A LANGUAGE FOR EMBEDDED AND CYBER-PHYSICAL SYSTEMS

A THESIS SUBMITTED TO THE GRADUATE DIVISION OF THE UNIVERSITY OF HAWAIʻI AT MĀNOA IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

IN

COMPUTER SCIENCE

MAY 2016

By

Paul Soulier

Thesis Committee:

Depeng Li, Chairperson
Henri Casanova
Peter Michael Seidel

Keywords: programming languages, embedded systems, memory management, compilers

ABSTRACT

As computers continue to advance, they are becoming more capable of sensing, interacting, and communicating with both the physical and cyber world in significant ways. Medical devices responsible for a person's well-being, electronic braking systems in automotive applications, and industrial control systems are examples of the many Cyber-Physical Systems (CPS) that utilize these computing capabilities. Given the potential consequences of software related failures in such systems, a high degree of safety, security, and reliability is often required.

Programming languages are one of the primary tools used by programmers to develop embedded and cyber-physical systems. They provide a programmer with the ability to transform complex designs into machine executable code. Of equal importance is their ability to help detect and avoid programming mistakes. For decades, embedded and cyber-physical systems have been developed predominantly with the C programming language—in large part, due to its expressive power and ability to program low-level characteristics of these systems that other languages can't. Although a powerful and widely used language, its type and memory unsafe pointers are a common source of programming errors.

Pretzel is a hypothetical programming language that addresses memory safety and type safety issues commonly found in C/C++ pointers while attempting to maintain comparable performance and expressiveness. To achieve this, Pretzel's type system provides a cohesive set of three distinct reference types; each with varying degrees of flexibility and runtime overhead. Additionally, the design also proposes techniques to minimize the performance impact of automatic reference counting.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1
INTRODUCTION

Cyber-physical systems[1] exist at the intersection of computation and the physical world. These systems perceive the world through sensors and affect change through actuators. In a form of feedback, sensors and external inputs influence computation that allows the system to interact with the physical world in a tangible way. CPS exist in various forms, sizes, and complexity including small, stand-alone devices (e.g., sensor nodes or implanted medical device) or embedded as a subcomponent in a larger system such as the fly-by-wire systems of aircraft or hard drives in computers.

CPS have become an intrinsic part of modern society. They can be found in appliances, medical devices, automotive applications, avionics, military weapons, industrial control systems, power grids, and countless other applications. The seemingly inexorable advances in hardware technology have enabled CPS to expand into new domains. Ubiquitous wireless connectivity has made possible the Internet of Things (IoT) where CPS will undoubtedly play a significant role. As new advancements are made in other disciplines (biotech, medicine, robotics), it is easy to envision any number of possible applications where CPS will be an essential component.

Given current and potential future applications of CPS, the ability to create safe, reliable, and secure software for these systems is self-evident. Developing such systems has been a long-standing challenge in computer science and software engineering. Security issues, the IoT, and entities that wish to exploit software flaws with malicious intent have further complicated the landscape of software development for CPS. Software engineering and design methodologies, formal verification, simulation, and various other techniques have been devised to aid in the production of safe and secure software. Programming languages are another such tool. Programming languages can provide a mechanism to reason about complex software systems in a way that can be understood by a computer. A language that can most effectively enable a programmer to accurately transform abstract concepts into code will be more likely to result in systems that operate as expected.

Many application domains have enjoyed benefits from improvements made to programming languages. Modern languages are more capable of detecting errors through their type system and many of the low-level and error prone aspects of programming have been eliminated through various abstractions. This has enabled the development of more reliable and complex applications. Embedded systems are an exception to this. Despite its flaws, the vast majority of embedded systems are still developed using the decades-old C programming language—an inherently unsafe language. The majority of safety issues in C revolve around the incorrect use of pointer types that result from a combination of common programming errors and a weak type system.

[1] The terms "cyber-physical systems, CPS, embedded systems, and low-level programming are used interchangeably throughout this document.

This work introduces the hypothetical language Pretzel. Pretzel is an imperative and object-oriented language that is both type and memory safe. It offers high-level abstractions common in modern programming languages while still permitting the low-level expressiveness necessary to program embedded systems. The primary contribution of Pretzel is its treatment of reference types.

1.1 Characteristics of Embedded and Cyber-Physical Systems

When considering a language designed for low-level systems, it is useful to characterize attributes that make these systems unique. Embedded systems occupy a domain quite different from many other applications. While there is no formal definition for what constitutes an embedded system, they have some specific characteristics that set them apart. Informally, an embedded system is one that is built for a specific purpose, interacts with the physical world in a tangible way, and operates within the bounds of some rigid physical constraints. CPS typically have most, if not all, of the following traits.

Constrained Environment Embedded systems are usually (but not always) built with more limited computational resources than other platforms. Even with the availability of relatively cheap and powerful hardware, it is not uncommon to find embedded systems with only a few kilobytes of memory running on slow, 16-bit (or even 8-bit) processors. Time and power are two additional resources embedded systems can find in short supply. For real-time systems, failure to complete a task within a specific time constraint can be just as fatal as a bad memory reference. Devices that operate in remote environments are powered by batteries with a finite lifetime and power consumption must be managed like any other limited resource.

Hardware Interaction Software alone can't directly interact with its physical surroundings—it needs hardware to do this. Software for embedded systems bridge the gap between the physical and cyber worlds. To do this, embedded system software operates at a very low-level, frequently interacting directly with hardware.

High Reliability Naturally, reliability is a goal for all software. However, there is a significant difference between a crash occurring in a web browser versus the flight control software of a rocket. Embedded systems are often used to control machinery, avionics, medical devices and various other technologies where the consequences of software failures are very high. As such, reliability is often a critical component to many cyber-physical systems.

Difficult to Debug and Upgrade Embedded systems are often found buried inside of larger system, in remote places, and even on martian planets. Diagnosing problems found in the field and updating software can be costly, exceptionally difficult, or even impossible. This makes the development of error-free software all-the-more important for embedded systems.

Purpose-Built Embedded systems are typically designed for a specific purpose. The more functions a system must perform, the more complicated the software becomes. Interactions between disjoints functional areas add complexity and tend to reduce reliability and performance while increasing the necessary computing resources.

1.2 Pretzel's High-Level Design

The C programming language is generally considered as the "gold standard" in terms of developing embedded systems. It is sufficiently expressive to implement software for virtually any hardware platform and offers performance matched by few other languages. Many of these qualities can be attributed to the C's pointer types. Pointers in C have virtually no restriction and provide programmers a great deal of flexibility in developing software. These traits, however, come at the expense of safety where pointers are a common source of programming errors.

Pretzel's goal is to provide the same expressiveness found in C/C++ while additionally ensuring type and memory safety. This is achieved through its reference types. Most languages provide a single, uniform reference type that either obscure important low-level details or expose unsafe semantics that can lead to programming errors. Rather than a single reference[2] type, Pretzel provides three distinct types: static, dynamic, and unique. Type and memory safety is provided through a combination of a static type system, runtime memory management, and omitting unsafe language features (e.g., type casting, C-style unions, etc.).

Static references correspond with objects that have a static lifetime (i.e., stack-allocated and global variables) and memory safety is guaranteed at compile time. Dynamic references correspond to objects with a lifetime that can't be determined at compile time (e.g., dynamically allocated heap memory) and safety is enforced through automatic reference counting (ARC). Unique references have dynamic lifetime, but memory safety and reclamation is determined at compile time. However, an object addressed by a unique reference can't be aliased by other pointers. Individually, and in combination, these methods of implementing pointers have been thoroughly studied. The primary contribution of this work is the cohesive combination of these three reference types in the context of low-level programming. The remainder of this section describes high-level concepts and specific design requirements of Pretzel.

1.2.1 Type and Memory Safety

Central to this work are the concepts of type and memory safety. A language that is type safe guarantees that object in memory is used only in accordance with the defined behavior of that type. Memory safety is the property that a reference may only access the fields of type it addresses and that the memory it refers to is valid. Valid memory is defined as memory that has been allocated

[2]The terms "reference" and "pointer" are used interchangeably.

only for an instance of a specific type and that can't be simultaneously used for another distinct type instance. Invalid memory references occur for various reasons such as referencing an object after it has been freed (a "dangling pointer") or accessing an element of an array by an index that extends passed its defined range.

These concepts are inter-related; a language that is not type safe can't be memory safe and visa versa. For example, if an instance of a type A is accessed as a type B (a type violation), a memory address may be accessed that does not belong to the instance of A causing a memory safety violation. A language that is considered "safe" must be both type and memory safe.

1.2.2 Data Representation

Data representation relates to the manner in which a program organizes and manipulates in-memory data structures. In many cases, managing the details of how memory is allocated and the specific placement of data is a burden that is best managed by the compiler or run-time environment. Embedded systems, on the other hand, tend to care more about these details.

A CPS routinely interfaces directly with hardware or communicates with other devices via well-defined data structure layouts (e.g., hardware register sets) and protocols. A program must have control over the specific layout of data structures down to individual bits. In addition to functional necessity, data representation can influence performance. The organization of a data structure can be tuned to optimize data locality to take advantage of CPU cache memory or optimally "pack" fields in a structure to minimize memory requirements.

1.2.3 Expression

The "expressiveness" of a language is the intangible quality that relates to how easily abstract concepts can be represented by a language's syntax and semantics. It's also somewhat of a subjective characteristic. For example, Python could be regarded as a very expressive language; it's easy to transform concepts into code. However, when taken in the context of embedded systems, the opposite is true; it's extremely difficult to write a Python program for an embedded application.

Related to expression is *expression transparency*—the ability for a programmer to accurately estimate the machine code that a compiler will generate for given statements of code [34]. For example, consider the code in Figure 1.1 (Java and C are mixed). Most C programmers can easily estimate how much memory the array consumes and will have a good idea what code (if any) will be executed. The Java statement is notably more difficult to asses. Expression transparency is a concept that is often disregarded in language design. It is, however, a characteristic of programming languages that is invaluable for embedded systems development. Pretzel attempts to achieves this, in part, through a well-defined object model (presented in Chapter 5).

```
class Point {
    int     x;
    int     y;
};

Point   p[100];                 // C statement
Point[] p = new Point[100];     // Java statement
```

Figure 1.1: Expression Transparency

	Intel	ARM
Platform	iMac (2010)	Raspberry Pi 2
CPU	3.2GHz Core i3	900MHz Cortex-A7
L1 Cache	64KB	16KB
Memory	12 GiB	1 GiB
Compiler	clang-600.0.57	GCC 4.9.2

Table 1.1: Test Platforms

1.2.4 Performance

Pretzel is intended to be usable in place of C and consequently must provide comparable performance. For common language constructs (e.g., conditionals, loops, etc.), the assumption is that these features can be implemented by a compiler in an equivalent manner to C. For elements of Pretzel that differ substantially from C/C++, performance comparisons may be of interest. Since Pretzel is a hypothetical language, there is no compiler and direct performance comparisons between executable code isn't possible. However, a method is still needed to draw any useful conclusions about the design of Pretzel. To make these comparisons, code generation is "simulated" by approximating what a theoretical compiler would generate by using hand-coded C or C++ for a given feature. These results are compared against the analogous feature written in C. Performance tests presented later in this thesis were run against on both an IA64 and ARM processor (see Table 1.1 for test platform details).

Space performance (i.e., the amount of memory a language construct requires) is another measure of performance that is of particular interest to embedded systems. When any specific language feature is used, the associated memory overhead can influence the software design decisions. Chapter 5 provides an object model for Pretzel that defines the underlying implementation of various language features and the associated memory and computational overhead.

1.2.5 Language Requirements

So far we have described general high-level goals for a low-level programming language. The following enumerates the specific design requirements of Pretzel.

1. ***Type and Memory Safe*** Pretzel must guarantee type and memory safety.

2. ***No Runtime*** Many embedded systems are resource constrained. It must be possible to implement the features of Pretzel without the use of a runtime environment such as a virtual machine or OS.

3. ***Optional Dynamic Memory*** Many embedded systems do not have sufficient resources to support a heap-allocated memory. Pretzel must be capable of producing programs without the use of dynamically allocated memory. Specifically, any language feature (aside from those related to dynamic memory) provided by the compiler must be achievable without the use of dynamic memory allocation.

4. ***Explicit Memory Management*** The programmer must have explicit control over when and where memory is allocated and deallocated.

5. ***Separate Compilation*** The ability to compile units of code separately is necessary for all but the smallest systems.

1.3 Thesis Organization

This chapter has described the general problems Pretzel is trying to solve and summarized the primary goals of the language. The remaining chapters introduce background information and the detailed design of Pretzel. The chapters are organized sequentially where later chapters built on information presented in earlier chapters.

The first part of this work is largely introductory material. Memory management, type safety, and memory safety are well-studied topics with a body of work dating back at least to the 1960's. Chapter 2 highlights the related work most relevant to this work as well as short summary of existing languages suited to embedded systems development. Most code examples are presented using Pretzel. Rather than describe the formal grammar of the language (which can be found in Appendix A), Chapter 3 provides a brief introduction to pertinent aspects of the language.

The remaining chapters present the core of this work. Chapter 4 introduces the reference types that form the foundation of Pretzel's type system. The Type Model described in Chapter 5 shows how the references types and other language features could be implemented by a compiler. Pretzel uses Automatic Reference Counting for garbage collection; Chapter 6 gives a couple methods for improving aspects of this garbage collection mechanism. Chapter 7 elaborates on the future direction of this research followed by closing remarks.

CHAPTER 2
BACKGROUND AND RELATED WORKS

In its entirety, the areas of research related to type safety, memory safety, and memory management is expansive and well beyond the scope of this work. The works covered in this chapter are limited to those that can be related to low-level programming. For a more comprehensive view of garbage collection, Wilson [43] gives an extensive survey on the topic. In addition to these topics, a collection of programming languages that address issues related to low-level programming are briefly described.

2.1 Memory Management

Memory management focuses on methods that help provide memory safety and reclaim unused memory (i.e., "garbage") from a running program. Such methods simplify the software engineering process by eliminating the need for a programmer to explicitly free unused memory. Perhaps more importantly, it is an essential component to providing memory safety. By automating garbage collection, a large class of errors related to referencing invalid memory (i.e., dangling pointers) are eliminated. This section reviews techniques for automatic garbage collection.

2.1.1 Tracing Garbage Collection

The origins of tracing garbage collection date back to 1960 where McCarthy [29] described a method to automatically reclaim unused memory in the LISP programming language. Since that time, various derivative versions of this original work have been designed, but they all work on the same basic premise. In tracing garbage collection, in-use memory is identified by *tracing* through memory that can be reached through "root nodes" (i.e., global variables, registers, or variables on the runtime stack.). Memory not reachable from the root nodes is implicitly unused and can be reclaimed for other use. Tracing GC guarantees the reclamation of only unused memory and, when given sufficient memory, can offer very good performance when compared to simple manual memory management. Tracing GC requires a runtime environment to perform the actual collection process. In addition to the runtime, tracing GC typically needs substantially more free memory (typically on the order of 2–5 times the actual memory required by an application [17]) to achieve acceptable performance.

Tracing GC also introduces non-deterministic timing behavior into a program that can pose challenges to certain applications. When a collection cycle begins, a program must generally be suspended for the duration of the process. Furthermore, it isn't possible for a program to know when a garbage collection cycle will begin or how long it will take to complete. As a result, applications may experience short "hick-ups" during normal execution. For real-time systems, the pauses introduced by garbage collection can pose a more serious problem. Tasks that must complete

within a fixed period of time can neither wait for GC to stop nor can they be interrupted to allow a collection cycle to begin. Real-time tracing garbage collectors [26, 2] have been developed such that collection cycles operate within deterministic bounds. Such collectors experience their own set of difficulties such as tuning timing parameters for a given application, performance, etc.

2.1.2 Automatic Reference Counting (ARC)

Automatic reference counting (ARC) was proposed by Collins [4] to address some of the issues in McCarthy's work [29] (namely performance degradation as memory utilization increases). The basic concept of ARC garbage collection is relatively intuitive. Each object has a count associated with it that tracks the number of references to it. Each time a new reference is bound to the object, the count is increased and decremented when a reference is unbound. When the reference count reaches zero, the object is returned to the free pool of memory.

ARC is good for systems where garbage generation and/or reference mutations are low since GC overhead only occurs when references change. The key benefits of ARC are its simplicity and deterministic behavior. ARC is conceptually simple to understand and can be implemented comparatively easily—sometimes without direct support from the programming language or runtime[1]. Because reference counts are tracked on each reference variable mutation, deallocation occurs immediately when an object is no longer in use. This is valuable for real-time systems where non-deterministic memory reclamation can impede the ability to meet timing constraints. The performance of ARC is also relatively consistent regardless of overall heap size.

ARC is typically slower (by about 30% [33]) than its tracing counterpart. This is largely an artifact of the reference count updates that must be performed. To remediate this cost, a number of techniques have been proposed. Deutsch and Bobrow [8] observed that reference count mutations for certain variables (i.e., references that exist in registers or local variables on the stack) do not need to be immediately applied. Their work describes a method of deferring reference counting updates, thus avoiding frequent and unnecessary updates. However, this introduces non-determinant timing to memory reclamation and requires additional runtime support. Bacon et al. [3] improve on this work slightly by removing the need for some of the runtime structures required for processing deferred reference counts. Further performance gains can be found by coalescing reference count updates as described by Levanoni and Petrank [24]. They observed that, for certain patterns, only the first and last reference count mutations of a sequence of operations are important. For example, when traversing a linked list, the only reference counts that need to be modified are for the first object and the last object; the reference counts of the intermediate objects is ultimately unchanged. Joisha [21] employs compile-time analysis to identify and eliminate unnecessary updates.

[1] One such example is the `share_ptr` library provided by C++. By using destructors and overloaded operators, the language provides enough flexibility to implement a reference counted pointer abstraction without any direct language support.

Another significant issue with ARC is the management of cyclic references. A reference cycle occurs when an object either directly or indirectly refers to itself through other objects or references. This results in a self-sustaining condition that prevents reference counts from reaching zero even when the memory is no longer reachable by any variables in a program. There are several approaches to dealing with this issue. A backup tracing collector can be periodically run to collect unused cyclic references [7]. In a similar vein, an ARC garbage collector can search for cycles by leveraging observations about reference counts. With this method, a reference count that is decremented, but does not reach zero, is a candidate for cycle detection. Periodically, those objects are tested by subtracting reference counts induced by interior references. If the count can be reduced to zero, the object is not in use and can be released. This method performs significantly worse than a backup tracing collector [10].

2.1.3 Region-Based Memory Management

Region-based memory management, originally proposed by Tofte and Talpin [39], dynamically allocates objects into regions. The compiler infers the lifetime of regions and automatically inserts points of allocation and deallocation. These points of deallocation are determined statically at compile time by using variable annotations provided by the programmer. Region-based memory management has the advantage that points of allocation and deallocation are determined at compile time; in theory leading to potentially better runtime performance than other garbage collection schemes. Additionally, since the soundness of a program can be verified at compile time, runtime errors are avoided. Region-based memory does not directly address data with lifetimes that do not follow the lexical structure of a program. Also, the necessity of code annotations significantly complicates language semantics.

Gay et al. [11] describe region-based memory management for dynamic memory. Their approach offers both explicit freeing of regions as well as reference counted regions and is dynamically checked at runtime. Grossman et al. [16] detail region-based memory management used in the Cyclone language [20]. Like Tofte and Talpin, their system uses code annotations, but minimizes the complexity by inferring common cases.

2.1.4 Linear Types

Linear types are another interesting method of memory management inspired by Girard's linear logic [13]. With linear types, an object can be referenced by only a single entity. Once that reference ceases to exist, there can be no other references and the object can be released. Linear types require little runtime overhead making them ideally suited for CPS. Although linear types provide guaranteed memory management, they do not allow the sharing of data through aliases. Walker and Watkins [42] examines combining linear types and region based memory management while Wadler [41] applies them to functional languages. Event-driven systems, common in CPS,

often communicate through messages. Fähndrich et al. [9] discuss efficient and safe message-based communication using linear types.

2.2 Type and Memory Safety

Type safety and memory safety are common research topics, but are less common in the context of low-level programming. Much of the work done for low-level programming has focused on correcting unsafe aspects of C through extensions or alternative dialects with varying degrees of compatibility with standard C.

Necula et al. developed the CCured [31] type system for C to enhance memory safety of pointer operations through the use of annotations. The type system adds pointer type qualifiers that facilitate programming idioms common to C while enhancing the safety of the language. These aid in the compiler's ability to statically verify many uses of pointers at compile time. For instances that cannot be checked statically, runtime checks are added to the code. The underlying representation of pointers is determined by the compiler and may vary in size. This presents challenges when interfacing with C libraries built with a standard compiler. This also poses a problem for systems that carefully manage data structure representation. CCured requires a tracing garbage collector which may limit its applicability to some embedded systems.

Deputy, by Condit et al. [5], provides an extension to the C language in the form of dependent types. Using annotations in C code, the programmer specifies parameters such as ranges and boundaries for various types. This enables the compiler to ensure program correctness by performing static compile time analysis and inserting runtime checks where necessary. By using this metadata, Deputy is able to avoiding changing program data structure organization and layout.

Cyclone [20] is a dialect of C that enhances the type system to avoid memory and type errors common in C code. By using additional syntax and type inference, Cyclone is capable of performing static analysis and inserting runtime checks when necessary to ensure memory violations do not occur. The language uses type inference and parametric polymorphism to provide a type-safe alternative to the idiomatic use of "void" pointers that provides a version of subtyping. Cyclone was developed with the explicit intent to preserve the expressive power of C in developing low-level software. This comes at the cost of syntax overhead and variables-sized pointer types.

2.3 Survey of Existing Languages

This section takes a brief look at languages that, in some form, address issues related to low-level programming. Also reviewed are various dialects of C that attempt to correct type or memory safety issues inherent to the language.

Ada [37] is a general-purpose, statically typed language designed for large applications requiring a high degree of reliability. Originally developed in the early 1980's, it has undergone several

revisions that continue to incorporate new language features (the most recent being Ada2012). Ada is well-suited for embedded systems, avionics, medical devices, or any other system based on modern software engineering principles.

The ***C*** programming language [18] was originally developed in the 1970's for the UNIX operating system and was based on concepts from the BCPL and B programming languages [22]. The language itself is relatively small and imposes few restrictions on what a programmer can do. This is both one of its greatest strengths and weaknesses alike. Programmers are free to manipulate memory, override types, etc. in virtually any conceivable way. However, this expressiveness allows for a class of common programming errors that are unchecked by the compiler. C is neither type or memory safe, yet depsite these issues it remains a standard language for low-level software development.

C++ [19] is a superset of the C language that adds object-oriented concepts as well as other modern language features. It is a relatively large and complicated language that offers a great deal of flexibility. An interesting example of its flexibility is *template metaprogramming*. C++ supports generic programming through its template mechanism. Although not originally intended for this purpose [40], templates also provide the capability to generate code and data at compile-time (a feature known as metaprogramming). C++ continues to evolve with subsequent revisions adding various new features. As a superset of C, C++ is also a type and memory unsafe language.

D [1] is a variant of C++, but with the addition of some built-in higher-level features such as garbage collection, associative arrays, etc. Although not strictly type or memory safe, it has some language-based features and programmer annotations that help enforce these qualities.

Rust [28] is intended to be a type, memory, and thread safe language suitable for low-level applications. Its type system is based on concepts from linear programming [41, 9] and Cyclone's region lifetimes [16]. The core of the language revolves around linear references in which an object may be bound to only a single reference. This provides the ability for the compiler to guarantee safety at compile time with very low runtime overhead. However, the use of linear type significantly reduces the expressiveness of the language. For example, since an object can only be referenced by a single pointer, it is not possible to represent some very basic data structures (e.g., doubly-linked lists) or to perform concurrent operations on a single object without reverting to unsafe reference types.

The ***nesC*** [12] language was developed to complement the TinyOS [25] operating system that targets the highly constrained environment of sensor networks. The language is a dialect of C and provides features tailored to event-driven programming common in such systems. The language does not support dynamic memory allocation and requires whole-program compilation. While this allows for very good code optimization and better compile-time error checking, it makes the language less suitable for larger applications.

FORTH [32], originally developed in the 1960's by Charles Moore, is another low-level programming language. The language is somewhat esoteric by today's standards, but very expressive.

It lacks many of the features found in modern languages and is, by design, type and memory unsafe. FORTH can be compiled or used in a "shell" environment—the latter property making it useful for direct hardware interaction by a user. It's a remarkably flexible language given its simplicity and is still used in niche applications where simple and efficient code is required.

Cyclone [20] is a C-like language that addresses many of the type and memory safety issues while maintaining programming idioms common to C (e.g., pointer arithmetic). The language utilizes region-based memory management for ensuring safe memory management. It was not initially designed with concurrency support, but subsequent work by Grossman [15] has been done to incorporate this into the language.

Assembly. A discussion on low-level languages wouldn't be complete without mentioning assembly language. Given the effectiveness of modern optimizing compilers, the extensive use of assembly is less common. However, when access to specific CPU instructions is required or to produce highly-optimized code for special cases, assembly is unavoidable.

CHAPTER 3
PRETZEL OVERVIEW

Pretzel is an imperative, statically-typed language with some object-oriented features. Its syntax and semantics draw from various existing languages, so the meaning of most language constructs should be relatively intuitive. Being a hypothetical language, there's no compiler implementation and, as such, an in-depth description of the syntax[1] and semantics of the language wouldn't serve much practical value. Instead, an informal overview of the language is given with a particular focus on elements of the language that use reference types.

Pretzel, like Java or C, is a block-structured language. All code resides in a block that is lexically scoped (e.g., an `if` statement or a `while` loop.) where a block generally consists of a key word and possibly additional constructs, a list of zero or more statements, and is terminated with a semicolon.

- Prefix notation is used for expressions[2]. For example, adding two integers would be written as `(+ 2 5)`.
- Comments are the same as C++ or Java (a `//` is a line comment and `/* ... */` is a block comment.
- Blocks are terminated by a semicolon; statements and expressions are not.
- Due to prefix notation, symbols can contain most characters. For example, `is-valid?` is a valid symbol for a variable, function, etc.

3.1 Primitive Types

Primitive types are data types defined by the language and form the basis for all compound objects that can be created. Pretzel's primitive types are similar to most languages and include integers, floating point numbers, and references. Integer types include both signed and unsigned representations and are available in platform *dependent* sizes (`int` and `uint`) and platform *independent* sizes (e.g., `uint32`, `int16`, etc.). Platform dependent variants are defined to be sufficiently large to uniquely enumerate the physical address space of the target platform. Integer types support typical arithmetic and bit-wise operations. Floating point types are assumed to exist as 32-bit (`float32`) and 64-bit (`float64`) IEEE representations also with typical arithmetic operations.

References come in one of three types: static, dynamic, or unique. Static references refer to lexically scoped objects such as local variables in a function, or member variables of a type. Dynamic references address objects dynamically allocated from the heap. All objects referred to by a dynamic reference are partially garbage collected through automatic reference counting (ARC).

[1] The formal grammar can be found in Appendix A.

[2] Prefix notation will be familiar any LISP programmer (and will probably annoy everyone else). The use of prefix notation was primarily to simplify the language grammar. Beyond that, neither prefix or infix notation has any technical impact to Pretzel.

Unique references are Pretzel's version of a linear type and address dynamically allocated memory. Except for references to arrays, all reference types are defined to be a single machine word in size (i.e., they're the same size as a `uint`).

3.2 Variables

The representation and use of data as variables strongly influences the design of a language. With three reference types, this is particularly true for Pretzel. The terms "object", "value", etc. take on slightly different meanings depending on the context of their usage. To avoid any ambiguities, we concretely define these terms.

MacLennan [27] provides a thoughtful description of the differences between *values* and *objects* in programming languages. In essence, a value is timeless and its meaning is immutable. In contrast, objects have state and can be differentiated between even when they have the same bit representation. In Pretzel, all runtime data are objects and values only exist at compile time. We define the following terms:

Object: An object is an instance of a primitive or user-defined type in memory. Objects have a unique memory address and can only be accessed through a reference.

Constants: Constants are strictly immutable objects. Constants exist as compile-time literal values, such as the number 2 or the string `"Pretzel"`. Conceptually, the number 2 is the symbol `2` that is bound to an immutable reference to an immutable object containing the binary representation of the number 2. Constants can also be created at runtime.

Reference: A reference (or pointer) addresses a specific object in memory and can't itself be addressed (i.e., "references to references" do not directly exist).

Variable: A variable is defined as the combination of a reference and an object to which the reference points.

Variables support two basic statement operations: the assignment (`=`) operator and binding (`->`) operator. An assignment statement operates on the object of a variable and has "copy" semantics whereas the binding operator modifies the reference of a variable and has "reference" semantics. For example, `a=b` overwrites the object `b` with the object `a` and the binding statement `r1->r2` binds the reference of `r1` to the object pointed to by `r2`.

Variable declarations have several forms: *fully-qualified reference*, *inferred reference*, *short-hand*, and *inferred short-hand*. A fully-qualified reference (declaration 3.1) declares a variable of a specific reference type that is not bound to an object. The inferred reference declaration (3.2) creates a variable of the same reference type as that of the expression and binds the variable to the result of the expression. The short-hand declaration (3.3) assumes a reference type of `const static` and

```
1  fun foo ()
2  let v1 : dynamic int        // Declare a dynamic reference
3      v1 -> dynamic (int)     // Allocate a new int and bind to 'v1'
4      v1 = 3                  // Set 'v1' to 3.
5
6      // The following two declarations are equivalent where the latter is
7      // uses the short-hand form.
8      let v2 -> const static (complex 1 2)
9      let v3 = (complex 1 2)
10
11     let v4 : int            // Declare a static reference to an int.
12 ;
```

Figure 3.1: Assignment and Binding Statements. This listing shows several variable declarations and the use of the assignment and binding operators.

implicitly allocates an object within the current scope. The inferred short-hand (3.4) does the same thing as a short-hand declaration but additionally assigns the result of the expression to the new variable.

$$\texttt{let}\ \textit{symbol}\ \texttt{:}\ \textit{ref-type}\ \textit{obj-type} \tag{3.1}$$

$$\texttt{let}\ \textit{symbol}\ \texttt{->}\ \textit{expr} \tag{3.2}$$

$$\texttt{let}\ \textit{symbol}\ \texttt{:}\ \textit{obj-type} \tag{3.3}$$

$$\texttt{let}\ \textit{symbol}\ \texttt{=}\ \textit{expr} \tag{3.4}$$

Figure 3.1 provides several examples of variable declarations, assignments, and binding statements. In lines 2–4, the variable `v1` is declared as a dynamic reference to an `int`, bound to a new instance of an `int`, and then assigned the value of 3. The code on Lines 6 and 7 both create a local variable for a complex number and line 8 shows an variable that has a reference bound to an uninitialized integer. There are several points to note:

- Line 2 creates a local reference only and is not bound to an object. Uninitialized reference variables such as this are bound to the special value `nil`. The `nil` value is an invalid address and will produce a runtime error if accessed.
- Immutable static references (i.e., variables declared with `const static`) do not require memory to store the reference since the address can be determined at compile time.
- For a local variable declaration, the code on Line 6 would be cumbersome to type for every local variables. Pretzel provides a syntactic short-hand that when the reference type is omitted from a variable declaration (as in Line 7 and 9), the default reference type is `const static`.

```
fun bar ()
    let a1 = [1 2 3 4]           // Statically size 4 element int array
    let a2 -> dynamic ([8]int)   // Dynamically allocated, static size
    let a3: dynamic []int        // Variable sized array reference

    a3 -> (a2 7 15)              // Out-of-bounds subarray, a3 -> nil
    a3 -> (a2 1 4)               // Bind a3 to a subarray of a2

    (a2 1) = (a1 3)              // Set index 1 of a2 to value at index 3 of a1.
    (print "%()\n" (a3.length)) // Would print '3'
;
```

Figure 3.2: Array Examples. This code shows various array declarations and operations on arrays.

3.3 Constants

Constants are strictly immutable. Unlike C/C++, where the immutability of a variable declared as `const` can be overridden, a variable declared as a constant in Pretzel can't be modified under any condition. In a variable declaration both the reference and the object types may be optionally declared as constant by prefixing the type with the `const` keyword.

3.4 Arrays

Arrays are not primitive types, but are built into the language and are a contiguous sequence of like-typed elements. Given an array with N elements of type T where the size of T is T_{size}, the size of the array is defined to be $A_{\text{size}} = N \times T_{\text{size}}$. Three operations are defined for an array: *length*, *selection*, and *sub-array*. The length operator returns the length of the array as a `uint`. The selection operation returns a reference to the element specified by a zero-based index. If the index exceeds the number of elements in the array, `nil` is returned. The sub-array operation returns a reference to a subset of the given array or `nil` if the range is not a subset of the array. The length of an array is not encoded in the array object[3], but rather it is in the reference to the array. Figure 3.2 shows some examples pertaining to arrays.

3.5 Functions

Functions in Pretzel follow the typical pattern found in many other languages; they have a name, optional parameters and an optional return type. All variables are references which implies a parameter passing style of pass-by-reference. As with variable declarations, parameters default to the reference type of `const static`. The `min` function in Figure 3.3 is accepting and returning

[3]Not combining the length with the array is critical for embedded systems. Arrays in protocols or hardware do not have associated lengths (or do not have them in a defined location). If length data were to be added to an array object, it would not be possible to use such an array to represent these data structures.

```
// Returns the reference to the smaller of the two inputs.
fun min (a:int b:int) : int
    return (if (< a b) a else b)
;

// Adds two values passed by reference and returns result by copy.
fun add (a:int b:int) := int
    let sum:int

    sum = (+ a b)

    return sum
;
```

Figure 3.3: Function Parameter Passing Style. The two functions above show the two parameter passing styles in Pretzel. The `min` function is passing and returning variables by reference whereas the `add` function also passes inputs by reference but returns by copy.

references rather than objects. This default behavior works much of the time, but just as often the need to pass copies arises.

The `add` function in the same figure shows such a case. It's not possible to return a reference to a local variable since it goes out of scope prior to returning to the caller; in this case, the `sum` variable. The `:=` syntax on the return type instructs the compiler to pass a reference for the result to the caller. The caller is then responsible for copying data into the result reference.

There is another problem with the `add` function; we can't make a call such as `(add 1 2)` since the values 1 and 2 are implicitly defined as constants while the parameters are not. The same syntax used for return types can be used for parameters as well. With this syntax, the caller creates copies of the input parameters and passes references to those copies to the function. The `add` function should instead be defined as `fun add(a:=int b:=int) := int` to resolve this. The `:=` syntax functionally provides ability to pass-by-value, but it's still pass by reference[4].

3.6 Parametric Polymorphism

Parametric polymorphism, or *Generics*, are assumed to exist in Pretzel and conceptually behave in much the same way as C++ templates. A generic type or function is defined with the type parameters following the symbol as in Figure 3.4. The type parameters can be qualified with a specific interface that defines what operations are required of the type. The `type` keyword is used when an object can be treated as an opaque object (for instance, the type for a container).

[4]A compiler would be free to optimize these routines into a true pass-by-value form so long as it doesn't change the semantics of the language.

```
type Vector {T:type} /* ...code... */ ;

fun min {T:comparable} (a:T b:T) : T
    return (if (< a b) a else b)
;
```

Figure 3.4: Generics. Example showing syntax of generics.

3.7 Types

Aggregate structures are defined with the `type` keyword and are used to create user-defined types. As an example, Figure 3.5 shows a partial representation of a "point" type. There are two things worth noting. First, variable declarations within a type are not prefixed by the `let` keyword. Secondly, there is both a `distance` *function* and *method*. Functions within a type simply have access to private fields of a type whereas methods have an implicit "this" pointer. Functionally, the `this` reference behaves identically to those in C++ or Java. Types also have *constructors* and *destructors* that serve the same purpose as those in C++.

```
type dynamic Point
    x: float
    y: float

    public
        // Constructor
        Point (x:float y:float)
            this.x = x
            this.y = y
        ;

        fun distance (a:Point b:Point) : float
            return (sqrt (+ (pow (- a.x b.x) 2) (pow (- a.y b.y) 2)))
        ;

        method distance (a:Point) : float
            return (sqrt (+ (pow (- a.x x) 2) (pow (- a.y y) 2)))
        ;
    ;

    impl stringable
        method to-string () : string
            return (strfmt "(%(), %())" x y)
        ;
    ;
;
```

Figure 3.5: Type Definitions. Example showing how composite types are defined.

3.8 Interfaces

Pretzel provides interfaces for type abstraction (this is in place of inheritance which is *not* a supported feature). Interfaces are a collection of unimplemented functions and methods that a type can later implement. The `Point` type defined in Figure 3.5 implements the `stringable` interface (lines 21–25 using the `impl` keyword. Type abstraction through interfaces can be applied either at compile or run-time.

In Figure 3.6, both `print-it` and `static-print-it` can accept any parameter that implements the `stringable` interface. However, the compiled code for both functions is different. Only a single instance of the `print-it` function is created by the compiler. When the `to-string` method is invoked, the correct method is found using dynamic method dispatch (this is comparable to Java interfaces). The `static-print-it` function is duplicated for each distinct type that is passed to it (this is analogous to C++ templates).

This differentiation provides flexibility to the programmer. The use of dynamic dispatch requires additional type information to be stored with each object (Chapter 5 discusses this) and additional computational overhead to find the correct method to call. The parametric version does not require this runtime overhead, but the type must be known at compile time and this also creates a "code bloat" affect since functions are duplicated. Type polymorphism is only allowed for types defined with the `dynamic` keyword (as in Figure 3.5, line 1).

```
1  interface stringify
2      method to-string () : string ;
3  ;
4
5  fun print-it (s:stringify)
6      (print "%()\n" (s.to-string)
7  ;
8
9  fun static-print-it {T:stringable} (s:T)
10     (print "%()\n" (s.to-string)
11 ;
12
13 fun main ()
14     pt = (Point 1 3)
15     (print-it pt)
16 ;
```

Figure 3.6: Interfaces. This example shows the two ways interfaces can be used: statically and dynamically.

3.9 Unions

A union is Pretzel's version of variant or sum types and shares some similarities to tagged unions in C++. Unions are defined with the `union` keyword and are checked with a `match` statement or expression as in Figure 3.7. In Pretzel, the type of a union `union` is determined when a variable is created. Once initialized, the type of a union is immutable for the duration of its lifetime. This restriction exists for reasons related to concurrency. Allowing the type of a union to change in a concurrent environment would require the application to constantly check the type of a union to ensure type safety. However, the contents of the union follow the mutability characteristics of the type the union was created as.

Unions may not be recursively defined (as can be done in functional languages like OCaml or ML). Permitting recursively defined unions would require the need for implicit dynamic memory allocation — something that is not permitted within the language. Without the use of dynamic memory, a recursive union would need to be defined and allocated at compile time and would serve little practical use.

```
union checked-uint
    uint
    None
;

fun checked-add (a:uint b:uint) : checked-int
    return (if (or (< (+ a b) a) (< (+ a b) b))
        None
    else
        (+ a b)
    )
;

result = (checked-add 0xFFFFFFFF 0x10)
match result
    uint => (print "0xFFFFFFFF + 0x10 = \%()\n" result.i)
    None => (print "Addition overflow")
;
```

Figure 3.7: Unions. The above code shows a union definition for a "checked" integer. The `checked-add` function returns a union with the result of the addition. If the addition results in an overflow, a type of `None` is returned. To access the result, a `match` statement must be used to ensure the result is valid.

3.10 Enumerations

The enumeration type in Pretzel closely resembles that of the C or Java enumeration type. An `enum` consists of a collection of identically typed, named entities with optional values. The space required by an instance of an enumeration type is the same as that of the type being enumerated. A `match`

statement can be used to convert from a type to an enumeration of that type as in Figure 3.8.

```
enum Color:int RED BLUE GREEN;

fun draw-colored-circle(c:Color) /* ...code... */ ;

fun main()
    let c:Color
    let i = (rand-int)

    match i
        Color => (draw-colored-circle i) ;
        else => /* Not a color, do nothing */;
    ;
;
```

Figure 3.8: Enumerations. A `match` statement is used to determine if a random integer is a member of the `Color` enumeration.

3.11 Concurrency

We do not specifically address the issue of concurrency in this work, but some aspects of it are still relevant to the reference type described herein. Concurrency is intended to be built into the language to allow various guarantees to be made. Dynamic and, with the exception of global memory, static references can't be shared between threads. There is assumed to be an `async` statement that allows a separate thread of execution to begin. For example, a function would be started as a separate task with `async (foo a b)`. An `async` call restricts the allowable types of parameters to unique references, constants, and dynamic references qualified with the `shared` keyword. These assumptions allow us to reason about the reference type system exclusively as a synchronous process.

3.12 Modules

The module system in Pretzel uses a hierarchical system to compose separate functional units of code and imposes a strict acyclic ordering on dependencies. This has a number of advantages from both language design and software engineering perspectives. However, as noted by Crary et al. [6], a strictly hierarchical module system can pose some practical problems when modules are mutually dependent. These problems are not considered a significant impediment; there are examples of languages that successfully use this strict dependency ordering (e.g., the Go language [14]). A thorough treatment of this topic is beyond the scope of this text, however some basic properties of the module system are crucial to the design of reference types in Pretzel.

Separate compilation is a necessity for any moderately-sized application. For many languages (most notably C/C++), a unit of code can be compiled prior to any of its dependencies. In Pretzel,

the lifetime analysis performed by the compiler to ensure memory safety requires metadata pertaining the use of parameters in functions and methods. By requiring a strict ordering of dependencies, the module system guarantees that any dependencies will have been compiled and the necessary metadata is available.

CHAPTER 4
REFERENCE TYPES

Performance and safety are traits of a programming language that are often in conflict with each other—performance is sacrificed for safety or safety for performance. This is especially true when considering a language with reference types. It seems to be a very difficult problem to design a type system that has a homogeneous reference system while simultaneously preserving the necessary performance and expressiveness required for embedded systems. Purely garbage collected systems simply have too much overhead and manual memory management allows too many safety violations.

Pretzel's approach is to provide a heterogeneous collection of reference types. All types are safe, but have differing semantics and runtime characteristics. This allows a programmer to selectively choose a reference type that has the necessary functionality while maximizing performance. Chapter 1 outlines a set of concrete goals for Pretzel. This chapter describes the method in which Pretzel achieves some of these goals through its reference types. A formalized notion of a *lifetime* is presented along with the semantics and memory safety guarantees of each reference type. After establishing the memory safety of each reference type individually, we show how the distinct reference types fit together to form a cohesive reference type system.

4.1 Lifetimes

The lifetime of an object is the time for which that object may legally be referenced by a program. The lifetime of an object may be computable at compile time (such as stack-allocated objects) or non-computable where the life of the object can only be determined at run time. The lifetime of all objects in a program at a given point in time is the partially ordered set $L = L_D \cup L_N$ where L_D are objects with a computable lifetimes and L_N is the set of objects with non-computable lifetimes. The computable lifetimes form an ordered set where $L_D = \{L_1, L_2, \ldots, L_N\}$ for some finite N. The lifetime L_a is said to outlive L_b if $a > b$. An object d with a non-computable lifetime is not comparable with any other lifetime (i.e.: $L_d \parallel L$). Recall from Section 3.2 that variables are composed of a reference and an object it points to. Given a variable v that references an object o, we define the following lifetime functions:

$$\mathcal{L}_R(v) = L_r \text{ where } L_r \text{ is the lifetime of the reference.} \tag{4.1}$$

$$\mathcal{L}_O(v) = L_o \text{ where } L_o \text{ is the lifetime of the object.} \tag{4.2}$$

To guarantee the memory safety of a program, the lifetime of any reference may never exceed the lifetime of the object to which it refers. Given the set of all variables V that exist during a program's

execution, the following condition must always hold true:

$$\mathcal{L}_R(v) \leq \mathcal{L}_O(v) \text{ for all } v \in V \tag{4.3}$$

4.2 The `nil` Object

References can be bound to the `nil` "object". Any use of a reference bound to `nil` must be detected or prevented. In general, there are two approaches to dealing with such a condition:

1. Trigger a runtime error if a `nil` reference is used.
2. Ensure a `nil` dereference can't occur.

Both options can be found in various languages, but Pretzel takes approach 1. It's easy to understand and easy for a compiler to implement and doesn't clutter language semantics. However, it has a potential performance impact to embedded systems. For platforms with a MMU, a `nil` reference can be assigned to a memory address that triggers a hardware fault avoiding performance issues. For platforms without an MMU (a scenario not uncommon for embedded systems), the compiler must insert conditional runtime checks.

4.3 Static References

This section describes static references, static objects, and the associated safety guarantees provided by the language. A static reference can only be bound to an object with a lifetime that can be verified by the compiler at compile time such that Equation 4.3 remains true. A program can be represented as an abstract syntax tree (AST) and we can further associate a scope with each block element[1] of the tree. Within a scope, references and variables may be allocated. When a program executes, it recursively traverses paths on the tree statically allocating memory for each scope it enters and deallocating the memory when it leaves. This is essentially a description of any stack-based lexical scoped programming language. A key aspect of this description is that static memory is allocated solely as a result of program execution.

To prove the safety guarantees of static references, when begin by formalizing the above notion of program execution. Let P be the tree representation of a program where each node in the tree is a scope and S is the set of all scopes in P. Each element s_i of S is the tuple (d_i, V_i) where

1. d_i is the depth of s_i in P.
2. V_i is the set of all statically allocated variables and references in s_i.
3. L_i is the lifetime associated with the scope s_i. All elements of V_i have a lifetime of L_i.

[1] Block elements are language constructs such as `if` statements, `while` loops, or functions.

Let e_i be a single execution path through P and E be the set of all execution paths where

$$e_i = (s_0, s_1, \ldots, s_{n-1}) : n \geq 2, s_i \in S, \forall s_i, s_{i+i} : d_{i+1} = d_i + 1$$

During program execution on any given execution path, when the program is in scope s_i all statically allocated memory in $V_a : a \leq i$ is guaranteed to be valid and all memory in $V_d : d > i$ is invalid. Stated another way, the lifetime of all scopes in $\mathcal{L}(V_a) > \mathcal{L}(V_d)$.

We can guarantee memory safety of static objects by ensuring that all binding statements do not violate Equation 4.3. If source code was always available and whole-program compilation was required, this would be a simple task. However, this is not the case; program units can be compiled separately and in many cases the source code is unavailable. This problem is addressed by compiler-generated reference binding annotations. Each function[2] has a set of visible mutable references R_M and a set of immutable references R_I that are accessed by the function[3]. For a given function f, the compiler assigns a set of reference bindings B. Each element b_i of B is a pair $\langle \rho_i, \tau_i \rangle$ where

1. ρ_i and τ_i represents a binding statement ρ_i `->` τ_i in f.

2. R_I are input parameters or immutable references that are in the global scope or can be reached indirectly through in input parameter.

3. R_M are mutable references that exist in the global scope, can be reached indirectly through an input parameter, or is the result returned by the f.

4. B is the set of binding annotations for f where

$$B = \{\langle \rho_1, \tau_1 \rangle, \langle \rho_2, \tau_2 \rangle, \ldots, \langle \rho_N, \tau_N \rangle\} \text{ where } \langle \rho_i, \tau_i \rangle : \rho_i \in R_M, \tau_i \in R_M \cup R_I \tag{4.4}$$

The binding annotations do not include reference parameters declared as pass-by-copy (the `:=` separator in a parameter declaration). The issue of separate compilation is resolved through the Module system described in Section 3.12. The module system for Pretzel requires that all modules are structured in a hierarchical manner consistent with a directed, acyclic graph. This property requires that for any given unit of compilation, all of its dependencies have been compiled. This ensures that reference binding annotations are available when any function is compiled.

Using the above description of code execution and the notion of binding annotations, we prove by induction that the compiler can guarantee Equation 4.3 for all static references. Every flow of execution corresponds with a sequence $e_i \in E$ with the last element s_N of e_i having the shortest

[2]Although only functions are mentioned, the concept holds for methods as well. A method is simply a function with an implicit "this" reference that can be used to access data fields of the type.

[3]It is assumed that the compiler has the ability to generate the set of reference bindings for a given function. Since the reference bindings are simply a set of binding statements that occur within a function, no code or data flow analysis is necessary. This makes the problem of enumerating the bindings of a function trivial.

lifetime and no sub-scopes. We use s_N for the basis. The lifetime of all references within lexical scope of s_N are known. Therefore Equation 4.3 can be guaranteed for all static objects local to s_N.

We assume for some $s_n \in e_i$ that Equation 4.3 holds true. We must prove that Equation 4.3 holds for s_{n-1}. The lifetimes of local variables can be checked as in the basis. We are primarily concerned with function calls made with reference parameters. For any function call f, the binding annotations show the bindings that can occur in f as they relate to the parameters that are passed to f. With this, we can guarantee that the parameters passed to f do not violate Equation 4.3. This process always depends on ability for the immediately preceding scope (the scope with the next longest lifetime) to also verify lifetimes are valid. If e_i were infinite, this would be a problem; but this isn't the case. Eventually, by following this logic, s_0 will be reached since e_i has a finite number of elements. In the "root" scope, all lifetimes are known and we can again apply the argument in the basis to s_0. This shows that the lifetimes for all static objects can be checked at compile time. ■

We present an example of this static analysis in Figure 4.1. In this code, the `Int` type simply wraps a static `int` reference and the `swap` routine exchanges the internal references within `Int`. The `bar` routine declares a local instance of `Int`, binds the internal reference to local variable, and then calls the `swap` routine. The only way this is can be valid is if both internal references manipulated by `swap` have the same lifetime; in this example they do not. The `swap` function has the binding annotations

$$B_{\texttt{swap}} = \{(\texttt{a.i}, \texttt{b.i}), (\texttt{b.i}, \texttt{a.i})\}$$

We assume the parameters passed to `bar` have a lifetime of L_n and the local variables in `bar` have a lifetime of L_{n-1} where $L_n > L_{n-1}$ as per the definition of scopes. Based on $B_{\texttt{swap}}$, we can see that `swap` binds `x.i` with `y.i`. If `swap` were called, $\mathcal{L}_R(\texttt{x.i}) > \mathcal{L}_O(\texttt{y.i})$ which violates Equation 4.3.

4.4 Dynamic References

Dynamic references address objects that are dynamically allocated during program execution and the lifetime of these objects can't be determined at compile time. To guarantee memory safety for dynamic references, an object allocated in memory must remain valid as long as one or more references to it exist. Pretzel uses automatic reference counting to ensure dynamically allocated memory remains valid. However, one of the issues with ARC are reference cycles. Currently there are no techniques to automatically deal with reference cycles in a way suitable for low-level programs. To manage the issue of reference cycles, Pretzel requires cycles to be broken by explicitly freeing memory with a `free` statement. This complicates ensuring memory safety with ARC since the programmer may free memory when references to the object still exist.

Objects without reference cycles behave in the same way as in traditional ARC; memory is

```
type Int
  public i:static int;
;

fun bar (x:Int)
  let y: Int
  let j: int

  y.i -> j
  (swap y x)
;
fun swap (a:Int b:Int)
  let tmp: static int
  tmp -> a.i
  a.i -> b.i
  b.i -> tmp
;
```

Figure 4.1: Static Reference Analysis. The `Int` type simply wraps up a static `int` reference. The `bar` function accepts an `Int` parameter and declares a local `Int` variable and binds the `y.i` reference to a local variable. The `swap` routine called by `bar` should not be allowed.

automatically reclaimed when the last reference to an object is removed. However, if a reference cycle exists, the memory will not be collected without the programmer explicitly freeing the memory[4]. Furthermore, a programmer may free any object at any time, but memory safety must still be maintained for any other dynamic references to the freed object. This conflict is managed by dividing the standard reference count into two parts: a guaranteed count and a non-guaranteed count. A non-zero guaranteed count ensures the object will remain in a valid, allocated state. The non-guaranteed count promises only that the object memory is still valid, but the object itself may be in an deinitialized state.

Traditional reference counting associates a single count to an object. To support the functionality described above, more information is required. For a given object, the associated RC metadata is the 3-tuple $\langle C_F, C_g, C_{\overline{g}} \rangle$ where

1. C_F is the *freed flag*. Its initial value is 0 and it is set to 1 when a `free` statement is applied to a reference of the object.

2. C_g is the set of *guaranteed* references and the guaranteed reference count is $|C_g|$.

3. $C_{\overline{g}}$ is the set of *non-guaranteed* references to the object and the non-guaranteed count is $|C_{\overline{g}}|$.

The set[5] of all references to an object C is $C_g \cup C_{\overline{g}}$. The guaranteed RC C_g is incremented when a local[6] reference is bound to the object and is decremented when the references goes out of scope.

[4]This has the unfortunate consequence of allowing memory leaks.

[5]Sets are used to describe reference counts because actual counts are not strictly necessary. Section 5.2 will use this fact in the data representation of the RC metadata.

[6]In this context, a "local reference" is a dynamic reference declared as a variable or passed as a parameter.

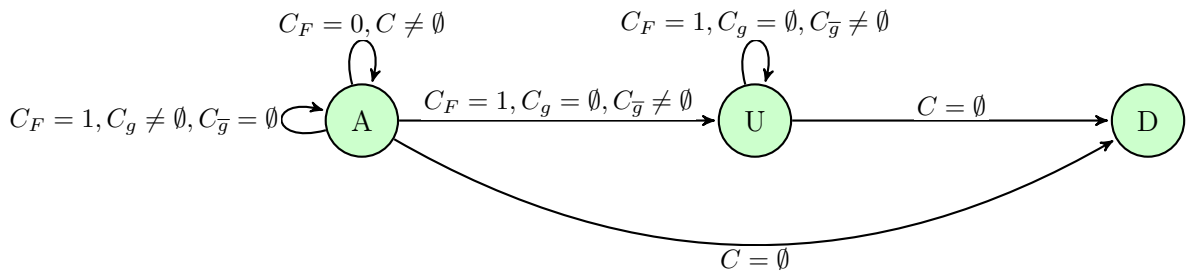

Figure 4.2: Dynamic Memory States. The above state diagram shows transitions induced by reference mutations to an object. transistions

An object may be also be accessed through a pointer contained within another type that is not a local reference, however this still results in the binding of a local reference. All method invocations are just function calls with the implied "this" parameter. Since parameters are local variables, this results in the necessary reference binding to increment and decrement C_g.

The RC metadata produces a state machine for the object's lifetime where every reference mutation induces a state change. An object has three states: *allocated*, *deinitialized*, and *deallocated*. Figure 4.2 shows the state transitions of an object that is dynamically referenced. As can be seen, memory is only deallocated when no references to it exist. In the allocated state, an object has been initialized and the object can be referenced. On a transition to the deinitialized state there can be no guaranteed references to it and the object is "deinitialized" by calling the object's destructor. This will cause any non-guaranteed references the object contains to be unbound and will break any reference cycles.

Subsequent evaluation of a reference to an object with $C_g = \emptyset$ and $C_{\overline{g}} \neq \emptyset$ will return `nil`. Once all references have been removed, the object moves to the deallocated state where the object is release to the free pool of memory. We can see from this, that if there are any active references to an object, its memory remains valid and ensures memory safety.

4.5 Unique References

Conceptually, unique references work on the simple concept that an object can't be aliased more than once—in other words a unique reference points to an object that has no other references to it. Deutsch and Bobrow [8] observed that most objects have only a single reference during their lifetime. Although this result was obtained for a Lisp-like language using processing workloads that may differ from what may be more commonplace today, it's still reasonable to assume a large percentage of objects still follow this pattern. Unique references in Pretzel (identified by by the `unique` keyword) capitalize on this observation. By making a unique reference a built-in language construct, they provide automatic memory management with performance that is almost unbeatable. A unique reference is essentially a reference counted object that can only have a reference count of 1 or 0.

```
1 fun foo ()
2     let x -> unique (X)
3     let y: dynamic Y
4
5     if (and (= x nil) (= x.y nil))
6         y -> x.y
7         async (bar x)
8         (y.baz)
9     ;
10 ;
```

```
1 fun bar (x:Unique X)
2     (x.y.baz)
3 ;
```

Figure 4.3: Reference Containment. This code shows an example of how a dynamic reference may be aliased between mutliple concurrent tasks.

This binary state is embodied by the existence of a reference to an object and, as a result, unique references do not require the overhead to store or update a reference count. Consequently, given an object m and a unique reference r_u with some lifetime L_n that points to the object, the lifetimes are $\mathcal{L}(r_d) = \mathcal{L}_R(m) = L_n$. Unique references naturally avoid race conditions due to the fact that they can only be referenced by a single pointer at any time.

While unique references offer an efficient mechanism to manage dynamically allocated memory, they are less flexible than static or dynamic references. Common data structures, such as a queue or doubly-linked lists, can't be described by unique references. A queue data type requires the last element to be referenced twice; once by a "tail" pointer and once by the next-to-last element. Each node in a doubly-linked list must also be referenced at least twice—once each by the predecessor and successor nodes.

Unique references also pose difficulties for types that contain dynamic references. Non-shared dynamic references must exist in only one task. Figure 4.3 illustrates this issue. Assume type `X` contains a dynamic reference to an instance of type `Y`. In line 6 a reference to `y` is saved. After that, `bar` is asynchronously executed where it receives the unique reference to `x`. The non-shared dynamic reference to `y` is now accessible to both tasks.

The semantics we wish to attain would allow dynamic references to "move" with a unique reference to maintain the race-free properties they have. This requires that the compiler prevent any such references from "escaping" an object bound to a unique reference. The compiler can prevent the unwanted behavior within a lexical scope by disallowing the binding of public dynamic references of an object to external references. For code compiled separately (where the compiler does not have access to code), the binding annotations described in Section 4.3 can be used. From definition 4.4, the bindings contain all possible targets of a binding operation for the given input parameters. The compiler simply needs to consult this list to determine if the use of a unique parameter in a function/method is valid.

4.6 Unifying the Reference Types

Thus far, we have presented the three reference types of Pretzel in isolation. Kept in isolation, the use of three reference types would pose some significant software engineering challenges. Functions would need to be duplicated (either manually or through parametric types) to accommodate each reference type even when the function doesn't depend on the specific type of reference. Fortunately, this problem is easily managed for a large portion of common programming idioms. Code is largely based on the lexical block structure of a program—namely function calls, parameters, and local variables. We can "bend" the semantics of dynamic and unique references to that of static references. We begin with the premise that the compiler can implicitly convert all reference types to static references.

The lifetime of an object bound to a dynamic references is determined by the existence of one or more references. We consider a block of code with a scope s_n in which all local variables have a corresponding lifetime of L_n. Assume there exists a local variable v that is a dynamic reference to an object m. A dynamic reference guarantees that the object it points to will not be released so long as it exists. If we pass v as a static reference to a function, the lifetime of the object referenced by v is always greater than that of the subroutine since a reference to m always exists in the caller. Therefore, the lifetime of m is deterministic within s_n and v can be treated as a static reference for all code blocks s_x where $x > n$.

The same general reasoning can be applied to unique references, but with some slightly different semantics. When a unique reference is passed to function with a static type, it temporarily assumes the lifetime of the function as a static reference. This does not violate the semantics of unique references which can be shown by examining object lifetimes. Given a unique reference r_u and a function f (and associated parameters) with lifetime L_f, $\mathcal{L}(r_u) > L_f$ by virtue of the lexical block structure of the code. Consequently, any static parameter whose origin is a unique reference will also have a lifetime of L_f. Within f, the unique reference is treated like a static reference. As per the rules defined in Section 4.3, no references to the parameter that is the converted unique reference may exist after the function returns. Upon return from the function, the unique reference will again be the sole reference to the object it points to.

CHAPTER 5
OBJECT MODEL

Thus far, Pretzel has primarily been described in terms of its features and semantics. However, we are still missing an important aspect of the language—namely *how* a compiler can implement those features. Pretzel is intended to be a practical language and, as such, there must be realistic methods to implement its features. For embedded systems and other low-level programming tasks, these implementation details are especially important. Many features require runtime data or code that is not explicitly created by a programmer, but nonetheless needs to be factored into the design of such systems.

The Object Model presented in this chapter describes the runtime constructs necessary to implement the various features associated with objects. The single requirement of the object model is that it can't alter the semantics of the language. Additionally, the object model has been designed with embedded systems in mind and attempts to minimize the impact to memory resources and performance. With C/C++ as the *de facto* standard for embedded systems, we also attempt to quantify some of the performance characteristics of the implementations described here with analogous features in C++.

5.1 Data Layout

Data layout pertains to ordering of data structures within physical memory. The rules governing such organization varies widely amongst languages. Very high-level languages such as Python or Scheme completely abstract such concerns. In contrast, lower-level languages like C and Ada provide explicit control over how data is placed in memory. Member variables in a type declared with the *short-hand* declaration syntax (declaration 3.3) reserve space only for the object of the variable. The reference itself is known at compile time as a constant offset from the `this` reference. In essence, variables declared this way behave identically to those in C/C++.

By default, Pretzel makes no guarantees regarding the positioning of a variable within a type definition and the compiler is permitted to organize data in whatever manner it wishes. This provides for various strategies to be employed based on desired optimization criteria. Consider the structure in Figure 5.1. Certain processors (e.g., ARM) issue multiple load/store instructions to access memory that is not aligned on specific boundaries. A compiler would need to insert wasteful padding to conform to the necessary alignment or incur the performance penalty of multiple memory accesses. An optimal strategy is to reorder the fields as shown on the left to avoid unnecessary padding or unaligned accesses[1].

[1]When compiled with a C compiler, the structure in Figure 5.1 is 32-bytes in size when the actual type definition only contains 23 bytes of defined data.

```
1 type X
2     sum: uint64
3     a: uint8
4     x: uint32
5     b: uint16
6     y: uint64
7 ;
```

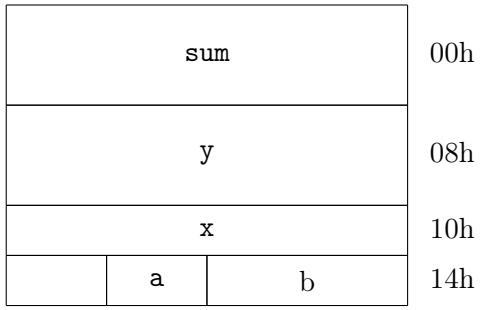

Figure 5.1: Data Layout. A structure definition (left) and an ideal data layout (right).

Allowing the compiler to order type fields automatically is generally optimal both in terms of programmability and performance; understanding the impact of data alignment on CPU load/store instructions and organizing structures for space is a task compilers excel at. However, there are situations when it is desirable or necessary to explicitly specify the order of fields in a data structure. Interfacing with memory-mapped hardware devices or implementing protocols are common situations that require explicit control over data layout. Optimizing for cache efficiency is another. A programmer is often aware of data usage patterns that compilers generally can't detect. Frequently used data can be grouped together to make efficient use of a CPU's cache architecture. Pretzel provides the `verbatim` type qualifier to indicate the fields of a type should be ordered in memory exactly as they appear in the type definition. Each data field of a type is placed at the next byte position (in increasing address order) immediately following the prior data field.

5.2 Automatic Reference Counting (ARC)

5.2.1 Storing Reference Counts

As described in Section 4.4, the reference counting mechanism requires three states to be associated to an object addressed by a dynamic reference; the guaranteed RC (C_g), the non-guaranteed RC ($C_{\overline{g}}$, and the free flag (C_F). These states are collectively referred to as the *reference count metadata*. When an object is dynamically allocated using the `dynamic` keyword, the overall size of the object is increased to accommodate this metadata. The structure of the RC metadata is the focus of this section.

Conceptually, C_g and $C_{\overline{g}}$ both count references. A straight-forward approach would be to simply represent these both as integers. Unfortunately, this creates problematic issues. In general, the variables used to store a reference count must either be large enough to hold the theoretical maximum number of references $C_{\max}$ or some mechanism must exist to handle an overflow condition. Neither option is desirable. The use of two counts that can represent $C_{\max}$ arguably consumes too much memory whereas managing an overflow condition has a non-trivial impact to runtime performance.

Alternatively, we can leverage the fact that modifications to the guaranteed count follows the first-in, last-out behavior of block-structured program flow. Rather than store an actual count for guaranteed references, C_g can be represented as a single bit where a value of 1 implies the object is guaranteed and 0 if it's not. Each time a reference would increment the guaranteed count, the old value of C_g is pushed onto the runtime stack and C_g is then set to 1. When the guard is removed, the old value is removed from the stack and replaces the current value of C_g.

We wish to limit the size of C to that of a basic reference. A machine with an n-bit word size can address 2^n unique bytes, a reference is $n/8$ bytes. The reference count can track $2^n - 1$ references which is always greater than or equal to the maximum number of possible references $2^n/(n/8)$. The C_F and C_g require 2-bits of data leaving $n/8 - 2$ bits for the non-guaranteed reference count.

A shared dynamic reference can't use the approach just described since modifications to the guaranteed reference count is no longer based on a single path of execution. For these references, C_g and $C_{\overline{g}}$ are maintained as two separate word-sized counts.

5.2.2 Uninitialized References

When a dynamic reference is explicitly freed through a `free` statement, the referenced object is "deinitialized" by calling the type's destructor once the guaranteed count reaches zero. However, non-guaranteed references to the object may still exist. To ensure memory safety, the memory the reference addresses can't be released until the non-guaranteed count is zero.

One approach approaches to manage this would be to separate the object from the RC metadata. A pointer to the object would be added to the RC metadata and a dynamic reference would point to the RC metadata. This would require that all objects are indirectly referenced by first accessing the pointer to the RC metadata and then following the pointer to the object. This has the advantage that memory for the object can be released immediately after the guaranteed count reaches zero leaving only the RC metadata which can be released once the unguarded count becomes zero.

Alternatively, the object and RC metadata can be combined as a single unit in memory. For performance reasons, this is the method used by Pretzel. As shown in Figure 5.2, the cost of indirectly referencing an object using the first method incurs a significant performance cost. These results were obtained by simply measuring the cumulative timing differences between the two methods when repeatedly accessing random objects from a large set of objects in memory. Since guaranteed reference counts are generally temporary in nature, the ability to free the memory associated to an object early doesn't merit the additional pointer that must be stored and the associated performance cost.

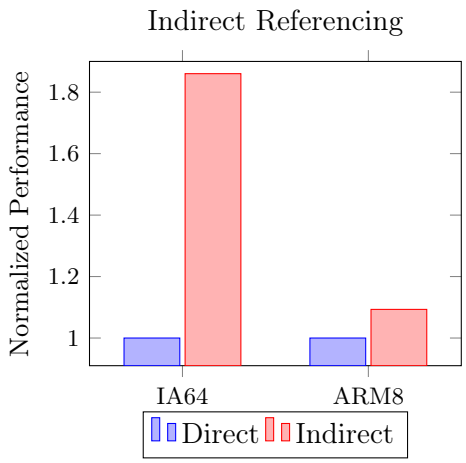

Figure 5.2: Indirect Object Referencing. Performance difference between object access directly through a single pointer and indirectly through two pointers.

5.3 Array References

In general, a reference is a single machine word in size. This is true for all references except for those that address arrays. Arrays are a fundamental data structure of most languages. Pretzel is no different in this regard, however, Pretzel has two somewhat contentious requirements. To ensure memory safety, the length of an array must be known any time an element is accessed. The length must either be stored as metadata that is attached to the array data or must be statically inferred by the compiler. Statically inferring length information for an array severely reduces programmability while adding length metadata to an array datatype is not viable when programming hardware devices or protocols that requires only array data to be present. Consequently, references to arrays vary in size depending on the type of array they address. What follows details the mechanism used by Pretzel to implement array references.

Array references exist in two forms: *fixed-length* and *variable-length*. Fixed-length arrays have an implicit length that is encoded in the type declaration of the variable and is known at compile-time. Variable-length arrays[2] are bound to a subset of another array and do not allocate memory for the content they represent, but additionally store the length of the array they refer to.

To provide memory safety, references to arrays must encode the range they cover. This encoding may be part of a variable declaration or the runtime representation of the reference. Static array references may require up to two words of data (the address of the first array element and the length of the sub-array it references). Dynamic references add an additional layer of complexity and may require up to three words to properly reference an array. The reference count of an array addressed by a dynamic reference is associated only to a fixed-length array. Variable-length arrays must additionally reference the fixed-length array to ensure the reference count is not reduced to zero.

[2]Variable-length arrays are similar to slices in other languages.

```
1 let arr-1: [0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15]
2 let arr-1a: static [4]int
3 let arr-1b: static []int
4
5 arr-1a -> (arr 4 8)
6 arr-1b -> (arr 9 12)
```

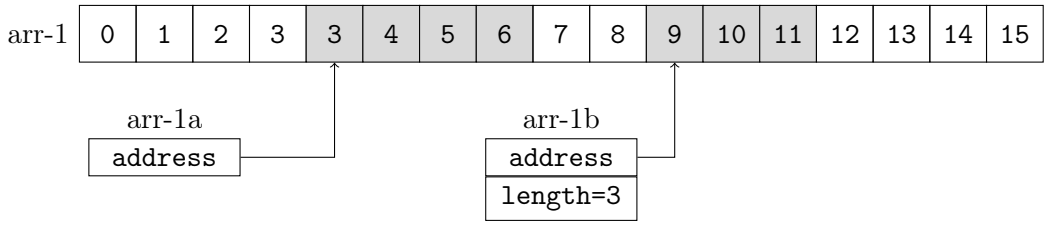

Figure 5.3: Static Array References. This figure shows the representation of static array references. The variable `arr-1` is a local array whose memory is allocated on the stack. The size of the array and address of the reference are known at compile time and therefore does not require memory beyond the array itself. Array `arr-1a` is a mutable reference with a fixed length whereas `arr-1b` is variable length and requires the length to be encoded in the reference.

Figures 5.3 and 5.4 show various array variable declarations the associated reference representations.

5.4 Interface Method Dispatch

Interfaces in Pretzel (or any other language that supports interfaces) require the ability to select a specific method from a set methods defined by an interface in the absence of type information. This is referred to as *dynamic method dispatch*. We are primarily interested in providing a proof of concept that this dispatch mechanism can work by adding one additional pointer to an object. The proposed method is not ideal and would likely need to be improved to be practical. Ideally, this method lookup process is fast. When single inheritance is used, C++ uses a "virtual table" to achieve this [36]. Since only one type is involved in single inheritance, this process is very fast for C++.

Pretzel uses interfaces and, as such, a type may implement multiple interfaces. We add an additional level of indirection to the method used in C++ to find the correct methods pointer. Once the correct set of methods is found, the dispatch mechanism works in the same way as C++. The *Interface Lookup Table* (ILU Table) shown in Figure 5.6 shows the structure of this table. When a dynamic method invocation is made (such as that from line 5 in Figure 3.6), the compiler must insert code to search for the methods that correspond with the type.

To gauge the performance of the linear search mechanism, we compared it against the idealized direct look-up. The body of the test methods contained a single addition statement to ensure the performance measurements were dominated by the dispatch mechanism. The simulated object for the linear search contained four interfaces and the test and lookup table was structured such that each method invocation required required two iterations to find the correct method. As can be

```
1 let arr-2 -> dynamic ([0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15])
2 let arr-2a: dynamic [4]int
3 let arr-2b: dynamic []int
4
5 arr-2a -> (arr 4 8)
6 arr-2b -> (arr 10 13)
```

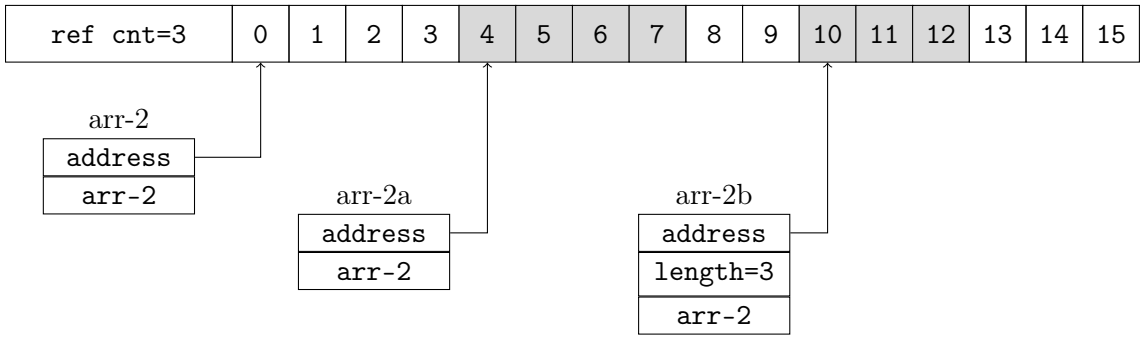

Figure 5.4: Dynamic Array References. This figure shows variable declarations for dynamic array references. For all references of this type, both the address to the start of the array and the address of the fixed-length array are required. Syntactically, the compiler can't distinguish between the declarations on lines 1 and 2, so both require two references, however the `address` field for `arr-2` is equal to `arr-2`. Also important to note is that the `address` field in the reference representations do not modify the reference count; each array reference contributes a single count to the RC.

seen in Figure 5.5, the linear search performs significantly worse than the idealized direct look-up. Although Pretzel's interfaces provide more flexibility, its performance would not compare well against a comparable single inheritance implementation in C++.

5.5 Metadata Layout

We have introduced a number of concepts that add additional metadata to instances of a type. The placement of this data in the memory layout of an object is crucial to support the distinct reference types of Pretzel. The only requirement is that the metadata does not alter the meaning of a static reference. In particular, the metadata must not be positioned between the address assigned to the `this` pointer and the object itself. We show the data placed prior to the `this` pointer, but it could also exist past the object data. The ILU pointer is part of the object and must be included by the `this` pointer. Array references have similar requirements. The primary difference being that the dynamic "owner" reference must not be visible to a static reference. To achieve this, the "owner" reference is placed as the last element of an array reference object.

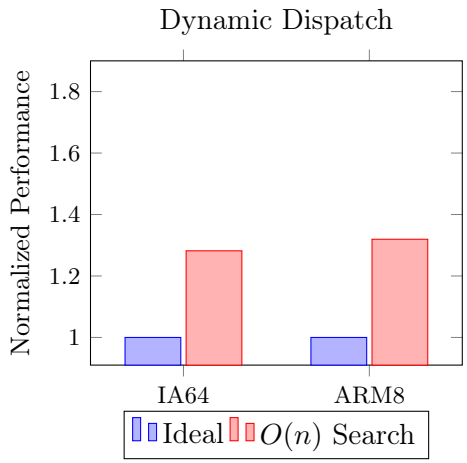

Figure 5.5: Dynamic Dispatch Performance. Performance diff. between idealized $O(1)$ and $O(n)$ method lookup for dynamic dispatch.

ID=stringable	00h
Offset 18h	
ID=B	
Offset 1Ch	
ID=C	
Offset 28h	
X.to-string	18h
X.foo	1Ch
X.bar	
X.baz	
X.bleep	28Ch
X.bloop	

```
interface stringable
    method to-string ;
;

interface B
    method foo() ;
    method bar() ;
    method baz() ;
;

interface C
    method bleep() ;
    method bloop() ;
;

type X
    impl stringable
        method to-string() /* ... code
            ...*/ ;
    ;

    impl B
        method foo() /* ... code ...*/ ;
        method bar() /* ... code ...*/ ;
        method baz() /* ... code ...*/ ;
    ;

    impl C
        method bleep() /* ... code ...*/ ;
        method bloop() /* ... code ...*/ ;
    ;
;
```

Figure 5.6: ILU Table. The ILU table in this figure shows three interfaces and the lookup table the dispatch mechanism would use to locate the correct method pointer.

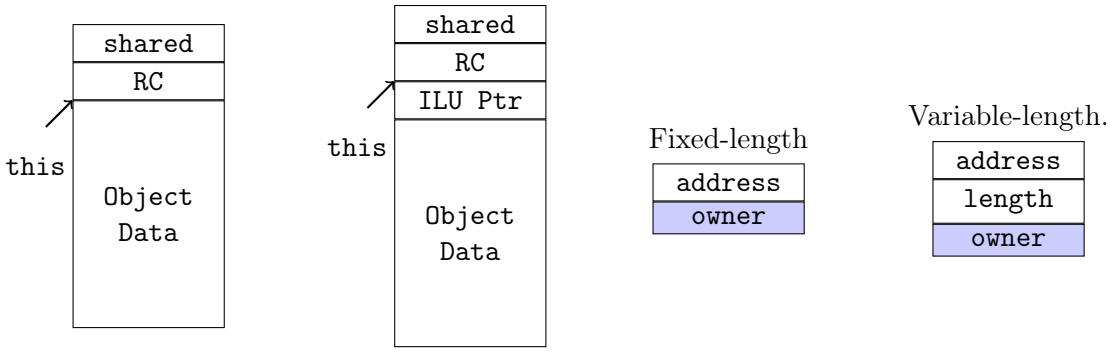

Figure 5.7: Object Metadata Layout. When a dynamic references are converted to static references, the positioning of metadata can't alter the expected layout of a static reference. The "this" arrow is the addressed that is stored in memory for reference types.

5.6 Type Conversions

Pretzel permits a number of implicit and explicit type conversions. Conversions in Pretzel are not a simple reinterpretation of data as in C/C++, but involve some type of data copy or transformation. This section details the runtime overhead associated with type conversion for enumerations, unions, and reference types.

5.6.1 Enumeration Types

Given an enumeration E_T of type T, an enumeration can be implicitly converted to the type $T$with a simple assignment operation. Conversions from an instance of type T to one of E_T requires a runtime check to ensure it is a member of E_T. The compiler may perform any number of optimizations to make this operation fast. For types that do not lend themselves to simple optimizations, worst case performance can be limited to $O(\log n)$. The elements of E_T must be known at compile time. Based on this property, a static binary tree can be built at compile time to guarantee this upper-bound on performance.

5.6.2 Union Types

A `union` is Pretzel's version of a sum or variant type. The in-memory structure contains a one-word tag followed by sufficient memory to contain the largest data type within the union. The tag field is a one-to-one mapping of an integer to a type and a sequential enumeration beginning at zero. This structure permits matching a type at run-time in $O(1)$ time by using a common compiler optimization [30] applied to C-like "switch" statements.

5.6.3 Reference Types

Section 4.6 described the semantics of converting between reference types. Here, we discuss the runtime implications to these type conversions. Converting from any reference type to a static reference has no runtime cost. The reference points to the object and, in the case of dynamic references, the associated metadata is simply ignored. Converting between dynamic and unique references, however, does have runtime implications.

Unique references do not contain extra memory for the metadata associated to dynamic references. When a conversion from a unique to a dynamic reference occurs, a new block of memory is allocated as would be done with a normal allocation for a dynamic reference. We are guaranteed that there is only the single reference to the object pointed to by the unique reference and that all references with the object are contained (i.e., they are not aliased by any reference external to the unique reference). This allows a "shallow" byte-for-byte copy to be performed from the old memory to the new memory. Of course, the allocation of the new memory region can fail; this is the reason for the semantics that result in neither reference being modified.

Converting from a dynamic to a unique reference follows the same logic; this type of conversion can only occur if there is a single reference to the dynamically referenced object. Although a unique reference is always smaller than a dynamic reference, the "allocate and copy" process is still desirable (although not strictly necessary). The reason for this relates to heap management efficiency. In a language like C or C++, the `free` library routine and `delete` operators accept a untyped pointer. This requires that the heap management software "remembers" how big the object that the pointer referenced. Typically size information is typically stored along with the object, but is hidden from the programmer. Pretzel always knows how big an object is either through static type information or through the ILU pointer for dynamic types. This eliminates the need to store size information in the heap.

CHAPTER 6
IMPROVING REFERENCE COUNTING

Developing a memory management strategy for embedded systems is a tricky proposition. The combination of a vast application space and equally varied hardware platforms makes it difficult for any one single memory management scheme to work well in all scenarios. Many modern programming languages abstract away as many of the details involved with memory management as possible; and for good reason. It's a constant source of bugs in languages like C/C++ and places greater software engineering demands on programmers. However, high-level abstraction obscures flexibility that is often necessary in embedded systems.

The goal of Pretzel's reference types is to provide as much abstraction as possible while still providing the ability to manage low-level aspects of memory management. To achieve this flexibility, dynamic memory allocation is a necessity in certain cases. As previously mentioned, automatic reference counting is used to ensure memory safety in the presence of dynamic memory allocation. However, ARC presents practical challenges when performance is considered. Many of the techniques proposed to improve ARC rely on runtime facilities which conflict with Pretzel's design requirements.

This chapter presents two improvements to ARC. Reference count mutations caused by reassigning references is a major cause for the lackluster performance of ARC. We present a compile-time optimization that can minimize such mutations. Secondly, when the reference count of a cycle-free recursive data type (e.g., a singly linked list) reaches zero, it's possible to cause a deeply recursive execution flow that can overflow the runtime stack. A method for avoiding this is also given.

6.1 Optimizing Reference Count Mutations

A naive approach to implementing ARC increments and decrements reference counts whenever a reference is modified. This approach is easy to reason about and relatively simple for a compiler to implement. Unfortunately, reference count mutations account for a significant portion of the inefficiencies found in ARC. Managing RC mutations also requires additional code which, while minimal, may be significant to some embedded platforms in terms of additional code. Two methods have been proposed to minimize these inefficiencies: *deferred* and *coalesced* reference counting. In deferred RC, Deutsch and Bobrow [8] (and later improved on by Bacon et al [3]) observed that updates to local reference variables do not need to immediately modify reference counts and are ignored. While these methods are effective, they both require runtime data structures and processing to implement and also non-deterministically defers the destruction of objects. These attributes are all specifically avoided in Pretzel. This section presents an algorithm for optimizing reference count mutations at compile time. The optimization presented applies to a special subset of programs and is not as robust as the work done by Deutsch and Bobrow [8] and Bacon et al [3]. However, since

```
1  method remove{T:type} (i:=int) dynamic T
2      let n: dynamic T
3      let tmp: dynamic T
4
5      if (= i 0)                    S_{A,0} = {tmp}  U_{A,0} = {head, n.next}
6          tmp -> head               S_{A,1} = {}  U_{A,1} = {n.next, head, tmp}
7          head -> head.next         S_{A,2} = {}  U_{A,2} = {n.next, head, tmp}
8
9          return tmp                S_{A,3} = {}  U_{A,3} = {n.next, head, tmp}
10     else                          S_{B,0} = {n, tmp}  U_{B,0} = {head, n.next, n.next.next}
11         n -> head                 S_{B,1} = {n, tmp}  U_{B,0} = {head, n.next, n.next.next}
12         (-- i)
13         while (> i 0)
14             n -> n.next           S_{B,2} = {n, tmp}  U_{B,1} = {head, n.next, n.next.next}
15         ;
16
17         tmp -> n.next             S_{B,3} = {n}  U_{B,2} = {head, n.next, n.next.next, tmp}
18         n.next -> n.next.next     S_{B,4} = {n}  U_{B,3} = {head, n.next, n.next.next, tmp}
19
20         return tmp                S_{B,5} = {n}  U_{B,4} = {head, n.next, n.next.next, tmp}
21     ;
22 ;
```

Figure 6.1: The RC optimization is applied to two code fragments A and B. The annotations to the right of the listing show stability sets used to determine which reference counts are necessary.

the optimization is performed at compile time, there is no runtime overhead and it fits within the design parameters of Pretzel.

6.1.1 The General Idea

There are various scenarios in which reference count mutations may be partly or entirely unnecessary. The goal of the algorithm is to determine, for each binding statement, which reference count mutations can be elided. The result of the optimization is metadata for each reference binding statement that the compiler can use to determine what RC mutations must be performed to ensure all live objects remain alive and ignore those that are unnecessary.

As an illustrative example, consider the `remove` method in 6.1 for a linked list[1]. The most obvious performance issue in this example is the loop on lines 12–14. It should also be relatively clear that modifying the reference counts for `n` and `n.next` are unnecessary. When the loop terminates, the RC of each `next` field will be the same as they were prior to the loop with the exception of the element that `n` refers to. When `n` eventually goes out of scope, the RC of the object it refers to is also restored to its original value. The code at lines 16–19, on the other hand, must modify reference counts; if it doesn't the object referred to by `n.next` will never be deallocated.

With a bit of reasoning, we can determine what should happen; the difficulty lies in how to

[1]We assume all the pertinent structures and types are defined.

instruct a compiler how to do this for an arbitrary fragment of code. To accomplish this, the code that can be optimized must first be determined. Any given function or method is composed of a number of statements that can be arbitrarily grouped together into code fragments so long as they remain in program order. For example, the code in 6.1 can be grouped into two fragments: lines 6–9 and 11–19. This optimization will work only on fragments of code that conform to specific criteria governed by the following two rules:

1. Only code that is race-free can be considered.

2. Code execution must be compile-time deterministic.

In Pretzel, a reference declared as `auto` is guaranteed to be accessible by only a single process of execution. Therefore, Rule 1 implies that any code containing a reference declared as an `auto shared` can't be optimized with this method. Rule 2 precludes a) any decision structure (i.e., an `if/else` or `match` statement) that contains the same reference binding in two or more execution paths or b) looping structures that contain binding operations whose operands are not mutually exclusive. Any fragment code that adheres to both rules is a candidate for optimization.

Assuming a program is grouped into fragments, each reference is designated as either *stable* or *unstable*. A stable reference is one for which there is no possible way for the reference count of the object to which it points can change. The RC of the object referred to by a stable reference will have the same RC leaving the optimization as it did prior to the optimization. For example, a reference passed by value to a routine or a local reference initialized to `nil` are stable. An unstable reference is one for which no guarantees can be made with regard to its reference count — mutating an unstable reference may change the RC of an object in a way that is non-deterministic at compile time. References stored within an object or a reference variable passed by reference are examples of an unstable reference.

Given a binding statement `p1 -> p2`, the reference variables `p1` and `p2` can be either stable or unstable. This gives four possible scenarios for determining which reference counts must be updated (the subscript after the reference variable indicates if it's stable or unstable).

1. $\texttt{p1}_s$ `->` $\texttt{p2}_s$: reference counts do not need to be modified and both variables remain stable.
2. $\texttt{p1}_u$ `->` $\texttt{p2}_u$: Both variables are unstable, therefore both remain unstable and the reference count for `p1` must be decremented and incremented for `p2`.
3. $\texttt{p1}_u$ `->` $\texttt{p2}_s$: The unstable variable `p1` is being bound to a stable reference. Therefore, the reference count only for `p2` need be decremented and `p2` becomes a stable reference.
4. $\texttt{p1}_s$ `->` $\texttt{p2}_u$: The results of this binding operation cannot be determined from the operation alone — the subsequent statements of the code fragment being optimized determine the final stability states of `p1` and `p2`.
   (a) If, after the optimization process completes and the stability state of `p1` is unchanged, the binding operation does not need to modify reference counts.

(b) Otherwise, `p1` transitions to unstable, the reference count of the object referred to by `p2` must be incremented.

The first three cases are reasonably intuitive. The stability for case 4 is a bit more complicated and is determined as the optimization proceeds through each statement in the code fragment. If the target `p2` of the binding operation in 4 is later rebound to another reference *and*, subsequent to that, the destination operand `p1` of the operation is used in a statement, then 4b applies, otherwise the optimized case 4a may be used.

6.1.2 The Algorithm

The goal of the algorithm is to build a sequence of stable and unstable reference sets for each statement in a program fragment containing a reference variable. We assume the compiler correctly identifies a program fragment F that qualifies for this optimization as described above. Each fragment F is described by the set of reference variables V used in F and the three sequences, T, S, and U, where:

- T is a sequence of program statements in F. Each T_i in T is the 3-tuple $\langle r_{\text{src}}, r_{\text{dst}}, N\rangle$ where
  - r_{dst} is the mutated reference variable that is the destination of a reference binding operation or $\emptyset$ if T_i is not a binding operation.
  - r_{src} is the source operand of a reference binding operation or $\emptyset$ if T_i is not a binding operation.
  - N is the set of reference variables used in T_i where $N \neq \emptyset$ and $r_{\text{dst}} \notin N$.
- S is a sequence of sets where each S_i contains stable references for the statement T_i. When the algorithm begins, S contains only the initial stable set of references, S_0 determined by the compiler.
- U is the same as S, but for unstable references.

The algorithm for determining stability sets is shown in Figure 6.2.

6.1.3 Performance Impact of Reference Counting

Figure 6.3 shows the simulated performance comparisons between the versions of the `remove` function without reference counting, non-optimized reference counting, and optimized reference counting. Both results show the the significant performance impact of naive reference counting. Somewhat unexpected results are shown for the ARM platform where the optimized reference counting performs on-par with no reference counting[2].

[2]The reasons for these results are unclear. Identical code was run on both platforms, however different compilers were used. It's possible the ARM GCC compiler used an optimization technique the IA64 Clang compiler did not; but that's just speculation.

```
1:  for i ← 1, i ≤ |T| do
2:      if r_dst(T_i) ∈ U_{i−1} and r_src(T_i) ∈ S_{i−1} then
3:          S ← (S, S_{i−1} ∪ {r_dst(T_i)})
4:          U ← (U, U_{i−1} − {r_dst(T_i)})
5:      else if r_dst(T_i) ∈ S_i and r_src(T_i) ∈ U_i then
6:          j ← i + 1
7:          S_tmp ← S_{i−1}
8:          U_tmp ← U_{i−1}
9:          while j ≤ |T| do
10:             if r_dst(T_j) = r_src(T_i) then
11:                 while j ≤ |T| do
12:                     if r_dst ∈ N(T_j) then
13:                         S_tmp ← S_tmp − {r_dst(T_i)}
14:                         U_tmp ← U_tmp ∪ {r_dst(T_i)}
15:                     end if
16:                     j ← j + 1
17:                 end while
18:             end if
19:             j ← j + 1
20:         end while
21:         S ← (S, S_tmp)
22:         U ← (U, U_tmp)
23:     else
24:         S ← (S, S_{i−1})
25:         U ← (U, U_{i−1})
26:     end if
27: end for
```

Figure 6.2: RC Optimization Algorithm

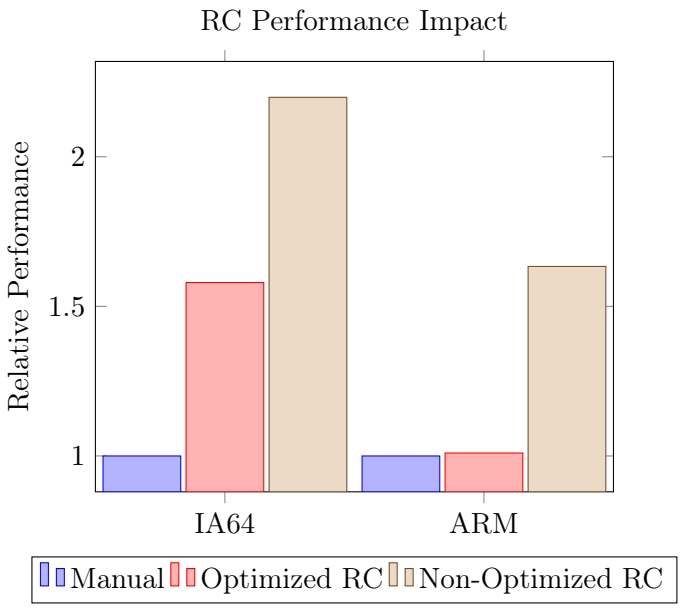

Figure 6.3: RC Performance. This figure shows the performance of a function similar to that in Figure 6.1 using normal C pointers, non-optimized reference counted pointers, and a reference counted version with the optimizations presented in this section.

6.2 Non-Recursive Deallocation

When the reference count of an object reaches zero, the destructor for that object is executed and the reference counts of any pointers in the object are decremented. If decrementing one of these reference counts results in a count of zero, that object is destroyed, and so on. This process is naturally recursive, is easy for a compiler to implement, and generally works well. However, when certain data structures (i.e., linked lists) are combined with limited stack space, stack overflows are a distinct possibility. While this particular problem can be avoided by removing elements from such data structures in a specific manner, this requires specific code to be written by a programmer. This section presents a non-recursive deallocation method for these types of data structures with the following guarantees:

- The algorithm must be sufficiently generic so as to allow a compiler to generate representative code.
- Recursion must be bounded to a constant depth that can be determined at compile time.
- No additional memory may be dynamically allocated during the deallocation process. If a deallocation occurs when heap memory has been exhausted and the deallocation process were to require heap memory, a deadlock would occur.
- The algorithm must complete in $O(n)$ time (which would be comparable to a hand-coded solution).

```
1:  procedure DEALLOCATE(O)
2:      repeat
3:          for i ← 0, i < |T| do
4:              FREED ← false
5:              while |L_i| > 0 do
6:                  FREED ← true
7:                  r ← L_{i,1}
8:                  L_i ← (L_{i,2}, L_{i,3}, ..., L_{i,n})
9:                  D ← 𝒟(r)
10:                 L ← ⟨(L_1, D_1), (L_2, D_2), ..., (L_n, D_n)⟩
11:             end while
12:         end for
13:     until FREED = false
14: end procedure
```

Figure 6.4: Non-Recursive Deallocation Algorithm.

6.2.1 Abstract Algorithm

To construct such an algorithm, we begin by viewing a type as a directed graph where each node is a type and an edge is a reference to another type. If the graph forms any strongly connected components, the type structure of the object being deallocated may produce the behavior described above. There are several algorithms to identify strongly connected components [35, 38] that a compiler could use to perform this analysis. For types without this structure, the simple recursive deallocation process is used. This algorithm applies to those types that do exhibit this structure. The following definitions are used in the algorithm presented in Figure 6.4.

1. T is the set of unique types found in the strongly connected component graph.

2. `FREED` is a Boolean flag indicating at least one object was deallocated.

3. L is the N-tuple $\langle L_1, L_2, \ldots, L_{|T|}\rangle$ where $N = |T|$ and each L_i is a sequence $(L_{i,n})_{1 \leq i \leq |T|, n \in \mathbb{N}}$ of references for a given type T_i. Each $L_{i,n}$ denotes the nth element in the sequence L_i and is a reference of type T_i.

4. The function $\mathcal{D}$ calls object O's destructor, releases the memory associated to O, and returns the N-tuple D. The N-tuple D is defined in the same way as L, but contains only references to objects with types in T that are contained within O.

We stated the algorithm must complete in $O(n)$ time. The outer "repeat" loop isn't necessary and can be rewritten as a conditional "break" after the while loop. The number of iterations the "for" loop executes is a compile-time constant. Only the "while" contributes a linear running time to the algorithm so it runs in $O(n)$ time. The operation of the algorithm is described as follows:

1. Lines 3–12: Process each distinct type found in the strongly connected components of the graph.
2. Lines 5–11: Keep deallocating objects of type T_i until the sequence is empty.
3. Line 6: Indicate a deallocation occurred. When one deallocation occurs, a reference may be added to any of the sequences in L. The algorithm checks at line 13 and makes another complete pass through all types before terminating if a single deallocation occurred.
4. Lines 7,8: Set r to the first element of the sequence L_i and then remove it from L_i.
5. Line 9: Call the destructor for the object referenced by r.
6. Line 10: Append the new references from Line 9 to the sequences of the appropriate type.

6.2.2 Implementation Details

The algorithm in Figure 6.4 presents an abstract description, but omits some important low-level details. Specifically, the algorithm is using sequences so it stands to reason a real-world implementation would need to do so as well. Each object being deallocated needs to be placed on a list for that type. Fortunately, Pretzel defines the reference count field as one machine word — big enough to hold a link pointer for this purpose. Since the object is being deallocated, the reference count field of the object is no longer needed for tracking references and can be re-purposed as a link pointer. A pointer to the head of the list must also be maintained for each type in T; $|T|$ machine words are needed for this. In practice, $|T|$ will be a very small number and the list pointers can be stored on the runtime stack.

6.2.3 Performance

To measure the efficiency of this algorithm, a standard linked list and binary tree were implemented in C++ using manual memory management to establish a "best-case" performance baseline. That code was then modified with a hand-coded version of the algorithm presented in this section. For both versions of the linked list, the list was populated with an increasing number of elements and then destroyed. The same process was followed for the two implementations of the binary tree with the tree being populated such that it was balanced.

Figure 6.5a show the results of the simulated algorithm against the manually coded version for a binary tree and Figure 6.5b show results for a linked list. Although difficult to see in the charts, the performance for the algorithm described in this section performs better up to a point in both cases. However, as the size of the structure increases, performance decreases by a constant scalar factor as compared to the hand-coded deallocation.

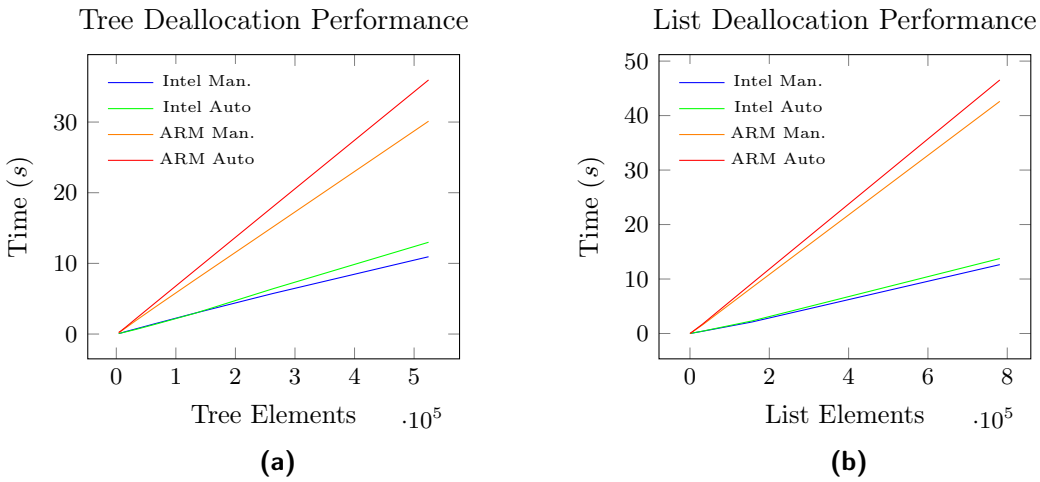

Figure 6.5: Non-Recursive Deallocation Performance. The charts in the above figure show performance of the described algorithm to avoid recursion that occurs when a recursive data structure is deallocated. The "Auto." in the key represents the algorithm described in this section and "Man." is manual hand-coded deallocation.

CHAPTER 7
FUTURE WORK AND CONCLUSION

On language design Landin [23] remarks in the paper "The Next 700 Programming Languages":

> "...we must systematize their design so that a new language is a point chosen from a well-mapped space, rather than a laboriously devised construction."

With this in mind, Pretzel has been designed for the domain of embedded and cyber-physical systems. Here we present future directions of research for Pretzel, some unsolved challenges that relate to low-level programming, and our conclusion.

7.1 Challenges and Future Work

Formal Soundness. The type system has been described with sufficient detail that they *should* provide the guarantees that are promised. Some properties have been proven, while others are taken as true based on informal reasoning. As a result, the complete formal soundness of Pretzel's reference types has not been proven. While not strictly necessary, a proof of the reference types would be complimentary to the safety goals of the language. If this can be done for Pretzel remains to be seen.

The Compiler. As a hypothetical language, Pretzel provides an effective mechanism to explore various concepts and ideas related to low-level programming. However, from a practical point of view, it's not overly useful. Additionally, it's not possible to make a real-world assessment. Adding safety to a language comes with some degree of unavoidable runtime overhead. Quantifying this overhead requires a compiler implementation to allow comparisons with other languages. Clearly, a working compiler implementation is one direction of future work.

Simplification. A common trait of many of the most successful languages is that of simplicity. A good language is naturally expressive and allows a programmer to transform abstract designs into concrete representations in code. Languages features that are complex and infrequently used become esoteric and can create code that is less readily understood. Not only is it easier for programmers to learn a simple language, it's also easier to develop the tools necessary to support the it (e.g.: compilers, debuggers, etc.). For embedded systems, where target hardware platforms have greater variety than other domains, this important. Implementing tool chains on multiple platforms becomes notably more difficult when the language itself is difficult to implement.

Although Pretzel avoids the burden of programmer-supplied annotations and provides useful defaults to common variable declarations, there is a degree of complexity that is added with multiple

reference types. Each of the three reference types have different semantics and runtime characteristics representing a marked increase in complexity over the homogeneous reference type as those found in C. Pretzel would benefit from any simplifications that can be made in this regard.

Safe Unsafe References. Pretzel reference types guarantee safety within the confines of the type system. However, embedded systems must often interact with hardware that does not conform to the rules of a type system. Interfacing with a C library or implementing a memory allocation scheme also requires the use or creation of code that does not conform to the type rules of the language. A mechanism that offers some safety guarantees to these types of programming patterns with negligible runtime impact would be a very interesting area of research.

Concurrency. We have occasionally alluded to concurrency throughout this work, but haven't provided any specific details on how concurrency interacts with the reference types described. With the prevalence of cheap multi-core processors, this is an area of Pretzel that needs to be completed. Furthermore, for efficiency reasons, event-based systems are frequently used to manage asynchronous tasks. However, event-based systems can be notoriously difficult to develop and scale poorly when complexity rises. Traditional thread models are generally avoided due to the high cost of context switching and high memory overhead. There exists a body of work that addresses this issue (mostly in relation to C), but still lack some desired traits. These concepts need to be incorporated into Pretzel.

Adopting a Language. Pretzel was designed with the intent to be useful in real-world applications. By far, the greatest hurdle faced by Pretzel (or any other new language) is the adoption by enough programmers to justify any long-term development and maintenance. Furthermore, organizations are generally reluctant to invest development resources into a tool that has questionable long-term viability. This creates a "catch-22" that is difficult to overcome.

7.2 Conclusion

Most application domains have seen regular improvements to language features and, possibly more importantly, to the safety guarantees such languages provide. Low-level programming is the exception where the vast majority of applications are written in C and, to a lesser extent, C++. With its weak type system, both C and C++ fail to recognize a number of common programming errors not found in other languages. In a field where safety and reliability are critical, low-level programming is long overdue for an update.

This work has introduced Pretzel: a hypothetical language designed to provide high-level language features with the low-level control that is necessary for the development of software used in embedded and cyber-physical systems. As one of the central aspects of low-level programming, references provide programmers with a great deal control in the development of software while simultaneously introducing semantics that can diminish the safety guarantees of a language. This work has centered on providing safety to reference types while simultaneously providing the flexi-

bility and (in theory) high performance. Pretzel achieves this by unifying three distinct reference types, each with their own advantages, into a single type system. This allows a programmer to select a reference type best suited for a given purpose while still retaining the ability to manipulate references in a semantically consistent manner.

In addition, this work has proposed two approaches to improve what are considered common deficiencies in automatic reference counting. Managing reference cycles in ARC is addressed by requiring the programmer to explicitly free memory. However, the implementation of ARC in Pretzel still guarantees dangling pointers are not possible. The performance impact of reference count updates is minimized by leveraging properties of Pretzel's reference types. Lastly, a non-recursive method for deallocating objects is also given.

APPENDIX A
FORMAL LANGUAGE GRAMMAR

This apendix provides the complete grammar for the language.

A.1 Comments

Comments in Pretzel are equivalent to those in C or other C-like languages and come in two forms: line comments and block comments. Line comments begin with `//` and continue until a newline is reached. Block comments begin with `/*` and continue until a closing `*/` is reached. Nesting of block comments is permitted.

A.2 Characters and Numbers

binary digit =	"`0`" | "`1`" ;
decimal digit =	"`0`"... "`9`" ;
hex digit =	*decimal digit* | "`a`"... "`f`" | "`A`"... "`F`" ;
special character =	"`~`" | "`-`" | "`!`" | "`#`" | "`%`" | "`^`" | "`&`" | "`*`" | "`_`" | "`+`" | "`=`" | "`|`" | "`<`" | "`>`" | "`?`" | "`/`" ;
character =	*decimal digit* | *special character* | "`a`"... "`z`" | "`A`"... "`Z`" ;

A.3 Operators and Punctuation

The following tokens are reserved for operators and punctuation and can't be used in an identifier.

```
@       :       ...     (       )
:=      ->      .       {       }
,       `(      )`      [       ]
```

A.4 Identifiers

Identifiers form elements of a program such as types or variables. Each identifier is a sequence of one or more characters or digits. Identifiers may also contain any character that is not an operator or delimiter and may begin with a digit. A qualified identifier is a sequence of two or more identifiers

separated by the selection operator (“`.`”). Qualified identifiers are used to select fields from a type or module.

simple ident = *character* − *decimal digit*, `{` *character* `}` ;

qualified ident = *simple ident*, “`.`”, `{` *simple ident*, “`.`” `}`, *simple ident* ;

ident = *simple ident* | *qualified ident* ;

A.5 Keywords

The following is a list of reserved keywords and may not be used as identifiers.

```
as          async       break       const       continue
dynamic     else        enum        for         fun
if          impl        import      interface   let
local       match       meta        method      module
nil         public      rec         return      shared
static      this        type        union       unique
verbatim    wait        with        while       yield
```

A.6 Literals

Literals represent various constant values in the language.

literal = *int literal*
| *float literal*
| *string literal*
| *array literal* ;

A.6.1 Integer Literals

Integer literals are a sequence of digits that represent either a signed or unsigned integer value. A prefix can be used to specify a binary or hexadecimal representation of an integer. By default, integer literals are signed, but can be postfixed with a `u` or `U` to specify an unsigned value.

bin literal = "0", ("b" | "B"), *binary digit*, { *binary digit* } ;

dec literal = ["+" | "-"], *decimal digit*, { *decimal digit* } ;

hex literal = "0", ("x" | "X"), *hex digit*, { *hex digit* } ;

int literal = (*bin literal* | *dec literal* | *hex literal*), ["u" | "U"] ;

A.6.2 Floating-Point Literals

decimals = *decimal digit*, { *decimal digit* }, ";" ;

exponent = ("e" | "E"), ["+" | "-"], *decimals*, ";" ;

float literal = "." , *decimals*, [*exponent*]
| *decimals*, "." , *decimals*, [*exponent*]
| *decimals*, *exponent*, ";" ;

A.6.3 String Literals

String literals are a sequence of zero or more characters enclosed by double quotes (`"`). A character may either be Within a string literal, the newline (`U+000A`), carriage return (`U+000D`), `"`, and can't appear as an unescaped character.

newline = ? Unicode code point U+000A ? ;

carriage return = ? Unicode code point U+000D ? ;

string char = ? Valid unicode code point ? − (*newline* | *carriage return* | "" | "\") ;

escaped char = "\", ("t" | "n" | "" | "\" | ("u" | "U"), "{", *hex digit*, [*hex digit*], "}") ;

string literal = "", { *string char* | *escaped char* }, "" ;

array literal = "[", { *expr* }, "]" ;

A.7 Types

base types = *ident* | *parametric type* | *fun type* ;

type = (["`const`" | "`meta`"], *base types* | *ref type*) | *array type* ;

A.7.1 Function Type

fun arg = *ident*, ("`:`" | "`:=`"), (*type* | *expr*) ;

fun arg list = "`(`", [*fun arg*, { *fun arg* }, ["`...`"]], "`)`" ;

result = ("`:`" | "`:=`"), ("`nil`" | *type*) ;

fun type = "`fun`", *fun arg list*, *result* ;

A.7.2 Parametric Types

parametric type = "`{`", *ident*, (*type* | *expr*), { (*type* | *expr*) }, "`}`" ;

A.7.3 Reference Types

ref qualifier = "`static`"
| "`unique`"
| "`dynamic`", ["`shared`"]
| ;

ref type = *ref qualifier*, (([*mutability qualifier*], *base type*) | *array type*) ;

A.7.4 Array Types

array type = "`[`", [*expr*], "`]`", *type* ;

A.8 Expressions

expr = *ident* | *literal* | *call expr* | *if expr* | *match expr* ;

A.8.1 Call Expression

call expr = "(", *ident*, { *expr* }, ")" ;

A.8.2 Conditional Expression (If/Else)

if expr = "(", "if", *expr*, *expr*, { "else", ",", "if", *expr*, *expr* }, "else" *expr*, ")" ;

A.8.3 Match Expression

match opts = ["as", *simple ident*], ["when", *expr*] ;

match expr = "(", "match", *expr*,
{ *expr*, { *expr* }, *match opts*, "=>", *expr* },
["else", *match opts*, "=>", *expr*]
")" ;

A.8.4 Memory Allocation Expression

mem alloc expr = *ref qualifier* − "static", *call expr* ;

A.9 Statements

stmt =
  let stmt
| *assignment stmt*
| *bind stmt*
| *if stmt*
| *match stmt*
| *block stmt*
| *verbatim stmt*
| *ctrl stmt*
| *import stmt*
| *module stmt*
| *expr*
| ;

A.9.1 Variable Assignment

assignment stmt = *expr*, ("=" | "->"), *expr* ;

A.9.2 Variable Declaration

let stmt = "`let`", *simple ident*, (":", *type*) | (("=" | "->"), *expr*) ;

A.9.3 Branch Control Structures

ctrl stmt =
  "`return`", [*expr*]
| "`yield`", [*expr*]
| "`break`"
| "`continue`" ;

A.9.4 Conditional Statement

if stmt = *expr*, { *stmt* }, { "`else`", "`,`", "`if`", *expr*, { *stmt* } }, ["`else`", { *stmt* }] ;

A.9.5 Match Statement

match stmt = "`match`", *expr*,
{ *expr*, { *expr* }, *match opts*, "`=>`", { *stmt* }, "`;`" },
["`else`", *match opts*, "`=>`", { *stmt* }, "`;`"]
"`;`" ;

A.9.6 Verbatim Block

verbatim = "`verbatim`",
{ *stmt* } "`;`" ;

A.10 Definitions

A.10.1 Function Definitions

parametric = *simple ident*, "`:`", (*type* | *expr*), { "`,`", (*type* | *expr*) } ;
parametric list = "`{`", *simple ident*, "`:`", "`}`" ;
function def = "`fun`", ["`meta`"], [*simple ident*], [*parametric list*], *fun arg list*, [*fun result*]
{ *stmt* }
"`;`" ;

A.11 Interfaces

interface def = "`interface`", *simple ident*,
{ ("`fun`" | "`method`"), *simple ident*, *fun arg list*, *fun result*, "`;`" }
"`;`" ;

A.12 Type Definitions

A.12.1 Methods

method ident = "`.`", *simple ident* | *simple ident*, ["`.`", *simpleident*] ;
method def = "`method`", [*method ident*], [*parametric list*], *fun arg list*, [*fun result*]
{ *stmt* }
"`;`" ;

A.12.2 Implementations

impl block = "`impl`", *ident*
{ *function def* | *method def* }
"`;`" ;

A.12.3 Type Definition

member var decl = *simple ident*, "`:`", *type* ;

public block = "`public`"
    { *type body* − *public block* }
"`;`" ;

type body = *member var decl*
| *function decl*
| *method decl*
| *public block*
| *impl block*
| *alias*
| *type decl* ;

type def = "`type`", ["`dynamic`" | "`meta`"], *simple ident*, [*parametric list*],
    { *type body* }
"`;`" ;

A.13 Enumerations

enum = "`enum`", *simple ident*, "`:`", *type*
    simple ident, ["`:`", *expr*]
"`;`" ;

A.14 Unions

union = "`union`", *simple ident*, [*parametric list*]
{ *simple ident*, [":", *type*] }
";" ;

A.15 Aliases

alias = "`alias`", *simple ident*, *type*

A.16 Modules

module decl = "`module`", *simple ident* ;

import stmt = "`import`", (*string literal* | *expr*), ["`as`", *simple ident*], { *simple ident* } ";" ;

A.17 Program Unit

unit body =	*unit public block*
	| *type decl*
	| *function decl*
	| *interface decl*
	| *alias*
	| *call expr*
	| *if expr*
	| *match expr* ;
unit public block =	"`public`",
	{ *unit body* − *unit public block* },
	"`;`" ;
program =	[*module decl*], { *import stmt* }, { *unit body* } ;

BIBLIOGRAPHY

[1] Andrei Alexandrescu. *The D programming language*. Addison-Wesley Professional, 2010.

[2] David F Bacon, Perry Cheng, and VT Rajan. A real-time garbage collector with low overhead and consistent utilization. *ACM SIGPLAN Notices*, 38(1):285–298, 2003.

[3] David F Bacon, Perry Cheng, and VT Rajan. A real-time garbage collector with low overhead and consistent utilization. *ACM SIGPLAN Notices*, 38(1):285–298, 2003.

[4] George E Collins. A method for overlapping and erasure of lists. *Communications of the ACM*, 3(12):655–657, 1960.

[5] Jeremy Condit, Matthew Harren, Zachary Anderson, David Gay, and George C Necula. Dependent types for low-level programming. In *Programming Languages and Systems*, pages 520–535. Springer, 2007.

[6] Karl Crary, Robert Harper, and Sidd Puri. *What is a recursive module?*, volume 34. ACM, 1999.

[7] John DeTreville. *Experience with concurrent garbage collectors for Modula-2+*. Digital, Systems Research Center, 1990.

[8] L Peter Deutsch and Daniel G Bobrow. An efficient, incremental, automatic garbage collector. *Communications of the ACM*, 19(9):522–526, 1976.

[9] Manuel Fahndrich and Robert DeLine. Adoption and focus: Practical linear types for imperative programming. In *ACM SIGPLAN Notices*, volume 37, pages 13–24. ACM, 2002.

[10] Daniel Frampton. *Garbage Collection and the Case for High-level Low-level Programming*. Citeseer, 2010.

[11] David Gay and Alex Aiken. Language support for regions. In *ACM Sigplan Notices*, volume 36, pages 70–80. ACM, 2001.

[12] David Gay, Philip Levis, Robert Von Behren, Matt Welsh, Eric Brewer, and David Culler. The nesc language: A holistic approach to networked embedded systems. In *Acm Sigplan Notices*, volume 38, pages 1–11. ACM, 2003.

[13] Jean-Yves Girard. Linear logic. *Theoretical computer science*, 50(1):1–101, 1987.

[14] Google. Go at google: Language design in the service of software engineering, 2012.

[15] Dan Grossman. Type-safe multithreading in cyclone. In *ACM Sigplan Notices*, volume 38, pages 13–25. ACM, 2003.

[16] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based memory management in cyclone. In *ACM Sigplan Notices*, volume 37, pages 282–293. ACM, 2002.

[17] Matthew Hertz and Emery D Berger. Quantifying the performance of garbage collection vs. explicit memory management. In *ACM SIGPLAN Notices*, volume 40, pages 313–326. ACM, 2005.

[18] ISO. Programming languages c. ISO 9899:TC2, International Organization for Standardization, Geneva, Switzerland, 1999.

[19] ISO. Programming language c++. ISO 14882:2011, International Organization for Standardization, Geneva, Switzerland, 2011.

[20] Trevor Jim, J Gregory Morrisett, Dan Grossman, Michael W Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of c. In *USENIX Annual Technical Conference, General Track*, pages 275–288, 2002.

[21] Pramod G Joisha. Compiler optimizations for nondeferred reference: counting garbage collection. In *Proceedings of the 5th international symposium on Memory management*, pages 150–161. ACM, 2006.

[22] Brian W Kernighan, Dennis M Ritchie, and Per Ejeklint. *The C programming language*, volume 2. prentice-Hall Englewood Cliffs, 1988.

[23] Peter J Landin. The next 700 programming languages. *Communications of the ACM*, 9(3):157–166, 1966.

[24] Yossi Levanoni and Erez Petrank. An on-the-fly reference-counting garbage collector for java. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 28(1):1–69, 2006.

[25] Philip Levis, Sam Madden, Joseph Polastre, Robert Szewczyk, Kamin Whitehouse, Alec Woo, David Gay, Jason Hill, Matt Welsh, Eric Brewer, et al. Tinyos: An operating system for sensor networks. In *Ambient intelligence*, pages 115–148. Springer, 2005.

[26] Henry Lieberman and Carl Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6):419–429, 1983.

[27] Bruce J MacLennan. Values and objects in programming languages. *ACM SIGPLAN Notices*, 17(12):70–79, 1982.

[28] Nicholas D Matsakis and Felix S Klock II. The rust language. *ACM SIGAda Ada Letters*, 34(3):103–104, 2014.

[29] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Commun. ACM*, 3(4):184–195, April 1960.

[30] Steven S. Muchnick. *Advanced compiler design implementation*. Morgan Kaufmann, 1997.

[31] George C Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. Ccured: type-safe retrofitting of legacy software. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(3):477–526, 2005.

[32] Elizabeth D Rather, Donald R Colburn, and Charles H Moore. The evolution of forth. In *ACM Sigplan Notices*, volume 28, pages 177–199. ACM, 1993.

[33] Rifat Shahriyar, Stephen M Blackburn, and Daniel Frampton. Down for the count? getting reference counting back in the ring. In *ACM SIGPLAN Notices*, volume 47, pages 73–84. ACM, 2012.

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

[35] Micha Sharir. A strong-connectivity algorithm and its applications in data flow analysis. *Computers & Mathematics with Applications*, 7(1):67–72, 1981.

[36] Bjarne Stroustrup. Multiple inheritance for c+. *Computing Systems*, 2(4), 1989.

[37] S Tucker Taft. *Ada 2005 Reference Manual. Language and Standard Libraries: International Standard ISO/IEC 8652/1995 (E) with Technical Corrigendum 1 and Amendment 1*, volume 4348. Springer, 2006.

[38] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1(2):146–160, 1972.

[39] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and computation*, 132(2):109–176, 1997.

[40] Todd L Veldhuizen. C++ templates as partial evaluation. *arXiv preprint cs/9810010*, 1998.

[41] Philip Wadler. Linear types can change the world. In *IFIP TC*, volume 2, pages 347–359. Citeseer, 1990.

[42] David Walker and Kevin Watkins. On regions and linear types. In *ACM Sigplan Notices*, volume 36, pages 181–192. ACM, 2001.

[43] Paul R Wilson. Uniprocessor garbage collection techniques. In *Memory Management*, pages 1–42. Springer, 1992.