MEMORY SAFETY THROUGH BRANDED TYPE INFERENCE

by
EMILY MONTERO

A thesis

Presented to the Department of Computer Science and the Robert D. Clark Honors College in partial fulfillment of the requirements for the degree of Bachelor of Science

November 2025

An Abstract of the Thesis of

Emily Montero for the degree of Bachelor of Science in the Department of Computer Science to be taken December 2025

Title: Memory Safety Through Branded Type Inference

Approved: *Michal Young, Associate Professor Emeritus*

Primary Thesis Advisor

Memory safety is a major concern in software development. Languages such as Rust allow for memory safety without sacrificing performance, by performing analysis at compile time. However, this complicates the process of writing software, and some minor performance sacrifices still remain in certain situations.

This thesis presents PhantomLatch, a programming language designed to leverage previous research to improve on the usability of Rust in particular situations. Languages such as Rust rely on a system of ownership and ”borrowing” references, where mutation requires exclusive access. PhantomLatch builds on this idea using branded types to track a separate object representing access to a reference, like in previous research (Yanovski et al., 2021). In PhantomLatch, these objects are called keys.

The novel aspect of PhantomLatch is the ability for the compiler to infer when keys must be borrowed, by implicitly converting a reference guarded by a key to an unguarded reference. Combined with using the key’s identifier as the relevant brand, this allows for writing programs leveraging the technique with much less boilerplate. This is further expanded with the novel concept of guarded classes, further expanding usability. Finally, PhantomLatch can implicitly create keys in some situations, allowing for a user-friendly software writing experience.

Table of Contents

1 Introduction **4**

1.1 Background 4

1.2 Existing Solutions 5

1.3 Research Question 6

2 Methodology **7**

3 Foundations **9**

3.1 Memory Management 9

3.2 Linear Types 10

3.3 Borrow Checking 11

4 The PhantomLatch programming language **13**

4.1 Overview 13

4.2 Grammar and Syntax 14

4.3 Type System 16

4.4 Borrow Checker 18

5 The PhantomLatch Compiler **22**

6 Evaluation **24**

7 Future Directions **28**

8 Conclusion **29**

9 Appendix A: Supporting Documents **30**

10 Appendix B: Examples of Compiler Errors **31**

11 Bibliography **36**

1 Introduction

1.1 Background

A major factor in developing software is the choice of *programming language*. The programming language influences how a program will be designed and built. Often, a tool called a *compiler* translates from a programming language into code understood by the processor. Theoretically, any programming language can express any computation. However, programming languages vary in ease of use, efficient execution, and protection from various errors.

Security is a pressing concern in software development, and avoiding vulnerabilities is usually a primary concern. One class of problem, called *memory safety errors*, is a major cause of these vulnerabilities. For example, Microsoft found that 70% of security vulnerabilities reported to them were caused by memory safety errors (Thomas, 2019). A memory safety error is an error caused by breaking the rules of memory safety. *Memory safety* refers to the guarantee that a program only accesses and uses memory in allowed ways. A memory safety error occurs when a program accesses memory that it is not allowed to, or memory that doesn't have the expected data present.

Memory safety involves a few essential tasks. The most interesting and difficult of these is memory management, which refers to managing the creation, access, and deletion of objects in a program. In programming, an *object* represents one or more memory cells used to represent a piece of data. Another major concept in memory management is references; a reference is an object that represents the ability to access another object in memory. For example, an item in a list may contain a reference to the next item in the list. References are important in programming because they allow for representing how objects are related to each other.

There are three main ways that memory safety can be achieved. First, the programmer can manually ensure their programs are memory safe. This is unreliable and leads to the aforementioned security vulnerabilities (Thomas, 2019). Second, memory safety can be achieved through tools that run while the program is active, such as garbage collection. However, this technique slows down program execution, which is unacceptable for some use cases such as systems programming (Pirelli, 2023). The third technique is to allow the compiler to ensure memory safety. This can achieve the same performance as manual memory safety, but requires the programming language to

be designed with this in mind, and often creates extra work when writing programs (Grossman et al., 2002; Pirelli, 2023; Racordon et al., 2022).

Programming languages have a feature called a *type system*. A type system is a simple set of rules that assigns a *type* to an object, and enforces rules on objects based on their type. For example, adding two objects makes sense if they are numbers; it does not make sense if they are something unrelated. These rules are ultimately enforced by the compiler. Modern compilers are designed with type systems in mind, and have been designed to provide helpful error messages when type system rules are broken. This is essential because type systems have strict rules that must be consistently enforced; it would be wildly impractical for the programmer to enforce these rules by hand.

Programming languages use their type system as a way to describe rules that are used to ensure objects are used correctly and safely. Thus, type systems are closely connected with memory safety. Programming languages designed to enforce memory safety typically work by creating a relatively complex type system that guarantees memory safety. Many solutions to memory management have been designed with the help of type systems, but they all have various trade-offs that make them unsuitable for some tasks.

1.2 Existing Solutions

One solution to memory management is *region-based* memory management (Grossman et al., 2002). This strategy allows for memory to be requested and used in big blocks called regions. Objects are created inside a region, and are only deleted when the entire region is released. Thus, interconnected objects can safely be created within a single region. One language that uses this strategy is Cyclone (Grossman et al., 2002). However, this technique isn't perfect and can slow down programs or use extra memory in some situations. It also adds slightly more work when writing programs (Grossman et al., 2002).

Another existing solution is *mutable value semantics* (Racordon et al., 2022). With this strategy, objects can be modified, but no references can exist except when moving data between functions. This limitation prevents accessing objects owned by other objects, such as items within a list, without the ability to access the entire parent object (Racordon et al., 2022).

Yet another existing solution exists in *borrow checking* (Crichton et al., 2023; Klabnik &

Nichols, 2018). With this technique, only one reference that can modify an object may exist at once. However, accessing an object's data can be done from multiple locations at once. This allows for accessing data owned by another object, such as items within a list. One popular language that uses this strategy is Rust (Matsakis & Klock, 2014).

However, borrow checking has some limitations. In particular, while the limitation on modifying objects allows for ensuring memory safety, it also prevents certain data structures from being easily represented (Yanovski et al., 2021). One proposed solution to this is to separate the permission to modify an object from the references to an object. Thus, a reference to an object can be held somewhere, and be used to mutate the object only when it is safe to do so (Yanovski et al., 2021).

The permission to mutate the object is represented as a separate object that only exists when compiling the program. This technique is used by an extension to Rust called GhostCell (Yanovski et al., 2021). The technique of associating an object with extra permissions or data that are only used at compile time is called *branded types*. This technique of separating the permissions allows for representing more data types than using Rust alone. However, it necessitates a separate token representing the mutation permission to be passed around. This creates extra work when writing programs, as well as complexity that makes maintaining programs more difficult.

1.3 Research Question

How can a programming language leverage a branded type system to allow for memory management with minimal extra work for the programmer? And how will such a language compare to existing memory management solutions in terms of ease of use and runtime performance?

2 Methodology

Testing a new system for compiler based memory management often requires the programming language to include specific new features. Thus, a programming language must usually be extended, such as in GhostCell (Yanovski et al., 2021), or created, such as in Cyclone (Grossman et al., 2002) or Vault (DeLine & Fähndrich, 2001).

Creating a programming language with only the minimal features required to test a new idea is a common type of project in programming languages research. This allows for seeing if the idea is viable, without being bogged down by all the other details needed to make a programming language practical for real software.

For this project, I developed a minimal systems programming language that uses branded types to enhance memory management. Like in Vault (DeLine & Fähndrich, 2001) or GhostCell (Yanovski et al., 2021), permission to mutate an object is passed through a compile-time object such as a key or token. The language will only contain features relevant to memory management, such as classes and references. The language does not implement features unnecessary to showcase this technique for memory management.

In GhostCell (Yanovski et al., 2021), passing around these keys and tokens imposes an additional burden on the programmer. Ideally, the compiler could completely determine what keys are necessary, and how they are passed around; this is impractical at best. Instead, my language implements inference in some situations, allowing for a flow of keys to be specified by the programmer when more complex behaviors are necessary. To achieve this, I take inspiration from the region inference in Cyclone (Grossman et al., 2002).

This project consists of three main components. The first consists of my programming language, named PhantomLatch. This is where the key based memory management is defined and justified. This first project component includes specifying the components of the language, such as the grammar, type system, and inference rules. It also includes an argument that the language is memory safe.

The second component of the project consists of a compiler that implements this programming language. This compiler uses existing tools for lexing, parsing, and code emission whenever

possible. This project focuses on the type system and inference rules, so there is no reason to build the other compiler components from scratch. Indeed, it is technically a cross compiler as it emits code in the C++ programming language. The compiler is the largest and most time intensive component of the project.

The third component of the project consists of using sample programs to evaluate the previous components. These consist of small programs written in PhantomLatch, which illustrate features of PhantomLatch's memory management. Some of these will have equivalent programs written in Rust, for comparison. The complexity added by the two implementations will be compared, to draw conclusions about the efficacy of PhantomLatch's memory management model.

3 Foundations

3.1 Memory Management

To describe the memory management of PhantomLatch, it is necessary to understand the memory management model it builds off of more clearly. PhantomLatch uses an ownership and borrow checking system of memory management, based on the paradigm present in Rust. Rust uses the borrow checking strategy for memory management, which relates to how references are handled (Crichton et al., 2023; Weiss et al., 2021).

The first major component of this strategy is *ownership*. This refers to the approach to memory management where each object in a program has one canonical location which controls access to it. This strategy is used not just in borrow checking, but has a long history in linear type systems and is closely related to how objects are managed in mutable value semantics (Crichton et al., 2023; Racordon et al., 2022).

For the following discussion, I use the term *field* to refer to a location, describable in a given programming language, that can access or store an object. Since a reference is technically another type of object, a field containing a reference still technically stores and refers to an object. The scope of a field refers to the space in a program where a field is reachable, and mostly applies to variables in functions. In particular, a variable (and thus the field it represents) is said to go out of scope when it is no longer accessible, such as when a function finishes executing.

Memory safety refers to a guarantee that the program will only access permitted memory, and use it in permitted ways. In particular, note that memory safety consists of the following properties:

- every memory write is on a field that's actually allocated at the moment
- every memory read is additionally on one that's actually initialized

Memory management, however, consists not just of the creation and use of objects, but their deletion as well. Thus, programming languages need to respect another property to effectively manage memory:

- The language allows any memory allocation to be deallocated by the code

Note that unlike the memory safety guarantees, there is no guarantee that this always occurs in the programming language. Rather, the guarantee states that the code can be modified to add deallocation, without changing the way memory is allocated or used in the program.

A *class* is a construct in programming languages that allows code to define custom types, which consist of a finite collection of field descriptions, which are usually named. Objects which have a class type contain fields, which can store their own data of the type specified by the class. Classes are used extensively in programming, for data structures such as lists, graphs, and abstractions of concepts like files, web sockets, and user accounts.

To *drop* a field means to invalidate it, and free the memory allocated for the object contained within (Crichton et al., 2023; Weiss et al., 2021). This is the vocabulary established by Rust, so I will use it here for simplicity. This operation is the opposite of initializing a field.

3.2 Linear Types

In the general case, memory safety of a program cannot be guaranteed. An arbitrary program might violate memory safety at an arbitrary point in its runtime, preventing certainty about memory safety; this reduces to the halting problem. Since the halting problem is provably undecidable, the memory safety of some programs is similarly unknowable. But if we restrict the behavior of a program, we can create a set of rules that can guarantee memory safety. This approach of using rules that guarantee a property, at the cost of discarding some unknown valid programs, is often called ”conservative”.

First, consider a language where objects can only be accessed through one field. This requires that the language not allow references. If the objects are restricted to only being used once, and created before use, this results in a *linear type system* (Racordon et al., 2022). This ensures memory safety, since objects must be initialized to access in any way, and the lack of references prevents an object from being accessed after it is used up. Allowing objects to be mutated and read multiple times, but still only reside in one location, is the foundation for *mutable value semantics* (Racordon et al., 2022). Such a system allocates memory for a field when initializing it. Given that all fields contain either a primitive object or an object consisting of fields, this is memory safe. When a field goes out of scope, it can be safely deallocated as it is no longer reachable. In the case of objects,

inner fields go out of scope when the one containing the object does.

However, this lack of aliases prevents the creation of any data structures which require an object to be accessible from more than one location (Klabnik & Nichols, 2018).

3.3 Borrow Checking

Borrow checking, such as in Rust, extends an ownership system to allow for references to be created, by restricting when references are valid and what conditions they need to exist (Crichton et al., 2023, Weiss et al., 2021). While the borrow checking rules in Rust are complicated, the core rules for references are relatively simple:

- A reference may only be taken from an initialized object
- Taking a reference prevents the object from being uninitialized (moved or dropped). This is called borrowing the object. Note that if the object being referenced is in a field, this same restriction must be placed on the field.
- Dropping a reference removes the above restriction, thus allowing
- Mutating an object requires exclusive access to the object; this way, nothing prevents the object or its children from being uninitialized. In particular, no other reference may borrow the object in any way.
- Reading fields of an object does not require exclusive access; however, all references must still be tracked so that they can be returned, hence allowing future mutation.
- Note that read-only objects still require borrows to be returned, because the objects can still be dropped which would invalidate any references.

Thus, Rust uses a system with two types of references: shared and exclusive. Multiple shared references can be borrowed from one field, allowing it to be read from any of them, while exclusive references must be the only borrow on the source field, but allow mutation of the object (Crichton et al., 2023, Weiss et al., 2021). Rust allows references to be borrowed from other references; Crichton et al. (2023) describe this as “stacked borrows” since borrowing from a reference builds upon that

reference having borrowed some previous object, and the borrows must be returned in this same order.

Some special types available in Rust have exclusions from these rules, and use runtime checks and other restrictions to allow mutating their contents with a shared pointer (Klabnik & Nichols, 2018). This pattern is known as *interior mutability*, and a major example is the RefCell type. RefCell is a special class, built into Rust, which allows the object stored inside to be mutated with only a shared reference to the RefCell. To ensure memory safety, RefCell does extra work at runtime, and throws an unrecoverable error if the access is invalid. This slightly slows down programs, and increases the odds of a program failing unexpectedly after creation (Klabnik & Nichols, 2018; Kyriazopoulos, 2024).

The Rust extension GhostCell by Yanovski et al. (2021) extends and loosens these rules to allow some data types to be better represented. In particular, some data types are best represented using circular chains of object access, such as doubly linked lists and graphs (Kyriazopoulos, 2024; Yanovski et al., 2021). The GhostCell extension adds a new type to Rust, also named GhostCell, which provides interior mutability without runtime processing. The way it works is that a GhostCell object is marked with a brand, which uniquely ties it to a single object of a type GhostToken (Yanovski et al., 2021). The GhostToken object must be borrowed in order to access the data within a GhostCell; thus, exclusive access to the interior object can be guaranteed even from shared access to the GhostCell. This can be used in some of the situations where RefCell is normally used, allowing for improved performance and reliability at the cost of code complexity and a limitation where the entire collection must be borrowed to access an item within (Yanovski et al., 2021).

4 The PhantomLatch programming language

4.1 Overview

The first component of my project is an informal specification for a programming language, called PhantomLatch. This consists of specifications for the various components of the programming language, which together provide a description for how to use and implement the language. This is the part of the project where I introduce my somewhat novel strategy to improve the usability of branded types for memory management, building on and inspired by the work in GhostCell (Yanovski et al., 2021), Vault (DeLine & Fähndrich, 2001), and Cyclone (Grossman et al., 2002).

PhantomLatch takes the concept of *guarded types* present in Vault (DeLine & Fähndrich, 2001), and adapts it to a more modern context. Guarded types in Vault are types annotated with the name of a key. This serves as a brand associating the type with a unique object, in this case called a key, which represents a unique name for the guarded resource. This allows references to be tracked by knowing how they relate to the source objects, allowing the compiler to assist with memory management (DeLine & Fähndrich, 2001).

PhantomLatch modernizes guarded types by tying them in with borrow checking. Instead of serving as a unique name for an object, a key in PhantomLatch serves the same purpose as a GhostToken in GhostCell (Yanovski et al., 2021). The benefit here is that unlike GhostCell, the same name can be used for both a token and the associated brand, simplifying source code. Also, this allows keys to be created when initializing an object, like in Vault (DeLine & Fähndrich, 2001), instead of needing to construct them separately like in GhostCell (Yanovski et al., 2021).

The most novel feature of PhantomLatch lies in a feature I call *guarded classes*. Like a guarded type, a guarded class is a class defined with a brand tying it to a key name. However, instead of representing a particular key, this is a generic parameter to the class. In essence, this represents a key that is associated with a particular instance of the class. The way this works is that guarded classes can only be used as guarded types; the key for the type becomes the key associated with the class for that particular object.

An object with a guarded class type thus requires the associated key to be borrowed in order to borrow the object, such as to call a method or mutate a field. Thus, within the context of a method,

the associated key can be accessed and used for other purposes, so long as it isn't being used to access any data on the object. This innovation allows for classes in PhantomLatch to be designed around the use of keys for memory management.

The need to borrow a guarded class object's key to invoke a method means that this operation can be done implicitly, allowing the method to be invoked in the same way as a non-guarded class method. Similarly, PhantomLatch allows a guarded type (of a nonguarded class) to be implicitly used as an equivalent unguarded one, automatically inferring the necessary key borrows. This allows guarded objects to be used the same as unguarded ones in situations like field access, method calls, and borrowing references. This same inference also applies when using references of guarded types. These rules are inspired by how regions are inferred in Cyclone (Grossman et al., 2002), which uses a simple set of rules like this to determine a region for a pointer whenever it isn't specified.

Finally, PhantomLatch contains one more inference rule / implicit operation: when creating a named field of a guarded class type, if no key is specified in the type, it is treated as a guarded type with a local key name identical to the field name. This allows guarded classes to be used similarly to unguarded ones, further hiding implementation details whenever possible.

The name for PhantomLatch refers to the features it is designed to illustrate. PhantomLatch uses a branded type and unique guard system based on GhostCell (Yanovski et al., 2021), thus the name Phantom. Similarly, PhantomLatch uses a concept of keys inspired by Vault (DeLine & Fähndrich, 2001), thus the name Latch.

4.2 Grammar and Syntax

The first component of a programming language is the grammar. This formally specifies how the text of the program is interpreted as various tokens, which combine to produce constructs such as expression and statements. Syntax refers to the structure in which symbols are combined to write programs in a language; it is the main property specified by the grammar.

The grammar of PhantomLatch is specified in the Grammar.g4 file in the supplementary materials. This file is formatted for consumption by the ANTLR4 parser generator (Parr, 2013). For the sake of brevity and consistency, this serves as the formal grammar specification for PhantomLatch. What follows is an overview of the syntax.

The syntax of PhantomLatch is based mainly on that of languages like C++ (ISO, 2024). PhantomLatch ends statements with semicolons, declares variables by starting with the type name, uses brackets for blocks of code, and so on. Here is an illustrative example:

```
int main(){
    int x = 7;
    List y = List();
    y.insert(x);
    return 0;
}
```

The syntax for references in PhantomLatch differs from C++ in that the ampersand & occurs before the type name, instead of after. Also, references can be qualified as mutable. Finally, references must be taken with a reference operator, which is also an ampersand. Example:

```
mut &Foo r = &foo;
```

The syntax for guarded types in PhantomLatch is the same as that of Vault (DeLine & Fähndrich, 2001), with a key name followed by a colon and the type. To create a new key for an object, add the key name followed by a colon to the constructor as well. Example:

```
K:List y = K:List();
```

The syntax for guarded classes is similar; when declaring the class, add a key name and a colon before the class name. Example:

```
class K:Node{
    int value;
    Node(int value){
        this.value = value;
    }
}
```

4.3 Type System

Types in PhantomLatch are split into two fundamental categories: copy types and memory types. Copy types are copied implicitly, and cannot be referenced. Memory types can be referenced, but can only be moved and not copied. PhantomLatch contains two copy types: primitive types, and reference types. PhantomLatch also has two memory types: class types and array types.

- Primitive types consist of simple data that can be copied freely. In particular, the two primitive types provided are integers and booleans.
- Reference types consist of a reference to a memory type. Reference types can be qualified with the mut keyword to mark them as exclusive; otherwise, they are shared.
- Class types are objects which have named fields or methods. Class types are the only ones with a guarded variant. PhantomLatch contains three built-in generic class types; other class types can be defined in code.
- Array types are arbitrary sized lists of another object type, which have a constant size that can be assigned at runtime.

There are three kinds of fields in PhantomLatch, which can contain values of these types.

- Local variables: variables within functions
- Object fields: values inside class objects, specified in the class definition
- Array items: locations within an array, addressable by integer index

One important aspect of memory safety is the initialization of fields. Fields are defined as initialized based on the following conditions:

- Local variables are initialized when a value is assigned to them
- Object fields are considered initialized when the object is constructed, which is when the class constructor returns

- Array items are considered initialized when the array object is created, since array items are initialized upon array allocation

Fields can be uninitialized in some situations:

- Class and array types are uninitialized when the value is moved out
- Class, array, and reference types are uninitialized when they are dropped
- Variables are automatically dropped when they go out of scope.
- Any fields contained in a class or array object are also dropped when the object is dropped.

The three built in class types in PhantomLatch are based on some of those in Rust (Klabnik & Nichols, 2018). The Optional class allows for storing either data or a ”none” value, and the data can be referenced or accessed after ensuring it is present. The SimpleRc class is a simple reference counter, allowing for shared ownership of data. Finally, the Box class allocates space dynamically, which is necessary for recursive data structures. The exact interfaces are as follows:

- `Optional<T>`:
  - `present() → bool`
  - `ref() → &T`
  - `mut_ref() → mut &T`
  - `into_inner() → T`
  - `take() → Optional<T>`
- `SimpleRc<T>`:
  - `ref() → &T`
  - `copy() → Optional<T>`
- `Box<T>`:
  - `ref() → &T`

- `mut_ref()` → `mut &T`
- `into_inner()` → `T`

4.4 Borrow Checker

The borrow checker in PhantomLatch is relatively simple in operation. It enforces a short set of conservative rules for reference borrowing that ensure memory safety. In this case, ”conservative” refers to how these rules guarantee memory safety, at the cost of rejecting an unknown amount of memory safe programs as well. The basic operations concerning the borrow checker are as follows:

- Assigning a primitive value to a field f, including that of an object. This functions by copying the underlying data. Examples:
  - `int x = 7;`
  - `foo.value = 12;`
  - `a[i] = 5;`
- Reading a primitive value from a field f, including that of an object or array. This functions by copying the underlying data. Examples:
  - `int w = x;`
  - `int w = foo.value;`
  - `int w = a[i];`
- Constructing class objects o with a constructor. This allocates memory and the constructor initializes it. Example:
  - `Foo x = Foo(...);`
- Moving memory objects o from a source field f_1 to a destination field f_2. This often functions by copying the underlying data, and invalidates the previous location. Example:

  - Foo y = x;
  - int[] b = a;

- Creating a reference r referring to a memory object o. Examples:
  - `& Foo z = & y;`
  - `& mut Foo z = & mut y;`

- Creating a reference r_2 from a source reference r_1. Example:
  - `& Foo r = z;`

- Assigning a guarded class object $k : o$. Example:
  - `K:Foo x = Foo(...);`

- Creating a key k during construction of a guarded object $k : o$. Example:
  - `K:Foo x = K:Foo(...);`

- Creating a guarded reference $k : r$ referring to a guarded object $k : o$ with key k. Example:
  - `& K:Foo z = & y;`

- Creating a guarded reference $k : r_2$ from a guarded reference $k : r_1$ with key k. Example:
  - `& K:Foo r = z;`

- Creating a reference r_2 from a guarded reference $k : r_1$ with key k. This occurs implicitly when using the reference in a non-guarded context.

- accessing a field f of an object o through a reference r. Example:
  - `int w = r.value;`

- dropping a class, reference, or array field f. Example:
  - `drop foo;`

- allocating an array a and assigning it to field f. Example:
  - `int[] a = memalloc int[12];`
- Accessing the field at index i of an array a. Example:
  - `int x = a[i];`

The borrow checker has a set of rules that it applies, to check whether each basic operation is valid. They are as follows:

- A field f must be initialized before being read or referenced in any way. Initialization is described above in the class info.
- Creating a reference r pointing to some object o or reference r_o creates a borrow on object o be borrowed. This prevents the containing field f_o from being dropped.
  - A borrow may take place from an object or reference, which may be guarded.
  - If field f_o lives within another object p, then object p must be borrowed as well to prevent f_o from being dropped or overwritten. This continues recursively until the field is a local variable.
  - If the borrow is exclusive, no object must currently be borrowing o. For the duration of the exclusive borrow, object o cannot be borrowed again.
- A guarded object $k : o$ is an object o associated with a key k. Borrowing object o requires borrowing key k in the desired manner. This applies to borrowing guarded references $k : r_o$ as unguarded as well. The object or reference is only borrowed in a shared manner.
- Creating an array a of a class type requires that the type have a 0-argument constructor, allowing for item initialization.
- Accessing a field f_i of object o through a reference r requires that r be initialized and unguarded.
- Mutating an object o directly requires o to have 0 borrows.

- Mutating an object o through a reference r requires r to be an exclusive reference.
- Reading a field f_i of object o may occur even if other borrows exist, but f_i may only be mutated if this access is exclusive.
- The body of a function or method F must unborrow every object it borrows, unless it returns a reference r_r or an object o.
  - In this case, all memory type arguments are borrowed by the return value.
  - If F is a method, r_r also borrows the invoking object o
  - These borrows are exclusive if r_r is an exclusive reference, or if an object o is returned.
- All relevant inputs to a function or method, including the invoking object, are borrowed by the returned value upon function return.
- Accessing index i of an array a causes a runtime check for validity. If this is out of range, an unrecoverable error is thrown.
- A method M must end with every class field initialized. This also applies to the class constructor.
- Borrows held by a class object or reference are unborrowed when the object is dropped.
- The state of borrow checking must be merged at the end of branches, such as if and while statements. This means the following:
  - Variable declaration and field initialization must match
  - Borrows from all branches are all combined
    - For example, if two branches borrow reference r from two different objects, r borrows both objects after the branch.
  - Borrow exclusivity must be valid after branches are combined; an object cannot have multiple outstanding exclusive borrows.

5 The PhantomLatch Compiler

The second component of my project is the PhantomLatch compiler that I created to implement the above language specification. This compiler is implemented in the C# programming language (Microsoft Corporation, n.d.). To complete the project in a reasonable amount of time, I used existing tools for all parts of the compiler that aren't relevant to the language's memory management features. In particular, I use the ANTLR4 parser generator (Parr, 2013) for the frontend, and emit C++ source code (ISO, 2024) as a backend instead of lowering my IR to any kind of assembly or object code.

The complete source code for my PhantomLatch compiler can be found in the supporting documents.

Components of the compiler that I created are the grammar, type checker, borrow checker, and code translator (that emits C++ code). I also created a C++ source file implementing the builtin classes for the language and other needed functions for code translation.

Parsing is done by ANTLR4, using the grammar file Grammar.g4. This is discussed further in the language section of this report. The other steps are done by handwritten C# code. The entrypoint to the code, which invokes all the other compiler steps, is found in Program.cs. The parse tree provided by ANTLR4 is converted to a custom structure in ParseTreeVisitor.cs.

Type checking is mainly done by the TypeChecker class found in TypeChecker.cs. This involves building a symbol table, and then verifying that types are valid and consistent in the code. This symbol table also serves as a sort of intermediate representation (IR) for the rest of the compiler. Some type checking for built in classes is also performed in BuiltinsHandler.cs. The symbol table is effectively a stack of dictionaries, which associate identifiers with definitions.

Borrow checking occurs in the file BorrowChecker.cs, and consists of operations performed on an object of class BorrowContext (defined in BorrowContext.cs). This context consists of a dictionary mapping tracked fields to their current state in the borrow checker. The basic flow of data the borrow checker is as follows:

- A function to borrow check each function in the code separately
- A function to borrow check a statement block

- A function to borrow check an expression. This function recursively borrows any needed components.
- A function to validate the guarded state of a source type and a destination type
- A function to have one field borrow another field. This function handles borrowing any keys if necessary
- A function to release a field when it is dropped.
- A function to merge two borrow contexts after a control flow branch ends.
- Many other miscellaneous helper functions

Finally, C++ code is generated in the CodeTranslator.cs file.

The compiler source code can be compiled with the dotnet C# compiler (Microsoft Corporation, n.d.). The compiler binary reads input from a file “data/input.txt”, and outputs C++ code in the “output” directory.

6 Evaluation

One useful consequence of using a static structure, like GhostCell (Yanovski et al., 2021) or the PhantomLatch guarded types, for field access is that any errors are detected at compile time. This prevents the program from crashing at runtime, thus improving reliability of code by rejecting mistakes before producing an executable. PhantomLatch thus encourages these benefits, by building this mechanism directly into the compiler, and allowing it to be leveraged at the class level.

One obvious limitation of my language is the lack of explicit destructors / drop functions. This means that users must remember to perform any operations needed to free memory with a more complex structure, such as reference counted loops. This increases the risk that a user creates a program with memory leaks. However, languages such as C++ and Rust show that supporting such functions in a language is a solved problem. Additionally, this feature is ultimately orthogonal to what is being tested here, and should be fully compatible with PhantomLatch's style of borrow checking.

There are many data structures that require back pointers, and thus require use of a RefCell or GhostCell in Rust. However, the easiest example is probably a doubly linked list. This data structure uses a node class, which has access to the nodes in the forward and back directions. It also contains a list class, which has access to a head node at the front of the list, and a tail node at the back of the list.

To evaluate the utility of PhantomLatch, I now compare a linked list implementation in PhantomLatch to an equivalent implementation in Rust using the GhostCell library.

My doubly linked list implementation in Rust is based on the safe implementation discussed in the book *Learning Rust With Entirely Too Many Linked Lists* (Kyriazopoulos, 2024). I adapted it to fit the GhostCell library, as this is the most relevant comparison I can draw.

My doubly linked list in PhantomLatch is an equivalent implementation to the Rust one, to the greatest extent allowed by PhantomLatch. Due to the minimal nature of features in PhantomLatch, the final code looks somewhat different.

My complete implementations can be found in the "examples" directory in the supporting documents. However, I highlight the push function here, and the same basic principles apply to

the rest of the code. Another notable detail is that the Rust implementation requires creating a GhostToken and brand, which must be passed as a parameter when creating or using the list. The PhantomLatch implementation, by contrast, can be used directly in a function variable with syntax identical to that of a list implementation using runtime checks.

The Rust implementation of push looks as follows:

```
pub fn push(&mut self, elem: i32, token: &mut GhostToken<'brand>) {
    let new_node: Link<'brand> = Node::new(elem);
    match self.head.take() {
        Some(old_head) => {
            old_head.borrow_mut(token).prev = Some(new_node.clone());
            new_node.borrow_mut(token).next = Some(old_head);
            self.head = Some(new_node);
        }
        None => {
            self.tail = Some(new_node.clone());
            self.head = Some(new_node);
        }
    }
}
```

The implementation in PhantomLatch is as follows:

```
void push(int x) mut{
    SimpleRc<K:Node> node = SimpleRc<K:Node>(Node(x));
    Optional<SimpleRc<K:Node>> old_head = head.take();
    if(old_head.present()){
        &K:Node inner = old_head.ref().ref();
        SimpleRc<K:Node> node_copy = node.copy();
        inner.SetPrev(Optional<SimpleRc<K:Node>>(node_copy));
        drop inner;
    }
    else{
```

```
        drop tail;
        tail = Optional<SimpleRc<K:Node>>(node.copy());
    }
    &K:Node n = node.ref();
    n.SetNext(old_head);
    drop n;
    drop head;
    head = Optional<SimpleRc<K:Node>>(node);
}
```

This has more boilerplate in a direct comparison, leaving Rust the victor in terms of implementation. However, the signature and usage of my function is simpler, which is useful since classes are usually used more often than they are created.

However, most of this boilerplate can be eliminated by implementing common language features. The SetPrev and SetNext functions only replace the previous and next values in the nodes. A borrow checker that can track a level of object fields would eliminate this need; the explicit drops could be eliminated.

I did not actually implement these features in the language, but applying those well understood changes would result in the following source code:

```
void push(int x) mut{
    SimpleRc<K:Node> node = SimpleRc<K:Node>(Node(x));
    Optional<SimpleRc<K:Node>> old_head = head.take();
    if(old_head.present()){
        &K:Node inner = old_head.ref().ref();
        SimpleRc<K:Node> node_copy = node.copy();
        inner.prev = Optional<SimpleRc<K:Node>>(node_copy);
    }
    else{
        tail = Optional<SimpleRc<K:Node>>(node.copy());
    }
    &K:Node n = node.ref();
```

```
    n.SetNext(old_head);
    head = Optional<SimpleRc<K:Node>>(node);
}
```

This still doesn't win over the Rust source code, but is almost equally readable and I believe would represent a competitive option.

7 Future Directions

There are many possible future research or utility projects that could build off of this work. This section will list some possible next steps for anyone wanting to build off of this project. These consist of two categories: future research directions, and ideas to develop the language from a research language into a fully usable language.

For research, a logical next step would explore adding generic key parameters, and trying to implement further inference of when keys are passed to functions and methods. A more complex, but probably useful, idea would be to extend the variety of brands used in memory management; for example, unique identifiers similar to the keys present in Vault (DeLine & Fähndrich, 2001) could be used to annotate types with more information about the relationship between references. This might allow for some shape analysis of data structures defined by classes, which might allow for eliminating further runtime checks such as those in the reference counter needed in many data structures.

For developing PhantomLatch, the obvious and critical step is to add clearly missing data types and functions. This includes things like immutable strings, arguments in main, and an IO mechanism. The other obvious features to add are those discussed in the evaluation section. These include type definitions, function fields being implicitly dropped, and explicit destructors for classes.

Another possible development for PhantomLatch would be implementing other brand-based memory management techniques which have already been developed previously. One example is branded integers, indicating bounds checking for a particular array. This would allow for requiring array access to be bounds checked, which would allow for removing the current unrecoverable exception behavior. This could be combined with mechanisms like for loops to make the language both more readable and more performant.

One concept I didn't tackle is recoverable exceptions, since they are not present in Rust (Klabnik & Nichols, 2018). Since the borrowing behavior of functions is reflected in their signatures, it follows that exceptions thrown by a function would be as well. This would limit the propagation of exceptions, and perhaps allow references to be borrowed by exceptions as well.

8 Conclusion

Based on the evaluation using a doubly linked lists, PhantomLatch in its current form is not competitive with existing solutions using GhostCell and Rust for class implementations. However, for consuming the implemented classes, PhantomLatch demonstrates a clear advantage by hiding most or all of the brand related implementation details. Even when implementing classes, the lack of a separate brand and token has value in simplifying code.

I believe PhantomLatch showcases an idea which would have utility in a production language. PhantomLatch, as a research language, is not useful for writing any software in it's current form. The most novel innovation in PhantomLatch is the guarded classes, which combined with inference allow for the smooth user experience when using classes involving keys.

The most time consuming part of this project was definitely implementing the compiler. Even a compiler with few features like this one is an intimidating piece of software to write without a team, and there are a lot of moving pieces which have to work perfectly. The language specification was also slow to write, mostly because of the care needed to ensure that the rules were both internally consistent, and consistent with the memory management strategy of borrow checking.

My research question was split into two parts. The first asked how a branded type system could assist with memory management; PhantomLatch more or less serves as my response to this. The evaluation showed that this approach is clearly usable. The second part of my research question asked how my solution stacks up with existing solutions. The lack of RefCell represents a theoretical performance improvement over Rust, but I was not able to perform any relevant benchmarks for various reasons. Thus, this remains an unproven claim. In terms of usability, GhostCell clearly outcompetes PhantomLatch for class implementation, but PhantomLatch wins for using the classes in other code.

9 Appendix A: Supporting Documents

- Zip Archive: Compiler Source Code
  - C++ code: library/library.cpp
  - C# code: AstNode.cs
  - C# code: BorrowChecker.cs
  - C# code: BorrowContext.cs
  - C# code: BuiltinsHandler.cs
  - C# code: CodeTranslator.cs
  - C# code: CompileResult.cs
  - ANTLR grammar: Grammar.g4
  - C# code: ParseTreeVistitor.cs
  - C# code: Program.cs
  - C# code: SybolEntry.cs
  - C# code: SymbolTable.cs
  - C# project: Thesis.csproj
  - C# solution: Thesis.sln
  - C# code: TypeChecker.cs
  - C# code: VarType.cs
- Zip Archive: examples
  - PhantomLatch code: double_list.txt
  - Rust code: ghost_linked_list.rs

10 Appendix B: Examples of Compiler Errors

In this section, I provide some examples of invalid PhantomLatch code, and the errors it produces in my compiler implementation. This serves as a way to test my compiler implementation, as well as an easy showcase of how PhantomLatch's borrow checker works. For clarity, these examples are each on a new page.

Example #1:

```
class Foo{
    int value;
    Foo(int value){
        this.value = value;
    }
}
int main() {
    Foo foo = Foo(12);
    println(foo.value);
    Box<Foo> box = Box<Foo>(foo);
    println(box.ref().value);

    //should causes an error; foo is uninitialized after being moved
    println(foo.value);

    box.mut_ref().value = 7;
    println(box.ref().value);
    foo = box.into_inner();

    //should causes an error; box is uninitialized after being moved
    println(box.ref().value);

    println(foo.value);
    drop foo;

    return 0;
}
```

Example #1 produces the following errors:

```
Failure with errors:
```

```
CompileError{borrowcheck, source not initialized: foo, at line 14:12}
CompileError{borrowcheck, source not initialized: box, at line 21:12}
```

Example #2:

```
class Foo{
    int value;
    Foo(int value){
        this.value = value;
    }
}

int main() {
    K:Foo foo = K:Foo(12);
    println(foo.value);

    &K:Foo f1 = &foo;
    mut &Foo f2 = f1;

    //illegal, f2 exclusively borrows K
    println(foo.value);

    f2.value = 13;
    println(f2.value);
    drop f2;

    //legal now that f2 is dropped
    println(foo.value);

    drop foo;
    return 0;
}
```

Example #2 produces the following errors:

```
CompileError{borrowcheck, cannot borrow Key:K shared;
```

```
unique borrow outstanding, at line 16:12}
```

11 Bibliography

Crichton, W., Gray, G., & Krishnamurthi, S. (2023). A Grounded Conceptual Model for Ownership Types in Rust. Artifact for "A Grounded Conceptual Model for Ownership Types in Rust," 7(OOPSLA2), 265:1224-265:1252. https://doi.org/10.1145/3622841

DeLine, R., & Fähndrich, M. (2001). Enforcing high-level protocols in low-level software. SIGPLAN Not., 36(5), 59–69. https://doi.org/10.1145/381694.378811

Grossman, D., Morrisett, G., Jim, T., Hicks, M., Wang, Y., & Cheney, J. (2002). Region-based memory management in cyclone. Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation, 282–293. https://doi.org/10.1145/512529.512563

ISO. (2024). ISO/IEC 14882:2024 Programming languages—C++ (Seventh). pub-ISO. https://www.iso.org/standard/83626.html

Klabnik, S., & Nichols, C. (2018). The Rust Programming Language. No Starch Press.

Kyriazopoulos, B. (2024). Learning Rust With Entirely Too Many Linked Lists. https://rust-unofficial.github.io/too-many-lists/

Matsakis, N. D., & Klock, F. S. (2014). The rust language. Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology, 103–104. https://doi.org/10.1145/2663171.2663188

Microsoft Corporation. (n.d.). C# standard specification. https://learn.microsoft.com/en-us/dotnet/csharp/specification/

Parr, T. (2013). The Definitive ANTLR 4 Reference (2nd ed.). Pragmatic Bookshelf.

Pirelli, S. (2023). Safe Low-Level Code without Overhead is Practical. Proceedings of the 45th International Conference on Software Engineering, 2173–2184. https://doi.org/10.1109/ICSE48619.2023.00183

Racordon, D., Shabalin, D., Zheng, D., Abrahams, D., & Saeta, B. (2022). Implementation Strategies for Mutable Value Semantics. The Journal of Object Technology, 21(2), 2:1. https://doi.org/10.5381/jot.2022.21.2.a2

Thomas, G. (2019, July 16). A proactive approach to more secure code. Security Research & Defense. https://msrc.microsoft.com/blog/2019/07/a-proactive-approach-to-more-secure-code/

Weiss, A., Gierczak, O., Patterson, D., & Ahmed, A. (2021). Oxide: The Essence of Rust (No. arXiv:1903.00982). arXiv. https://doi.org/10.48550/arXiv.1903.00982

Yanovski, J., Dang, H.-H., Jung, R., & Dreyer, D. (2021). GhostCell: Separating permissions from data in Rust. GhostCell: Separating Permissions from Data in Rust (Artifact), 5(ICFP), 92:1-92:30. https://doi.org/10.1145/3473597