and Koskinen, E. Transactional boosting: A methodology for highly-concurrent transactional objects. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New York, NY, USA, 2008), PPoPP '08, ACM, pp. 207–216.

[13] Herlihy, M., and Moss, J. E. B. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture* (New York, NY, USA, 1993), ISCA '93, ACM, pp. 289–300.

[14] Herlihy, M. P., and Wing, J. M. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst. 12*, 3 (July 1990), 463–492.

[15] Herman, N., Inala, J. P., Huang, Y., Tsai, L., Kohler, E., Liskov, B., and Shrira, L. Type-aware transactions for faster concurrent code. In *Proceedings of the Eleventh European Conference on Computer Systems* (New York, NY, USA, 2016), EuroSys '16, ACM, pp. 31:1–31:16.

[16] Hoare, C. A. R. Communicating sequential processes. *Commun. ACM 21*, 8 (Aug. 1978), 666–677.

[17] Intrinsics for Intel transaction synchronization extensions (Intel TSX). https://software.intel.com/en-us/node/524021. Accessed: March 14, 2018.

[18] Jung, R., Jourdan, J.-H., Krebbers, R., and Dreyer, D. Rustbelt: Securing the foundations of the rust programming language. *Proc. ACM Program. Lang. 2*, POPL (Dec. 2017), 66:1–66:34.

[19] Kulkarni, M., Nguyen, D., Prountzos, D., Sui, X., and Pingali, K. Exploiting the commutativity lattice. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2011), PLDI '11, ACM, pp. 542–555.

[20] Lesani, M., and Palsberg, J. Decomposing opacity. In *Distributed Computing* (Berlin, Heidelberg, 2014), F. Kuhn, Ed., Springer Berlin Heidelberg, pp. 391–405.

[21] Matsakis, N. D., and Klock, II, F. S. The rust language. In *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology* (New York, NY, USA, 2014), HILT '14, ACM, pp. 103–104.

[22] Pankratius, V., and Adl-Tabatabai, A.-R. A study of transactional memory vs. locks in practice. In *Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures* (New York, NY, USA, 2011), SPAA '11, ACM, pp. 43–52.

[23] Rajwar, R., Herlihy, M., and Lai, K. Virtualizing transactional memory. In *Proceedings of the 32Nd Annual International Symposium on Computer Architecture* (Washington, DC, USA, 2005), ISCA '05, IEEE Computer Society, pp. 494–505.

[24] Rayon: A data parallelism library for rust. https://github.com/rayon-rs/rayon. Accessed: March 17, 2018.

[25] Rossbach, C. J., Hofmann, O. S., and Witchel, E. Is transactional programming actually easier? In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New York, NY, USA, 2010), PPoPP '10, ACM, pp. 47–56.

[26] The Rust programming language. https://www.rust-lang.org. Accessed: March 14, 2018.

[27] Shavit, N., and Touitou, D. Software transactional memory. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing* (New York, NY, USA, 1995), PODC '95, ACM, pp. 204–213.

[28] Sutter, H. The trouble with locks. *C/C++ Users J. 23*, 3 (Mar. 2005).

[29] Weihl, W. E. Commutativity-based concurrency control for abstract data types. *IEEE Trans. Comput. 37*, 12 (Dec. 1988), 1488–1505.