A Functional Approach to

Memory-Safe Operating Systems

by

Rebekah Leslie

A dissertation submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy
in
Computer Science

Dissertation Committee:
Mark P. Jones, Chair
James Hook
Andrew Tolmach
Jonathan Walpole
Gernot Heiser
Jeanette R. Palmiter

Portland State University

ABSTRACT

Purely functional languages—with static type systems and dynamic memory management using garbage collection—are a known tool for helping programmers to reduce the number of memory errors in programs. By using such languages, we can establish correctness properties relating to memory-safety through our choice of implementation language alone. Unfortunately, the language characteristics that make purely functional languages safe also make them more difficult to apply in a low-level domain like operating systems construction. The low-level features that support the kinds of hardware manipulations required by operating systems are not typically available in memory-safe languages with garbage collection. Those that are provided may have the ability to violate memory- and type-safety, destroying the guarantees that motivate using such languages in the first place.

This work demonstrates that it is possible to bridge the gap between the requirements of operating system implementations and the features of purely functional languages without sacrificing type- and memory-safety. In particular, we show that this can be achieved by isolating the potentially unsafe memory operations required by operating systems in an abstraction layer that is well integrated with a purely functional language. The salient features of this abstraction layer are that the operations it exposes are memory-safe and yet sufficiently expressive to support the implementation of realistic operating systems. The abstraction layer enables systems programmers to perform all of the low-level tasks necessary in an OS implementation, such as manipulating an MMU and executing user-level programs, without compromising the static memory-safety guarantees of programming in a purely functional language. A specific contribution of this work is an analysis of memory-safety for the abstraction layer by formalizing a meaning for memory-safety in the presence of virtual-memory using a novel application of non-interference security policies. In addition, we evaluate the expressiveness of the abstraction layer by implementing the L4 microkernel API, which has a flexible set of virtual memory management operations.

DEDICATION

For Joe Hurd, who inspires me every day and without whose loving support this work would not have been possible.

ACKNOWLEDGMENTS

This research could not have been accomplished without the help of countless others. First and foremost, I would like to thank my advisor, Mark Jones, for his unwavering support throughout my graduate school career. In my early days as a student his courses inspired and challenged me, and our initial projects together set the direction for my research career. In the years that followed, Mark has made every effort to be involved with my work and to help me succeed. This work has benefited immensely from his contributions, in both research content and in presentation.

The members of the Hasp and Programatica research projects have always provided an intellectually stimulating and friendly climate in which to conduct research. Andrew Tolmach made significant contributions to the design of the abstraction layer presented here, including the memory-safety analysis, and directly influenced its implementation through his work on the House operating system and the original H interface. James Hook has also been extremely supportive, encouraging me to see the bigger picture and the broader connections of my work. My fellow students have provided me with many valuable discussions and pointers over the years. I am particularly grateful to Iavor Diatchki and Emir Pašalić, who befriended me in those early days when graduate school was a frightening and overwhelming place. I am also very grateful to Tim Chevalier and John Ochsner for proofreading this document.

I owe a great debt to my colleagues upon whose work my abstraction layer

is based, as well as the maintainers of the bare metal Haskell compiler. Thomas Hallgren made significant contributions to the House operating system and the first version of the H interface, and has always been available to answer questions and to help with debugging. Kenneth Graunke created the version of the Haskell runtime system that my implementation relies on, leveraging the work of Adam Wick. In addition to his technical contributions, Adam has been an excellent resource throughout the process of writing my dissertation, providing a rare example of person in the local functional programming community doing primarily low-level systems work. Don Stewart taught me about compiler pragmas in GHC and helped me to optimize my L4 implementation.

I would also like to thank Rex Page at the University of Oklahoma. He gave me my first start in research and functional programming as an undergraduate and without his guidance I would not be where I am today.

Finally, I would like to thank Joe Hurd. There are not words to express how valuable his love and support have been in enabling me to complete this process. Moreover, the quality of this work has benefited immeasurably from our countless discussions and from his proofreading efforts.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

Software correctness is a persistent challenge that is increasingly relevant as computers become more seamlessly integrated with every aspect of modern life. Software controls the flight systems in planes and the brake systems in cars; a single failure could have a catastrophic effect. Software protects our private health information and sensitive financial data; identity theft is as simple as a software breach. We rely on software for so much, yet we still do not have sufficient techniques for assuring that it will not go wrong. In 1996, the Ariane 5 rocket exploded less than 40 seconds after lift-off due to an incorrect conversion of a 64-bit floating point number into a 16-bit integer [72]. More recently, undisclosed software problems with the braking system of the Toyota Prius automobile led to a recall of more than 125,000 cars [1]. Huge public failures like these show that mistakes can get through, even in well tested and widely deployed systems. Despite all of the software engineering and validation techniques that have been developed over the years, it is still not easy to produce software that is correct and secure.

One source of difficulty is that many real-world systems are written in low-level programming languages like C or assembly language. Any complex program is likely to contain bugs during development, but these languages lack strong static typing and memory-safety guarantees that can protect against common programming errors. Memory-safety violations—such as dereferencing a null pointer, writing beyond the bounds of an array, failing to free memory, or freeing memory more than once—are particularly prevalent in programs written in these languages due to the direct and manual way that memory is managed.

Over the past few years, Coverity's Scan project [15] has embarked on massive studies using static analysis tools to catalog the number and kind of errors that exist in large, real-world, open source programs. Their 2009 study, for example, covered 11.5 billion lines of code from 280 open source projects including the Linux, FreeBSD, and NetBSD operating systems [14]. According to the Scan project data, null pointer dereferences and resource leaks together make up roughly 50% of the errors found in open source software in both the 2008 and 2009 studies (out of 27,752 defects found in the 2008 study and 38,453 defects found in the 2009 study). We classify both null pointer dereferences and resource leaks as *memory-safety violations*. Other memory-safety violations—such as using memory after it has been freed, accessing beyond the end of an array (buffer overflows), and casting a value to a type with a larger representation (allowing the programmer to inadvertently overwrite an adjacent value)—are also prevalent in the code studied by Coverity; these errors account for more than 10% of the remaining bugs. Buffer overflows and failure to release resources properly also made it into the 2009 CWE/SANS Top 25 Most Dangerous Programming Errors list, a collaboration between the SANS Institute and MITRE to identify the worst security risks in software today [83]. Clearly, mistakes stemming from memory-safety violations are a serious issue in real software.

With software failures becoming so common, even in widely deployed systems, it is obvious that current techniques for implementing and vetting software are not sufficient for preventing errors. The hardware industry faced a similar crossroads after the famously expensive Intel floating-point division bug in 1994 [50]. Since then, formal verification has become a major component of the hardware development process at Intel and other major manufacturers [42]. After proving that the hardware designs and implementations are correct, there is less that can go wrong when the systems are deployed to end users. Formal verification is a powerful technique for increasing assurance and reducing bugs that can be applied to components of the software stack to increase reliability in that domain as well.

1.1 CORRECTNESS IN OPERATING SYSTEMS

Operating systems are a particularly important category of software because of their role at the base of a typical software stack. An operating system performs privileged, low-level manipulations of hardware that higher level software components depend on for their functionality. If the operating system goes wrong in some way that results in a system crash, then every application running in the system will be affected. Even if the operating system does not crash, it can still leak information to the wrong place if it does not manage protection boundaries between applications correctly. The degree of centralized control that an operating system typically has in such a software stack creates a huge incentive to establish the correctness of the OS implementation. All applications that run on top of the operating system benefit from the correctness of the underlying layer, no matter what their ultimate function is, and operating system correctness provides a foundation for reasoning about the correctness of the application layer. For these reasons, correctness in the operating systems domain is a particularly important problem.

The seL4 project—a joint work by researchers at NICTA, Open Kernel Labs, and the University of New South Wales—verified the functional correctness of their microkernel design and implementation, the first time that such a feat has been accomplished for a general-purpose operating system [60]. Specifically, the team proved a refinement of an abstract functional specification to a high-performance C implementation via an executable model (derived from a Haskell prototype). In their verification effort, they focused on the following four types of properties.

- Low-level invariants that establish the memory-safety of the implementation, including, for example, that there are no null pointer dereferences.

- Typing invariants that establish the type-safety of the implementation beyond what is provided by the weak type system of C.

- Data structure invariants that establish correctness properties for data structure usage, for example, that a particular structure is accessed with consistent assumptions about the structure's layout throughout the entire program.

- Algorithmic invariants that describe kernel-specific properties of the implementation.

A key result of their proof is that the kernel is guaranteed to be free from all of the common memory-safety violations that the Coverity project typically finds in open source software: buffer overflows, null pointer dereferences, uses of pointers at the wrong type, and memory leaks. Note, however, that the project required considerable human effort: the executable model (written in Haskell) and the associated C implementation are small, containing 5,700 lines of code and 8,700 lines of code, respectively, but the verification required roughly 160,000 lines of Isabelle/HOL [76] script [60].

While this verification effort is impressive, it also demonstrates just how much work is required to verify a piece of software as complex as an operating system. The project took place over four years with at least a dozen participants, all of whom are highly trained. The project members estimate that the verification effort alone (not including the implementation of the kernel or the design effort) required 20 person years of effort [60]. Some of that work was invested in reusable infrastructure building activities, but they still estimate that a similar verification for a new kernel using the same methodology would require an additional 6 person years, and probably more for someone without the expertise and knowledge built up during the course of the seL4 verification. The verification is not something that can easily be repeated by a typical programmer, nor are the proofs directly reusable. If we design a new kernel, or even modify the current seL4 implementation, then a significant fraction of the verification effort will need to be repeated for the new software artifact. The seL4 team have verified their ARM implementation, but not their port to the x86 [60]. Certainly these artifacts could be verified as well, but this shows that, even for relatively minor implementation differences, the verification changes are time consuming enough to be a deterrent.

1.2 PURELY FUNCTIONAL LANGUAGES FOR MEMORY-SAFETY

We would like to obtain some of the assurance of a project like seL4 in a way that is more scalable and reusable, without paying such a high cost for verification again and again. One approach is to use a purely functional language[1] for the implementation, rather than a low-level language like C. Purely functional languages offer many language features with software engineering benefits, such as expressive type systems, higher-order functions, polymorphism, abstract datatypes, and powerful module systems. These mechanisms facilitate compile-time bug detection and the development of clear, concise, and modular programs with a high degree of reuse. Such benefits alone might be enough to recommend pure languages for operating system implementations, but we are particularly interested in the fact that pure functional languages provide memory-safety, garbage collection, and fine-grained effect tracking in the type system. By implementing operating systems in a purely functional language, it is possible to leverage these properties to automatically obtain many of the guarantees that the seL4 team worked so hard to prove—the type

[1] Throughout this document we will use the terms purely functional languages and pure languages synonymously to refer to typed functional languages with pure semantics and dynamic memory management using garbage collection. This feature set is typical for functional languages with pure semantics, and is assumed in our work.

system enforces them. For example, in the purely functional language Haskell [78], it is impossible, in principle, to dereference a null pointer, so we know without any testing or proofs that any code written in Haskell avoids such bugs[2]. Writing software in a purely functional language has further benefits for reasoning because the semantics are close to those of mathematical functions, so programs are amenable to formal reasoning for establishing any properties that cannot be derived from types alone.

In practice, however, it is often assumed that we cannot utilize the benefits of purely functional languages in operating systems development because of a gap between the requirements of operating systems implementations and the facilities provided by standard purely functional languages. For example, these languages provide strong safety guarantees—like memory-safety—by relying on a run-time system that controls the layout and configuration of memory (with a garbage collector to manage storage usage). However, a primary function of an operating system is to manage the memory of the machine. Most language run-times do not support the ability to access hardware memory-management facilities directly, instead, the programmer must access these hardware features through foreign calls. Foreign function interfaces allow programmers to work with low-level (and potentially unsafe) functions and values in an otherwise pure language. Declarations in the language introduce assumptions about the type of an externally defined value or function, and everything is assumed to be used in a safe way. As we will illustrate in the next section, an approach based on foreign function interface definitions enables one to write powerful programs in pure languages with limited restrictions due to the type system, at the cost of significantly weakening the safety argument for the entire program. A mistake in the foreign code can destroy the

[2] The Foreign Function Interface extension to Haskell [12] does make it possible to dereference a null pointer. In this case, we refer to Haskell 98 without extensions, but will cover the impact of the foreign function interface separately.

safety of the program completely.

1.3 EXAMPLE: LOW-LEVEL PROGRAMMING IN HASKELL

As a concrete example, let us examine the Foreign Function Interface (FFI) extension of the Haskell language [12], which enables Haskell programmers to perform a variety of low-level programming tasks. For example, the FFI provides the ability to call C functions, extending Haskell with new behaviors by adding new primitives. Of course, there is no static typing or guaranteed memory-safety for the extensions written in C, so using the FFI introduces proof obligations that must be satisfied to avoid compromising the type- and memory-safety of the language. In addition to the ability to call C functions, the FFI extension includes support libraries that introduce unchecked pointer types, pointer arithmetic operations, and assignments to memory through pointers, all within Haskell and with Haskell types. But if these primitives are used incorrectly, they can corrupt the Haskell heap, invalidating all of the Haskell type- and memory-safety properties! Essentially, the FFI extension introduces the power of manual and direct memory management that we have in C, along with all of its problems and potential for mistakes. The rationale for writing operating systems in a purely functional language was to obtain the strong safety benefits of the language, yet the facilities available for writing low-level software break those very same strong properties.

1.4 ASSURANCE THROUGH A SMALL TRUSTED COMPUTING BASE

To deal with the dangers of low-level memory manipulation, we consider an operating system implementation in a purely functional language based on the idea of a small *trusted computing base* (TCB). Lampson et al. [64] define the TCB of a system as a “small amount of software and hardware that security depends on and that we distinguish from a much larger amount that can misbehave without affecting security.” The basic idea is to isolate the essential services of the system behind a small, well-defined software component. This component must only be accessed through a restricted interface because the code in the TCB has more privileges and therefore more potential to wreak havoc if an error occurs. By keeping the TCB small and its interface well-defined, there is more hope of constructing a system that is correct overall than with a monolithic design, particularly when the behavior of higher-level layers is limited to a set of safe lower level operations. Many types of operating system kernels exist based on different TCB design principles, including separation kernels [81, 21], microkernels [11, 2, 70], and hypervisors [16, 29, 5].

In our work, we apply the trusted computing base approach to safe operating systems programming in a purely functional language. The trusted computing base should be a small set of essential services that are specifically designed for writing operating systems. These services must be written in a low-level language like C or in the unsafe portion of a purely functional language in order to perform their function. However, these services can be added to the language run-time system or packaged up as an abstraction layer that can be integrated with programs in the purely functional language, with the expectation that no code above the abstraction layer will access the unsafe facilities of the language. Purity provides the ability to distinguish contexts that rely on the safe abstraction layer from contexts that rely on unsafe language features using the type system. Figure 1.1 illustrates how the pieces fit together. For this approach to work, the operations of the abstraction layer must be sufficiently expressive to support the implementation of real operating systems in the pure functional language, with no additional reliance on a foreign function interface or other unsafe primitives. In addition, the operations themselves must be memory-safe, even though their implementations may use potentially unsafe operations on the underlying hardware. The client will

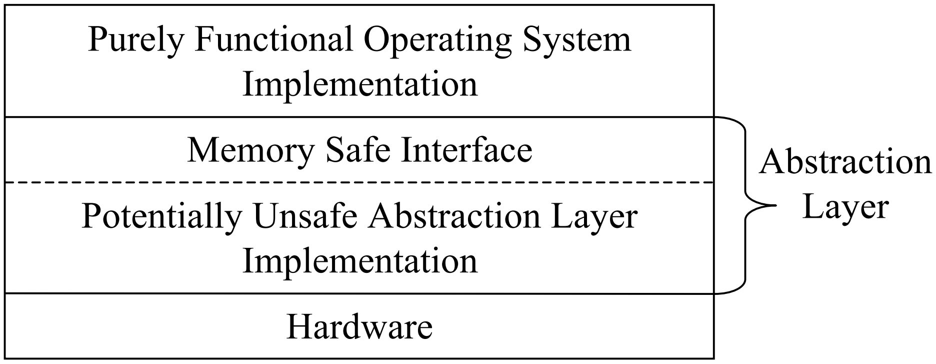

Purely Functional Operating System Implementation
Memory Safe Interface
Potentially Unsafe Abstraction Layer Implementation
Hardware
Abstraction Layer

Figure 1.1: The system organization when dividing an operating system into an unsafe abstraction layer and a purely functional layer.

not have access to pointers or direct memory writes or any of the other hazardous facilities, just those operations that are essential for writing operating systems.

1.5 THIS WORK

The primary focus of this work is to design a set of operations that is memory-safe and sufficiently expressive to support writing operating systems.

> **Thesis Statement:** The gap between the requirements of operating system implementations and the features of purely functional languages can be bridged by isolating potentially unsafe memory operations in a memory-safe abstraction layer that is well integrated with the functional language.

In this dissertation, we work specifically with Haskell as the primary implementation language, though the main techniques should work for any purely functional language and, in many respects, would even apply to the broader class of memory-safe languages, such as Java. The primary contributions of this dissertation are as follows:

- A design of a functional, memory-safe abstraction layer that is sufficiently expressive to support the implementation of real operating systems using only the operations of the abstraction layer (Chapter 4).
- A formal definition of memory-safety for our system (Chapter 5).
- A formal model of the internal operation of the abstraction layer that provides a specification of the behavior for each function in the API and connects the formal notion of memory-safety to our implementation (Chapter 5).
- An implementation of the abstraction layer design that is integrated with the purely functional language Haskell (Chapter 6).
- An implementation of the L4 microkernel API [62] in Haskell, based on the abstraction layer implementation (and without additional use of the Haskell FFI), as a demonstration that the interface is sufficiently expressive to support the implementation of realistic operating systems (Chapter 7).
- An analysis of the cost of our approach through a targeted analysis of the inter-process communication performance of the L4 implementation (Chapter 8).

In the remainder of this dissertation, we will elaborate on each of these contributions in turn. Chapters 2 and 3 review relevant background material on operating systems and Haskell programming. Chapters 4, 5, and 6 present the design, formal model, and implementation of the abstraction layer, respectively. Chapter 7 describes the L4 case study. Chapter 8 discusses the performance results and demonstrates our approach to optimizing the L4 implementation. In the remaining chapters, we review related work (Chapter 9) and present conclusions (Chapter 10).

Chapter 2

BACKGROUND: CORE CONCEPTS OF INTEL IA32 PROCESSORS

In the design of our abstraction layer, we would prefer to remain as architecture neutral as possible, but it is inevitable that certain hardware features will be exposed when constructing an abstraction that is so close to the hardware. CPU mechanisms such as registers, privileged mode execution, interrupts, and hardware-supported virtual-address translation are important features that support operating systems development. Interrupt handling plays an important role in our implementation of user program execution; we expose architecture-specific details about interrupts and registers to support client-level interrupt handlers. Memory management is an essential part of the design and memory-safety analysis for the abstraction layer; we abstract over some details in the API, but the specification and implementation rely heavily on the specific features of the underlying hardware. In this chapter, we present background material about our target architecture, the Intel IA32 platform [52], that is necessary to understand the abstraction layer design and implementation. We examine the fundamentals of the execution environment, including the register set and interrupt-handling mechanism, in Section 2.1 and review the virtual-to-physical address translation mechanism in Section 2.2. Readers already familiar with these details may skip ahead to the next chapter.

2.1 EXECUTION ENVIRONMENT

In this section, we explain the essential concepts of the execution environment available on the IA32 architecture. These facilities strongly influence the design of the lowest levels of the abstraction layer. The goal of this section is to help the reader develop enough familiarity with the hardware platform to understand the abstraction layer design and the techniques used to execute user programs in our implementation. Section 2.1.1 explains the register set of the platform; Section 2.1.2 describes the modes of execution; Section 2.1.3 discusses faults and interrupt handling; and Section 2.1.4 provides a brief introduction to port I/O. We will present the abstraction layer mechanisms for exposing these concepts to client kernels in Section 4.5, and address the implementation details in Section 6.8.

2.1.1 Registers

The registers of the machine store the execution state of the running program. The IA32 architecture has eight general purpose registers, six segment registers, a flags register that stores information about the state of the machine, and an instruction pointer that stores the location of the next instruction to execute. Figure 2.1 shows the names and sizes of these registers. Intuitively, the CPU can be programmed to switch to a new program by loading a new set of values into these registers. If we want an opportunity to return to the original program at a later stage, then we must save the register values in memory before making the switch.

General purpose registers are available to store operands to computations, results of operations, and pointers into memory. The machine associates a special semantics with the ESP register, though it is referred to as general purpose, which is designated for holding the stack pointer of programs.

The segment registers each hold a segment selector that serves as an index into the *global descriptor table*. The entries of the global descriptor table describe

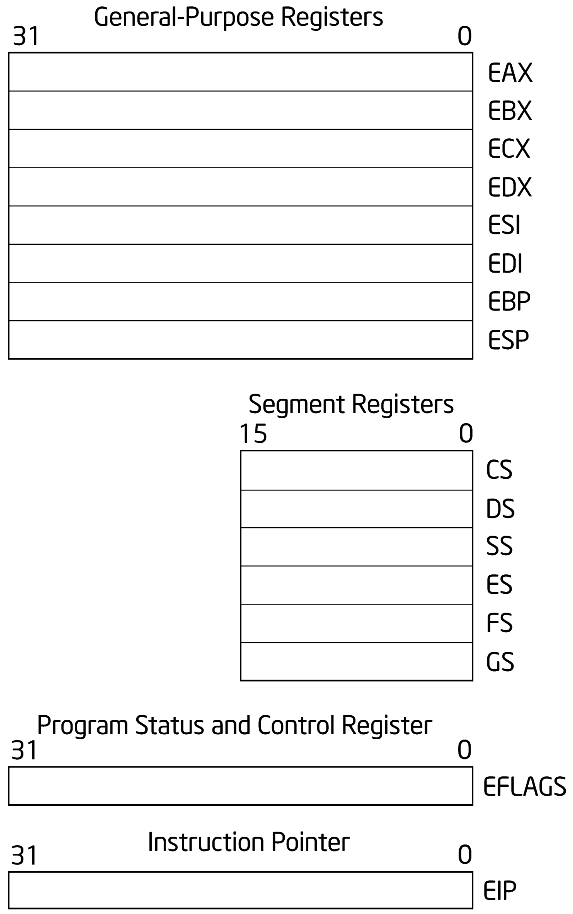


Figure 2.1: The registers of the IA32 architecture. This figure is a reproduction of Figure 3-4 from the Intel Architectures Software Developer's Manual [52].

areas of memory, called segments, that may be used in programs to control access to the linear address-space. At any given point, the CPU has access to up to six distinct segments via the six segment registers. The access rights available on each segment are defined by the corresponding entry in the global descriptor table. There is a code segment for storing instructions (pointed to by the CS register) a stack segment for storing the program stack (pointed to by the SS register), and four data segments (pointed to by the DS, ES, FS, and GS registers). In our work, we use a *flat memory model*: all of the segments overlap and contain the entire address space, as shown in Figure 2.2[1].

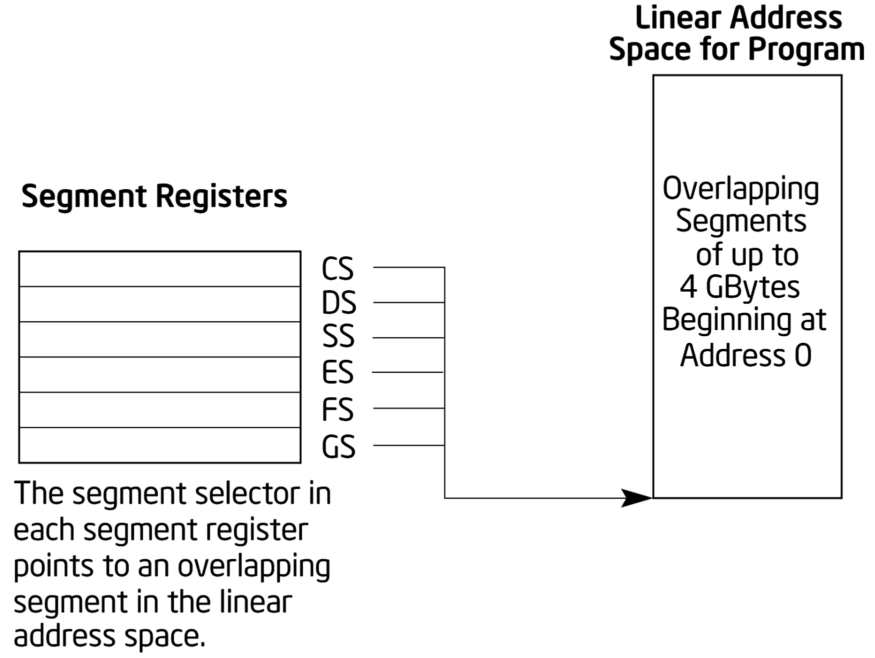

Linear Address Space for Program
Segment Registers
CS
DS
SS
ES
FS
GS
Overlapping Segments of up to 4 GBytes Beginning at Address 0
The segment selector in each segment register points to an overlapping segment in the linear address space.

Figure 2.2: A flat memory model configuration. Every segment overlaps and contains the entire address space of the machine. This figure is a reproduction of Figure 3-6 from the Intel Architectures Software Developer's Manual [52].

2.1.2 Modes of Execution

The IA32 has four modes of execution, called rings, that correspond to different privilege levels. Lower numbered rings afford the executing code greater access to hardware features. If software attempts to access functionality that is forbidden

[1] Our L4 implementation requires a special configuration of the GS register that does not fit the flat memory model to satisfy a detail of the L4 specification. Special uses of the segment registers like this can be incorporated into the implementation when necessary.

in the current ring, the hardware faults and the execution of the offending code is suspended (faults will be covered in Section 2.1.3). Ring 0 is typically used for kernel code, while applications typically run in Ring 3. Rings 1 and 2 are designed for operating system services like device drivers. Our abstraction layer implementation, like many systems, only uses two privilege levels: kernel code runs in Ring 0 (also called kernel-mode or supervisor mode) and all other code runs in Ring 3 (also called user-level mode).

2.1.3 Faults and Interrupts

During the course of execution, running software may encounter an interrupt or exception. There are three sources of interrupts and exceptions: external interrupts from hardware devices, programmable software interrupts (triggered using the `INT n` instruction), and faults due to program error conditions (such as division by zero or attempts to access to supervisor-level hardware features from user-level). For the purposes of this thesis, we will ignore the distinction between interrupts and exceptions and will refer to both kinds of events simply as interrupts.

When an interrupt occurs, the CPU branches out of the currently executing program to the handler for that particular event. A special data-structure called the *interrupt descriptor table* (IDT) specifies the mapping between interrupts and their associated handlers. Each interrupt is identified by a unique number called an *interrupt vector*. When an interrupt with a particular vector number occurs, control transfers to the handler associated with that vector in the IDT.

For code executing in user-mode, the occurrence of an interrupt causes the processor to switch privilege level and begin executing in kernel-mode. The hardware sets a new stack segment value and stack pointer so that the kernel-mode interrupt handler does not need to share a stack with user programs. The values describing the handler stack come from the *task-state segment* (TSS), which is a special segment on IA32 that supports hardware-assisted task-switching. The abstraction layer implements task management in software, but we make fundamental use of the ability to install an interrupt-handler stack via the TSS. As we will see in Section 6.8, we use the interrupt-handler stack for saving the register state of user programs between executions. The client kernel may save the state of multiple user programs simultaneously by creating multiple interrupt-handler stacks (called *fault-contexts* in our design). These interrupt-handler stacks are only used for saving the state of user programs—the Haskell code runs using a separate kernel stack.

At the start of an interrupt handler, the hardware automatically pushes certain parts of the register-state onto the stack. For user-mode interrupts, this state contains the values of the instruction pointer, stack pointer, flags register, code segment register, and stack segment register of the executing process. For kernel-mode interrupts, the handler runs on the same stack as the original code, so the hardware does not save the stack segment or stack pointer. In our abstraction layer implementation, we save additional information about the state of user-mode programs so that we have enough information to resume the program after handling the interrupt. This information includes the general purpose registers, the data segment registers, and the error code value for the interrupt (for faults that include supplemental information about the nature of the exception in the form of an error code). The hardware supports the ability to suppress interrupts using the CLI instruction; we use this capability to disable interrupts in kernel-mode.

Interrupts are the mechanism that enable user-level code to transfer control back to the kernel. For program-error exceptions, the kernel will typically repair the fault and resume the original process (possibly with the help of an external interrupt handler). For unrecoverable faults, the kernel may choose a new user-level program to run after suspending or killing the faulting process. User-level code may also request operations from the kernel using interrupts: system calls can be implemented using software interrupts whose handlers invoke the appropriate routines within the kernel. Operating system developers can ensure that control always returns to the kernel by configuring a recurring timer interrupt that transfers control to the kernel at regular intervals. If timer interrupts are configured and handled appropriately, then a user-process cannot take over the machine, even if it never faults or issues a system call—the kernel will get an opportunity to execute and make scheduling decisions when the next timer interrupt occurs.

2.1.4 Input/Output

Many PC devices (including timers, serial ports, the keyboard, mouse, and network interfaces) are controlled by reading and writing data and control information to specified *ports* through special *in* and *out* instructions. The CPU communicates with these devices via one of sixteen interrupt lines available on the hardware. These interrupt lines deliver *interrupt requests* (IRQs) to the processor via a multiplexer called the *programmable interrupt controller* (PIC). On the PC architecture, the multiplexing behavior is divided between two circuits: a *master* PIC handles IRQ 0 through IRQ 7 and a *slave* PIC handles IRQ 8 through IRQ 15. The kernel enables, disables, or acknowledges interrupts on a particular line by writing to the registers of the programmable interrupt controllers, which are connected to specific ports. Only the master PIC communicates with the processor, so one of its interrupt lines must be used for the slave to communicate interrupt requests to the master. Many of the remaining lines have a standardized role on the PC architecture. For example, the timer interrupt is usually associated with IRQ 0.

2.2 VIRTUAL-MEMORY MANAGEMENT

Virtual memory enables different user-level processes to have independent views of memory. Each view of memory contains a different mapping from *virtual addresses*—the addresses that a process reads and writes—to *physical addresses*—addresses into the physical memory installed on the underlying machine. We refer to the collection of mappings as an *address-space* and the set of mapped virtual addresses as a *virtual address-space.* A process can only access memory locations that are mapped in its virtual address-space. Any attempt to access an address that is not mapped will cause a program exception called a *page fault.*

In this section, we provide an overview of the virtual-memory management facilities available on IA32, including an in-depth look at the data structures for translating virtual-addresses to physical-addresses. Section 2.2.1 introduces the virtual-to-physical address translation structures; Section 2.2.2 discusses the protection mechanisms available for controlling access to mappings; Section 2.2.3 presents detailed representation information about the data structures; and Section 2.2.4 provides an example of updating the translation structures.

2.2.1 Address Translation Structures

Mappings are added to the virtual address space at the granularity of a *memory page*, the size of which varies depending on the architecture of the machine. The IA32 supports two sizes of physical page: four kilobytes and four megabytes. The mechanism for accessing physical memory via a virtual address also varies depending on the architecture. On the IA32, the hardware supports virtual address spaces by performing a translation between virtual and physical addresses in hardware by reading translation tables that are themselves stored in physical memory pages. Concretely, these tables are organized into a two-level data-structure. The first level of the structure is a single table called the *page-directory.* Entries in the page-directory point to second level tables called *page-tables*. Page-tables store the physical addresses of 4 KB physical pages (the target of a mapping). Each page-directory and page-table is itself stored in a single 4 KB physical page. An address-space is represented by a pointer to the beginning of a page-directory. The hardware has a special register, called *CR3*, that stores the currently active address space pointer. The value of this register determines the memory map that the hardware will use to perform address translation. Figure 2.3 shows the two-level page-table structure of the IA32 translation tables.

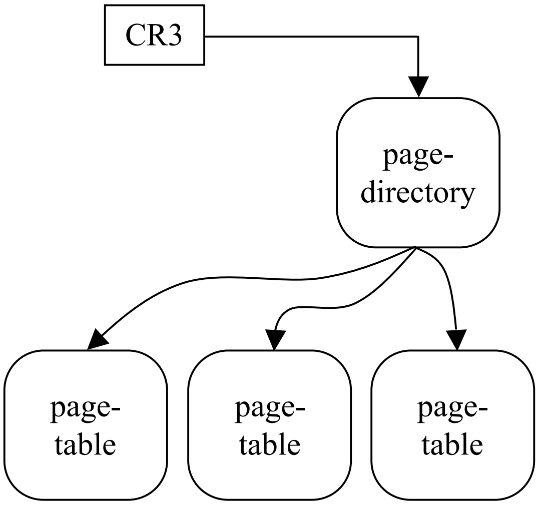


Figure 2.3: On the IA32, virtual-to-physical address translations are performed using a two-level data-structure that resides in memory. Each virtual address space has a single first level table called a page directory that points to potentially many second level tables called page tables. We identify address spaces using the address of the page directory in memory (a physical address), which is stored in the CR3 register. In the figure, arrows indicate a pointer from one structure to another.

The actual translation from virtual to physical addresses is performed by using the virtual address to access information in the two-level structure. Conceptually, a virtual address consists of three distinct components: the page-directory index, the page-table index, and the offset, as shown in Figure 2.4. The page-directory index identifies the appropriate entry in the page-directory for this virtual address. The page-directory entry tells us which page-table we should use for the next step in the translation. The page-table index allows us to look up the *page-table entry* in this particular page-table. The page-table entry stores the address of the physical page that the virtual address in question maps to, if any. Combining the offset (which can be thought of as an index into a memory page) and the address of the physical page gives the final result: the physical address mapped to by a particular virtual address. Figure 2.4 illustrates the translation process.

It is not always necessary to include the page-table step in a virtual-to-physical address translation. When a large enough set of contiguous virtual addresses are mapped to a contiguous set of physical addresses, we can store the physical address in the page-directory entry instead of the page-table entry. Each page-directory addresses 4 MB of memory, so such mappings have a size that is a multiple of 4 MB and must be aligned to a 4 MB boundary as well. Mappings made with page-directory entries are said to be implemented with 4 MB pages, which are also called *superpages*. Mappings made with page-tables are said to be implemented with 4 KB pages, because this is the amount of memory addressed by a page-table entry. Using superpages reduces the number of translation lookaside buffer (TLB) entries occupied by a mapping. The TLB provides a cache of virtual-to-physical address translations, which avoids the need for a page-directory look-up on every memory access. Using fewer TLB slots on a large mapping reduces the frequency with which entries must be invalidated to make room for new cached translations. Superpages also save memory because they avoid the need for allocating page-table storage. Figure 2.5 illustrates a virtual-to-physical address translation using a superpage rather than a page-table.

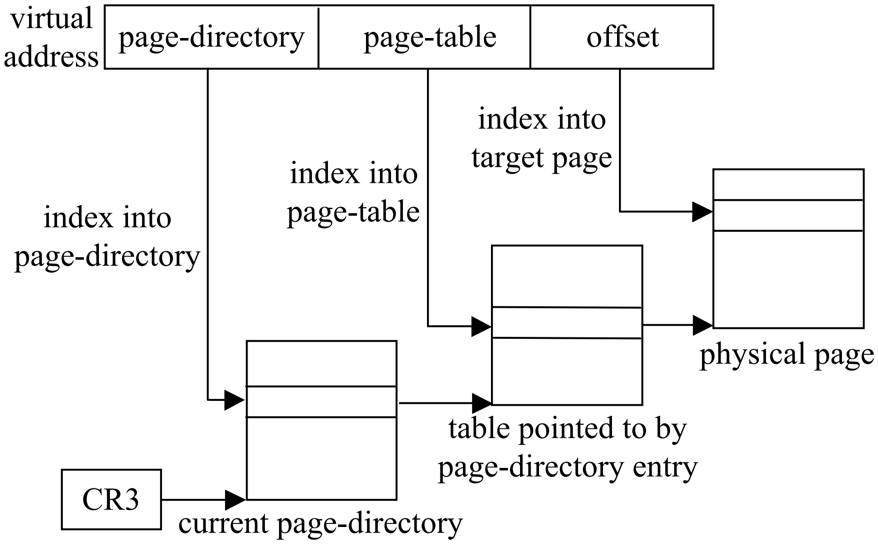


Figure 2.4: Virtual to physical address translation on the IA32 using 4 KB pages. The most significant bits of the virtual address are used as an index into the current page-directory (found in the address specified by the CR3 register) to locate an appropriate page-table. The middle bits are then used as an index into that page-table to find the physical page that the virtual address is mapped to. The complete physical address is then computed by using the low bits of the virtual address as an index into the physical page.

2.2.2 Protection

The IA32 architecture supports process separation and access control through per-page memory protection mechanisms. These protection mechanisms allow access to be restricted based on the following conditions:

- **Privilege level:** Privilege level protection controls the privilege required to access a mapping; the two possible values are user and supervisor. Proper

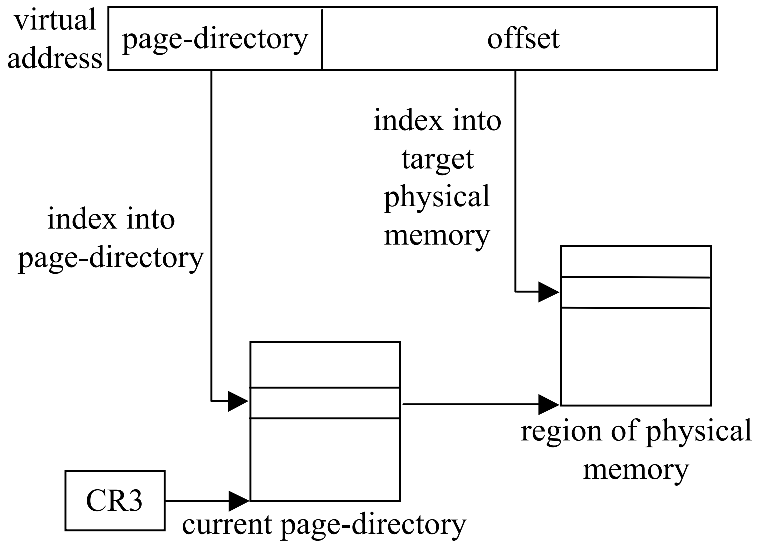

virtual address
page-directory
offset
index into target physical memory
index into page-directory
region of physical memory
CR3
current page-directory

Figure 2.5: Virtual to physical address translation on the IA32 using 4 MB pages. As with 4 KB mappings, the high bits of the virtual address are used as an index into the current page-directory (specified by the CR3 register). However, in this case, the page-directory entry stores a physical address, rather than a page-table location. The remainder of the virtual address (what was the page-table index and the offset) is treated as the offset from that physical address. The result of the translation is value of the physical address stored in the page-directory entry added to the offset.

configuration of privilege level permissions is essential for enforcing separation between the kernel and user programs. In our abstraction layer, we map the kernel code and data in every address-space with supervisor permissions so that the kernel can run in any space without any risk that a user program will access or overwrite the kernel data. Any attempt by a user program to access memory that is mapped with supervisor permissions causes a program error called a *protection fault*.

- **Access Type:** Protection based on access type allows us to set particular areas of memory as read-only. Writing to a read-only mapping fails and triggers a page fault.

Memory protection based on privilege and access type both play an important role in the abstraction layer implementation.

2.2.3 Translation Table Formats

Figure 2.6 shows a detailed look at the format of each entry in a page-directory. The top twenty bits contain a pointer to a 4 KB or 4 MB page; this enforces the minimum alignment of the pointer and leaves the remaining twelve bits for extra configuration information. Figure 2.6 also shows the format for page-table entries. The formats of the two entry types are identical except that a page-table entry will always contain a page base address pointing to a 4 KB page. A page-directory entry may point to a 4 MB page or a 4 KB page; the expected format of the page base address is indicated by Bit 7 of the page-directory entry format.

These formats also illustrate the protection mechanisms for controlling access to installed memory mappings. Bit 0 indicates whether or not a particular page is *present*; the memory mapped by the entry can only be accessed when this bit is set. Otherwise this memory cannot be read or written by the user or the kernel, and any attempt to do so will cause a page fault. Bit 1 of each entry specifies the read/write privilege for a page or set of pages. When this bit is set to zero the corresponding memory is read-only, but when the bit is set, the memory can be read and written. Bit 2 of each entry is called the user/supervisor bit. This bit controls the privilege level protection for the mapping. If the bit is set, then the memory mapped by the entry is accessible in both user- and kernel-mode; otherwise the memory is accessible only in kernel-mode.

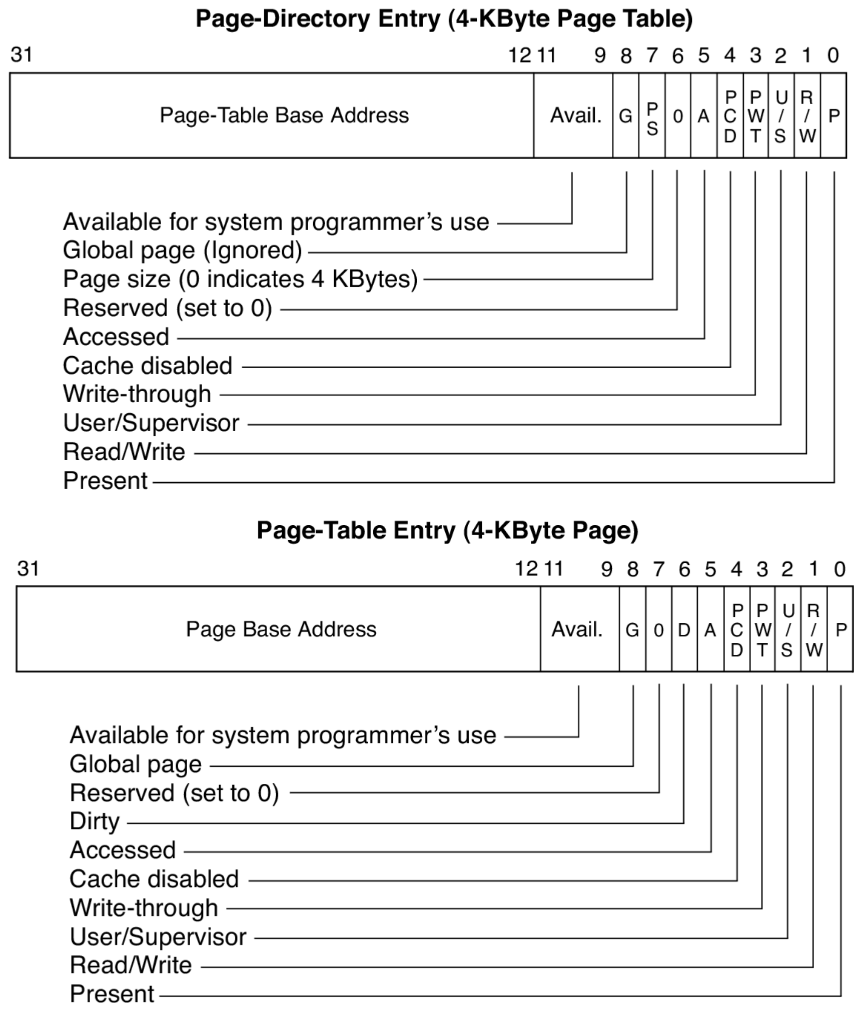


Figure 2.6: Format for page-directory entries and page-table entries (when using 4 KB pages). The top twenty bits contain the aligned address of the memory being mapped. The low bits contain configuration information, including permissions and mode restrictions. This figure is a reproduction of Figure 3-14 from the Intel Architecture Software Developer's Manual [51].

In addition to dictating how a memory mapping may be accessed, the page-directory and page-table entries also provide feedback about how the memory described by a mapping has been accessed. This information is stored in the accessed and dirty bits (Bits 5 and 6 of the entries, respectively). The accessed bit is set by the MMU when the page (or group of pages) has been read. The dirty bit is set when the page has been written. The dirty bit is not used for page-directory entries that point to page-tables, so it always corresponds to a single page of data.

2.2.4 Example: Adding a Virtual-To-Physical Mapping

As an example of address-space management in a typical IA32-based operating system, consider the steps required to add a new virtual-to-physical mapping. To make additional physical memory visible in an address-space, the operating system must set up the relevant translation table entries to point to this physical memory. The entries to modify are determined based on the location in the virtual address-space where the physical memory will be mapped. The mapping procedure will be different depending on the size of the memory area being mapped and whether or not 4 MB superpages will be used. Assuming that we wish to add a mapping that will use a single page-table, the operating system must take the following steps:

1. Find the appropriate page-directory entry. This entry can be found by using the page-directory index bits from the virtual address where we would like to add the mapping. Because we assume that the mapping will fit in a single page-table, we know that we will only need to modify the memory pointed to by a single page-directory entry.

2. Determine whether or not the page-directory entry already points to a page-table.

3. If necessary, create a new page-table out of a free page of memory and update the page-directory entry with the address of the new table.

4. Modify the appropriate entries of the page-table. The entries to modify are determined by the page-table indexes of the virtual addresses for which we

are adding mappings. We will modify one page-table entry per 4 KB page of memory being mapped.

The procedure for mappings that are implemented with superpages is similar. The key differences are that we will never need to allocate a page-table (Step 3) and we will only modify page-directory entries and not page-table entries (Step 4).

Chapter 3

BACKGROUND: FUNDAMENTALS OF HASKELL PROGRAMMING

Haskell is a strongly typed, purely functional programming language that originated in the late 1980s as an attempt to unify the many lazy, purely functional languages that existed at that time [47]. Laziness and a pure semantics are key defining features of Haskell. The term laziness indicates a non-strict, call-by-need evaluation order, where the value of an expression will not be computed until that value is actually needed by another part of the computation. Expressions with values that are not required are never evaluated. This characteristic of Haskell frees the programmer from having to worry about execution order and wasted computation, but can have a surprising impact on performance and space usage when a seemingly simple expression triggers the evaluation of many delayed computations. Laziness fits naturally with a pure semantics—where functions do not perform any side-effects—because the programmer will not be able to observe evaluation order through the ordering of effects. The Haskell language provides a special mechanism for tracking the use of effects in the type system so that side-effecting programs can remain pure; we will cover this topic in Section 3.4.

In addition to laziness and purity, Haskell supports many of the standard functional language mechanisms for abstraction and reuse, including a module system, higher-order functions, polymorphism, and abstract datatypes. Memory-safety is guaranteed with the help of dynamic memory management using garbage collection. The language is defined by the Haskell 98 Report [78], but there are numerous extensions in common use, including the foreign function interface [12] (which is essential for the abstraction layer implementation). Extensions supported by the

Glasgow Haskell Compiler (GHC) [31] have become de facto standards and are described in the GHC User's Guide [91].

To illustrate the basic language features available in Haskell, consider the definition of factorial:

```
factorial :: Int -> Int
factorial n = if n == 0 then 1 else n * factorial (n - 1)
```

This function takes a single integer argument and returns an integer result, as specified by the type declaration on the first line. The code for `factorial` is a single equation that names the argument `n` on the left-hand side of the definition. Haskell supports pattern-patching in the left-hand side of definitions, but name patterns like `n` match any value. The right-hand side defines the behavior of the function. In this case, we multiply the value of `n` by the result of a recursive call to `factorial` on `n-1` unless `n` has already reached zero.

To illustrate the pattern matching facilities available in Haskell functions, we can rewrite the definition of `factorial` as two equations that encode the stopping condition `n == 0` as a pattern rather than a conditional.

```
factorial :: Int -> Int
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

This definition implements exactly the same behavior as before. We now have two equations: one for the base case and one for the recursive case. In the first equation, we use the pattern `0` as an implicit test for whether or not the argument to `factorial` equals zero. If the argument does equal zero, then the pattern matches and the result that is returned from the function will be 1, the right-hand side of the equation. If the argument is not zero, the pattern will not match and the second equation will be used. The pattern `n` matches any value, so the computation proceeds by executing the right-hand side of the second equation.

The factorial example illustrates some basic elements of functional programming in Haskell. In the remainder of this chapter, we will introduce other essential concepts of Haskell that will appear in the code samples throughout the rest of this dissertation. For a more comprehensive overview of Haskell, the reader is encouraged to seek out one of the textbooks on the subject [77, 49, 46]. Section 3.1 introduces datatypes and more sophisticated pattern matching constructs. Section 3.2 demonstrates the use of parameterized datatypes such as lists. Section 3.3 describes a mechanism for defining predicates on types using type classes and qualified types. Finally, Section 3.4 discusses the relationship between side-effects and purity in Haskell. The concepts that we describe here are standard; readers already familiar with Haskell may prefer to skip ahead to the next chapter.

3.1 DATATYPES

Algebraic datatypes are a fundamental abstraction mechanism in Haskell. The language defines the most commonly used structures like pairs and lists (see Section 3.2 for an in-depth look at lists) as well as a powerful mechanism for declaring user-defined types. As a simple example, consider the definition of a basic binary tree in Haskell.

```
data Tree = Branch Int Tree Tree
          | Leaf
```

New datatypes are introduced by the keyword data. The left-hand side of the definition provides a name for the new type; this is called the *type constructor*. The right-hand side describes the values of the type with one or more *value constructors* (referred to throughout the dissertation simply as constructors). Each of these constructors takes zero or more arguments as necessary to produce a value of the type. In our example, there are two constructors for values of type Tree: Branch and Leaf. A Branch contains an integer describing the value stored at that node, a left subtree, and a right subtree, while a `Leaf` caries no data. We construct values of type `Tree` by applying the value constructors to arguments of the appropriate type. For example,

```
Leaf
Branch 5 Leaf Leaf
Branch 5 (Branch 4 Leaf Leaf) (Branch 6 Leaf Leaf)
```

all represent values of type `Tree`.

Functions over the `Tree` datatype can be defined using pattern matching to distinguish the different kinds of values. The following function computes the number of data values in a given binary tree.

```
size :: Tree -> Int
size (Branch v left right) = 1 + size left + size right
size Leaf = 0
```

As we saw in the factorial example, we can write functions using multiple equations where the cases are distinguished by pattern matching. The first equation checks whether or not the `Tree` argument was produced with a `Branch` constructor. If so, we call the function recursively on both of the branch's subtrees and increment the result. The second equation covers the case where the `Tree` is a `Leaf`, though pattern matching on `Leaf` is not strictly necessary because there are only two constructors. Leaf nodes do not carry data values, so we return zero in this equation.

In fact, we can rewrite `size` using a wildcard pattern to avoid the unnecessary binding of `v` and to avoid testing whether the value in the second equation is a `Leaf` (we know by a process of elimination that it must be).

```
size :: Tree -> Int
size (Branch _ left right) = 1 + size left + size right
size _ = 0
```

The definition is exactly the same as the one we examined previously, but with `v` and `Leaf` replaced by the wildcard pattern, `_`.

An unsatisfying aspect of algebraic datatype definitions, like our binary tree example, is that we have to remember the order of the constructor arguments when pattern matching or constructing new values. The Haskell type-checker helps when the arguments have different types (by signaling an error if we make a mistake), but types alone do not always capture the semantics of the argument. For example, the `Branch` constructor for the binary tree datatype takes two trees as arguments: one for the left subtree and one for the right. If we accidentally confuse these two parameters, there is nothing in the types that will help us. Particularly in larger datatypes that are used in complicated programs, maintaining this mental state can be unwieldy and error prone.

As an alternative to the previous datatype definition, we can use Haskell's record syntax to associate names with each of the arguments of a given value constructor. If we recast our binary tree definition using record syntax, for example, then we obtain:

```
data Tree = Branch {
              value :: Int,
              left  :: Tree,
              right :: Tree
            }
          | Leaf
```

Given this definition, we can extract the left subtree of a branch `b` by writing `left b`. Records allow us to create new values concisely as well, providing a mechanism to define new values in terms of existing ones. For example, to replace the integer stored in a binary tree branch record, `b`, with the number `40`, we would write `b{ value = 40 }`.

The definition of `Tree` as a record still permits us to define `size` in exactly the same manner that we did previously, but we can also rewrite the definition to use the accessor names instead of using pattern matching.

```
size :: Tree -> Int
size Leaf = 0
size b = 1 + size (left b) + size (right b)
```

In this case, the named pattern `b` matches an entire branch, rather than just one of its arguments. We reorder the equations so that the `Leaf` case appears first; in this way we guarantee that every argument to the second equation is actually a `Branch`. Applying the function `left` or `right` to a `Leaf` value would cause a run-time error.

The `data` keyword is the primary mechanism for introducing new types in Haskell, but the language does provide two additional mechanisms that are useful in some circumstances. A datatype defined using the `newtype` keyword takes exactly the same form as one defined using `data`; the difference is that a `newtype` must have exactly one constructor with exactly one argument. Programmers use `newtype` to create a wrapper for an existing type, be it a built-in type or a user-defined type, with a new constructor that the programmer controls. For example, we can make our binary tree example more abstract by hiding the fact that values stored in the tree are integers. We create a type synonym for `Int` called `TreeValue` that we will use to define our tree.

```
newtype TreeValue = TreeValue Int
```

Using the same name for the type constructor and the value constructor, as shown in the definition of `TreeValue`, is common practice in datatype definitions with only one constructor. This is possible because type and value constructors have separate name-spaces. The datatype definition for `Tree` can now be written to use `TreeValue` in place of `Int`.

```
data Tree = Branch TreeValue Tree Tree
          | Leaf
```

Declarations of this sort are very useful when designing an abstraction layer or library, because they allow the programmer to use standard types without exposing the library's internal representations to its clients.

Types constructed with the `newtype` keyword are not type synonyms in the truest sense of the word, because the two types involved in the definition are not interchangeable. Haskell does provide a mechanism for defining actual type synonyms using the `type` keyword. With `type`, the connection between the two types is always visible—they are the same type. A `type` definition looks similar to the other type declarations, but the right-hand side may only contain an existing type with no constructors.

```
type TreeValue = Int
```

With this version of `TreeValue`, we can use the name `TreeValue` in place of `Int` as documentation in a type signature, but the values of the two types are indistinguishable. This mechanism avoids the overhead of creating an extra constructor, but it does not provide the strong guarantees of a true abstraction.

3.2 PARAMETERIZED DATATYPES

In addition to the simple kinds of datatypes that we saw in Section 3.1, Haskell supports parameterized datatypes for capturing common structural patterns that can be applied to any type (analogous to void pointers in C, templates in C++, or generics in Java). A parameterized datatype definition takes the same form as any other datatype declaration, but the type constructor takes one or more type variable arguments that may be instantiated to any type. As an example, the tree datatype from Section 3.1 can be defined to store values of any type, rather than just integers.

```
data PTree a = PBranch a (PTree a) (PTree a)
             | PLeaf
```

The type of the value PLeaf is PTree a; there is nothing about this value that constrains the type of a. In contrast, the type of PBranch 5 PLeaf PLeaf is PTree Int; using the integer 5 in the value slot of the PTree specializes the type variable a to Int. As another example, we could create the PTree value PBranch "george" PLeaf PLeaf; this value has type PTree String.

Strings themselves illustrate the use of a very common parameterized datatype in Haskell: lists. Strings are simply a type synonym for a list of characters.

```
  type String = [Char]
```

Square brackets are the Haskell syntax for lists. A list that contains any kind of value would have a type of the form [a] (where a is a type variable) while a list that has been instantiated to a specific type would have a type of the form [Type]. Type variables always begin with a lower-case letter while type names always begin with an upper case letter. We write list values using the square bracket syntax as well:

```
[]                    -- the empty list :: [a]
['z']                 -- a singleton character list :: [Char]
[11, 40, 19]          -- a three element integer list :: [Int]
[[1],[2]]             -- a list of integer lists :: [[Int]]
```

The square brackets and commas are syntactic sugar for the actual list constructors. We have already seen the empty list constructor, []. The other value constructor for lists is called *cons*; it adds a new element to the front of an existing list and is written using an infix colon operator. The definition of cons involves a recursive use of [].

```
  data [a]  = [] | a : [a]
```

Rewriting our examples using the constructors explicitly, instead of the syntactic sugar, gives us:

```
[]
'z' : []
11 : (40 : (19 : []))
(1 : []) : ((2 : []) : [])
```

Lists are a common datatype in Haskell that appear frequently in the code for our abstraction layer in later chapters. Another common parameterized type is pairs (also called tuples). Although we will not cover this datatype deeply here, it is important to note that Haskell supports n-element tuples, for example, `(a,b)` and `(a, b, c, d)`. This syntax works at both the type and value level. There is no single element pair, but there is a zero element pair, written `()` and pronounced “unit”.

Defining functions on parameterized datatypes works in much the same as any other declaration. Some functions are specific to a particular instantiation of a parameterized type, but often there are interesting polymorphic functions as well. The Haskell libraries include a multitude of functions on lists, many of which are completely generic. An example that we use frequently in the abstraction layer source code is the `map` function for applying a given function to every element of a list.

```
  map :: (a -> b) -> [a] -> [b]
  map _ []     = []
  map f (a:as) = f a : (map f as)
```

`map` takes two arguments: a higher-order function that turns values of type `a` into values of type `b` and a list of values of type `a`. The definition of `map` will apply the function to each element of the argument list, producing a list with elements of type `b`. Functions are first class values that may be passed as arguments to other functions by name; they are distinguished in the type by the parentheses around the type of the higher-order argument.

Container types like list and pair are an obvious use of polymorphism, but there are many other programming patterns that we can capture using polymorphism. A pattern that we see frequently in programming is the use of special values to indicate failure. For example, the `malloc` function in C returns a pointer to a newly allocated area of memory—unless the request fails, in which case it returns a null pointer. The potential for error is not documented by the type, so you have to rely on intuition and the library documentation to realize that the function might fail. Worse yet, the null pointer is easily confused with a normal pointer to memory (as evidenced by how frequently null pointer dereferences occur in practice). In Haskell, we can encapsulate this pattern of an optional value (`malloc` either returns a pointer or nothing) with a parameterized type called `Maybe`.

```
data Maybe a = Just a
             | Nothing
```

Values produced with the `Just` constructor are equivalent to the values expressible in the type `a`, but with an extra wrapper to lift them into the `Maybe` type. `Nothing` is a special value that cannot be confused with a normal value of type `a`; it can be interpreted as the absence of a valid value. If we were to define `malloc` in Haskell, we might declare the result type of the function to be `Maybe (Ptr a)` (where `Ptr` is another parameterized type), returning `Nothing` if the memory cannot be allocated.

3.3 TYPE CLASSES AND QUALIFIED TYPES

Polymorphic functions on arbitrary types are extremely useful, but often in practice, we want to define polymorphic functions over a class of types that are constrained in some way. Consider the function for testing if a particular value is a member of a list. We might be tempted to write the type for this function as

```
elem :: a -> [a] -> Bool
```

because the function can operate on lists of any type. The problem is that `elem` as declared must perform the same way on all types; how can `elem` determine if the value is in the list when the equality comparison that is necessary to make that determination might be different for different types?

Implementing `elem` in a polymorphic way requires the ability to define overloaded operators, like an equality test, that may have different implementations on different types. This overloading is called *ad hoc polymorphism*, in contrast to the *parametric polymorphism* that we examined in the previous section. Haskell provides a mechanism to support ad hoc polymorphism called type classes; each type class is associated with a set of overloadable functions [96, 55, 56]. There are many standard classes in Haskell to support common operations such as comparison, numeric operations like addition, and displaying the values of a type. In particular, the built-in class for equality comparisons that we need in `elem` is called `Eq`, and is defined as follows:

```
class  Eq a  where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
```

The `Eq` class defines two methods: `==` for testing equality and `/=` for testing inequality. We define implementations of these functions for a particular type by declaring it as an *instance* of `Eq`. The instance declaration for the Boolean type, `Bool`, provides an example of the syntax.

```
instance Eq Bool where
  (==) True  True  = True
  (==) False False = True
  (==) _     _     = False
```

Type class declarations may provide default definitions for overloaded functions in addition to declaring their types. Though our example definition of `Eq` only shows the type signatures for the equality comparisons, the built-in definition of `Eq` does provide a default definition of /= in terms of ==. Thus, the instance declaration for Bool only needs to provide a definition of ==.

Standard Haskell libraries define type class instances for many standard types, but when we define our own types we might also wish to add instances for standard type classes. We can declare an instance of Eq for our first version of binary trees using the same pattern as for Bool.

```
instance Eq Tree where
  (==) Leaf Leaf = True
  (==) (Branch v1 l1 r1) (Branch v2 l2 r2)
    = v1 == v2 && l1 == l2 && r1 == r2
  (==) _ _ = False
```

The first equation states that two Leaf values are equal. The second equation states that two Branch values are equal if all of the arguments to the constructor are equal; here we rely on the instance of Eq on integers and on recursive calls to the == function on trees that we are currently defining. The final equation is a catch all for the two cases where the constructors are not the same; these values will never be considered equal.

Now that we have an overloaded equality test, we are able to define the elem function in a polymorphic way.

```
elem :: Eq a => a -> [a] -> Bool
elem _ [] = False
elem x (y:ys) = x == y || elem x ys
```

Note that the type signature includes some extra information. The clause Eq a is a constraint on the type variable a indicating that the type a must be an instance of the type class Eq. This reflects the use of the == method from the Eq class to compare values of type a. A type containing such a constraint is called a qualified type and it restricts the polymorphism of the elem function to the set of types on which Eq is defined [55, 54]. A function may rely on multiple classes in its definition; these constraints are all added to the left of the =>.

```
example :: (C a, D b, E a) => a -> b -> c
```

In this example, there are two constraints on the type variable `a`—it should be an instance of both `C` and `E`—and just one—`D`—on the type `b`.

As mentioned previously, there are several standard classes defined by the Haskell libraries. Some examples are:

- **Ord:** Defines operations on totally ordered types. This class provides the usual comparison operators (`<`, `<=`, `>`, and `>=`) as well as minimum and maximum operators.

- **Show:** Supports the conversion of values of the instance type into strings (typically for printing to the screen). The most frequently used function from the `Show` class is `show :: a -> String`.

- **Storable:** Allows values of the instance type to be transferred to and from memory. This class is part of the foreign function interface, and includes methods for directly accessing memory.

- **Num:** Supplies basic operations on numeric types like addition, subtraction, and multiplication.

- **Bounded:** Provides operations that name the upper and lower bounds of a type, `maxBound` and `minBound`.

For the full list, see the Haskell library documentation [32]. In addition to these predefined classes, Haskell also allows the definition of new classes using the same syntax that we saw in the earlier examples.

3.4 MONADS

Purity is one of the primary motivations for implementing operating systems in Haskell. With a pure semantics, every function is a function in the mathematical sense: the same set of inputs always produce the same output. Programs are easier to reason about and debug because there is no hidden data flow. To develop an understanding of purity in practice, consider the definition of a simple mathematical function like `align`. The `align` function takes an unsigned word argument and zeros out the low `n` bits, where `n` is an integer parameter to the function.

```
align :: Word32 -> Int -> Word32
align w n = (w ‘shiftR‘ n) ‘shiftL‘ n
```

One technique for aligning a word is to shift the word right by the number of bits to be cleared, then left, relying on the fact that the bits shifted in from the right will be zeros. We define `align` with a straightforward application of the right- and left-shift functions from the Haskell library for bit-level manipulation (`Data.Bits`). `Word32` is the name for the Haskell type of unsigned 32-bit words and is equivalent to the C type `unsigned int`.

Comparing the Haskell definition to an equivalent definition in C illustrates a superficial similarity.

```
unsigned int align(unsigned int w, int n) {
  return (w >> n) << n;
}
```

The substantial part of both functions can be written in a single line using built-in shift operators. Both contain a type signature indicating that the function takes an unsigned word and an integer and returns an unsigned integer.

If we write the type for the two functions in a common syntax, then it would appear exactly the same in both cases:

```
align :: Word32 -> Int -> Word32
```

The essential difference between the two definitions is that the C type of this form and the Haskell type of this form do not mean the same thing. In C, a function with this type may perform arbitrary side-effects, such as direct memory accesses, global variable manipulations, file I/O, and printing to the screen. The language allows direct and unchecked access to any of the facilities available on the machine from any point in any program. None of this information is reflected in the type. In contrast, functions in Haskell do not perform any visible side-effects. A pure Haskell function can never dereference a null pointer because we know from its type that it does not access memory directly.

Purity in Haskell helps to identify bugs and prevents certain classes of error from ever occurring. However, to deal with systems applications and many other real-world tasks, our programs must somehow be allowed to perform side-effects. In Haskell, this is accomplished using a special mechanism called a monad [94]. In general, a monad consists of two parts: a parameterized datatype that names the monad and a set of operations that define the side-effects supported by the monad. For example, the `IO` monad captures interactions between Haskell and the outside world, and defines operations for tasks such as printing to the screen or reading from a file. A function that uses an operation defined in the `IO` monad will have the monadic type `IO a` where `a` is a type variable representing the result of the computation (as with the other parameterized types we have examined).

`IO` is the most pervasive effect in Haskell because so many programs rely on interactions with the outside world in some capacity, whether it is a user interacting with the program or a device that the program controls. In fact, every Haskell program runs in the `IO` monad at some level, even if the program contains mostly pure computation, because the entry point to a Haskell program is fixed to be a function called `main` of type `IO ()` (indicating that the function runs in the `IO` monad and returns no data in the result). Still, there are many other useful monads that are common in Haskell programs. The following examples of monads defined by the standard Haskell libraries illustrate the kinds of side-effects that we can capture using monads. Unlike the `IO` monad, the side-effects represented by these monads are not persistent; they model side-effects in a pure way that does not affect the outside world.

- **State:** There are no global variables or stateful computations in pure Haskell. We can model state by passing around any stateful components of our computation as explicit parameters to every function, but this is clumsy and error prone in practice. The `State` monad allows us to add state to our Haskell programs without explicit parameters by encoding the state-passing style implicitly.

- **Read-Only State:** A function might depend on environment data, supplied by the calling context, that remains constant within the function but may be extended or modified for functions it calls. As with state, we can address this need by passing the environment as a parameter to every function or we can use the `Reader` monad to pass around the environment implicitly.

- **Failure:** We saw in Section 3.2 that the `Maybe` datatype allows us to augment any type with a special failure value called `Nothing`. In fact, the `Maybe` type is also a monad that can be used to handle errors or exceptions. Computations in the `Maybe` monad may produce the value `Nothing`, at which point the monad will skip the remainder of the computation and return the value `Nothing`.

- **Exceptions:** The Haskell libraries also provide a more traditional interface that allows the programmer to try, catch, and throw named exceptions. This functionality is defined by the `Error` monad.

These are just a few of the common effect patterns expressible as monads. See the literature for more details about these and other examples [95, 65, 79].

Now that we have introduced monads, we can demonstrate how the obligatory "Hello World" program is written in Haskell.

```
helloWorld :: IO ()
helloWorld = putStrLn "Hello World!"
```

The `helloWorld` function runs in the `IO` monad so that it may print a line to the standard output device using the Haskell library function `putStrLn` of type `String -> IO ()`. The function `putStrLn` is one of the many side-effecting operations supported by the `IO` monad.

Every monad contains some number of custom operations, like `putStrLn`, that define the unique behavior of that monad. In addition to these custom operations, there are two standard functions that must be defined for every monad. These functions are called return and bind. The bind function, typically written as `>>=`, allows us to sequence monadic expressions; execution order is important for monadic computations because the side-effects in one monadic operation can affect the outcome of later operations. The `return` function lifts pure values into the monad so that we can use the entire pure subset of Haskell within our monadic computations.

The type of the return function is simple: for each monad `m`, the associated `return` function takes a value of any type and lifts it into the monad.

```
return :: a -> m a
```

For example, the return function for the `IO` monad would be a polymorphic function with the following type:

```
return :: a -> IO a
```

The bind function connects the output of one monadic computation to the input of the next, thus sequencing the two computations together. The type for bind reflects this sequencing.

```
(>>=) :: m a -> (a -> m b) -> m b
```

The second argument to bind is a function; the definition of bind will apply the function to the value produced by the first computation.

As an example of how we might use bind in a real program, consider the function for reading a line that is part of the standard `IO` monad library in Haskell.

```
getLine  :: IO String
```

Executing the `getLine` function reads a line from the standard input device. We can combine this function with `putStrLn` using bind to define a program that reads a string supplied by the user and echoes that string back to the screen:

```
echoLine :: IO ()
echoLine = getLine >>= putStrLn
```

This function reads a line from standard in and prints the result using the `putStrLn` operation.

The ability to sequence computations together allows us to define interesting monadic functions, but connecting operations using explicit calls to bind is cumbersome when writing large programs. Haskell provides syntactic sugar known as *do-notation* to enable seamless sequencing of monadic operations. We can rewrite the echo example using do-notation as follows:

```
echoLine :: IO ()
echoLine = do line <- getLine
              putStrLn line
```

Here we have the opportunity to give our intermediate result a name using the syntax `name <- ...`; this `name` can be used to refer to that result at any later point in the computation. We will use do-notation when writing monadic code almost exclusively throughout the rest of the dissertation.

We can use do-notation in any monad that provides definitions for the standard functions return and bind. For example, the `Maybe` monad allows us to string together computations that might fail without explicitly checking the results of the intermediate steps. Once a failure is encountered, the remaining computation is abandoned and the result of the computation is `Nothing`. These failures can include explicit exceptions that occur when a step in the computation returns Nothing, or implicit failures, like a non-exhaustive pattern match in a do-notation pattern binding. If none of the steps fail, then the computation runs to completion and produces a result wrapped in a Just constructor. The standard definitions of the return and bind operations for the Maybe monad are as follows:

```
return :: a -> Maybe a
return x = Just x

(>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
(>>=) Nothing  _ = Nothing
(>>=) (Just x) f = f x
```

The definition of bind illustrates how the threading process works: sequencing an operation with a previous operation that failed produces Nothing. Just values are threaded through the computation without modifying the value of the underlying type.

As a simple example of using the Maybe monad, rather than just the type, let us consider a function on binary trees that might fail: projecting the left subtree of a branch. For a tree made with the Branch constructor, leftTree will return the left subtree, but for a Leaf node, there will be no subtree to return. We can write this function using the Maybe type to capture failure without making use of monadic notation:

```
leftTree :: Tree -> Maybe Tree
leftTree t = case t of
               Branch _ l _ -> Just l
               Leaf         -> Nothing
```

This definition checks the value t using a case expression instead of using multiple equations; the two forms are equivalent. Using do-notation, we can write this same function as:

```
leftTree :: Tree -> Maybe Tree
leftTree t = do (Branch _ l _) <- return t
                return l
```

We must use `return` twice: once to lift the pure tree value into the maybe monad and once to lift the result of the computation (the left subtree). If `t` is a `Leaf`, then the expression

```
(Branch _ l _) <- return t
```

will cause a pattern match failure, but this will be caught and the result of `leftTree` will be `Nothing`. In this case, the use of do-notation does not make the definition any clearer, but in larger examples where multiple functions with a `Maybe` type are strung together this syntax can avoid a lot of syntactic noise.

Haskell does not limit us to the standard set of effects; programmers can easily define new monads to capture their own effect patterns or side-effecting operations. To create a new monad, the programmer must supply implementations for return and bind that are appropriate for the new monad and define any non-standard operations that implement the effects of the new monad. New monads may also be created with monad building blocks called monad transformers. Most of the standard monads have an associated monad transformer definition that adds the functionality of that monad to another monad [69]. This allows us to combine the effects of standard monads to produce custom monads that capture the necessary effects for a particular program. For example, we can use monad transformers to construct a monad that supports both exceptions and global state.

3.5 MODULES

The Haskell module system enables programmers to precisely control the types and functions that are exported from one module to another. Type constructors can be exported without making the value constructors for that type visible, and specific value constructors for a type might be exported while others remain hidden. Our abstraction layer implementation relies on the precise control afforded by the module system to enforce a strong abstraction boundary between the abstraction layer and its clients.

A module definition gives a name to a Haskell module and optionally may declare the list of types and functions that the module exports. Every definition in the module will be exported when a specific export list is not supplied.

```
module M where ...  -- everything exported
module M (f, g) where ... -- only functions f and g are exported
```

Recall our original definition of a binary tree, with a type constructor called `Tree` and two value constructors called `Branch` and `Leaf`. If we wish to make the entire datatype definition visible through a module, we use the following notation in the export list:

```
module M (Tree(..)) where ...
```

Clients of this module will be able to construct arbitrary values of the `Tree` type. Alternatively, we can export just the type constructor by omitting the `(..)`.

```
module M (Tree) where ...
```

Now, other modules can refer to the type `Tree` but cannot make any values of that type. `M` can guarantee properties of `Tree` values, for example, that every tree is balanced, because it controls the production of every value. A middle ground, which is less useful in this particular example, is to export just some of the value constructors for a type. For example,

```
module M (Tree, Leaf) where ...
```

allows modules other than `M` to construct `Leaf` values but not branches.

We will not show many module definitions in the code samples throughout the dissertation, but the abstraction mechanisms provided by the module system are essential for memory-safety enforcement in our implementation. The ability to hide the functions and datatype representations used within the implementation is crucial. Hiding value constructors for visible types allows us to use normal Haskell values as capabilities when requesting restricted abstraction layer operations, because the key safety properties for the operation are already guaranteed at value construction time.

Chapter 4

A MEMORY-SAFE ABSTRACTION LAYER FOR OPERATING SYSTEM CONSTRUCTION IN HASKELL

A major focus of this dissertation is the definition of a memory-safe abstraction layer that is sufficiently expressive to support the implementation of real operating systems. To demonstrate the feasibility of the design, and to show that the abstraction layer can be integrated with a purely functional language, we will also provide an implementation of the abstraction layer as a Haskell library (see Chapter 6). We call this Haskell library *the H interface*, and will use the term H as a shorthand for both the design of the interface (indicating the types and functions that make up our abstraction layer) and its implementation. The H interface is a direct descendant of the abstraction layer of the same name from the House operating system [39], although there are many differences between the original and current versions of the API and implementation.

Figure 4.1 illustrates the relationship between the Haskell run-time system, the H interface, and a Haskell operating system implementation. The H implementation runs on bare metal with support from the GHC run-time system; we will cover this implementation in more detail in Section 6.3. A Haskell operating system may access the types and operations of the H interface design, but not the details of the implementation. We refer to the operating systems that run on top of H as client kernels because they are clients of the H interface. Client kernels also have access to the standard facilities of the Haskell run-time system, but we can only guarantee memory-safety for clients that avoid calling the potentially unsafe primitives of the Foreign Function Interface [12]. Applications run on top of a client kernel by using

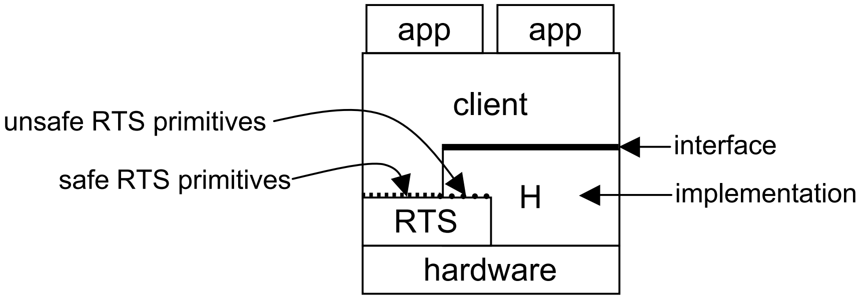


Figure 4.1: The basic architecture of H. H runs on bare metal, using support from the GHC run-time system for the portions that are written in Haskell. Client kernels—operating systems written using the primitives of the H interface design—run with the support of the H implementation and the GHC run-time system (because they are written in Haskell). Client kernels are allowed to access the safe facilities of the run-time system, but only the H implementation my use the potentially unsafe primitives. Applications run on top of the client, making use of the operating system API and, indirectly, H's facilities for executing arbitrary user binaries.

the operations for running user programs that are provided by the H interface.

In this chapter, we will introduce the operations of the H interface. This presentation will include a discussion of every type and function that is accessible to clients through the public API. For the most part, details of the private implementation are left until Chapter 6. Details of the implementation that are directly visible through the API—for example, the conditions necessary for a function to succeed—or that affect the high-level memory-safety analysis of the design will be included in this chapter.

The memory management facilities of the abstraction layer are the distinguishing feature of the H interface and are a core contribution of this dissertation. The choices made in the design of the memory management primitives have a significant impact on our ability to demonstrate memory-safety for our system and on our potential ability to implement a variety of operating systems on top of the abstraction layer. This chapter introduces the essential abstraction layer operations for managing physical and virtual memory and outlines the concepts that will form the basis of our memory safety analysis (presented in Chapter 5). Though our focus is on memory management, we present every aspect of the abstraction layer design for completeness. The H interface supports the following essential services for constructing operating systems in Haskell:

- **Memory management:** H provides facilities for working with the virtual memory management data structures described in Chapter 2. Clients of the interface may modify the virtual address space seen by user-level programs by adding a mapping, modifying the permissions on a mapping, or removing an existing mapping. Clients may also add mappings to the kernel virtual address space (the virtual-to-physical address translations that are accessible within the operating system). H supports virtual memory management by providing mechanisms for allocating and freeing virtual memory translation tables that live in memory (including both page-directories and page-tables). Correct and safe physical memory management is an essential component of the H design.

- **Protected execution of arbitrary user binaries:** H supports user-mode execution of programs written in any language. Our current implementation assumes that user-level object files are stored in the ELF binary format [13]. Execution of a running user-level program is automatically paused when a fault or interrupt occurs. Interrupt handlers within the H implementation save the state of that program so that it can be resumed again later. The operating system that is written on top of H may access the register state of suspended programs and use this information to respond appropriately to system call requests, faults, or other events.

- **Program module loading:** The H interface passes bootloader modules through to the operating system so that, for example, the OS can execute programs that are loaded by the bootloader. H provides facilities that support mapping memory pages within an executable module to user-level programs.

- **Low-level I/O operations:** H allows unrestricted access to the I/O ports on the underlying machine (IA32 in our implementation), following the original design and implementation [39]. Allowing unrestricted access prevents client kernels from enforcing resource management policies on I/O devices. In this work we focus on safety with respect to virtual memory management, but control over I/O devices is an interesting topic for future work.

- **Debugging:** H provides basic screen printing utilities for debugging during the development process.

Many of the services are very similar to those that were available in earlier versions of the H interface [39], particularly the aspects of the design that relate to user program execution, I/O, and debugging.

The remainder of this chapter describes the design of the H interface in detail. Section 4.1 describes the fundamental concepts that shape our memory-safety argument for the interface primitives. Section 4.2 provides an example that illustrates the combination of static and dynamic checking that we employ to guarantee safety in our implementation. Section 4.3 presents the basic data structures that we use to describe memory and executable modules. Section 4.4 explains the facilities for managing virtual address-spaces. Section 4.5 covers user-process execution and Section 4.6 covers I/O port access.

4.1 CHARACTERIZING MEMORY

Memory serves various purposes within a kernel implementation. Some pages store kernel code and data, some are mapped to user processes for their code and data, and—on the IA32—some store tables describing the translation from virtual to physical addresses (see Section 2.2). From a correctness and safety perspective, different operations are valid on a page of memory depending on its function in the system at a particular moment in time. Kernel code pages, for example, should never be mapped to a user thread, and in many cases should not be written to by anyone once the boot process is complete. User code and data pages, on the other hand, may be mapped freely and might even be shared among multiple user processes. Finally, page-tables and page-directories need to be written to by the client kernel, but only in very specific and limited ways.

We reflect the different roles of memory into our design by associating a dynamic status value with each physical page of memory. The dynamic status reflects each page's current use in the system, which falls into one of four categories:

Normal Page: A page of memory that does not have any special semantics associated with it by the kernel. Normal pages may contain user or kernel data; they may also be shared between multiple user processes or between a user process and the kernel if the client kernel requests such sharing.

Page-Directory Page: A page that is in use as a page-directory. In our model, we assume that page-directory pages may be dynamically created from unallocated normal memory pages during the execution of an operating system.

Page-Table Page: A page that is in use as a page-table. We distinguish page-tables from page-directories because certain operations are only valid on page-directory pages. As with page-directories, page-tables can be dynamically created from normal memory pages.

Environment Page: Any page that must be present for the kernel code to run. This might include the code and heap for the run-time system as well as the kernel code itself. In our implementation, this set includes the C and Haskell heaps and the kernel code.

Status information plays an important role in protecting memory-safety for an H-based system. Separating environment memory and translation table memory (page-directories and page-tables) from user memory (allocated normal pages) is a fundamental requirement for memory-safety in operating systems. We must ensure that no user program can interfere with the virtual-to-physical memory mappings of the kernel or other users and that no one can corrupt the execution environment[1]. We will formally define this memory-safety property in Chapter 5. Tracking the current usage of each memory page allows the interface implementation to enforce memory-safety through dynamic checks that guarantee that a client of H never changes the status of a page in an unsafe way.

Client kernels cannot subvert the intended scheme for protecting status transitions because H does not provide them with direct memory access. Instead, client kernels control memory indirectly through the operations of the H interface. In a sense, this splits the two hardware privilege levels that we traditionally rely on for kernel implementations into three, with the third level being implemented purely in software. An H client has the full power necessary to configure and run the system—including setting the policies for how user programs will interact—but every operation is checked for safety violations. The strong type system and

[1] We use the term *execution environment* to refer to the collection of services that have an implicit role in the execution of user programs. This includes the run-time system, the C and Haskell heaps, the code and data for H, and the code for the kernel.

module system of Haskell give us confidence that the restrictions in place in the software privilege level are effectively enforced. A client will never accidentally circumvent the protections without a type error or other compile time warning. A similar notion of reduced privilege through software controls also appears in hardware virtualization systems where every hardware operation must go through a virtual machine monitor [5]. The primary focus of the checking in virtual machine monitors is to ensure fair multiplexing of the machine resources rather than memory-safety in the individual kernels, but the mechanisms for achieving resource protection are similar.

We encode memory-safety properties in terms of page status values by restricting the states in which certain status transitions may take place. Figure 4.2 illustrates the allowable transitions between status values. The safety of a particular transition typically depends on the current state of the page. For example, a normal memory page can only be converted into a page-table or page-directory if it is not currently mapped to any users. Some transitions are never allowed: the set of environment pages is constant so no page can ever become or be converted from such a page. This allows us to enforce strong separation between the kernel (including H) and the execution environment without a complicated set of run-time checks[2].

[2] For the purposes of reasoning about memory-safety, we ignore the indirect effect that the kernel has on the environment pages by executing. During the course of execution, the Haskell run-time system will allocate and manipulate memory on the kernel's behalf within the protected environment pages. We assume that the run-time system functions correctly. Verifying that GHC maintains a well-formed and memory-safe heap, where updates within client kernel data structures do not affect the contents of H data structures, is an interesting but separate problem.

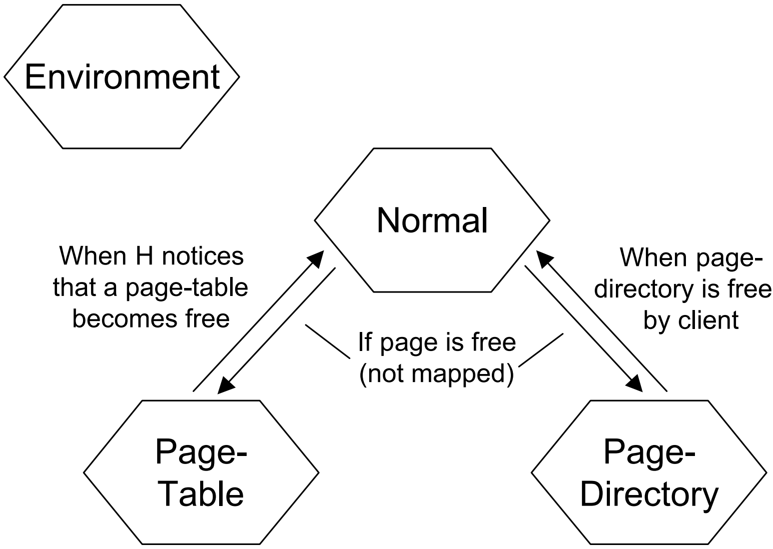

Environment
Normal
When H notices that a page-table becomes free
When page-directory is freed by client
If page is free (not mapped)
Page-Table
Page-Directory

Figure 4.2: The allowable transitions that memory pages may make between different status assignments. A free page—a normal page that is not mapped in any page-directory—may be turned into a page-table or a page-directory page. H manages page-table memory, automatically freeing a page-table page when it no longer contains any entries. The client explicitly allocates and frees page-directories using the H API functions that we will cover in Section 4.4.1. Environment pages never change status.

4.2 EXAMPLE: ADDING A MEMORY MAPPING IN H

Recall from Section 4.1 that an environment page is any page that is required for the kernel to run, such as the implementation language heap or a page of the kernel code. In the implementation of H, we ensure that such a page is never mapped to a user process by making sure that the clients of H are never given a mechanism for accessing those pages. Clients access memory through a *memory handle*, which is a tightly controlled datatype for describing regions of physical memory. The datatype, called `PhysicalRegion`, contains a description of the physical-address range it represents and a type that indicates whether the region represents normal memory (RAM) or I/O memory (like the video RAM). We use the Haskell module system to restrict client access to memory handles: when we export the datatype from the module where it is defined, we export the type name, so that programmers can refer to the type `PhysicalRegion`, but not the value constructor. By using this approach, we can treat memory handles like an unforgeable capability, even though they are just regular values.

H constructs `PhysicalRegion` handles that precisely describe the memory available to the client—the physical memory on the machine that is not being used for environment pages. The `initialRegions` primitive returns these handles.

```
initialRegions :: H [PhysicalRegion]
```

An essential requirement of the H implementation is that `initialRegions` constructs handles to the correct set of pages: no environment pages should be included in the regions referenced by the handles and every other usable page of memory should be included in exactly one region. The absence of environment pages from the set of initial regions helps us to establish the correctness of the H implementation overall: to be memory-safe, H should never construct a handle to any environment page. The client generates new handles through the `deriveRegion` primitive, which creates a `PhysicalRegion` handle to a sub-region of an existing region (with the same physical type).

```
deriveRegion :: PhysicalRegion -> Fpage Physical -> Maybe PhysicalRegion
```

The key property of `deriveRegion` is that it never creates a handle to memory that is not contained in a `PhysicalRegion` that was returned by a call to `initialRegions`. In this way, we establish an inductive argument in support of the property that H does not create handles to environment pages. If the regions exposed by `initialRegions` do not include any environment pages, and if `deriveRegion` is implemented correctly, then `deriveRegion` will never return a handle to an environment page. Figure 4.3 illustrates the relationship between these two primitives.

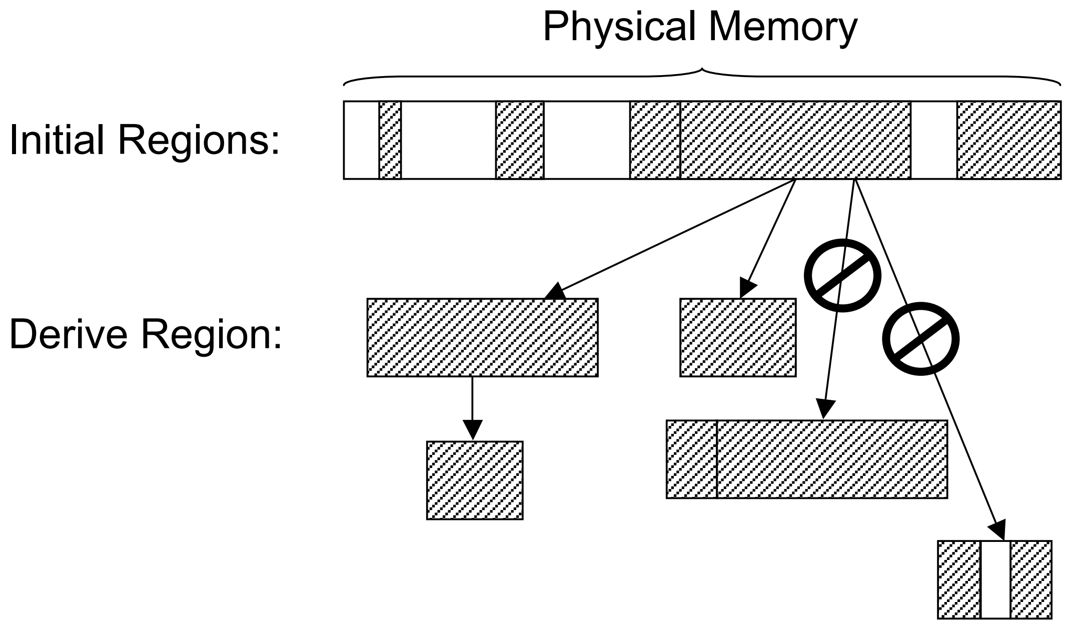


Figure 4.3: Illustrating the relationship between `initialRegions` and `deriveRegion`. `initalRegions` exposes the portion of physical memory that is not in use for environment pages. In the diagram, available memory is shown in gray while protected environment memory (that cannot have handles constructed to it) is shown in white. `deriveRegion` creates new handles to sub-regions of existing regions, but will not create a handle to any memory that lies outside of the initial regions. In the diagram, unadorned arrows indicate a region that can successfully be derived from an existing region, while arrows with a cross through them indicate a region that cannot be derived because it combines multiple existing regions or it includes environment pages.

To add a mapping to a virtual address-space, the client must supply H with the handle for the physical memory to be mapped. This design ensures that the client can never map any environment pages to a user process. We present the full design of the user-level memory management facilities in Section 4.4.2. Abstracting over the details and datatypes that we have not covered yet gives the type for the function that adds a new mapping:

```
addMapping :: PageMap -> VirtualAddressRange -> PhysicalRegion -> H Bool
```

`PageMap` is an abstraction for an address space. In the IA32 implementation of H, the term page-map refers to the collection of mappings reachable through a page-directory (see Chapter 2.2). We use this terminology to avoid over-specializing the interface design to two-level tables. We sometimes refer to page-directories and page-tables generically as page-map pages, but only in cases where the distinction is not important. The `VirtualAddressRange` describes a range of virtual addresses to be mapped—in general this range can be as small as a physical page and as large as the amount of the available memory. The `PhysicalRegion` parameter is a handle to physical memory. The result, of type `Bool`, indicates whether or not the mapping was successfully added to the virtual address space.

Other properties discussed in Section 4.1—such as the requirement that a page-table page is never mapped to a user process—cannot be expressed using types alone because the role of each page may change over time. For these properties, we rely on dynamic checks based on page status values at the time of a requested operation. As an example of dynamic safety checks in the H interface, consider the internal implementation of the `addMapping` function whose signature was given earlier. This function must perform checking to ensure that the parameters are valid and to ensure that the desired page-type properties hold. A physical region can only be mapped if all of the pages that make up the region are normal pages—mapping a page with page-table or page-directory status to a user program will violate our desired memory-safety property. The `addMapping` function will only add new memory mappings when the page status checks indicate that the operation is safe.

4.3 SYSTEM CONFIGURATION

In this section, we examine the low-level structures that are provided by H for communicating information about the hardware configuration to a client kernel. Section 4.3.1 presents the basic structures that H uses to represent memory, including more specifics about the `initialRegions` and `deriveRegion` primitives. Section 4.3.2 explains the mechanisms available in H for loading executable modules through a bootloader and mapping them into user space.

4.3.1 Physical Memory Configuration

The foundation of our correctness argument for H centers around the correct management of physical memory. One mechanism for controlling physical memory is to statically restrict the set of memory pages that the client may access. We achieve this by encapsulating the representation of memory pages as abstract types, implemented using the Haskell module system.

Flexible Address Ranges

We introduce the notion of flexible address ranges for describing areas of memory. In general, an area of memory is a range of addresses described by an arbitrary start address and an arbitrary length. In practice, not all start addresses or lengths are useful. For example, an operating system running on the IA32 architecture can never map 512 bytes because the architecture does not allow memory mappings that are smaller than a page (4096 bytes).

Following the example of L4 [62], we describe areas of memory with a type that captures the implicit practical restrictions on area sizes and locations as explicit restrictions on the values of that type. The restrictions on these *flexible pages* (*flexpages* or *fpages* for short) are as follows:

- The size of a flexpage is 2^n bytes for some natural number n, such that 2^n is greater than or equal to the minimum page size on the target architecture. This means that every byte in the flexpage is uniquely identified by an n-bit offset.

- The start address of a flexpage is aligned to its size. This means that the start address will be a multiple of the flexpage size.

Though the flexpage scheme does rule out some potentially useful address ranges (e.g., some structures used in graphics programming take up an odd number of pages), we have not encountered a case in practice where the size and alignment requirements are overly restrictive. If necessary, the client can round up to the next valid flexpage or use multiple flexpages to precisely describe a memory area of interest. Flexpages can be used to describe both virtual and physical regions of memory.

We define a new type for describing flexpage sizes called `LogSize`. The client may freely construct values of this type, but the semantics of the type capture the flexpage size requirements: `LogSize` represents the base-2 logarithm of the total size in bytes. For example, a `LogSize` value of 12 corresponds to a region of size 2^{12} bytes. The only exception is that the `LogSize` value zero is special; it encodes the zero-size flexpage, rather than a flexpage of length one. Some values of `LogSize` are invalid on a particular architecture by the flexpage rules because they correspond to region lengths that are less than a physical page. Such values are treated as zero when they are used in the construction of flexpages.

Every flexpage is guaranteed to obey the size and alignment requirements because we hide the representation behind the private type `Fpage a`. The type parameter `a` is meant to be either `Physical` or `Virtual` to distinguish the kind of memory referred to by the flexpage. Thus, a flexpage describing a range of virtual addresses would have the type `Fpage Virtual`.

```
data Physical
data Virtual
```

`Physical` and `Virtual` are phantom types whose sole purpose is to refine the type we give to address ranges. We cannot construct values of these types. We choose this representation for flexpages to provide better documentation in the types and to help avoid confusion between flexpages used for different purposes. We use the same technique for addresses, representing both virtual and physical addresses as 32-bit words but using a phantom type to improve documentation.

```
type Addr a = HWord
```

As with flexpages, `Physical` and `Virtual` are the only intended instantiations of the type parameter `a` for addresses.

With a private constructor, the client must create flexpages through the interface function for making flexpages, called `fpage`. The arguments to `fpage` are the base address of the memory area and the desired size.

```
fpage :: Addr a -> LogSize -> Fpage a
```

The constructor always returns a valid flexpage by aligning the base address to the size. If the size is too small (less than the page size on the target architecture), then the result will be a zero-size flexpage. We could make better use of the types by returning a `Maybe (Flexpage a)` instead of the zero value of the flexpage type. The choice does not impact safety. We choose the C-style failure model in this case to provide a better match with the typical uses of the primitive in our L4 implementation.

Access Rights

Access rights control whether or not the contents of a particular region of memory may be read, written, or executed. Every memory mapping has an independent set of access rights associated with it, represented by the type `Perms`, so a single physical page might be mapped with different permissions in two distinct address-spaces. A client controls the assignment of access rights to memory regions through the mapping management functions of the interface, which we will cover in Section 4.4.2.

Regions of Physical Memory

At start-up, the H implementation reserves a portion of the physical memory installed in the machine to use as environment pages. The rest of the available memory on the machine is given to the client kernel to use for any purpose—for page-table memory, for user-processes, or for its own kernel-specific data structures. Some environment pages, such as the memory that stores the kernel code, have the same location in any run of the system. Other protected areas of memory are dynamically configured by H. For example, H chooses the size and location of the Haskell heap based on the available memory of the machine. As we saw in Section 4.1, these pages cannot be safely mapped to a user-process. Any remaining memory becomes available to the client through `initialRegions`. Figure 4.4 illustrates the division of memory between the environment and the client.

We use flexible address ranges to describe the regions of physical memory that are available to the client. Accordingly, the size and alignment restrictions on flexpages also apply to regions: the size of a region must be a whole number of physical pages and the start address must be aligned to the region's size. Each physical region also has a static type that describes the nature of the memory contained in the physical region (normal RAM or memory-mapped IO pages).

```
data PhysicalType = RAM | IOM IOType
```

We support two kinds of memory-mapped I/O regions: video RAM and the frame buffer for VBE graphics[3]. We consider this to be a proof of concept for exporting

[3] VBE (VESA BIOS Extension) is a graphics standard intended to simplify access to graphics

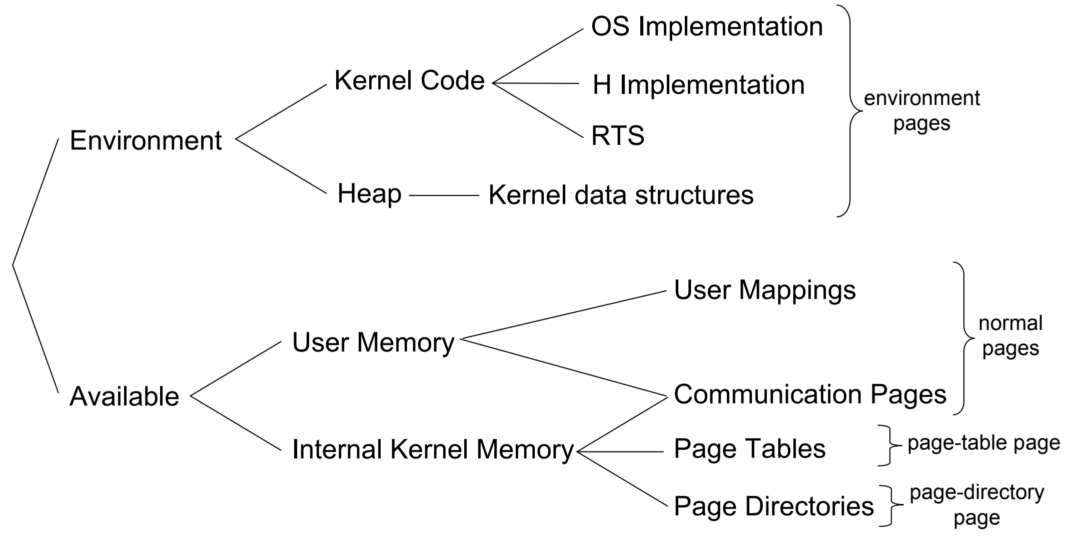


Figure 4.4: During the bootstrapping process, H reserves a portion of physical memory for the kernel's execution environment. The remaining memory becomes available to the client for use as page-tables, page-directories, user memory, and communication pages for sharing information between the kernel and user processes.

I/O device information using physical regions, rather than an complete implementation. Other kinds of memory-mapped I/O could easily be supported by extending `IOType` and adding the appropriate configuration code in H.

```
data IOType = VideoRAM
            | FrameBuffer (Addr Physical) HWord
```

The arguments to the frame buffer constructor describe the location and size of the buffer. We store this information so that a client can obtain precise information about the frame buffer location, which may not have a valid region size or

devices without specific knowledge about the target hardware [3]. We expose the frame buffer to support client-level graphics drivers, but this functionality is not used in existing clients of H.

alignment. There may be additional VBE graphics information that a client implementing VBE graphics will need from from the bootloader; any such information can easily be added if necessary.

Now that we have introduced the types for describing address ranges and physical memory types, we can examine the implementation of the memory handle type presented in Section 4.2. A `PhysicalRegion` is a record with two fields: a physical flexpage representing the area described by the handle and a memory type.

```
data PhysicalRegion = PhysicalRegion {
                        region  :: Fpage Physical,
                        memType :: PhysicalType
                      }
```

Clients of the interface can examine the contents of physical regions—such as their size and type—but cannot create new values of the type. Recall from Section 4.2 that clients access regions through a combination of two primitives: `initialRegions` discovers the available memory on the machine and `deriveRegion` creates sub-regions of those initial regions.

```
initialRegions  :: H [PhysicalRegion]
deriveRegion :: PhysicalRegion -> Fpage Physical -> Maybe PhysicalRegion
```

We will cover the process of dividing physical memory into environment pages and the other categories in detail in Section 6.3. For now, it is sufficient to understand that H reserves a portion of physical memory that should not be accessed by the client and that it creates `PhysicalRegion` handles to all of the areas of memory that the client is allowed to use.

The following rules describe the relationship between an existing region and a new region produced by `deriveRegion`. Assuming that the existing region starts at physical address, *start*, and has a size, *logsize*, a `deriveRegion` request will produce a valid sub-region of the original when the following properties are true of the new start address, *new_start*, and size, *new_logsize*:

$$\begin{aligned} \mathit{new\_logsize} &\geq \mathit{pageSize} \\ \mathit{new\_logsize} &\leq \mathit{logsize} \\ \mathit{new\_start} &\geq \mathit{start} \\ \mathit{new\_start} + 2^{\mathit{new\_logsize}} &\leq \mathit{start} + 2^{\mathit{logsize}} \\ \mathit{new\_start} \mod (2^{\mathit{new\_logsize}}) &= 0 \end{aligned}$$

If these restrictions are met, then `deriveRegion` returns a new region based at *new_start* (*start* + *off*) with size *new_logsize*. Otherwise, the function returns `Nothing` to indicate that the requested region could not be created.

4.3.2 User-Code Modules

The H interface provides limited support for managing the code and data placed in memory by the bootloader. Currently, we only expose information about executable multiboot modules to clients, but the design could be expanded to incorporate the rest of the multiboot standard. Executable modules have been sufficient for running user-space programs in our L4 implementation (see Chapter 7).

There are two key pieces of information in an executable multiboot module: the entry point to the module and a description of the memory that it occupies. The entry point is simply a word that contains the starting address of the program. The memory that is used by a module is described by a collection of physical regions.

```
data Module = Module {
                modArea  :: [PhysicalRegion],
                modEntry :: HWord
              }
```

As with physical regions, the client can examine the contents of an executable module, but cannot construct new modules themselves. We represent module memory as a list of physical regions because the number of pages occupied by a module may not fit the region size requirements. Clients are allowed to map module memory to users, as with any memory described by a handle, but the physical regions that are associated with bootloader modules are not exported through the normal physical region interface (i.e., they are not contained in `initialRegions`). We choose to separate the memory in this way to ensure that the distinction between free memory and bootloader memory is clear to the client, even though the client could ultimately use both types of memory in the same way.

The client discovers the modules loaded in the current run of the system using the `modules` command, which returns a module descriptor for every executable bootloader module except for the kernel. There would be a memory-safety violation if the client could access the kernel module, because then the client could overwrite its own code and the code for H.

```
modules :: H [Module]
```

There are no other operations for working with the `Module` type.

4.4 VIRTUAL-MEMORY MANAGEMENT

Virtual-to-physical memory mappings are a key component of memory-safety because they establish the views of memory accessible to user programs and the kernel. The presence of a memory mapping can be seen as permission to read, modify, or execute a particular page of memory. We enforce the desired separation between entities in the system—the execution environment, the kernel, and the users—by appropriately controlling these views. We rely on a combination of static, type-based arguments and dynamic checking to enforce the safe construction of memory mappings. We particularly focus on the safety of the virtual memory management operations: page-directory creation and deletion (Section 4.4.1); adding, modifying, and removing user-visible mappings (Section 4.4.2); adding kernel-visible mappings (Section 4.4.3); and reading/writing memory mapped in kernel space

(Section 4.4.3).

In addition to safety, generality is an important goal of the virtual memory interface design. We hope to support user-level separation policies in client kernels [82, 66] by providing expressive memory management functions that do not include unnecessary policy of their own. A correct implementation of H will guarantee that there is no accidental flow between user programs—that is, no data movement through memory that the client has not explicitly enabled through a mapping operation—but not restrict intentional communication. For example, memory-safety requires that page-directories and page-tables are not mapped to users, but we do not impose any sharing policy on normal page sharing between user-level programs or a user and the kernel. Throughout the design, our goal is to supply mechanisms that do not preclude separation between user programs, rather than a policy that enforces it [68].

4.4.1 Address Spaces

The virtual-address space of the machine is split into two components: user-space, which contains the mappings for user-level programs and data; and kernel-space, which contains the mappings for kernel code and data. We use the user/supervisor bit in page table and page directory entries, as described in Section 2.2, to ensure that memory in kernel-space is not accessible to user programs. Our implementation places the boundary between kernel-space and user-space at 3 GB so that the kernel lives in high memory. The choice of boundary and kernel location is arbitrary, but consistent with Linux [61].

Client kernels have total control over the user portion of the virtual address-space. H will never place mappings in that area except for those explicitly requested by the client. There are no restrictions on the virtual addresses that can be mapped. Kernel-space is divided into two areas: one that H controls and one that the client controls. H must own part of this address-space to protect the kernel code, H code and data, and Haskell run-time system. The remainder of the virtual address-space is managed by the client kernel. H identifies the parts of kernel space that are available to the client through the constant `kernelMappableVirtualAddresses`.

```
kernelMappableVirtualAddresses :: H [Fpage Virtual]
```

We describe the kernel controlled addresses as a list of flexpages describing the free virtual-address ranges within kernel memory.

Clients manipulate the mappings of a user address-space by inserting and modifying entries in the page-map data structure. H provides access to page-maps through a restricted set of operations, because page-map correctness is important for the safety of the system. If a client could construct arbitrary page-maps, then they could circumvent the protection mechanisms of H by corrupting H's private data or the execution environment itself (essentially, the client could obtain full access to physical memory). We maintain the integrity of each page-map data structure by defining a type called `PageMap` with a hidden representation. Clients must use the interface to allocate page-maps in a controlled and safe way. When allocating a page-map, H converts a free page of memory into a page-directory by setting its status to the page-directory value and installing mappings for the environment pages. If the argument to the conversion is not free, then the operation will not be safe. Once the page has been converted, the client may not use that page for any other purpose until the page-map is explicitly freed. We enforce this restriction by dynamically checking the status of pages the client tries to map: any with the page-directory status will not be mappable.

Page-map creation happens in two stages. First the client turns a page-sized physical region into a new value called a page-map page. The function `createPageMapPage` validates the size of a region and, if the size is correct, creates an unforgeable `PageMapPage` handle containing that region. Both page-directories and page-tables are created from pages that have been pre-validated with createPageMapPage. A PageMapPage also stores information about where the region should be mapped in the kernel virtual address space so that H may read and write to the page. The second parameter to createPageMapPage is the address where the client would like the page-map page to be mapped for H (which must lie in the kernel controlled portion of the address space); the full details of H's access to page-map memory will be covered in Section 6.4.

```
createPageMapPage :: PhysicalRegion -> Addr Virtual
  -> H (Maybe PageMapPage)
```

If the parameters pass the validity checks, then createPageMapPage returns a page-map page that can be passed as an argument to the page-map creation function or supplied for H to use as a page-table (the H implementation allocates page-tables on demand in the mapping functions using client-supplied memory pages, we will cover these operations in Section 4.4.2).

```
allocPageMap :: PageMapPage -> H (Maybe PageMap)
```

Page-map allocation creates an empty page-directory (with the page-directory status) that only contains the mappings for the environment pages. Before creating the page-directory, allocPageMap will check if the page is free; this is a necessary condition for safely creating a new page-directory. As a side effect, allocPageMap will install an H-accessible mapping for the page-directory at the kernel virtual address specified by the PageMapPage.

The client frees a page-map with the analogous function, freePageMap. The result of freePageMap is a handle describing the memory previously occupied by the page-directory and the memory for any page-tables that become free as a result of the operation. These newly freed pages are returned as a PhysicalRegion list for reuse by the client kernel.

```
freePageMap :: PageMap -> H [PhysicalRegion]
```

Our implementation of H allows the current page-map to be freed by switching to a default system page-map that only contains mappings for the environment pages. A version of `freePageMap` that does not allow the current page-map to be freed would also be a reasonable implementation choice, but would require a slight adjustment to the type of the operation to signal failure.

Page-map management illustrates an important aspect of the H design: the combination of static, type-based safety guarantees with run-time checks. There are some properties that we can enforce entirely statically using Haskell datatypes. For example, our implementation can rely on the fact that a `PageMapPage` corresponds to a single page of memory because the only function for creating a `PageMapPage` checks that. Other properties are best enforced through a combination of static and dynamic checks. Through the implementation of `allocPageMap`, we can be sure that every page with the page-directory status is mapped to an address that H can read and write. We also know that H makes every such page accessible through a `PageMap` handle. But even though the client cannot construct `PageMap` values—except through `allocPageMap`—we cannot be sure that every `PageMap` has the page-directory status. This asymmetry stems from the fact that H clients can free `PageMap` values but H cannot reclaim the handles because they are just values that clients may hold onto even after the corresponding page-directory has been freed. Thus, the H implementation must combine the static information from types with dynamic status validity checks to be sure that `PageMap` operations are safe. More details about the implementation techniques used to guarantee safety will be covered in Sections 6.4 and 6.5.

4.4.2 User Memory

Each page-map structure has an independent set of mappings in the user portion of the virtual address-space that control which memory is accessible to user processes. The same physical memory may be mapped in the user area of many page-maps if so desired. The client modifies these mappings indirectly using the operations of H. There are three essential operations available to the client: adding a new mapping to a contiguous block of physical memory, modifying the read/write permissions on an existing mapping, and removing a mapping. Each of the mapping functions operate on a single previously allocated page-map.

As with the creation of `PageMap` objects, the client explicitly manages any additional memory that H requires for page-map storage. For example, with a two-level page-map, the function for adding a mapping might allocate a page-table page in which to store the mapping. When the client removes mappings from a page-map, the memory that stored those mappings will be freed and returned to the client if the pages are no longer needed for other mappings. H never retains control of any page that is not in use.

The `addMapping` operation adds mappings from a range of virtual addresses (expressed as a flexpage) to a block of physical memory of the same size. By using a `PhysicalRegion` value, we guarantee that the physical memory is safe for the client to map to user programs.

```
addMapping :: [PageMapPage] -> PageMap -> Fpage Virtual
                -> PhysicalRegion -> Perms -> H (Maybe Bool)
```

The additional arguments expose some of the complexity of `addMapping` that we skirted over in its initial introduction in Section 4.2. The `Perms` argument specifies the access rights to attach to the mapping. These permissions control the read/write/execute permission on the mapping, but not the user/supervisor setting because all mappings added through `addMapping` must be in user-space. The `PageMap` argument indicates the address-space where the mapping should be added. The list of `PageMapPages` is a supply of pre-validated memory pages that the client provides to H for page-table storage. In the current implementation of H, we always use zero or one of these pages. A `Just` result indicates success of the mapping operation; the Boolean component indicates whether or not a `PageMapPage` was needed. For implementations that potentially use more than one `PageMapPage`, the Boolean result may not be a good fit; a natural number indicating the number of page-tables allocated would be more appropriate.

The `modifyMapping` function changes the permissions attached to an existing mapping. As such, `modifyMapping` will never need to allocate new page-map storage; it works with the structures that are already there. The arguments to `modifyMapping`, in order, are the page-map to modify, the virtual flexpage of the mapping to be modified, and the new permissions to attach to the mapping. The result of `modifyMapping` indicates whether or not the modification succeeded.

```
modifyMapping :: PageMap -> Fpage Virtual -> Perms -> H Bool
```

If the virtual flexpage does not correspond to an existing mapping (either the flexpage overlaps with the kernel portion of the virtual address-space or the flexpage corresponds to memory that is not mapped in the specified page-map), then the function does not change the page-map. We do not allow the client to set the permissions of a mapping to nothing using this function—for that they must use `removeMapping`. Attempting to do so causes `modifyMapping` to return `False` without making any changes to the page-map.

The final operation on mappings is deletion. The function `removeMapping` deletes the mapping that corresponds to a particular virtual flexpage. The arguments to the function are the page-map to modify and the flexpage of the mapping to be removed. The result is the list of pages that were freed by the operation.

```
removeMapping :: PageMap -> Fpage Virtual -> H (Maybe [PhysicalRegion])
```

The operation behaves optimistically and modifies any portion of the provided flexpage that it can (even if some portion of the flexpage is not mapped in the specified address space). The areas of memory that get modified by the operation will always be flexpage-sized regions.

The client may also read information about a user mapping that is already present in a page-map, for example, whether or not a particular page is mapped, and what the usage information is for a mapped page (whether the page has been read or written). We represent page-map entry information with values of the type `MappingInfo`. The client reads the mapping information for a particular page of virtual memory using the function `readMapping`.

```
readMapping :: PageMap -> Addr Virtual -> H (Maybe MappingInfo)
```

The first argument is the page-map to examine and the second argument is a virtual address in the page of interest. H aligns the virtual address argument to a page-boundary, rounding down if necessary, and finds the information for that virtual address by consulting the page-directory and page-table bits. If the page is not mapped, then `readMapping` returns `Nothing`. Otherwise, the function returns a `MappingInfo` object that the client can examine using accessor functions provided by H.

```
mappingAddr :: MappingInfo -> Addr Physical
accessed    :: MappingInfo -> Bool
dirty       :: MappingInfo -> Bool
```

The `mappingAddr` function returns the physical address that the virtual page of interest is mapped to. The client can also check whether or not the page has been read with the `accessed` function, and whether or not the page has been written with the `dirty` function.

4.4.3 Kernel Memory

H supports the ability to add kernel mappings that are visible in every virtual address-space. The only constraint is that the mappings lie in the portion of kernel-space not in use by H. Any memory that may be mapped to users may be mapped into kernel space. We call these mappings *client kernel mappings* to distinguish them from the H mappings that we use internally for page-map pages. The client manages the addresses where page-map pages are mapped, but, for safety reasons, cannot control the actual mapping operations on such pages.

When the client adds a client kernel mapping, H returns a handle through which the client can read and write the mapped memory. We represent handles with the type `KernelMapping`, which is partially abstract. A `KernelMapping` contains a virtual flexpage describing the mapped region, a physical region describing the underlying memory, and a set of read/write permissions. Clients may access these components of a `KernelMapping` but cannot construct a mapping handle themselves.

We use kernel region handles to ensure that the client only accesses memory that has been mapped. Without this requirement, the client might try to access protected memory (the type of the mapping function guards against such behavior) or the client might request a memory access that causes H to page fault. We choose to prevent kernel faults through checking whenever possible. The lack of page faults in H makes memory errors easier to find and debug.

Handles alone, however, are not enough to prevent page faults; it is also important to guarantee that kernel mappings are accessed with valid permissions. We take the approach that the client can control the permissions attached to kernel mappings, even though this requires extra checking in the implementation of the write operations. A simpler (and potentially more efficient) approach would be to force all kernel mappings to have read/write permissions so that faults can never occur.

Initially, there are no client-accessible mappings in the kernel memory area. A client inserts mappings into this region by calling `addKernelMapping` with the physical region to map to the kernel, the location in virtual memory at which to add the mapping (as a virtual flexpage), the permissions to attach to the mapping, and a Boolean value that indicates whether or not the new mapping should be user accessible.

```
addKernelMapping :: PhysicalRegion -> Fpage Virtual -> Perms
                      -> Bool -> H (Maybe KernelMapping)
```

A mapping cannot be added if the requested location for the mapping is not fully contained in the available portion of the kernel address-space or if the target memory is not a normal page.

The client accesses kernel mappings through interface read and write functions called `readKernelMapping` and `writeKernelMapping`. These operations are similar to the standard `peek` and `poke` functions in the `IO` monad except that they are guaranteed to preserve memory-safety. The first argument to both functions is the kernel mapping to access. The second argument is a word offset that specifies where in the mapping to read or write. The read function returns the word that lies at the requested offset, and the write function requires an extra parameter containing the word to write to memory.

```
readKernelMapping  :: KernelMapping -> HWord -> H HWord
writeKernelMapping :: KernelMapping -> HWord -> HWord -> H Bool
```

The offset parameter is implicitly forced to lie within the region described by the specified kernel mapping. That is, $\mathit{old\_offset} \bmod \mathit{mapping\_size}$. The write operation may still fail because the permissions of the mapping may not be sufficient.

We also provide specialized read and write operations for accessing page-sized kernel mappings (or the first page of a larger region). The maximum offset is a static constant that corresponds to the page size on the machine. We use a bounded type, `Offset`, to describe the restrictions on these accesses.

```
newtype Offset = Offset HWord

instance Bounded Offset where
  minBound = Offset 0
  maxBound = Offset (pageWords - 1)
```

The specialized read and write functions behave similarly to the normal read and write functions, except that the offset is constrained to lie in a single page. The expectation, borne out in the performance results of Chapter 8, is that the specialized versions will be faster because the address validation is done relative to a compile-time constant, rather than a parameter that must be read from the kernel mapping structure at run-time.

```
readWordAtOffset  :: KernelMapping -> Offset -> H HWord
writeWordAtOffset :: KernelMapping -> Offset -> HWord -> H Bool
```

These operations can be used on any kernel mapping regardless of its size, because we never create a kernel mapping that is smaller than a page, but only to access the first page of data. Typically we only use the specialized functions on page-sized mappings, but avoiding a distinction between different kernel mapping sizes prevents additional complications in the interface.

4.5 USER PROCESS EXECUTION

H defines a minimal set of constructs in support of user-level execution. We avoid fixing a specific notion of thread or process, and instead provide the basic building blocks necessary for managing such structures. In particular, we provide fault contexts (a mechanism for saving and accessing the register state of user programs) and interrupt/IRQ handling. With these low-level constructs, the client can define a variety of higher-level representations of user code.

Our representation of register values is an abstraction of the register set on the underlying machine which we define as an algebraic datatype called `Register`.

```
data Register
  = EDI | ESI | EBP | BogusESP | EBX | EDX | ECX | EAX | DS | ES | FS
  | GS | Handler | Code | EIP | CS | Eflags | ESP | SS
```

A fault context is a collection of the values contained in the registers when execution switches from user-mode to kernel-mode, along with some information that describes the reason for the mode switch (the nature of the interrupt or IRQ). Typically the client only accesses the register values of the fault context because H packages up the interrupt or IRQ information in the `Interrupt` and `IRQ` datatypes. We omit the definition of these types here, but will discuss the details of their implementation in Section 6.8.

We represent fault contexts using a handle type called `FaultContext`. The client explicitly allocates fault contexts for running user programs. The client kernel might use a single fault context to run many user programs (with the kernel managing the register values between runs) or the client might allocate a separate fault context for every user-level entity.

```
allocFaultContext :: H FaultContext
```

The client is not responsible for freeing the fault contexts that it creates: the Haskell garbage collector automatically reclaims the associated memory.

The client accesses the information contained in any fault context register using `readRegister` and `writeRegister`. These functions access a particular register from a particular fault context.

```
readRegister :: FaultContext -> Register -> H HWord
writeRegister :: FaultContext -> Register -> HWord -> H ()
```

The register read and write functions always succeed.

H provides a single function for jumping to user-level code, called `execute`.

```
execute :: PageMap -> FaultContext -> H Interrupt
```

The inspiration for this function comes from the `execContext` primitive in the original H interface [39]. We reuse many of the implementation techniques from that implementation, with some minor adjustments, particularly to the way fault contexts are handled . The client specifies the page-map to install prior to execution (which controls the memory that will be accessible) and the fault context to restore (which determines the starting state of the code). H sets up the machine state according to these parameters and then switches the machine into user mode. The user code begins executing at the instruction pointer stored in the fault context parameter and continues running until control returns to the kernel through an interrupt, fault, or IRQ (recall the process through which mode switches occur that we described in Chapter 2). User-process execution is guaranteed to terminate because H sets up a timer that will eventually interrupt the user program and return control to the client.

When control returns to kernel-mode, H packages up the interrupt information and returns that information to the client kernel. The interrupt format is specific to the interrupts that may occur on the underlying hardware platform and is defined as a datatype with one constructor per type of interrupt. Some interrupts carry extra information, for example, an IRQ contains the number of the IRQ that occurred (described by the `IRQ` type) and a page fault carries information about the nature of the fault (the `PageFaultErrorCode`).

Interrupts and faults fall under the domain of H, but I/O interrupts are managed by the client kernel. We supply functions for individually enabling, disabling, and acknowledging a particular IRQ:

```
enableIRQ :: IRQ -> H ()
disableIRQ :: IRQ -> H ()
maskAckIRQ :: IRQ -> H ()
```

The IRQ management functions always succeed.

Originally, the H interface design allowed the client kernel some control over interrupts and faults, namely, the ability to set whether or not interrupts occur in kernel-mode, using the functions `enableInterrupts` and `disableInterrupts` [39].

```
enableInterrupts  :: H ()
disableInterrupts :: H ()
```

Though we think that control over kernel interrupts is an important part of the API, our current implementation does not provide enough support for multi-threaded kernels to allow kernel interrupts. We believe that better multi-threaded support is possible by modifying our implementation to use multiple Haskell threads. The H API would require some extension to enable the client to install Haskell handlers for kernel interrupts. Then, the C-level interrupt handler could invoke the appropriate Haskell handler (in a new thread) when a kernel-mode interrupt occurs. These issues are an interesting topic for future work. Under the current scheme, clients with a short path through the kernel should not encounter issues. Other kernels will have to implement their own method for reducing the delay in handling interrupts.

4.6 INPUT/OUTPUT

In this dissertation, we do not diverge significantly from the input/output facilities of the original H design [39]. We support I/O port access by lifting the port operations of the underlying architecture into Haskell. We also provide basic debugging facilities for use during kernel development.

4.6.1 Ports

The client kernel accesses I/O ports through a direct lifting of the underlying assembly instructions for port access on the IA32. We do not consider safety issues concerning I/O ports in this work. If client kernels avoid using DMA, the port operations will not affect the integrity of the memory structures of the kernel. In future designs it would be interesting to explore greater control over I/O ports through the interface, for example, the ability to control an IOMMU.

Each port has a 16-bit port identification number with a fixed semantics on the machine. We do not attach a semantics to port numbers, and simply represent them using a type synonym for 16-bit words called `Port`. The client may read or write a byte, a short, or a word to any port.

```
inB  :: Port -> H HByte
inS  :: Port -> H HShort
inW  :: Port -> H HWord
outB :: Port -> HByte  -> H ()
outS :: Port -> HShort -> H ()
outW :: Port -> HWord  -> H ()
```

The port operations do not fail.

4.6.2 Debugging

H provides screen printing functionality for debugging purposes. Normally, clients will implement their own printing functions for user programs, but during client development it is very convenient to be able to print messages for testing and debugging. Any impure function, such as output, must be in the H monad, but we do not consider these functions to be a core part of the H design.

Using a stripped down video driver written in C, the client can print a single character to the screen. We implement utility wrappers on this simple function for printing strings and lines.

```
putch    :: Char -> H ()
putstr   :: String -> H ()
putstrln :: String -> H ()
```

The printing functions are memory-safe, but they might fault if the video RAM is not mapped in the current page-map. This is an exception to our usual philosophy that H operations should prevent faults whenever possible and it did lead to unexpected crashes during the early stages of developing H. An alternate implementation might use the serial port instead of the screen and avoid the issue of page faults entirely.

Chapter 5

FORMALIZING OPERATING SYSTEM MEMORY-SAFETY AS A NONINTERFERENCE PROPERTY

A primary motivation for implementing operating systems using Haskell is that the language provides strong type- and memory-safety guarantees. From our choice of implementation language alone, we know that our client kernels do not contain null pointer dereferences, unsafe type casts, or any other safety errors[1]. The same type- and memory-safety guarantees do not hold for the operations of the H interface, because the H implementation uses potentially unsafe features of Haskell (such as pointers), and foreign calls to C. For this code, we must demonstrate safety, rather than relying on language-based guarantees.

Unfortunately, defining a memory-safety property for the H operations is not straightforward. In programming languages, safety typically refers to the absence of unchecked run-time errors with respect to the formal semantics of the language and memory-safety refers to the absence of a collection of memory-related run-time errors, such as a null-pointer dereference or a memory access outside the bounds of an array. If the operations of the H interface have been implemented correctly, then

[1] The correctness of the Haskell compiler and the generated code is required for Haskell programs to be type- and memory-safe. For the purposes of this dissertation, we assume the correctness of the entire GHC implementation, including the compiler and run-time system. Like many large systems, the correctness of GHC has been established through testing and through a feedback loop between its large community of users and the developers. Ultimately, we would like to implement H using a high-assurance run-time system, like the one that is currently under development at Portland State University for the Habit language [73].

they should be memory-safe according to this colloquial definition, but this notion of memory-safety is not sufficient for our purposes. We want to guarantee that the H operations preserve language-based memory-safety in our client kernels, but even an H operation that is itself memory-safe with respect to the Haskell semantics can corrupt the run-time system and introduce memory-safety errors into the client. This is because the H operations have access to all of physical memory, including the Haskell heap and run-time system data. Thus, for an H operation to preserve language-based memory-safety at the client kernel level, we need a memory-safety property for our H operations that is stronger than traditional definitions. In addition to demonstrating that the H operations are memory-safe with respect to the colloquial definition, we must also demonstrate that they do not corrupt the Haskell run-time system or configure the virtual-memory translation tables in a way that would enable such corruption by a client or user program.

The essential characteristics of the H interface with respect to run-time system integrity can be expressed as two properties:

- **Execution Environment Integrity:** Memory occupied by the execution environment should be distinct from memory that is used for other purposes. No user-process or client kernel should be able to write to environment pages or alter the mappings to those pages. As discussed in Section 4.1, the execution environment refers to all of the software components that are essential for the client kernel to execute, including the run-time system, Haskell heap pages, and C heap pages.

- **Address-Space Integrity:** Memory that implements the view in a particular address-space (such as a page-table or page-directory page) should be distinct from the memory that is used for other purposes. No user-process should be able to write to these pages. Client kernels should only write to these pages using operations that are appropriate for their type.

We define the memory-safety property for the H interface as the combination of execution environment integrity, address-space integrity, and traditional memory-safety. If our implementation of H satisfies this definition of memory-safety, then we can be sure that our kernel will not crash because the run-time system has been corrupted and that a buggy or malicious program running on top of H cannot insert values into the Haskell heap. In the remainder of this chapter, we focus on execution environment integrity and address-space integrity because these are novel properties that have not been explored in other work. We leave the proof that H is memory-safe in a traditional sense as future work (see Chapter 10).

To enforce execution environment integrity and address-space integrity within the H implementation we utilize a combination of run-time checks and abstract datatypes to make sure that no direct writes to the run-time system memory via H operations are possible. Indirect writes to the Haskell heap during the course of the H's execution do occur, but the semantics of the language, assuming a correct implementation, guarantee that such writes will not cause a type- or memory-safety violation. If execution environment integrity and address-space integrity hold for the H implementation, then we can ensure that the other system components do not corrupt the run-time system using the following mechanisms:

- **Client Kernels:** Our client kernels are written entirely in the safe portion of Haskell and do not make foreign calls. We can therefore rely on the safety of Haskell for preventing unsafe accesses to the run-time system data because the run-time system does not permit Haskell programs to access its data structures or violate the safety guarantees of the language. Execution of the client kernel will impact the values stored in the run-time system memory, but not in a way that can cause memory-safety errors.

- **User Programs:** For user programs, we rely on hardware protection using the MMU. When we configure the run-time system code and heap, we map

> all of the memory with kernel-only permissions. User programs always run in user-mode and will fault if they attempt to access the run-time system data.

We assume that client kernels behave in a safe way (when running on an uncorrupted run-time system) as part of our assumption that the GHC implementation is correct, type-safe, and memory-safe. To verify that user-programs do not corrupt the run-time system, we need to formalize the property that the execution environment pages are always mapped with kernel-only permission, but do not need to model any other aspects of user-process execution.

In the remainder of this chapter we will show how the execution environment integrity and address-space integrity components of the memory-safety property can be formalized as a separation property by instantiating the Rushby noninterference framework [82]. We will connect the instantiation of the framework to our implementation via an abstract model of the H operations. Establishing a formal connection between the model and the implementation is an interesting topic, but we do not tackle this work in this dissertation. Nevertheless we have attempted to maintain an informal level of consistency between these two descriptions of H and to point out any places where they diverge.

We begin by reviewing the Rushby framework for reasoning about noninterference properties in Section 5.1. Section 5.2 discusses the notational style that we will use throughout the chapter. Section 5.3 presents our instantiation of the system state. Section 5.4 describes our division of the system into protection domains and our instantiation of the various components of the Rushby framework to support our memory-safety interpretation. Section 5.5 explains the safety property in terms of our security policy between domains. Section 5.6 defines the static invariants on the state that we expect all operations in our system to uphold. Section 5.7 specifies the dynamic behavior of the key H operations. Section 5.8 defines the execution function for our Rushby instantiation and sketches a proof that our instantiation satisfies the necessary properties set out by the Rushby framework.

5.1 BACKGROUND: RUSHBY'S NONINTERFERENCE FRAMEWORK

This section summarizes Rushby's framework for reasoning about system security [82]. The basis of this framework is an abstract model of computer systems and a representation of noninterference policies. Rushby uses these foundational elements to define security, to formalize assumptions, and to prove that a system satisfying those assumptions is secure.

5.1.1 System Model

Rushby [82] models computer systems as state machines. An action is a state transformer that also produces some output. The behavior of the system is specified by the functions *step* and *output*; *step* performs a single state transition whereas *output* extracts the result of the action.

Definition 1. *A system (machine), M, consists of*

- *a set of* states, *S, with an initial state* $s_0 \in S$,
- *a set of* actions, *A, and*
- *a set of* outputs, *O,*

together with execution functions,

- *step* $:: S \times A \rightarrow S$, *and*
- *output* $:: S \times A \rightarrow O$. □

To express security constraints on the system, Rushby supplements the basic execution model with a notion of *protection domains* and a *security policy.* Actions in the system are partitioned into a set of domains, D, as specified by the function $dom :: A \rightarrow D$. A reflexive binary relation on domains, $\rightsquigarrow$ (pronounced *interferes*), expresses the security policy of the system.

$$\rightsquigarrow :: D \times D \rightarrow Bool$$

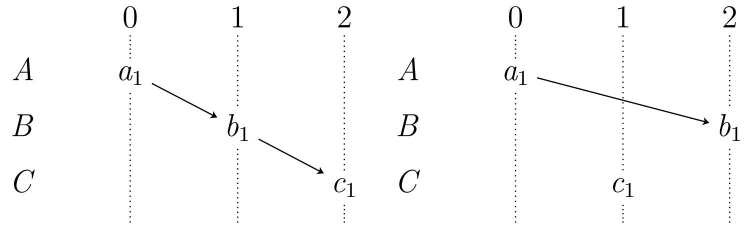
A domain u interferes with a domain v if information is allowed to flow from u to v, meaning that v can observe the effects of u's actions. The symbol $\not\rightsquigarrow$ (pronounced *does not interfere*) represents the complement relation.

While the specific nature of states is abstract, the proof of the security definition requires some mechanism for determining what information is observable to a particular domain in each state. To this end, Rushby introduces an equivalence relation, $\sim :: S \times S \times D \rightarrow Bool$, called the *view-partitioning relation.* Two states, s and t, are equivalent from the perspective of a domain u, written $s \overset{u}{\sim} t$, if u cannot distinguish s and t. The precise meaning of $\sim$ is a parameter to the formulation. This relation extends naturally to an equivalence, $\approx$, over a set of domains, C.

$$s \overset{C}{\approx} t \equiv \forall u \in C.\ s \overset{u}{\sim} t$$

A key characteristic of the model is the abstract nature of states, actions, and outputs. System developers have significant flexibility in the instantiation of these parameters, so the formulation can be applied to a wide variety of systems. The *output* function is a good example of this flexibility. Bevier and Young [7] criticize the use of *output*, because many systems do not produce output in the traditional sense. Their model replaces *output* with a *view* function, that extracts a portion of the state. However, the distinction between *output* and *view* is gratuitous, as *view* is a valid instantiation of *output* in the Rushby framework.

$$policy = \{A \rightsquigarrow B, B \rightsquigarrow C\}$$



(a) a_1 interferes with c_1 via intervening action b_1

(b) a_1 cannot interfere with c_1

Figure 5.1: Intervening actions allow indirect interference between domains. A, B, and C are domains in a system. The policy contains $A \rightsquigarrow B$ and $B \rightsquigarrow C$. In the diagram, columns represent execution steps, where the numeric label corresponds to the state an action executes in (e.g., a_1 executes in s_0 producing s_1). An arrow between two actions indicates potential interference.

5.1.2 Characterizing Domain Interactions

A security policy specifies which domains are allowed to interfere, but does not capture all information flow between domains. For example, consider a system with three domains, A, B, and C, and a policy $\{A \rightsquigarrow B,\ B \rightsquigarrow C\}$, as pictured in Figure 5.1. Though the policy does not explicitly state that A interferes with C, information can flow from A to C indirectly through actions in B. If $\rightsquigarrow$ is transitive, then A interferes with C in all situations. However, we are concerned with systems where $\rightsquigarrow$ is not assumed to be transitive, so we need a strategy for determining if information flow from A to C occurs in a particular sequence of actions. A only interferes with C through B, so an action in A only interferes with an action in C when an intervening action in B is present.

Rushby captures information flow between domains formally with a function called *sources* of type $A^* \times D \to \wp(D)$. The notation A^* represents the set of finite sequences whose elements belong to the set A. The empty sequence is denoted by [] and the sequence obtained by prepending a to as is denoted by $(a{:}as)$. The notation $\wp(D)$ represents the set of subsets of the set D. Given a domain, u, and a sequence of actions, α, $\mathit{sources}(\alpha, u)$ is the set of domains that might leak information to u when α executes.

Definition 2. *The function, sources, which determines via what domains information can flow to a particular domain u, is defined as follows:*

$$\begin{array}{lcl}
\mathit{sources} & :: & A^* \times D \to \wp(D) \\
\mathit{sources}([],\ u) & = & \{u\} \\
\mathit{sources}(a{:}as,\ u) & = & \begin{cases} \mathit{sources}(as, u) \cup \{\mathit{dom}(a)\} \\ \quad \mathit{if}\ \exists v.\ v \in \mathit{sources}(as, u) \wedge \mathit{dom}(a) \rightsquigarrow v \\ \mathit{sources}(as, u) \\ \quad \mathit{otherwise} \end{cases}
\end{array}$$

Given a non-empty sequence of actions, $(a{:}as)$, sources checks whether an action, a, interferes with an action in $\mathit{sources}(as, u)$. This condition is true if an intervening action in as allows information to flow from $\mathit{dom}(a)$ to u, or if the security policy allows direct interference between $\mathit{dom}(a)$ and u. □

Security under a noninterference policy means that actions not allowed to interfere with a domain, u, are truly unobservable to u. Rushby captures this interpretation of security by simulating system execution. Executing a sequence of actions should produce the same results, from the perspective of u, as executing the same sequence with non-interfering actions removed.

The function *ipurge* removes all actions that do not interfere with a particular domain from a sequence of actions. The purge process relies on *sources* to analyze whether each action in the sequence may leak information to the domain in question.

Definition 3. *We define a function, ipurge, which, given a list of actions, α, and a domain, u, returns a list containing the actions in α that interfere with u.*

$$\begin{array}{lcl} \mathit{ipurge} & :: & A^* \times D \rightarrow A^* \\ \mathit{ipurge}([], u) & = & [] \\ \mathit{ipurge}(a : as, u) & = & \begin{cases} a{:}\mathit{ipurge}(as, u) & \mathit{if}\ \mathit{dom}(a) \in \mathit{sources}(a{:}as, u) \\ \mathit{ipurge}(as, u) & \mathit{otherwise} \end{cases} \end{array}$$

When $\alpha = a{:}as$, ipurge uses sources to determine if a interferes with u. If not, a is purged. □

Rushby models the execution of a sequence of actions with *run*, which is a natural extension of *step* from Section 5.1.1.

$$\begin{array}{lcl} \mathit{run} & :: & \mathrm{S} \times A^* \rightarrow \mathrm{S} \\ \mathit{run}(s, []) & = & \mathrm{s} \\ \mathit{run}(s, (a{:}as)) & = & \mathit{run}(\mathit{step}(s,a), as) \end{array}$$

We often wish to run a sequence of actions from the initial state, s_0, or to extract the output produced by an action following such an execution. The functions *do* and *test* are shorthands for these two operations.

$$\begin{array}{lcl} \mathit{do} & :: & A^* \rightarrow S \\ \mathit{do}(\alpha) & = & \mathit{run}(s_0, \alpha) \end{array}$$

$$\begin{array}{lcl} \mathit{test} & :: & A^* \times \mathrm{A} \rightarrow O \\ \mathit{test}(\alpha, a) & = & \mathit{output}(\mathit{do}(\alpha), a) \end{array}$$

We now use these functions to state Rushby's formulation of system security:

Definition 4 (Security Property)**.** *Let s be the state that results from running an arbitrary sequence of actions, α, from the initial state, and let t be the state that results from running the corresponding purged list. A system is secure if the output produced by executing any action is the same in s and t.*

$$\mathit{test}(\alpha, a) = \mathit{test}(\mathit{ipurge}(\alpha, \mathit{dom}(a)), a) \quad \square$$

5.1.3 Establishing Security

Rushby employs a technique called unwinding to prove that a system satisfies the security property. Many noninterference formulations use this approach [7, 33, 93]. Unwinding establishes that a system with well-behaved state transitions is secure. A set of *unwinding conditions* specify the expected properties of state transitions, and the *unwinding theorem* links these conditions to the security property.

Rushby specifies three unwinding conditions—output consistency, local respect, and weak step consistency—and proves that the security property holds for systems that satisfy these conditions.

Definition 5 (Output Consistency). *A system is* output consistent *if the output produced by executing an action in equivalent states is the same in both states.*

$$s \overset{dom(a)}{\sim} t \;\Rightarrow\; output(s, a) = output(t, a)$$

□

Definition 6 (Local Respect). *A system* locally respects *the security policy if the effect of an action, a, which may not interfere with a domain, u, is unobservable to u. That is, the state produced by executing a is equivalent, from u's perspective, to the state before a executed.*

$$dom(a) \not\rightarrowtail u \;\Rightarrow\; s \overset{u}{\sim} step(s, a)$$

□

Definition 7 (Weak Step Consistency). *A system is* weakly step consistent *if the states that result from executing an action in equivalent states are equivalent.*

$$s \overset{u}{\sim} t \wedge s \overset{dom(a)}{\sim} t \;\Rightarrow\; step(s, a) \overset{u}{\sim} step(t, a)$$

□

Theorem 1 (Unwinding)**.** *Let $\rightsquigarrow$ be a policy and M a view-partitioned system that is*

1. *output consistent,*
2. *weakly step consistent, and*
3. *locally respects $\rightsquigarrow$.*

Then M is secure for $\rightsquigarrow$.

Rushby proves the unwinding theorem using properties of *sources* and $\approx$, the details of which are in his paper [82].

5.2 NOTATION

We use Haskell syntax for the definitions and specifications presented in the remainder of this chapter. We choose Haskell as the specification language in place of a traditional mathematical notation to enable us to type check the model throughout the development process. This approach is less formal than developing our model of H directly in the language of a theorem prover. When instantiating the Rushby framework using Haskell, we model the system state as a datatype. We also model the set of domains, D, the set of actions, A, and the set of outputs, O, as types. The security policy is a relation between domains that we implement as a function. All functions in the formalism are encoded as Haskell functions in our instantiation.

Our model is not executable, but in our experience type checking alone was still useful for catching bugs. Because we use types to represent the essential features of the Rushby framework, the compiler acts as a consistency checker on the definitions. Using a programming notation also helps us to avoid overlooking the behavior of cases that might be abstracted over in a purely mathematical definition. For example, the Haskell compiler provides warnings for incomplete pattern matches, which indicates that we might have forgotten to insert the behavior of an operation in one or more cases. Every property and specification that we present in this chapter has been type checked to establish a minimum level of consistency. Since developing the Haskell model, we have also translated the code into HOL Light [43]. The translation was very direct and did not require any nontrivial modifications to the specifications, giving us a greater degree of confidence in the model. We present the Haskell model here because the reader will already be familiar with the notation from our review in Chapter 3.

In some cases, aspects of the specification might be uncomputable or might be semantically unclear when viewed through a Haskell lens. In particular, we use mathematical constructs that are not defined in Haskell so that we can easily express the appropriate properties in our specifications. For example, we introduce the function `forall` of type `(a -> Bool) -> Bool` to express the property that a particular predicate holds of every member of a type `a`. Throughout this chapter, we will use the convention that

```
forall (\arg -> -- :: Type
  p arg)
```

is equivalent to

$$\forall arg \in \mathit{Type}.\ p\ arg$$

even though we do not provide a definition for the Haskell function `forall` in our model. The specifications throughout this chapter also use an `exists` function of type `(a -> Bool) -> Bool` that is an analog to the mathematical construct ∃. Our uses of `exists` follows a similar notational convention to our uses of `forall`.

5.3 MACHINE MODEL

As a starting point for our Rushby instantiation, we develop an abstract model of IA32 hardware that is sufficiently expressive to describe the aspects of memory management that are relevant for describing execution environment and address-space integrity. The machine model serves as the state component of the Rushby framework, and will ultimately help us to provide meaning for the memory-safety property of our system. The representational choices that we make in the model are important because they impact the aspects of system behavior that our safety property will be able to capture. Any concept that we hope to describe with a property in the remainder of the dissertation must have appropriate hooks in the state. As such, our state will include high-level components that reflect the internal state of H as well as low-level components that reflect the state of the machine. If we do not capture enough details about the machine and H's internal state, then we will not be able to express the appropriate properties of our implementation in our specifications. However, we also wish to avoid clouding the model with extraneous information; if we include too many details about the machine that are not directly related to our notion of memory-safety, then our specification will become difficult to understand and to reason about.

Our ultimate goal is to capture the execution environment and address-space integrity properties presented at the beginning of this chapter, so we use these properties as the driving force guiding which concepts to include in our machine model. Capturing these properties requires a fairly detailed model of the structure of page-tables and page-directories so that we can reflect what happens to these pages during updates. We must also distinguish between memory pages used for different purposes, because we cannot protect environment and page-map pages if we cannot identify them. We ignore many other details of the machine that do not directly affect the interactions between the domains of the system (the client operating system, user programs, H, and the run-time system) and the protected classes of memory. For example, the dirty bit of a page describes whether or not that page has been written to and therefore provides information about the state and history of that page. This information is essential for a high fidelity machine model, and is potentially important for a model that aims to capture confidentiality properties about information flow between domains. However, the value of this bit, and even its existence, does not affect the integrity of the Haskell run-time system. For this reason we leave dirty bits out of our memory model, along with many other details of the machine that are not related to the properties at hand.

The rest of this section introduces the components that are necessary for our model. Section 5.3.1 describes our representation of virtual and physical memory, including the virtual-address translation hardware. Section 5.3.2 presents a higher level view of memory in the form of physical and virtual memory regions. Section 5.3.3 combines the fundamental concepts of the model into a state type that will form the basis of our formalization.

5.3.1 Virtual and Physical Memory

As in the design of the interface, we associate a dynamic status with each page of physical memory that reflects its current usage as an environment page, as a page-directory page, as a page-table page, or as a normal page of client-controlled memory. In the design chapter (Chapter 4) we explored the idea that the set of operations available on different types of page are different. In the model, we go one step further and distinguish the type of data stored in each page to capture the semantics that H associates with each of these page types.

Table 5.1 catalogs the primitive types that we use to model physical and virtual addresses. We build on these primitive types and introduce a datatype to represent memory pages called `Page`. We model the `Status` type described in Section 4.1 by encoding each possible status value as a constructor for the `Page` type. Here, we

Name	Width (bits)	Description
`SuperpageAddress`	10	Address aligned to a 4 MB page boundary. Corresponds to the top 10 bits of an address.
`SuperpageOffset`	10	Offset into a 4 MB page (identifies a 4 KB page).
`PhysicalPageAddress`	20	Address aligned to a 4 KB physical page. Produced by combining a `SuperpageAddress` with a `SuperpageOffset`.
`PageDirectoryIndex`	10	Index into a page-directory. Corresponds to the top 10 bits of a virtual address.
`PageTableIndex`	10	Index into a page-table. Corresponds to the ten bits below the page-directory index.
`VirtualPageAddress`	20	Address aligned to a 4 KB virtual page. Produced by combing a `PageDirectoryIndex` with a `PageTableIndex`.
`Offset`	12	Offset into a 4 KB page. Corresponds to the low 12 bits of a virtual or physical address.
`PhysicalAddress`	32	Address in physical memory. Produced by combining a `PhysicalPageAddress` with an `Offset`.
`VirtualAddress`	32	Address in virtual memory. Produced by combing a `VirtualPageAddress` with an `Offset`.
`RegionLength`	32	Length of a region of memory.

Table 5.1: The primitive types of the model. We break virtual and physical addresses into these fine-grained components to capture the meaning of each element of an address in the types. This approach provides better documentation and reflects the alignment constraints of various operations in the model.

identify five possible status values for pages: environment, normal, page-directory, page-table, and uninstalled. The addition of an uninstalled status value reflects the fact that it is possible to address physical pages that are not available on the machine; this possibility must be handled explicitly in the formal model for completeness. Each constructor has a single argument that describes the data currently contained in the page, except for uninstalled pages which contain no data.

```
data Page = Environment PageData
          | Normal PageData
          | PageDirectory DirectoryPageData
          | PageTable TablePageData
          | NotInstalled
```

The use of different types for the contents of page directories, page tables, and environment/normal pages reflects the semantic distinction between these pages from the perspective of H. In this chapter, we will use the following predicates, defined in the obvious way, to test the status of specific pages.

```
isEnvironment, isNormal, isPageDirectory, isPageTable,
  isInstalled  :: Page -> Bool
```

We use tests like these in the specification to describe conditions that must hold for the set of pages with a particular status, especially in the well-formedness definitions of Section 5.6.

Both environment pages and normal pages store generic data as far as H is concerned. H will never write to or read from locations in these pages. We model the contents of these pages so that we can model the execution of the full system: the Haskell run-time system reads from and writes to the environment pages; a kernel reads from and writes to normal pages that are mapped into kernel-space; and user-level programs read from and write to normal pages that are mapped into user-space. Because there are no special semantics associated with the data of these pages, we model their contents as bytes and a complete page as a function from `Offset`s to bytes.

```
type PageData = Offset -> Word8
```

The semantics of the data contained in page-directories and page-tables is specified by the IA32 architecture. We model page-directory data as a function from a page-directory index (the top ten bits of a virtual address) to a page-directory entry (called `DirectoryContents`). Not all page-directory indexes contain an entry—we explicitly represent this partiality with a `Maybe` type.

```
type DirectoryPageData = PageDirectoryIndex -> Maybe DirectoryContents
```

A page-directory entry is either a superpage (for entries that use a 4 MB page) or a page-table (for entries that use 4 KB pages). A superpage entry contains the physical address of the 4 MB page mapped by the entry. A page-table entry contains one of these physical page addresses that points to the appropriate page-table. We represent 4 KB-aligned `PhysicalPageAddress`es as a pair containing a `SuperpageAddress` and a `SuperpageOffset`.

```
data DirectoryContents = Superpage SuperpageAddress
                       | Table PhysicalPageAddress

type PhysicalPageAddress = (SuperpageAddress,SuperpageOffset)
```

We model page-table data as a function from page-table indexes to page-table entries. A page-table entry is simply the physical address of the 4 KB page mapped by the entry.

```
type TablePageData = PageTableIndex -> Maybe PhysicalPageAddress
```

As with page-directories, the possibility for unmapped pages is captured with a `Maybe` type.

We ignore permissions on page-directory and page-table entries. In our experience, read/write permissions complicate the formalization of properties that relate to virtual-to-physical address translation and are not necessary for capturing our notion of memory-safety. Correct configuration of kernel mappings using the user/supervisor bit is essential for defining the memory-safety property—because the H implementation enforces separation between the run-time system and the user-programs by configuring the run-time system pages to be accessible only in kernel-mode—but we model this separation without directly encoding permissions in the state by using well-formedness conditions (see Section 5.6). Any errors in the configuration or specification will be caught through a violation of the top-level security property, which assumes that every action produces a well-formed state, rather than through a permissions check. We will examine the security property in Section 5.5.

The page-directory and page-table indexes that serve as offsets into pages of those types come from our representation of page-aligned virtual addresses as a page-directory index paired with a page-table index, matching the semantics given to virtual addresses by the hardware (see Section 2.2).

```
type VirtualPageAddress = (PageDirectoryIndex, PageTableIndex)
```

This is similar to the representation of `PhysicalPageAddress` described previously as a superpage address paired with a superpage offset.

We model supervisor-only mappings by partitioning the virtual address-space into a user-space component and a kernel-space component. Mappings that reside in kernel-space are treated as though they are not accessible to user programs, even though we do not explicitly model the distinction between user and supervisor permission. User-space mappings are accessible to anyone. We keep the nature of the user-kernel boundary abstract, but provide functions for testing which space an address lies in.

```
isUserAddress, isKernelAddress :: VirtualAddress -> Bool
isUserPageAddress, isKernelPageAddress :: VirtualPageAddress -> Bool
```

There are space testing functions for both page-aligned and non-aligned virtual addresses. Certain operations of the model, such as adding a new user mapping, are only valid for addresses in one portion of the address-space.

5.3.2 Memory Regions

As in the H design, regions of memory play an important role in our model. A physical region is any contiguous range of physical pages. This simple notion of regions leads to a simple representation as the address of the first page in the region paired with the length of the region (in pages).

```
data PhysicalRegion = PhysicalRegion {
  physRegionStart  :: PhysicalPageAddress,
  physRegionLength :: RegionLength
}
```

Our representation of physical regions in the model is more general than the representation presented in the design chapter (see Section 4.3.1) because we allow any grouping of contiguous physical pages. The flexpage constraints are convenient when implementing operations on physical regions, but they are not essential to the safety of the interface. The only restriction we place on physical regions in the model is that the entire region must lie in available memory. A more refined model could add the flexpage constraints to the `PhysicalRegion` type to more closely approximate the level of the implementation.

Though our representation of regions is simple, it is not the best fit for some of the properties that we would like to write about regions. For example, quantification over the pages of a region occurs frequently, which might make a list or set operation more ideal. We avoid these representations because they make it harder to ensure that the pages of a given region are contiguous in physical memory. Instead, we define functions that map our representation onto structures that make property formulation easier. For example, `toListPhysicalRegion` enumerates the pages contained in a particular region.

```
toListPhysicalRegion :: PhysicalRegion -> [PhysicalPageAddress]
```

`memberPhysicalRegion` determines whether a particular page is contained by a given physical region.

```
memberPhysicalRegion :: PhysicalPageAddress -> PhysicalRegion -> Bool
```

Another useful function on regions calculates whether one region is a subregion of another.

```
isPhysicalSubregion :: PhysicalRegion -> PhysicalRegion -> Bool
```

We track the initial set of physical regions (the regions that are returned by a call to the interface primitive `initialRegions`) and the set of all active region handles (the initial regions plus those that have been produced by a call to the interface function `deriveRegion`) in a type called `RegionState`. This approach differs from our implementation, where we use an abstract datatype to represent physical regions and therefore do not need to track the set of active handles. We choose to track set of active handles in the state of our model because we can quantify over this set as a convenient mechanism for expressing properties about the memory that is pointed to by region handles, for example, that a `PhysicalRegion` never includes any environment pages. We will explore these properties in Section 5.6.

```
data RegionState = RegionState {
  initialRegions :: Set PhysicalRegion,
  allRegions :: Set PhysicalRegion
}
```

The initial regions and the set of all regions are tracked separately because H provides a function for querying the set of initial regions at any time during system execution.

Virtual regions are handled in the same fashion as physical regions and describe contiguous ranges of virtual pages.

```
data VirtualRegion = VirtualRegion {
  virtRegionStart  :: VirtualPageAddress,
  virtRegionLength :: RegionLength -- in pages
}
```

As with physical regions, we define utility functions on virtual regions to make working with them easier: `toListVirtualRegion` converts a virtual region into the list of pages that the region contains and `memberVirtualRegion` tests if a page belongs to a particular virtual region.

```
toListVirtualRegion :: VirtualRegion -> [VirtualPageAddress]
memberVirtualRegion :: VirtualPageAddress -> VirtualRegion -> Bool
```

Virtual regions must belong to either the user portion of the address space or the kernel portion of the address space. We disallow regions that span both spaces or that overflow. The predicates `isUserRegion` and `isKernelRegion` indicate which portion of the address space a region describes.

```
isUserRegion, isKernelRegion :: VirtualRegion -> Bool
```

The virtual region must be entirely contained in the appropriate portion of the address space for the corresponding predicate to return `True`. Thus, there is no region for which both `isUserRegion` and `isKernelRegion` are both true but there may be regions for which neither predicate holds.

5.3.3 Machine State

In our model, the state tracks the current page-directory (the CR3 register), the reference page-directory, the status and contents of each page available on the machine, and the current set of region handles. The reference page-directory is a special page-directory that contains all of the kernel-mappings, including the mappings for the Haskell run-time system. We use the reference page-directory to represent kernel-only mappings in lieu of an explicit representation of the user/supervisor bit. Any page that is mapped in the reference page-directory is treated as though it were mapped with kernel-only permission in the hardware. We also use the reference page-directory as an implementation technique for ensuring that the run-time system, H, and the kernel have the same view of memory in every address-space (see Section 6.7). Modeling the reference page-directory in the state allows us to formalize the property that every address space has a consistent view of kernel-space.

```
data State = State {
  cr3       :: PageDirectory,
  reference :: PageDirectory,
  status    :: PhysicalPageAddress -> Page,
  regions   :: RegionState
}

type PageDirectory = PhysicalPageAddress
```

We represent CR3 and the reference page-directory as physical address pointers to page-directories. We introduce a type synonym called `PageDirectory` for documentation. The page data is stored in a `status` component that maps physical page addresses to the information about the corresponding physical pages. The region handles are contained in a `RegionState` structure.

Not all values of the state type are acceptable. Some possible values of this type might describe states that are unsafe or that H should never allow the system to enter. In Section 5.6 we will formulate a series of well-formedness constraints that describe the essential properties we expect of a memory-safe state. For example, the set of region handles should never provide access to environment pages. In a safe system, these constraints will hold on the initial state (s_0 is well-formed), and every action in the system will produce a well-formed state given a well-formed input state. This inductive argument will help us to demonstrate safety for arbitrary executions of the system using only properties of the local transitions from state to state. We will define a single predicate on states called *wellFormed* that encompasses all of the well-formedness constraints of our model. In terms of this predicate, the inductive argument corresponds to the following properties.

$$\mathit{wellFormed}\ s_0$$

$$\forall s \in \mathit{State}, \forall \mathit{op} \in A.\mathit{wellFormed}\ s \Rightarrow \mathit{wellFormed}\ \mathit{step}(s,\ \mathit{op})$$

Recall that `A` is the set of actions in the Rushby framework. We will examine the specific nature of these actions for our system model in Section 5.4.1.

Example: Address Translation The state associated with page-directory pages and page-table pages represents a mapping in memory that the hardware uses to perform virtual-to-physical address translations. The constructs in our model provide a basis for defining this translation function. We model address translation in hardware with two functions: `translatePage` computes the physical page address that a virtual page address maps to in any page-directory and `translation` translates a virtual-address to a physical one in the current page-directory (CR3). In both cases, the result is a `Maybe` value because of the potential that the requested virtual address is not mapped.

`translatePage` is a general function that may be used to examine and compare the mappings in any page directory. This function walks the hardware tables, first examining the page-directory entry for the virtual-address and then (if necessary), examining the appropriate page-table. We write `translatePage` using do-notation in the `Maybe` monad (see Section 3.4) to handle failures within the definition concisely (including pattern-match failures).

```
translatePage :: State -> PageDirectory -> VirtualPageAddress
  -> Maybe PhysicalPageAddress
```

```
translatePage s pd (pdi,pti)
  = do PageDirectory dirpage <- return (status s pd)
       pde <- dirpage pdi
       case pde of
         Table ppa -> do PageTable tablepage <- return (status s ppa)
                         tablepage pti
         Superpage spa -> return (spa, toSuperpageOffset pti)
```

There are several places where the computation might fail because the page being translated is not mapped or because the data structures do not contain well-formed data. For example, the status of the page-directory being traversed must be `PageDirectory`. Some potential errors—like a page-directory entry that contains a `Table` but does not point to a page with the `PageTable` status—represent a misconfiguration of the system state and we would like to ensure that this never happens in our implementation. Ensuring that every page pointed to by a `Table` entry in a page-directory is a page-table in the current state is exactly the kind of property that we will capture with well-formedness constraints in Section 5.6.

Translating a virtual address to a physical one in the current page-directory may be accomplished with a straightforward invocation of `translatePage` on the page-directory currently installed in `cr3`.

```
translate :: State -> VirtualAddress -> Maybe PhysicalAddress
translate s (vpa, off) = do ppa <- translatePage s (cr3 s) vpa
                            return (ppa, off)
```

Testing whether or not a particular page is mapped in a particular page-directory is accomplished by testing if `translatePage` returns a value constructed with `Just`.

```
mappedPage :: State -> PageDirectory -> VirtualPageAddress -> Bool
mappedPage s pd vpa = isJust (translatePage s pd vpa)
```

Typically we only care if a page is unmapped in all page-directories, rather than in some particular page-directory. The function `unmappedPage` is true when no page-directory maps a given physical page.

```
unmappedPage :: State -> PhysicalPageAddress -> Bool
unmappedPage s ppa
  = forall (\pd -> -- :: PhysicalPageAddress
      forall (\vpa -> -- :: VirtualPageAddress
        case translatePage s pd vpa of
          Nothing -> True
          Just ppa' -> ppa /= ppa'))
```

The wrapper `unmappedNormalPage` tests if a page has a normal status and is unmapped in all page-directories.

```
unmappedNormalPage :: State -> PhysicalPageAddress -> Bool
unmappedNormalPage s ppa = isNormalPage s ppa && unmappedPage s ppa
```

These functions help us to express properties of `translatePage` more concisely in our specification.

5.4 PROTECTION DOMAINS

In this section we continue the process of instantiating the Rushby formalism to fit our system. We focus on the essential components of protection domains: the set of domains in our system, the actions of those domains, and the portion of the system state that is accessible to each domain. We define the policy that governs interactions between domains in Section 5.5.

Protection domains are the unit of resource protection and they represent the actors in a Rushby-style system. Each domain has a portion of the global state that it may view—its resources—and a set of actions that it may perform—its executable instructions. The security policy describes the effect that these actions may have on the system state by specifying the domains that will be able to view the results of a particular domain's actions. For example, a separation kernel might be formalized by instantiating the set of domains to be the set of user processes with a security policy that states that no process may interfere with any other.

In our system, the domains correspond to the basic components of our system: the H interface; the client kernel; and the user programs. Each of these components has a different privilege level and therefore has the ability to perform different actions and to access different portions of the system state. In addition to these three domains, we model the execution environment as a domain so that we can capture the effect that the Haskell run-time system has on the system state during execution. This gives the following list of domains, with the described set of allowable behaviors:

E The environment domain. This domain is an abstract representation of the Haskell run-time system. The environment domain may arbitrarily write to any environment page and its view includes all of those pages. In this way, we do not assume any particular behavior of the run-time system, but allow the domain to observe any changes made to the environment memory during another domain's turn to execute.

H The H domain. This domain represents computations performed by the H interface. The H domain can view all of the protected system state, but should not write to environment pages. Otherwise, H has full control over the system state, such as the status of each memory page, the contents of page-map pages, and the set of memory handles. The actions available to the H domain include all of the operations of the H interface design.

K The client kernel domain. This domain represents computations performed by the client kernel. The client kernel domain can view the normal pages that have been mapped into kernel-space and may write to these pages.

U The user program domain. We group all user programs together because we are not concerned about interactions between individual user-level entities, just the impact that these entities have on the system state as a group. The user domain can view all of the normal pages that have been mapped into the user portion of the address space and can write to these pages.

We instantiate the set of domains, `D`, by defining a datatype with values that correspond to each of these protection domains.

```
data D = E | H | K | U
```

5.4.1 Domain Actions

Each domain may perform certain actions that transform the system state. Table 5.2 describes the set of actions available to each domain. The effects of an action are restricted to the portion of the system state that the associated domain is allowed to modify according to the security policy, which we will cover in Section 5.5. Note that this may be a subset of the portion of the state that the domain observes.

The actions of the H domain correspond to the operations available in the H API. Though we do not formalize every H operation, the actions and specifications correspond to a representative set of the H operations. We focus on the subset of functions in H that are critical for memory-safety and ignore the facilities for I/O, debugging, and module loading. The only operation that we do not include from the memory management interface is `modifyMapping` for modifying the permissions attached to a mapping, which is conceptually equivalent to removing the existing mapping and then adding it back with the new permissions. In our current work we do not model permissions, so there is no interesting behavior of `modifyMapping` to specify. A write operation is available to the environment, kernel, and user domains.

The set of actions, `A`, corresponds to a type with one constructor for each action listed in Table 5.2. The arguments to the constructor are the parameters to the action.

E	`WriteE va val`	Write value `val` to location `va`, if `va` is within the set of environment pages.
H	`DeriveRegionH pr ppa len`	Derive a new region with start address `ppa` and length `len` from the existing region handle `pr`.
	`AllocatePageDirectoryH ppa`	Convert the free normal page `ppa` into a page-directory page.
	`FreePageDirectoryH pd`	Free the page-directory page `pd`.
	`AddMappingH pd pts pr vr`	Add a user-space mapping from the virtual region `vr` to the physical region `pr` in the address space represented by the page-directory `pd`. Allocate page-tables from the list of free normal pages `pts` as needed.
	`RemoveMappingH pd vr`	Remove the mapping to `vr` in address-space `pd`.
	`AddKernelMappingH pr vr`	Add a kernel-space mapping from the virtual region `vr` to the physical region `pr`.
	`ExecuteH pd`	Change the currently active page-directory to `pd`.
K	`WriteK va val`	Write value `val` to location `va`, if `va` is within the set of normal pages mapped in kernel space.
U	`WriteU va val`	Write value `val` to location `va`, if `va` is within the set of normal pages mapped in user space.

Table 5.2: The actions of each protection domain. None of the actions have outputs because their effect is observed through the state. We include write operations on memory but not reads, because the ability to read is implicit in observation.

```
data A = WriteE VirtualAddress Word8
       | DeriveRegionH PhysicalRegion PhysicalPageAddress RegionLength
       | AllocatePageDirectoryH PhysicalPageAddress
       | FreePageDirectoryH PageDirectory
       | AddMappingH PageDirectory [PhysicalPageAddress]
           PhysicalRegion VirtualRegion
       | RemoveMappingH PageDirectory VirtualRegion
       | AddKernelMappingH PhysicalRegion VirtualRegion
       | ExecuteH PageDirectory
       | WriteK VirtualAddress Word8
       | WriteU VirtualAddress Word8
```

None of the actions have outputs—their effect is observed through the state. Thus, there is no need for an action like read, because the ability to read is implicit in observation. We instantiate the Rushby function that maps actions to their domain, called *dom*, in a straightforward manner.

```
dom :: A -> D
dom (WriteE _ _)               = E
dom (DeriveRegionH _ _ _)      = H
dom (AddMappingH _ _ _ _)      = H
dom (RemoveMappingH _ _)       = H
dom (AddKernelMappingH _ _)    = H
dom (AllocatePageDirectoryH _) = H
dom (FreePageDirectoryH _)     = H
dom (ExecuteH _)               = H
dom (WriteK _ _)               = K
dom (WriteU _ _)               = U
```

The naming scheme for actions makes the mapping obvious—each action belongs to the domain specified by the final letter of its name.

In Section 5.7 we will specify the appropriate behavior of each action in the Rushby model as a relation between before and after states. These specifications provide a connection between the abstract notion of memory-safety and the H implementation code. A correct implementation of an H operation must satisfy the corresponding specification. The link between a functional action of type `A` and the corresponding specification is provided by `actionSpec`:

```
actionSpec :: A -> (State -> State -> Bool)
actionSpec (WriteE va w)                  = writeE va w
actionSpec (DeriveRegionH pr ppa len)     = deriveRegionH pr ppa len
actionSpec (AllocatePageDirectoryH ppa) = allocatePageDirectoryH ppa
actionSpec (FreePageDirectoryH pd)        = freePageDirectoryH pd
actionSpec (AddMappingH pd pts pr vr)     = addMappingH pd pts pr vr
actionSpec (RemoveMappingH pd vr)         = removeMappingH pd vr
actionSpec (AddKernelMappingH pr vr)      = addKernelMappingH pr vr
actionSpec (ExecuteH pd)                  = executeH pd
actionSpec (WriteK va w)                  = writeK va w
actionSpec (WriteU va w)                  = writeU va w
```

Though `actionSpec` relates all values of type `State`, we do not consider all possible states to be valid. There are certain basic properties of states that must hold in order for memory-safety to be meaningful in an H-based system. A state that satisfies these properties is *well-formed*; we will define well-formedness precisely in Section 5.6. We integrate well-formedness into our model by projecting the `actionSpec` relation to well-formed states (as described in Section 5.8.1); within the relational specifications themselves we will therefore assume that the before and after states are well-formed.

View-Partitioning We capture the notion of observable state through a set of view types that represent the portion of the state that is accessible to each domain. There is a distinct view type for each domain: `EView` describes the state of the environment domain, `HView` describes the H domain state, `KView` describes the kernel domain state, and `UView` describes the user domain state. We also define a view function for each domain that extracts the portion of a particular state that is visible to that domain.

The environment domain observes the data of the environment pages. It implicitly relies on the virtual-to-physical mappings that allow environment page data to be read and written, even though the environment cannot observe the contents of page-directories and page-tables directly. A fundamental property of the environment domain is that the environment will always have the same pages available to it, no matter what address-space is currently active. Furthermore, no one may alter the mappings for the environment domain pages, for example, by remapping an environment page at a different virtual-address. By observing the environment page data, and implicitly observing the mappings that link virtual addresses to physical ones, the environment domain can also witness the existence of multiply mapped physical pages. For example, if the environment domain has access to two virtual-addresses that map to the same physical address, called v_1 and v_2, then the domain has the potential to notice that writes to v_1 change the data that is accessible through v_2.

We represent the view of the environment domain as a function from virtual page addresses to the observable information about pages. The use of virtual addresses captures the implicit reliance on virtual-to-physical mappings. We define a reference count function based on virtual addresses that computes how many pages are also mapped to the same underlying physical page. We use the type `Nat`, for natural numbers, to indicate that the reference count should not be negative. Using this function, we define the observable information about a particular environment page to be the contents of that page paired with the reference count for that page. There is no observable data for pages that do not have an environment status.

```
data EView = EView {
  eObservablePages :: VirtualPageAddress -> Maybe (PageData, Nat)
}
```

The view function of the environment domain finds the pages in the specified state with the `Environment` type and constructs an `EView` value where precisely these pages are observable. We assume the existence of a list containing the addresses of all available virtual page addresses, called `allVirtualPages`, for computing the reference count.

```
viewE :: State -> EView
viewE s = EView observable
  where observable vpa
          = do ppa <- translatePage s (cr3 s) vpa
               Environment contents <- return (status s ppa)
               return (contents, referenceCount s (cr3 s) ppa)

referenceCount ::
  State -> PhysicalPageAddress -> PhysicalPageAddress -> Nat
referenceCount s pd ppa
  = length [translatePage s pd vpa == Just ppa | vpa <- allVirtualPages]

allVirtualPages :: [VirtualPageAddress]
```

Pages that are not a member of the environment will never be projected in the observable list.

The H domain observes the entire system except for the contents of user and kernel data pages. Thus, the view of H contains the current page-directory, the status mapping for physical pages, the region handles, and the reference page-directory. The `pages` function returns `Nothing` for pages whose status is `Normal`. In this way, H can deduce that a physical address mapped to `Nothing` is `Normal` but it cannot see the contents.

```
data HView = HView {
               currentPdir     :: PageDirectory,
               pages           :: PhysicalPageAddress -> Maybe Page,
               hRegionHandles  :: RegionState,
               referencePdir   :: PageDirectory
             }
```

The view function of the H domain projects cr3, the region handles, and the reference page-directory from the specified state. The view also includes page descriptions for all physical pages whose status is not Normal.

```
viewH :: State -> HView
viewH s = HView (cr3 s) pages (regions s) (reference s)
  where pages ppa = case status s ppa of
                      Normal _  -> Nothing
                      page      -> Just page
```

Unlike other domains, H may observe the contents and status of pages in its observable set. The association of each page with a status is a private part of the H state, so other domains may only observe page contents.

The kernel observes the contents of kernel data pages and environment pages. Though the kernel should not modify the environment data, it might distinguish the state of the Haskell run-time system indirectly through its own execution. As with the environment domain, we represent the observation function as a virtual page mapping to capture the indirect effect of virtual-to-physical memory mappings. The observable data about a page includes its contents and its reference count. Region handles are also observable, because they are a tool for communicating information between H and the kernel.

```
data KView
  = KView {
      kObservablePages :: VirtualPageAddress -> Maybe (PageData, Nat),
      kRegionHandles   :: RegionState
    }
```

The view function of the client kernel projects the region handles from the specified state and locates all physical pages that either have the Environment status or have the Normal status and are mapped at a kernel address.

```
viewK :: State -> KView
viewK s = KView observable (regions s)
```

```
  where
    observable vpa
      = do ppa <- translatePage s (cr3 s) vpa
           let rcount = referenceCount s (cr3 s) ppa
           case status s ppa of
             Environment p -> return (p, rcount)
             Normal p | isKernelPageAddress vpa -> return (p, rcount)
             _ -> Nothing
```

The user domain observes normal pages of memory that are mapped in user space. Users cannot observe the contents of environment pages because the user programs are not writen in Haskell and therefore do not rely directly on the run-time system to execute[2]. The data observable about user pages is the contents and the reference count. For any pages outside of the observable set, the result will be `Nothing`.

```
data UView = UView {
  uObservablePages :: VirtualPageAddress -> Maybe (PageData, Nat)
}
```

The view function of the user domain projects those pages that have a `Normal` status in the specified state and are mapped in a user-space address.

```
viewU :: State -> UView
viewU s = UView observable
  where
    observable vpa
      = do ppa <- translatePage s (cr3 s) vpa
           case status s ppa of
```

[2] Even if the user processes were written in Haskell, they would need to execute in their own user-level run-time system to maintain adequate separation between the kernel and the user processes. Thus, we would never expect user-level programs to be able to observe the execution environment.

```
              Normal contents | isUserPageAddress vpa ->
                return (contents, referenceCount s (cr3 s) ppa)
              _ -> Nothing
```

To tie our notion of view types and projection functions into the Rushby framework, we bring the per-domain definitions together into a unified view type and view projection function.

```
data View = ViewE EView | ViewH HView | ViewK KView | ViewU UView

view :: D -> State -> View
view E s = ViewE (viewE s)
view H s = ViewH (viewH s)
view K s = ViewK (viewK s)
view U s = ViewU (viewU s)
```

By parameterizing over the domain being viewed, we abstract away from the specific domains of our system and create a generic operation that could be defined in any Rushby instantiation to capture observable state.

The `view` function is the mechanism that we use to define Rushby's view-partitioning relation on states, $\overset{u}{\sim}$. Recall that two states s and t are equivalent from the perspective of domain u if u cannot distinguish s and t through its view of the state.

```
viewEquiv :: D -> State -> State -> Bool
viewEquiv u s t = view u s == view u t
```

Note that the views of domains are not actually comparable in Haskell as written because they contain functions. We define a symbol to represent the comparison on functions so that we can type check our definitions and properties, but will never compute this value. The intuitive meaning behind the comparison is that functions within a view would produce the same output given the same input.

`view` also helps us to give meaning to the `output` function in the Rushby model. H-based systems do not perform output in a traditional sense—all of the effects of operations are confined to the state. Thus, to obtain the execution environment integrity and address-space integrity properties that we set out to prove at the beginning of the chapter, we treat the output produced by an action as the projection of the state visible by the domain of that action. Because outputs are simply views, we instantiate the output type `O` to `View`.

```
type O = View

output :: (State, A) -> O
output (s, a) = view (dom a) (step (s,a))
```

Recall from Section 5.1 that *step* is an abstract function in the Rushby framework that computes the state produced by executing an action `a` in a state `s`. We will define *step* in Section 5.8. The `output`s produced by running the same action in two states will be equal when the states observable to that domain after executing the action are equal. We will see when we discuss the policy of our system how this captures our desired memory-safety properties.

Treating an action's output as equivalent to the view of the action's domain has an interesting ramification: the output consistency unwinding condition of Rushby now reduces to a special case of the weak step consistency. We sketch the proof of this property of our instantiation in Section 5.8.

5.5 POLICY

At the beginning of this chapter we identified execution environment integrity and address-space integrity as two key isolation properties for an H-based system. Ultimately, these properties are simply a mechanism for realizing the intuitive notions of memory-safety that we expect from our system, for example, that H does not modify the Haskell heap directly even though H's implementation uses the potentially unsafe primitives that would allow it to do so. A security policy bridges the gap between resource-level isolation properties and the intuitive understanding of memory-safety by capturing the interactions that occur between domains via resources like memory. Essentially, the policy specifies the portion of the state that a particular domain may affect: a domain A may modify the state that is present in the view of any domain B where $A \rightsquigarrow B$. The absence of an interference relationship between two domains, written $A \not\rightsquigarrow B$, guarantees that B is unaffected by the execution of A (in a system that correctly upholds the policy). The security provided by a policy heavily depends on the system model used in the instantiation because the set of domains, the representation of the state, and the set of actions will influence the properties that can be captured by the policy. In this section, we will examine the considerations that go into the construction of a meaningful Rushby instantiation and the specific rationale behind some of our modeling choices (Section 5.5.1). We will also present the security policy for our system (Section 5.5.2).

5.5.1 Modeling Approach

To determine the appropriate security policy for our system, we must carefully consider the desired interactions between domains from a memory-safety perspective. As mentioned previously, we want the policy to capture the fact that H does not modify the Haskell heap. However, a memory-safe H implementation that does not modify the Haskell heap is useless if the H API exports operations that allow the client kernel or a user process to modify the heap directly or through some chain of events. Thus, we would like the policy to reflect that neither the kernel domain nor the user domain interferes with the environment domain. Along the same lines, H will not be able to enforce safety if its data gets corrupted, so we want a policy that protects H from the client and the user programs.

With these high-level memory-safety properties in mind, we can develop a security policy that will enforce the intended interactions between domains. We present the steps involved in creating a meaningful noninterference policy here, even though some of the steps describe aspects of the model that were already covered in earlier sections. The formulation of the security policy is tightly intertwined with the other concepts of the system model, so we present these steps as a summary of the Rushby instantiation process, and as a way to provide some intuition about the security policy we will develop throughout the rest of the section.

The first step in creating a policy is to identify the resources that will be described by the security property. Essentially, this corresponds to the resources that are modeled in the system state. In our case, the focus on memory-safety narrows our attention to resources that relate to memory management. In other systems, the resource of interest might be something like I/O ports or time. The second step is to determine the granularity of the operations and observations on that resource. The actions and views presented in Section 5.4 reflect our ideas about the appropriate granularity in an H-based system. In particular, we focus on memory pages because this is a natural fit with the operations of H (the H design was in turn motivated by the granularity of memory management operations on the hardware). A noninterference formulation of memory-safety for a programming language might look very different, for example, using variables or words as the granularity for memory operations. The final step in our Rushby instantiation is to determine a security policy that captures the intended interactions between domains. For example, we can express the fact that H cannot corrupt the run-time system by creating a policy where $H \not\rightsquigarrow E$.

In general, we must strike a very delicate balance when crafting our noninterference policy. The relationship between the set of domains, the observable state, and the set of actions can be difficult to manage. Some common pitfalls include:

- **Identifying Too Few Domains:** Identifying the right set of domains is crucial. Interference within a domain is impossible to capture except in very special circumstances, so any system components that are grouped together will be allowed to interact freely. For example, if we combined the H and environment domains into a single high-privilege domain, then we could not prove that H does not corrupt the Haskell heap under our security policy.

- **Inserting Too Few Policy Arrows:** The policy arrows must accurately reflect the observations that are possible given the system model (including the effects of each action and the representation of the state). If the policy does not accurately reflect the observable actions then we will not be able to prove that the system implements the policy.

- **Inserting Too Many Policy Arrows:** In a system where every domain interferes with every other domain, there is nothing interesting to be learned from a noninterference proof. The policy must express enough noninterference to capture the intended safety properties, like our $H \not\rightsquigarrow E$ example. There is a tension between this need and the requirement that the security policy accurately reflects the observable actions in the system. The set of actions must be carefully chosen to model just those domain interactions that contribute to the properties that we hope to capture.

We carefully chose the set of domains and the actions available to each domain to support our ability to define a policy that enforces our desired memory-safety policy. In this section, we will identify the interference relationships necessitated by these choices.

Before moving on to the actual policy, let us consider one final example that illustrates the tensions that arise when designing a system model for a noninterference instantiation. When modeling Haskell execution in our system, we chose to introduce an action in the environment domain, `WriteE`, that may change any location in the Haskell heap. This action is designed to represent the execution of a Haskell program on top of the run-time system, such as H or the client kernel. In this treatment of execution, we do not need to introduce any operation where a domain other than the environment makes changes to the state that are observable to the environment domain. Alternatively, we could model the relationship between the execution of Haskell programs and the changes that happen in the heap explicitly. Both H and the kernel would then include an action, `haskellStep`, that would nondeterministically change some portion of the heap to reflect allocation, collection, and computation. Aside from the fact that the alternative model is more complex, this would be a perfectly valid way to represent Haskell execution. The problem is that our policy would need to include the interference arrows $H \rightsquigarrow E$ (H interferes with the run-time environment) and $K \rightsquigarrow E$ (the client kernel interferes with the run-time environment) to accurately capture the observable actions of the system. With this model, we cannot prove that H does not corrupt the Haskell heap because we cannot distinguish legitimate modifications from illegitimate ones. The policy $H \rightsquigarrow E$ states that all interference from H to E is allowed, so an incorrect H implementation that corrupts the heap (and would cause a client kernel to crash) still satisfies the security policy. In noninterference models, it is essential that one only model the aspects of information flow that affect the desired security property, in our case memory-safety, so that the policy does not become so broad as to permit invalid system behaviors.

The actions of Section 5.4 are carefully crafted to support a policy that will enforce the high-level memory-safety properties that we are interested in while providing a good match for the design described in Chapter 4. Throughout the rest of the section, we will examine each domain and each action to determine the appropriate policy connections to other domains. Though the policy is guided by the actions in this sense, the actions were also guided by the desired policy during the process of designing the model.

5.5.2 Interference Between Domains

We build our policy by examining the semantics of the actions in each domain. Starting with the environment domain, there is only one action available to the domain—`writeE`—which modifies a value in an environment page. As such, any domain that can view environment page data can be affected by the execution of E. In the views presented in Section 5.4, both H and K observe the environment pages because these domains are written in Haskell (the language supported by E) and therefore directly depend on the state of the Haskell heap to execute. If we had an additional environment E' containing a Java run-time system, then any system component written in Java would depend on (and therefore observe) E'. The environment domain itself also observes the environment page data. In our system, all domains observe the state of their domain. Though this may seem obvious, there are meaningful secure systems where it is essential that the state a domain can observe is disjoint from the state the domain can write to. Collecting these policy constraints leads us to introduce three policy arrows for the environment domain:

$$E \rightsquigarrow E$$

$$E \rightsquigarrow H$$

$$E \rightsquigarrow K$$

Already, we have introduced a policy arrow that impacts our security argument. We originally said that only H may modify the page-directories and page-tables, and that we would like to capture this by demonstrating that no domains interfere with H. But this claim is too strong—the environment domain must interfere with H. However, we expect that the environment domain will interfere with H through a very limited interface, namely the environment pages, and will not write to any of the data pages owned by H. This is an assumption of our model and implementation—we trust the run-time system to execute correctly within its own memory area. By encoding the interactions between the run-time system and the other actors in the security policy, we have made the nature of this assumption explicit.

The H domain has many actions that we need to consider, because these are the actions that critically impact the memory-safety of the system. The operational specifications for these actions are complex (we will examine them in Section 5.7), but fortunately the memory-safety policy is simple. H may interfere with every domain except the environment.

$$H \rightsquigarrow H$$

$$H \rightsquigarrow K$$

$$H \rightsquigarrow U$$

H interferes with the kernel by modifying the virtual-to-physical memory mappings that the kernel may access and by extending the set of region handles available for memory mappings. H interferes with the user domain by changing the virtual-to-physical memory mappings in user-space. The bootstrapping code of H constructs the initial view of the environment, but after that point H may not change anything in the environment domain. We do not model the behavior of the bootstrapping code, but rather assume an initial state that occurs after the set up process completes.

The kernel domain does not have much power in our model. We purposefully leave its behavior and policies abstract. The only action available to the kernel is to write to kernel-mapped pages, `writeK`. If the underlying physical memory being written to is also mapped in a user accessible address, then the user domain will observe a kernel write. Thus, K may interfere with U.

$$K \rightsquigarrow K$$

$$K \rightsquigarrow U$$

We choose a model where K does not interfere with H. This comes from our representation of actions where each H API function is modeled as an action in the H domain that may occur at any time. This is similar to the treatment of Haskell execution. By modeling actions in this way, we do not need to introduce any actions in K that interfere with H. The only kernel action is a write function and certainly this should not modify the H data structures. We think that associating actions with the callee rather than the caller is a useful technique in general for modeling abstraction barriers in a noninterference framework. Without this technique, interference appears across abstraction boundaries in the policy and nothing can be learned about the system from the security property (a classic complaint about noninterference frameworks).

The user domain is just as abstract as the kernel domain. The user may write to user-mapped pages via the function `writeU`, which may be observable to the kernel domain if the same physical memory is mapped in kernel space.

$$U \rightsquigarrow K$$

$$U \rightsquigarrow U$$

Note that U does not interfere with H, even though, in the real implementation, the user domain can perform actions that change the hardware state in ways that are observable to H. For example, U might change the value of the dirty bit for a page by writing to that page. Such changes do not affect the integrity of the H data structures, so we do not need to include a policy arrow $U \rightsquigarrow H$ when modeling our memory-safety property. The model supports a proof of the unwinding conditions for this policy by abstracting away the portions of the state that are not relevant to memory-safety. Thus, if we were to model the dirty bits in the machine state, then the view of U would include these bits so that user actions could make modifications, but the view of H would not, to reflect the fact that H should not rely on the value of the dirty bits in any operations.

In the future it would be interesting to extend the techniques used for modeling memory-safety in H to examine process separation in a specific kernel, like seL4 [60]. Effectively reasoning about process safety in a realistic kernel will most likely require a dynamic security policy, rather than the static policy illustrated here, to account for domain creation and deletion. Our earlier work extends the Rushby framework to support the ability to formalize dynamic noninterference policies of this sort [66].

$$\begin{gathered} \mathit{policy} = \{E \rightsquigarrow E, E \rightsquigarrow H, E \rightsquigarrow K, \\ H \rightsquigarrow H, H \rightsquigarrow K, H \rightsquigarrow U, \\ K \rightsquigarrow K, K \rightsquigarrow U, \\ U \rightsquigarrow K, U \rightsquigarrow U\} \end{gathered}$$

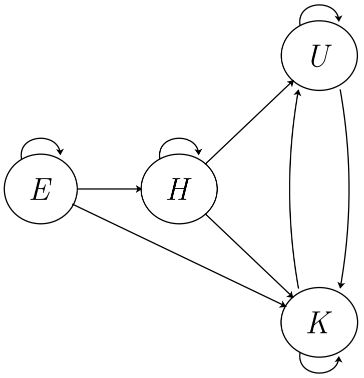


Figure 5.2: The security policy of our system. The environment interferes with H and the kernel because they depend on the environment to run. H interferes with the kernel and the users by the same reasoning. The kernel and the user domains interfere freely with each other in our model because we do not put any restrictions on their communication through memory. All domains interfere with themselves.

We bring the policies of each domain together in Figure 5.2 to illustrate the overall interactions between the domains. From the figure, we can see that no domain interferes with the environment domain (except itself) and that only the environment domain interferes with H. This policy matches our definition of memory-safety: the environment domain represents the environment pages and the H domain represents the page-table and page-directory pages; if these domains are not interfered with then the integrity of the corresponding pages will be maintained. Thus, if our system obeys the security policy, then we can conclude that our desired memory-safety property holds. By design, memory-safety is guaranteed by our system organization and the available set of actions. Unfortunately, the actions may be implemented in a way that introduces violations of the policy and thereby the memory-safety property. In Sections 5.6 and 5.7 we will explore the properties that our action implementations must satisfy in order to enforce the security policy.

5.6 WELL-FORMEDNESS

Well-formedness conditions codify the basic configuration properties that are necessary to support memory-safety in an H-based system. These conditions express what it means for a value of the state type to be valid. Many of the well-formedness constraints relate directly to the invariants presented in the design chapter (Chapter 4), while others reflect more fundamental assumptions about the machine model. For example, we require that the CR3 register always points to a valid page-directory. We connect well-formedness to memory-safety by specifying that every action in our system will produce a well-formed state (given a well-formed input state) if the operation is memory-safe.

We do not relate well-formedness to the unwinding conditions of the Rushby framework directly, but anticipate that well-formedness of states will be essential to any proof of the security property. The well-formedness conditions are such that we would not expect the domains to satisfy the security policy from Section 5.5 if the actions did not maintain well-formedness. Section 5.8 illustrates the use of well-formedness conditions for demonstrating aspects of the unwinding conditions.

5.6.1 CR3 and Reference Page-Directory Are Page-Directories

We define a well-formedness constraint on states to describe the property that the CR3 register, represented by the `cr3` state component, must point to a page-directory. The same property must also hold of the reference page-directory. These predicates are true if the page pointed to by the appropriate state component has a page-directory status.

```
cr3IsPageDirectory :: State -> Bool
cr3IsPageDirectory s = isPageDirectory (status s (cr3 s))

referenceIsPageDirectory :: State -> Bool
referenceIsPageDirectory s = isPageDirectory (status s (reference s))
```

The reference page-directory reflects a known good view of kernel-space that an implementation can use as a baseline when creating new page-directories. In the model, the reference directory helps us to express properties of the kernel-space mappings and the view of memory seen by privileged domains such as the Haskell run-time system and H. We define several well-formedness constraints that describe the characteristics of the reference page-directory and its relationship to the other page-directories in the system.

5.6.2 Reference Page-Directory Maps Kernel-Space Addresses

The first characteristic of the reference page-directory that we capture through well-formedness is the notion that this directory only maps kernel-space addresses. No user-space mappings may be present. The formulation uses the `translatePage` function from Section 5.3.3.

```
referenceMapsKernelAddresses :: State -> Bool
referenceMapsKernelAddresses s
  = forall (\vpa -> -- :: VirtualPageAddress
      isKernelPageAddress vpa
      || isNothing (translatePage s (reference s) vpa))
```

If any user virtual page addresses are mapped in the state, then this well-formedness condition is false, otherwise it is true.

5.6.3 Reference Page-Directory Maps Every Environment Page

The next important property of the reference directory is that it contains a mapping to every environment page in the system. According to the design presented in Chapter 4 and the operation specifications in Section 5.7, no operation will ever create a new environment page. If all environment pages are mapped in every state, then the set of pages that are observable to the environment domain will not change from state to state. This is important for memory-safety because our security policy states that no domain is allowed to interfere with the environment domain. If H (or a client kernel) could change the pages observable to the environment domain then this policy would not hold. Establishing that the environment pages are mapped in the reference directory is the first step towards ensuring the policy.

```
referenceContainsEnvironment :: State -> Bool
referenceContainsEnvironment s
  = forall (\ppa -> -- :: PhysicalPageAddress
      not (isEnvironment (status s ppa))
      || exists (\vpa -> -- :: VirtualPageAddress
                  case translatePage s (reference s) vpa of
                    Nothing   -> False
                    Just ppa' -> ppa == ppa'))
```

The formulation of `referenceContainsEnvironment` quantifies over each physical page address. For any environment page, there must exist a virtual page address whose translation in the reference page-directory equals the address of the environment page.

5.6.4 All Page-Directories Contain Reference Mappings

The previous property establishes that every environment page is mapped in the reference page-directory. If we can establish that every other page-directory in the system contains the same kernel-space mappings as this directory, then we can guarantee that there is a consistent view of kernel-space no matter which page-directory CR3 points to. From this we can deduce that the set of pages accessible to the environment domain does not change from state to state. We express this property in Haskell by quantifying over each kernel-space virtual page address in pages with a page-directory status. The translation of the virtual address in a given page-directory must match the translation of the same address in the reference directory.

```
pageDirectoriesContainReference :: State -> Bool
pageDirectoriesContainReference s
  = forall (\ppa -> -- :: PhysicalPageAddress
      case status s ppa of
        PageDirectory _ ->
          forall (\vpa -> -- :: VirtualPageAddress
            isUserPageAddress vpa ||
            (translatePage s (reference s) vpa
             == translatePage s ppa vpa))
        _ -> True)
```

This property ignores the translation of user-space addresses. None are mapped in the reference directory and we expect user-space mappings to be different in the other page-directories.

5.6.5 Environment Pages Are Only Mapped to Addresses That Are Mapped in the Reference Page-Directory

In addition to protecting the mappings of the environment domain, we need to ensure the integrity of the run-time system data. H has direct access to the environment memory through the unsafe operations of the FFI, but we trust H not to purposefully manipulate this memory. The operation specifications will help us to validate that H does not modify the environment data accidentally. Client kernels cannot modify the environment data because of the privilege restrictions imposed by H in software: H does not expose any operation that would allow a client to modify environment memory. For the user domain, we use hardware rings to enforce separation between user processes and the execution environment. The final property of the reference directory captures the requirement that all mappings to environment pages are present in the reference directory. Because the reference directory contains no user-space mappings, this is equivalent to saying that no environment page is ever reachable through a user-space address.

```
environmentOnlyInReference :: State -> Bool
environmentOnlyInReference s
  = forall (\pd -> -- :: PhysicalPageAddress
      forall (\vpa -> -- :: VirtualPageAddress
        case translatePage s pd vpa of
          Just ppa -> not (isEnvironment (status s ppa))
                      || mappedPage s (reference s) vpa
          Nothing -> True))
```

The `environmentOnlyInReference` property is false if any page-directory contains a mapping to an environment page that is not also present in the reference directory.

5.6.6 Mapped Pages Are Available

Another dimension of our memory-safety property is the protection of page-directory and page-table contents. This aspect of memory-safety depends on the dynamic behavior of the H operations because, unlike environment pages, the set of pages with the page-table and page-directory status is not fixed. Still, there are fundamental properties that must hold of the translation table structures (which correspond to the `status` field of the state) in order for them to be well-formed. The first such property states that no virtual address maps to a physical address that is not installed on the machine. A table/directory entry must contain nothing or a mapping to available memory.

```
mappedPagesAreAvailable :: State -> Bool
mappedPagesAreAvailable s
  = forall (\pd -> -- :: PhysicalPageAddress
      case status s pd of
        PageDirectory _ ->
            forall (\vpa -> -- :: VirtualPageAddress
              case translatePage s pd vpa of
                Nothing  -> True
                Just ppa -> isInstalledPage s ppa)
        _ -> True)
```

We quantify over page-directories and virtual page addresses. The value of the property `mappedPagesAreAvailable` will be false if the translation function indicates there is a mapping installed but no such page exists.

5.6.7 Page-Table Pointers Are Page-Tables

We also define a well-formedness property that ensures that page-directory entries are consistent with the rest of the state. In particular, if a page-directory entry contains a page-table pointer, but the page pointed to by that address is not a page-table according to the status field of the state, then something is wrong.

Maybe an incorrect pointer was installed in the directory or maybe the status was not appropriately updated, but we will not necessarily be able to protect the contents of the page-table in such a state.

```
tablePointersArePageTables :: State -> Bool
tablePointersArePageTables s
  = forall (\ppa ->  -- :: PhysicalPageAddress
      forall (\pdi -> -- :: PageDirectoryIndex
        case status s ppa of
          PageDirectory pd ->
            case pd pdi of
              Just (Table ppa') -> isPageTable (status s ppa')
              _  -> True
          _ -> True))
```

We specify `tablePointersArePageTables` by quantifying over all possible page-directory entries (any index into a page with the page-directory status). The property is true if every entry that maps to a table entry points to a page with the page-table status.

5.6.8 Regions Are Consistent and Disjoint From Environment

The last component of the state is the set of region handles. The first well-formedness constraint on region handles simply provides consistency between the two components of the `RegionState`. Recall from Section 5.3.3 that we track the set of initial regions and the set of all regions. The initial regions must be a subset of the total set of regions.

```
initialRegionsAreRegions :: State -> Bool
initialRegionsAreRegions s
  = Set.isSubsetOf (initialRegions r) (allRegions r)
 where
   r = regions s
```

The region handles describe the memory that may be used by client kernels for their own purposes. This includes mapping memory to the user domain and mapping memory into the free portion of kernel-space with read-write access for the kernel domain. If the memory described by the regions included any environment pages, then it would be possible to violate the policy that no domains interfere with the environment. The property `environmentOnlyInReference` would be violated if the client mapped environment memory to the user domain, but we cannot avoid extra kernel-space mappings to the environment with the properties we have examined so far. Instead, we provide a strong guarantee that the kernel cannot access the environment memory by defining a well-formedness constraint stating that no region handle contains a reference to an environment page.

```
regionsAreNotEnvironment :: State -> Bool
regionsAreNotEnvironment s
  = not (any (existsRegion isEnvironmentPage)
             (Set.elems (allRegions (regions s))))
```

`isEnvironmentPage` is a predicate that tests the status of a physical address in a particular state. We define similar predicates on the other status types as well.

```
isEnvironmentPage :: State -> PhysicalPageAddress -> Bool
isEnvironmentPage s ppa = isEnvironment (status s ppa)

isNormalPage, isPageDirectoryPage, isPageTablePage, isInstalledPage
  :: State -> PhysicalPageAddress -> Bool
```

`regionsAreNotEnvironment` is true for states where the region handles do not contain a reference to any physical page with the status `Environment`.

5.6.9 Putting It All Together

We use these well-formedness constraints to express the idea that there are certain fundamental properties enforced by every operation in the system by constraining the allowed values of `State` to the well-formed set. We formulate a predicate on states called `wellFormed` that encompasses all of the well-formedness constraints on states.

```
wellFormed :: State -> Bool
wellFormed s = cr3IsPageDirectory s &&
               referenceIsPageDirectory s &&
               referenceMapsKernelAddresses s &&
               referenceContainsEnvironment s &&
               pageDirectoriesContainReference s &&
               environmentOnlyInReference s &&
               mappedPagesAreAvailable s &&
               tablePointersArePageTables s &&
               initialRegionsAreRegions s &&
               regionsAreNotEnvironment s
```

The well-formedness conditions capture our intuition for what properties will be necessary to prove the unwinding conditions described in Section 5.1. Most correspond to the informal invariants that we created during the process of designing and implementing H to enforce our notion of memory-safety. It is possible that the set of well-formedness conditions is not sufficient to prove the unwinding conditions, but the predicate can be extended as necessary.

5.7 CONNECTING THE MODEL AND IMPLEMENTATION

Well-formedness provides the baseline for a well-behaving system. The next step in assuring memory-safety is to specify the effects of each action of the model. Specifying the permitted behavior of each action will make the interactions between domains more concrete and will help us to gain confidence that the security policy holds between our domains. We present the action specifications as relations on well-formed states. We opted against a functional style that directly describes the transformation each action performs on states because we found that approach forced us to specify details that are not relevant for safety. Our goal is to present a specification that supports memory-safety and prevents memory leaks via the H operations with a minimal number of extraneous details. The specifications serve as a further instantiation of the actions presented in Section 5.4.1 and provide a foundation for future verification work.

A particular benefit of the relational style is that nondeterminism is built in. Given a particular before state, the relational specification of an action may hold for many different after states. This property of the relational model allows us to leave aspects of the system that are not relevant for safety out of the model. For example, when we specify the behavior of user program execution, we can ignore any changes that occur in user memory pages. The manner in which a user program changes the contents of these pages is irrelevant to our safety argument and, with a relational model, we can leave these contents unspecified. This is in stark contrast to a functional model where we must specify a single result state that describes the particular changes a user program makes to memory, even though we do not care about these values. Nondeterminism must be added to such a model explicitly, for example, by using an oracle to generate values, but the overall specification will be more complicated. Of course, the downside to the nondeterminism of the relational model is that the gap between the specification and the actual implementation (where actions are functions) is larger than with a functional model. We choose to start with the relational model because it is easy to capture the properties of interest, knowing full well that a refinement into a functional model will likely be an important stepping stone on the path to any full verification of H.

A further benefit of the relational style is that it separates the demonstration that the actions produce well-formed states from the demonstration that the actions refine their specification. Specifying the connection of the action specifications to the functions of our H implementation is straightforward; there is a single action specification, `op` that corresponds to each memory management operation, `opImp`, in H. We must show that, for every action `op` in the model, the implementation `opImp`

- preserves well-formed states
  $\forall s \in \mathit{State}.$ `wellFormed s` $\Rightarrow$ `wellFormed (opImp s)`

- implements the specification when the action is possible
  $\forall$`s` $\in \mathit{State}.$

      `wellFormed s` $\wedge$ ($\exists$`s'. actionSpec op s s'`) $\Rightarrow$

      `actionSpec op s (opImp s)`

- and preserves the system state when the action is invalid
  $\forall$`s` $\in \mathit{State}.$

      `wellFormed s` $\wedge$ ($\forall$`s'.` $\neg$`(actionSpec op s s')`) $\Rightarrow$

      `opImp s` $=$ `s`

An implementation that satisfies these properties is a valid instantiation for the action that corresponds to `op` in our Rushby model. Proving that our implementation satisfies these properties is future work. Note that well-formedness alone is not sufficient for memory-safety because some aspects of a correct implementation rely on the dynamic behavior of the operation.

5.7.1 Specification of WriteE

The first action in our model is the `WriteE` operation of the environment domain. `WriteE` has additional arguments of type `VirtualAddress` (the location to write) and `Word8` (the value to write). The intended behavior of `WriteE` is to modify a location in one of the pages owned by the environment domain. According to the security policy (see Section 5.5), the environment domain can interfere with itself and H, but should only write to pages with the environment status. Thus, in our specification, writeE is true when the contents of a single byte in a single environment page change. writeE is false if the status of any page changes or if the contents of any non-environment page change.

```
writeE :: VirtualAddress -> Word8 -> State -> State -> Bool
writeE va val state state'
  = cr3 state == cr3 state'
    && reference state == reference state'
    && regions state == regions state'
    && case translate state va of
         Nothing -> False
         Just (ppa,off) ->
           forall (\someppa -> -- :: PhysicalPageAddress
             if someppa /= ppa
                then status state someppa == status state' someppa
                else case status state someppa of
                       Environment pdata ->
                         case status state' someppa of
                           Environment pdata' ->
                             pdata' == updatePageData off val pdata
                           _ -> False
                       _ -> False)
```

The first few conjuncts are frame rules stating that this operation does not change the cr3 or reference page-directory settings and that it does not change the set of region handles. The only state component that is relevant for the writeE operation is the status field. We will see similar frame rules in each of our specifications—each of our actions only modifies a portion of the state.

5.7.2 Specification of DeriveRegionH

The DeriveRegionH action creates a new region handle from an existing one. The additional arguments are the starting region (of type PhysicalRegion) and the parameters for the new region: a start address of type PhysicalPageAddress and a length of type RegionLength. The most important property of this action is that it maintains our inductive argument on regions: if the initial regions do not point to any environment memory then no region we create will either. We specify this by requiring that the starting region is a member of the existing handle set and that the new region being created is a subregion of the starting one. The set of all regions in the after state must contain exactly one more region handle than the set of all regions in the before state, and this handle must match the region description given by the start and length arguments.

```
deriveRegionH :: PhysicalRegion -> PhysicalPageAddress -> RegionLength
  -> State -> State -> Bool
deriveRegionH pr ppa len state state'
  = cr3 state == cr3 state'
    && reference state == reference state'
    && status state == status state'
    && initialRegions (regions state) == initialRegions (regions state')
    && Set.member pr allOld
    && isPhysicalSubregion newRegion pr
    && Set.isSubsetOf allOld allNew
    && goodDerivedRegion (Set.toList (Set.difference allNew allOld))
  where
    allOld = allRegions (regions state)
    allNew = allRegions (regions state')
    newRegion = PhysicalRegion ppa len
    goodDerivedRegion [r] = r == newRegion
    goodDerivedRegion _   = False
```

The derive region action only modifies the set of all region handles; deriveRegionH will be false if any other state component changes.

5.7.3 Specification of AllocatePageDirectoryH

AllocatePageDirectoryH turns a physical page into a page-directory page. There are important safety issues at play in this operation, because the physical page could introduce a channel for the user domain or kernel domain to access page-directory contents. Page-directory pages are owned by the H domain, so this would violate the security policy. We prevent such a channel by specifying that the page being converted must not be mapped in any portion of any address space. The page must also have the normal status, because a page with any other status is currently in use by H or the environment. For `allocatePageDirectoryH` to be satisfied, the status of the newly created page-directory must match the reference page-directory (containing all of the same kernel-space mappings and no user-space mappings) and the status of all other pages must be unchanged.

```
allocatePageDirectoryH :: PhysicalPageAddress -> State -> State -> Bool
allocatePageDirectoryH ppa state state'
  = cr3 state == cr3 state'
    && reference state == reference state'
    && regions state == regions state'
    && unmappedNormalPage state ppa
    && forall (\someppa -> -- someppa :: PhysicalPageAddress
         if someppa /= ppa
            then status state someppa == status state' someppa
            else status state' someppa
                 == status state' (reference state'))
```

5.7.4 Specification of FreePageDirectoryH

`FreePageDirectoryH` is the essentially the inverse of `AllocatePageDirectoryH`: the action frees a page-directory page and converts it to an unmapped normal page. We specify `freePageDirectoryH` using `allocatePageDirectoryH` with the additional condition that the physical page must be zeroed in the after state. The page `zeroPageData` is a constant that maps every offset to zero.

```
freePageDirectoryH :: PageDirectory -> State -> State -> Bool
freePageDirectoryH pd state state'
  = allocatePageDirectoryH pd state' state
    && case status state' pd of
         Normal pagedata -> pagedata == zeroPageData
         _               -> False -- never happens

zeroPageData :: PageData
```

`freePageDirectoryH` requires that the page-directory does not contain any user mappings when it is freed. This is stronger than the requirements of the actual H implementation—we specify as part of the API that the page-directory should be empty of user mappings but allow the free to proceed in all cases. To make H match the specification, we would need to revise the type of the `freePageMap` operation to permit failure if the operation is invoked on a non-empty directory. This semantic dimension of free is not critical for memory-safety, but can introduce space leaks if any page-tables that are in use for mappings are not reclaimed.

5.7.5 Specification of AddMappingH

Adding a mapping to the user portion of the address-space is the most complicated action in our model. We split the specification into two parts: `addMapping` specifies most of the behavior of adding a mapping (in such a way that `removeMappingH` will be the inverse operation) and `addMappingH` supplements the `addMapping` specification with requirements that are specific to `AddMappingH`. The `AddMappingH` action adds a mapping from a virtual region to a physical region in a specified page-directory. When adding the mappings, H may need to allocate page-tables to support the translation of the addresses of the virtual region. As we saw in the design chapter (see Section 4.4.2), the client kernel provides the memory for these page-tables. We model this by including a parameter to the add mapping action that is a list of physical pages that may be converted into page-tables as needed.

This gives addMapping and addMappingH the following type:

```
addMapping, addMappingH :: PageDirectory -> [PhysicalPageAddress]
  -> PhysicalRegion -> VirtualRegion -> State -> State -> Bool
```

An important semantic difference between the two specifications is that addMapping expects a precise list of the set of page-tables that are needed for the operation while addMappingH expects a superset of the necessary page-tables.

Due to the large number of parameters with very specific semantics, much of the complexity of the specification stems from the parameter validation properties. The specification must handle the following conditions:

- The address specified as the page-directory to add the mappings to must actually be a page-directory in the before state.
- Each of the potential page-table pages must be a free page. This corresponds to having a normal status and not being mapped in any page-directory.
- None of the potential page-tables may lie within the physical region to be mapped.
- All of the pages within the physical region must be mappable, which corresponds to having the normal status.
- The virtual region and physical region parameters must be regions of the same length.
- The virtual region must entirely consist of user-space addresses (this action cannot add kernel-space mappings).
- None of the virtual addresses within the virtual region should be mapped—we do not allow over-mappings via this action.

These properties appear following the frame rules (AddMappingH does not change cr3, reference or regions) in addMapping.

```
addMapping pd pts pr vr state state'
  = cr3 state == cr3 state'
    && reference state == reference state'
    && regions state == regions state'
    && isPageDirectoryPage state pd
    && case status state pd of
         PageDirectory pdirdata -> True
         _ -> False
    && all (unmappedNormalPage state) pts
    && all (\ppa -> not (memberPhysicalRegion ppa pr)) pts
    && all (isNormalPage state) (toListPhysicalRegion pr)
    && physRegionLength pr == virtRegionLength vr
    && isUserRegion vr
    && forall (\somevpa -> -- :: VirtualPageAddress
         if memberVirtualRegion somevpa vr
            then translatePage state pd somevpa == Nothing
            else translatePage state pd somevpa
                 == translatePage state' pd somevpa)
```

The final conjunct, which specifies that the pages of the virtual region are not mapped, also specifies the relationship between the virtual-to-physical translation of the before state and the after state. For any page that is not in the virtual region, the translation must not be affected by the `AddMappingH` action.

The remainder of the specification is solely focused on the relationship between the before and after states. The fundamental property we expect as a result of adding a mapping is that, in the after state, each page address in the virtual region maps to the corresponding page address in the physical region.

```
&& and (zipWith isMappedTo (toListVirtualRegion vr)
                           (toListPhysicalRegion pr))
```

`isMappedTo` is a locally-defined utility that tests if the translation of a given virtual address matches a given physical address. Here, `pd` is the same as the page-directory argument of `addMapping`.

```
isMappedTo vpa ppa = translatePage state' pd vpa == Just ppa
```

Note that the relational style allows us to specify the observable change to the translation tables (that the addresses within the virtual region are now mapped to the appropriate physical addresses) without specifying the mechanism. For example, an implementation of this specification is free to use superpages or not. With a functional model we would have to make a commitment about what is ostensibly an implementation detail here.

The effect of `AddMappingH` on pages of memory that do not lie in the physical region being mapped should be limited in a memory-safe system. Pages in the set of new page-tables will have the page-table status in the new state and must be installed in a page-directory entry of the page-directory being modified. Environment pages and all normal pages that are not in the set of new page-tables do not change. We allow the contents of the page-directory being modified and of any page-table to change.

```
&& forall (\someppa -> -- :: PhysicalPageAddress
    if elem someppa pts
       then tableMappedInDirectory state' pd someppa
       else case (status state someppa, status state' someppa) of
              (Normal pagedata, Normal pagedata') ->
                pagedata == pagedata'
              (PageTable _, PageTable _) -> True
              (PageDirectory pdirdata, PageDirectory pdirdata') ->
                someppa == pd || pdirdata == pdirdata'
              (Environment pagedata, Environment pagedata') ->
                pagedata == pagedata'
              (NotInstalled, NotInstalled) -> True
              _ -> False)
```

Allowing any page-table to change is somewhat weak—there is a specific set of page-tables that we expect to change because they are part of the translation of the addresses in the virtual region. This set of page-tables is difficult to identify in an abstract fashion. The weaker property will not allow any safety violations because the translations implemented by the page-tables must be predictable (only the translations for the pages being mapped change), but the specification might allow some unexpected safe behaviors.

In addition to the parameter validation properties we discussed previously, `addMappingH` requires that the list of potential page-tables does not contain any duplicates and that the physical region being mapped is a member of the set of all regions. Note that the implementation does not need to check that the physical region provided to `AddMappingH` is a member of the valid region set because the Haskell type system ensures that only valid members of the `PhysicalRegion` type are passed as parameters to the H operation.

```
addMappingH :: PageDirectory -> [PhysicalPageAddress] -> PhysicalRegion
  -> VirtualRegion -> State -> State -> Bool
addMappingH pd pts pr vr state state'
  = length (nub pts) == length pts
    && Set.member pr (allRegions (regions state))
    && any isAddMappingTables (inits pts)
  where
    isAddMappingTables used = addMapping pd used pr vr state state'
```

We identify the precise set of page-tables needed for `addMapping` by specifying that `addMappingH` is true so long as some sublist of the potential page-table list (calculated using the function `inits`) satisfies the `addMapping` specification when combined with the other parameters.

5.7.6 Specification of RemoveMappingH

The reverse of adding a mapping is removing a mapping with `removeMappingH`. This operation is essentially the inverse of `addMapping`: it removes the mappings from the specified virtual region to some underlying physical region and frees some number of page-tables. We expect all of the preconditions of addMapping—for example, that the page-table pages are unmapped normal pages and that the virtual region does not map to any memory—to be the post-conditions of removeMappingH. The challenge in specifying the remove operation as the inverse of addMapping is that we must determine the underlying physical region and the list of page-tables that become free so that they may be supplied as arguments to the addMapping relation. These values can be calculated using the before and after states. We assume the existence of a list containing every value of the physical page address type, called allPhysicalPages, for use in computing the list of freed page-tables.

```
removeMappingH ::
  PageDirectory -> VirtualRegion -> State -> State -> Bool
removeMappingH pd vr state state'
  = case translatePage state pd (virtRegionStart vr) of
      Nothing -> False
      Just ppa -> let pr = PhysicalRegion ppa (virtRegionLength vr) in
                    addMapping pd pts pr vr state' state
  where
   pts = filter freedPageTable allPhysicalPages
   freedPageTable ppa
     = case (status state ppa, status state' ppa) of
         (PageTable _, Normal pagedata) -> pagedata == zeroPageData
         _  -> False

allPhysicalPages :: [PhysicalPageAddress]
```

The use of addMapping as opposed to addMappingH is important. We allow the remove operation to unmap regions of memory that do not correspond to physical regions in the set of regions handles. For example, if the region A is mapped using addMappingH and the adjacent region B is also mapped using addMappingH, we permit an invocation of removeMappingH that deletes the mapping to the region

A ∪ B in a single call, even if A ∪ B is not in the region handle set. This will not introduce safety issues because we will never unmap anything that was not mapped and the specification for adding a mapping ensures that all user mappings correspond to valid physical regions.

5.7.7 Specification of AddKernelMappingH

Adding a kernel mapping is conceptually similar to adding a user mapping, but is much simpler because we assume that the page-tables for kernel-space are preallocated. Thus, there are no potential page-tables to manage in the specification of AddKernelMappingH. We know that the view of kernel-space is consistent in every page-directory from our well-formedness conditions, so there is no need to specify a page-directory to update either. The remaining arguments are a physical region and a virtual region. We specify many of the same requirements for these parameters as we did for the parameters to addMappingH: the region lengths must be the same, the pages of the physical region must mappable, and the physical region must be in the set of region handles. For kernel mappings, the virtual region must contain only addresses that lie in kernel-space.

```
addKernelMappingH ::
  PhysicalRegion -> VirtualRegion -> State -> State -> Bool
addKernelMappingH pr vr state state'
  = cr3 state == cr3 state'
    && reference state == reference state'
    && regions state == regions state'
    && Set.member pr (allRegions (regions state))
    && all (isNormalPage state) (toListPhysicalRegion pr)
    && physRegionLength pr == virtRegionLength vr
    && isKernelRegion vr
```

The relationship between the before and after states is very similar to the relationship that we specified for user mappings. Only the virtual-to-physical translation for pages in the virtual region being mapped may change. We express this requirement as two properties. First we state that the virtual-to-physical translation for any address that does not belong to the virtual region will not change.

```
  && forall (\somevpa -> -- :: VirtualPageAddress
       memberVirtualRegion somevpa vr
       || translatePage state (reference state) somevpa
          == translatePage state' (reference state') somevpa)
```

Next, we state that the contents of physical memory stays the same, except for page-table pages, which might be modified to reflect the new virtual-to-physical mappings in kernel-space.

```
 && forall (\someppa -> -- :: PhysicalPageAddress
      if tableMappedInDirectory state (reference state) someppa
         then isPageTable (status state' someppa)
         else status state someppa == status state' someppa)
```

In the after state, each page address of the virtual region will be mapped to the corresponding page address in the physical region.

```
  && and (zipWith isMappedTo (toListVirtualRegion vr)
                             (toListPhysicalRegion pr))
where
  isMappedTo vpa ppa
    = translatePage state' (reference state') vpa == Just ppa
```

We describe the update to the kernel-space mappings in terms of the reference page-directory. The kernel-space mappings of every other page-directory must also be updated to match the reference page-directory in order for the action to produce a well-formed state.

5.7.8 Specification of ExecuteH

We deliberately model very little of the user domain's state and, in turn, very little of the behavior of H with respect to the user domain. Though the manner in which H saves and restores user registers is important for correctness, it does not affect our ability to protect the data structures of H or the run-time system. That is, incorrect management of user data does not impact memory-safety. We still model the H action for executing a user process, but the connection to the implementation is loose because we only specify the behavior that relates to the memory-safety critical portions of the state. In our model, the only observable effect of `ExecuteH` is that the current page-directory changes. The argument to `ExecuteH` is the address of the page to install as a page-directory. The value of `cr3` in the after state must equal this page.

```
executeH :: PageDirectory -> State -> State -> Bool
executeH pd state state'
  = reference state == reference state'
    && status state == status state'
    && regions state == regions state'
    && case status state pd of
         PageDirectory _ -> cr3 state' == pd
         _ -> False
```

We verify the status of the page-directory argument, but this is not strictly speaking necessary. Well-formedness already specifies that the value of `cr3` in the after state be a valid page-directory. We include the property for documentation and to help line up the specification with the mechanism for enforcing well-formedness in the implementation.

5.7.9 Specification of WriteK

The final actions of our model are the write operations for the kernel and user domains. The specifications for these operations are very similar. `WriteK` modifies a single location in the set of pages that the kernel domain may access. This set is equivalent to the set of pages with normal status that are mapped to a kernel-space address. `writeK` is true when a single value in a single kernel-space mapped normal page changes. `writeK` is false if the contents of any other page change or if the status of any page changes. We define `writeK` in terms of an abstract write predicate, `writeA`, that we will also use in the specification of `WriteU`.

```
writeA :: VirtualAddress -> Word8 -> State -> State -> Bool
writeA va val state state'
  = cr3 state == cr3 state'
    && reference state == reference state'
    && regions state == regions state'
    && case translate state va of
         Nothing -> False
         Just (ppa,off) ->
           forall (\someppa -> -- :: PhysicalPageAddress
             if someppa /= ppa
                then status state someppa == status state' someppa
                else case status state someppa of
                       Normal pdata ->
                         case status state' someppa of
                           Normal pagedata' ->
                             pdata' == updatePageData off val pdata
                           _ -> False
                       _ -> False)

writeK :: VirtualAddress -> Word8 -> State -> State -> Bool
writeK va val state state'
  = writeA va val state state'
    && isKernelAddress va
```

5.7.10 Specification of WriteU

The `writeU` specification is identical to `writeK` except that the user domain may modify normal pages that are mapped to a user-space address, rather than a kernel-space address. Otherwise the relationship between the before and after states is exactly the same. We use `writeA` to express this common relationship.

```
writeU :: VirtualAddress -> Word8 -> State -> State -> Bool
writeU va val state state'
  = writeA va val state state'
    && isUserAddress va
```

5.8 VERIFYING MEMORY-SAFETY

Verifying memory-safety for the model of H, and ultimately the implementation, are important steps for increasing the assurance of the abstraction layer. A proof of the unwinding conditions is particularly important, because such a proof is necessary to demonstrate that our instantiation of the Rushby framework is valid. In this section, we complete the instantiation of the Rushby framework by providing a definition of the *step* function that connects our action specifications to our noninterference model. We also state the theorem that the completed instantiation satisfies the unwinding conditions. The proof of this theorem remains as future work, but we outline the expected high-level proof structure and sketch some the basic steps that it requires. Chapter 10 will cover our ongoing and future work to complete the proof and to formally verify the unwinding conditions for our model of H.

5.8.1 Completing the Rushby Instantiation

Recall from Section 5.1.1 that the Rushby framework models system execution using a state transformer function:

$$step :: (S, A) \to S.$$

This function produces the state that results from applying a given action in a particular state. The *step* function is the final piece that is missing from our system instantiation, which, according to the definition in Section 5.1.1, is a machine M that consists of:

- a set of states, S, (defined in Section 5.3);
- a set of actions, A, (defined in Section 5.4.1);
- a set of outputs, O (also defined in Section 5.4.1);
- an execution function, $\mathit{step} :: S \times A \rightarrow S$; and
- an output function, $\mathit{output} :: S \times A \rightarrow O$ (defined in Section 5.4.1).

The action specifications presented in Section 5.7 describe the intended effect of each action on the system state, so our definition of *step* should collect each of these action meanings into a single execution function. However, the relational style of our specifications is not a natural fit with the *step* function because it allows a single action to produce an arbitrary number of valid result states.

To bridge the gap between our specification and the semantics of *step* within the Rushby framework, we introduce an intermediate property,

$$\mathit{canStepTo} :: \mathit{State} \rightarrow A \rightarrow \mathit{State} \rightarrow \mathit{Bool},$$

which describes a relational notion of *step*.

$$\mathit{canStepTo}\ s\ a\ s' \ = \ \mathit{wellFormed}\ s \wedge \mathit{wellFormed}\ s' \wedge \mathit{actionSpec}\ a\ s\ s'$$

$\mathit{canStepTo}\ s\ a\ s'$ is true for any well-formed states s and s' where the after state s' is reachable from the before state s via the action a. This property encapsulates the notion that the two states are related by the specification for action a by projecting the `actionSpec` relation to well-formed states.

We can use *canStepTo* to produce the set of all after states that are related to a particular before state by an action. For example,

```
relatedStates s a = [s' | s' <- allStates, canStepTo s a s']
  -- allStates = list of all values of type State
```

describes the set of states that are related to `s` by `a`. We use `relatedStates` to define *step* by returning an arbitrary state from the set of possible after states.

We know that it is possible to pick an arbitrary state in this way by the axiom of choice. If there are no after states related to a particular before state—for example, because the before state is not well-formed—then *step* acts as the identity function.

```
step :: (State, Action) -> State
step (s, a) = case relatedStates s a of
                []     -> s
                states -> pickArbitrary states
```

We design this definition of *step* with the invariants of Sections 5.3.3 and 5.7 in mind:

- every action produces a well-formed after state when applied to a well-formed before state; and
- given a valid action, a, in a before state, s, the after state, s' will satisfy $\mathit{actionSpec}\ a\ s\ s'$.

In our definition of *step*, invalid actions cannot modify the state, because an implementation should block such actions without allowing them to make any changes to the system. In practice, an implementation would likely return error information to the actor describing the cause of the failure (for example, an integer error code). We do not model return values of operations at all in our formalization, so invalid actions appear to be no-ops in our definition of *step*.

5.8.2 Properties of the H Specification and System Model

In order for our instantiation of Rushby to be valid, the system model must satisfy the three unwinding conditions, which ensure that actions in the system behave in a way that is consistent with the policy. One goal of the policy for our instantiation is to enforce the execution environment integrity property, ensuring that the run-time system cannot be corrupted. Thus, an important part of the proof of the unwinding conditions will involve demonstrating that no other domain can interfere with the environment domain. In this section, we present two supporting lemmas that we have already identified as relevant to a complete proof that actions of other domains are not visible to the environment, and sketch their proofs to illustrate the kind of techniques that are useful for verifying the unwinding conditions.

Lemma 1 (No H Changes to Environment Mappings)**.** *No action belonging to the H domain modifies the virtual-to-physical memory mappings for any memory page with the* `Environment` *status:*

$$\begin{aligned}
&\forall s \in \mathit{State},\ \mathit{pd} \in \mathit{PageDirectory},\ \mathit{vpa} \in \mathit{VirtualPageAddress},\ a \in \mathit{Action}.\\
&\quad \mathit{dom}(a) = H \ \wedge\ \mathit{isEnvironmentPage}\ s\ (\mathit{translatePage}\ s\ \mathit{pd}\ \mathit{vpa})\ \Rightarrow\\
&\quad \mathit{translatePage}\ s\ \mathit{pd}\ \mathit{vpa} = \mathit{translatePage}\ (\mathit{step}(s,a))\ \mathit{pd}\ \mathit{vpa}
\end{aligned}$$

Proof Sketch. There are seven cases: one for each action belonging to the H domain. `DeriveRegionH`, `AllocatePageDirectoryH`, and `FreePageDirectoryH` do not modify any memory mappings and this property is clearly stated in the specification of each action. Thus, we prove the lemma in a straightforward manner for each of these cases. The lemma is more difficult to prove for the `AddMappingH`, `RemoveMappingH`, and `AddKernelMappingH` cases, because these actions do change the virtual-to-physical mappings for a set of addresses. We must demonstrate that the mappings visible to the environment domain remain constant even in the presence of these updates to the virtual-memory translation structures. Let us examine the particular case of `AddMappingH` as an example of how we might proceed.

Case: AddMappingH The `AddMappingH` action takes four parameters: a page-directory in which to add new mappings, a list of physical addresses that correspond to pages that may be turned into page-tables, a physical region to map, and a virtual region in which to add the new mappings. In order for the lemma to be true, the following properties must hold of our definition:

(i) The virtual region must not contain any address that is already mapped to a page with the environment status;

(ii) The physical region must not contain any pages that have the `Environment` status; and

(iii) No page supplied as a possible page-tables may have the `Environment` status.

The first property stems from a line in the specification that states that the virtual region can only be mapped if it contains entirely user-level addresses. When combined with the well-formedness conditions `environmentOnlyInReference`—which states that pages with the `Environment` status are not mapped to any addresses that are not mapped in the reference page-directory—and the condition `referenceMapsKernelAddresses`—which states that the reference page-directory does not map any user-level addresses—we can conclude that the action will not map any virtual region that is already mapped to an environment page. We can establish the second property from the fact that `AddMappingH` will only map physical pages that belong to a physical region contained in the `allRegions` component of the system state. The well-formedness condition `regionsAreNotEnvironment` guarantees that no such region contains a page with the `Environment` status, so the specification for `AddMappingH` upholds the second property. The third property is covered by the following line in the specification,

```
all (unmappedNormalPage state) pts
```

which states that every element of `pts`, the list of page-table pages, must be a free page with the `Normal` status. Thus, none of the page-table pages can be environment pages. Having demonstrated these three key properties, we can conclude that the lemma holds for `AddMappingH`.

The requirements for proving the `RemoveMappingH` and `AddKernelMappingH` cases are very similar to the `AddMappingH` case, as are the specifications of these actions. RemoveMappingH is defined with the same specification as AddMappingH, so the proof should be very similar (though dealing with unmapping rather than mapping). The specification for AddKernelMappingH is largely the same but with some simplifications because kernel mappings do not need to allocate page-tables. Next, let us examine the proof for ExecuteH, the final case necessary for this lemma.

Case: ExecuteH ExecuteH installs a given page-directory as the current page-directory by changing the value of CR3 in the state. The specification states that any page with the PageDirectory status may be installed in this way. We can use the well-formedness conditions to demonstrate that even this weak restriction is sufficient to ensure that ExecuteH cannot modify the virtual-to-physical memory mappings of the environment domain. The well-formedness condition pageDirectoriesContainReference tells us that any page with the PageDirectory status will contain all of the mappings that are present in the reference page-directory. When combined with referenceContainsEnvironment—which states that the reference page-directory includes mappings to all of the pages with the Environment status—and the condition environmentOnlyInReference—which states that there are no mappings to environment pages that are not contained in the reference page-directory—we can conclude that every PageDirectory contains the same mappings to the environment pages. Thus, the environment domain will not be able to distinguish the mappings available in any page-directory in a well-formed state.

□

Lemma 2 (No H Changes to Environment View)**.** *No action belonging to the H domain modifies the view of the environment domain:*

$$\forall s \in State,\ a \in Action.\ dom(a) = H \ \Rightarrow\ s \overset{E}{\sim} step(s, a)$$

Proof Sketch. The view of the environment domain includes the location, contents, and reference count for each memory page that is mapped with the `Environment` status. We know that no H action changes the location at which an environment page is mapped from Lemma 1. The reference count of an environment page will not change unless a virtual-to-physical mapping for that page is added or removed, so we can also conclude that no H action changes the reference count as a consequence of Lemma 1. We can prove that no action changes the contents of an environment page by examining the specification of each action.

The specifications for `DeriveRegionH` and `ExecuteH` include frame conditions stating that they do not modify the status of any page. The status value of an environment page includes its contents, so these actions cannot change the contents of an environment page. `AllocatePageDirectoryH` and `FreePageDirectoryH` will only change the status of the page provided as an input value, which the specification states must not be an environment page. The specifications for `AddMappingH` and `RemoveMappingH` explicitly state that the data contained in any page with the `Environment` status does not change. The specification for `AddKernelMappingH` states that all physical memory stays the same except for pages with the `PageTable` status. □

5.8.3 Proving the Unwinding Conditions

Now that we have completed the definition of our system and built a collection of lemmas describing some of its properties, we can prove that our instantiation of the Rushby framework is valid. If our system is indeed a valid instantiation, then we can conclude that our system also satisfies the Unwinding Theorem from Section 5.1.3 and that our action specifications enforce the execution environment integrity and address space integrity properties.

Theorem 2 (Valid Instantiation)**.** *The definitions provided throughout this chapter for the components of the Rushby framework,*

(i) the machine, M,

(ii) the set of domains, D, and

(iii) the policy that governs domain interactions,

form a valid instantiation that satisfies the unwinding conditions.

Proof Sketch: We present an outline of the high-level steps necessary to prove the correctness of Theorem 2. A full proof remains as future work (see Chapter 10).

Step 1: Unfold the definition of *step* to produce subgoals in terms of *canStepTo*. We begin with the unfolding for local respect, which is defined directly in terms of *step*:

$$dom(a) \not\rightsquigarrow u \;\Rightarrow\; s \overset{u}{\sim} step(s, a).$$

For the case where there exists at least one state that is related to s by a, unfolding the definition of *step* gives the following statement of local respect:

$$dom(a) \not\rightsquigarrow u \;\Rightarrow\; s \overset{u}{\sim} pickArbitrary\ (relatedStates\ s\ a)$$

For the case where no related state exists, the right-hand side of the property reduces to $s \overset{u}{\sim} s$, which is trivially true. If we let s' equal an arbitrary state that is related to s by a, then the property reduces to:

$$dom(a) \not\rightsquigarrow u \wedge canStepTo\ s\ a\ s' \;\Rightarrow\; s \overset{u}{\sim} s'$$

by substitution.

A similar unfolding technique applies to weak step consistency, which has the following definition in terms of *step*,

$$s \overset{u}{\sim} t \;\wedge\; s \overset{dom(a)}{\sim} t \;\Rightarrow\; step(s,a) \overset{u}{\sim} step(t,a)$$

The intermediate proof steps follow the same pattern as in local respect, so we skip straight to the final representation where s' has been substituted for an arbitrary state that is related to s by a and t' is an arbitrary state that is related to t by a.

$$s \overset{u}{\sim} t \wedge s \overset{dom(a)}{\sim} t \wedge canStepTo\ s\ a\ s' \wedge canStepTo\ t\ a\ t' \;\Rightarrow\; s' \overset{u}{\sim} t'$$

As before, the property is trivially true if there is no after state that is related to the before state by the action.

Step 2: Demonstrate that output consistency reduces to weak step consistency under our instantiation of the output function *output*. Recall the definition of output consistency from Section 5.1.3:

$$s \overset{dom(a)}{\sim} t \;\Rightarrow\; output(s,a) = output(t,a).$$

Expanding the definition of *output* gives us:

$$s \overset{dom(a)}{\sim} t \;\Rightarrow\; view\ (dom(a))\ (step(s,a)) = view\ (dom(a))\ (step(t,a)).$$

Now the right-hand side of the arrow is equivalent to our definition of the view-partitioning relation.

$$s \overset{dom(a)}{\sim} t \wedge s \overset{dom(a)}{\sim} t \;\Rightarrow\; step(s,a) \overset{dom(a)}{\sim} step(t,a)$$

Applying the same unfolding techniques that we employed in Step 1 for local respect and weak step consistency, we obtain,

$$s \overset{dom(a)}{\sim} t \wedge canStepTo\ s\ a\ s' \wedge canStepTo\ t\ a\ t' \;\Rightarrow\; s' \overset{dom(a)}{\sim} t'.$$

Recalling the definition of weak step consistency,

$$s \overset{u}{\sim} t \wedge s \overset{dom(a)}{\sim} t \wedge \mathit{canStepTo}\ s\ a\ s' \ \wedge \mathit{canStepTo}\ t\ a\ t' \ \Rightarrow\ s' \overset{u}{\sim} t'$$

we can see that output consistency is now equivalent to weak step consistency in the special case where u equals $dom(a)$. Thus, in future efforts to verify the correctness of our system, it will be sufficient to prove just the local respect and weak step consistency unwinding conditions for our actions.

Step 3: Prove local respect by cases. The proof of local respect splits into twelve top-level cases. These cases come directly from the definition of the property, which quantifies over all noninterfering actions.

$$dom(a) \not\rightsquigarrow u \wedge \mathit{canStepTo}\ s\ a\ s' \ \Rightarrow\ s \overset{u}{\sim} s'$$

A noninterfering action is one that belongs to a domain that does not interfere with some other domain. There are six cases in our security policy where one domain is guaranteed not to interfere with another: $E \not\rightsquigarrow U$, $H \not\rightsquigarrow E$, $K \not\rightsquigarrow E$, $U \not\rightsquigarrow E$, $K \not\rightsquigarrow H$, and $U \not\rightsquigarrow H$. To prove local respect we must prove that all of the actions in each of the noninterfering domains do not produce observable effects in the state of the uninterfered with domain. Each of E, K, and U only have one action, whereas the H domain contains seven. Instantiating the definition of local respect gives us the following proof obligations, accompanied by a brief discussion of the intuition behind the proof of each property:

(a) $\mathit{canStepTo}\ s\ \texttt{WriteE}\ s' \Rightarrow s \overset{U}{\sim} s'$

This case describes the fact that a write performed by the environment domain must not be observable to any user-level programs. We use hardware-based protection to ensure that user-level programs cannot access the environment pages, which are mapped with kernel-only permission. Thus, our implementation must always map the environment pages with kernel-only permissions. The well-formedness condition `environmentOnlyInReference` describes the fact that the environment pages are only mapped in the reference page-directory, which, according to the `referenceMapsKernelAddresses` condition, contains only kernel-mapped addresses. Thus, these two aspects of well-formedness combine to ensure that all well-formed states satisfy the necessary correctness criteria to make this case of local respect true.

(b) $\mathit{canStepTo}\ s\ \texttt{DeriveRegionH}\ s' \Rightarrow s \overset{E}{\sim} s'$

Lemma 2 demonstrates that no action belonging to the H domain modifies the view of the environment domain. The `DeriveRegionH` action belongs to the H domain, so we can conclude that any successful step via this action does not affect E's view.

(c) $\mathit{canStepTo}\ s\ \texttt{AllocatePageDirectoryH}\ s' \Rightarrow s \overset{E}{\sim} s'$

Follows from Lemma 2, similarly to the previous case.

(d) $\mathit{canStepTo}\ s\ \texttt{FreePageDirectory}\ s' \Rightarrow s \overset{E}{\sim} s'$

Follows from Lemma 2.

(e) $\mathit{canStepTo}\ s\ \texttt{AddMappingH}\ s' \Rightarrow s \overset{E}{\sim} s'$

Follows from Lemma 2.

(f) $\mathit{canStepTo}\ s\ \texttt{RemoveMappingH}\ s' \Rightarrow s \overset{E}{\sim} s'$

Follows from Lemma 2.

(g) $\mathit{canStepTo}\ s\ \texttt{AddKernelMappingH}\ s' \Rightarrow s \overset{E}{\sim} s'$

Follows from Lemma 2.

(h) $\mathit{canStepTo}\ s\ \texttt{ExecuteH}\ s' \Rightarrow s \overset{E}{\sim} s'$

Follows from Lemma 2.

(i) $\mathit{canStepTo}\ s\ \texttt{WriteK}\ s' \Rightarrow s \overset{E}{\sim} s'$

The specification of `WriteK` states that the action will only modify a virtual address that is mapped to a physical page with the `Normal` status type. Thus, `WriteK` cannot be used to modify environment pages or the translation pages that affect the view of the environment domain.

(j) $\mathit{canStepTo}\ s\ \texttt{WriteU}\ s' \Rightarrow s \overset{E}{\sim} s'$

`WriteU` and `WriteK` share a common specification, except for the portion of the action description that states whether user or kernel addresses are changed, so the same reasoning applies as in the previous case.

(k) $\mathit{canStepTo}\ s\ \texttt{WriteK}\ s' \Rightarrow s \overset{H}{\sim} s'$

Some of the rationale applied in the E case applies here, namely, that `WriteK` will only change the data that is contained in normal pages. The view of the H domain does not permit it to observe the contents of normal pages, so writes to normal pages will not be observable. The frame conditions in the `WriteK` specification ensure that the reference page-directory and regions state components, both of which are observable to H, do not change. The action does not change the status of any page. Thus, no state component that H can observe is modified by `WriteK`, so the view of the state seen by H in the before and after state will be the same.

(l) $\mathit{canStepTo}\ s\ \texttt{WriteU}\ s' \Rightarrow s \overset{H}{\sim} s'$

As described earlier, `WriteU` is specified in terms of the same relation as `WriteK` so the same rationale applies as in the previous case.

At the time of writing, we have mechanically verified cases (a), (b), (h), (i), (j), (k), and (l), following the basic intuition described here. We anticipate that proving the cases that relate to mappings management will be more complex than the proofs so far, but are hopeful that the sketches presented here will help direct us towards a proof architecture that minimizes the difficulty.

Step 4: Prove weak step consistency by cases. Recalling the definition of weak step consistency that we derived in Step 1,

$$s \overset{u}{\sim} t \wedge s \overset{dom(a)}{\sim} t \wedge \mathit{canStepTo}\ s\ a\ s' \ \wedge \mathit{canStepTo}\ t\ a\ t' \ \Rightarrow\ s' \overset{u}{\sim} t'$$

we can see that this proof will require 40 cases: one per action for each of the four protection domains.

We can eliminate 12 of these 40 cases immediately because weak step consistency can be derived from local respect for any action, a, such that $dom(a)$ does not interfere with u. Recall the definition of local respect:

$$dom(a) \not\rightarrowtail u \wedge \mathit{canStepTo}\ s\ a\ s' \ \Rightarrow\ s \overset{u}{\sim} s'.$$

$\mathit{canStepTo}\ \ s\ \ a\ \ s'$ is an assumption of weak step consistency, so when $dom(a) \not\rightarrowtail u$, we can conclude that $s \overset{u}{\sim} s'$. Similarly, we can conclude that $t \overset{u}{\sim} t'$ using local respect. The assumption $s \overset{u}{\sim} t$ combined with transitivity of $\sim$ gives us the desired conclusion, $s' \overset{u}{\sim} t'$.

It remains to prove the remaining 28 cases where the domain of the action, a, is permitted to interfere with the domain, u. Intuitively, weak step consistency captures the property that information cannot flow from a

$$\begin{tikzcd}
S \arrow[r, "a"] \arrow[d, "{(view\ (dom(a)),\ view\ u)}"'] & \mathcal{P}(S) \arrow[d, "view\ u"] \\
V_{dom(a)} \times V_u \arrow[r, dotted] & \mathcal{P}(V_u)
\end{tikzcd}$$

Figure 5.3: An illustration of the intuition behind weak step consistency. S represents the set of states, $V_{dom(a)}$ represents the view of the acting domain, and V_u represents the view of the domain u. With our relational specification, a single before state will be related to a set of after states by an action, a. For weak step consistency to hold, there must exist a function (represented by the dotted line) from $V_{dom(a)} \times V_u$ to $\mathcal{P}(V_u)$.

third-party domain (a domain that is neither the actor nor the domain, u) to u. We can express this property by saying that the view of u in the after state must be a function of the views of $dom(a)$ and u in the before state. To handle nondeterministic actions, we express the property by saying that the set of u views in the possible after states is a function of the before views. Figure 5.3 illustrates this intuition.

To complete the proof of weak step consistency, we can employ the same reasoning techniques that were used in the proof sketches of Lemma 1, Lemma 2, and local respect. Specifically, we must demonstrate the existence of a function from the before views to the set of after views in each case. We sketch the proof of an example case in more detail to illustrate the application of these techniques to weak step consistency.

Example Case: Let $a = WriteE\ va\ val$ and $u = H$.

In this case, we must prove that a write action in the environment domain does not update the state in such a way that information from the K or

U domains becomes visible through the view of H. By design, `WriteE` stores *val* in memory at virtual-address *va* and should not make any other changes to the state. We can see from the specification of `WriteE` that *va* must map to a page with the `Environment` status in the before state in order for there to be any related after states. The status of all other physical pages, which includes their contents, does not change. The contents of environment pages are within H's view, so H can observe the data value that is written. Thus, we must show that the views of E and H contain all of the information necessary to perform the write, because otherwise H might be able to distinguish the after states, s' and t', based on invisible differences between the before states, s and t.

The only information used to perform the write is the virtual-to-physical mapping for the page address *va*. H can observe the virtual-to-physical mappings for every page-directory, including the reference page-directory, which is used by the environment domain. To connect this explanation to the intuition presented in Figure 5.3, we can also construct the function that maps $V_E \times V_H$ to $\mathcal{P}(V_H)$:

```
\(e :: EView, h :: HView) ->
  if isJust (eObservablePages e (fst va)) then {h'} else {}
where
  h' = h{ pages = pages' }
  pages' ppa = if ppa == ppa' then page' else pages h ppa
  ppa' = translatePageH h (referencePdir h) (fst va)
  page' = case pages h ppa' of
            Just (Environment pagedata) ->
              let pagedata' off =
                    if off == snd va then val else pagedata off
              in Just (Environment pagedata')
            x -> x
```

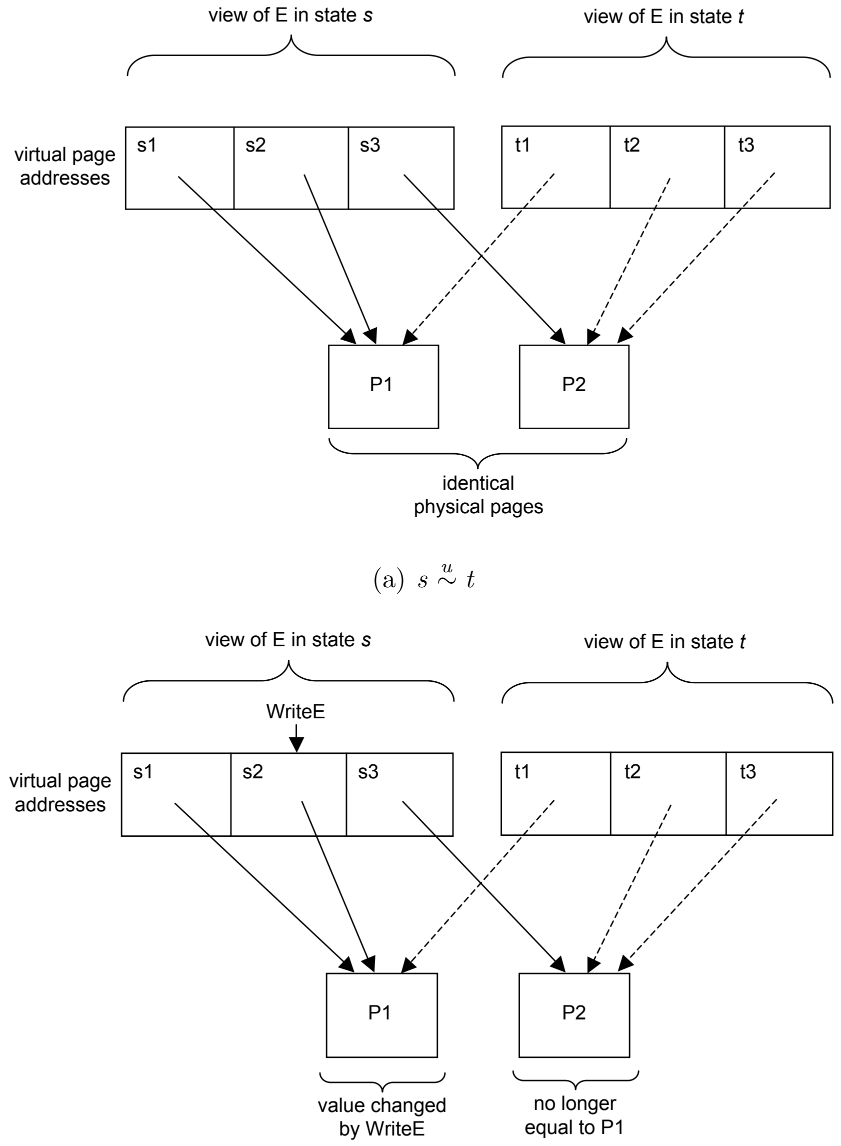
Here, `translatePageH` is simply a specialized version of `translatePage` that performs the address translation using the structures in the H state, rather than a global state. By defining this function, and by our previous argument that `WriteE` only uses information from $V_E \times V_H$, we can conclude that weak step consistency holds in this case. We leave the proof of the remaining 27 cases as future work.

□

5.8.4 Model Validation

Though our proof of the unwinding conditions is still incomplete, just the act of sketching out the proof has already led to minor changes in the formalization. The model presented in this chapter incorporates the appropriate adjustments, but in this section we describe the rationale for the changes in to illustrate the process of identifying and correcting an oversight in the model.

The first change concerned the view functions for observing memory in the environment, kernel, and user domains. The views in the original model did not contain any notion of a reference count on pages (see Section 5.4.1 for a discussion of domains and their views). The view of each domain was a partial mapping from virtual page address to contents without any additional information, which seemed like a reasonable abstraction for the functionality provided by virtual memory on the IA32. Unfortunately, weak step consistency cannot be proved with the original view definitions. The problem is best illustrated through an example, shown in Figure 5.4.



(a) $s \overset{u}{\sim} t$

(b) $s \overset{u}{\not\sim} t$

Figure 5.4: Motivation for reference counts in the views of the environment, kernel, and user domains. A domain can observe the fact that a particular physical page is multiply mapped because a single write will change the value of more than one location. However, without reference counts, the domain will not be able to distinguish two states with different multiple mapping configurations. This is a violation of weak step consistency, because executing the same action in (seemingly) equivalent before states will not produce equivalent after states.

In Figure 5.4(a), the states s and t are indistinguishable because the visible virtual addresses all map to the same values. However, a single `WriteE` action produces states that are no longer equivalent because of the hidden sharing, as shown in Figure 5.4(b). Weak step consistency guarantees that the after states produced by executing an action in equivalent before states will be equivalent, which is not true in this case. We can conclude from this example that physical page sharing between multiple virtual addresses is an important part of the environment domain's view. After making this observation, we updated the view of the environment domain to include a reference count that reflected page sharing. The same rationale applies for the kernel and user domains, so we added reference counts to their views as well.

Another change we made to the model was to introduce the page status value `NotInstalled`. During the course of developing the model, we experimented with a few different approaches for handling the possibility that some addressable pages of physical memory may not be present in a given system configuration. At one point, we explicitly included a parameter describing the list of installed pages and constructed all of our specifications using forall and exists operators over this list. This approach did seem like a good fit once we began to think about mechanization, so we were motivated to introduce the status value `NotInstalled`.

5.9 SUMMARY

In this chapter, we provided a definition of memory-safety for the H interface in terms of a noninterference security policy. We instantiated the Rushby framework [82] to formalize two novel properties called execution environment integrity and address-space integrity that supplement the traditional definition of memory-safety. These integrity properties ensure that the Haskell run-time environment cannot be corrupted, even when we use potentially unsafe operations in the implementation of H.

Our work in this dissertation introduces the framework and methodology for reasoning about memory-safety as a noninterference property. Much future work remains before we can conclude that our implementation of H is memory-safe according to our definition. In particular, we must:

- Demonstrate that H is memory-safe according to the colloquial definition (for example, that no H operation should dereference a null pointer).

- Prove the unwinding conditions for the specifications described in Section 5.7. We have sketched some of the steps necessary for this proof in Section 5.8 and formally verified a few cases, but a complete proof is necessary before we can conclude that the specifications are memory-safe.

- Establish a formal connection between the model and the implementation. Without formally connecting the specifications to the real operations of H, we cannot use our formalism to conclude anything about the operations that are actually called by client kernels. Section 5.7 describes the properties that a valid instantiation must satisfy; we must prove these properties of our implementation to establish that our implementation is memory-safe.

As we pursue these topics, it is possible that we will need to make changes to certain aspects of the formalization presented here. In particular, the specifications, the well-formedness constraints, and the implementation of the H primitives might not satisfy the properties above without some modification. For example, the well-formedness conditions might be insufficient, in which case we would add any necessary constraints on the state.

Chapter 6

IMPLEMENTING THE ABSTRACTION LAYER

In this chapter we describe the implementation of the abstraction layer. We focus on the essential implementation techniques, rather than providing an exhaustive catalog of every function and its implementation. Sections 6.1 and Section 6.2 describe our implementation of H using a monad with fine-grained type constraints captured by type classes. Section 6.3 explains the bootstrapping code in H and the techniques that are used for communicating hardware configuration information to the client kernel. Section 6.4 discusses our approach to managing page-tables and page-directories. Section 6.5 describes the implementation of the techniques for enforcing memory-safety. Section 6.6 presents the implementation of the function that adds user-space mappings. Section 6.7 explains our implementation of kernel-space mappings. Finally, Section 6.8 explains our mechanism for executing user programs from Haskell.

6.1 SAFELY ENCAPSULATING THE ABSTRACTION LAYER OPERATIONS

We implement the H interface as a monad that supports precisely those operations that we described in the abstraction layer design in Chapter 4. This monad, called H, replaces the `IO` monad as the base monad in an H-based system. Under the hood, we implement the H monad using the facilities of the `IO` monad, but the underlying implementation is not observable to the client kernel. Clients of the H interface cannot access any effectful operations that are not explicitly exported by the H monad. All of the unsafe functionality from the IO monad is inaccessible because of the static scoping mechanisms available through the Haskell module system. Figure 6.1 illustrates this design.

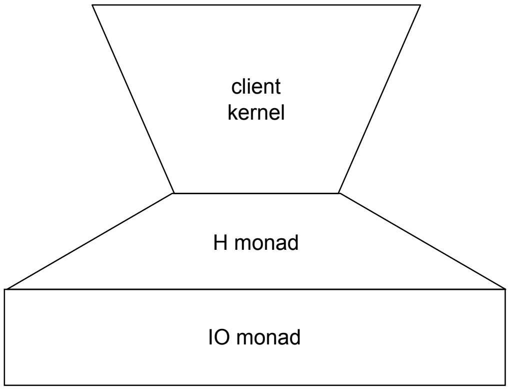


Figure 6.1: The H monad interface. Though we implement the H monad using the potentially unsafe features of the IO monad, the Haskell compiler ensures that client kernels only access the H monad operations.

The H monad is essentially a wrapper for the IO monad representation. Because the representation is hidden, we can also store private local state needed by the H implementation without being concerned that the state will be corrupted by a client kernel. The local state could be anything that we find useful in the implementation of H; for example, we introduce a single state component for tracking the virtual addresses where page-tables and page-directories are mapped (these mappings allow H to read and write page-tables/directories without page faulting).

We track the virtual address mappings for page-tables and page-directories using the Haskell library type for dictionaries called Map (defined in the standard library Data.Map). The HMap dictionary maps physical addresses to virtual addresses. The physical addresses represent the physical locations of page-tables or page-directories that we would like to access in the implementation of H. The virtual addresses correspond to locations in kernel-space through which H can access the specified physical page without faulting. We assume that the arbitrary physical and virtual addresses permitted by the `HMap` type are aligned to the page-size on the machine. A physical page that is not in the dictionary cannot be accessed by H.

```
type HMap = Map (Addr Physical) (Addr Virtual)
```

Section 6.4 will cover more details about this mechanism for physical memory access in H.

We incorporate our dictionary into the H monad using a state monad transformer with the `IO` monad as the base. We define H as a newtype with a hidden constructor so that the client cannot observe the representation of the monad.

```
newtype H a = H { unH :: ST HMap IO a }
```

`unH` extracts the underlying computation from inside the H monad constructor. We use `unH` as a convenience within the implementation of the monad; this function is also hidden from the client.

Within the definition of our H primitives, we use the `liftIO` operation to embed computations from the `IO` monad into the H monad. This is necessary because many of the low-level services that H uses to provide operating system support, for example, foreign function calls, must run in the `IO` monad. For monads constructed with a monad transformer, a value from the underlying monad can be turned into a value of the transformed type (in this case, H) using a function called `lift`, as shown in the definition of `liftIO`.

```
liftIO :: IO a -> H a
liftIO m = H (lift m)
```

It is crucial that the client cannot access `liftIO`: this function converts any `IO` operation into an H operation and would eliminate our ability to distinguish potentially unsafe computations from safe ones if used improperly. We rely on the module system, covered in Chapter 3.5, to restrict calls to this function.

The only operation available to the client is the run operation that executes a computation of type H and produces a result of type `IO`. We must include such a function because we cannot change the type of the `main` function required by Haskell: the client kernel must contain a top-level function called `main` of type `IO ()`. We intend for the client to invoke `runH` only once in the `main` function to produce a type-correct program. The client could produce unsafe code by running the safe H component of their program interspersed with unsafe `IO` operations, but the violation of the intended H model would be obvious from inspecting the code.

```
runH :: H a -> IO a
runH (H m) = do (x,_) <- (unST m) Map.empty
                return x
```

We run an H computation by supplying the state-monad run function with an initial value (an empty memory map) for the state component.

6.2 TYPE CLASSES FOR FINE-GRAINED EFFECT TRACKING

The H monad allows a client kernel to access the abstraction layer operations in a tightly controlled, safe way. The type alone tells us which side-effects a function might perform: a monadic type `H a` signals that a function may perform any side effect defined by the H monad while a non-monadic type guarantees that the function is side effect free. Expressing this kind of explicit effect tracking using types is one of the benefits to writing our kernel in Haskell, but the granularity of the effect tracking is still fairly coarse because of the diversity of effects available through H. Most functions that run on top of H will only require a subset of the H monad's functionality, but a type of the form `H a` does not allow us to distinguish which of H's operations a particular function might use.

In Chapter 3 we demonstrated how type classes allow us to express predicates on Haskell types. In particular, we can define predicates on monadic types that capture dependencies on specific monadic operators. For example, a timer interrupt handler that has no arguments and only accesses the port I/O functionality of H can be written with the type:

```
timerInterrupt :: (Port m) => m ()
```

The `Port m` constraint indicates that `timerInterrupt` depends on the functionality defined in the `Port` class. The definition of `Port` includes all of the H functions for accessing I/O ports. We know from the type alone that `timerInterrupt` will not access any of the H operations that are not described in the `Port` class—the type is a static guarantee that `timerInterrupt` does not modify any user mappings, delete any page-directories, or execute any user programs (which would be rather surprising behavior for a timer interrupt handler, after all).

The client may use these classes in isolation to define functions with precise types or in combination to create functions that access a variety of the effects available through H. The `timerInterrupt` function is an example of the former case. In contrast, consider the function for creating a new thread from our L4 kernel: this function allocates a fault context for the new thread and might allocate a new page-directory (if we are creating the thread in a new address-space). If the thread is being created in a new address-space, the function must also set up some initial user-space mappings in the new page-directory. Using the predicates defined in Table 6.1, we give `createThread` the following type:

```
createThread :: (Execution m, UserMemory m, Paging m) => m ()
```

Decomposing the H monad into fine-grained classes allows us to give more refined types to the functions of the client kernel, strengthens our compiler-checked documentation, and has the potential to make verification easier. The set of behaviors that are possible in the `timerInterrupt` function are much more limited than

Class Name	**Class Operations**	**Class Name**	**Class Operations**

`Paging`	`allocPageMap`
	`freePageMap`
	`createPageMapPage`
		`IRQControl`	`enableIRQ`
			`disableIRQ`
			`maskAckIRQ`

`UserMemory`	`addMapping`
	`modifyMapping`
	`removeMapping`
	`readMapping`
		`Execution`	`allocFaultContext`
			`readRegister`
			`writeRegister`
			`execute`

`KernelMemory`	`addKernelMapping`
	`readKernelMapping`
	`writeKernelMapping`
	`readWordAtOffset`
	`writeWordAtOffset`
		`Modules`	`modules`

`Port`	`inB`
	`inS`
	`inW`
	`outB`
	`outS`
	`outW`
		`Debug`	`putch`
			`putstr`
			`putstrln`

Table 6.1: The division of the H interface functions into type classes. See Chapter 4 for the details of each of these functions.

those in the `createThread` function, which is useful knowledge for both formal and informal reasoning.

To demonstrate the mechanism for defining type classes on monadic types in Haskell, we will examine the definition of the `UserMemory` type class and the accompanying definitions that are necessary to use the class in our client kernels with H. As shown in Table 6.1, the `UserMemory` class describes the operations on the user-portion of the address-space. It contains four operations: adding a user-space mapping; modifying the permissions on a user-space mapping; removing a user-space mapping; and reading an existing mapping.

The first step in our definition of UserMemory is to declare the class and the functions that it supports.

```
class Monad m => UserMemory m where
 addMapping    :: [PageMapPage] -> PageMap -> Fpage Virtual
                    -> PhysicalRegion -> Perms -> m (Maybe Bool)
 modifyMapping :: PageMap -> Fpage Virtual -> Perms -> m Bool
 removeMapping :: PageMap -> Fpage Virtual -> m (Maybe [PhysicalRegion])
 readMapping   :: PageMap -> Addr Virtual -> m (Maybe MappingInfo)
```

Note that the declaration of UserMemory includes a signature for each of the operations in the H interface that performs user-space mappings, but replaces each use of the H type constructor with a type variable m. Because UserMemory defines a predicate on monadic types, we include the constraint Monad m for the type variable m.

We declare two instances of the UserMemory class. The first describes how the functions of UserMemory are implemented for H, binding the overloaded names to the non-overloaded primitives defined by the H interface implementation. The name Impl.addMapping refers to the H implementation of the addMapping function that is defined in a module called Impl, and so on.

```
instance UserMemory H where
  addMapping    = Impl.addMapping
  modifyMapping = Impl.modifyMapping
  removeMapping = Impl.removeMapping
  readMapping   = Impl.readMapping
```

The second instance is for monads constructed with a monad transformer. This allows us to use the functions of UserMemory in transformed versions of the H monad, for example, a monad that adds a state component to H using a state-monad transformer.

```
instance (MonadT t, Monad (t m), UserMemory m) => UserMemory (t m) where
  addMapping    sr pm r fp p = lift (addMapping sr pm r fp p)
  modifyMapping pm fp p      = lift (modifyMapping pm fp p)
  removeMapping pm fp        = lift (removeMapping pm fp)
  readMapping   pm va        = lift (readMapping pm va)
```

The definitions in the transformer instance lift the definitions of the `UserMemory` functions into the transformed monad. The ability to use the functions of H in transformed versions of the monad is important in our case study (Chapter 7), where we will incorporate kernel-specific state-components on top of H.

The definitions for the other type classes follow the same pattern. These classes are easy to define and modify, so client kernel developers can create any organization of the H operations that turns out to be useful. In our case study, the compiler-checked documentation was useful during development and many functions did end up with very precise types. However, the organization presented here did not capture the common patterns of side-effect usage as well as it could, in particular, with respect to the `Debug` class. Adding a simple print statement to track down a bug required us to modify the type signature for the problematic function, as well as the signatures for all of the calling functions, their calling functions, etc. In a future version of the implementation, we would construct the H type classes differently so that debugging facilities are available in all H operations, rather than creating a special type class for debug operations. This would reduce the number of type signature modifications needed during debugging.

6.3 BOOTING

The H interface executes with support from a modified version of the GHC run-time system [31][1]. The modifications allow us to run Haskell programs on bare

[1] The original effort to port the GHC run-time system to bare metal was done by Jérémy Bobbio and Sébastian Carlier as part of the hOp project (unfortunately, the project website is no

metal by removing GHC's dependencies on an underlying operating system. The fundamental services provided by the run-time system, such as garbage collection and program execution, are not affected, but some of GHC's higher-level library features need to be removed.

Even though the run-time system is designed to run on bare metal, certain facilities must be in place before Haskell code can start executing. For example, we must explicitly set aside memory for the Haskell heap because there is not an underlying operating system to provide memory to GHC on demand. In this section, we present the bootstrapping tools that help us to reserve memory for the run-time system, H, and the client kernel, as well as the mechanisms through which we communicate configuration information safely from the start-up code to the client kernel.

The primary function of the bootstrapping code is to divide the available physical memory between the execution environment and the client kernel. Along the way, H performs the following initialization and configuration tasks:

- **Allocate and Initialize Page Status Values:** Every page of physical memory has an associated status value that H uses to enforce memory-safety (as discussed in Section 4.1 and Chapter 5). The bootstrapping code is responsible for allocating and initializing the data structure that tracks status information throughout the execution of H (see Section 6.5 for details about this data structure and its operations).

- **Initialize the Haskell Heap:** Our Haskell code cannot run until we provide the run-time system with memory for a heap. H selects a portion of the

longer online). Andrew Tolmach and Thomas Hallgren developed the first version of the run-time system used in the H interface in conjunction with their work on the House operating system [39]. Our current work is based on a version of the run-time system created by Kenneth Graunke [34] using patches to GHC created by Adam Wick for the HaLVM project [40] at Galois, Inc. [28].

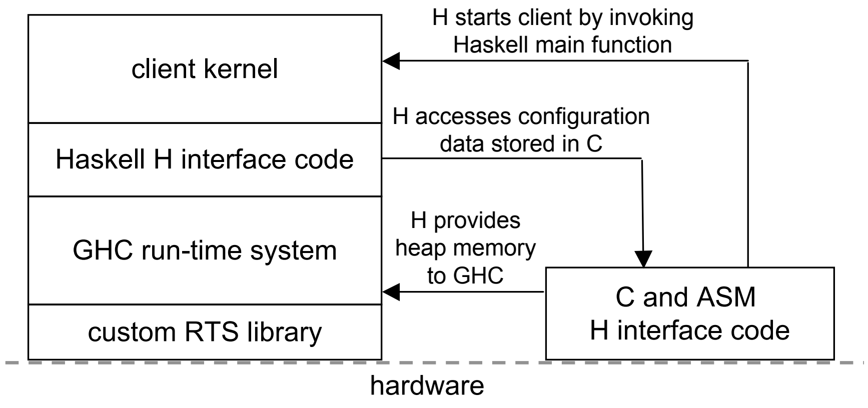

client kernel
Haskell H interface code
GHC run-time system
custom RTS library
H starts client by invoking Haskell main function
H accesses configuration data stored in C
H provides heap memory to GHC
C and ASM H interface code
hardware

Figure 6.2: The low-level architecture of H. A custom run-time system library provides GHC with bare metal implementations of important system services, such as timers. The C portion of H configures the system and provides the GHC run-time system with memory for the Haskell heap. Once the bootstrapping process is complete, H invokes the `main` function of the client kernel and we begin executing Haskell code.

available physical memory, registers this memory with the run-time system, and updates the status of the pages to reflect their new use as protected environment pages.

- **Save Module Information From Grub**[2]: H passes the set of executable modules from Grub to the client kernel by constructing safe Haskell wrappers of the raw module descriptors (see Section 4.3.2). To enable us to construct these wrappers later, the bootstrapping code saves the module descriptors in a persistent data structure that is accessible from Haskell.

[2] Grub is a bootloader package maintained by the GNU Project [37]. It runs before the kernel and passes on important configuration information from the BIOS, and information about data loaded on the machine, such as user programs and the kernel (in the form of modules).

- **Configure Kernel Space Mappings:** The bootstrapping code is responsible for setting up the reference page-directory that we use to guarantee a consistent view of kernel-space in all address spaces. We will cover the specific design of the reference page-directory in Section 6.7.

- **Compute the Initial Regions:** Any free memory remaining after the rest of the initialization tasks is passed on to the client kernel via the primitive `initialRegions`. The bootstrapping code computes the free areas of memory and saves a description of these initial regions in a data structure that the Haskell code can access later.

Once all of the configuration and initialization tasks are complete, we can begin executing Haskell code. H calls into Haskell by invoking the main function of the client kernel. The client kernel begins running and may request configuration data from the Haskell H interface, such as the set of initial regions. H handles such a request by using the FFI to access the information that we saved in C during the bootstrapping process. Figure 6.2 illustrates these components of the H architecture and the start-up procedure.

All of the configuration information computed during start-up is stored in C arrays. The Haskell code processes this raw description of the data to create Haskell data structures that H safely exports to the client. As an example of this technique, let us consider the implementation of the `modules` function, which returns the set of executable modules loaded by Grub (see Section 4.3.2 for more information about the `modules` function).

The boot-time information about modules comes from a bootloader by Mark P. Jones, called *mimg*, that runs after Grub and before the kernel initialization code. When mimg runs, it generates headers that describe the layout of physical memory, which the kernel initialization code parses to create the module structures. Each header is three words long with the following format:

address of first byte	address of last byte	entry point

The first word contains the start address of the region being described, the second contains the end address of the region, and the third contains the entry point of the module (if it is executable). An entry point of 0xFFFFFFFF signals that module is not executable. We access the array of headers by creating a pointer to the array using the FFI.

```
foreign import ccall unsafe "memory.h & hdrs"
  c_hdrs :: Ptr (Ptr HWord)
```

The `foreign` keyword signals that we are creating a foreign function that we will `import` from C (a `ccall`). The portion of the declaration that is in quotes describes the location and name of the C variable or function being accessed. In the example, the declaration imports the C variable `hdrs` with the Haskell name `c_hdrs`. The keyword `unsafe` signals to the compiler that an imported function does not call any Haskell functions (which requires extra bookkeeping).

Within the header array, the first element of the array contains the number of entries. The remaining entries follow the previously described header format. We assume that the first header describing an executable module refers to the kernel. All other executable modules from the headers array will be exported to the client kernel by `modules`. The compile-time configuration of the system must ensure that the ordering invariant holds, because exporting the kernel module would potentially allow the kernel code to be overwritten.

The `modules` function, shown in Figure 6.3, parses the header array and returns a list of `Module` structures. Much of the work done by `modules` is to split the regions described by the array into smaller flexpage-sized regions so that we can export the memory area occupied by each module using the `PhysicalRegion` type (see Section 4.3.2 for a discussion of this design choice). We use a function from the bootstrapping code (written in C) called `c_regionToFlexpages` to do the splitting.

```
modules :: H [Module]
modules =
  liftIO $
  do hdrPtr <- peek c_hdrs
     numHdrs <- peekByteOff hdrPtr 0
     fpPtr <- mallocBytes 160
     ms <- mapM (readModule hdrPtr fpPtr)
                [1..numHdrs]
     free fpPtr
     return (tail (catMaybes ms))
where
  nextPage e = align (e+(1<<<12)) 12
  createRegion :: Ptr HWord -> Int
    -> IO PhysicalRegion
  createRegion fpPtr i =
    do fpval <- peekByteOff fpPtr i
       let fp = Fpage fpval
       return (PhysicalRegion fp RAM)
```

```
readModule :: Ptr HWord -> Ptr HWord -> Int
                -> IO (Maybe Module)
readModule hdr fp i =
  do entry <- peekByteOff hdr (12*i)
     if entry /= 0xffffffff
        then do first <- peekByteOff hdr (12*i-8)
                last <- peekByteOff hdr (12*i-4)
                nfps <- c_regionToFlexpages first
                          ((nextPage last) - 1) fp
                rs <- mapM (createRegion fp)
                        [0,4..(4*(nfps-1))]
                return (Just (Module rs entry))
        else return Nothing
```

Figure 6.3: The modules function. This function processes the module configuration information supplied by the bootstrapping code in C to produce a list of Haskell Module structures.

```
foreign import ccall "memory.h regionToFlexpages"
  c_regionToFlexpages :: HWord -> HWord -> Ptr HWord -> IO Int
```

c_regionToFlexpages takes the start and end address of the region to be broken up and returns a corresponding list of flexpages in an array supplied by the caller. We allocate this array explicitly from the Haskell heap as temporary storage and then process the result to create a list of physical regions occupied by a module. For simplicity we allocate enough space to hold the maximum possible flexpages that c_regionToFlexpages could generate. This is a waste of space and could be optimized in future versions, but we only expect modules to be called once during start-up so the overhead should not be significant. The actual parsing of the headers array is done using the pointer access functions `peek` and `peekElemOff` defined in the foreign function interface.

Initial region information is communicated from C to Haskell to the client using essentially the same approach that we use for communicating module data. When the bootstrapping code begins executing, an array of header information, passed on from *mimg*, accurately describes the available memory on the machine. Throughout the bootstrapping process, we allocate portions of that available memory for the Haskell heap and for various configuration data arrays. When we finally enter Haskell, the headers array no longer provides an accurate description of memory usage, so the bootstrapping code must save a new description of available memory that accounts for all of our allocations. To make the generation of `PhysicalRegion` values easier, the C code stores flexpage-sized regions of memory that the Haskell code can use without any additional processing. As before, we import the array containing the free regions using the FFI.

```
foreign import ccall unsafe "memory.h & initial_regions"
  c_initial_regions :: Ptr (Ptr HWord)
```

The start-up code guarantees that this pointer is appropriately initialized before we begin executing Haskell code; it is never modified after that point.

The implementation of `initialRegions` (shown in Figure 6.4) turns each flexpage in the initial regions array into a `PhysicalRegion` to export to the client. We read the length of the array from the first word and map the region construction function across the remaining entries. All of these regions are initialized with normal RAM as the region type. The memory mapped I/O regions are initialized via a different mechanism: we explicitly construct each I/O region in Haskell and combine the results into a list called `ioRegions`. We currently only include video RAM in the initial regions, but the set of supported I/O regions is easily extensible (we have already added the support necessary for the VBE frame buffer, but this

```
initialRegions :: H [PhysicalRegion]
initialRegions = liftIO $ do fps <- peek c_initial_regions
                             numfps <- peekByteOff fps 0
                             ramrs <- mapM (newR fps) [4,8..(4*numfps)]
                             iors  <- ioRegions
                             return (ramrs ++ iors)
  where
    newR :: Ptr HWord -> Int -> IO PhysicalRegion
    newR fpPtr i = do fpval <- peekByteOff fpPtr i
                      return (PhysicalRegion (fpageFromWord fpval) RAM)
```

Figure 6.4: The `initialRegions` procedure. The bootstrapping code produces an array of flexpages that describe the free memory on the machine that has not been claimed for kernel purposes. The `initialRegions` procedure processes this array to create physical region handles that cannot be forged by the client.

functionality is not thoroughly tested). `initialRegions` combines the result of the region processing for conventional memory with the `ioRegions` list to produce the set of available physical regions.

6.4 PRECISE KERNEL CONTROL OVER PAGE-MAP MEMORY

Client kernels have full control over the memory that H will use to store page-tables and page-directories in the H interface design. Clients also specify the virtual addresses where these pages of memory should be be mapped in kernel-space. The H implementation must track all of the physical pages and virtual addresses that the client has supplied for the interface to use. Section 6.1 presented the mechanism for tracking client-supplied memory in the monad, whereas this section describes the implementation techniques that we employ to allow client control over page-map memory in a way that is memory-safe and that prevents kernel page faults. We present the routines that help H to enforce safe status transitions in the implementation of memory operations and the approach that we use to guarantee that H will not page fault during memory management operations, even when accessing client-supplied page-tables and page-directories that can only be mapped at client-chosen virtual addresses.

The PageMapPage type introduced in Section 4.4.1 is an abstraction for representing either a page-table or page-directory page that the client wishes to donate to H. A page-map page contains a page-sized physical region that will be used to store the table or directory and a kernel-space virtual-address where the page will be mapped. We define the type in a straightforward fashion using a record with a field for each component.

```
data PageMapPage = PageMapPage {
                     physicalPage :: PhysicalRegion,
                     mappedAddr   :: Addr Virtual
                   }
```

Each PageMapPage must satisfy certain invariants in order for H to safely use the page: the physical region component must be a single page of normal RAM and the virtual address must be a kernel-space address that is available to the client for mapping. The client-accessible function constructing values of the PageMapPage type checks that the invariants hold on the physical region virtual address that the client is attempting to turn into a page-map page. The utility function isKernelMappableVirtualAddress tests if a particular address is in the set of kernelMappableVirtualAddresses (introduced in Section 4.4.1), and will be presented in Section 6.7.

```
createPageMapPage :: PhysicalRegion -> Addr Virtual
  -> H (Maybe PageMapPage)
createPageMapPage pr@(PhysicalRegion _ RAM) va
  | regionSize pr == pageBits
    = do isClientAddr <- isKernelMappableVirtualAddress va
```

```
        if isClientAddr then return (Just (PageMapPage pr va))
                        else return Nothing
createPageMapPage _ _ = return Nothing
```

H will never create a `PageMapPage` that does not satisfy the invariants and there is no other mechanism available to the client for creating values of the `PageMapPage` type.

Once the client creates a `PageMapPage`, they can supply that page to any H function that might need to allocate a page-directory or page-table. The physical page and the virtual address contained in the page-map page remain free until H actually needs the memory. At that point, H performs any state-dependent safety checks that could not be performed in advance, converts the status of the page to the appropriate type (page-table or page-directory, depending on the context in which the page is being used), and installs a mapping from the specified virtual address to the physical page.

The `addHMapping` function (shown in Figure 6.5) is responsible for adding the association between the physical region and the virtual address to the `HMap` dictionary. Before updating the dictionary, `addHMapping` must make sure that the virtual address component is not already mapped to a page-table or page-directory in the dictionary. After a successful call to `addHMapping`, the `HMap` dictionary will contain an entry for the page-map page. We do not modify the status of the page here; the caller updates the status (and checks the safety of the transition) as appropriate for the intended use of the page.

Installation of the virtual-to-physical mapping where H can read and write the page-map page is handled by the utility `installPageMapPage`, also shown in Figure 6.5. `installPageMapPage` adds a kernel-space mapping from the virtual address contained in the page-map page to the start address of the physical region component. The implementation ensures that the mapping will be visible in every address space. (We will cover the techniques for adding mappings in Sections 6.6

```
addHMapping :: PageMapPage -> H Bool
addHMapping (PageMapPage pr va) =
  do inuse <- isMappedToH va
     if inuse then return False
        else do update (\hms -> ins hms)
                return True
  where
    paddr = regionStart pr
    ins   = Map.insert paddr va
```

```
installPageMapPage :: PageMapPage -> H Bool
installPageMapPage pmp =
  do vpdir <- systemPageMapMappedAddress
     pt_addr <- liftIO $
                  c_find_pt vpdir (mappedAddr pmp)
     if pt_addr == (-1) then return False else
        do liftIO $
                  c_add_table_entries vpt vfp ps 0 0x7
           return True
  where
    vpt = fromIntegral pt_addr + kernelSpace
    vfp = fpageToWord
            (fpage (mappedAddr pmp) pageBits)
    ps  = regionStart (physicalPage pmp)
```

Figure 6.5: Utilities for validating and installing a `PageMapPage`. The `addHMapping` function adds an entry to the `HMap` dictionary for a particular page-map page after making sure that the virtual address component is free. The `installPageMapPage` function adds a mapping in kernel space from the virtual address component of a page-map page to the physical page component. The address-space manipulations are done with the help of the C functions `c_find_pt` and `c_add_table_entries`—these functions will be covered in more depth in Section 6.6. `systemPageMapMappedAddress` is a pointer to our implementation of the reference page-directory presented in Chapter 5.

and 6.7.)

As we saw in Chapter 4, H automatically returns the memory for any page-tables that become free as the result of a `removeMapping` operation. Client kernels explicitly free page-directories using the `freePageMap` operation. Before returning one of these pages to the client, H must remove the entry for the page from the

HMap dictionary. As with the utilities for creating a page-map page, the caller is responsible for updating the status of the page as appropriate.

```
removeHMapping :: Addr Physical -> H ()
removeHMapping pa = update
  (\(hms::HMap) -> Map.delete pa hms)
```

The implementation of `removeHMapping` uses the state-monad function `update` (which applies a given function to the state) in conjunction with the library function `Map.delete` to modify the `HMap` dictionary.

Allowing the client to control the physical and virtual memory that H uses for page-tables and directories guarantees that we always reserve precisely the right amount of memory for page-map storage. If we did not allow the client to control page-map pages in this way, then H would need to designate areas of memory statically for page-map storage. Undoubtedly the static solution would lead to an underutilization of resources or a premature exhaustion of the memory pool (even if there were free pages/addresses available in the client kernel). Instead, with our dynamic approach, H must do some extra tracking and dynamic checking to ensure that the memory-safety invariants of the system are never violated and that H does not page fault. However, we improve memory utilization and allow clients greater flexibility to define their own memory management policies.

6.5 ENFORCING SAFE PAGE STATUS TRANSITIONS

As we saw in Section 4.1, each page of physical memory has a dynamic status that reflects its current usage in the system. We identified four types of physical memory—page-directory pages, page-table pages, normal pages, and environment pages—and defined a limited set of transitions between these page types. Run-time checking of every transition is a fundamental part of our memory-safety enforcement mechanism. Every H operation must satisfy the specification described in

Section 5.7, and, more generally, the high-level memory-safety property introduced in Chapter 5. In this section, we will examine the implementation of page status tracking in H and the utilities that H uses to validate status transitions during execution.

We represent dynamic status values in the interface using a Haskell datatype. The datatype has four constructors—`Protected`, `Reserved`, `PageMapHandle`, and `Mapped`—that correspond directly to the four roles for memory—environment pages, page-table pages, page-directory pages, and normal pages—respectively.

```
data Status = Protected | Reserved | PageMapHandle | Mapped HWord
```

The `Mapped` constructor takes a single argument that serves as a reference count for the number of times that a particular page is currently mapped. Reference counting is an important part of our mechanism for ensuring that user processes are never able to access memory when it is in use by the kernel for paging structures or other data.

We track page status values in an array that maps each page of physical memory to an integer status value. We define the array in C and create a Haskell interface to the standard operations on the array. We use a C array so that we can easily initialize the status array in the start-up code for H: the start-up code configures the environment and set of available pages, and initializing the status values during boot prevents us from having to package this information up to pass to Haskell. Each integer status value maps to a value of the `Status` type that we defined in Section 4.1: -1 corresponds to a `Protected` environment page, -2 corresponds to a `Reserved` page-table page, -3 corresponds to a `PageMapHandle` page-directory page, and any number greater than or equal to 0 corresponds to a `Mappable` normal page. For normal pages, the status value also captures a reference counter, that is, the number of places where the page is currently mapped.

The interface to the status array contains three functions that we lift into Haskell using the foreign function interface: an operation for reading the status of a page, an operation for modifying the status of a page, and an operation for modifying the status of many pages. For the single-page functions, the argument of type HWord is the address of the page whose status is being accessed; for the region update function the arguments of type HWord are the addresses of the start and end of the region.

```
foreign import ccall "memory.h readPageStatus"
  c_readPageStatus :: HWord -> IO Int

foreign import ccall "memory.h updatePageStatus"
  c_updatePageStatus :: HWord -> Int -> IO ()

foreign import ccall "memory.h updateRegionStatus"
  c_updateRegionStatus :: HWord -> HWord -> Int -> IO ()
```

These functions do not perform any checking or validation; they rely on the calling function to validate any transition being requested before modifying the array. Throughout the implementation of H, we will update status values using wrapper functions that lift the behavior of these basic primitives into the H monad and check to make sure that the status transition being performed is safe.

Reading the status array does not introduce any safety issues, so the wrapper function is very simple. We read the integer status value from the array and convert this number into a Status value with the toEnum function. We define a custom instance of the Enum class—which describes operations for converting between integers and datatypes, including the conversion functions toEnum and fromEnum—for the Status type using the previously described mappings between integers and the constructors of this type.

```
readPageStatus :: Addr Physical -> H Status
readPageStatus page
  = do status <- liftIO $ c_readPageStatus page
       return (toEnum status)
```

We assume that the status array never contains an invalid number—we rely on the correctness of the initialization code and the status array update functions to enforce this property.

Correctly managing the status array is crucial from a memory-safety perspective. If we allow an invalid transition to occur by writing an invalid value into the array, the system loses the ability to protect the integrity of environment and page-map pages. We define the predicate `checkTransition` to validate a possible page status transition before we actually update the status array. This predicate reads the current status of a specified page and checks whether or not the proposed new status represents a valid transition. `checkTransition` is false when we attempt to modify the status of an environment page or convert a normal page that is mapped to a user into a page-table or page-directory.

```
checkTransition :: Status -> Addr Physical -> H Bool
checkTransition new pa
  = do old <- liftIO $ c_readPageStatus pa
       case (toEnum old, new) of
         (Protected, _)                     -> return False
         (_, Protected)                     -> return False
         (Mapped 0, Reserved)               -> return True
         (_,        Reserved)               -> return False
         (Mapped 0, PageMapHandle)          -> return True
         (_,        PageMapHandle)          -> return False
         (_, _)                             -> return True
```

If `checkTransition` returns `True`, then we know that updating the status array with the proposed value will be safe.

To modify the status of a single page, we validate the safety of the requested transition using `checkTransition`, updating the underlying status only if the transition turns out to be safe.

```
updatePageStatus :: Addr Physical -> Status -> H Bool
updatePageStatus pa status
  = do proceed <- checkTransition pa status
       if proceed
          then liftIO $ c_updatePageStatus pa (fromEnum status)
          else return ()
       return proceed
```

The result of updatePageStatus indicates whether or not we successfully updated the page status.

To modify the status of an entire region, we must validate the status transitions for the pages of the region individually. We again use checkTransition for this purpose. A region update may only proceed if all of the status transitions are safe.

```
updateRegionStatus :: Addr Physical -> Addr Physical -> Status -> H ()
updateRegionStatus start end status
  = do proceeds <- mapM (checkTransition status)
                        [start, start+pageSize..end]
       if and proceeds
          then liftIO $ c_updateRegionStatus start end (fromEnum status)
          else return ()
```

The status update functions are private to the implementation of H, even though the update functions only allow safe transitions. H relies on the correctness of the status array and the client kernel cannot be permitted to affect H's internal structures.

In addition to the basic primitives for manipulating status settings, we define a number of utility functions that make updating the status values of an entire region easier. For example, we frequently need to increment the reference count for a mappable region, even if the reference count of the individual pages is not the same.

```
incrementMappedCount :: PhysicalRegion -> H ()
incrementMappedCount pr
  = do refCounts <- mapM readCount pagelist
       if (all (>= 0) refCounts)
          then zipWithM_ updatePage refCounts pagelist
          else return ()
  where
    start = regionStart pr
    pagelist = [start, start+pageSize..regionEnd pr]
    readCount x = do count <- readPageStatus x; return (fromEnum count)
    updatePage refCount paddr
      = updatePageStatus paddr (Mapped (fromIntegral refCount+1))
```

If any page in the region is not mappable, then `incrementMappedCount` will fail, possibly after updating the status for a portion of the region. The caller is responsible for checking that the pages of the region are actually mappable before invoking this function. The analogous function, `decrementMappedCount`, decrements the reference count of every page in a mappable region.

The status manipulation functions defined in this section provide a basis for enforcing memory-safety in the memory-management functions of the interface. As an example, let us consider the definition of `allocPageMap`. We first introduced this function in Section 4.4.1 as the operation that converts a `PageMapPage` into a page-directory.

```
allocPageMap :: PageMapPage -> H (Maybe PageMap)
allocPageMap pmpage
  = do usable <- updatePageStatus page PageMapHandle
       if usable
          then do added <- addHMapping pmpage
                  installed <- installPageMapPage pmpage
                  if added && installed
                     then do liftIO $ c_alloc_pdir (mappedAddr pmpage)
                             return (Just (PageMap page))
```

```
                       else return Nothing
            else return Nothing
    where
      page = regionStart (physicalPage pmpage)
```

The first step in allocating a page-directory is to make sure that converting the supplied page-map page into a page-directory will be safe. We invoke the function `updatePageStatus` for this purpose: if the page-map page is not free, then `updatePageStatus` will return false and we will not proceed with the page-directory allocation. Otherwise, the transition is safe and we update the page status. Once we know that the page-directory can be allocated, we add a mapping for the page-directory to the `HMap` dictionary using `addHMapping`, install a virtual-to-physical mapping for H using `installPageMapPage`, and initialize the page-directory using `c_alloc_pdir`. We return a `PageMap` to the client that contains the physical address of the page-directory, but we hide this representation using a private newtype constructor.

```
  newtype PageMap = PageMap { pdir :: Addr Physical }
```

We use the address, rather than a pointer, because we never access the page-map structure directly from Haskell.

6.6 DIVISION OF CODE BETWEEN C AND HASKELL

We implement the H interface using a combination of Haskell and C code, with a small amount of assembly for accessing the hardware registers. Throughout the implementation, we employ a strategy for dividing the code between C and Haskell that places any potentially unsafe operations in C and any dynamic checking or parameter validation needed for safety in Haskell. As a result, many operations that could be coded in Haskell directly are instead implemented in C. We choose this strategy for simplicity: each portion of the implementation is responsible for maintaining certain structures and invariants and the interface between the two languages is clear. Note that the total amount of the unsafe code is the same regardless of the approach; the techniques that would be necessary to implement the potentially unsafe operations in Haskell have the potential to break type- and memory-safety if used improperly.

In this section, we will demonstrate our approach to partitioning the interface by examining the implementation of the `addMapping` function that we introduced in Section 4.4.2. `addMapping` installs a new virtual-to-physical mapping in user-space, using the C API to modify the underlying translation table structures as necessary. The specifics vary with the nature of the mapping: the C API provides functions for adding page-directory entries and adding page-table entries so that superpages can be used wherever possible. The C API also defines functions related to page-table management; there are primitives for locating an existing page-table, installing a new page-table in a page-directory, uninstalling a page-table from a page-directory, and zeroing a page. Table 6.2 summarizes the name, type, and function of each primitive we will need in the definition of `addMapping`.

The implementation of `addMapping` is relatively complex because of the various safety and correctness conditions the function must handle. The algorithm for adding a mapping includes steps that perform the following tasks:

- **Parameter Validation:** The client specifies the virtual address range to map and the physical region that the address will map to as parameters. The virtual address region must be fully contained in user-space (to avoid mapping over the kernel environment) and the physical region must be mappable in the current state (to avoid giving the user control over a page-table or environment page).

- **Determining the Mapping Type:** The size of the memory region being mapped determines the underlying mechanism that we will use to add the

Function	**Description**
`zero_page` unsigned vaddr	Zero a page of memory. The parameter `vaddr` must be mapped in the current address-space. We use this function before installing a client-supplied page as a page-table or page-directory and before freeing a page to a user.
`find_pt` unsigned vpdir unsigned vaddr	Find the page-table that corresponds to a particular virtual-address. The argument `vpdir` is the address of the page-directory where the page-table should be searched for—this must be a kernel-space virtual address that is mapped in the current page-directory. The function returns the physical address of the underlying page-table, if one exists, and −1 otherwise.
`install_pagetable` unsigned vpdir unsigned vaddr unsigned ptab	Install a newly allocated page-table into the page-directory specified by `vpdir`. We use the virtual address argument, `vaddr`, to calculate the appropriate index within `vpdir` to modify. The function assumes that `vpdir` is mapped and that the entry being modified is not already in use.
`add_table_entries` unsigned vptab unsigned fp unsigned phys unsigned useraccess unsigned perms	Add entries to a page-table. We use this function when mapping less than 4 MB of memory into an address-space. `vptab` is the virtual-address of the page-table being modified; this address must be mapped in the current page-directory. `fp` is a flexpage that describes the virtual region being mapped; `phys` is the first physical address in the mapping, `useraccess` specifies whether or not the mapping will be user-accessible; and `perms` defines the read/write permissions for the mapping. We require that the size of `fp` is less than 4 MB.
`add_dir_entries` unsigned vpdir unsigned fp unsigned phys unsigned useraccess unsigned perms	The analog of `add_table_entries` for adding page-directory entries. We use this function to add superpage mappings when a region of memory larger than 4 MB is being mapped. `vpdir` is the mapped virtual address of the page-directory being modified. The other parameters are the same as `add_table_entries`. We require that the size of `fp` is greater than or equal to 4 MB.

Table 6.2: The C API for managing page-tables and page-directory entries.

mapping: for mappings smaller than 4 MB we use page-tables while for larger mappings we use superpages (this is always possible because of the size and alignment restrictions on flexpages).

- **Page-Table Allocation:** If the mapping will use a page-table, `addMapping` must determine whether or not a page-table is already in place at the appropriate page-directory entry. If a page-table is not in place, then `addMapping` will allocate a new page-table using a page-map page supplied by the client. Our implementation will never allocate more than one page-table because the alignment and size restrictions on flexpages guarantee that superpage entries may be used for any region that cannot be mapped with a single page-table.

- **Updating the Page-Table or Page-Directory:** The `addMapping` function uses the C API to modify a page-table or page-directory as appropriate for the mapping type.

To make the implementation more digestible, we break the definition of `addMapping` into a number of small functions that roughly correspond to these tasks. The rest of the section presents the definitions of these functions.

The first step in the algorithm for adding a mapping is to validate the virtual and physical region parameters supplied by the client. The physical region must be mappable in the current state, meaning that none of the pages contained in the region are a page-table, page-directory, or environment page. The virtual region must be entirely contained in user-space and the two regions must be the same size.

```
validParameters :: PhysicalRegion -> Fpage Virtual -> H Bool
validParameters phys vfp
  = do validTarget <- regionIsMappable phys
       return (validTarget
               && (fpageEnd  vfp < kernelSpace)
               && (fpageSize vfp == regionSize phys))
```

The result of validParameters is true when the parameters satisfy all of the required conditions.

We determine the mapping type by looking at the size of the virtual flexpage being mapped. A size value greater than or equal to 22 bits indicates that the flexpage is large enough to use superpages for the mapping. The rest of the implementation for this case is simple: we simply invoke the C API function for adding directory entries to map the region (c_add_directory_entries is the Haskell wrapper for add_directory_entries—we use the convention of adding a c_ prefix for all of the C API functions).

```
checkMappingType :: Addr Virtual -> PageMapPage -> Fpage Virtual
  -> PhysicalRegion -> H (Maybe Bool)
checkMappingType vpdir pmp vfp phys
  | fpageSize vfp >= 22 =
      do liftIO $
           c_add_directory_entries vpdir fpw (regionStart phys) 1 perms
         return (Just False)
```

Adding a mapping using a page-table is more complicated because we must determine whether or not a page-table already exists for the address range we would like to modify. We use the C API function find_pt to look for an existing page-table. The next step of the algorithm is determined by the result of this function: when a page-table is present we modify the page-table entries straightaway using mapWithoutAllocation; otherwise we allocate a new page-table along the way using mapWithAllocationIfSafe.

```
  | otherwise =
      do pt_addr <- liftIO $ c_find_pt vpdir fpw
         if pt_addr /= (-1)
            then mapWithoutAllocation (fromIntegral pt_addr) vfp phys
            else mapWithAllocationIfSafe vpdir pmp vfp phys
where
  fpw = fpageFromWord vfp
```

We return a Maybe Bool value from checkMappingType that indicates whether or not the mapping succeeded (the Maybe component) and whether or not we needed to allocate a page-table (the Bool component).

The definition for mapWithoutAllocation is a relatively straightforward wrapper for the C function that adds page-table entries. We use a lookup function on the state component of the H monad called getHMappedAddress to determine where the page-table of interest is mapped in the kernel virtual address-space.

```
getHMappedAddress :: Addr Physical -> H (Addr Virtual)
```

After we modify the page-table entries, we update the status for the physical region using the incrementMappedCount from Section 6.5.

```
mapWithoutAllocation :: Addr Physical -> Fpage Virtual -> PhysicalRegion
  -> H (Maybe Bool)
mapWithoutAllocation pt_addr vfp phys =
  do vptab <- getHMappedAddress pt_addr
     liftIO $ c_add_table_entries vptab (fpageFromWord vfp)
                (regionStart phys) 1 perms
     incrementMappedCount phys
     return (Just False)
```

mapWithoutAllocation always succeeds. The result indicates to the caller that we did not need to allocate a new page-table.

The definition of mapWithAllocationIfSafe is further broken up into pieces that handle the extra steps required to allocate a new page-table. This function tries to convert the supplied page-map page into a page-table by setting its status value to Reserved. If this update fails then the page-map page was not free and we cannot use it to allocate a new table.

```
mapWithAllocationIfSafe :: Addr Virtual -> PageMapPage -> Fpage Virtual
  -> PhysicalRegion -> H (Maybe Bool)
mapWithAllocationIfSafe vpdir ptab vfp phys =
  do safe <- updatePageStatus (regionStart (physicalPage ptab)) Reserved
     if safe then mapWithAllocation vpdir vfp phys
             else return Nothing
```

Once we know that the page-map page is safe to use as a page-table, we proceed with the installation process. First we install a mapping to the page in kernel-space and add a link between the physical and virtual addresses in the HMap dictionary.

```
mapWithAllocation :: Addr Virtual -> Fpage Virtual -> PhysicalRegion
  -> H (Maybe Bool)
mapWithAllocation vpdir =
  do success <- installPageMapPage ptab
     success' <- addHMapping ptab
     if success && success'
        then do liftIO $ allocateAndModifyTable vpdir vfp phys
                incrementMappedCount phys
                return (Just True)
        else return Nothing
```

If adding the kernel-space mapping for the page-table succeeds, then we proceed with the actual installation of the page-table into the appropriate page-directory and add the entries that represent the new mapping.

```
allocateAndModifyTable :: Addr Virtual -> Fpage Virtual
  -> PhysicalRegion -> IO ()
allocateAndModifyTable vpdir vfp phys =
  do c_zero_page (mappedAddr ptab)
     c_install_pagetable vpdir fpw (regionStart (physicalPage ptab))
     c_add_table_entries (mappedAddr ptab) (fpageToWord vfp)
       (regionStart phys) 1 perms
```

As with mapWithoutAllocation, we increment the reference count for the pages of the physical region after the mapping has been successfully installed.

The top-level definition of addMapping kicks off the process by checking the parameters and invoking the checkMappingType function.

```
addMapping :: [PageMapPage] -> PageMap -> Fpage Virtual
  -> PhysicalRegion -> Perms -> H (Maybe Bool)
addMapping [] _ _ _ _ = return Nothing
addMapping (ptab:_) (PageMap pdir) vfp phys perms =
  do parametersok <- validParameters phys ptab vfp
     if parametersok
        then do vpdir <- getHMappedAddress pdir
                checkMappingType vpdir
        else return Nothing
```

The equations demonstrate an additional constraint on the parameters of the `addMapping` function: we require that the client provides at least one potential page-table page, regardless of the type of mapping that will be done. Future implementations of the interface could relax this constraint.

The definition of `addMapping` demonstrates the algorithm that we use to add virtual-to-physical mappings in a safe way. All of the safety checking and validation is done in Haskell, while all of the manipulations of the translation table data structures are done in C. The implementations of our other mapping functions are very similar so we omit their definitions here. All of the functions of the interface use the same basic pattern where safety critical checks are performed in Haskell while potentially unsafe computations are performed in C.

6.7 THE KERNEL VIRTUAL-ADDRESS SPACE

As discussed in Section 4.4.3, H allows the client kernel to add kernel mappings that are visible in every virtual address-space. The client only has access to a portion of the kernel virtual-address space—the rest is reserved by H for the execution environment and other H data. In this section, we discuss the techniques that we use to implement kernel mappings and the operations available to the client for modifying kernel-space.

The start-up code partitions the kernel virtual-address space into a client-controlled area and an H-controlled area based on the amount of memory that we need to keep mapped in H. We report this division of kernel-space to the client using the same techniques that we employed for lifting the initial regions through the interface in Section 6.3: we define an array in C containing the virtual flex-pages that belong to the client-controlled area and convert the array into a Haskell list using a small wrapper function. The client reads the list of kernel-mappable virtual addresses to determine which kernel-space address it is allowed to map.

```
kernelMappableVirtualAddresses :: H [Fpage Virtual]
```

We also define utilities that test whether or not a particular address falls in the set of client-controlled addresses. We use this function to validate virtual-address values passed to H as parameters; any function that adds a mapping in kernel-space must check that the mapping will fall in the appropriate range.

```
isKernelMappableVirtualAddress :: Addr Virtual -> H Bool
```

The addresses in the client-controlled area may be used for any purpose. The client adds a mapping to one of these addresses using an `addKernelMapping` function that behaves similarly to the function for adding a user mapping.

Allowing the client to add mappings in kernel-space increases the difficulty of maintaining a consistent view of kernel memory in every address-space. Consistency between address-spaces is a critical property for the H-controlled portion of kernel-space (otherwise we might enter an address-space where the Haskell heap is not mapped, for example), but consistency for client-controlled mappings is arguably less important. We considered a design that only added new kernel mappings to the current address space, requiring the client kernel to add the mapping in other address-spaces explicitly. Ultimately, we decided against this design because we did not want to allow page faults to happen in the kernel.

To enforce consistency for client-controlled mappings, we need a mechanism for modifying all of the page-directories at once. Explicitly modifying every page-directory in the system every time the client adds a kernel mapping is possible, but may not be practical because of the additional per-mapping overhead. Instead, we use an implementation technique that guarantees that new mappings will be visible in every address-space with a single update. The trick is to ensure that the kernel portion of each page-directory is immutable and identical. When we allocate a new address-space, it gets initialized with a copy of the values from a reference page-directory. Within the reference page directory, the entries that correspond to H-controlled addresses are mapped to kernel code and data. The client-controlled addresses are mapped to page-tables that we pre-allocate during start-up. Because every address-space uses the same page-tables to map the same set of addresses, updates to page-table entries will automatically be seen in every address-space. Figure 6.6 illustrates this configuration. We avoid the need to update page-directories individually, but we pay a cost in (potentially wasted) space by allocating the page-tables for kernel-space before they are needed.

The implementation of `addKernelMapping` is shown in Figure 6.7. Adding a kernel-space mapping is similar to adding a user-space mapping, except that we will never need to allocate a page-table and we will never map a region using superpage entries. The algorithm contains the following steps:

- **Parameter Validation and Region Splitting:** Only kernel-space addresses that are in the client-controlled portion of kernel-space may be mapped using `addKernelMapping`. The physical region to which the virtual address range will be mapped must be mappable (no page currently in use as a page-table, page-directory, or environment page), and the length of the physical and virtual regions must be the same. For regions that are larger than 4 MB, `addKernelMapping` splits the region into a collection of small regions that are all less than or equal to 4 MB in size.

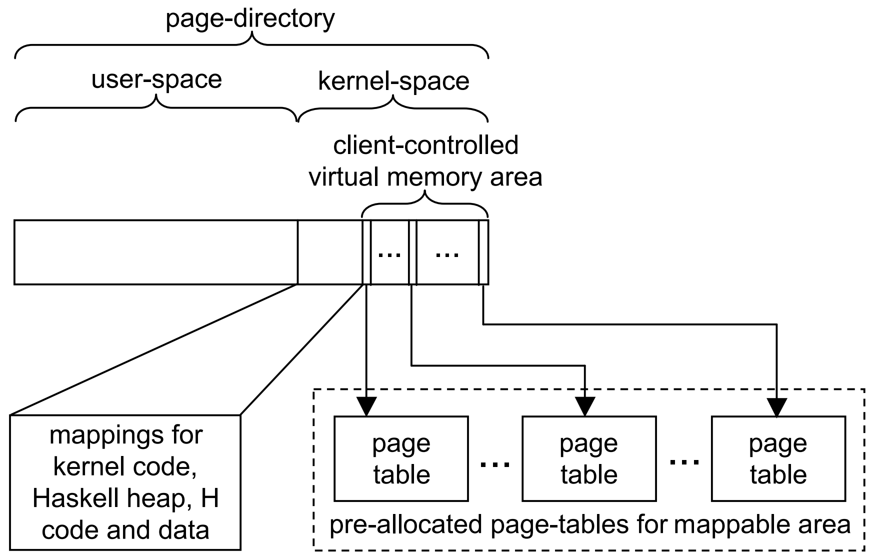


Figure 6.6: Implementing a consistent view of kernel-space across every address-space. The mappings in the H-controlled portion of kernel-space (all kernel-space addresses not in the client-controlled area) do not change over time, so we can simply install these entries in every new page-directory that we create. The mappings for client-controlled addresses may change, but they will always be mapped via the same set of page-tables. We initialize a page-directory by installing the page-table addresses for the shared kernel-space tables. Thus, when any of the page-table entries is modified, the effect will be seen in every address-space.

- **Adding Page-Table Entries For Each Region:** We use the C API function for adding new page-table entries (called `c_add_table_entries`, see Section 6.6) to add the new mappings. We use the reference page-directory to locate the appropriate page-tables to modify for simplicity, though we could copy the appropriate mappings from the current page directory.

```
addKernelMapping :: PhysicalRegion -> Fpage Virtual -> Perms -> Bool -> H (Maybe KernelMapping)
addKernelMapping phys vfp perms user =
  do validTarget <- regionIsMappable phys
     mappablefp <- isKernelMappableVirtualFlexpage vfp
     if validTarget && mappablefp && (fpageSize vfp == regionSize phys)
         && length mrs == length rs
        then do vpdir <- systemPageMapMappedAddress
                performMappings vpdir rs (split vfp)
        else return Nothing
 where userbit = if user then 1 else 0
       mrs = map (deriveRegion phys) (split (region phys))
       rs  = catMaybes mrs

 performMappings vpdir regions vfpages =
   do pt_addrs <- liftIO $ mapM (c_find_pt vpdir) fpws
      zipWithM3_ addKernelMappings pt_addrs regions fpws
      return (Just kmapping)
   where fpws = map fpageToWord vfpages
         kmapping = KernelMapping { kernelFpage   = vfp,
                                    kernelRegion  = phys,
                                    kernelPerms   = perms,
                                    kernelAddress = nullPtr `plusPtr` fpageStart vfp
                                  }

 addKernelMappings pt_addr phys fpw =
   if pt_addr == (-1) then error "missing kernel page-table"
      else let vptab = fromIntegral pt_addr + kernelSpace
           in  liftIO $ c_add_table_entries vptab fpw (regionStart phys) userbit perms
```

Figure 6.7: The implementation of the `addKernelMapping` function. The algorithm is similar to `addMapping`, but we avoid the complexity of page-table allocation because we pre-allocate all of the tables for kernel-space during start-up. Unlike `addMapping`, `addKernelMapping` may only map regions using page-tables, so the implementation splits large regions into a collection of 4 MB-sized regions before invoking the standard page-table modification function (using a function for splitting flexpages called `split`).

- **Constructing a `KernelMapping` Handle:** If the mapping operation is successful, we provide the client kernel with a handle to the memory through which the kernel-mapping can be read and written. The handle, represented with the type `KernelMapping`, contains all of the information that is necessary for H to check the validity of future memory accesses, as well as a pointer for actually performing the reads and writes. Neither the pointer nor the constructor for the type are observable to the client.

We do not implement a remove operation on kernel-mappings because of our policy that kernel code will not fault. We have no way to revoke or invalidate the kernel-mapping handles owned by the client, so there would be no way to protect memory accesses if the client could remove kernel mappings. Relaxing the requirement that the kernel does not fault or implementing a more elaborate capability system are both valid options if there turned out to be a strong use-case for kernel-mapping removal.

6.8 USER-PROGRAM EXECUTION

From the perspective of an H-based kernel, executing a user-program is conceptually similar to calling any other H function. The client invokes the `execute` operation from the H interface, supplying a description of the program to be run, and at some point later H returns to the client with a description of the interrupt that brought us back into kernel-mode. The implementation of `execute` maintains the illusion of simplicity: all of the complexity is handled by an assembly-language routine that we call when we wish to start a user program. This routine does not return to Haskell until the user-program has executed and produced an interrupt, but to the Haskell code, the assembly function behaves the same as any other FFI call.

Figure 6.8 illustrates the relationship between each of the components in our

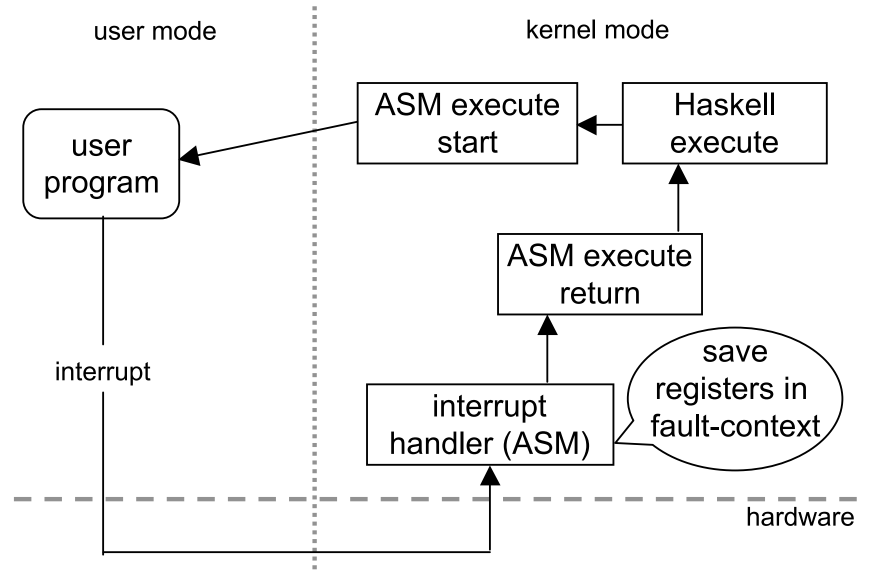


Figure 6.8: The H mechanism for executing a user program. We split the definition of the H function `execute` into Haskell and assembly-language portions. The assembly-language function is in turn split into a "start" component that performs the context switch to user code and a "return" component that resumes Haskell execution as if we were returning from a normal function. An interrupt handler written in assembly is responsible for saving the register state of the user-program before returning to Haskell.

user-program execution implementation. The Haskell function `execute` initiates the process by calling our execution function written in assembly, which in turn context-switches to the user code. When an interrupt occurs, we enter an assembly-language handler in kernel-mode that saves the register state of the program and returns to Haskell via a special function that behaves as if we were returning from a normal call. In the rest of this section, we will explore each of the components that make up user-program execution in more depth.

The assembly function takes two parameters: a pointer to the program's saved register state and the physical address of the page-directory to install. The FFI does not allow us to call assembly functions directly, so we invoke the function via a C wrapper.

```
extern unsigned asm_execute(unsigned pmap, unsigned* fault_context);

foreign import ccall unsafe "execution.h asm_execute"
  c_execute :: HWord -> Ptr HWord -> IO HByte
```

Within `asm_execute`, we save the kernel's register state, set the new page-directory, and restore the register state of the user program. Once the proper user state is in place, we issue the `iretl` instruction to return to the user program at the point where it was interrupted.

The `FaultContext` type introduced in Section 4.5 encapsulates our representation of the register state for user programs. We save the register state of user programs in blocks of memory that we allocate from the Haskell heap and access via a pointer[3].

```
newtype FaultContext = FaultContext (Ptr HWord)
```

The client cannot observe the internal representation of `FaultContext`s, but may read or write the value of individual registers using the H functions `readRegister` and `writeRegister` (as explained in Section 4.5).

When we execute a user process, we install a pointer to the appropriate fault-context in the stack pointer field (called ESP0) of the task-state-segment (see Section 2.1.3). When we switch to kernel-mode upon receiving an interrupt, the

[3] We choose to allocate `FaultContext`s from the heap so that `FaultContext` values can be garbage collected. We perform the allocation using the function `mallocForeignPtrBytes` to avoid the need for a finalizer—the run-time system will collect the fault-context memory when the pointer is no longer in use.

hardware installs this pointer as the active stack pointer in the ESP register. This allows us to save the registers of the running program by pushing the values onto the current stack in the interrupt handler. When we resume the user program after servicing an interrupt, we pop the saved values off the stack into the registers and the user state is restored. The client has full control over how many stacks H will use. For example, the client can allocate a single fault-context on which all user programs will run or the client can allocate a distinct fault-context for every user thread.

The H function `execute` is essentially a Haskell wrapper for the `c_execute` function that sets up all of the user state and returns from the interrupt. The client kernel provides `execute` with a `PageMap` specifying the page-directory to install for the user program and a `FaultContext` containing the program's register state. `execute` sets the ESP0 field of the task-state-segment to point to the fault-context and then invokes `c_execute` to return to the user program.

```
execute :: PageMap -> FaultContext -> H Interrupt
execute (PageMap pm) fc@(FaultContext fcPtr)
  = do liftIO $ pokeByteOff c_tss_esp0 0 (fcPtr ‘plusPtr‘ fcLen)
       vector <- liftIO $ c_execute pm fcPtr
       code <- readRegister fc Code
       faultAddr <- liftIO $ peek c_last_fault_addr
       return (vectorToInterrupt vector code faultAddr)
  where
    fcLen = 76
```

When `c_execute` returns, we know that the user program has executed and produced an interrupt described by `vector`. We return a high-level description of the interrupt that occurred by converting the numeric vector into a value of the `Interrupt` type.

```
data Interrupt
    = DivideError
    | NMIInterrupt
    | ExternalInterrupt IRQ
    | ProgrammedException HByte
      ...
```

The Interrupt type contains a constructor for representing each of the possible results from c_execute. We omit most of the definition because it is a straightforward encoding of the semantics of the IA32 interrupt vectors.

6.9 SUMMARY

In this chapter, we outlined the techniques employed for implementing the abstraction layer, with a particular focus on the mechanisms that allow us to run Haskell code on bare metal, to run user-level C programs, and to enforce memory-safety in the H operations. An important aspect of the implementation that we have not covered is its size: Table 6.3 shows the source lines of code in the abstraction layer, broken down by language. The total size of the interface is about 3000 lines of code, with more than half of code written in C. We have not made a significant attempt to optimize the implementations of the H interface operations, so it is possible that the size could be decreased with additional implementation effort.

Purpose	**Language**	**Source Lines of Code**
Implementation	Haskell	1096
Implementation	C	1633
Implementation	ASM	491
Interface	Haskell	241

Table 6.3: The source lines of code for H in Haskell, C, and Assembly Language. We count the interface code (special modules to be imported by clients that include a restricted set of module exports and the H type class instances) separately from the implementation code.

Chapter 7

CASE STUDY: INTER-PROCESS COMMUNICATION IN L4

The design of the H interface is only successful if the resulting API is sufficiently expressive to support the construction of operating systems in purely functional languages. In this chapter, we demonstrate the expressiveness of our abstraction layer by showing how it can be used to implement a version of the L4 microkernel API [62] in Haskell using the operations of the H interface. L4 is a second-generation microkernel with several designs and implementations [70, 57, 75, 84]. The wealth of existing implementations demonstrates that L4 is mature and popular enough to make it a "real world" system, and supplies us with a baseline for evaluating the performance of our architecture.

An essential characteristic of L4 is that the kernel only includes features that absolutely cannot reside at user-level for functional or security reasons [70]. Many features that one typically expects from an operating system, such as device drivers and memory management, are instead implemented outside of the kernel as user-level servers [38]. To support user-level memory management, L4 must provide a sophisticated set of virtual memory management primitives as part of the kernel API. The L4 virtual memory API is sufficiently expressive to support a wide range of user-level applications, including a port of Linux that runs as a user-level server [44] and a platform for secure application execution [30] with GUI support [23]. If the primitives of H are sufficiently general to implement the L4 API, then we have reason to be confident that H could also support the implementation of other systems.

We successfully implemented the major system calls of L4 using the H interface.

The implementation is based on the X2 API [62]. Our implementation includes address space management, thread management and scheduling, message- and memory-based inter-process communication, and the start-up protocols for threads and address spaces. Though we do not implement every feature described in the API, our kernel is sufficiently full-featured to allow us to run a realistic L4 program. We think that this set of functionality sufficiently demonstrates the utility of our abstraction layer and that a fully API-compliant version of L4 could be completed as future work with additional engineering effort.

In this chapter we will present our implementation of the L4 inter-process communication (IPC) system call. We choose to focus on this aspect of L4 because it is a central service provided by the operating system in a microkernel architecture, and has long been used as a benchmark for evaluating the viability of microkernel designs and implementations. The original focus of L4 was to demonstrate that good performance was possible in a microkernel. IPC performance was the primary target [44, 70, 71]. As a result, many modern L4 implementations of IPC are finely tuned. L4 provided a contrast to earlier microkernel designs such as Mach [2] whose IPC performance was sufficiently poor to hinder their adoption. Over time, security and separation issues in IPC have become increasingly important in the L4 community as well, leading to secure variants of the original design such as seL4 [84] and L4.sec [59, 25]. Haskell has previously been used as a prototyping tool in the design of an L4 kernel, but never as the implementation language due to performance concerns [18].

Our presentation of IPC includes an explanation of many fundamental concepts of the L4 design, as necessary to explain the algorithms and data-structures that appear in the IPC code. Section 7.1 discusses threads and address spaces. Section 7.2 introduces the basic concepts of IPC and the data-structures that we use to represent messages. Section 7.3 discusses an L4-specific data-structure for managing memory mappings. Section 7.4 describes the monad in which our kernel runs. Section 7.5 explains how we handle errors that come up during IPC processing. The remaining sections cover the IPC algorithm itself. Section 7.6 presents the rendezvous component of IPC: connecting a sender to a willing receiver. Section 7.7 outlines the process for transferring data from the sender to the receiver. The performance of our L4 kernel will be covered in Chapter 8.

7.1 THREADS

Threads are an important abstraction in L4. Each thread is associated with an address space that provides the resource context in which the thread will execute. The Thread type describes the important features of a thread, such as the address-space in which the thread executes, called its parent.

```
data Thread = Thread {
    parent    :: HWord,        -- Domain owning this thread
    threadId  :: ThreadId,     -- global identifier
    halted    :: Bool,         -- currently halted?
    scheduler :: ThreadId,     -- scheduler thread's ID
    priority  :: HWord,        -- priority (set by scheduler)
    timeleft  :: HWord,        -- time left to run
    timeslice :: Timeslice,    -- time to run per time scheduled
    quantum   :: Quantum,      -- total time to execute
    context   :: FaultContext, -- saved register state
    waiting   :: [ThreadId],   -- threads sending to this
    status    :: ThreadStatus, -- ready/waiting/etc
    utcb      :: UTCB          -- thread control block
  }
```

The threadId is a unique global identifier for the thread. halted, scheduler, priority, timeleft, timeslice, and quantum are parameters related to thread scheduling that we ignore in our discussion of IPC. A thread's context contains the register state from the last time the thread was suspended (see Section 4.5). The waiting queue stores the identifiers of threads that are waiting to send a message to this thread; we will cover this topic in Section 7.6. The `status` field contains information about the execution state of the thread, for example, whether it is runnable, inactive, or blocked waiting to send an IPC message. `utcb` stores the location of the thread's user-level thread control block (UTCB).

Every thread in the system has a UTCB, which is a structure in memory that is accessible to both the thread and the kernel. The thread uses the UTCB to communicate system call parameters to the kernel—such as the data to be transferred by an IPC—and the kernel uses the UTCB to pass results back to the thread—such as an error code or the incoming data from an incoming message. The UTCB also contains configuration information about the thread, such as its pager, interrupt handler, and global identifier. We define a datatype where each constructor represents a field in the UTCB[1].

```
data UTCBField
  = MyGlobalId        | ProcessorNo       | UserDefinedHandle
  | Pager             | ExceptionHandler  | COPPreemptFlags
  | ErrorCode         | XferTimeouts      | IntendedReceiver
  | ActualSender      | ThreadWord0       | ThreadWord1
  | MR1  | MR2  | MR3  | MR4  | MR5  | MR6  | MR7  | MR8  | MR9  | MR10
  | MR11 | MR12 | MR13 | MR14 | MR15 | MR16 | MR17 | MR18 | MR19 | MR20
  | MR21 | MR22 | MR23 | MR24 | MR25 | MR26 | MR27 | MR28 | MR29 | MR30
  | MR31 | MR32 | MR33 | MR34 | MR35 | MR36 | MR37 | MR38 | MR39 | MR40
  | MR41 | MR42 | MR43 | MR44 | MR45 | MR46 | MR47 | MR48 | MR49 | MR50
  | MR51 | MR52 | MR53 | MR54 | MR55 | MR56 | MR57 | MR58 | MR59 | MR60
  | MR61 | MR62 | MR63
  | BR0
  | InvalidField
```

The collection of fields that begin with `MR` are called *message registers*. As we will see in Section 7.2, the message registers are the locations that are used for

[1] We do not implement the buffer register fields of the UTCB because we do not implement the string item functionality of the L4 API.

data transfer during an IPC message. BR0 is a buffer register that stores an IPC parameter called the acceptor. We access the fields of a UTCB using the functions readUTCBField and writeUTCBField.

```
readUTCBField :: (KernelMemory m) => UTCB -> UTCBField -> m HWord
writeUTCBField :: (KernelMemory m) => UTCB -> UTCBField -> HWord -> m ()
```

These functions provide direct access to the memory that stores each UTCB using the KernelMemory portion of the H API (see Sections 6.2 and 4.4.3).

Essentially, each address space consists of a PageMap (see Section 4.4.1) that describes the memory mappings available in that space. Internally, we track some additional information about each address space in a data-structure called a Domain.

```
data Domain = Domain {
                space          :: PageMap,
                domainId       :: HWord,
                privileged     :: Bool,
                numThreads     :: Int,
                numActive      :: Int
              }
```

The space field contains the page-map for the address-space. domainId is a unique identifier for the space. We include this feature for implementation purposes—address spaces do not have global identifiers that are visible through the L4 API. Instead, address spaces are referred to by a *space specifier*, the thread identifier of a thread associated with the address space of interest. A privileged address space has extra rights in an L4-based system, for example, certain system calls like thread creation may only be performed by a thread in a privileged address space. The number of threads, numThreads, tracks the count of threads whose parent is this domain. Some of these threads may be inactive, so we separately track the number of active threads, numActive. The kernel frees a domain when it no longer contains any threads.

7.2 IPC MESSAGES

Threads communicate using the IPC system call of the L4 API. IPC in L4 is synchronous, so both the sender and the receiver must invoke the IPC system call before a message transfer can proceed. There are two potential phases to every IPC operation: a send and a receive. Any given IPC request may include just a send, just a receive, or both. One of the arguments to IPC is the identifier of the thread the caller wishes to communicate with. When a thread is sending an IPC, it must specify the identifier of the intended receiver. Receiving threads have more leeway. A receiver may indicate a specific sender in its request using the intended sender's identifier or the receiver may specify one of two special identifiers:

- **anythread** is a thread identifier that matches any thread in the system.
- **anylocalthread** is a thread identifier that matches any thread in the same address space as the receiver.

There is a third constant of the thread identifier called *nilthread* that does not match any thread. One use of nilthread is to indicate to the kernel which phases of IPC a particular call should include, for example, setting the receiver argument to nilthread requests an IPC with no send phase.

The IPC mechanism allows for both direct data transfer and memory sharing. In a typical L4-based system, a user-level pager controls all non-kernel memory and acts as the memory manager for the rest of the system. This pager makes use of the memory sharing facilities of IPC to map memory into other address spaces. I/O and interrupts are handled by user-level servers as well; the kernel merely provides support for getting the information to the appropriate handler thread. When an interrupt occurs, the kernel catches the event and creates an IPC message on behalf of the faulting thread. The message is sent to the appropriate handler for the interrupt type, which might be different from thread to thread. The message contains data that describes the fault or event, but unlike normal messages, the data is passed directly in the message and is not read from the message registers of the sending thread. When the handler thread is done processing the event, the handler sends a message back to the faulting thread, which is caught by the kernel. The kernel completes the processing of the interrupt, for example, by restarting the faulting thread.

Each of the special message types has different processing requirements in the kernel. The L4 manual defines a protocol that characterizes the interactions between the kernel (acting on behalf of the faulting thread) and the user-level fault handler. We introduce a datatype in our L4 implementation to represent the different kinds of messages that may be sent.

```
data SendType      = FromMRs
                   | SInterrupt
                   | SPageFault HWord HWord Perms
                   | SException HWord HWord
                   | SPreempt   Clock
```

A message sent `FromMRs` is a normal message. (MR is an abbreviation for message register.) Each thread in L4 has 64 memory-mapped message registers that the thread may use to pass data in an IPC message. During the course of a normal IPC operation, the data being sent is transferred from the message registers of the sending thread to the message registers of the receiver. An `SInterrupt` message is the message the kernel sends when an I/O interrupt occurs. There is one interrupt handler thread per I/O interrupt, so no extra information about the nature of the interrupt is needed. The interrupt is disabled until the kernel receives a response from the handler thread. An `SPageFault` message notifies the memory manager of a thread that the thread page faulted. The kernel sends the pager the address where the fault occurred, the instruction pointer of the faulting thread, and the type of fault (read or write). `SException` alerts a thread's exception handler. The message carries the interrupt vector of the exception and the error code. The final message type is SPreempt, which notifies the scheduler of a thread that the thread has been preempted because it exhausted its available time to run. The message contains the time at which the preemption occurred.

We also define a type to describe the kinds of messages that may be received. Most of the receive messages have an analog in SendType.

```
data ReceiveType  = ToMRs        -- normal IPC message
                  | RInterrupt   -- interrupt response message
                  | RPageFault   -- page fault response message
                  | RException   -- exception response message
                  | RStartup     -- IP/SP of a new thread for start up
```

ToMRs describes a normal message where the payload of the message will be transferred into the message registers of the receiver. RPageFault, RInterrupt, and RException are the replies to SPageFault, SInterrupt, and SException, respectively. The RStartup message is sent to a thread by its pager when the thread begins executing. The start-up message contains the initial stack pointer and instruction pointer values for the receiving thread. The kernel uses these values to set the appropriate registers of the receiver.

The message registers contain the payload of an IPC message for both direct data transfer messages and memory sharing requests. In the case of a direct data transfer, the contents of the message registers are untyped from the perspective of the kernel—it simply moves the data from the sender to the receiver. In the case of a memory sharing request, the message registers contain descriptions of the mapping operation to be performed. The kernel must process these *typed items* and modify the address space of the receiver as appropriate for the request before copying the descriptions into the receiver's message registers.

There are two data structures in L4 that describe memory sharing requests: map items and grant items[2]. A thread requests to share memory with another thread by sending an IPC message with one or more of these data structures stored in its message registers. A map item requires two message registers for storage and contains: a flexpage that describes the region to be mapped, the permissions to attach to the memory when it is mapped in the address space of the receiving thread, and a *send base* parameter that is used to reconcile any differences between the amount of memory that the sender is trying to map and the amount of memory that the receiver is willing to receive. The receiver specifies the window in which to add the new mappings in a parameter called the *acceptor*. The acceptor is a flexpage that indicates the starting address where mappings will be placed as well as a maximum amount of memory to map. Grant items are very similar to map items but have different semantics. When a thread sends a grant item to another thread it gives up its own rights to the memory being sent during the transfer.

The function `reconcileFpages` calculates the actual send window and receive window to use for a memory sharing operation based on the value of the map or grant item, the acceptor of the receiver, and the send base specified by the sender.

```
reconcileFpages :: HWord -> HWord -> HWord -> (L4Fpage, L4Fpage)
```

We introduce the type `L4Fpage` to describe the send and receive windows. In L4, a flexpage is conceptually similar to the flexpages described in H, except that an L4 flexpage also includes a set of permissions to attach to the region of memory.

```
data L4Fpage = L4Fpage { vflexpage :: Fpage Virtual, perms :: Perms }
```

The result of `reconcileFpages` is a new source region and a new destination region that are guaranteed to have the same size. If no reconciliation is possible

[2] The API describes a third request type for copying large blocks of data called a string item, but we do not implement this functionality.

(for example, because the receiver is not willing to accept any memory mappings), then this guarantee is satisfied by returning nilpage as both the send and receive window.

The kernel reads the acceptor and the message data parameters to IPC from thread UTCBs. The other parameters to the system call—the intended IPC partner, the amount of time the caller is willing to wait for the operation to complete, and the type of the message (its SendType or RecvType)—are packaged up into an IPCType value that describes the request being made. In some cases these parameters will be read from registers when the IPC system call interrupt is received by the kernel, while in other cases the message will be created by the kernel in response to other events, such as page faults.

```
data IPCType = Sending   { partner    :: ThreadId,
                           timeout    :: Timeout,
                           stype      :: SendType }
             | Receiving { partner    :: ThreadId,
                           timeout    :: Timeout,
                           rtype      :: ReceiveType }
```

In our implementation, we only allow two possible timeout values: zero and infinity (any non-zero value). A zero timeout indicates that the thread is not willing to block waiting for the IPC; the system call will return if the intended partner is not ready. An infinite timeout indicates that the thread will block indefinitely; when the intended partner makes a matching request the thread will wake up. The L4 API allows for clock-triggered timeouts where the kernel wakes up a blocked thread after a certain amount of time, but our current implementation does not support this aspect of the L4 API.

An IPCType value captures basic information about an IPC message, whether that IPC occurs immediately following a system call request or at some point later when the partner for the operation makes their IPC request. The status field of the Thread data structure indicates the current state of the thread: blocked waiting for an IPC, ready to run, halted, or inactive. When an IPC operation gets blocked because the partner is not ready, the kernel saves the `IPCType` for later use. We encode these possible states with the `ThreadStatus` type.

```
data ThreadStatus = Runnable
                  | Blocked IPCType
                  | Inactive
                  | Halted
```

The parameter of the `Blocked` constructor describes the pending IPC, whereas `Runnable`, `Inactive` and `Halted` threads do not need any additional context.

7.3 THE MAPPING DATABASE

The *mapping database* is an L4-specific kernel data structure that records the relationships between memory mappings that are shared by user threads. This structure helps us to manage the relatively complex address-space interactions that are possible using the L4 API. For example, a thread running in an address space may map memory to another address-space that may in turn share the mapping with additional parties. At some later point, the original thread may wish to revoke access to the memory from the other address spaces, including the derived mappings that it has no direct knowledge of. The mapping database is a tree that captures all of the relevant information for performing such a revocation. Revocation corresponds to the L4 system call `unmap` but is also necessary during IPC because the API allows threads to add a new mapping in an area of virtual memory that is already mapped (implicitly removing the original mapping first).

We represent the mapping database as a list of trees. Each tree describes a memory mapping—the physical area occupied by the mapping, the address-space to which the memory is mapped, the virtual address region where the mapping lies, and the permissions with which the memory is mapped—as well as any mappings that have been derived from it through sharing. The tree datatype, `MapTree`, is a record containing a field for each of these pieces of information.

```
type MappingDB = [MapTree]

data MapTree = MapTree {
                 mtregion   :: PhysicalRegion,
                 ownerSpace :: PageMap,
                 varea      :: Fpage Virtual,
                 mtperms    :: Perms,
                 children   :: [MapTree]
               }
```

The elements of the top-level list of trees correspond to the regions of physical memory that the kernel maps into user-space directly, rather than as the result of a sharing request. In L4, all free memory is initially mapped to a privileged memory manager address-space called σ_0 by the kernel. The children of a tree identify mappings of a region from one user-process to another.

There are three basic operations on the mapping database, which correspond to the three system call requests that user threads may use to affect memory mappings:

- **shareMapping:** Sharing a mapping is used to handle a map item request through IPC—it maps a specified region of memory to some target address-space without modifying the source address-space. In terms of the mapping database, sharing a mapping corresponds to adding a new entry to the children of the source mapping.

- **replaceMapping:** Replacing a mapping is used to handle a grant item request through IPC—it maps a specified region of memory to some target address-space and removes it from the source address-space. In terms of the mapping database, replacing a mapping corresponds to transferring control of a node in the tree, including of the children, to a new address space.

- **restrictMapping:** Restricting a mapping reduces the permissions that are attached to a particular region of memory, possibly to zero (unmapping the memory). The `unmap` system call relies on this functionality. In terms of the mapping database, restricting a mapping modifies the permissions on an existing node and its children.

Each of these operations is predicated on the assumption that the source address-space owns the memory that corresponds to the mappings that are being manipulated. For example, to map the page at virtual-address x from address-space A to B, there must be a node in the mapping database with A as the owner space and page x as a sub-region of the virtual area. If a requested mapping database operation is valid, then the mapping database operations update both the page-maps of the address-spaces (using the H API) and the database as appropriate. In the remainder of this section we will examine the types and behavior of these functions individually.

In addition to the three operations that correspond directly to L4 system calls, we introduce a fourth operation for initializing the mapping database. We install the kernel-to-user virtual-memory mappings (those regions of memory directly mapped to a user address-space from the kernel) with the function `mapRegion`. Without this step, there would be no user-level mappings available for threads to share. The primary use for `mapRegion` is to set up the initial state of $sigma_0$, which receives mappings to all available physical memory during the kernel initialization process. Because the memory mapped by `mapRegion` passes directly from the kernel to a user, the mapping does not need to be derivable from an existing user-space mapping. In fact, the opposite constraint holds: `mapRegion` will only respect the semantics of the mapping database if the memory being mapped does not overlap any existing mappings in the database.

The type of `mapRegion` reflects the fact that the operation potentially modifies the mapping database structure as well as the user-level portion of an address-space (recall that the UserMemory type class encapsulates user-level mapping functions from H). The mapping database is accessed within the implementation of mapRegion through a state component that is present in the L4 kernel monad (see Section 7.4 for more details).

```
mapRegion :: (StateMonad MappingDB m, UserMemory m)
  => [PageMapPage] -> PageMap -> Fpage Virtual -> PhysicalRegion
     -> Perms -> m (Maybe Bool)
```

The arguments to mapRegion are: a list of page-map pages to use for page-table allocation (if necessary), the page-map to modify, a virtual-flexpage that describes where to install the mapping, a region that corresponds to the physical memory to be mapped, and the permissions to attach to the mapping. The result indicates whether or not the operation completed successfully. We omit the code for this (and the other mapping database operations) for brevity.

shareMapping and replaceMapping each add a memory mapping to a target address-space that is derived from an existing mapping in a source address-space. The only difference is in the effect that the operations have on the source address-space when adding the mapping. As such, the type signatures for the two functions appear very similar.

```
shareMapping ::
 (StateMonad MappingDB m, StateMonad [PhysicalRegion] m, UserMemory m)
  => [PageMapPage] -> PageMap -> L4Fpage -> PageMap -> L4Fpage
      -> m (Maybe Bool)

replaceMapping ::
 (StateMonad MappingDB m, StateMonad [PhysicalRegion] m, UserMemory m)
  => [PageMapPage] -> PageMap -> L4Fpage -> PageMap -> L4Fpage
      -> m (Maybe Bool)
```

As with mapRegion, the first argument to both functions is a list of pages that

H may use for page-tables as necessary. The first page-map and `L4Fpage` describe the source of the mapping and the second describe the target. The implementation searches the mapping database for a tree whose owner space matches the source page-map and whose virtual area contains the requested range in the `L4Fpage`. The permissions are automatically downgraded if the operation attempts to share/replace a mapping with greater permissions than the source itself has. The result indicates whether or not the operation succeeded. Failure stems from an invalid source mapping or a list of potential page-tables that is not sufficient to complete the operation. Any existing mappings in the target virtual area will be implicitly removed by the share/replace operation.

The function `restrictMapping` modifies the permissions attached to an existing mapping, including all of the derived mappings. Reducing the permissions on a mapping to nothing removes that mapping. The permission restriction may be applied to the source mapping and all derived mappings, or just the derived mappings while leaving the source mapping alone.

```
restrictMapping ::
  (StateMonad MappingDB m, StateMonad [PhysicalRegion] m, UserMemory m)
   => PageMap -> Fpage Virtual -> Perms -> Bool -> m MappingInfo
```

The type for `restrictMapping` is similar to the operations so far: the page-map specifies the address-space, the virtual flexpage specifies the area to modify, and the permissions value specifies the new permissions. The additional components of the type are the Boolean value to control source mapping restriction and the `MappingInfo` result which describes the current state of the accessed and dirty bits of the memory region.

7.4 MANAGING STATE WITH THE KERNEL MONAD

Our L4 kernel runs in an extension of the H monad called `Kernel`. This monad captures the side effecting behaviors that are specific to L4 and not provided by

H. These behaviors could include any side effects that are available in Haskell, but in our implementation we only use the state monad. We introduce several state components that correspond to key kernel data structures and add them onto the H monad using the monad transformer for the state monad.

7.4.1 Kernel Memory Allocator

The foundation of the Kernel monad is an allocator that tracks the free memory available to the kernel. There are two components to the allocator monad: a list of virtual addresses and a list of page-sized physical regions. The virtual addresses represent locations in kernel-space where we are allowed to add mappings. The initial state is derived from the kernelMappableVirtualAddresses constant in H. The physical regions represent free pages of physical memory. This pool is initialized using a portion of the physical memory given to the kernel by the initialRegions function.

```
type Allocator = ST [Addr Virtual] (ST [PhysicalRegion] H)
```

We define wrapper functions on the standard state monad get and set operations (which read and write to the state component, respectively) for allocating and freeing virtual addresses and physical pages.

```
allocMappableVirtualAddress :: (StateMonad [Addr Virtual] m)
  => m (Maybe (Addr Virtual))
freeMappableVirtualAddress  :: (StateMonad [Addr Virtual] m)
  => Addr Virtual -> m ()

allocPhysicalPage :: (StateMonad [PhysicalRegion] m, Debug m)
  => m (Maybe PhysicalRegion)
freePhysicalPage  :: (StateMonad [PhysicalRegion] m, Debug m)
  => PhysicalRegion -> m ()
```

The most common use for the virtual-address allocator is to find a location to map page-table and page-directory pages so that the pages may be read and written by

H. Similarly, the most common use for the physical page allocator is to allocate memory to back page-tables and page-directories. Thus, a typical pattern is to allocate a physical page and a virtual-address together. allocPageMapPage combines the two operations into a single function and produces a PageMapPage that is ready to supply as an argument to H (recall from Section 4.4.1 that a PageMapPage combines a free physical page with a kernel-mappable virtual address).

```
allocPageMapPage ::
  (StateMonad [PhysicalRegion] m, StateMonad [Addr Virtual] m, Paging m)
   => m (Maybe PageMapPage)
 allocPageMapPage
   = do mpage  <- allocPhysicalPage
        mvaddr <- allocMappableVirtualAddress
        case (mpage, mvaddr) of
          (Just page, Just vaddr) -> createPageMapPage page vaddr
          (Nothing, Nothing)      -> return Nothing
          (Nothing, Just vaddr)   -> do freeMappableVirtualAddress vaddr
                                        return Nothing
          (Just page, Nothing)    -> do freePhysicalPage page
                                        return Nothing
```

Conversely, freePageMapPage frees a PageMapPage.

```
freePageMapPage ::
  (StateMonad [PhysicalRegion] m, StateMonad [Addr Virtual] m)
   => PageMapPage -> m ()
 freePageMapPage pmp = do freePhysicalPage (physicalPage pmp)
                          freeMappableVirtualAddress (mappedAddr pmp)
```

The result of allocPageMapPage is a Maybe value because of the possibility that either the physical memory pool or the virtual-address pool is empty.

7.4.2 Kernel State

We build on the allocator monad to manage the rest of the L4 state. We divide the state into two components: the mapping database and a `System` data structure that stores all of the other kernel data.

```
type Kernel = ST System (ST MappingDB Allocator)
```

The `System` data structure tracks various information about the kernel, such as the queue of runnable threads, the identifier of the thread that is currently executing, and the system clock.

```
data System = System {
                runnable  :: [ThreadId],
                current   :: ThreadId,
                clock     :: Clock,
                threadMap :: Map ThreadId Thread,
                domainMap :: Map HWord Domain
              }
```

The `threadMap` and `domainMap` fields are essential: they track the state of every thread and address-space in the system. Recall from Section 7.1 that a `Thread` describes a thread's state and a `Domain` describes an address-space's state. The `threadMap` maps global thread identifiers to thread structures using Haskell's built-in dictionary type, `Map`. The `domainMap` is the analog for address-spaces and maps domain identifiers to domain structures. We access the mapping database and system state components using the standard monad operations `get` and `set`.

7.5 ERROR HANDLING

During the process of performing an IPC, various error conditions might occur that should be reported to the calling thread. For example, if a thread requests to send a message to a thread that does not exist, then we report that no partner can be found. We introduce a datatype that categorizes the possible error conditions, including a value for the absence of any error.

```
data IPCError = NoError
              | Timeout
              | NoPartner
              | Canceled
              | Protocol
```

`NoError` corresponds to a successful IPC operation. A `Timeout` error occurs when the intended partner is not waiting for a message but the sender or receiver specified they did not wish to wait. A `NoPartner` error occurs when the specified partner does not exist. A pending IPC may be aborted through an L4 system call named exchange registers, resulting in the `Canceled` error being sent to the thread that was blocked waiting. A `Protocol` violation happens when the kernel is expecting one of the special messages described by `SendType` and `RecvType` but the format of the message being processed does not match.

When the sending thread encounters an error, we communicate information about the problem to the user program by writing an error descriptor value in the error code field of the thread's UTCB. We signal that the thread should check this error code by setting a bit in message register zero (stored in the machine register ESI on IA32). If the thread was running at the time of the IPC, then we restart the thread. Most of the functionality for signaling the error is implemented in the function `setError`; this allows us to share common code between the functions that signal sender errors and receiver errors. The `HWord` argument named `recv` distinguishes a call to `setError` from a sending thread (`recv` value of `0`) from a call by a receiving thread (`recv` value of `1`).

```
setError :: (StateMonad System m, Execution m, KernelMemory m) =>
  Thread -> Bool -> HWord -> IPCError -> m ()
setError t running recv err
  = let ecode = ((fromIntegral (fromEnum err)) <<< 4) .|. recv
    in  do writeUTCBField (utcb t) ErrorCode ecode
           mr0 <- readRegister (context t) ESI
           writeRegister (context t) ESI (setBit mr0 15)
           restartThread t running
```

```
sendError :: (StateMonad System m, Execution m, KernelMemory m) =>
  Thread -> Bool -> SendType -> IPCError -> m ()
sendError st running FromMRs err = setError st running 0 err
sendError t running _ _ = stopThread t running
```

Receiver errors are handled in much the same way.

```
recvError :: (StateMonad System m, Execution m, KernelMemory m) =>
  Thread -> Bool -> ReceiveType -> IPCError -> m ()
recvError rt running ToMRs err = setError rt running 1 err
recvError t running _ _ = stopThread t running
```

The `Enum` instance of the `ErrorCode` type maps error code constructors to the appropriate numeric value to be stored in the UTCB. By using this technique, we will only have to update one place in the code if the mapping ever changes and we reduce the potential for mistakes that could be caused by using the wrong hard-coded error code value. However, we do introduce the construction and destruction of unnecessary values (and in turn garbage) that are not essential for the computation. Perhaps defining error code constants at the top-level would be a better trade off between functional style and performance in future design iterations.

7.6 THREAD RENDEZVOUS

The rendezvous component of IPC handles all aspects of the operation except for the actual transfer of data. The primary task is to locate a partner for the thread making the request. When a partner cannot be found, the rendezvous algorithm either saves the state of the thread so that it can complete the operation later or else sets an error code and resumes the thread. We organize the code so that the rendezvous portion of the code always stores any error that occurs, even if it occurs during message transfer.

Recall from Section 7.2 that each IPC message potentially includes a send phase, a receive phase, or both. In our implementation, we define a top-level function that corresponds to each phase.

```
send :: Thread -> IPCType -> Kernel ()
recv :: Thread -> IPCType -> Kernel ()
```

The `Thread` argument describes the state of the calling thread. The `IPCType` contains the parameters. Here there is an invariant that is not captured in the type: `send` should only be invoked with a `IPCType` value that matches the `Sending` constructor and `recv` should only be invoked with a `Receiving` value.

Sending threads must specify the global thread identifier of the intended receiver as a parameter to IPC[3]. Receivers may specify a global identifier, any-thread, or any-local-thread. Because of these differences, the partner location protocol is different for `send` and `recv`. In the send operation, we simply read the `Thread` structure of the intended receiver from the thread-map and check if that thread is waiting to receive a message from the sender. If the intended receiver is waiting to receive a message from the sender, then we try to transfer the message, possibly getting an error. If the receiver is not waiting, then the operation cannot proceed. If the sender specified an infinite timeout, then we block the thread and return to the scheduler; otherwise we signal an error and resume the thread. Figure 7.1 shows the code for `send`.

[3] We do not implement communication via local thread identifiers, although communication between local threads is still supported via global identifiers.

```
send :: Thread -> IPCType -> Kernel ()
send st send@(Sending to tout stype) =
  do mrt <- readThread to
     case mrt of
       Nothing -> sendError st True stype NoPartner
       Just rt -> case (status rt) of
                    Blocked (Receiving rid _ rtype) | idmatch rt rid ->
                      do ipcerr <- transferMessage st stype rt rtype
                         if ipcerr == NoError
                            then do insertThread rt{status=Runnable}
                                    insertRunnable rt{status=Runnable}
                                    nextPhase st{status=Blocked send} True
                            else do recvError rt False rtype ipcerr
                                    sendError st True  stype ipcerr
                    _  -> {- partner not blocked receiving -}
                      if tout == Zero
                         then sendError st True stype NoPartner
                         else do insertThread st{status=Blocked send}
                                 let ws' = waiting rt ++ [sid]
                                 insertThread rt{waiting = ws'}
                                 done
  where
    sid = threadId st
    idmatch rt rid =  (rid == anythread)
                   || (rid == sid)
                   || (rid == anylocal && parent st == parent rt)
```

Figure 7.1: The send phase of an IPC operation. This function locates the specified receiver and saves any errors that occur to the sender's message registers. Message transfer, which we will cover in Section 7.7, is handled by `transferMessage`. The `nextPhase` utility restarts the sending thread or initiates a receive phase, as appropriate.

We introduce a utility called `nextPhase` to handle restarting the sending thread once the send phase completes. The code for this function is shown in Figure 7.2. For send-only IPCs, we resume executing the sending thread immediately. Otherwise, we initiate the receive phase of the IPC request as appropriate for the message type being processed. If the send was a normal transfer from message registers, then the receive parameters come directly from the message registers of the caller as well. In all other cases, the kernel forges the parameters for the receive operation because it is acting on behalf of a faulting or otherwise incapacitated thread. As we saw in the discussion of `SendType` and `RecvType` in Section 7.2, many of the message types come in pairs: `SInterrupt`/`RInterrupt`, `SPageFault`/`RPageFault`, and `SException`/`RException`. In these cases, `nextPhase` generates a receive request of the appropriate type based on information from the current message. An `SPreempt` message does not have a receive phase, but causes the kernel to stop the running thread. Senders without a receive phase are restarted immediately or added to the kernel's runnable queue.

The algorithmic outline for `recv` is similar to `send`: locate a sending thread, transfer the message to the receiver, signal any error that occurs, and restart the sender and receiver. The key difference between the two phases is that partner location is more complicated in the receive phase because threads may request to receive a message from a specific thread (described by its global identifier), any local thread, or any thread in the system.

The first step in the receive algorithm is to locate a partner that is waiting to send a message to this thread. The receiver may specify a specific thread as the sender using its global ID or the receiver may request to receive from a broader group of threads (either any thread in the system or any thread in the thread's local address space). The `locatePartner` function looks up the `Thread` structure for a thread that matches the specification of the receiving thread. If the receiver specified a specific sender, then we read the structure that corresponds to that

```
nextPhase :: Thread -> Bool -> Kernel ()
nextPhase t running
  = case (status t) of
      Blocked (Sending pid _ SInterrupt) ->
        insertThread t{status = Blocked (Receiving pid Infinite RInterrupt)}
      Blocked (Sending pid _ (SPageFault _ _ _)) ->
        insertThread t{status = Blocked (Receiving pid Infinite RPageFault)}
      Blocked (Sending pid _ (SException _ _ _)) ->
        insertThread t{status = Blocked (Receiving pid Infinite RException)}
      Blocked (Sending _   _ FromMRs) | not (halted t) ->
        do from <- readRegister (context t) EDX
           if from == nilthread
              then restartThread t running
              else do tout <- readRegister (context t) ECX
                      let rcv = Receiving from (recvTimeout tout) ToMRs
                          t’  = t{ status = Blocked rcv }
                      insertThread t’
                      if running then recv t’ rcv else insertThread t’
      Blocked (Sending _   _ (SPreempt _)) ->
        stopThread t running
      _ -> restartThread t running
```

Figure 7.2: Restart a sending thread by initiating a receive phase or resuming the thread’s execution. The steps to take are determined by the type of the message that was just sent and whether or not the sending thread was running at the time the message was processed.

```
recv :: Thread -> IPCType -> Kernel ()
recv rt recv@(Receiving from tout rtype)
  = do mst <- locatePartner
       case mst of
         Nothing -> noPartner
         Just st -> checkPartner st
  where
    noPartner :: (StateMonad System m, Execution m, KernelMemory m) => m ()
    noPartner | from == anythread = timeout rt
              | from == anylocal = timeout rt
              | otherwise = recvError rt True rtype NoPartner

    timeout :: (StateMonad System m, Execution m, KernelMemory m) => Thread ->  m ()
    timeout rt | tout == Zero = recvError rt True rtype NoPartner
               | otherwise = do insertThread rt{status=Blocked recv}
                                done

    checkPartner :: Thread -> Kernel ()
    checkPartner st
      = do mrt' <- readThread (threadId rt)
           let rt' = fromJust mrt'
           let rid = threadId rt'
           case (status st) of
             Blocked (Sending sid _ stype) | sid == rid ->
               do ipcerr <- transferMessage st stype rt' rtype
                  if ipcerr == NoError
                     then do restartThread rt' True
                             nextPhase st False
                     else do recvError rt' True rtype ipcerr
                             sendError st False stype ipcerr
             _ -> timeout rt
```

Figure 7.3: The receive phase of an IPC operation. This function locates a sender by looking up a specified thread or searching the receiver's waiting queue. When a sender can be found, `checkPartner` transfers a message from the sender and stores any errors that occur in both threads' message registers.

thread identifier. If the receiver specified a broader group of senders, then we search the receiver's waiting queue to find an appropriate thread.

```
locatePartner :: (StateMonad System m) => m (Maybe ThreadId)
locatePartner
  | isGlobalId from   = readThread from
  | from == anythread = findAny (waiting rt)
  | from == anylocal  = findLocal (waiting rt) []
  | otherwise         = return Nothing
```

We use the functions `findAny` and `findLocal` to search the waiting queue. `findAny` returns the first thread identifier in the waiting queue of the receiver. `findLocal` returns the first identifier where the thread belongs to the same domain as the receiving thread.

```
findAny   :: (StateMonad System m) => [ThreadId] -> m (Maybe ThreadId)
findLocal :: (StateMonad System m) => [ThreadId] -> m (Maybe ThreadId)
```

The result of `locatePartner` is a `Maybe Thread` to reflect the fact that the receiver might have specified the identifier of a thread that does not exist or that the waiting queue might be empty (or not contain any local threads).

The remainder of the code for `recv` is shown in Figure 7.3.

7.7 MESSAGE TRANSFER

Once the kernel identifies a sender and receiver that are ready to communicate via IPC, we are ready to transfer the message from the sender to the receiver. In some cases a message transfer simply involves moving data from one location to another, but in other situations IPC requires more involved processing by the kernel. For example, a thread start-up message signals the kernel to set a thread's EIP and ESP registers to particular values. Even during normal message transfers, L4 allows users to send typed items to map or grant memory; these items cause the kernel to modify the address-space of the receiving thread (and sometimes the sender as well in the case of a grant operation).

To handle the complexity of the message transfer operation, we divide the implementation into three functions that handle different aspects of the process.

- **transferUntyped** copies data from the message registers of the sender to the message registers of the receiver. This data is not processed by the kernel in any way.
- **transferTyped** copies typed items from the sender's message registers to the receiver's message registers. Each typed item corresponds to a memory sharing request that potentially affects the address-spaces of the threads and the mapping database. We introduce a utility called `transferTypedItem` to handle the processing. `transferTyped` invokes this utility for each typed item being sent and copies the ones that are successfully processed to the receiver.
- **transferMessage** is the top-level message transfer function. This function checks that the type of message being sent and the type of message being received are compatible, and if they are, `transferMessage` performs any type-specific kernel processing (such as setting EIP). `transferMessage` invokes functions for copying untyped and typed items from the message registers as appropriate for the type of message being processed.

The message transfer function succeeds even when the kernel cannot copy the entire message because some of the memory sharing requests were not valid. The only failures we encounter in our implementation at this stage are protocol errors; these occur when a special message is being sent but the format does not match the format specified in the L4 API.

The `transferMessage` function takes the sending thread, the type of message being sent, the receiving thread, and the type of message being received as parameters.

```
transferMessage :: Thread -> SendType -> Thread -> ReceiveType
                     -> Kernel IPCError
```

We introduce one equation for each valid combination of send-type and receive-type. Any invalid combinations are caught by a catch-all equation that returns a protocol error.

The first equation implements normal message transfer. A description of the message is available through the ESI register of the sender, which semantically corresponds to MR0. This descriptor contains the number of untyped words and the number of typed items to be processed. `transferMessage` uses the `transferUntyped` and `transferTyped` functions to perform the transfer.

```
transferMessage st FromMRs rt ToMRs
  = do -- read message information --
       mr0 <- readRegister (context st) ESI
       -- transfer message --
       mrt <- transferUntyped (utcb st) (utcb rt) MR1 (mr0 .&. 0x3f)
       t' <- transferTyped st rt mrt ((mr0 >>> 6) .&. 0x3f)
       -- store updated message tag --
       let mr0' = mr0 .&. 0xffff003f .|. (t' <<< 6)
       -- store sender information --
       writeRegister (context rt) ECX (threadId st)
       writeRegister (context rt) ESI mr0'
       return NoError
```

The kernel writes the message descriptor into the receiver's register, updating the number of typed items to reflect the number of items that were successfully shared. The global identifier of the sender is also passed on to the receiver.

To send a page fault message, the kernel writes information describing the fault into the message registers of the receiver. This information includes the address of the fault, the instruction pointer of the faulting thread, and the permissions with which the location was accessed. In addition to transferring the page fault data, the kernel constructs a message descriptor for the message and saves it in the receiver's MR0.

```
transferMessage st (SPageFault addr eip rwx) rt ToMRs
  = let mr0 = (0xffe <<< 20) .|. (rwx <<< 16) .|. 2
    in  do writeRegister (context rt) ESI mr0
           writeRegister (context rt) ECX (threadId st)
           writeUTCBField (utcb rt) MR1 addr
           writeUTCBField (utcb rt) MR2 eip
           writeUTCBField (utcb rt) MR3 (threadId st)
           writeUTCBField (utcb st) BR0 (1<<<4)
           return NoError
```

The mapping from page fault data to registers is part of an L4 protocol that the page fault handler uses to parse the message.

After processing a page fault, the handler responds by sending a reply to the faulting thread. Again, the format is specified by an L4 protocol. The kernel expects a message with a single typed item (which corresponds to two typed-item words). This typed item specifies a map or grant item to add to the receiver's address-space that will service the page fault. The kernel processes the typed item as with any other using the `transferTypedItem` utility.

```
transferMessage st FromMRs rt RPageFault
  = do mr0 <- readRegister (context st) ESI
       if ((mr0 >>> 6) .&. 0x3f) /= 2
          then return Protocol
          else do mti <- transferTypedItem st MR1 rt False
                  case mti of
                    Nothing -> return Protocol
                    Just _  -> return NoError
```

A protocol error occurs if the message being sent does not have the right format or if the typed item cannot be processed successfully.

The remaining equations for `transferMessage` rely on similar implementation techniques to those we have examined so far. For messages being sent in response to a fault or exception to a handler, the kernel writes information describing the fault into the message registers of the receiver. For messages being sent from message registers, the kernel reads the parameters from the registers, checks that they obey the appropriate protocol, and updates the receiving thread as appropriate for the message type. We omit the complete code.

The kernel transfers untyped data between threads using `transferUntyped`. This function transfers one untyped item at a time from a specified source UTCB to a specified destination UTCB. The field being copied and the number of remaining untyped items in the message are also parameters.

```
transferUntyped :: (KernelMemory m) => UTCB -> UTCB -> UTCBField
  -> HWord -> m UTCBField
transferUntyped sutcb rutcb f num
  | num <= 0  = return f
  | otherwise = do val <- readUTCBField sutcb f
                   writeUTCBField rutcb f val
                   transferUntyped sutcb rutcb (succ f) (num - 1)
```

When the number of remaining items reaches zero, `transferUntyped` returns the next field where data should be copied from. This will be the starting point for transferring typed items. To transfer an untyped item, the function reads a value from the UTCB of the source and writes it to the UTCB of the destination.

The algorithm for transferring typed items is conceptually similar, but is made more complicated by the possibility that transferring a typed item can fail. Because of this potential failure, we must track the number of typed items that are successfully processed so that we can supply this number to the receiver. We must also track the source message register UTCB field and the destination message register UTCB field separately, because they will not necessarily be the same. We implement `transferTyped` using a recursive helper function that tracks all of the relevant information.

```
transferTyped :: (StateMonad System m, StateMonad [PhysicalRegion] m,
                  StateMonad [Addr Virtual] m, StateMonad MappingDB m,
                  KernelMemory m, Paging m, UserMemory m, Debug m)
                   => Thread -> Thread -> UTCBField -> HWord -> m HWord
transferTyped st rt f num = transferTyped' f f num 0
  where
    transferTyped' sf rf n acc
      | n <= 0    = return acc
      | otherwise
          = do mti <- transferTypedItem st sf rt (n /= 0)
               case mti of
                 Nothing -> transferTyped' (next sf) rf (n-1) acc
                 Just (w1,w2) ->
                   do writeUTCBField (utcb rt) rf w1
                      writeUTCBField (utcb rt) (succ rf) w2
                      transferTyped' (next sf) (next rf) (n-1) (acc+1)
    next = succ . succ
```

Typed items are processed using `transferTypedItem`, which is shown in Figure 7.4. When a transfer succeeds, `transferTyped` copies the map or grant item into the UTCB fields of the receiver. In any case, we continue processing the message until we reach the end of the typed item list.

7.8 SUMMARY

In this chapter we provided a basic overview of our L4 implementation and an in-depth look at our approach to IPC. We will cover the performance of IPC in Chapter 8. Table 7.1 describes the size of our L4 kernel, in source lines of code, overall and for our IPC implementation alone. The overall line count includes all of the code (outside of H) in the implementation. This includes our implementation

```
transferTypedItem ::
  (StateMonad System m, StateMonad [PhysicalRegion] m, StateMonad [Addr Virtual] m,
   StateMonad MappingDB m, KernelMemory m, Paging m, UserMemory m, Debug m)
    => Thread -> UTCBField -> Thread -> Bool -> m (Maybe (HWord,HWord))
transferTypedItem st si rt more
   = do base     <- readUTCBField (utcb st) si
        fpage    <- readUTCBField (utcb st) (succ si)
        acceptor <- readUTCBField (utcb rt) BR0
        let (snd, rcv) = reconcileFpages fpage acceptor (base .&. 0xfffffc00)
        msd <- readDomain (parent st)
        mrd <- readDomain (parent rt)
        case (msd, mrd) of
          (Just sd, Just rd) ->
            do mpmp <- allocPageMapPage
               case mpmp of
                 Nothing -> return Nothing
                 Just pmp ->
                   do b <- mapFunction (testBit base 1) [pmp] (space sd) snd (space rd) rcv
                      case b of
                        Nothing -> return Nothing
                        Just False -> do freePageMapPage pmp
                                         return (Just (setBitTo base 0 more, l4FpageToWord rcv))
                        Just True -> return (Just (setBitTo base 0 more, l4FpageToWord rcv))
          _ -> return Nothing
  where
    mapFunction b = if b then replaceMapping else shareMapping
```

Figure 7.4: Process a single map or grant item. This function uses the operations of the mapping database (and in turn H) to modify the address space of the receiving thread (in the case of map and grant) and the address space of the sender (in the case of grant). The result is an updated typed item that reflects any changes made to the mapping area due to a mismatch between the sent flexpage and the receiver's acceptor.

of thread scheduling, thread creation and deletion, and address-space management.

Description	**Source Lines of Code**
L4 Kernel	2119
IPC	320
Mapping Database	242

Table 7.1: The source lines of code for our L4 implementation. Source lines of code do not include blank lines or comments.

All of the code was written in Haskell using the H interface; no extra foreign calls or potentially unsafe primitives were needed. The total lines of code is just over 2,100. Even when combined with the size of the primitives for H (presented in Section 6.9), the size of the kernel is around 5,300 lines of code. Typical C/C++ implementations of L4 are around 10,000 lines of code, though this is not truly a fair comparison because these other kernels may be more portable or support more of the API. Using Haskell does not seem to be a tremendous advantage or disadvantage when it comes to code size in this application.

Chapter 8

PERFORMANCE RESULTS

In designing and implementing the H interface, our focus is on the safety properties of the interface rather than the performance. However, performance is an important characteristic of systems software and we must address the performance costs that come with our approach to safety. In this chapter, we will evaluate the performance of H in the context of the L4 inter-process communication implementation. We choose to analyze the performance of L4, rather than directly examining H, for the following reasons:

- **Comparability:** L4 has numerous implementations which allows us to evaluate our performance results against an equivalent system that does not contain the safety overheads introduced by an H and that is written in a low-level language. There is no directly comparable system to H, so an evaluation of the H primitives in isolation would be artificial.

- **Relevance:** The H interface allows programmers to construct operating systems in Haskell. Even if the H primitives are fast, we must consider the overheads that stem from writing our operating system on top of the interface using Haskell. By using a kernel-level benchmark, we can observe the cost of the architecture as a whole.

By comparing to an optimized C system that does not contain the same kinds of safety checks or the additional indirection of a low-level abstraction layer, we can provide an initial estimate of the performance cost of an H-based system compared to a traditional approach to systems software development.

Our goal is to analyze the performance of an H-based system, rather than to demonstrate that our L4 kernel achieves performance on par with a production C kernel. Neither H nor our implementation of L4 were implemented with performance or optimization as a priority, so we expect poor performance to start. The initial results demonstrate that it is feasible to implement operating systems in a safe language without introducing memory-safety violations and to quantify the costs of a naive approach to implementing such systems. Much as the L4 architects demonstrated that microkernels could perform well through targeted optimization [70, 71], it might be possible that turning a similar eye to kernels built using the H architecture (as well as further research into compiling functional languages) will allow us to increase performance and ultimately demonstrate the viability of the safe language approach.

To demonstrate that our initial results do not reflect the best possible performance reachable with our approach, we devote a portion of this chapter to optimization techniques for an H-based system. The optimizations improve the performance of our IPC implementation significantly. We focus our optimization energy on the IPC implementation discussed in the previous chapter as a proof of concept, but we use general Haskell optimization techniques that can be applied to the entire kernel.

The remainder of this chapter presents our performance analysis. Section 8.1 describes the mechanisms we use to evaluate IPC, the environment in which we run our tests, and the profiling techniques we use to measure the Haskell run-time system behavior. Section 8.2 presents our initial results. Section 8.3 demonstrates techniques for optimizing the key components of an H-based system: the algorithms that run on top of the interface, the interface itself, and the code generated by the Haskell compiler. Section 8.4 summarizes the effects of our optimizations and our overall performance results.

8.1 TEST CONSTRUCTION AND MEASUREMENT ENVIRONMENT

The starting point for our performance evaluation is a standard L4 benchmark called ping-pong. This benchmark measures IPC performance by creating two threads—"ping" and "pong"—and calculating the average time it takes for these threads to send and receive a series of IPC messages of different sizes. Ping-pong can be configured such that either the threads run in different address spaces (an inter-address-space measurement); or the threads run in the same address space but communicate using the normal IPC system call; or the threads run in the same address space and communicate using a special IPC call that is optimized for local communication. All of our measurements use the inter-address-space variant.

Ping-pong reports two statistics about IPC: the average number of CPU cycles taken and the average time per message. The cycle count is measured using the IA32 instruction `rdtsc`, which reads the processor's time-stamp counter register. The initial value of the time-stamp counter after a processor reset is zero. Subsequently, the processor increments the time-stamp counter on every clock cycle [53]. The elapsed time is measured using the L4 system call `SystemClock`, which reads L4's internal clock. We implement the L4 clock in our kernel by configuring a timer at a particular frequency and incrementing the clock appropriately on each timer interrupt.

The source code for our experiments comes from the University of Karlsruhe Pistachio distribution of L4 [63]. The following pseudo-code describes the basic algorithm of the test:

```
create two threads "ping" and "pong"
for i = 0, 4, 8, ..., 60:
    read clock/cycle count
    for ROUNDS times:
      send i words of data from ping to pong
```

```
  send i words of data from pong to ping
read clock/cycle count
compute average cycles and time taken per message
```

The loop-variable `i` controls the amount of data that is transferred by each message (in words). We measure the average IPC time for payloads in the range 0 to 60 words, incrementing the size by 4 words with each loop. `ROUNDS` controls the number of IPC messages that are sent with each payload size. The original source code for ping-pong uses a ROUNDS value of 128K. The original source also uses a 32-bit cycle counter, so we lower the number of `ROUNDS` in our Haskell benchmarks to avoid problems with overflow[1].

We run our experiments on an HP Mini 110 net-book with a 1.66 GHz Atom N455 processor with a 512 KB L2 cache and 1 GB of RAM. Our platform during development and optimization was a VirtualBox [92] machine with 256 MB of RAM running on a 2.33 GHz MacBook Pro. The VirtualBox environment was beneficial because it provided fast feedback about changes to the code and made it easy to set up new tests. In particular, VirtualBox was very useful for identifying the optimal garbage collector settings for the L4 kernel because of the sheer number of tests that needed to be evaluated. We discuss the details of the garbage collector experiments in Section 8.2.3—these are the only experiments where we will report results collected on a virtual platform rather than hardware.

For all of the experiments, we set the Haskell heap size to 16 MB. Other parameters of the garbage collector vary across the test suite. For the initial measurements, the garbage collector is configured with the default settings of GHC

[1] On a 1.66 GHz processor, the 32-bit cycle counter will wrap every $2^{32} * (1/1,660,000,000)$, or 2.59, seconds. We must choose a value for ROUNDS such that all of the IPC calls will complete in less than this amount of time. In our experiments we used a value of 7,128. We decided against changing the source to use a 64-bit cycle counter to minimize the changes to the test harness; the benchmark is already parameterized by `ROUNDS` so that was a minor change.

(version 6.8.2) [31].

In the rest of this section, we will provide a closer look at the code being evaluated and the Haskell-specific mechanisms we put in place for measuring our kernel. Section 8.1.1 explores the steps taken through the kernel during each IPC request; these are the aspects of our implementation that will be measured by our experiments. Section 8.1.2 discusses the extensions that we make to H, L4, and ping-pong to obtain more detailed information about the performance of our Haskell kernel, including the percentage of time spent in garbage collection.

8.1.1 Anatomy of an IPC Request

In this section, we review the steps that an IPC request takes on its way into the Haskell kernel as well as the basic components of our IPC implementation. Section 6.8 and Chapter 7 cover these topics, respectively. We review the steps here in preparation to evaluate the performance of the IPC implementation: we will measure the number of cycles that our kernel spends in each phase of the implementation in Section 8.2.1. These fine-grained measurements are the starting point of our performance analysis. Each phase of the algorithm is a potential source of inefficiency—and an opportunity for optimization. Figure 8.1 illustrates the path of an IPC and the connections between each of the phases.

Context-Switching A user program sends an IPC message by invoking the appropriate system call via the IA32 instruction `INT n`. Control transfers back to H via a kernel-mode interrupt handler that gets invoked by the hardware. We resume execution in H midway through the `execute` function—recall from Section 6.8 that `execute` runs a user program by calling out to C—and then we return a description of the interrupt to the client kernel. Between the moment when the user program issued the system call interrupt and the moment when the client kernel receives the

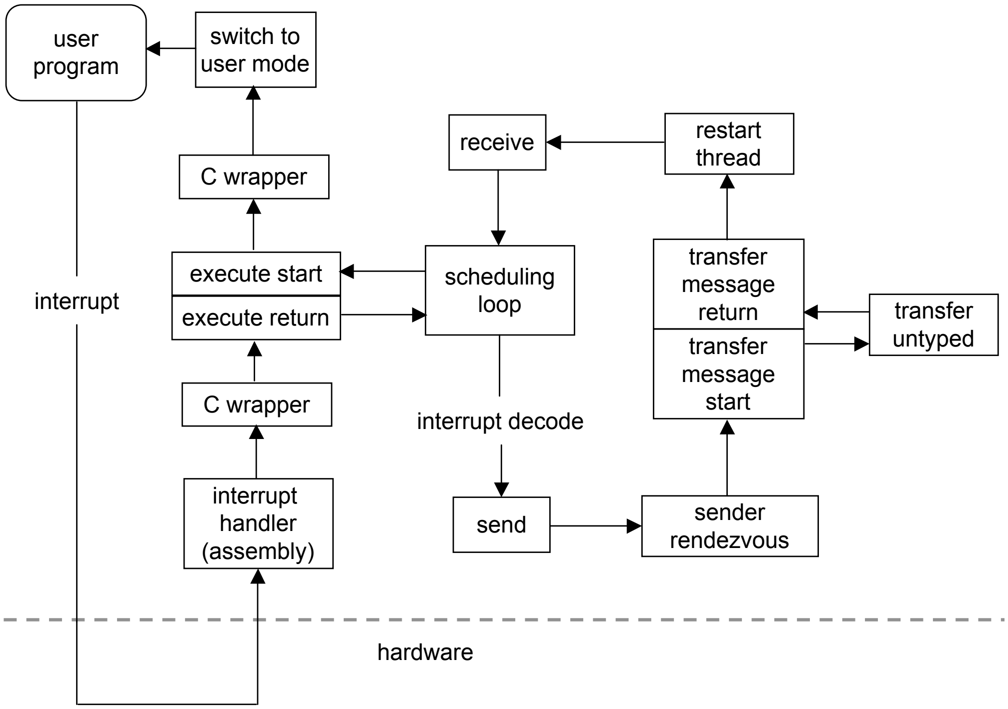


Figure 8.1: The flow of execution through the kernel when a user program issues an IPC request. The interrupt handler passes control back to H which then returns to the client kernel. The client kernel handles the IPC request, invokes the scheduler to restart the user program, and calls H to switch the processor back to user-mode.

interrupt information the program passes through an assembly language interrupt-handler, a C wrapper for that assembly function, and then finally enters Haskell. The program travels through a similar path in reverse when we return to user-mode. We refer to each transition from a user-mode C program to a kernel-mode Haskell kernel (or vice versa) as a context switch.

In Section 8.2.1 we will measure the cost of context switches indirectly by computing the difference between the total time for each IPC and the time that is spent in the Haskell kernel. This technique provides an approximation of the context-switching cost, but also includes other overheads that have nothing to do with context switching. Laziness is one culprit: any computation that has not been forced by the time we return to C will be executed before we return to user-mode. Thus, it can appear that certain costs are part of context-switching even though they originated in a different area of the program. In Section 8.2.2 we examine a test that specifically targets context-switching time so that we can obtain a more accurate measurement.

Scheduling The scheduling loop manages the execution of user programs. Each time we enter the loop, we look for a runnable user thread, load its saved state, and run the thread using the `execute` function from H. When `execute` returns, we decode the interrupt that we received from the user thread and invoke the appropriate handler in the kernel.

The interrupt decoding portion of the scheduling loop is inefficient: at the assembly language level we knew precisely which interrupt had occurred but the process of returning via `execute` requires us to decode the interrupt again. We could reduce the time spent in scheduling by invoking the appropriate Haskell handlers directly without going via `execute`. Such an approach would require some re-architecting of H, but would be feasible if the extra scheduling time turns out to be a bottleneck. We measure the scheduling time by subtracting the total time spent in the IPC system call handler from the time spent in the Haskell kernel.

Sender Rendezvous The rest of the phases that an IPC request passes through are steps in the IPC implementation itself, rather than kernel overheads. In general, the path of a particular IPC might vary depending on the type of the IPC request (send-only, receive-only, or send-and-receive) and on whether or not the desired partner is ready to send or receive. In the IPC messages produced by ping-pong, there is no such variety. Every message is a send-and-receive request and the tests are structured such that the sender always succeeds in transferring a message while the receiver always waits. We take advantage of these patterns when structuring our IPC profiling points and focus our measurements more heavily on the steps of the send phase of the implementation.

The sender rendezvous phase of IPC is where the sending thread attempts to locate the intended receiver of its IPC message. The kernel must look up the current status of the intended receiver to verify that that thread is waiting to receive a message from the sending thread. The algorithm is not inherently complex, but involves a number of state monad accesses and data structure manipulations.

Transfer Message The transfer message phase performs the real work of IPC. It determines the type of IPC message being sent, coordinates the transfer of the untyped words and typed data from the sender to the receiver (as appropriate for the message type), and updates the message registers of the receiver with a description of the message. For evaluation purposes, we measure the time spent in transfer message and the time spent in the transfer untyped phase (the phase that transfers untyped words) separately.

Transfer Untyped The transfer untyped phase copies the untyped words of a message from the sender to the receiver. Most of the cost of an IPC message comes from this phase, and we will take an in-depth look at the cost of our implementation in Section 8.3.

Thread Restart Once the kernel transfers a message from the sender to the receiver, both threads are ready to run again. The receiver automatically becomes runnable—there is no other state the thread can enter. The sender becomes runnable unless their IPC request also included a receive phase. In that case, the kernel constructs the appropriate receive request and invokes the receive phase of the implementation. The thread restart cost is the time between the completion of the transfer message phase and the start of the receive phase (all send requests in the ping-pong test also include a receive phase).

Receive Phase The receive phase encompasses all of the work done to receive a message. In the ping-pong tests, we know by construction that the messages are always transferred in the send phase and that the receiver always blocks. Thus, our measurements of the receive phase examine the cost of searching for an acceptable sender and blocking when such a sender is not found. There is an inherent inefficiency in our implementation with respect to this benchmark: we know that the sender cannot complete its receive phase because the intended partner of the receive phase just became runnable (the intended sender is the thread that was the receiver in the last message transfer). All of the computation spent looking up the sender is wasted. A future optimization would be to prioritize the thread that just unblocked (because it is runnable) or to check if the receive phase will be pointless in advance. There are papers that study these kinds of trade-offs for L4 implementations [80].

8.1.2 Haskell Profiling on Bare Metal

The ping-pong benchmark from Pistachio [63] reports two statistics about the program under test: the average cycles taken per message transfer and the average time taken per message transfer (in microseconds). The cycle count is collected via `rdtsc` and the microsecond count is determined by querying the L4 SystemClock system call. In a Haskell program, there are additional sources of potential inefficiencies that are not easily measured using the existing system calls or hardware functions, such as the amount of time spent in garbage collection. To help us understand our performance results more deeply, we extended our H and L4 implementations to support the collection of `rdtsc`-style cycle counts from within Haskell and to support some simple garbage collection profiling. This section will cover some details of these profiling extensions.

Note that the standard distribution of GHC includes sophisticated profiling tools for understanding the performance of Haskell programs [91]. Unfortunately, our bare metal port of the GHC run-time system cannot support the usual profiling mechanisms, because they rely heavily on OS-level timing facilities for collecting data and on an underlying file system for reporting the data. Ultimately, adapting the real profiling tools to work in a bare metal environment would be a valuable exercise that would greatly improve our ability to understand the performance of Haskell kernels. The profiling tools presented here are a stop-gap to enable us to collect data without spending too much extra engineering effort.

Garbage Collection Profiling Dynamic memory management with garbage collection is a key mechanism for enforcing memory-safety in Haskell, but it is important to understand the overheads that it introduces. We implement basic garbage collection profiling tools that allow us to determine how much of our execution time is being spent on garbage collection, the effect of various optimizations on the garbage collection behavior of the program, and the impact of different run-time system parameter settings. To implement these tools, we must extend the run-time system of GHC to track the information of interest. This information is in turn exported via H and L4 so that we can query the garbage collection statistics from user programs.

In GHC, garbage collection occurs in a run-time system function that is called `GarbageCollect`. Because of the profiling tools that are normally part of GHC, the `GarbageCollect` function already contains hooks to support statistics gathering and program profiling. At the start of every garbage collection, the function `stat_startGC` is called. Normally, this function starts any garbage collection related statistics gathering. At the end of every garbage collection, an analogous function, `stat_endGC`, is called.

```
void GarbageCollect(...) {
  ... initialization ...
  stat_startGC();
  .. do garbage collection ...
  stat_endGC();
  ... clean up ...
}
```

We add our profiling instrumentation to stat_startGC and stat_endGC to guarantee that the code is called exactly once per garbage collection. We introduce a run-time system variable that counts the number of garbage collections (incremented in stat_startGC) and a variable that accumulates the number of cycles spent in garbage collection throughout the life of the system. We compute the cycle count by querying rdtsc once in stat_startGC and once in stat_endGC and adding the difference to our accumulator.

To collect the garbage collection statistics, we need to export the values of our counters through the H interface to the client kernel. We export the run-time system accessor functions to the client by adding straightforward FFI wrappers of the functions to the H interface. These H functions allow the client to query the garbage collection statistics, but do not introduce any safety issues.

```
foreign import ccall unsafe "getGarbageCollections"
  getGarbageCollections :: IO HWord64

getNumCollections :: H HWord64
getNumCollections = liftIO $ getGarbageCollections

foreign import ccall unsafe "stat_getElapsedGCTime"
  getElapsedGCTime :: IO HWord64

getCollectionTime :: H HWord64
getCollectionTime = liftIO $ getElapsedGCTime
```

We also add two new system calls to our L4 kernel so that user programs can query the current number of garbage collections and the time spent in garbage collection throughout the kernel's execution so far. Our version of ping-pong uses these system calls to integrate garbage collection statistics into the experimental results that are reported for each test.

In-Kernel Profiling Haskell programs do not have access to the normal hardware mechanisms for taking timing measurements without making a foreign function call. But client kernels are not allowed to invoke the foreign functions directly, so we extend H with an interface to the `rdtsc` counter to allow in-kernel profiling measurements. The in-kernel profiling tools allow us to start and stop cycle count measurements at any point in our Haskell code so that we can obtain fine-grained information about where time is being spent. To support in-kernel profiling, we add three new functions to the `Debug` class of the H interface:

```
startMeasurement :: HWord -> H ()
stopMeasurement  :: HWord -> H ()
readMeasurement  :: HWord -> H HWord32
```

Each function takes a word as a parameter: this word is an identifier for the measurement being taken. We allow a fixed number of simultaneous measurements; the maximum is arbitrarily set to eight. The semantics of each function is straightforward: `startMeasurement` starts the timer for a particular measurement using `rdtsc`; `stopMeasurement` stops a measurement; and `readMeasurement` returns the average number of cycles taken for a particular measurement. The counter is reset every time that the measurement is read. Although this is not a good design for a general profiling suite, it was well-suited to our particular use-case.

We implement the measurement functions in C so that we can access the `rdtsc` counter using inline assembly instructions. The H functions that are exposed to client kernels simply lift the C functions into Haskell. As with the garbage collection profiling functions, we add a system call that allows the user program to call the `readMeasurement` function so that we can report the results of in-kernel profiling from the user-level program. This approach minimizes the extent to which the profiling extensions impact the performance results. By designing `readMeasurement` to compute the average cycle count and clear the counter, we only needed to expose one primitive to user-level programs.

8.2 INITIAL MEASUREMENTS

The initial measurements of our L4 kernel using ping-pong reflect the performance of a naive H-based operating system without any optimization. We choose these measurements as our starting point to provide a baseline against which to compare our optimized implementations. Section 8.2.1 presents the initial results for the Haskell kernel including a breakdown of the time spent in each phase of IPC. Section 8.2.2 takes an in-depth look at the cost of context-switching between a kernel written in Haskell and a user program written C. Section 8.2.3 examines the role of garbage collection in the initial results and provides a comparison between different run-time system configurations.

8.2.1 Ping-Pong Performance

We expect there to be costs that accompany the use of an H-based architecture. The additional dynamic checks that we include in the H primitives to enforce memory-safety, the abstraction barrier that we place between the client kernel and the machine, and our choice of Haskell as an implementation language are all potential drains on performance. We compare the performance of our kernel to the Pistachio distribution [63] of L4 to obtain a real measure of the cost of safety in our approach. Pistachio is written in C++ and is highly optimized, so it serves as a reasonable representative of the “typical” approach to operating systems. Furthermore, Pistachio is based on the same version of the L4 API as our implementation (X2 [62]), so the two kernels are directly comparable in terms of functionality. We use the Pistachio implementation of ping-pong for all of our IPC measurements. Table 8.1 shows the results from running the inter-address-space variant of ping-pong on Pistachio for a one-way IPC on our test platform.

Words	**Cycles**	μ**s**
0	1,270	0.76
4	1,332	0.80
8	1,337	0.80
12	1,348	0.81
16	1,354	0.81
20	1,365	0.82
24	1,371	0.82
28	1,379	0.82
32	1,389	0.83
36	1,395	0.83
40	1,405	0.84
44	1,410	0.84
48	1,422	0.85
52	1,427	0.85
56	1,435	0.86
60	1,446	0.86

Table 8.1: Result of running ping-pong on Pistachio in our experimental environment. Ping-pong tests the average number of cycles and average time in microseconds required to send various numbers of words over IPC (from 0 to 60 in increments of 4).

To test the Haskell kernel, we run a slightly modified version of ping-pong that reports garbage collection statistics in addition to the standard results. We use the garbage collection profiling from Section 8.1.2 to calculate the total number of garbage collections that occur during all IPC messages of a certain size (for example, 321 garbage collections occur when testing IPC with a payload of zero words). We use a total count of garbage collections rather than an average per message because the number of collections per message is less than one. Table 8.2 presents our initial results.

We expected the performance of our initial implementation to be poor, but even so these results are a bit disappointing. Examining the data, we see that our kernel takes roughly 30,000 additional cycles for every additional 4 words of data that we transfer. When we do not transfer any words of data, the Haskell kernel is 67 times worse than Pistachio, but by the time we reach 60 words our kernel is 368 times worse. We might expect a high cost for using the H interface, but would not expect such a large number of cycles to be spent for each word transferred. This suggests that something costly is happening in the routine that transfers data between the sender and the receiver.

Our understanding of the algorithmic structure of the code suggests that we should examine the `transferUntyped` function that copies untyped words from the sender to the receiver. We use the in-kernel profiling tools introduced in Section 8.1.2 to calculate the number of cycles that are spent in that function (on average). We also measure the average cycles spent in the other phases of IPC discussed in Section 8.1.1: overhead/context-switching, scheduling, sender rendezvous, transfer message, thread restart, receive phase, and garbage collection. The measurements for these phases are shown in Table 8.3. By measuring each phase of the algorithm, we can determine the most profitable places to spend our optimization energy and learn more about the behavior of our program. Note that the total cycles per IPC is worse for the measurements that use in-kernel profiling; this slowdown is due to the cost of the profiling measurements themselves.

Words	**Cycles**	μs	**GCs**	× **P**
0	85,223	53	321	67
4	119,324	73	451	90
8	149,068	89	583	111
12	178,781	106	715	133
16	208,498	126	845	154
20	237,995	142	978	174
24	267,410	162	1,109	195
28	296,954	179	1,240	215
32	326,074	195	1,373	235
36	355,614	215	1,503	255
40	385,259	232	1,635	274
44	414,741	251	1,766	294
48	444,155	268	1,897	312
52	473,276	285	2,030	332
56	502,743	301	2,160	350
60	532,215	318	2,292	368

Table 8.2: Initial results for our Haskell implementation of L4 on H. The table shows the number of words transferred over IPC, the average cycles taken per message, the average time in microseconds per message, and the total number of garbage collections that occurred during the test (not per IPC). The final column is the ratio between the average cycles taken for a particular test and the equivalent result from Pistachio (see Table 8.1 for the full results). This number indicates how many times slower our implementation is than the Pistachio baseline.

The results in Table 8.3 confirm that our implementation of `transferUntyped` is the source of the high cost for transferring each word. The base cost of executing `transferUntyped` when no data is transferred is 3,530 cycles. This increases by roughly 30,000 cycles for every 4 words to a peak of 450,265 cycles when 60

Words	OH	Sch.	SR	TM	TU	Rst.	Rcv.	GC	Total
0	6,789	26,194	7,903	14,710	3,530	16,395	24,085	1,471	101,077
4	7,184	26,486	7,895	14,732	35,996	17,035	24,127	2,041	135,496
8	7,552	26,191	8,057	14,849	65,945	17,032	24,310	2,639	166,575
12	7,986	25,808	7,948	14,998	95,358	17,103	24,456	3,191	196,848
16	8,375	25,497	7,920	14,850	125,435	17,148	24,395	3,728	227,348
20	8,974	25,098	7,953	15,114	154,878	17,120	24,510	4,350	257,997
24	9,423	24,698	8,009	15,144	184,812	17,253	24,505	4,930	288,774
28	10,137	24,131	8,058	15,161	214,211	17,284	24,493	5,540	319,015
32	10,668	23,505	8,037	15,122	243,620	17,290	24,555	6,102	348,899
36	11,425	22,872	7,982	15,145	273,235	17,336	24,583	6,621	379,199
40	12,259	22,127	7,997	15,058	303,063	17,515	24,634	7,185	409,838
44	12,967	21,579	8,028	15,213	332,575	17,291	24,464	7,775	439,892
48	13,979	20,528	7,962	15,306	362,215	17,319	24,695	8,410	470,414
52	14,622	20,131	8,119	15,148	391,677	17,310	24,652	8,921	500,580
56	15,606	19,147	8,143	15,180	420,793	17,303	24,661	9,501	530,334
60	16,771	18,007	7,991	15,289	450,265	17,312	24,742	10,079	560,456

Table 8.3: A breakdown of the cycles spent in each phase of the IPC implementation. OH stands for overhead and includes the time spent to context-switch between the Haskell kernel and the C user program. Sch. is the time spent in the scheduling loop. SR stands for send rendezvous, TM stands for transfer message, and TU stands for transfer untyped. Rst. describes the phase of IPC spent restarting the thread. Rcv. is the time spent in the receive phase of the IPC operation. The phases are explained in detail in Section 8.1.1.

words are transferred. These results indicate that `transferUntyped` is an appropriate place to start our performance analysis. Copying data should have a relatively low, constant cost, so there should be hope to improve the performance of `transferUntyped` through optimization. Section 8.3 will provide a detailed walk-through of the `transferUntyped` implementation and show that our original implementation has (and realizable) opportunities for optimization of its performance.

8.2.2 Context-Switching Time

Before moving on to optimization, we will take a closer look at some of the base costs involved in our Haskell kernel. This section examines the cost of context-switching between the Haskell kernel and a user program written in C. In the previous section, we measured the cycles spent for “context-switching”. This measurement provides a rough estimate of how many cycles our L4 implementation requires to enter and leave the kernel, but it also includes other overheads such as the cycles required for the profiling functions and misattributed costs due to laziness. To isolate the actual time taken to switch between the user and the kernel, we created a new test specifically to measure the cost of context-switches.

The context-switch test works much like ping-pong except that the user program invokes a “no-op” system call instead of IPC. The interrupt passes through assembly to H to the Haskell kernel as explained in Section 4.5. In the kernel, the interrupt handler returns immediately. As in ping-pong, we execute the no-op system call several thousand times and compute the average cycles per context-switch using `rdtsc`. Over 115,000 tests, the average number of cycles taken for a context switch is 3,289 cycles. This reflects the path of a single crossing from user-to-kernel mode or kernel-to-user mode—the round-trip of a system call requires two such switches. The context-switch time is consistent with the overhead of transferring zero words of data that we measured in Section 8.2.1.

We perform the context switch from Haskell to C using an optimized version of the `execute` primitive from Section 4.5. Recall that the original definition of the `execute` primitive returns an `Interrupt` data structure to describe the user event that brought control back to the kernel:

```
  execute :: PageMap -> FaultContext -> H Interrupt
```

Creating the `Interrupt` data structure is wasteful because it introduces an additional case-split on the interrupt vector and allocates extra space. Context-switches via this primitive take an average of 3,517 cycles. Our alternative primitive, called `executeWithoutAllocation`, returns the interrupt vector that caused the switch to kernel model as a raw byte:

```
  executeWithoutAllocation :: PageMap -> FaultContext -> H HByte
```

For page fault handling, we also need to add a primitive that accesses the address where the last fault occurred:

```
  readLastFaultAddress :: H HWord
```

All of the other information contained in an `Interrupt`, such as the error code of a fault, is accessible through existing H operations.

Switching to the non-allocating version of `execute` reduced the overall context-switch time by roughly 7% and the time spent in garbage collection by 20%. We use the alternative primitive in all of our tests (including our initial results). Despite these improvements, it does not appear that the use of an `Interrupt` structure is the most significant bottleneck in context switching. Pistachio completes an entire IPC in less time than a single switch into the Haskell kernel takes, so we know that we are a long way from the limits imposed by the machine. There are no obvious bottlenecks in the code, so it is not clear whether the context-switching overhead is inherent in our use of Haskell or if it is simply a product of the H design for executing user code. Further analysis and improvement of the context-switching time is an interesting topic for future work.

8.2.3 The Effect of Garbage Collection

Garbage collection is an inherent cost for a Haskell kernel. In the results of Table 8.3, we saw that garbage collection did not appear to be a significant bottleneck in IPC when compared to the other phases of the algorithm. As another perspective on the cost of garbage collection, Table 8.4 compares the average time spent in garbage collection to the average time spent in IPC overall (measured in microseconds) for each of the IPC payloads. In most cases, garbage collection takes less than 2% of the total time.

Words	**GC μs Per IPC**	**μs Per IPC**	**Time in GC**
0	0.73	53	1.38%
4	1.05	73	1.45%
8	1.36	89	1.52%
12	1.68	106	1.58%
16	2.04	126	1.62%
20	2.40	142	1.69%
24	2.70	162	1.67%
28	3.04	179	1.70%
32	3.34	195	1.70%
36	3.74	215	1.74%
40	4.06	232	1.75%
44	4.41	251	1.76%
48	4.73	268	1.77%
52	5.06	285	1.78%
56	5.39	301	1.79%
60	5.68	318	1.79%

Table 8.4: Portion of time spent in garbage collection in our initial results. By comparing the average number of microseconds spent in garbage collection to the average number of microseconds spent in IPC overall, we see that garbage collection is only 1–2% of our total IPC cost.

While 2% of IPC execution time may not seem significant, garbage collection is still important because of secondary effects that are not obvious from garbage collection profiling alone. In a garbage collected language, the programmer does not typically control the placement of data structures in memory and they may be moved during garbage collection to another address. From an algorithmic perspective the programmer does not care—their program produces the same result no matter where their data structures live. But from a performance perspective the results might be very different: moving the data structures of the running program can have significant cache effects that are invisible to the programmer except through testing.

GHC allows the user to control many configuration settings for the run-time system [91]. We experimented with two of these parameters in an attempt to understand the effects of garbage collection and to identify the optimum garbage collector settings for the L4 kernel. The "-A" flag controls the size of the initial allocation area and has a default size of 256 KB. The "-g" flag controls the number of generations and has a default setting of 2.

Figures 8.2 and 8.3 illustrate the impact that various garbage collection settings have on IPC performance. We vary the allocation area used by the collector between 256 KB and 4096 KB in combination with either 2, 3, or 4 generations. We then compare the best fit line for the data produced by each test. The best-fit line takes the form $a + bn$, where a is the base cost (in cycles) and b is the cost of transferring each additional word. Figure 8.2 illustrates the impact the garbage collection settings have on the base. Four generations and an allocation area of 4096 KB clearly produces the lowest base cost, most likely because fewer garbage collections are performed. Figure 8.3 describes the changes to the gradient of the best fit line with each setting. In this case the combination of four generations with a 4096 KB allocation area has the worst gradient while a setting of two generations with a 512 KB allocation area has the lowest cost per word.

The graphs alone do not indicate which settings are the best, but they provide us with a guide to the data sets. Using the graphs to navigate the results, we

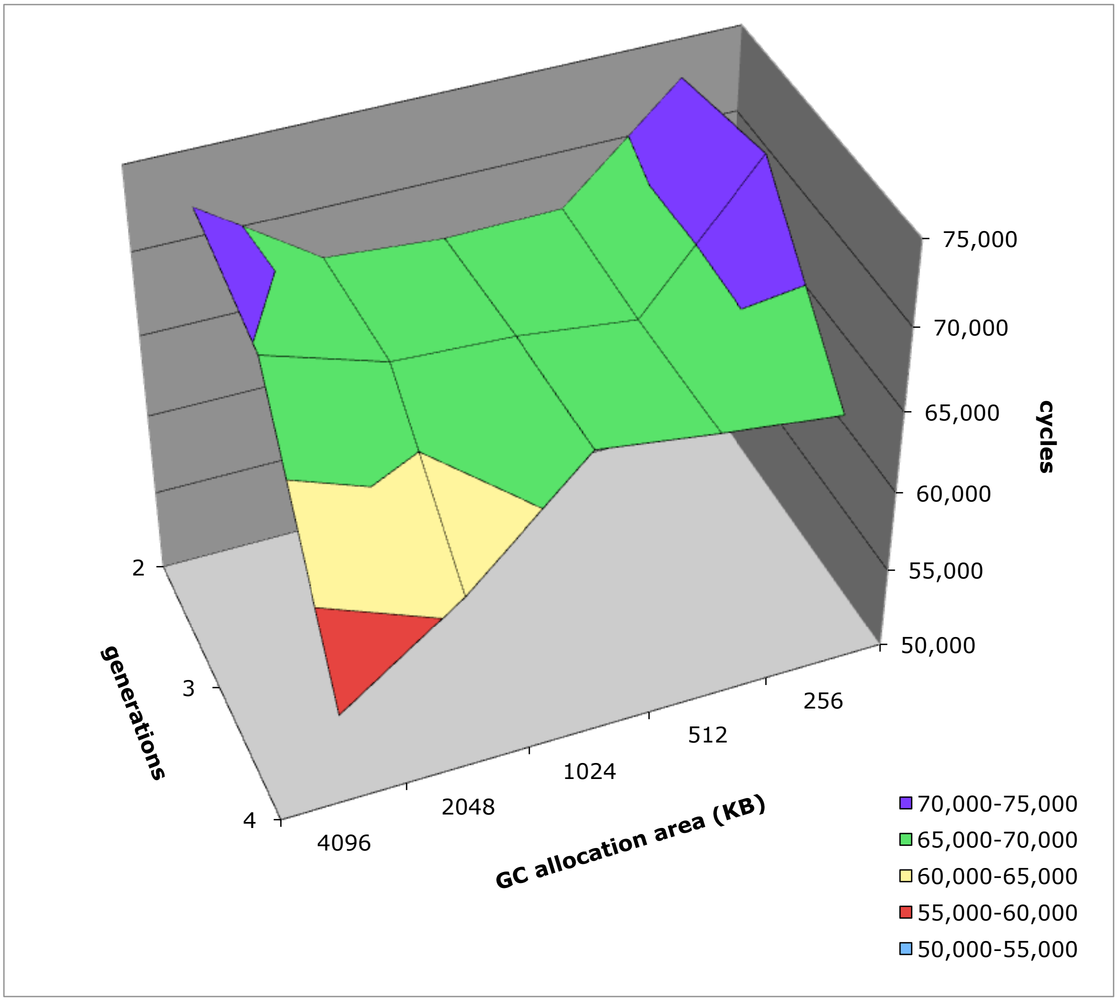


Figure 8.2: The effect of garbage collector allocation area size and number of generations on IPC. These tests measure the base-line cost, where no data is transferred (a 0 word IPC). For a best-fit line of the form $a + bn$ found using linear regression, this figure represents the impact of the parameter settings on the base number of cycles per IPC, a.

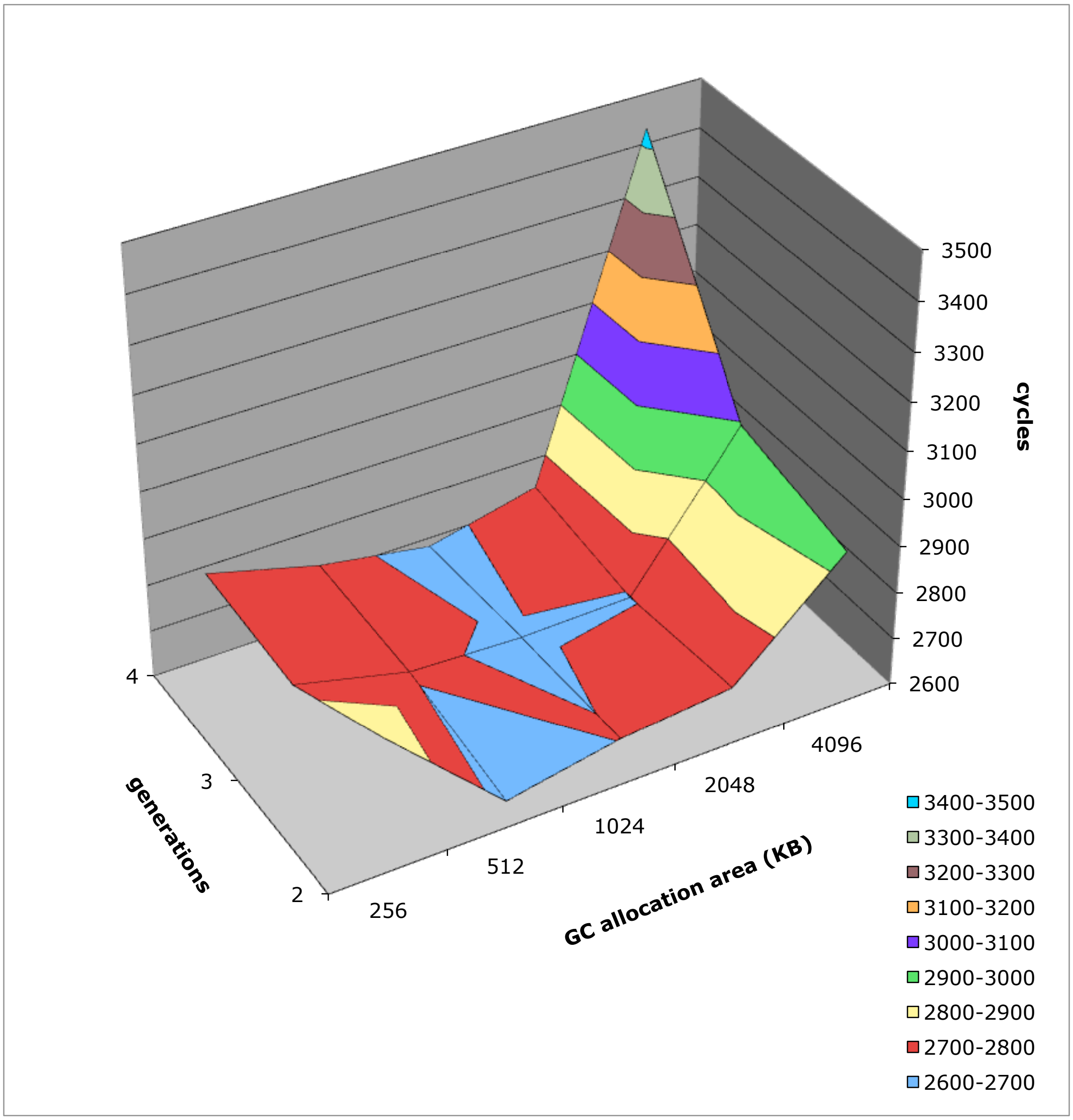


Figure 8.3: The effect of garbage collector allocation area size and number of generations on IPC. For a best-fit line of the form $a + bn$, this figure represents the impact of the parameter settings on the additional cycles needed per word of data transferred, b.

determined that an allocation area that is 512 KB large and two generations lead to the best performance for our L4 kernel. Even though garbage collection was only 2% of our overall IPC time, changing the garbage collection settings improved IPC time by 7–12% and reduced garbage collection time by 53%. This is likely due to the secondary effects of reduced garbage collection, such as better cache behavior in the client kernel. We will use the optimal settings for all of our experiments throughout the rest of the chapter.

8.3 OPTIMIZING THE KERNEL

The performance results for our Haskell kernel indicate a high base cost for our approach to safety—which includes writing the client-kernel in Haskell, accessing hardware through an additional layer of indirection, and adding safety checks that are not always present in other kernel implementations—as well as a significant per-word overhead in the implementation of transfer untyped. Not all of these costs are inherent—they indicate areas in need of optimization. In this section, we will examine the code of `transferUntyped` to identify the sources of that function's poor performance and to illustrate techniques for optimizing an H-based system. Though we examine the techniques in the context of the transfer untyped phase, they use general mechanisms for optimizing Haskell programs that should apply equally well to all aspects of the L4 implementation, but that would require additional engineering effort.

The source of performance inefficiencies in a Haskell program is often less obvious than in a program written in a lower-level language because Haskell programs are garbage collected, lazily evaluated, and written in a source language that is much more abstract than the machine they will eventually run on. In an H-based system, there are additional subtleties because issues might lie in the interface implementation, the kernel implementation, or in the design of the interface itself.

We identify three aspects of the L4 implementation to focus on when optimizing our kernel:

- **Algorithm Design:** An advantage of Haskell is that the language facilitates abstraction and allows the programmer to create concise programs that are easy to understand. However, the choice of abstractions can significantly affect performance. For example, the programmer might make heavy use of datatypes; this technique could potentially hurt performance due to construction/destruction time and garbage collection.

- **Interface Design:** In designing the primitives of the H interface, we attempted to anticipate the needs of operating systems implementers and to create the minimum sufficient interface. Unfortunately, a primitive will not always be the most performant abstraction on the first attempt. This point is illustrated by the original `execute` function, which creates an unnecessary data structure and increases our context-switching time. Once we see how the primitives get used in kernel implementations, we can revise the design and implementation of the primitives to eliminate unanticipated inefficiencies.

- **Quality of Generated Code:** GHC allows the programmer to annotate their Haskell code to control many facets of their compiled program, such as the inlining behavior of the compiler on certain functions, the representation of datatypes, and the strictness of function parameters. Targeted use of these annotations can significantly improve the performance of Haskell programs.

Attention to each of these categories is essential for achieving good performance. In the rest of this section, we examine the algorithmic complexity of the transfer untyped function (Section 8.3.1), the H facilities for copying memory (Section 8.3.2), and the use of compiler annotations within H and L4 (Section 8.3.3).

8.3.1 Identifying Algorithmic Inefficiencies

In the L4 implementation, we employ a coding style that makes heavy use of abstractions to capture common patterns and emphasize code clarity. This coding style is extremely beneficial during development: abstractions enable better type-based documentation, improve debugging, and allow localized changes to the implementation. Unfortunately, choosing the wrong abstractions can create significant performance issues. Our initial experiments show that removing certain abstractions improves the performance of IPC. Abstraction itself is not to blame; the problem is that we designed many of our abstractions before we understood the ways in which the abstractions would be used in the kernel implementation.

As a simple example, consider the implementation of `transferUntyped`. We define the message transfer function as a recursive function that transfers one word of data per call. The parameters to the function are the sender's UTCB structure, the receiver's UTCB structure, the UTCB field currently being transferred (recall from Section 7.1 that we defined a datatype with one constructor per UTCB field to document the mapping between field names and addresses), and the number of remaining words to transfer.

```
transferUntyped :: (KernelMemory m) => UTCB -> UTCB -> UTCBField
  -> HWord -> m UTCBField
transferUntyped sutcb rutcb f num
  | num <= 0  = return f
  | otherwise = do val <- readUTCBField sutcb f
                   writeUTCBField rutcb f val
                   transferUntyped sutcb rutcb (succ f) (num - 1)
```

We use the utility functions `readUTCBField` and `writeUTCBField` to access the thread UTCBs. Each iteration of `transferUntyped` moves one word and makes a tail recursive call, incrementing the field being accessed and decrementing the number of words left to transfer (the function returns when there are no words left). This code seems innocuous enough and the algorithm being implemented by the function is clear, but the per-word transfer cost of more than 7,000 cycles indicates that something costly is happening under the surface. By examining the definitions of the read and write operations on UTCB structure, the issues will become more clear.

Each UTCB structure is a pair containing the `KernelMapping` through which we can access the UTCB memory and an `Addr Virtual` describing the location of the UTCB (because each kernel-mapping can contain more than one UTCB structure).

```
type UTCB = (KernelMapping, Addr Virtual)
```

To read a field from a UTCB, we use the H function `readWordAtOffset` (introduced in Section 4.4.3).

```
readWordAtOffset  :: KernelMapping -> Offset -> H HWord
```

`readWordAtOffset` reads the value stored at a particular offset in a region of memory mapped in kernel-space. The offset must lie in the first page of the mapped region (arbitrary offsets are read with `readKernelMapping`); the `Offset` type captures the intended bounds on the offset. The utility `readUTCBField` is a straightforward wrapper for `readWordAtOffset`. The function extracts the kernel mapping that we will read from the supplied UTCB, computes the offset into the kernel mapping where the UTCB lies, and adds the UTCB offset to the offset of the message register being accessed to produce an appropriate call to `readWordAtOffset`.

```
readUTCBField :: (KernelMemory m) => UTCB -> UTCBField -> m HWord
readUTCBField utcb field
  = readWordAtOffset (fst utcb)
      (Offset (fromIntegral (utcbOffset utcb + fromEnum field)))
```

There are a few things that we should notice about this code:

- We read from the same KernelMapping each time we call readUTCBField during the transfer untyped phase of IPC (the kernel-mapping describes the sender's UTCB, which does not change). Despite this consistency, we will take the first element of the sender UTCB pair to find the appropriate KernelMapping every single time we transfer a word.
- We recompute the offset of the sender's UTCB within the kernel-mapping for each word transferred as well, even though this value is also constant.
- The UTCBField datatype adds inefficiency: we allocate unnecessary values and incur costs to convert field names back and forth to integer offsets. A datatype seemed like a nice abstraction for keeping track of the UTCB offsets, and had nice implications for safety, but we are not getting any benefit from the abstraction here, just the cost of destructing the value.

The code for writeUTCBField has exactly the same problems, plus the additional cost of checking whether or not the writeWordAtOffset function succeeded. Making this check is wasteful because we do not do anything useful when a failure does occur. writeWordAtOffset only fails if the client writes to a read-only page; once our initial testing confirms that the UTCB pages are configured correctly, we can be sure that this check will always succeed.

```
writeUTCBField :: (KernelMemory m) => UTCB -> UTCBField -> HWord -> m ()
writeUTCBField utcb field val
  = do b <- writeWordAtOffset
              (fst utcb)
              (Offset (fromIntegral (utcbOffset utcb + fromEnum field)))
              val
       if b then return () else error "error: utcb write error"
```

The readUTCBField and writeUTCBField functions demonstrate how important it is to consider abstractions in the final context in which they will be called. Neither function is problematic in and of itself (though the safety checks do make these operations more expensive that a normal read or write in C); the performance issues stem from the fact that we invoke these functions in a loop.

To avoid the repeated computations of our initial implementation, we break the abstractions for reading/writing UTCB fields and inline the calls to the functions `readWordAtOffset` and `writeWordAtOffset` in the definition of `transferUntyped`. We also eliminate the use of `UTCBField` values in the data transfer loop by incrementing the offsets directly. We remove the argument of type `UTCBField` from the `transferUntyped` function because it is unnecessary; the untyped words always start at `MR1`. Ultimately we would like to remove the `UTCBField` abstraction from the kernel entirely, but for now we leave the rest of the implementation alone and return a `UTCBField` value from `transferUntyped` for compatibility with the existing interfaces.

```
transferUntyped :: (KernelMemory m) => UTCB -> UTCB -> HWord
  -> m UTCBField
transferUntyped sutcb rutcb numleft
  = transferUntypedItem (sutcboff+1) (rutcboff+1) numleft
  where
    utcbOffset utcb = ((mask (snd utcb) utcbalign) + 256) ‘div‘ 4
    rutcbloc = fst rutcb
    rutcboff = utcbOffset rutcb
    sutcbloc = fst sutcb
    sutcboff = utcbOffset sutcb
    transferUntypedItem readaddr writeaddr num
      | num <= 0
          = return (toEnum (fromIntegral (writeaddr - rutcboff)))
      | otherwise
          = do val <- readWordAtOffset sutcbloc (Offset readaddr)
               writeWordAtOffset rutcbloc (Offset writeaddr) val
               transferUntypedItem (readaddr+1) (writeaddr+1) (num-1)
```

The structure of the algorithm is the same as the original implementation, but we now use a helper function to implement the recursion. The helper function copies a word of data from a specific source offset to a specific destination offset until all of the words have been transferred.

The performance improvement we see when moving to the inlined version of untyped data transfer is staggering. Table 8.5 shows the average cycles and microseconds per IPC, the speedup of the new implementation over the original implementation, and the comparison to Pistachio. Because we are targeting the function that copies words from the sender to the receiver, the impact of the new algorithm is more pronounced as we transfer more words. The cycle count speedup is 22% when transferring just 4 words—increasing to 75% for the peak of 60 message registers. The total time spent in garbage collection for all tests decreased by 64% in the optimized version. See Section 8.4 for a graphical comparison of all of the optimizations discussed in this section.

Inspired by the improvements we achieved through inlining, we decided to experiment with the H read and write functions themselves. Our implementation of `transferUntyped` uses the `read/writeWordAtOffset` operations from H, but these functions require us to allocate `Offset` values for describing the offset we would like to access. H provides additional read and write functions called `readKernelMapping` and `writeKernelMapping` that take normal words as offsets instead:

```
readKernelMapping  :: KernelMapping -> HWord -> H HWord
writeKernelMapping :: KernelMapping -> HWord -> HWord -> H Bool
```

`readKernelMapping` and `writeKernelMapping` may access any offset within a `KernelMapping`, not just an offset within the first page.

We reimplemented `transferUntyped` to use the functions `readKernelMapping` and `writeKernelMapping` for transferring data from the sender to the receiver, hoping to save cycles on the construction, destruction, and garbage collection of

Words	**Cycles**	μs	**GCs**	**Speedup**	× **P**
0	86,336	52	323	-1%	68
4	92,505	56	339	22%	69
8	95,295	57	357	36%	71
12	98,775	60	374	45%	73
16	101,386	61	390	51%	75
20	104,362	63	406	56%	76
24	107,497	64	423	60%	78
28	110,250	67	440	63%	80
32	113,421	68	457	65%	82
36	116,303	70	473	67%	83
40	119,165	73	491	69%	85
44	122,169	74	507	71%	87
48	125,564	75	523	72%	88
52	127,909	77	540	73%	90
56	130,726	80	561	74%	91
60	134,316	81	573	75%	93

Table 8.5: The result of inlining the read/write word functions from the H interface directly into our inner loop. The speedup column compares the cycles per message in the inlined implementation to the performance of the unoptimized version. As in Table 8.2, the "× P" column indicates how many times slower the algorithm under test is than Pistachio.

`Offset` values. The new code was exactly the same as our optimized implementation except for the use of these functions. Surprisingly, this simple change was up to 5 times slower than the version using `readWordAtOffset` and `writeWordAtOffset`, most likely due to the fact that the bounds checks are based on dynamic constants stored in records rather than static constants. It is also possible that we are not paying a high cost for `Offset` values anyway because of compiler optimizations. Finally, there are implementation differences that make `readWordAtOffset` and `writeWordAtOffset` more strict. We will see an example in the next section where strictness alone causes a similar performance disparity between two small functions. Our experiment demonstrates that subtle choices in the API design and implementation can significantly impact the performance of the client kernel. In the next section we will specifically examine the effects of an optimization at the interface level to improve our performance even further.

8.3.2 Role of H Primitive Design in Performance Results

The design choices we make in the H interface strongly influence the algorithms that will be used in the client kernel because the client cannot access low-level hardware features via any other mechanism. Sometimes our design choices affect the memory-safety argument for H in an essential way and therefore cannot be optimized significantly. More often though, the design decisions reflect an attempt to anticipate the needs of kernel implementers, and they do not need to be set in stone. In this section, we describe the addition of a new primitive to the H interface to demonstrate the impact of interface design on performance and the ease with which we can extend the interface.

In our previous definitions of `transferUntyped`, we always copied one word of data at a time from the sender to the receiver using the read and write functions of the H interface. In each call to read or write, we must check that the offset being accessed falls within the first page of the `KernelMapping` argument in order for the access to be safe. (The kernel-mapping may be longer than one page, but these functions are optimized to use static checks; every kernel-mapping is at least one page long.) The `writeWordAtOffset` performs an additional check on the permissions of the mapping to ensure that we do not page fault in kernel-mode. Thus, we incur two checks for every word we read and three for every word we write.

```
readWordAtOffset  :: KernelMapping -> Offset -> H HWord
readWordAtOffset km off@(Offset offset)
  | off >= minBound && off <= maxBound
    = liftIO $ peekElemOff (kernelAddress km) (fromIntegral offset)
  | otherwise = ...

writeWordAtOffset :: KernelMapping -> Offset -> HWord -> H Bool
writeWordAtOffset km off@(Offset offset) value
  | (kernelPerms km) .&. w /= 0 && off >= minBound && off <= maxBound
    = do liftIO $
           pokeElemOff (kernelAddress km) (fromIntegral offset) value
         return True
  | otherwise = ...
```

The bounds checks are necessary to enforce memory-safety, but the sheer number of checks we perform during an IPC transfer suggests that we do not have the right interface primitive for the kernels that we want to implement.

The heart of the problem is that we must validate every memory access independently. If we copy the data in a block, rather than one word at a time, we can validate the entire data transfer with just a few bounds checks. We do not need to check every intermediate memory access if the starting and ending offsets both fall within the bounds of the kernel mapping that is being accessed. Recognizing this, we introduce a new H primitive called `memcopy`, similar to the standard `memcopy` operation available in C, that provides the ability to copy multiple words from a source kernel-mapping to a destination kernel-mapping with a single call.

The safety guarantees are exactly the same as with a bulk copy implemented via multiple calls to `read/writeWordAtOffset`.

```
memcopy ::
  KernelMapping -> HWord -> KernelMapping -> HWord -> HWord -> H ()
memcopy source sstart dest dstart numwords
  | ((kernelPerms dest) .&. w) == 0
    || sstart < 0 || dstart < 0
    || sstart + numwords > pageSize - 1
    || dstart + numwords > pageSize - 1
    = error "bad parameters to memcopy"
  | otherwise
    = liftIO $ loop soffset doffset numwords
        where
          loop :: Int -> Int -> HWord -> IO ()
          loop s d n = if n == 0 then return ()
                          else do val <- peekElemOff sourceaddr s
                                  pokeElemOff destaddr d val
                                  loop (s+1) (d+1) (n-1)
          sourceaddr = kernelAddress source
          destaddr   = kernelAddress dest
          soffset    = fromIntegral sstart
          doffset    = fromIntegral dstart
```

The `memcopy` function takes five arguments. The first two describe the source of the data to be copied: the source includes the `KernelMapping` where the data will be read from and the offset from which to begin copying data. Similarly, the destination includes the `KernelMapping` where the data will be written to and the offset where `memcopy` should begin writing. The final argument specifies the number of words to copy.

We redefined `transferUntyped` yet again to use `memcopy` instead of the per-word read and write functions that we used to copy data between the sender and the receiver in the optimized version of Section 8.3.1. Table 8.6 shows the

Words	**Cycles**	μ**s**	**GCs**	**Speedup**	× **P**
0	90,776	56	336	-5%	71
4	91,408	54	336	1%	69
8	91,510	56	336	4%	68
12	91,470	54	336	7%	68
16	92,193	56	340	9%	68
20	92,348	56	341	12%	68
24	92,659	56	343	14%	68
28	92,922	56	345	16%	67
32	93,236	56	347	18%	67
36	93,556	56	349	20%	67
40	93,651	57	351	21%	67
44	94,024	56	352	23%	67
48	94,353	56	354	25%	66
52	94,572	57	355	26%	66
56	94,805	57	357	27%	66
60	94,768	57	359	29%	66

Table 8.6: Performance of IPC when using a multi-word copy to transfer IPC messages instead of individual read/write word functions. The speedup column compares these results to the inlined implementation in Section 8.3.1.

performance of IPC after this change, including a calculation of the speedup of IPC compared to the previously optimized results. As we transfer more words, the impact of the multi-word copy becomes more significant. This is to be expected because the issue with the old primitives was the high per-word transfer cost that they imposed on IPC. The cost of the `memcopy` version is higher when we do not transfer any words because the implementation unnecessarily incurs the checking cost when no data will be transferred. This is easily fixable by modifying the caller of `transferUntyped`—the new version should only be called when there is actually data to be copied across.

Though the `memcopy` operation is specific to the `transferUntyped` routine, adding a new primitive to optimize the performance of a common operation is a general technique for optimizing an interface. This is one of the generic techniques that we will consider when pursuing future efforts to improve the performance of L4.

8.3.3 Using Compiler Pragmas to Fine-Tune Performance

GHC performs many optimizations while compiling Haskell code that are not always obvious to the programmer from looking at the source. To give the programmer more control over the program that will actually run, GHC provides a wealth of compiler annotations that influence the behavior of the compiler directly and indirectly during the optimization phase. In this section, we examine a few of the techniques for controlling the behavior of GHC-generated code: strictness annotations, data structure unpacking, and inlining pragmas. The full selection of annotations is described in the GHC user's manual [91]. We apply these techniques to our kernel, focusing on the transfer untyped phase of IPC, and examine the performance results.

Strictness annotations are a very powerful, albeit subtle, mechanism for controlling the behavior of a Haskell program. They allow the programmer to force the evaluation of a value whose computation might otherwise be delayed until a later point in the program. A strictness annotation is written `!`. Some changes to the strictness of a Haskell program inherently improve performance, but the real importance of strictness annotations comes from the secondary effects that they induce. Properly placed strictness annotations can enable better function inlining and more constructor elimination by the compiler. The presence or absence of those optimizations can be significant. For example, a bug in our initial implementations of `readWordAtOffset` and `writeWordAtOffset` made those functions lazy in the Offset parameter (recall that these functions take a KernelMapping and the Offset in that mapping to access). This strictness difference alone slowed down our version of transferUntyped with these functions inlined (presented in Section 8.3.1) by up to 248%. The garbage collection time in the lazier version went up by 220%. All because of a strictness difference in two parameters!

To illustrate the use of strictness annotations in a Haskell program, we will examine the definition of memcopy that we presented in the previous section. We focus on the definition of the inner loop that actually transfers the data.

```
loop :: Int -> Int -> HWord -> IO ()
loop !s !d n = if n == 0 then return ()
               else do val <- peekElemOff sourceaddr s
                       pokeElemOff destaddr d val
                       loop (s+1) (d+1) (n-1)
```

Our original definition of loop did not include strictness annotations on s and d. Although it appeared that we were creating a strict function that uses the source address s to read a value and the destination address d to write a value, we were actually building a delayed computation (called a thunk). We need to force the evaluation of s and d explicitly, as shown in the new definition of loop, because otherwise these values will not actually be demanded within the loop code.

Another annotation available with GHC is the UNPACK pragma. We use unpack annotations to signal the compiler to unpack the contents of a constructor field into the constructor itself, effectively eliminating a level of indirection. Unpacking is the analog of inlining for data structures. This pragma is always used in conjunction with a strictness annotation. As an example, consider the definition of the KernelMapping datatype that we rely on in the definition of memcopy.

```
data KernelMapping = KernelMapping {
                       kernelFpage   :: {-# UNPACK #-}!(Fpage Virtual),
                       kernelRegion  :: {-# UNPACK #-}!PhysicalRegion,
                       kernelPerms   :: {-# UNPACK #-}!Perms,
                    kernelAddress :: {-# UNPACK #-}!(Ptr HWord)
                  }
```

Each field of the KernelMapping record can be unpacked to inline the definition and unbox the value into the definition of KernelMapping. Thus, when we access a field of KernelMapping, such as kernelFpage, we will be able to use the value of the field directly. Unpacking is not always a win though: if an unpacked data structure is used in an unoptimized function, the values may have to be reboxed before they can be used.

The final compiler pragma we will discuss in this section allows the programmer to specify inlining behavior for a particular function. For example, we could specify

```
{-# INLINE memcopy #-}
```

to tell the compiler to inline the memcopy function or

```
{-# NOINLINE memcopy #-}
```

to signal that we would not like memcopy to be inlined. In our experience, the compiler typically does a much better job inlining when you do not use an INLINE pragma. For example, in the memcopy example, we observed a performance degradation when we added an inline pragma.

Despite the bad results we experienced by explicitly inlining memcopy, there are places in our L4 implementation where inlining pragmas are helpful. For example, the L4 kernel utilizes a plethora of state monad get/set variants for accessing specific components of the L4 state. These functions cannot always be inlined because of overloading. The readThread utility reads a Thread structure from the thread map of the L4 system state using its thread id. We can add an inline pragma to the definition as follows.

```
{-# INLINE readThread #-}
readThread :: (StateMonad System m) => ThreadId -> m (Maybe Thread)
readThread !tid = do sys <- get
                     return (lookup (identifier tid) (threadMap sys))
```

Words	Cycles	μs	GCs	Speedup	× P
0	77,233	46	269	15%	61
4	77,875	47	272	15%	58
8	78,356	47	273	14%	59
12	78,691	47	275	14%	58
16	79,485	47	277	14%	59
20	79,993	49	280	13%	59
24	80,342	49	281	13%	59
28	80,386	49	282	13%	58
32	81,160	49	284	13%	58
36	81,852	49	285	13%	59
40	82,386	50	288	12%	59
44	82,795	50	289	12%	59
48	83,372	50	291	12%	59
52	83,827	50	293	11%	59
56	84,428	52	295	11%	59
60	84,766	50	296	11%	59

Table 8.7: Performance of IPC when using strictness annotations and compiler pragmas for datatype unpacking and inlining in the essential components that affect IPC. The speedup column compares these results to the multi-word copy implementation in Section 8.3.2.

Our experiments indicate that explicitly inlining `readThread` improves the overall performance of IPC. We also add a strictness annotation to the thread identifier argument to prevent the computation from being delayed.

Table 8.7 summarizes the results of a variety of small optimizations through compiler annotations. The most significant changes are making the inner loop of `memcopy` strict and using unpacked strict fields in the data structures that `memcopy` relies on. This is the only area where we invested focused effort on optimization. We also added strictness annotations to some of the scheduling and state monad read/write utilities, but we did not optimize the calling code of any of these functions so the optimizations will have limited impact. We observe modest performance improvements over the unoptimized memcopy-based implementation in Section 8.3.2. Our final results indicate that our kernel is about 60 times slower than Pistachio, although there is still plenty of room for improvement by optimizing the other phases of IPC.

So far, our optimization efforts have mostly affected the transfer untyped phase of IPC. Though we focused on optimizing the transfer of untyped words, the compiler annotations do impact the performance of other phases of IPC. A difficult aspect of optimizing Haskell code by trying to control the compiler is that the annotations do not always improve performance. Table 8.8 shows the breakdown of the cycles spent in each phase of IPC: many parts of IPC are faster than the original implementation but some are slower. These results are comparable to Table 8.3 in Section 8.2.1. One notable result is that the overhead is much lower and more consistent than in the initial measurements. In general, the increased strictness we added through our annotations made the cost per phase much more consistent, even in phases where the performance got worse (such as thread restart).

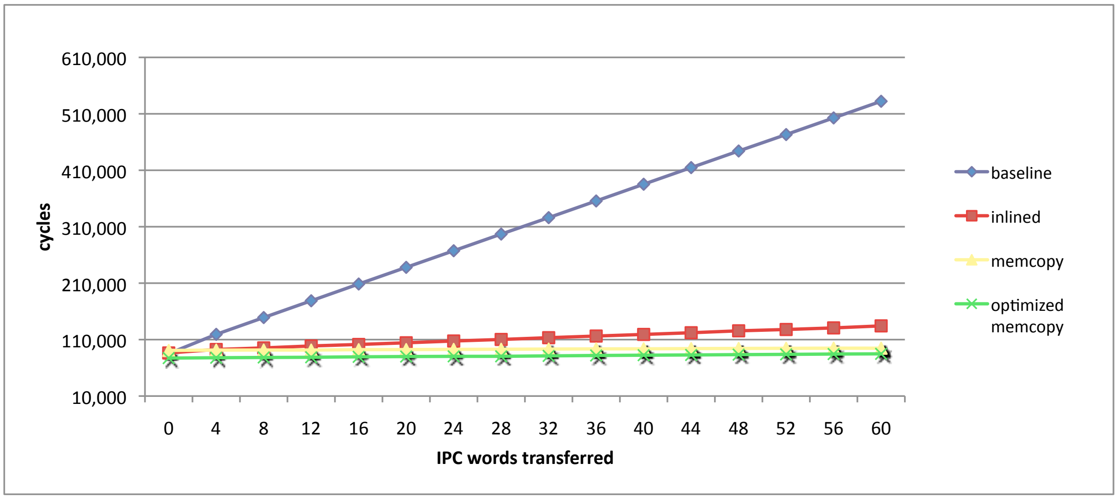
8.4 SUMMARY

Sections 8.3.1, 8.3.2, and 8.3.3 describe incremental optimizations to our original IPC implementation. In this section, we evaluate the overall effectiveness of our optimization techniques by comparing the performance of each variant of IPC. We also compare the cycles spent in each phase of IPC in our final implementation to the initial IPC breakdown from Section 8.2.1.

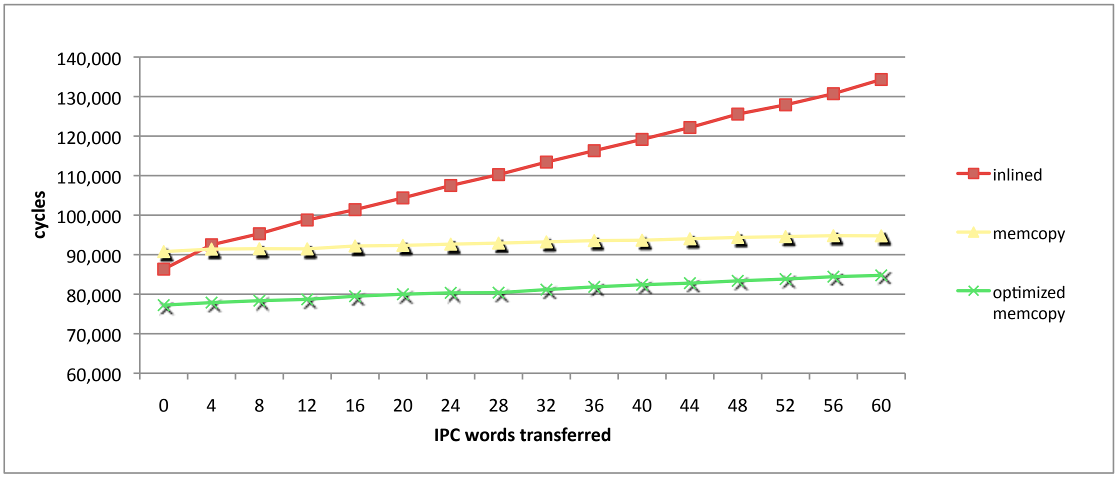
Figure 8.4 illustrates the performance of each IPC algorithm. Figure 8.4(a)

Words	**OH**	**Sch.**	**SR**	**TM**	**TU**	**Rst.**	**Rcv.**	**GC**	**Total**
0	3,102	22,742	6,140	17,549	474	18,412	16,688	920	86,027
4	3,143	22,605	6,073	18,408	482	18,456	16,686	932	86,785
8	2,600	22,779	6,069	18,759	490	18,574	16,593	948	86,812
12	2,790	22,712	6,150	19,209	489	18,583	16,597	941	87,471
16	2,120	22,990	5,973	20,510	507	18,810	16,418	917	88,245
20	3,075	22,865	6,085	20,398	493	18,557	16,644	931	89,048
24	2,754	22,760	6,077	20,800	485	18,564	16,612	940	88,992
28	2,654	22,859	6,115	21,252	482	18,500	16,607	977	89,446
32	2,657	22,953	6,064	21,681	486	18,529	16,617	974	89,961
36	2,788	22,803	6,150	22,180	481	18,439	16,615	979	90,435
40	2,932	22,757	6,079	22,639	479	18,477	16,591	990	90,944
44	2,487	22,782	6,096	23,033	496	18,598	16,641	984	91,117
48	2,710	22,724	6,139	23,669	494	18,526	16,622	988	91,872
52	2,749	22,797	6,019	24,084	484	18,568	16,609	995	92,305
56	2,620	22,803	6,126	24,531	500	18,642	16,591	1,008	92,821
60	3,247	22,867	6,096	25,214	483	18,668	16,601	1,006	94,182

Table 8.8: A breakdown of the cycles spent in each phase of the optimized IPC implementation. Our optimization efforts targeted the transfer untyped (TU) phase of the algorithm, and we see a significant reduction in the number of cycles as compared to the original breakdown we saw in Table 8.3. OH stands for overhead and includes the time spent to context-switch between the Haskell kernel and the C user program. Sch. is the time spent in the scheduling loop. SR stands for send rendezvous, TM stands for transfer message, and TU stands for transfer untyped. Rst. describes the phase of IPC spent restarting the thread. Rcv. is the time spent in the receive phase of the IPC operation. The phases are explained in detail in Section 8.1.1.



(a) All experiments, including the unoptimized baseline.



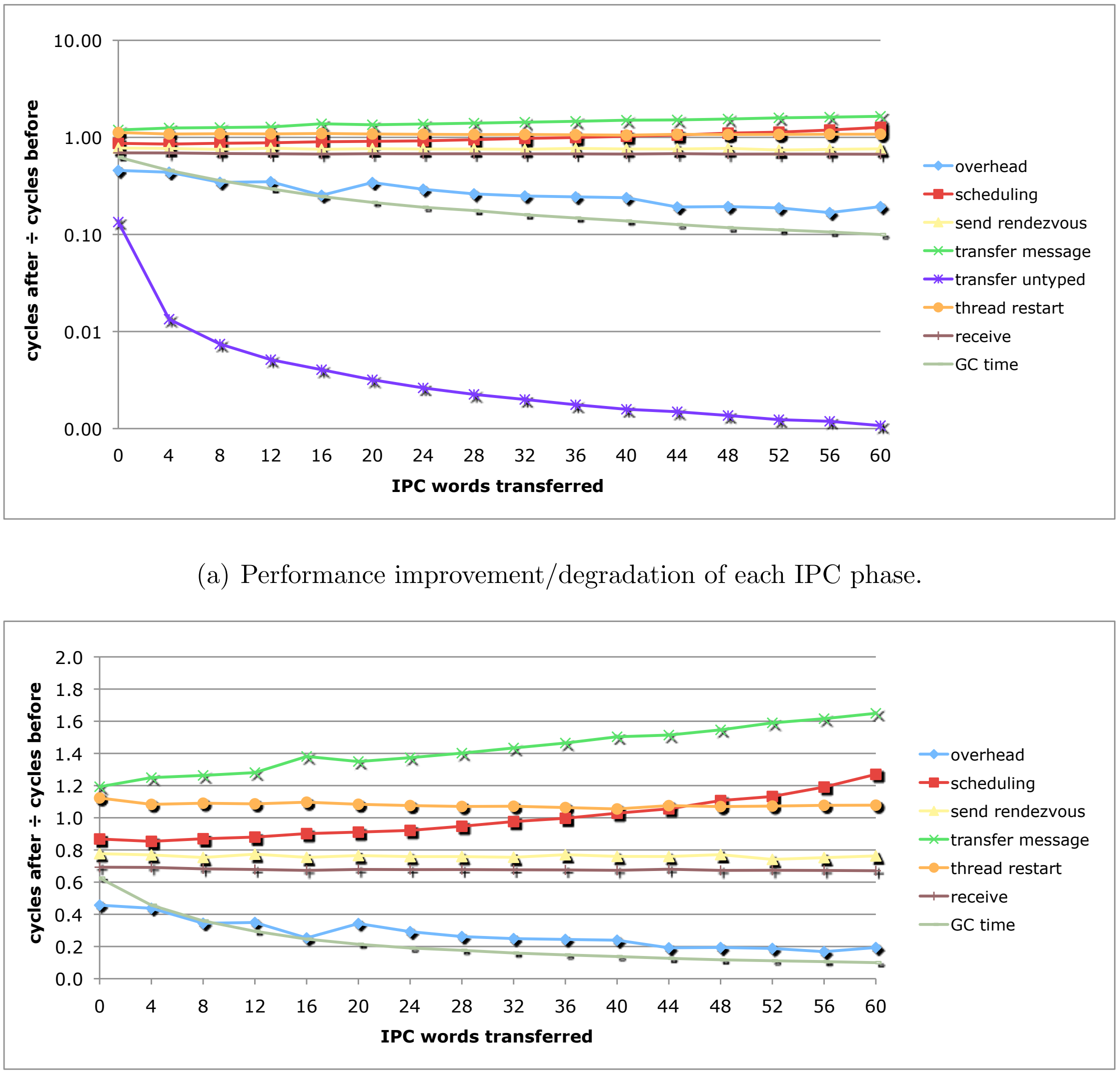
(b) The results of the optimized IPC implementations.

Figure 8.4: The effects of each optimization presented in this chapter on the overall performance of IPC as measured by the ping-pong benchmark. Figure 8.4(a) demonstrates that all of our optimizations achieve an improvement over the initial, unoptimized implementation. Figure 8.4(b) focuses on the optimizations alone to more clearly demonstrate the effects of each modification.

includes our initial results. From this graph, it is clear that each algorithm provides an improvement over the unoptimized implementation. Figure 8.4(b) only shows the optimized versions to make a comparison between these three versions easier. As one would hope, each subsequent optimization performs better than the previous implementation. Note, however, that the cost per word is higher in the optimized memcopy implementation than in the normal implementation of memcopy; it appears from the graph that if enough words are transferred then memcopy will eventually outperform the optimized version. This phenomenon is likely caused by unexpected side effects from our unpacked data structures. Unpacking data type definitions has the potential to speed up computations, but can cause a slowdown when the calling code is not also optimized. Because we focused on a single routine, we may have introduced additional inefficiencies in other phases of the IPC algorithm. The per-phase breakdown of IPC costs in Table 8.8 is consistent with this analysis.

Figure 8.5 illustrates the performance change between the the initial IPC measurement and the most optimized version. Each line in the graph represents a single phase of the IPC algorithm. The x-axis is the number of message registers that were sent in a particular test and the y-axis is the ratio between the final results and the initial results. Even though each optimization pass improved the overall IPC performance, some phases of the algorithm perform worse in the optimized version.

Unsurprisingly, the transfer untyped phase where we spent all of our optimization energy performs much better in the optimized implementation than in the initial implementation. The time spent on garbage collection and overhead is also lower. The reduction in overhead may be due to the increased strictness of the optimized program. In the original implementation, any computations that were delayed due to laziness were ultimately forced when we switched back to user-mode (if not before). As such, computations that had nothing to do with overhead or



(a) Performance improvement/degradation of each IPC phase.

(b) Performance change in unoptimized phases.

Figure 8.5: The overall effect of optimization on each phase of the IPC algorithm. Figure 8.5(a) illustrates the ratio between the cycles taken per IPC in the final implementation and the original implementation. Transfer untyped—the function where we focused our optimization energy—improves dramatically while the impact on other phases varies (some perform slightly better and some perform slightly worse). Figure 8.5(b) focuses on the phases of IPC other than transfer untyped. We see the most improvement in the number of cycles taken for overhead and garbage collection.

context-switching could appear in this phase of IPC. The portions of IPC devoted to sender rendezvous and the receive phase also improve somewhat, while thread restart is not particularly effected. Scheduling is faster on small numbers of message registers and slower for large numbers of message registers. Transfer message is the only phase that performs significantly worse in the optimized version—this is the obvious place to look next for further optimization opportunities. Because transfer message is using all of the same data structures as transfer untyped, but is not optimized to allow the kernel to make use of the strictness and unpacking annotations, the performance of this routine is the most negatively affected by the changes we made to improve the performance of transfer untyped. Further improvements to `transferUntyped` may also be possible.

The techniques we presented in Section 8.3 demonstrate that it is possible to improve the performance of an H-based system dramatically through targeted optimization. Section 8.3.1 examined the algorithmic behavior of the untyped data transfer function and confirmed that we can obtain large performance improvements through refactoring. The rest of the L4 kernel is coded in a similar style and makes heavy use of abstractions throughout: both in functions that capture common patterns (potentially introducing redundant computation) and in data structures that provide type-based documentation to help eliminate mistakes (increasing the cost of storing and manipulating simple data like words). Now that we have working H interface and L4 implementations, there is potential to reconsider these abstractions to increase performance. Further performance gains may be possible by adding additional strictness annotations and by experimenting with inlining and unpacked datatypes as in Section 8.3.3. Though we are not currently aware of any API issues, a targeted optimization of a particular IPC phase or other kernel function might unearth more inefficiencies. The experiments that we have presented in this dissertation provide strong evidence that, by pursuing algorithmic, API, and compiler optimizations, we have the potential to further reduce the performance gap between our Haskell kernel and typical L4 implementations.

Chapter 9

RELATED WORK

In this chapter, we relate the concepts of the H interface design and implementation to existing work in the fields of operating systems and programming languages. We focus on the core topic areas of system architecture (Section 9.1), functional operating systems (Section 9.2), programming language environments (Section 9.3), verified operating systems (Section 9.4), safe operating systems (Section 9.5), and virtualization (Section 9.6).

9.1 SYSTEM ARCHITECTURE

Separation between policy and mechanism is a fundamental concept in many modern systems, including H. The Hydra system is an important example of this concept for the operating systems domain [68]. The primary goal of the Hydra design is to enable user-processes to control resource allocation policies for the system. User-processes cannot be given free reign because of safety and fairness issues, so the kernel mechanisms are designed to balance control and safety. For example, Hydra implements a parameterized scheduler. Key scheduling parameters, such as the amount of time that each process should run before being stopped, are provided by per-process user-level schedulers. The mechanisms introduced in Hydra were very influential on future systems, particularly the microkernel community, and very similar mechanisms in support of user-level policies appear in kernels like L4 [62] and Mach [2].

The H interface aims to strike much the same balance between policy and mechanism in a setting where the client is itself an operating system. We strive to minimize policies in H that are not directly related to safety because any policy in H restricts the applicability and flexibility of the abstraction layer. Much of our design effort was devoted to the mechanisms that support client control over page-map pages and the kernel virtual address-space. Providing these mechanisms in a safe way presents a challenge, and guaranteeing that the client policies do not introduce safety violations is a major theme of our safety analysis.

Unlike Hydra, we do not need to manage the complexity of multiple competing policies because H is designed for use by a single client at any given time. In Hydra, each user-process might set its own policy for resource usage. Hydra must be an arbiter between these processes to ensure safety and fairness. In some ways, the competition between user-processes is analogous to the competition for resources between the environment, the H interface, and the client in our system. The difference in our setting is that the run-time system and H interface resource allocation policies are fixed; only the client's memory usage policy will vary at run-time.

9.2 FUNCTIONAL OPERATING SYSTEMS

The idea of applying functional programming languages to the operating systems domain has a long history. Many of the early examples represent a very different era in pure functional programming before monadic I/O was incorporated into programming practice. As such, the focus of these works is on mechanisms for handling effects, using techniques for stream processing and I/O in a continuation-passing style. These challenges are very different from those we encounter today. Early examples of functional operating systems include Nebula [58] and the Kent Applicative Operating System (KAOS) [17, 89].

The Hello [26] project implements an operating system in Standard ML and addresses various language design and efficiency issues, such as how to access hardware devices and how to handle interrupts in a garbage-collected language. It builds on the results of the Fox project, where Standard ML was used for systems programming. In particular, it includes FoxNet, an efficient implementation of the TCP/IP protocol stack [8]. Compared to these projects, a significant new feature of the H interface is its support for controlling memory management hardware, which allows us to run code written in other languages safely.

Though unpredictable performance results sometimes occur in Haskell due to laziness, it is possible to write lazy functional programs that satisfy real-time guarantees. The Embedded Gofer project extended the Haskell language specifically to support programming embedded devices [98, 97]. The language extensions provide access to I/O device registers and asynchronous exception support for handling interrupts, as well as the implementation and adoption of an incremental garbage collector to satisfy the guarantees that are necessary in real-time environments. The techniques employed in Embedded Gofer are a useful guide if we extend H to support real-time guarantees. For performance reasons alone, experiments with different garbage collector implementations would be interesting.

9.3 PROGRAMMING LANGUAGES

Some people consider the use high-level functional languages to implement operating systems to be too much of a departure from traditional systems programming techniques. A less radical approach is to add safety to a low-level language. There are a number of programming languages that support a greater degree of safety than C, while still supporting the low-level and imperative features that are traditionally associated with operating systems programming. The nature of the safety guarantees and the programming constructs available varies widely with these safe systems languages.

Cyclone was not designed specifically for systems programming, but as a safe replacement for C. Cyclone's focus is on maintaining the transparency and control that programmers enjoy with C, including direct control over resources such as memory, while providing strong type- and memory-safety guarantees [36, 35]. Just as in Haskell, Cyclone prevents common memory-safety errors like buffer overflows and null pointer dereferences. Certain properties, like bounds checks on arrays, must be dynamically checked at run-time. A great feature of Cyclone is that run-time checks are eliminated by the compiler whenever possible, for example, when an array access can be statically determined to be in-bounds. Cyclone achieves safety without using garbage collection by employing a region-based type-system that manages the scope and life-span of pointers.

Cyclone provides the same level of protection against memory-safety errors as higher-level languages like Haskell. A Cyclone programmer cannot dereference a null pointer or free memory twice, but still maintains a high degree of control over resources. Implementing an operating system in Cyclone would not require a special interface like H because the language already supports direct control over resources in a safe way. However, the existence of the H interface allows us to express memory-safety guarantees that are specific to the operating systems domain (as we saw in Chapter 5); these kinds of properties are not captured automatically by Cyclone's language facilities. We could write an H-like abstraction layer in Cyclone, but Haskell provides many important features that we utilized in the construction of H. The Haskell type system is more expressive than the type system of Cyclone; when combined with Haskell's module system we are able to create a strong abstraction barrier between the H internals and the client operating system. This abstraction barrier is an essential part of our memory-safety argument. Because Cyclone provides the power of C in a memory-safe language, we could greatly increase the assurance argument for H by recoding the low-level portions of our implementation in Cyclone instead of C.

On the other end of the spectrum is Modula-3—a type-safe, object oriented language with garbage collection that was designed specifically for systems programming [74]. Like Haskell, Modula-3 supports isolation between components via a powerful module system. Potential safety violations introduced by the user of unsafe code, like foreign function calls, are explicitly documented in the program. Modula-3 supports the implementation of safe operating systems like Spin [6], which we will cover in Section 9.5. Though Modula-3 is a powerful language for safe systems programming, its imperative nature and lack of a pure semantics makes Modula-3 programs less amenable to reasoning.

The BitC programming language aims to cover the middle ground by including many of the features traditionally associated with high-level languages in a way that will be amenable to systems programming [9, 86, 85]. Resource control and transparency are central to this philosophy. For example, BitC provides machine-level, fixed size representations of types and control over data layout. BitC supports garbage collection, but also allows the programmer to write code that does not perform dynamic allocation. Another goal of BitC is assurance; the designers intend to develop a formal mechanized semantics for the language. Other features of BitC include polymorphism, datatypes, pattern matching, and higher order procedures. Though BitC is not pure, it is type-safe and supports type classes. When BitC matures it will be a viable platform for developing safe operating systems (see the discussion of Coyotos in Section 9.5).

Another interesting line of related work relates to foreign function interface design. The construction of safe foreign function interfaces avoids the kinds of vulnerabilities that necessitate the use of an abstraction layer to support memory-safe low-level programming in high-level languages. One approach is to type check unsafe foreign code as if it were code in the safe native language to catch errors that would otherwise be permitted by the foreign type system. This approach has been applied to OCaml's C foreign function interface and successfully uncovered a number of bugs in existing foreign code [27]. An interesting challenge arises from the fact that C code can observe that there are values of different types with the same representation. The designers of the OCaml to C system use type inference to determine what OCaml type should be assigned to a value based on its usage in C. They use dataflow analysis to ensure that C code properly registers any pointers it has into the OCaml heap and makes the appropriate dynamic checks and offset calculations when accessing OCaml datatypes. Such a system would greatly improve our ability to assure the behavior of the foreign calls that we use in the implementation of the abstraction layer, and might even eliminate the need for such an abstraction layer. The major complexity that we foresee relates to the fact that our foreign calls can change the system state in ways that are not accounted for in the OCaml to C type system, such as modifying a page-table.

The Scheme community has a long history of trying to incorporate operating systems features into the run-time system of the language. This is a different exercise than trying to expose operating systems constructs built on top of a (nearly) standard run-time system, but must address many of the same safety and protection issues. The DrRacket (formerly DrScheme) programming environment supports GUI services, a notion of protection domain specifically related to GUI elements (the integrity of the GUI of the environment itself must be maintained), the ability to halt a program, and the ability to reclaim a program's resources if it has gone awry [24]. Wick and Flatt added support for process-based memory accounting in a shared heap [99]. These are higher level concerns than we address with the H interface because they deal with issues of process management, but it is interesting to note that the division between environment structures and client structures is a key distinction in both systems.

A variety of techniques have also been developed in the Java community for adding operating system support to the language in a safe way. Java operating systems can provide the same level of memory-safety as Haskell, and in theory encounter similar challenges to reconcile control over resources with garbage collection and other run-time system services. In practice, the mechanisms necessary to overcome these challenges are very different between Java systems and the H interface because the interface that the language run-times present to the programmer are so different.

The KaffeOS operating system, like DrScheme, treats the language run-time as the operating system kernel [4]. KaffeOS extends the normal Java run-time system with support for resource management, isolation between software components, and shared memory based communication. Essentially, this creates a notion of process in Java. KaffeOS makes use of the user/kernel CPU modes to protect kernel objects from being accessed by user processes. KaffeOS has the power to manage CPU and memory resources, to terminate a process, and to reclaim a process' memory upon termination, much like a traditional operating system.

The high level of the interface to DrScheme and KaffeOS means that many policy decisions (such as when to terminate a thread or how to partition resources) are made inside the respective run-time systems. This characteristic restricts the ability to write general purpose operating systems in either setting without modifying the run-time core itself. By contrast, these policy decisions are implemented in the Haskell client of an H-based system. No modification of the H implementation or the Haskell run-time system is required. The services exposed by H are designed to facilitate writing operating systems kernels in a safe language with minimal special purpose support from a run-time system. Process separation is supported in the traditional way using hardware-based protection techniques. It would be interesting to combine the techniques for run-time system supported processes with the techniques from H for memory-safe operating system construction to obtain the benefits of memory- and type-safety at every level of the system.

Luna also introduces a notion of separated tasks in Java, but unlike KaffeOS, Luna relies fully on the type system to guarantee isolation [45]. The boundaries between software processes are expressed in the types, making distinctions between tasks explicit. A major contribution of Luna is the design and implementation of a mechanism for communicating between tasks that supports accurate resource accounting and maintains strong isolation boundaries. Inter-task communication happens via a special type called a remote pointer; these pointers allow data objects to be shared between tasks but can be dynamically revoked at any time. Luna is implemented through a combination of source-level and run-time system extensions, with most of the support for the new features being provided by the RTS. Though Luna addresses a very different aspect of systems programming than H—inter-process communication rather than virtual-memory management—both systems face the challenge of integrating operating systems features into a safe language. The Luna designers heavily emphasize the notion of "types as capabilities" that is at the heart of our design for H. Remote pointers provide a mechanism for revocation that, if available for data values in Haskell, would eliminate the need for many of the dynamic status checks in the H implementation.

The Sing# language—a type-safe, garbage collected extension of C#—has also been used for operating systems development. We delay discussion of this topic until Section 9.5 where we present the Singularity operating system [22, 48], which is implemented in Sing#. The Habit programming language [90], which is a Haskell-like language designed specifically for systems programming, is highly relevant to our work on the H interface, but we delay our discussion of it until Chapter 10 because the the concepts are Habit are tightly integrated with our planned future work.

9.4 VERIFIED OPERATING SYSTEMS

The seL4 microkernel [60] is closely related to our work in that both projects share the goal of creating formally verified operating systems. The difference between the two projects lies in the approach. The seL4 team achieved the ultimate goal of verifying a complete operating system using a traditional implementation of their operating system design in C. That verification was a refinement proof of an abstract functional specification to the high-performance C implementation via an executable model written in Haskell. The only problem with this approach is the cost, approximately 20 person years of effort, and the lack of potential for reuse [60]. Our approach is to demonstrate that we can obtain a subset of the memory-safety properties proved by the seL4 team automatically by using a pure functional language for operating systems implementation rather than a low-level and non-memory-safe language like C.

We are hopeful that the techniques used by the seL4 team to verify their operating system could also be used to verify our abstraction layer implementation. That would give us a greater degree of assurance in the correctness of our design and implementation. Furthermore, unlike the seL4 implementation, the H interface could be reused as the basis for many different operating system implementations, thus allowing us to reap the benefits of the verification effort multiple times over. Because the potentially unsafe operations of the client operating systems would be isolated in H, there would be less need to apply formal verification techniques to the client code.

Verve is a type- and memory-safe operating system that is verified using a very different approach than seL4, making use of automated verification tools rather than interactive theorem proving [100]. The architecture of Verve is very similar to the architecture of H-based operating systems. The services of a traditional operating system kernel are split into two layers: a critical low-level layer that provides essential abstractions of the hardware (called the Nucleus) and a higher-level kernel written in a type- and memory-safe language (typed assembly language produced via C#). The Nucleus of Verve is a direct analog to H, although the services provided by the two interfaces are very different. The Nucleus is lower-level in some respects, allowing clients to install their own interrupt handlers directly into the IDT, for example, but higher-level in others, providing drivers for the keyboard and the screen. The most significant difference between the two interfaces is the treatment of memory. Verve is designed around the idea of software processes where isolation is enforced using the type system. Supporting virtual memory management operations introduces new challenges for maintaining language-based safety properties. Our work with H demonstrates safety, albeit informally, even in the presence of client-controlled hardware processes. Exciting areas of future work would include writing H in the BoogiePL language, and experimenting with encoding memory-safety for hardware processes using Hoare logic so that the H implementation could be automatically verified using the same tools as the Verve Nucleus.

9.5 SAFE OPERATING SYSTEMS

The SPIN operating system is an extensible system that supports the flexible construction of safe operating system components using the language-based abstraction barriers of their implementation language, Modula-3 [6]. The designers of SPIN wished to avoid implementing application-specific services at user-level—such as custom page fault handlers—because of the high cost of context switching between kernel and user mode. Instead, such application-specific services are implemented as kernel extensions that run in the same address space as the kernel. The SPIN design supports a capability-based interface to the core kernel services that a kernel extension might access to ensure protection of the SPIN data structures. Protection domains within the kernel address space are simply distinct name-spaces that are protected statically by the Modula-3 compiler. Though the ultimate system architecture of SPIN is fairly different from systems that we might construct using our abstraction layer, their use of capabilities is similar to the way that datatypes are used as tokens in the H interface API and their use of protected name-spaces is similar to the way that we protect the internals of the H interface using abstraction barriers in Haskell. An advantage that our approach has over SPIN is the use of a pure functional language (Modula-3 is impure and imperative). Operating systems that are written on top of H can make use of the effect tracking facilities of Haskell to reduce the occurrence of bugs and pure functional languages are more amenable to formal reasoning than a language like Modula-3. Furthermore, we take steps to formally characterize the safety guarantees that our abstraction layer supports (see Chapter 5, while in SPIN they are left implicit.

The KeyKOS, EROS, and Coyotos systems reflect a long history of work in the realm of secure operating systems. KeyKOS was a successful production kernel first developed in the 1970s, with much of the attention in the design going to robustness, reliability and security [41, 10]. KeyKOS is an operating system that uses a capability model to enforce security. The system is divided into a set of fundamental objects that are the targets of all operations. The semantics of these operations describe the protection model for the system. EROS is an implementation of the KeyKOS design focused on the performance and correctness of the capability mechanism [87, 88]. The EROS developers verified that the capability mechanism provides confinement, showing essentially that the capabilities protect the integrity of system objects from unauthorized modification. The techniques employed to formalize confinement are similar to those that we employed to describe memory-safety for H. Coyotos is a follow-on project to EROS that aims to more fully realize the high-assurance potential of the system. The confinement proof for EROS was done for a high-level model, not the real system. In Coyotos, the developers hope to prove the essential security properties for the actual implementation. A key part of this endeavor is the development of a new language that will be amenable to verification—the developers will use the BitC programming language for the Coyotos implementation and make use of BitC's formal semantics in any security proofs [85]. The security goals of all these systems are very similar to the goals of the H interface. It is interesting to note that, even though our implementations rely on very different techniques to achieve security—capabilities versus a language-based abstraction barrier—the formalization techniques and essential security properties are very similar.

The Singularity project takes the idea of software-based isolation from SPIN even further. In Singularity, all user processes, drivers, and other system components run in the highest privilege address space with the kernel [22, 48]. The type-safety of the implementation language, Sing# (a type-safe, garbage collected extension of C#), provides software-based isolation between these components, despite the shared address space. A novel feature of the Singularity operating system is an efficient message-passing mechanism that supports communication between components in the system. The communication channels in Singularity are strongly typed. Resource ownership is tracked using a linear type system to ensure isolation by guaranteeing that blocks of memory are never owned by more than one process. Statically verified channel contracts provide additional assurance that components interact correctly. The result is a system with strong isolation guarantees that use a combination of modern language techniques, like memory- and type-safety, and verification techniques.

The Singularity project relies on some of the same essential features of modern programming languages as we do, in particular memory-safety and a strong, static type system. In some ways, the type system of Sing# is more expressive than the type system of Haskell. Linear types would enable us to avoid much of the dynamic checking that needs to be done in the implementation of the abstraction layer. The ability to express, and statically verify, resource interactions (like the channel contracts in Singularity) could also be useful for establishing memory-safety of the H interface. However, the Sing# type system is less expressive in other ways, for example, the language does not support Haskell's ability to track effects. Despite the similar mechanisms for achieving safety, the goal of our work is different than that of Singularity. We aim to provide a library for writing operating systems in Haskell, rather than to construct a single system with a specific property, like isolation. Another difference is that we provide the ability to run code in any language, which Singularity does not support [22].

9.6 VIRTUALIZATION

Virtual machine monitors attempt to multiplex hardware invisibly across multiple operating systems. This goal differs from that of our abstraction layer, where we want to expose the hardware to a single OS. Despite the different goals, our abstraction layer does share some common features with virtual machine monitor designs, in particular with paravirtualization systems like Xen [5]. A paravirtualized system does not provide binary compatibility for operating systems, rather, each guest operating system must be ported to run on the virtual machine monitor. The VMM exports a low-level, hardware-like interface, so the modifications to the guest are analogous to porting to a new architecture or base library. Though an operating system designer that wishes to use our abstraction layer must also reimplement their design in a functional programming language, the modifications to the architecture-level primitives will be similar to a port to Xen.

Much like the H interface, Xen provides an interface of hypervisor calls that guest VMs use to request privileged operations. Memory management and CPU management are particularly important aspects of the Xen design. When managing memory, the guest cannot modify page tables directly. The primary focus is on exporting a safe set of API calls that cannot be invoked by a guest OS in a way that corrupts the internal structures of Xen or disrupts the execution of another guest. Unlike our work, the designers of Xen do not formally characterize the safety guarantees of their API. For limited resources, like memory, Xen uses a static partitioning among guests. Though the H interface only supports a single guest operating system, the facilities for dynamic memory exchange between the abstraction layer and the guest might be useful in a system like Xen for enabling dynamic system configurations without sacrificing the strong isolation properties that are currently present between Xen VMs.

Chapter 10

CONCLUSIONS

In this dissertation, we have demonstrated that it is possible to bridge the gap between the requirements of operating system implementations and the features of purely functional languages. We accomplished this goal by isolating the potentially unsafe operations that are required by operating systems in a memory-safe abstraction layer called the H interface. The H interface design is integrated with the purely functional language Haskell to support the development of memory-safe Haskell operating systems that do not make direct use of the foreign function interface.

An essential property of the H interface is memory-safety. The meaning of the term memory-safety is a bit murky in the operating systems domain, but we have identified two critical memory-safety issues that are relevant in the context of the H interface: no part of the system, including the H operations, should be permitted to affect the Haskell run-time system and no user program or client of the H interface should have direct access to the memory management hardware. We formalized these properties using the Rushby noninterference formalism [82] and connected the abstract formalism to our implementation with a relational specification of the memory management functions in H.

We demonstrated the expressiveness of the H interface by implementing the L4 microkernel API using the abstraction layer primitives. No special workarounds or uses of the foreign function interface were necessary. L4 supports user-level memory management, so the set of virtual memory management primitives is particularly rich. The experience of developing L4 using H was a positive one. Implementing

L4 in a strongly typed, memory-safe environment made bugs easy to detect when they occurred (relative to our experience implementing part of the H library in C) and helped us to catch many errors at compile time.

Performance is another important characteristic of H. Our goal was never to produce the best performing L4 implementation ever, but our approach to safety is not useful if the techniques are so inherently poorly performing that the resulting systems are unusable. We evaluated the performance of our approach using an IPC benchmark for L4. The performance of our L4 implementation is significantly worse than a comparable C kernel, so there is room for improvement. We have demonstrated techniques for optimizing Haskell programs in general and H-based systems in particular. Though we only applied these optimizations to a narrow area of our program, the changes radically improved the performance of IPC. These experiments illustrate the potential to optimize performance throughout the L4 and H implementations.

The major contributions of this dissertation are:

- The design and implementation of a memory-safe abstraction layer for implementing operating systems in a purely functional language. The abstraction layer is sufficiently expressive to support the implementation of real operating systems in Haskell. Our implementation of the L4 microkernel provides evidence of the abstraction layer's expressivity.

- A formalism for describing memory-safety by instantiating an abstract noninterference framework. We identify the properties that are necessary and meaningful for a Haskell library that controls the memory management hardware of the underlying machine and demonstrate a unique use of noninterference.

- A performance analysis of a bare metal Haskell program. The bare metal execution environment for Haskell is not well suited to performance measurement. We introduced facilities for analyzing the performance of our Haskell kernel and outlined an optimization path for any H-based system. We demonstrated the utility of the proposed optimizations by employing them to reduce the cost of transferring 60 words in an L4 IPC message in our kernel, producing a final version that was more than 6 times faster than the original. Our kernel is 60 times slower than the Pistachio implementation in C compared to 368 times slower before optimization.

FUTURE WORK

Our work opens up many avenues for future exploration. Some of these topics are direct extensions of the work presented here, while others represent substantially new directions for our work.

Haskell Device Drivers and Other Systems Applications

The H interface demonstrates a technique for encapsulating the potentially unsafe operations necessary for operating systems in a memory-safe Haskell library. Here, memory-safety is specifically linked to the operating systems domain, as are the specific primitives defined in the abstraction layer. There are many other systems programming applications, such as device drivers, that may require access to a different set of hardware facilities than are supported by H. The implementer of a Haskell device driver would encounter the same issues that motivated H when working with the foreign function interface directly. However, we can apply the same techniques for abstraction, interface design, and safety analysis in these application areas to encapsulate the necessary behaviors for all aspects of systems programming.

Performance Optimization

In this work, we were able to improve upon our initial performance results significantly by applying targeted optimizations to the inner loop that performs data transfer during an IPC message. Even with these improvements, the performance of our Haskell kernel is significantly worse than standard C implementations. Fortunately, there are many optimization opportunities that remain. Chapter 8 described three essential areas to focus on when optimizing an H-based system: algorithmic design, better abstractions in the H interface, and improved code generation through compiler annotations. There are also opportunities for deeper modifications (and possibly significant performance gains) by examining the efficiency of the data structures used by H, both internally and as they are exposed to the client.

Within the IPC implementation alone, the scheduling, transfer message, and thread restart phases take many more cycles than one would expect. These are obvious areas to direct our optimization energy next. By applying the techniques outlined in this dissertation to these new aspects of the algorithm, we hope to further reduce the overhead of IPC.

Verification

Verification is an obvious area to focus on in our future work. A formal proof of the unwinding conditions for our Rushby instantiation is our first priority. Proving the unwinding conditions is sufficient to demonstrate memory-safety for our specification and would go a long way towards validating our approach. Completing the proof of the unwinding conditions will give us the opportunity to turn our attention upward toward higher level properties of Haskell operating systems, such as user-program separation, or downward, to a full verification of our implementation.

To this end, we have begun a joint project to formally verify the unwinding conditions. The Haskell specification and system model have been translated into a formal model in HOL Light [43]. Using this model, we have mechanically verified our earlier sketch that the output consistency unwinding condition reduces to weak step consistency (see Section 5.4.1). A proof of local respect is currently in progress, while the verification of weak step consistency remains as future work.

Completing the proof of the unwinding conditions for the H specification will allow us to pursue properties of Haskell operating systems that depend on memory-safety. Separation properties have been a particular focus of our past work [66, 67, 39], and are an interesting area to pursue in the context of a Haskell operating system. Our experience defining memory-safety in terms of a noninterference security policy will provide valuable insight into any future efforts in this domain.

Operating Systems Exploration

Through the H interface, we hope to enable a wide range of operating systems exploration in Haskell. We demonstrate the feasibility of such exploration through our L4 implementation, but this is just one of many possibilities. We are keen to see how H generalizes by using the interface to implement other operating systems. One obvious candidate is the House operating system that originated the idea for H in the first place [39]; the API provided by the abstraction layer presented in this dissertation is radically different from the original version of H, and porting House to the new interface would be a valuable learning experience. Virtual machine monitors are another interesting domain; H was not specifically designed to support VMMs, so such a project would potentially push the interface design in exciting new directions.

Another dimension of systems exploration is in programming language research to better facilitate safe systems programming. Many of the ideas generated by this work, and the challenges that we encountered when using Haskell for operating systems programming, have influenced the design of a new language being developed at Portland State University called Habit [90]. Habit aims to provide the level of resource control that is necessary for systems programming, including control over memory layout and direct access to memory through references [20, 19], in a strongly typed functional language. Unlike other languages in this space, Habit retains all of the features that made Haskell an attractive candidate for our work—purity, monadic effects and type classes—along with many other valuable features of safe languages like garbage collection. The addition of a powerful module system is planned. From an assurance perspective, Habit greatly improves upon Haskell because it will provide a verified run-time system [73], whereas Haskell's 50,000 line C run-time is always the elephant in the room when it comes to assurance arguments. There are opportunities for improvements in the performance space as well. Besides the fact that Habit allows more direct control over resources, which should be an automatic performance win, one of the language's major departures from Haskell is that Habit is strict. Laziness caused unpredictable and severe performance problems in our IPC implementation, as mentioned in Section 8.3.3; there is hope that a strict language will simplify the optimization process and even avoid certain performance problems entirely.

Extending the connection between Habit and the H interface is also a promising topic for future work. Many of the features implemented by H are unnecessary in Habit—the language supports safe facilities for systems programming already, including many of the features necessary for writing an operating system. However, some aspects of H must be primitive to the language and are not yet part of the Habit design, such as the ability to control the memory management hardware. Work is underway on the Habit project to support the addition of new primitives in a modular way that incorporates the verification contract between the primitive and the rest of the system explicitly. This provides a perfect opportunity to integrate the essential memory management facilities of H fully into Habit. Moving from an H-based platform in Haskell to a fully integrated operating systems programming environment in Habit will give us all of the safety and engineering benefits of Haskell, but with better performance, greater assurance, and more control over resources.

A further benefit of connecting the H primitives to Habit is that the dynamic checks necessary for safety could be generated by the compiler when an H primitive is used, rather than being an integral part of the H primitive itself. In this way, the compiler could insert safety checks in precisely those places where they are necessary. In the implementation of H we do not have any information about the context in which a primitive is called, so we must always assume the call is unsafe and perform run-time checks for each one. We expect that there are many contexts that are statically known to be safe. Cyclone illustrates the feasibility of this idea for memory-safety checks in a C-style language and we expect the techniques would apply equally well to our notion of memory-safety. Though there are many other bottlenecks in H, reducing the number of run-time checks would certainly improve performance.

REFERENCES

[1] Toyota recall information website. http://www.toyota.com/recall/abs.html.

[2] Michael J. Accetta, Robert V. Baron, William J. Bolosky, David B. Golub, Richard F. Rashid, Avadis Tevanian, and Michael Young. Mach: A new kernel foundation for unix development. In *USENIX Summer*, pages 93–113, 1986.

[3] Video Electronics Standards Association. *VESA BIOS EXTENSION (VBE) - Core Functions Standard, Version 3*, September 1998. `http://www.vesa.org`.

[4] Godmar Back and Wilson C. Hsieh. The KaffeOS Java runtime system. *ACM Transactions on Programming Language Systems*, 27(4):583–630, 2005.

[5] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Timothy L. Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP 2003)*, pages 164–177, October 2003.

[6] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. Fiuczynski, D. Becker, S. Eggers, and C. Chambers. Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the 15th Symposium on Operating Systems Principles*, pages 267–284, Copper Mountain, Colorado, 1995.

[7] William R. Bevier and William D. Young. A state-based approach to noninterference. In *Proceedings of the Computer Security Foundations Workshop*, pages 11–21, 1994.

[8] Edoardo S. Biagioni, Robert Harper, and Peter Lee. A Network Protocol Stack in Standard ML. *Journal of Higher-Order and Symbolic Computation*, 14(4):309–356, 2001.

[9] BitC language website. `http://www.bitc-lang.org`.

[10] Allen C. Bomberger, William S. Frantz, Ann C. Hardy, Norman Hardy, Charles R. Landau, and Jonathan S. Shapiro. The KeyKOS nanokernel architecture. In *Proceedings of the USENIX Workshop on Microkernels and Other Kernel Architectures*, pages 95–112, 1992.

[11] Per Brinch Hansen. The nucleus of a multiprogramming system. *Communications of the ACM*, 13(4):238–241, 1970.

[12] Manuel M. T. Chakravarty, Sigbjorn Finne, Gergus Henderson, Marcin Kowalczyk, Daan Leijen, Simon Marlow, Erik Meijer, Sven Panne, Simon Peyton Jones, Alastair Reid, Malcolm Wallace, and Michael Weber. *Haskell 98 Foreign Function Interface (1.0)*, 2003. `http://www.cse.unsw.edu.au/~chak/haskell/ffi`.

[13] TIS Committee. *Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification, Version 1.2*, May 1995.

[14] Coverity. Coverity Scan open source report 2009. Online white paper, `http://scan.coverity.com/report/Coverity_White_Paper-Scan_Open_Source_Report_2009.pdf`, 2009.

[15] Coverity. Scan Project. `http://scan.coverity.com`, 2009.

[16] John Criswell, Andrew Lenharth, Dinakar Dhurjati, and Vikram S. Adve. Secure virtual architecture: a safe execution environment for commodity operating systems. In Andrew Herbert and Kenneth P. Birman, editors,

Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP 2005), pages 351–366. ACM, October 2005.

[17] John Cupitt. A brief walk through KAOS. Technical report, Computing Laboratory, University of Kent at Canterbury, February 1989.

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

[19] Iavor S. Diatchki and Mark P. Jones. Strongly typed memory areas. In *Proceedings of ACM SIGPLAN 2006 Haskell Workshop*, pages 72–83, Portland, Oregon, September 2006.

[20] Iavor S. Diatchki, Mark P. Jones, and Rebekah Leslie. High-level views on low-level representations. In *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming*, pages 168–179, Tallinn, Estonia, September 2005.

[21] Information Assurance Directorate. U.S. government protection profile for separation kernels in environments requiring high robustness. `http://niap-ccevs.org/cc-scheme/pp/pp.cfm/id/pp_skpp_hr_v1.03`, June 2007. Version 1.03.

[22] Manuel Fähndrich, Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen C. Hunt, James R. Larus, and Steven Levi. Language support for fast and reliable message-based communication in Singularity OS. In Yolande Berbers and Willy Zwaenepoel, editors, *Proceedings of the First EuroSys Conference*, pages 177–190. ACM, April 2006.

[23] Norman Feske and Christian Helmuth. A Nitpicker's guide to a minimal-complexity secure GUI. In *Proceedings of the 21st Annual Computer Security Applications Conference*, pages 85–94, 2005.

[24] Matthew Flatt, Robert Bruce Findler, Shriram Krishnamurthi, and Matthias Felleisen. Programming languages as operating systems (or revenge of the son of the Lisp machine). In *Proceedings of the 4th ACM International Conference on Functional Programming (ICFP 1999)*, pages 138–147, 1999.

[25] Torsten Frenzel. Design and implementation of the L4.sec microkernel for shared-memory multiprocessors. Diploma Thesis, Dresden University of Technology, August 2006.

[26] Guangrui Fu. Design and Implementation of an Operating System in Standard ML. Master's thesis, University of Hawaii at Manoa, August 1999.

[27] Michael Furr and Jeffrey S. Foster. Checking type safety of foreign function calls. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2005)*, pages 62–72. ACM, 2005.

[28] Galois website. `http://www.galois.com`.

[29] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: a virtual machine-based platform for trusted computing. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP 2003)*, pages 193–206, October 2003.

[30] Genode website. `http://www.genode.org`.

[31] GHC website. `www.haskell.org/ghc`.

[32] GHC libraries. `http://www.haskell.org/ghc/docs/latest/html/libraries`.

[33] J. Goguen and J. Meseguer. Unwinding and inference control. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 75–86, 1984.

[34] Kenneth Graunke. Extensible scheduling in a Haskell-based operating system. Master's thesis, Portland State University, 2010.

[35] Dan Grossman. Quantified types in an imperative language. *ACM Transactions on Programming Languages and Systems*, 28(3):429–475, 2006.

[36] Dan Grossman, Michael Hicks, Trevor Jim, and Greg Morrisett. Cyclone: a type-safe dialect of C. *C/C++ Users Journal*, 23(1), January 2005.

[37] GNU GRUB website. `http://www.gnu.org/software/grub`.

[38] Andreas Haeberlen and Kevin Elphinstone. User-level management of kernel memory. In *Proceedings of the Eighth Asia-Pacific Computer Systems Architecture Conference (ACSAC'03)*, Aizu-Wakamatsu City, Japan, September 2003.

[39] Thomas Hallgren, Mark P. Jones, Rebekah Leslie, and Andrew Tolmach. A principled approach to operating system construction in Haskell. In Olivier Danvy and Benjamin C. Pierce, editors, *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming (ICFP 2005)*, pages 116–128. ACM, September 2005.

[40] HaLVM website. `http://halvm.org/wiki`.

[41] Norm Hardy. The KeyKOS architecture. *Operating Systems Review*, September 1985.

[42] John Harrison. Floating-point verification. In John Fitzgerald, Ian J. Hayes, and Andrzej Tarlecki, editors, *Proceedings of the Formal Methods, International Symposium of Formal Methods Europe (FM 2005)*, volume 3582 of *Lecture Notes in Computer Science*, pages 529–532. Springer-Verlag, 2005.

[43] John Harrison. HOL Light: An overview. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics, TPHOLs 2009*, volume 5674 of *Lecture Notes in Computer Science*, pages 60–66, Munich, Germany, 2009. Springer-Verlag.

[44] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Sebastian Schönberg, and Jean Wolter. The performance of μkernel-based systems. In *Proceedings of the Sixteenth ACM Symposium on Operating System Principles (SOSP 1997)*, pages 66–77, 1997.

[45] Chris Hawblitzel and Thorsten von Eicken. Luna: a flexible Java protection system. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, OSDI '02, pages 391–401, 2002.

[46] Paul Hudak. *The Haskell School of Expression: Learning Functional Programming through Multimedia.* Cambridge University Press, June 2000.

[47] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A history of Haskell: Being lazy with class. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, HOPL III, pages 12–1–12–55, 2007.

[48] Galen C. Hunt and James R. Larus. Singularity: Rethinking the software stack. *SIGOPS Operating Systems Review*, 41(2):37–49, 2007.

[49] Graham Hutton. *Programming in Haskell*. Cambridge University Press, January 2007.

[50] Intel Corporation. Statistical analysis of floating point flaw: Intel white paper. `http://www.intel.com/support/processors/pentium/fdiv`, 2004.

[51] Intel Corporation. *IA-32 Intel Architecture Software Developer's Manual (Volume 3a)*, January 2006. `http://www.intel.com/products/processor/manuals/index.htm`, date viewed: 3 March 2010.

[52] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual (Volume 1: Basic Architecture)*, January 2011.

[53] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual (Volume 2B: Instruction Set Reference, N-Z)*, January 2011.

[54] Mark P. Jones. A theory of qualified types. In Bernd Krieg-Brückner, editor, *Proceedings of the 4th European Symposium on Programming (ESOP '92)*, volume 582 of *Lecture Notes in Computer Science*, pages 287–306. Springer, February 1992.

[55] Mark P. Jones. A system of constructor classes: Overloading and implicit higher-order polymorphism. *Journal of Functional Programming*, pages 52–61, 1995.

[56] Simon Peyton Jones, Mark Jones, and Erik Meijer. Type classes: An exploration of the design space. In *Proceedings of the Haskell Workshop*, 1997.

[57] Universität Karlsruhe. Universität Karlsruhe L4 website. `http://www.l4ka.org`.

[58] Kent Karlsson. Nebula: A functional operating system. Technical report, Programing Methodology Group, University of Göteborg and Chalmers University of Technology, 1981.

[59] Bernhard Kauer. L4.sec implementation - kernel memory management. Diploma Thesis, Dresden University of Technology, May 2005.

[60] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In Jeanna Neefe Matthewsand Thomas Anderson, editor, *Proceedings of the 22nd ACM SIGOPS Symposium on Operating Systems Principles (SOSP 2009)*, pages 207–220, New York, NY, USA, October 2009. ACM.

[61] Greg Kroah-Hartman. *Linux Kernel in a Nutshell.* O'Reilly & Associates, Inc., Cambridge, MA, USA, 2007. `http://www.kroah.com/lkn`.

[62] L4ka Team. *L4 eXperimental Kernel Reference Manual*, January 2005.

[63] L4Ka::Pistachio website. `http://os.ibds.kit.edu/l4ka/projects/pistachio`.

[64] Butler Lampson, Martín Abadi, Michael Burrows, and Edward Wobber. Authentication in distributed systems: theory and practice. *ACM Trans. Comput. Syst.*, 10(4):265–310, 1992.

[65] John Launchbury and Simon L Peyton Jones. State in Haskell. *Lisp and Symbolic Computation*, 8:293–341, December 1995.

[66] Rebekah Leslie. Dynamic intransitive noninterference. In *Proceedings of the First IEEE International Symposium on Secure Software Engineering*, March 2006.

[67] Rebekah Leslie, Levent Erkök, and Flemming Andersen. Formalizing information flow in a Haskell hypervisor. In *Proceedings of the Workshop on Microkernels for Embedded Systems*, Sydney, Australia, January 2007.

[68] R. Levin, E. Cohen, W. Corwin, F. Pollack, and W. Wulf. Policy/mechanism separation in Hydra. In *Proceedings of the Fifth ACM Symposium on Operating Systems Principles*, SOSP '75, pages 132–140, 1975.

[69] Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *Proceedings of the 22nd ACM Symposium on Principles of Programming Languages (POPL 1995)*, pages 333–343, 1995.

[70] Jochen Liedtke. Improving IPC by kernel design. In *Proceedings of the 14th ACM Symposium on Operating System Principles (SOSP 1993)*, pages 175–188, 1993.

[71] Jochen Liedtke. On micro-kernel construction. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles (SOSP 1995)*, pages 237–250, 1995.

[72] Jacques-Louis Lions. ARIANE 5, Flight 501 failure, report by the inquiry board, 1996.

[73] Andrew McCreight, Tim Chevalier, and Andrew P. Tolmach. A certified framework for compiling and executing garbage-collected languages. In Paul Hudak and Stephanie Weirich, editors, *ICFP*, pages 273–284. ACM, September 2010.

[74] Greg Nelson, editor. *Systems Programming with Modula-3*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1991.

[75] NICTA. NICTA L4 website. `http://ertos.nicta.com.au/research/l4`.

[76] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL—A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[77] Bryan O'Sullivan, John Goerzen, and Don Stewart. *Real World Haskell*. O'Reilly Media, Inc., 1st edition, 2008.

[78] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries, The Revised Report*. Cambridge University Press, 2003.

[79] Simon L. Peyton Jones and Philip Wadler. Imperative functional programming. In *Proceedings of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 71–84, 1993.

[80] Sergio Ruocco. User-level fine-grained adaptive real-time scheduling via temporal reflection. In *Proceedings of the 27th IEEE Real-Time Systems Symposium (RTSS 2006), 5-8 December 2006, Rio de Janeiro, Brazil*, pages 246–256, December 2006.

[81] John Rushby. The design and verification of secure systems. In *Proceedings of the Eighth ACM Symposium on Operating System Principles (SOSP 1981)*, pages 12–21, December 1981. (ACM *Operating Systems Review*, Vol. 15, No. 5).

[82] John Rushby. Noninterference, transitivity, and channel-control security policies. Technical report, December 1992.

[83] SANS Institute and MITRE. 2009 CWE/SANS top 25 most dangers programming errors. `http://cwe.mitre.org/top25`, 2009.

[84] seL4 website. `http://www.ertos.nicta.com.au/research/sel4`.

[85] Jonathan Shapiro, Michael Scott Doerrie, Eric Northup, Swaroop Sridhar, and Mark Miller. Towards a verified, general-purpose operating system kernel. In *Proceedings of the NICTA Workshop on Operating System Verification*, pages 1–19, October 2004.

[86] Jonathan Shapiro, Swaroop Sridhar, and Scott Doerrie. *The BitC Language Specifiction*, 2008.

[87] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. EROS: A fast capability system. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, SOSP 1999, pages 170–185, 1999.

[88] Jonathan S. Shapiro and Samuel Weber. Verifying the EROS confinement mechanism. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2000.

[89] W. Stoye. Message-based functional operating systems. *Science of Computer Programming*, 6:291–311, May 1986.

[90] The High Assurance Systems Programming Team. *The Habit Programming Language: The Revised Preliminary Report*, November 2010.

[91] The GHC Team. *The Glorious Glasgow Haskell Compilation System User's Guide, Version 7.0.2*, 2011. `http://www.haskell.org/ghc/docs/7.0-latest/users_guide.pdf`.

[92] VirtualBox website. `http://www.virtualbox.org`.

[93] David von Oheimb. Information flow control revisited: Noninfluence = Noninterference + Nonleakage. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS 2004)*, volume 3193 of *LNCS*, pages 225–243. Springer, 2004.

[94] Philip Wadler. Comprehending monads. In *Mathematical Structures in Computer Science*, pages 61–78, 1992.

[95] Philip Wadler. The essence of functional programming. In Ravi Sethi, editor, *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 1992)*, pages 1–14. ACM, 1992.

[96] Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad-hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 1989)*, pages 60–76, 1989.

[97] Malcolm Wallace and Colin Runciman. Extending a functional programming system for embedded applications. *Software Practice and Experiences*, 25(1):73–96, 1995.

[98] Malcolm Wallace and Colin Runciman. Lambdas in the liftshaft—functional programming and an embedded architecture. In *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture (FPCA 1995)*, pages 249–258. ACM, 1995.

[99] Adam Wick and Matthew Flatt. Memory accounting without partitions. In *Proceedings of the 4th International Symposium on Memory Management (ISMM 2004)*, pages 120–130. ACM, 2004.

[100] Jean Yang and Chris Hawblitzel. Safe to the last instruction: automated verification of a type-safe operating system. In Benjamin G. Zorn and Alexander Aiken, editors, *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2010)*, pages 99–110. ACM, June 2010.