Type-Safe Operating System Abstractions

Dartmouth Computer Science Technical Report TR2004-526

A Thesis

Submitted to the Faculty

in partial fulfillment of the requirements for the

degree of

Doctor of Philosophy

in

Computer Science

by

Lea Wittie

DARTMOUTH COLLEGE

Hanover, New Hampshire

June 30, 2004

Examining Committee

_____

(chair) Dr. Chris Hawblitzel

_____

Dr. Doug McIlroy

_____

Dr. Stefan Monnier

_____

Dr. Sean Smith

_____

Carol Folt

Dean of Graduate Studies

Abstract

Operating systems and low-level applications are usually written in languages like C and assembly, which provide access to low-level abstractions. These languages have unsafe type systems that allow many bugs to slip by programmers. For example, in 1988, the Internet Worm exploited several insecure points in Unix including the finger command. A call to finger with an unexpected argument caused a buffer overflow, leading to the shutdown of most Internet traffic. A finger application written in a type-safe language would have prevented its exploit and limited the points the Internet Worm could attack. Such vulnerabilities are unacceptable in security-critical applications such as the secure coprocessors of the Marianas network [11, 21], secStore key storage from Plan 9 [6], and self-securing storage [25].

This research focuses on safe language techniques for building OS components that cannot cause memory or IO errors. For example, an Ethernet device driver communicates with its device through IO operations. The device depends on FIFO queues to send and receive packets. A mistake in an IO operation can overflow or underflow the FIFO queues, cause memory errors, or cause configuration inconsistencies on the device. Data structures such as FIFO queues can be written safely in safe languages such as Java and ML but these languages do not allow the access to the low-level resources that an OS programmer needs. Therefore, safe OS components require a language that combines the safety of Java with the low-level control of C.

My research formalizes the concurrency, locks, and system state needed by the safety-critical areas of a device driver. These formal concepts are built on top of an abstract syntax and rules that guarantees basic memory safety using linear and singleton types to implement safe memory load and store operations. I proved that the improved abstract machine retains the property of soundness, which means that all well-typed programs will be able to execute until they reach an approved end-

state. Together, the concurrency, locks, and state provide safety for IO operations and data structures.

Using the OSKit from the University of Utah as a starting point, I developed a small operating system. I ported the 3c509 Ethernet device driver from C to Clay, a C-like type-safe language that uses a type system powerful enough to enforce invariants about low-level devices and data structures. The resulting driver works safely in a multi-threaded environment. It is guaranteed to obtain locks before using shared data. It cannot cause a FIFO queue to overflow or underflow and it will only call IO operations when invariants are satisfied.

This type-safe driver demonstrates an actual working application of the theoretical components of my research. The abstract machine is powerful enough to encode a given OS specification and enforce a provably matching implementation. These results lead towards fundamentally secure computing environments.

Acknowledgements

I would like to thank my advisor, Dr. Chris Hawblitzel, for his guidance over the past three years. This thesis would not have been possible without the countless hours and many late-night work sessions he spent helping me. In addition, I would like to thank my committee members, Dr. Doug McIlroy, Dr. Stefan Monnier, and Dr. Sean Smith for diligently reading my thesis and providing extensive comment.

I would also like to acknowledge the help of several others: Tom Cormen for his support and for guidance in the proper use of the English language; my family and friends for their support and encouragement; the departmental secretaries, Kelly Clark, Delia Mauceli, and Sammie Travis for answering questions and help with paperwork; my system administrators Wayne Cripps, Tim Tregubov, and Sandy Brash for all their help with the department network and printers; and Dr. Scot Drysdale for making the departmental TGIF possible. I would finally like to thank my many office plants for growing well over an inch a week and taking up all available free space.

Contents

1 Introduction

Despite advances in programming languages and programming techniques, today's operating systems often contain bugs. These bugs can be exploited by malicious hackers to allow undesirable actions to occur. For example, in 1988, the Internet Worm exploited Unix's finger command using a buffer overflow, leading to the shutdown of most Internet traffic. Such vulnerabilities are unacceptable in security-critical applications such as the secure coprocessors of the Marianas network [11, 21], secStore key storage from Plan 9 [6], and self-securing storage [25].

Bugs are not just present in older operating systems. The SecurityFocus Vulnerabilities Archive [24] collects reports of security vulnerabilities found in current operating systems and applications. Hundreds of bugs have been reported this year alone. SecurityFocus also lists the consequences of the bugs including buffer overflows, code injection, circumvented read/write permissions, keystroke interception, private information disclosure, and root password theft.

A study by Chou et al. [4] examined the number and location of bugs in the Linux OS. The most frequent bugs found were:

- Calling blocking functions with interrupts disabled or a lock held.
- Returning NULL pointers from routines.
- Allocating stack variables that don't fit on the fixed-size kernel stack.
- Making inconsistent assumptions about whether a pointer is NULL.
- Not checking array bounds or loop bounds from user input.
- Not releasing locks or double locking.
- Not restoring disabled interrupts.
- Using freed memory.

One reason for the presence of bugs in operating systems is that most are writ-

ten in non-type-safe languages like C and assembly. Writing applications in a safe language (Java, ML) would eliminate many vulnerabilities. Java, for example, can detect NULL pointers and inserts run-time checks for array bounds. It also implements locks in clauses which both acquire and release so the releasing step cannot be forgotten. Java's garbage collector ensures that memory is not freed until the operating system is done using it. However, safe languages such as Java and ML are not used for operating systems programming because they do not easily give programmers access to the low-level resources they need. For example, Java does not allow the casting of one data type to any other data type. Only a limited set of casts are allowed. Java also cannot perform the atomic operations provided by assembly calls or make IO calls without calling C functions. C and assembly, while not type-safe, provide easy access to such low-level abstractions.

This research focuses on type-safe language techniques for building OS components that cannot cause memory or IO errors. For example, an Ethernet device driver communicates with its device through IO operations. The device depends on FIFO queues to send and receive packets. A mistake in an IO operation can overflow or underflow the FIFO queues, cause memory errors, or cause configuration inconsistencies on the device.

My research formalizes the concurrency, locks, and system state needed by the safety-critical areas of an operating system. These formal concepts are built on top of an abstract syntax and rules that guarantee basic memory safety using linear and singleton types to implement safe memory load and store operations. Together, the concurrency, locks, and state provide safety for IO operations and data structures.

I proved that the improved abstract machine retains the property of soundness, which means that all well-typed programs will be able to execute until they reach an approved end-state. Any well-typed program respects the abstractions of its language. A type checker for a language that implements the abstract machine will

catch violations of the driver's safety abstractions.

Using the OSKit from the University of Utah as a starting point, I developed a small multi-threaded operating system. The OS runs several network applications including ping, and uses the 3c509 Ethernet device driver. I ported the driver from C to Clay, a C-like type-safe language that matches the abstract machine. Chou et al. [4] found that the device drivers, which accounted for 70% of the OS code, accounted for almost 90% of the bugs. The researchers list possible explanations for the unexpectedly high error rates of drivers. One likely explanation is that drivers are written by a wide range of programmers who know the device very well but may be unfamiliar with the kernel interface. The programmers are likely to make mistakes and, using cut and paste, propagate the mistakes over many drivers. The I/O hardware is often incorrectly documented which leads to programmers trying to cope with an incorrect specification.

The resulting driver works safely in a multi-threaded environment. It is guaranteed to obtain locks before using shared data. It cannot cause a FIFO queue to overflow or underflow and it will only call IO operations when invariants are satisfied.

2 Type Systems and Safe Languages

Not all type systems are equally powerful; a weak static type system forces the language to perform run-time safety checks. For example, type-safe languages like Java and ML perform automatic run-time checks on array bounds. Unsafe languages (C, C++) expect the programmer to include run-time checks. In a large program such as an OS, there are likely to be many run-time bounds checks that are left out or slow the program down considerably. The study by Chou et al. discussed in section 1 found many forgotten bounds checks in Linux. The inserted run-time bound checks are one reason that operating systems are not usually written in Java. If these checks could be moved to compile-time, the program would have the same safety guarantees without continual run-time bounds checks. For the purposes of this thesis, safe means that a specification (such as Java's "no bounds are violated") is enforced through compile-time and run-time checks.

In addition, none of the languages mentioned above are able to catch aliasing errors automatically. An aliasing error occurs when several copies, or aliases, of a system-state value are in existence and the system-state changes and then an outdated copy is mistakenly used. Some aliasing errors could be caught by keeping and checking reference counts but this would contribute to the already expensive run-time checks. Compile time checks for aliasing errors would provide safety without adding to the run time burden.

Advanced types such as integer arithmetic, singleton, and linear types offer a compile-time solution. This section discusses static techniques for array bounds check elimination and alias elimination.

```
T                                                  // return type
array_get                                          // function name
[int N, type T, int I; 0<=I && I<N]                // type parameters
(Array[N,T] array, Int[I] i);                      // function parameters

void array_set [int N, type T, int I; 0<=I && I<N]
(Array[N,T] array, Int[I] i, T t);
```

Figure 1: These array access functions use singleton types and arithmetic comparisons to avoid run-time bounds checks.

2.1 Integer Arithmetic and Singleton Types

Usually, array bounds checks are required at run-time because the exact index into the array and the exact size of the array aren't known until run-time. If information about the index and size was captured in the static type system, bounds checks could be performed at compile-time rather than at run-time. Doing so requires a system for statically reasoning about the run-time values of index and size.

Xi and Pfenning [31] added arguments to integer types and added arithmetic capabilities ($==, !=, <, >, +, -$) to their type system in order to make comparisons between type arguments. These additions let them write invariants such as $0 \leq index < size$ into their function types. To connect these type variables to the run-time variables they used singleton types. A singleton type is a type with one element. The singleton type $Int[3]$ is the type of integers whose value is 3.

Together, the singleton types and arithmetic comparisons allowed them to encode invariants about the run-time values of an index and array size. They demonstrated how to eliminate run-time array bounds checks using these invariants. Though their examples used ML, this section uses the syntax of Clay (a safe variant of C with polymorphism).

The array access functions in Figure 1 avoid run time bounds checks on the array access by using singleton types and arithmetic comparisons. The square brackets after the function name contain type parameters and arithmetic comparisons on

those parameters. The types of function parameters use the type parameters. The type $Array[N, T]$ is an array of length N containing values of type T. The type $Int[I]$ is a singleton integer whose value is I. The *array_get* function takes an array of length N containing values of type T and an index with value I and returns a value of type T. *array_set* takes an array of length N containing values of type T, an index with value I, and a value of type T and returns nothing. These functions and the array type are polymorphic and will work with any type of array. In this example, the comparisons state that the index must be both greater than or equal to 0 and less than the length of the array.

Type comparisons are checked at compile time. When the *array_get* function is called on an array and an index

```
Array[6,int] array = ...;
int x = array_get(array, 4);
```

where *array* has type $Array[6, int]$ and 4 has type $Int[4]$, the compiler compares the input and parameter types and gets the following facts:

$$N == 6,\ T == int,\ I == 4$$

From there, the compiler generates the constraint

$$0 <= I \ \&\& \ I < N$$

as shown in the function header. Using the above mentioned facts, this constraint is equivalent to

$$0 <= 4 \ \&\& \ 4 < 6$$

As long as these constraints are true, the function call is safe and won't violate array boundaries. Since this is checked at compile time, the code does not need a run-time

```
void swap
[int M, type T, int J, int K; 0<=J && 0<=K && J<M && K<M]
(Array[M,T] array, Int[J] j, Int[K] k)
{
    let temp = array_get(array, j);
    array_set(array, j, array_get(array, k));
    array_set(array, k, temp);
}
```

Figure 2: This array swap function uses singleton types and arithmetic comparisons to avoid run-time bounds checks.

bounds check. Calls to the *array_set* function generate similar constraints.

Figure 2 shows a more complex example of compile-time bounds checks. The *swap* function swaps two values in an array using the *array_get* and *array_set* functions defined in Figure 1. *swap* takes the indices of these values as singleton integers to relate their values to the array length. The arithmetic comparisons between the singleton integers and the array length ensure that the indices are legal array indices. Assuming that a call to *swap* follows the invariants stated by its arithmetic comparisons, the following facts are known:

$$0 <= J \;\&\&\; 0 <= K \;\&\&\; J < M \;\&\&\; K < M$$

Inside the *swap* functions, there are two calls to *array_get*. The first call uses the facts known in the swap function and produces these facts and constraints:

$$N == M,\ T == T,\ I == J,\ 0 <= J \;\&\&\; J < M$$

The second call produces the constraints:

$$N == M,\ T == T,\ I == K,\ 0 <= K \;\&\&\; K < M$$

At a glance the facts known in *swap* imply the constraints in the two calls to

```
Array[6,int] array = ...;
let a = get_index_from_user();
let b = get_index_from_user();
swap(array, a, b);
```

Figure 3: A function call whose inputs are user entered without a run-time bounds check is unsafe.

```
Array[6,int] array = ...;
let a = get_index_from_user();
int b = get_int_from_user();
if (0<=a && a<6 && 0<=b && b<6)
    swap(array, a, b);
```

Figure 4: A function call whose inputs are safely user entered. The bounds check makes the function call safe.

array_get. These constraints allow the compiler to verify that calls to *swap* that obey its input comparisons will make safe calls to *array_set* and *array_get*.

The constraints shown above are fairly simple to check. However, checking many such constraints by hand would be tedious. Xi and Pfenning [31] showed that constraints produced by type checking can be solved efficiently using an automated constraint checker such as Omega [23].

Some run-time bounds checks cannot be avoided. Statically type checking bounds on user input can be difficult since the input values won't be known until run-time. With singleton types and arithmetic comparisons, the call to *swap* in Figure 3 generates the constraint

$$\emptyset \Longrightarrow (0 \leq J \;\&\&\; 0 \leq K \;\&\&\; J < M \;\&\&\; K < M)$$

which fails to type check since "nothing" cannot imply $(0 \leq J \;\&\&\; 0 \leq K \;\&\&\; J < M \;\&\&\; K < M)$. To pass the type checker, the programmer needs to add an if-then statement as seen in Figure 4 and only call *swap* if its invariants are met. Even though this situation requires run-time bounds checking to be safe, the function invariant helps because its presence forces the programmer to include a bounds

```
LinArray[N,T] lin_array_set
[int N, type T, int I; 0<=I && I<N]
(LinArray[N,T] array, Int[I] i, T t);

LinArray[6,int] array_a = ....;
let array_b = array_a;        // array_a is now invalid
lin_array_set(array_a, 4, 0); // fails: array_a invalid
lin_array_set(array_b, 4, 0); // array_b is now invalid
```

Figure 5: An array set function for linear arrays. A linear variable is invalid between being read and being written.

check on the user input that might otherwise be forgotten. Safe languages such as Java insert many run time bounds checks during compilation instead of requiring the programmer to insert them.

2.2 Linear Types

There aren't many easy ways to use run-time checks to avoid aliasing errors. Reference counting is sometimes possible but can have a high cost in running time. Types based on the linear logic of Girard are used to prevent aliasing errors without run-time checks [29].

A linear value has only one reference to it. A linear variable is able to be accessed exactly once and cannot be duplicated or discarded. Any attempt to copy a linear variable creates a duplicate but invalidates the original. Therefore, linear variables are explicitly handed from one function to the next. Figure 5 shows a linear array. Creating *array_b* invalidates *array_a* so future attempts to set index 4 to a 0 are type errors. The second call to *lin_array_set* sets index 4 of *array_b* to 0 but doesn't catch the returned array so *array_b* also becomes invalid. A safe call to *lin_array_set* would look like:

```
let array_b = lin_array_set(array_b, 4, 0);
```

This call invalidates *array_b* so the *lin_array_set* function has the only valid copy and then catches the array when it is returned and reassigns it to *array_b*.

```
void set_irq (int n); // Post: Interrupt request level=n
void f1 ();           // Pre: level<=5, Post: level=5
void f2 ();           // Pre: level=3,  Post: level=3

set_irq(3);           // Set the irq to 3
f1();                 // Change the irq to 5
f2();                 // Could deadlock if level!=3
```

Figure 6: A set of C function calls that depend on the interrupt request level. The last function causes deadlock if it is called with the wrong interrupt level.

```
type IrqLevel[int N];
IrqLevel[N] set_irq [int N; 0<=N](Int[N] n);
IrqLevel[5] f1 [int N; 0<=N && N<=5](IrqLevel[N] irq);
IrqLevel[3] f2 (IrqLevel[3] irq);

IrqLevel[3] irq = set_irq(3);    // Set the irq to 3
irq2 = f1(irq);                  // Use the irq
f2(irq);                         // Could still deadlock
```

Figure 7: A set of Clay function calls that depend on the interrupt request level. The last function could still cause deadlock if it is called with the wrong interrupt level even though the invariants have been moved from the comments to the code.

It is a common mistake to call functions that depend on a system parameter that does not have the expected value. If there are many references to the system parameter, aliasing errors can occur. Figure 6 shows such an example in C.

Two functions that depend on the system interrupt request (IRQ) level are called. The IRQ numbers represent the current priority of the system IRQ level. A higher number means a higher priority. The first function changes the IRQ but the programmer does not remember to test for it and calls the second function which required the old, lower, IRQ value. This situation could lead to deadlock because an unexpectedly high IRQ value might prevent interrupts and traps from being handled when they should be. The $f2$ function may rely on interrupts that need an IRQ level of 3.

Figure 7 shows the same example, in Clay (not C) with the pre and post conditions included in the code as invariants using singleton types and arithmetic comparisons. A value of type $IrqLevel$ is passed to and from functions as evidence that

```
@type IrqLevel[int N];   // @ means IrqLevel is linear
IrqLevel[N] set_irq [int N, int R] (IrqLevel[R] irq, Int[N] n);
IrqLevel[5] f1 [int N; 0<= N && N<=5](IrqLevel[N] irq);
IrqLevel[3] f2 (IrqLevel[3] irq);

IrqLevel[3] irq2 = set_irq(irq, 3); // Invalidates irq.
irq3 = f1(irq2);                    // Invalidates irq2.
f2(irq2);                       // Fails since irq2 is invalid
```

Figure 8: A set of function calls that depend on the linear interrupt request level. The last function call will fail to type check because it takes an invalidated IrqLevel.

```
IrqLevel[3] irq2 = set_irq(irq, 3);
IrqLevel[5] irq3 = f1(irq2);
IrqLevel[3] irq4 = set_irq(irq3);
IrqLevel[3] irq5 = f2(irq4);         // Correct!
```

Figure 9: A set of function calls that depend on the linear interrupt request level. The function headers are shown in Figure 8. This code uses the linear correctly and succeeds.

the run-time IRQ level is equal to N whenever the function is called.

The *set_irq* function takes a singleton integer of value N and returns an $IrqLevel[N]$. The $f1$ function now takes an $IrqLevel[intN]$. The arithmetic comparisons enforce the invariant $0 \leq N \leq 5$. The function returns an $IrqLevel[5]$. The $f2$ function now takes an $IrqLevel[3]$. The type system will ensure that calls to $f2$ pass in an $IrqLevel[3]$. When the code in Figure 7 is compiled, the type of irq is $IrqLevel[3]$. This irq is passed to $f1$ which calls *set_irq* to change the system parameter to 5 and returns an $IrqLevel[5]$. At this point an error is still made because irq no longer represents the system parameter but is passed to $f2$. This error still sneaks by so stronger measures are necessary.

The type system uses singleton types and arithmetic comparisons to enforce invariants and uses linear types to avoid aliasing problems with the irq level. To prevent passing around outdated values, variables of type $IrqLevel$ will need to be linear so they become invalid when they are used. Figure 8 shows the example with

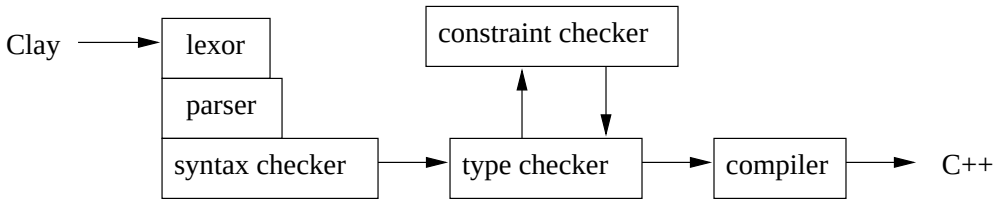

Figure 10: The Clay compiler

linear types added. Calls to set_irq take a linear $IrqLevel$ and invalidate it. They return a new $IrqLevel$ that accurately reflects the system parameter. Calls to $f1$ do the same. Calls to $f2$ return the original $IrqLevel$ to be re-used since it is unchanged. Because $irq2$ is linear, the call to $f1$ invalidates it and the error seen in the previous aliasing examples will be caught by the type checker that reports that $f2$ has been handed an invalid $IrqLevel$. Figure 9 shows a corrected version of the function calls. Making variables of type $IrqLevel$ linear prevents aliasing errors such as passing an outdated version of the irq to a function.

Singleton and linear types allow compile-time detection of aliasing errors and invariant failures. Where missing run-time checks are necessary, invariants will fail at compile-time alerting the programmer to the omission.

2.3 The Type-Safe Language Clay

Clay[1] is a type-safe variant of C++ with an advanced type system. This language was developed at Dartmouth College primarily by Chris Hawblitzel, Gary Morris, Eric Krupski, and Ed Wei. The language currently supports singleton types, linear types, polymorphism, and type inference. Clay is able to detect many errors including buffer overflows, kernel stack overflows, NULL pointer uses, freed memory uses, and aliasing errors at compile time. I have made small additions to the Clay compiler as needed. These additions include bitwise operators (<<, >>, &, |, ...) and hexadecimal numbers.

[1]C w/ LineAr tYpes, C w/ Linear ArithmetYc, CLassy AcronYm

Currently many necessary safety checks in operating systems are done at run time or not at all. Clay moves the burden of error checking from run time to compile time thereby allowing the programmer to find errors while compiling and lessening the number of errors the end user sees.

The Clay compiler consists of a lexer, parser, syntax checker, type checker, the Omega constraint checker [23], and a compiler to C. Figure 10 shows a diagram of the compilation steps. The type checker produces constraints, which are run through the Omega constraint checker. After type checking is successful, the code compiles to C. When a Clay file foo.clay is compiled ("clay foo.clay"), it produces a file named foo.cc that contains the C++ translations of foo.clay and all the extern declarations it needs.

2.3.1 Size 0 Types

The first aliasing example (Figure 6) attempted to use the interrupt request level without passing an explicit value around. In solving the aliasing problem, an explicit irq value of type *IrqLevel* had to be passed. The compiler guaranteed that the code passed the irq value safely and only called irq functions when allowed as stated in the invariants. The compiled code comes with this guarantee and does not need the irq value any more since all the checks on it were done at compile time and the checks were removed from the compiled code.

Most types take up space in memory and their values can have run-time affects. The *IrqLevel* type doesn't affect anything at run time and nothing else depends on it at run time. It doesn't need to take up any space in memory or even exist at run time.

Clay types can be grouped by the amount of space they use in memory. An *int* in C++ uses 32 bits of memory, a *short* uses 16 and a *char* uses 8. In Clay, these types belong to higher groups called kinds, which reflect their linearity and the amount

of memory they use. The type *int* has kind *type*32. This is the most common kind and is abbreviated to *type*. The type *IrqLevel* can have the kind *type*0 meaning it uses no space in memory. It is linear so the full kind would be @*type*0. By making *IrqLevel* linear and size 0, the compiler can check all the constraints on the interrupt request level and no longer need the *IrqLevel* at run-time.

Size 0 types allow invariants to be tracked throughout code without taking up memory or slowing down the code at run time. The resulting compiled code is safe (provably free of aliasing errors and provably matching the specification) and can be just as fast as the original code.

2.3.2 Type Erasure

The type parameters and comparisons used during compile time are unimportant during run-time when the compiled code has already been proven safe. After type checking, the Clay compiler erases the type parameters and comparisons before producing C++ code. When *IrqLevel* is linear and size 0, the *irq* functions from the previous example change from this:

```
@type0 IrqLevel[int N];
IrqLevel[N] set_irq [int N, int R] (IrqLevel[R] irq, Int[N] n);
IrqLevel[5] f1 [int N; 0<= N && N<=5](IrqLevel[N] irq);
IrqLevel[3] f2 (IrqLevel[3] irq);
```

to this:

```
void set_irq (unsigned long n);
void f1 ();
void f2 ();
```

The resulting code has the safety guarantees from the compiler and is in C++, a language commonly used for operating systems. This means that parts of an operating system can be written in Clay and compiled to C++ and then compiled

together with the rest of an operating system. Function calls are possible from Clay to C++ and vice versa. Creating a safe operating system is a daunting task but with Clay, important parts can be written safely and other parts can left in C or C++.

2.3.3 Clay Syntax

Clay syntax is similar to C. This section explains the differences needed to read the code examples in the rest of the thesis.

Variable declarations

Variables must be initialized when they are declared. Unlike C, Clay has some type inference and can guess the correct type from the initial value. The type can also be stated explicitly. The following are declarations:

```
int x = 3;
```

```
let x = 3;
```

The first creates a signed 32 bit integer with value 3. The second asks Clay to infer the type of x. Clay chooses $Int[3]$, a 32 bit singleton integer.

Function headers

Unlike C, Clay function headers can include type parameters and comparisons between those parameters. The format of a standard Clay function is:

```
inline return_type name [type parameters; comparisons]
  (parameters) limit;
```

As in C, the inline tag is optional. The return type can be any Clay type. Clay is capable of returning more than one value at a time using the tuple type $(\phi[\tau_1, \tau_2, \ldots])$. The ϕ in the type must be @ (linear) if any of the types in the tuple are linear. The function name must begin with a lowercase letter. The type parameters used in the function parameters must be declared in the square brackets along with any

boolean comparisons between them. The *swap* function shown in Figure 2 shows type parameters and comparisons:

```
void swap
  [int M, type T, int J, int K; 0<=J && 0<=K && J<M && K<M]
  (Array[M,T] array, Int[J] j, Int[K] k)
{ ... }
```

Types and type parameters (other than type *int*) must begin with an uppercase letter. The type variables in the *array_get*, *array_set*, and *swap* functions don't need to have different names. As with parameter names, scope allows different functions to use the same type parameter names. The names in *swap* are different in these examples for purposes of simplicity.

The limit is optional and only appears when the function manipulates only size 0 types. If the whole function has no actual run-time effect, it and any calls to it can be erased. The keyword for a limit is $limited[N]$ where N is a positive integer. A limited function can only call functions with an equal or lower limit value. An unlimited function acts as if its limit was infinity. This allows the compiler to erase functions with no direct run-time effect.

Functions used in Clay but written in C++ have two syntax choices. The first:

```
native return_type function_name [type parameters; comparisons]
  (parameters) limit;
```

uses the *native* tag and is just the function header. (native implies that the function is written in a *native* non type-safe language such as C++.) If the function can be erased, it is $limited[N]$ and there is no corresponding C++ function. For example, a function which throws away the $IrqLevel$ state-type value when it hits 0

```
native void pitch_irq(IrqLevel[0] irq) limited[0];
```

will be completely erased by the compiler.

The second syntax option for native functions is

```
return_type function_name [type parameters; comparisons]
  (parameters) limit = native { ...}
```

The C++ function header and body can be declared inside the native brackets.

Type parameters

Type parameters are declared with a kind. They can use the kinds for types as well as *int* and some abbreviations that include some comparison information about the *int*. A integer type parameter could be used as follows:

```
void foo [int N] (Int[N] n);
```

Each type parameter is declared with its kind. The comparisons between type parameters are phrased as a boolean expression where the possible boolean operators are comparative ($<$, $>$, $\leq$, $\geq$, $==$), additive ($+$,$-$), and logical (&&,||). The following shows several boolean operators being used in the arithmetic comparison:

```
void foo [int N, int M; (N+3>M || M==10) && N>=0](Int[N] n, Int[M] m);
```

Clay uses some abbreviations for frequently used combinations of type parameters and comparisons. For example, kind s32 means a signed 32-bit integer and u32 indicates an unsigned 32-bit integer. The compiler can handle differing sizes of kind abbreviations. If a function takes a singleton integer whose value can be 0 to 3, the kind of its type parameter can be declared as u2 to indicate it is unsigned and the data occupies two bits:

```
void foo [u2 N] (Int[N] n);
```

This is the same as:

```
void foo [int N; 0<=N && N<4] (Int[N] n);
```

Type declarations

Types are declared in two ways:

```
type Int[int N] = native
```

```
typedef TwiceInt[int N] = Int[N+N]
```

The first type is the singleton integer. It is declared to have kind *type* meaning it is non-linear and takes up 32 bits of memory. The keyword *native* means the type is defined in C by the Clay compiler. Because $Int[N]$ has kind *type*, it will become an *unsigned int* when compiled so the programmer does not need to create a type in C. The second type is built on top of an existing type and Clay can infer its kind from the kind of $Int[N]$. The kinds and some base types are built into clay. Using a type declaration, a programmer can create new types.

Existential types

Clay uses existential type parameters when the kind of the type parameter is known but its value is not. A function that gets external data (from a device or a user) may not know the exact value it returns. Such a function might look like:

```
exists [u8 B] Int[B] query_user_for_a_char [u16 A] (Int[A] a);
```

This function takes a singleton integer in the range of 0 to 65535 (16 bits) and returns some singleton integer in the range of 0 to 255 (8 bits). The type $exists\ [u8\ B]\ Int[B]$ allows the exact value to be determined at run time.

Packing and unpacking types

When the $query\_user\_for\_a\_char$ function is called, the returned value is existential and will need to be unpacked [16] before it can be used as a singleton integer. This is accomplished with square brackets after the *let* in its assignment statement:

```
let[] c = query_user_for_a_char(200);
```

Type *int* is defined as an abbreviation for $exists\ [s32\ I]\ Int[I]$ so an *int* can be converted to a singleton integer by unpacking:

```
int x = 3;      // x : exists [s32 I] Int[I]
let[] y = x;    // y : Int[I] where I's value isn't known
```

By unpacking, the type system can use the value as a singleton integer but it is not able to recognize that the value is 3. Types can also be packed as other types to add an existential:

```
let x = 3;               // x : Int[3]
let y = pack [int] (x); // y : exists [s32 I] Int[I]
```

Again, the specific value 3 is lost when packing occurs.

3 A Formal Abstract Machine with Concurrency and Locks

The formalization of locks and concurrency begins with part of an abstract language presented by Dartmouth's language theory group in a technical report [12]. It provides the syntax and rules for an abstract machine which includes linear memory. Operating systems are multi-threaded so I added concurrency to the abstract machine. I also added a formal lock system to allow safe threads to share data safely. We had already proven the basic abstract machine to be sound in [12] so I re-proved soundness on my abstract machine with concurrency and locks. Soundness means that programs or expressions written in the abstract syntax can always evaluate until they reach an approved end-state and types are proved during the evaluation. This means that using my abstract machine, a multi-threaded expression that uses locks cannot get stuck in a non-end state and cannot suddenly convert the locks into some other type. Such an expression is forced to use the lock rules to acquire and release the lock and may not access a locked resource without first acquiring it. My abstract machine also prevents the release of a lock without holding a matching resource. This section presents the basic abstract machine and its proofs of soundness. It then shows my addition of concurrency and locks and explains the new soundness proofs.

3.1 The Basic Abstract Syntax

The syntax of the abstract machine is composed of linearity, environments, kinds, types, expressions, and values.

linearity

$$\phi = \text{non} \mid \text{lin}$$

Linearity is represented by ϕ, which can be non (non-linear) or lin (linear).

arithmetic

$$i = \cdots \mid -2 \mid -1 \mid 0 \mid 1 \mid 2 \mid \cdots$$

$$b = \text{true} \mid \text{false}$$

The types and expressions use some abbreviations. i represents the integers, positive and negative. b represents the booleans.

kinds

$$K = \overset{\phi}{i} \mid \text{int}$$

Types are categorized into different kinds by their linearity and memory usage. The syntax includes two kinds: $\overset{\phi}{i}$ and int. The first kind is for types which take up space in memory. The i is an integer denoting how many words of memory the types of that kind use and the ϕ states if the kind is linear or non-linear. For example $\overset{\text{non}}{1}$, $\overset{\text{lin}}{1}$, and $\overset{\text{lin}}{0}$ correspond to Clay's kinds *type*, @*type*, and @*type*0. The second kind, *int*, is the kind of type parameters (e.g. in the singleton integer $Int[N]$, N has kind *int* while the singleton integer has kind $\overset{\text{non}}{1}$). As type parameters, kind *int*s don't take up space in memory. Therefore, kind *int* is not the same as C's type *int*.

types

$$\tau = \tau_1 \xrightarrow{\phi} \tau_2 \mid \phi\langle\overrightarrow{\tau}\rangle \mid I \mid \forall\alpha : K.\tau \mid \exists\alpha : K.\tau \mid \text{Int}(I) \mid \text{Mem}(I, \tau) \mid \text{bool}$$

$$I = \alpha \mid i$$

The types listed above are (in order) a function type which takes a type and returns another type (the function can be linear meaning called only once), a tuple of other types (the tuple can be linear), a type variable, an integer, a universal type,

an existential type, a singleton integer type, a memory state type, a lock type, and a boolean type. The singleton integer type and the memory state type are used together to provide linearly accessed memory. The remaining types are necessary for a sufficiently flexible abstract syntax.

expressions

$$e = i \mid b \mid x \mid e_1\, e_2 \mid e\tau \mid \phi\langle \overrightarrow{e} \rangle \mid \lambda x : \tau \xrightarrow{\phi} e$$

$$\mid \Lambda\alpha : K.v \mid \text{let}\, \langle \overrightarrow{x} \rangle = e_1 \,\text{in}\, e_2 \mid \text{if}\, e_1 \,\text{then}\, e_2 \,\text{else}\, e_3$$

$$\mid \text{pack}[\tau_1, e] \,\text{as}\, \exists\alpha : K.\tau_2 \mid \text{unpack}\, \alpha, x = e_1 \,\text{in}\, e_2$$

$$\mid \text{fix}\, x : \tau.\text{v} \mid \text{load}(e_{\text{ptr}}, e_{\text{Mem}}) \mid \text{store}(e_{\text{ptr}}, e_{\text{Mem}}, e_{\text{v}}) \mid \text{fact}$$

The expressions are the legal statements in the abstract syntax. They include (in order) integers, booleans, variables, application of one expression to another, application of an expression to a type, a tuple of other expressions (the tuple can be linear), a function abstraction which replaces a variable inside an expression, a type abstraction which replaces a type inside an expression, a *let* expression, which assigns a variable to one expression and substitutes that variable in another expression, an $if-then-else$ expression, an expression to *pack* an expression of one type as another type, and expression to *unpack* a type in an expression, an expression for recursive functions (fixed-point combinator [22]), *load* and *store* expressions for linear memory, and a $fact$ which is a value of memory state-type.

values

$$v = i \mid b \mid \Lambda\alpha : K.v \mid \text{pack}[\tau_1, v] \,\text{as}\, \exists\alpha : K.\tau_2 \mid \lambda x : \tau \xrightarrow{\phi} e \mid \phi\langle \overrightarrow{v} \rangle \mid \text{fact}$$

Some of the expressions are in an end-state, unable to progress to other expressions. These are the values. Expressions such as *load* progress to other expressions

and are therefore not in an end-state (*load* progresses to the loaded value).

environments

$$M = \{1 \mapsto v_1, \cdots, n \mapsto v_n\}$$

$$\Psi = \{1 \mapsto \tau_1, \cdots, n \mapsto \tau_n\}$$

$$\Delta = \{\alpha_1 \mapsto K_1, \cdots, \alpha_n \mapsto K_n\}$$

$$\Gamma = \{x_1 \mapsto \tau_1 \cdots, x_n \mapsto \tau_n\}$$

$$C = \Psi; \Delta; \Gamma$$

The environments are the memory environment (M), the type environment (Ψ), the kind environment (Δ), and the the variable environment (Γ). M maps memory locations to values. Ψ maps memory locations to types and shows the type of each value in M. Δ maps type variables to kinds and Γ maps variables to types. Some of the variables in Γ may be linear. Ψ, Δ, and Γ are usually used together and are collectively called C, the context. These environments allow the rules of this abstract machine to keep track of the values, types, and kinds used in an expression. For shorthand purposes,

- $\overset{\text{non}}{C}$ is the context C without any linear types.
- $C_\varnothing = \varnothing; \varnothing; \varnothing$, the empty context.
- C_1, C_2 is the union of C_1 and C_2.

3.2 The Basic Abstract Rules

The abstract rules relate kinds to types, types to expressions, and allow expressions to progress to other expressions until an end-state is reached.

kinding rules

$$(K\_IVAR) \quad \Delta \vdash i : \text{int}$$

$$(K\_BOOL) \quad \Delta \vdash \mathit{bool} : \overset{\text{non}}{1}$$

$$(K\_TVAR) \quad \Delta, \alpha : K \vdash \alpha : K$$

$$(K\_ALL) \quad \frac{\Delta, \alpha : K \vdash \tau : \overset{\phi}{i}}{\Delta \vdash \forall \alpha : K.\tau : \overset{\phi}{i}}$$

$$(K\_SOME) \quad \frac{\Delta, \alpha : K \vdash \tau : \overset{\phi}{i}}{\Delta \vdash \exists \alpha : K.\tau : \overset{\phi}{i}}$$

$$(K\_ABS) \quad \frac{\Delta \vdash \tau_1 : \overset{\phi_1}{i_1} \quad \Delta \vdash \tau_2 : \overset{\phi_2}{i_2}}{\Delta \vdash \tau_1 \xrightarrow{\phi} \tau_2 : \overset{\phi}{1}}$$

$$(K\_NLTUPLE) \quad \frac{\forall j.(\Delta \vdash \tau_j : \overset{\text{non}}{i_j})}{\Delta \vdash \text{non}\langle \overrightarrow{\tau} \rangle : \overset{\text{non}}{i}} \text{ where } (i = \sum_j i_j)$$

$$(K\_LTUPLE) \quad \frac{\forall j.(\Delta \vdash \tau_j : \overset{\phi_j}{i_j})}{\Delta \vdash_{\text{lin}} \langle \overrightarrow{\tau} \rangle : \overset{\text{lin}}{i}} \text{ where } (i = \sum_j i_j)$$

$$(K\_INT) \quad \frac{\Delta \vdash I : \text{int}}{\Delta \vdash \text{Int}(I) : \overset{\text{non}}{1}}$$

$$(K\_MEM) \quad \frac{\Delta \vdash I : \text{int} \quad \Delta \vdash \tau : \overset{\text{non}}{1}}{\Delta \vdash \text{Mem}(I, \tau) : \overset{\text{lin}}{0}}$$

The kinding or type well-formedness rules show the kind for each type. Rules in the form of a fraction state that the top part implies the bottom part. Statements in the form $A : B : C$ means that value A has type B and kind C. Part of this syntax is seen in the kinding rules to pair a type with a kind.

For example, as seen in the rule (K_INT), when type I has kind *int* given the mapping of variables to types, that mapping also holds for type $\text{Int}(I)$, a singleton

integer, which has kind non-linear size 1. This kind means the singleton integer takes up the space for an integer in memory. Rule (K_MEM) shows that when type I has kind *int* and type τ has kind non-linear size 1 given the mapping of variables to types, type $\mathrm{Mem}(I, \tau)$ has kind linear size 0 using the same mapping. This kind uses no space in memory.

These two rules show the kinds for singleton integer types and memory state types. The singleton integer types are used to point to memory locations and need to take 1 word of memory. The memory state-types are used to track the type currently residing at a memory location and control linear access to it.

typing rules

$$(T\_M) \quad \frac{\Psi = \Psi_{spare}, \Psi_e \quad \Psi_e; \Delta; \Gamma \vdash e : \tau \quad \forall i \in dom(\Psi).(C_\varnothing \vdash M(i) : \Psi(i))}{\Psi; \Delta; \Gamma \vdash (M, e : \tau)}$$

$$(T\_VAR) \quad \overset{\text{non}}{C}, x : \tau \vdash x : \tau$$

$$(T\_BOOL) \quad \overset{\text{non}}{C} \vdash b : \mathrm{bool}$$

$$(T\_FACT) \quad \overset{\text{non}}{C}, I \mapsto \tau \vdash \mathrm{fact} : \mathrm{Mem}(I, \tau)$$

$$(T\_INT) \quad \overset{\text{non}}{C} \vdash i : \mathrm{Int}(i)$$

$$(T\_LOAD) \quad \frac{C_1 \vdash e_{\mathrm{ptr}} : \mathrm{Int}(I) \quad C_2 \vdash e_{\mathrm{Mem}} : \mathrm{Mem}(I, \tau)}{C_1, C_2 \vdash \mathrm{load}(e_{\mathrm{ptr}}, e_{\mathrm{Mem}}) :^{\mathrm{lin}} \langle \tau, \mathrm{Mem}(I, \tau) \rangle}$$

$$(T\_STORE) \quad \frac{C_2 \vdash e_{\mathrm{Mem}} : \mathrm{Mem}(I, \tau_1) \quad C_3 \vdash e_{\mathrm{v}} : \tau_2 \quad C_1 \vdash e_{\mathrm{ptr}} : \mathrm{Int}(I) \quad C_1, C_2, C_3 \vdash \tau_2 : \overset{\text{non}}{1}}{C_1, C_2, C_3 \vdash \mathrm{store}(e_{\mathrm{ptr}}, e_{\mathrm{Mem}}, e_{\mathrm{v}}) : \mathrm{Mem}(I, \tau_2)}$$

$$(T\_TABS) \quad \frac{C, \alpha : K \vdash v : \tau}{C \vdash \Lambda\alpha : K.v : \forall\alpha : K.\tau}$$

$$(T\_TAPP) \quad \frac{C \vdash e : \forall\alpha : K.\tau_1 \quad C \vdash \tau_2 : K}{C \vdash e\tau_2 : [\alpha \mapsto \tau_2]\tau_1}$$

$$C = \overset{\text{non}}{C}, C_1, \cdots, C_n$$

$$(T\_TUPLE) \quad \frac{\forall i.(C_i \vdash e_i : \tau_i) \quad C \vdash \phi\langle\overrightarrow{\tau}\rangle : K}{C \vdash \phi\langle\overrightarrow{e}\rangle : \phi\langle\overrightarrow{\tau}\rangle}$$

$$(T\_ABS) \quad \frac{\Delta \vdash \tau_1 : K \quad \overset{\phi}{\Psi}; \Delta; \overset{\phi}{\Gamma}, x : \tau_1 \vdash e : \tau_2}{\overset{\phi}{\Psi}; \Delta; \overset{\phi}{\Gamma} \vdash \lambda x : \tau_1 \xrightarrow{\phi} e : \tau_1 \xrightarrow{\phi} \tau_2}$$

$$C_1, C_2 = \Psi; \Theta; \Delta; \Gamma$$

$$(T\_APP) \quad \frac{C_1 \vdash e_1 : \tau_a \xrightarrow{\phi} \tau_b \;\; C_2 \vdash e_2 : \tau_a}{C_1, C_2 \vdash e_1 e_2 : \tau_b}$$

$$(T\_LET) \quad \frac{C_a \vdash e_a : \langle\overrightarrow{\tau}\rangle \quad C_b, \overrightarrow{x : \tau} \vdash e_b : \tau_b}{C_a, C_b \vdash \text{let}\langle\overrightarrow{x}\rangle = e_a \,\text{in}\, e_b : \tau_b}$$

$$(T\_FIX) \quad \frac{C \vdash \tau :\overset{\phi}{i} \quad C, x : \tau \vdash \text{v} : \tau}{C \vdash (\text{fix}\, x : \tau.\text{v}) : \tau}$$

$$(T\_PACK) \quad \frac{\begin{array}{c} C \vdash \tau_1 : K \quad C \vdash \exists\alpha : K.\tau_2 :\overset{\phi}{i} \\ C \vdash e : [\alpha \mapsto \tau_1]\tau_2 \end{array}}{C \vdash \text{pack}[\tau_1, e] \,\text{as}\, \exists\alpha : K.\tau_2 : \exists\alpha : K.\tau_2}$$

$$(T\_UNPACK) \quad \frac{\begin{array}{c} C_1 \vdash e_1 : \exists\alpha : K.\tau_1 \\ C_2, \alpha : K, x : \tau_1 \vdash e_2 : \tau_2 \end{array}}{C_1, C_2 \vdash \text{unpack}\, \alpha, x = e_1 \,\text{in}\, e_2 : \tau_2}$$

$$(T\_IF) \quad \frac{C_a \vdash e_1 : \text{bool} \quad C_b \vdash e_2 : \tau \quad C_b \vdash e_3 : \tau}{C_a, C_b \vdash \text{if}\, e_1 \,\text{then}\, e_2 \,\text{else}\, e_3 : \tau}$$

The typing rules show the type for each expression. As mentioned above, another part of $A : B : C$ is seen here to indicate that a value has a particular type ($A : B$). The first typing rule (T_M) states that given environments for an expression e of type τ ($\Psi_e; \Delta; \Gamma$) and the rest of the environments for M (Ψ_{spare}), if for every memory location i in M, the mapping $M(i)$ has type $\Psi(i)$, then the expression e is correctly typed in the full environment. (As a note: $TABS$ is an abbreviation for type abstraction and $TAPP$ is an abbreviation for type application.)

The typing rules for integers, $facts$, $load$, and $store$ form the next piece of the linear memory puzzle. The expression i has type $\text{Int}(i)$ in a context stripped of linear values. Typing rule (T_FACT) states that when the context is separated into linear and non-linear sections, where memory location I maps to τ in the linear type environment the expression $fact$ has type $\text{Mem}(I, \tau)$. $\overset{\text{non}}{C}$ is the context C will all the linear values removed.

The rules for load and store are slightly more complex. The (T_LOAD) rules says that when the pointer has type $\text{Int}(I)$ in one environment and the matching Mem has type $\text{Mem}(I, \tau)$ in another environment, the $load$ expression has type $^{\text{lin}}\langle \tau, \text{Mem}(I, \tau)\rangle$ in a combined environment. This means the expression eventually evaluates to (returns) a tuple type of the stored value type and the Mem. The Mem has to be returned because it is linear and future loads and stores will need the Mem. The combined environment has the mappings of both separate environments. Environment combining is defined to make contradictory mappings (two mappings of the same integer to different types or values) impossible. The rule for $store$ is similar to $load$ and has three environments to combine. The type of a $store$ expression is $\text{Mem}(I, \tau)$. This means $store$ takes a pointer, a Mem, and a value to store and eventually evaluates to type $\text{Mem}(I, \tau_2)$ where $value : \tau_2$. These rules ensure that the given Mem matches the pointer to the linear memory and that memory cannot be accessed without its Mem.

evaluation rules

The evaluation rules determine how an expression progresses to a new expression. The memory environment, M, has been omitted from rules that do not affect it.

$$(E\_LOAD) \quad (R, M, \text{load}(i, fact)) \rightarrow (RM, ^{\text{lin}}\langle M(i), fact\rangle)$$

$$(E\_STORE) \quad (R, M, \text{store}(i, fact, v)) \rightarrow (R, [i \mapsto v]M, fact)$$

$$(E\_ABSAPP) \quad (\lambda x : \tau \xrightarrow{\phi} e_1)v_2 \rightarrow [x \mapsto v_2]e_1$$

$$(E\_TABSTAPP) \quad (\Lambda\alpha : K.v)\tau \rightarrow [\alpha \mapsto \tau]v$$

$$(E\_LET) \quad \text{let}\,\langle x_1, \cdots, x_n\rangle = \phi\langle v_1, \cdots, v_n\rangle\,\text{in}\,e \rightarrow [x_1 \mapsto v_1, \cdots, x_n \mapsto v_n]e$$

$$(E\_IF1) \quad \text{if}\,true\,\text{then}\,e_1\,\text{else}\,e_2 \rightarrow e_1$$

$$(E\_IF2) \quad \text{if}\,false\,\text{then}\,e_1\,\text{else}\,e_2 \rightarrow e_2$$

$$(E\_UNPACK) \quad \text{unpack}\,\alpha, x = (\text{pack}[\tau_1, v_1]\,\text{as}\,\tau_2)\text{in}\,e_2 \rightarrow [\alpha \mapsto \tau_1, x \mapsto v_1]e_2$$

$$(E\_FIX) \quad \text{fix}\,x : \tau.\text{v} \rightarrow [x \mapsto \text{fix}\,x : \tau.\text{v}]\text{v}$$

$$\frac{e \rightarrow e'}{(M, e) \rightarrow (M, e')}$$

The evaluation rules apply to expressions when the sub-expressions inside them have become values. The (E_LOAD) rule takes a singleton integer pointer to memory and a $fact$ and evaluates to a linear tuple of the value in memory and the $fact$. The typing rules (T_LOAD), (T_INT), and (T_FACT) ensure that the singleton integer matches the $fact$ and the expression evaluates to a value of the type mentioned in the $fact$ and an identical $fact$. The kinding rules (K_INT) and (K_MEM) ensure that the singleton integer is still usable and the the previous $fact$ became unusable and was replaced by the new $fact$.

The (E_STORE) rule functions similarly to the (E_LOAD) rule but changes the mapping of memory location i from its old value to the new value. The expression evaluates to a new $fact$ whose type contains the type of the newly stored value. The final evaluation rules states that if an expression evaluates to a new expression, its environment evaluates with it. This rule is used for those evaluation rules which do not affect M.

Since *load* and *store* are the only expressions which access the memory environ-

ment mapping, all memory accesses are linear and use Mems. (Note: $ABSAPP$ is an abbreviation for an expression abstraction applied to an expression and $TABSTAPP$ is an abbreviation for a type abstraction applied to a type).

congruence evaluation rules

$$E[e] = e\tau \;\mid\; \text{pack}[\tau_1, e]\,\text{as}\,\exists\alpha : K.\tau_2 \;\mid\; \text{unpack}\,\alpha, x = e\,\text{in}\,e_2$$

$$\mid\; ee_2 \;\mid\; v_1 e \;\mid\; \phi\langle e_i, \cdots, e_{k-1}, e, e_{k+1}, \cdots, e_n\rangle \;\mid\; \text{let}\,\langle\overrightarrow{x}\rangle = e\,\text{in}\,e_2$$

$$\mid\; \text{if}\,e\,\text{then}\,e_2\,\text{else}\,e_3 \;\mid\; \text{load}(e, e_{\text{Mem}}) \;\mid\; \text{load}(v_{\text{ptr}}, e)$$

$$\mid\; \text{store}(e, e_{\text{Mem}}, e_{\text{v}}) \;\mid\; \text{store}(v_{\text{ptr}}, e, e_{\text{v}}) \;\mid\; \text{store}(v_{\text{ptr}}, v_{\text{Mem}}, e)$$

$$(congruence\,rule) \quad \frac{(M, e) \rightarrow (M', e')}{(M, E[e]) \rightarrow (M', E[e'])}$$

The congruence rules determine the order in which parts of an expression can evaluate. $E[e]$ is an expression containing a sub-expression e. The rule states that the e part of the expression must evaluate first if it is not already a value. By writing the congruence rules in this format, they don't have to each be written out separately. The congruence rule states that if a sub-expression and its environment evaluate to a new sub-expression and environment, the outer expression evaluates as well. This allows the parts of an expression to progress until they are all values, at which point the whole expression evaluates. The congruence rule for tuples has been changed to make tuple evaluation order non-deterministic

erasure rules

The Erasure rules apply after type checking is complete and strip out the extraneous types from each expression. Expressions of kind size 0 such as $e : \text{Mem}(I, \tau)$ have no direct run-time affect and take up no space assuming that they do not contain infinite loops. While infinite loops are not generally desirable, they can be considered safe

for the purposes of this research if they do not violate memory. After an expression e is erased, it is referred to by d.

$$(ER\_Me) \quad \text{erase}((M, e)) \;=\; (\,\text{erase}(M),\, \text{erase}(e))$$

$$(ER\_M) \quad \text{erase}(M) \;=\; \{1 \mapsto \text{erase}(v_1), \cdots, n \mapsto \text{erase}(v_n)\}$$

$$(ER\_i) \quad \text{erase}(i) \;=\; i$$

$$(ER\_b) \quad \text{erase}(b) \;=\; b$$

$$(ER\_x) \quad \text{erase}(x) \;=\; x$$

$$(ER\_APP) \quad \text{erase}(e_1\, e_2) \;=\; \text{erase}(e_1)\, \text{erase}(e_2)$$

$$(ER\_APPT) \quad \text{erase}(e\,\tau) \;=\; \text{erase}(e)$$

$$(ER\_TUPLE) \quad \text{erase}(\phi\langle e_1, \ldots, e_n\rangle) \;=\; \langle\, \text{erase}(e_1), \ldots, \text{erase}(e_n)\rangle$$

$$(ER\_FUN) \quad \lambda x : \tau \xrightarrow{\phi} e \;=\; \lambda x \longrightarrow \text{erase}(e)$$

$$(ER\_TFUN) \quad \text{erase}(\Lambda\alpha : K.v) \;=\; \text{erase}(v)$$

$$(ER\_LET) \quad \text{erase}(\text{let}\,\langle \overrightarrow{x} \rangle = e_1 \,\text{in}\, e_2) \;=\; \text{let}\,\langle \overrightarrow{x} \rangle = \text{erase}(e_1) \,\text{in}\, \text{erase}(e_2)$$

$$(ER\_IF) \quad \text{erase}(\text{if}\, e_1 \,\text{then}\, e_2 \,\text{else}\, e_3) \;=\; \text{if}\, \text{erase}(e_1) \,\text{then}\, \text{erase}(e_2) \,\text{else}\, \text{erase}(e_3)$$

$$(ER\_PACK) \quad \text{erase}(\text{pack}[\tau_1, e] \,\text{as}\, \exists\alpha : K.\tau_2) \;=\; \text{erase}(e)$$

$$(ER\_UNPACK) \quad \text{erase}(\text{unpack}\,\alpha, x = e_1 \,\text{in}\, e_2) \;=\; \text{let}\, x = \text{erase}(e_1) \,\text{in}\, \text{erase}(e_2)$$

$$(ER\_FIX) \quad \text{erase}(\text{fix}\, x : t :\overset{\phi}{i}\, .v) \;=\; \text{fix}\, x.\, \text{erase}(v) \,\text{where}\, i > 0$$

$$(ER\_FIX0) \quad \text{erase}(\text{fix}\, x : t :\overset{\phi}{0}\, .v) \;=\; \langle\rangle$$

$$(ER\_FACT)\, \text{erase}(\text{fact}) \;=\; \langle\rangle$$

$$(ER\_LOAD) \quad \text{erase}(\text{load}(e_{\text{ptr}}, e_{\text{Mem}})) \;=\; \text{load}(\,\text{erase}(e_{\text{ptr}}),\, \text{erase}(e_{\text{Mem}}))$$

$$(ER\_STORE) \quad \text{erase}(\text{store}(e_{\text{ptr}}, e_{\text{Mem}}, e_{\text{v}})) \;=\; \text{store}(\,\text{erase}(e_{\text{ptr}}),\, \text{erase}(e_{\text{Mem}}),\, \text{erase}(e_{\text{v}}))$$

Size 0 sub-expressions erase to $\langle\rangle$ when they become values. An erased program (a big expression) will have the same intended run-time behavior as the typed program. This allows a compiler (such as for Clay) to safely compile to an unsafe language (such as C) after type checking is complete.

untyped expressions

$$d = i \;\mid\; b \;\mid\; x \;\mid\; d_1\, d_2 \;\mid\; \langle\, \overrightarrow{d}\,\rangle \;\mid\; \lambda x \longrightarrow d \;\mid\; \text{let}\, \langle\, \overrightarrow{x}\,\rangle = d_1 \,\text{in}\, d_2$$

$$\mid\; \text{if}\, d_1 \,\text{then}\, d_2 \,\text{else}\, d_3 \;\mid\; \text{fix}\, x.u \;\mid\; \text{load}(d_{\text{ptr}}, d_{\text{Mem}}) \;\mid\; \text{store}(d_{\text{ptr}}, d_{\text{Mem}}, d_{\text{v}})$$

untyped values

$$u = i \;\mid\; b \;\mid\; \lambda x \longrightarrow d \;\mid\; \langle\, \overrightarrow{u}\,\rangle$$

After type erasure, the remaining expressions and values use space in memory or have an actual run-time affect. The load and store expressions will only be legal here if the original typed expressions were legal so linear memory is still enforced. The symbol for expressions has changed from e to d and from v to u, a step backwards indicating that the types have been removed.

untyped evaluation rules

$$(D\_LOAD) \quad (L, \text{load}(i, \langle\rangle)) \to (L, \langle L(i), \langle\rangle\rangle)$$

$$(D\_STORE) \quad (L, \text{store}(i, \langle\rangle, u)) \to ([i \mapsto u]L, \langle\rangle)$$

$$(D\_ABSAPP) \quad (\lambda x \longrightarrow d_1)u_2 \to [x \mapsto u_2]d_1$$

$$(D\_FIX) \quad \text{fix}\, x.u \to [x \mapsto \text{fix}\, x.u]u$$

Several evaluation rules cannot apply to the untyped expressions. The above untyped evaluation rules work where the previous rules have the wrong number of arguments. The other untyped evaluation rules have the same format as the original evaluation rules and simply require a name change from (E_RULE) to (D_RULE) so they have been omitted.

untyped environments

$$L = \{1 \mapsto u_1, \cdots, n \mapsto u_n\}$$

The environments handling types do not exist after type erasure. This leaves the memory environment which maps memory locations to values. This environment now handles untyped expressions and values and following the re-naming trend, its name is changed from M to L.

A simple typed expression in this abstract syntax might look like:

```
(M, load(500,store(500, fact, 3)))
```

This expression stores the value 3 into memory location 500 and then loads it. It progresses as follows (with types shown for clarity):

```
(M,         load(500,store(500:Int[500], fact:Mem[500,I], 3:Int[3])))
([500->3]M, load(500:Int[500], fact:Mem[500,3])) By (E_STORE)
([500->3]M, lin<3,fact>:<Int[3],Mem[500,3]>)     By (E_LOAD)
```

The same expression would appear as follows after type erasure:

```
(L, load(500,store(500,<>,3))
```

and would progress as follows:

```
(L,         load(500,store(500,<>,3))
([500->3]L, load(500,<>))            By (D_STORE)
([500->3]L, <3,<>>)                  By (D_LOAD)
```

In an operating system stack memory can be viewed as a linear array of words. The abstract machine allows safe access to a linear memory stack. The typing and

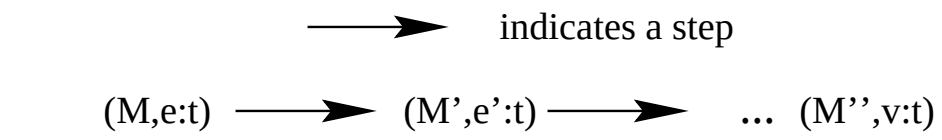

Figure 11: Preservation: Types are preserved as an expressions steps.

Figure 12: Progress: Non-values will step.

kinding rules enforce the linearity. Hawblitzel et al. [12] use Mem to build lists, arrays, and regions of nonlinear data that must be accessed linearly.

3.3 Soundness Proofs

[14] presents the soundness proofs for the basic abstract machine. Soundness means that well-typed programs will never get stuck and will follow the same specification as the program progresses. A soundness proof consists of a proof of type preservation and a proof of progress. Progress says that an expression which obeys the language specification will make progress until it becomes a value. Preservation says that the new expressions created when the initial expression progresses to a value also obey the specification. The report also shows a proof of erasure that shows that the size 0 types can be erased without changing the run-time behavior.

Formally stated, the theorems are as follows:

Preservation: If $\Psi_e; \Delta; \Gamma \vdash e : \tau$, $\Psi_{spare}, \Psi_e; \Delta; \Gamma \vdash (M, e : \tau)$ and $(M, e) \rightarrow (M', e')$, then $\Psi'_e; \Delta; \Gamma \vdash e' : \tau$ and Ψ_{spare}, Ψ'_e;
$\Delta; \Gamma \vdash (M', e' : \tau)$. If (M, e) is well-typed and (M, e) progresses in one step to some (M', e') then e' has the same type as e. This means that the type of an expression is preserved when its subexpressions progress. Figure 11 shows a visual representation of type preservation.

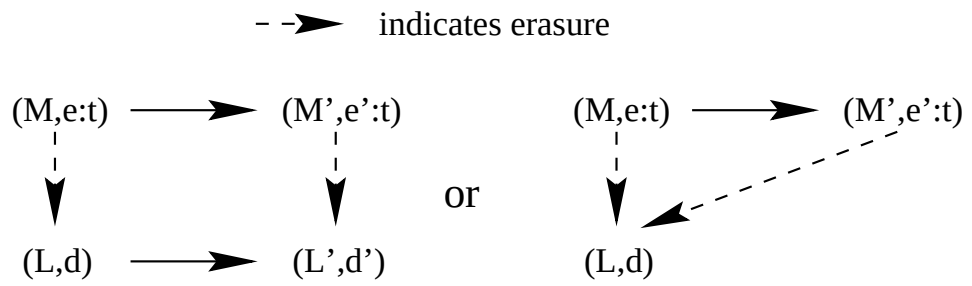

Figure 13: Erasure1: If an expression steps to another expression, there is a distance of zero or one steps between their erasures.

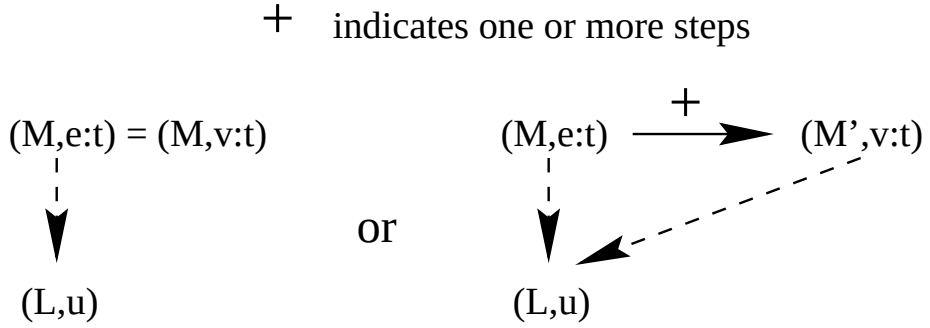

Figure 14: Erasure2: If the erased expression is a value, then the typed expression evaluates in zero or more steps to a value which erased to the original erased expression.

Progress: If (M, e) is closed and well-typed ($C_{Me} \vdash (M, e : \tau)$ for some τ and $C_{Me} = \Psi_{Me}; \varnothing; \varnothing$), then either e is a value or else there is some (M', e') so that $(M, e) \rightarrow (M', e')$. This proof shows that well-typed expressions cannot get stuck until they become values. Figure 12 shows a graphic example of progress.

Progress definitions

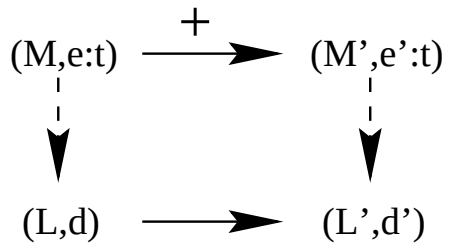

Figure 15: Erasure3: Each step taken by an untyped expression is equivalent to one or more steps taken by its typed expression. The extra steps handle only types.

- $(M_1, e_1) \overset{?}{\mapsto} (M_2, e_2)$ means (M_1, e_1) progresses in zero or one steps to $(, M_2, e_2)$.
- $(M_1, e_1) \overset{*}{\mapsto} (M_2, e_2)$ means (M_1, e_1) progresses in zero or more steps to (R_2, M_2, e_2).
- $(M_1, e_1) \overset{+}{\mapsto} (M_2, e_2)$ means (M_1, e_1) progresses in one or more steps to (M_2, e_2).
- $(M_1, e_1) \overset{?,(evaluation-rule)}{\mapsto} (M_2, e_2)$ means (M_1, e_1) progresses to (M_2, e_2) by applying the given evaluation rule zero or one times.
- $(M_1, e_1) \overset{*,(evaluation-rule)}{\mapsto} (M_2, e_2)$ and $(M_1, e_1) \overset{+,(evaluation-rule)}{\mapsto} (M_2, e_2)$ are defined in the same way as the above definition.

Erasure: If (M, e) is closed and well-typed ($C_{Me} \vdash (M, e : \tau)$ for some τ and $C_{Me} = \Psi_{Me}; \varnothing; \varnothing$), then the following holds:

1. If $(M, e) \rightarrow (M', e')$ then $\text{erase}((M, e)) \overset{?}{\rightarrow} \text{erase}((M', e'))$. If (M, e) evaluates in one step to (M', e'), then $\text{erase}((M, e))$ evaluates in zero or one steps to $\text{erase}((M', e'))$. This proof shows that the intended run-time affect is not lost during erasure. The erased expression will follow a parallel progression to that of the typed expression. Figure 13 shows a graphic example of this erasure sub-theorem.

2. If $\text{erase}(e)$ is a value then $(M, e) \overset{*}{\rightarrow} (M', v)$, $\text{erase}((M, e)) = \text{erase}((M', v))$. If the erasure of e is a value then (M, e) evaluates in zero or more steps to some (M', v') where the erasure of (M, e) equals the erasure of (M', v'). This proof states that only extraneous types are erased from the values. The steps that took (M, e) to (M', v) must have handled only extraneous types. Figure 14 shows a graphic example of this erasure sub-theorem.

3. If $\text{erase}((M, e)) \rightarrow (L', d')$ then $(M, e) \overset{+}{\rightarrow} (M', e')$, $\text{erase}((M', e')) = (L', d')$. If $\text{erase}((M, e))$ evaluates in one step to (L', d') then (M, e) evaluates in one or more steps to (M', e'), and $\text{erase}((M', e'))= (L', d')$. This proof states that a step taken by an erased (M, e) is equivalent to one or more steps taken by the

unerased (M, e). One of the steps had an actual run-time affect. The others handled only extraneous types. Figure 15 shows a graphic example of this erasure sub-theorem.

3.4 Adding Concurrency to the Abstract Machine

The basic abstract machine does not contain any notion of multiple processes or threads. A single thread is basically a big expression. A tuple of expressions can simulate a machine running several threads.

$$\langle \text{thread}_1,\ \text{thread}_2,\ \text{thread}_3 \rangle$$

In the basic abstract machine these threads would evaluate one at a time and not interact because there is no means of passing data between them.

A simple non-deterministic process manager is implemented in the congruence rule for tuples. This rule allows any non-value sub-expression in a tuple to evaluate next so the overall evaluation order is non-deterministic. The congruence rule for tuples is non-deterministic and evaluation of the tuple causes some thread to take a step so all threads will eventually progress by taking interleaved steps in a non-deterministic fashion. This allows formal concurrency in my abstract syntax.

The following shows a simple example of several threads using the concurrent abstract syntax:

$$\langle \text{store}(500, \text{fact}, 3), \text{store}(504, \text{fact}, 4), \text{store}(508, \text{fact}, 5) \rangle$$

Each thread stores an integer into a memory location. A valid progression of steps

for this multi-thread program could be

$$\langle \text{store}(500, \text{fact}, 3), \text{store}(504, \text{fact}, 4), \text{store}(508, \text{fact}, 5) \rangle$$

$$\langle \text{fact}, \text{store}(504, \text{fact}, 4), \text{store}(508, \text{fact}, 5) \rangle$$

$$\langle \text{fact}, \text{store}(504, \text{fact}, 4), \text{fact} \rangle$$

$$\langle \text{fact}, \text{fact}, \text{fact} \rangle$$

or any other combination of three threads which each take one step.

A series of *let* expressions can force deterministic behavior in a tuple when needed.

$$\text{let } x = e_1 \text{ in let } y = e_2 \text{ in let } z = e_3 \text{ in } \langle x, y, z \rangle$$

This *let* expression forces the sub-expressions in the tuple to evaluate in order. Therefore, the change which allows non-deterministic evaluation order does not prohibit any previous functionality.

3.5 Adding Locks to the Abstract Machine

In a typical operating system, access to data structures and devices may be shared by several threads. The abstract machine with concurrency does not have a way to share data safely. I added syntax and rules to my abstract machine to implement shared data. This section discusses some basic locking issues and shows abstract machine additions which will prevent locking errors that cause memory safety violations.

State types can control access to data structures and devices in operating system. To share this access, a state type can be locked to allow only one thread to access the device or data at a time. Untyped locks are vulnerable to many safety violations. A safely typed lock needs to take these vulnerabilities into account.

In general, a lock is associated with a bit of memory. When the bit value is 0, the lock is free and any thread may claim it, changing the value to 1. When the bit

```
acquire(lock);       acquire(lock);       acquire(lock);
// critical section  // critical section  // critical section
release(lock);       release(lock);       release(lock);
```

Figure 16: Correct access to the same critical section by 3 threads.

```
acquire(lock);                            acquire(lock2);
// critical section  // critical section  // critical section
release(lock);                            release(lock2);
```

Figure 17: Three attempts to access the critical section by acquiring its lock. The first code section is correct while the second and third code sections incorrectly acquire the lock.

value is 1, the lock is already held by a thread and the other threads must try again later or wait. If the lock is implemented so that those wanting the lock wait until it is free, the lock is called "blocking". When the thread holding the lock releases it, the bit value is reset to 0.

Typically each lock is associated with a critical section or set of critical sections. In Figure 16, three threads acquire a lock, access the critical section, and release the lock. If the threads are all running concurrently then the order in which they acquire the lock is non-deterministic.

3.5.1 Problems with Locks

Forget to acquire

Standardly, locks in C (the Posix libraries) do not enforce the matching of a particular lock with a critical section. They also do not force the thread to acquire any lock at all when accessing the critical section. This allows the following to occur.

The first code section in Figure 17 correctly acquires the lock for the critical section. The second code section forgets to acquire any lock and doesn't prevent simultaneous access by the first code section. This could cause inconsistent data. For example, if both critical sections read some shared memory and then write to it, the second section to read might see the original value or the value just set by the

```
acquire(lock);        acquire(lock);        acquire(lock);
// critical section   // critical section   // critical section
release(lock);                              release(lock2);
```

Figure 18: Three attempts to release the critical section's lock. The first code section is correct while the second and third code sections incorrectly release the lock.

```
acquire(lock);
acquire(lock);
```

Figure 19: A re-entrant lock.

first section. The third code section acquires the wrong lock and not only doesn't prevent simultaneous access by the first code section but it ties up the lock2 for no reason.

Forget to release

The first code section in Figure 18 correctly acquires the lock for the critical section. The second code section forgets to release the lock and could cause deadlock if any further code sections attempt to acquire the lock. The third code section forgets to release lock1 and releases lock2 instead. Forgetting lock1 could cause deadlock as in the second code section. If some other code section was still holding lock2, then this could cause inconsistent data when another code section acquires lock2.

Re-entrant locks

A re-entrant lock as seen in Figure 19, occurs when a thread tries to acquire a lock it already holds. The thread running this code section will wait forever because it cannot acquire the lock until it releases it. This can cause deadlock if the lock system is unprepared for such errors. This type of error can easily occur in large code sections where the programmer forgets the lock is still held.

```
acquire(lock);
release(lock);
release(lock);
```

Figure 20: Releasing an unheld lock.

Release unheld locks

If a thread releases a lock it does not hold, as seen in Figure 20, it could cause some other thread to lose its hold on the lock causing inconsistent data.

The above-mentioned locking errors can cause data inconsistencies and deadlock in code vital to the running of an operating system. Deadlock is technically safe as long as memory is not violated but data inconsistencies are a patent safety violation. A multi-threaded safe operating system kernel would therefore need support for safe locks.

3.5.2 Safer Locks

In order to add support for locks, the following types, expressions, values, and environments were added to the abstract machine:

types

$$\tau+ = \text{Lock}(I, \tau_2)$$

The type $\text{Lock}(I, \tau)$ means that a lock exists for a value of type τ and the lock bit is located at memory location I.

values

$$v+ = \text{lock}$$

The value *lock* is the lock itself and has type $\text{Lock}(I, \tau)$.

expressions

$$e+ = \text{create\_ lock}(e_{\text{ptr}}, e_{\text{Mem}}, e_{\text{resource}}) \;\mid\; \text{acquire}(e_{\text{Lock}}, e_{\text{ptr}})$$

$$\mid\; \text{release}(e_{\text{Lock}}, e_{\text{ptr}}, e_{\text{resource}}) \;\mid\; \text{lock}$$

The *create_lock*, *acquire*, and *release* expressions allow for creation, acquiring, and release of a lock.

environments

$$R = \{1 \mapsto \langle \overrightarrow{v_1} \rangle, \cdots, n \mapsto \langle \overrightarrow{v_n} \rangle\}$$

$$\Theta = \{1 \mapsto \langle \overrightarrow{\tau_1} \rangle, \cdots, n \mapsto \langle \overrightarrow{\tau_n} \rangle\}$$

The new resource environment, R, and its type Θ track the lock controlled resources and their linear lock bit pointers. R contains the set of lock bit Mem values and the currently free resources. Θ contains the lock bit Mem type and the resource type even when the lock is held and resource is not in the R environment.

The evaluation, type well-formedness, type checking, and erasure rules needed to enforce the lock abstractions are as follows:

evaluation rules

$$(E\_CREATELOCK) \quad (R, M, \text{create\_ lock}(i, \text{fact}, v_{\text{resource}}))$$
$$\rightarrow ([i \mapsto \langle \text{fact}, v_{\text{resource}} \rangle]R, M, \text{lock})$$

$$(E\_ACQUIRE1) \quad (R, M, \text{acquire}(\text{lock}, i))$$
$$\rightarrow ([i \mapsto \langle \text{fact} \rangle]R, [i \mapsto 1]M, v_{\text{resource}})$$
$$\text{where } R(i) = \langle \text{fact}, v_{\text{resource}} \rangle \text{ and } M(i) = 0$$

$$(E\_ACQUIRE2) \quad (R, M, \text{acquire}(\text{lock}, i)) \rightarrow (R, M, \text{acquire}(\text{lock}, i))$$
$$\text{where } R(i) = \langle \text{fact} \rangle \text{ and } M(i) = 1$$

$$(E\_RELEASE1) \quad (R, M, \text{release}(\text{lock}, i, v_{\text{resource}}))$$
$$\rightarrow ([i \mapsto \langle \text{fact}, v_{\text{resource}} \rangle]R, [i \mapsto 0]M, \langle\rangle)$$
$$\text{where } R(i) = \langle \text{fact} \rangle \text{ and } M(i) = 1$$

$$(E\_RELEASE2) \quad (R, M, \text{release}(\text{lock}, i, v_{\text{resource}}))$$
$$\rightarrow ((R, M, \text{release}(\text{lock}, i, v_{\text{resource}})))$$
$$\text{where } R(i) = \langle \text{fact}, v_{\text{resource}} \rangle \text{ and } M(i) = 0$$

The congruence rules that would let each of these expressions progress one non-value parameter at a time are of the same format at those mentioned above and allow the parameters to evaluate from left to right until all parameters are values.

The *create_lock* expression evaluates to a *lock* when its parameters are a pointer describing the location of the lock bit (I), the $fact$ for the lock bit, and the resource state-type value to be locked. The $fact$ for the lock bit is not returned so that only the lock can change the value of this bit. The $fact$ and the resource are both stored in the R environment under $R(I) = \langle \text{fact}, v_{\text{resource}} \rangle$ and their types are stored in the Θ environment under $\Theta(I) = \langle \text{Mem}(I, \tau), \tau \rangle$.

The *acquire* expression evaluates to itself when its parameters are values but $M(I) = 1$ meaning the lock is already held. Thus, the expression blocks until the lock is freed. When $M(I) = 0$ and the parameters are the *lock* and the pointer to its lock bit, the expression evaluates to the resource state-type value. The resource state-type value, being linear, is removed from the R environment though its type is still stored in the Θ environment. The value of the lock bit changes to 1 indicating the lock is now held so $M(I) = 1$, $R(I) = \langle \text{fact} \rangle$, $\Psi(I) = \text{Int}(1)$, and $\Theta(I) = \langle \text{Mem}(I, \tau), \tau \rangle$.

The *release* expression evaluates to itself when its parameters are values but $M(I) = 0$ meaning the lock was already free. When $M(I) = 1$ and the parameters are a lock, a pointer to the lock bit (I), and the locked resource state-type value, the resource is once again stored in the R environment and the lock bit value is

changed to 0 indicating that the lock is free so $M(I) = 0$, $R(I) = \langle \text{fact}, v_{\text{resource}} \rangle$, $\Psi(I) = \text{Int}(0)$, and $\Theta(I) = \langle \text{Mem}(I, \tau), \tau \rangle$. A discussion of why the "releasing a free lock" expression blocks is shown in Section 3.5.5.

kinding rules

$$(K\_LOCK) \quad \frac{\Delta \vdash I : \text{int} \quad \Delta \vdash \tau :^{\text{lin}} 0}{\Delta \vdash \text{Lock}(I, \tau) :^{\text{non}} 0}$$

The kinding rule (K_LOCK) specifies that type *Lock* is non-linear and takes no space in memory. The *Lock* depends on a singleton integer type which is non-linear and some type which is linear size 0. The singleton integer type refers to the memory location of the lock bit and the other type refers to the linear size 0 resource to be locked. The non-linearity allows the values of type *Lock* to be duplicated and passed around in operating system code so that any thread which might need to acquire the lock has a copy of it.

valid lock states

$$(T\_FREE) \quad \text{Int}(0); \langle \tau_{Mem}, \tau_{resource} \rangle \vdash \langle \tau_{Mem}, \tau_{resource} \rangle$$

$$(T\_HELD) \quad \text{Int}(1); \langle \tau_{Mem}, \tau_{resource} \rangle \vdash \langle \tau_{Mem} \rangle$$

The two valid lock states are held and free. When a lock is free, the Ψ environment maps its lock pointer to Int(0) and the Θ environment maps its lock pointer to $\langle \tau_{Mem}, \tau_{resource} \rangle$. In this case, the resource is stored in the Θ environment. When a lock is held, the Ψ environment maps its lock pointer to Int(1) and the Θ environment maps its lock pointer to $\langle \tau_{Mem} \rangle$. Here, the resource is being used and thus not stored in the Θ environment.

typing rules

$$\Psi = \Psi_{spare}, \Psi_e, \Psi_{R1}, \cdots, \Psi_{Rn} \quad \Psi_e; \Theta; \Delta; \Gamma \vdash e : \tau$$
$$\forall i \in dom(\Psi).(\varnothing; \Theta; \varnothing; \varnothing \vdash M(i) : \Psi(i))$$
$$(T\_RMe) \quad \frac{\forall i \in dom(\Theta).(\Psi_{Ri}; \Theta; \varnothing; \varnothing \vdash R(i) : \tau_i \text{ and } \Psi(i); \Theta(i) \vdash \tau_i)}{\Psi; \Theta; \Delta; \Gamma \vdash (R, M, e : \tau)}$$

$$(T\_LOCK) \quad \varnothing; \Theta, I \mapsto \langle \mathrm{Mem}(I, \mathrm{Int}(J)), \tau_2 \rangle; \Delta; \overset{\text{non}}{\Gamma} \vdash \mathrm{lock} : \mathrm{Lock}(I, \tau_2)$$

$$(T\_CREATELOCK) \quad \frac{C_1 \vdash e_{\mathrm{ptr}} : \mathrm{Int}(I) \quad C_2 \vdash e_{\mathrm{Mem}} : \mathrm{Mem}(I, \mathrm{Int}(0)) \quad C_3 \vdash e_{\mathrm{resource}} : \tau : \overset{\text{lin}}{0}}{C_1, C_2, C_3 \vdash \mathrm{create\_lock}(e_{\mathrm{ptr}}, e_{\mathrm{Mem}}, e_{\mathrm{resource}}) : \mathrm{Lock}(I, \tau)}$$

$$(T\_ACQUIRE) \quad \frac{C \vdash e_{\mathrm{Lock}} : \mathrm{Lock}(I, \tau) \quad e_{\mathrm{ptr}} : \mathrm{Int}(I)}{C \vdash \mathrm{acquire}(e_{\mathrm{Lock}}, e_{\mathrm{ptr}}) : \tau}$$

$$(T\_RELEASE) \quad \frac{C_1 \vdash e_{\mathrm{Lock}} : \mathrm{Lock}(I, \tau) \quad e_{\mathrm{ptr}} : \mathrm{Int}(I) \quad C_2 \vdash e_{\mathrm{resource}} : \tau}{C_1, C_2 \vdash \mathrm{release}(e_{\mathrm{Lock}}, e_{\mathrm{ptr}}, e_{\mathrm{resource}}) :^{\mathrm{non}} \langle\rangle}$$

(T_LOCK) specifies that the lock type matches the contents of the Θ environment. Additionally, the value of the lock bit must be 0 meaning the lock starts out unheld. $(T\_CREATELOCK)$ ensures that the pointer to the lock bit matches the fact for the lock bit. It also ensures that the returned lock matches the resource type and the lock bit location. $(T\_RELEASE)$ ensures that the lock matches the pointer and the resource.

erasure rules

$$(ER\_RMe) \quad \mathrm{erase}((R, M, e)) \;=\; (\,\mathrm{erase}(M),\, \mathrm{erase}(e))$$

$$(ER\_LOCK) \quad \mathrm{erase}(\mathrm{lock}) \;=\; \langle\rangle$$

$$(ER\_CREATELOCK) \quad \mathrm{erase}(\mathrm{create\_lock}(e_{\mathrm{ptr}}, e_{\mathrm{Mem}}, e_{\mathrm{resource}}))$$
$$= \mathrm{create\_lock}(\,\mathrm{erase}(e_{\mathrm{ptr}}),\, \mathrm{erase}(e_{\mathrm{Mem}}),\, \mathrm{erase}(e_{\mathrm{resource}}))$$

$$(ER\_ACQUIRE) \quad \mathrm{erase}(\mathrm{acquire}(e_{\mathrm{Lock}}, e_{\mathrm{ptr}})) \;=\; \mathrm{acquire}(\,\mathrm{erase}(e_{\mathrm{Lock}}),\, \mathrm{erase}(e_{\mathrm{ptr}}))$$

$$(ER\_RELEASE) \quad \text{erase}(\text{release}(e_{\text{Lock}}, e_{\text{ptr}}, e_{\text{resource}}))$$
$$= \text{release}(\text{erase}(e_{\text{Lock}}), \text{erase}(e_{\text{ptr}}), \text{erase}(e_{\text{resource}}))$$

The erasure rules remove the lock and the resource state-type value but keep the pointer to the lock bit. Expressions that were legal under the evaluation, type well-formedness and type checking rules will still have the same safety properties after erasure. Therefore the untyped locks will still be handled safely.

untyped evaluation rules

$$(D\_CREATELOCK) \quad (L, \text{create\_ lock}(i, \langle\rangle, \langle\rangle)) \rightarrow (L, \langle\rangle) \text{ where } L(i) = 0$$

$$(D\_ACQUIRE1) \quad (L, \text{acquire}(\langle\rangle, i)) \rightarrow ([i \mapsto 1]L, \langle\rangle) \text{ where } L(i) = 0$$

$$(D\_ACQUIRE2) \quad (L, \text{acquire}(\langle\rangle, i)) \rightarrow (L, \text{acquire}(\langle\rangle, i)) \text{ where } L(i) = 1$$

$$(D\_RELEASE1) \quad (L, \text{release}(\langle\rangle, i, \langle\rangle)) \rightarrow ([i \mapsto 0]L, \langle\rangle) \text{ where } L(i) = 1$$

$$(D\_RELEASE2) \quad (L, \text{release}(\langle\rangle, i, \langle\rangle)) \rightarrow (L, \text{release}(\langle\rangle, i, \langle\rangle)) \text{ where } L(i) = 0$$

Untyped evaluation rules are necessary for each of the *create_lock*, *acquire*, and *release* evaluation rules because the Mem, the $Lock$, and the size 0 $resource$ have been erased to $\langle\rangle$ to create the untyped expressions.

3.5.3 Example of Locks in the Abstract Machine

The following example of locks in the abstract machine creates a lock for a $fact$ which guards memory location 500. The lock is acquired so 50 can be stored in the

guarded location and then the lock is released:

$$
\begin{array}{l}
(\lambda x :^{\mathrm{lin}} \langle \mathrm{Mem}(500, \tau_{lock}), \mathrm{Mem}(504, \tau_{resource}) \rangle \xrightarrow{\mathrm{non}} \\
\quad \mathrm{let}\, \langle x_{lock}, x_{resource} \rangle = x \,\mathrm{in} \\
\quad\quad \mathrm{let}\, y = \mathrm{create\_lock}(500, x_{lock}, x_{resource}) \,\mathrm{in} \\
\quad\quad\quad \mathrm{release}(y, 500, \mathrm{store}(504, \mathrm{acquire}(y, 500), 50))) \\
(^{\mathrm{lin}} \langle \mathrm{fact}, \mathrm{fact} \rangle)
\end{array}
$$

This example evaluates using the rules in the abstract machine.

$$
\begin{array}{l}
(R, [500 \mapsto 0]M, \quad (\lambda x :^{\text{lin}} \langle \text{Mem}(500, \tau_{lock}), \text{Mem}(504, \tau_{resource}) \rangle \xrightarrow{\text{non}} \\
\quad \text{let } \langle x_{lock}, x_{resource} \rangle = x \text{ in} \\
\qquad \text{let } y = \text{create\_lock}(500, x_{lock}, x_{resource}) \text{ in} \\
\qquad\quad \text{release}(y, 500, \text{store}(504, \text{acquire}(y, 500), 50)))) \\
(^{\text{lin}} \langle \text{fact}, \text{fact} \rangle)) \\
\qquad \xrightarrow[(E\_ABSAPP)]{} \\
(R, [500 \mapsto 0]M, \quad \text{let } \langle x_{lock}, x_{resource} \rangle =^{\text{lin}} \langle \text{fact}, \text{fact} \rangle \text{ in} \\
\qquad \text{let } y = \text{create\_lock}(500, x_{lock}, x_{resource}) \text{ in} \\
\qquad\quad \text{release}(y, 500, \text{store}(504, \text{acquire}(y, 500), 50))) \\
\qquad \xrightarrow[(E\_LET)]{} \\
(R, [500 \mapsto 0]M, \quad \text{let } y = \text{create\_lock}(500, \text{fact}, \text{fact}) \text{ in} \\
\qquad\quad \text{release}(y, 500, \text{store}(504, \text{acquire}(y, 500), 50))) \\
\qquad \xrightarrow[(E\_CREATELOCK)]{}) \\
([500 \mapsto \langle \text{fact}, \text{fact} \rangle]R, [500 \mapsto 0]M, \quad \text{let } y = \text{lock} \text{ in} \\
\qquad\quad \text{release}(y, 500, \text{store}(504, \text{acquire}(y, 500), 50))) \\
\qquad \xrightarrow[(E\_LET)]{} \\
([500 \mapsto \langle \text{fact}, \text{fact} \rangle]R, [500 \mapsto 0]M, \\
\qquad \text{release}(\text{lock}, 500, \text{store}(504, \text{acquire}(\text{lock}, 500), 50))) \\
\qquad \xrightarrow[(E\_ACQUIRE1)]{} \\
([500 \mapsto \langle \text{fact} \rangle]R, [500 \mapsto 1]M, \quad \text{release}(\text{lock}, 500, \text{store}(504, \text{fact}, 50))) \\
\qquad \xrightarrow[(E\_STORE)]{} \\
([500 \mapsto \langle \text{fact} \rangle]R, [500 \mapsto 1, 504 \mapsto 50]M, \quad \text{release}(\text{lock}, 500, \text{fact})) \\
\qquad \xrightarrow[(E\_RELEASE1)]{} \\
([500 \mapsto \langle \text{fact}, \text{fact} \rangle]R, [500 \mapsto 0, 504 \mapsto 50]M, \quad \langle\rangle)
\end{array}
$$

The first step evaluates the function abstraction and sets x equal to the pair of

$facts$. The second step separates x into its component $facts$. This step evaluates the top level *let* expression and substitutes $facts$ for x_{lock} and $x_{resource}$. The third step evaluates the *create_lock* subexpression to a *lock*. The fourth step evaluates the remaining *let* expression and substitutes the *lock* into the remaining inner expression. The fifth step evaluates the *acquire* expression to the resource, a $fact$. The sixth step evaluates the *store* expression using that $fact$ and stores the value 50 into the guarded memory location. This step returns the $fact$ which is released back into R on the seventh step. As the lock is acquired and released, the M and R environments are updated. This example showed the safe usage of a lock in the abstract machine.

3.5.4 Prevented Lock Problems

The additional abstract syntax and rules prevent many of the aforementioned lock problems.

When access to the actual resource requires the v_{resource}, the lock must be acquired before the resource can be accessed. This access must be controlled by enforcement functions such as load and store, which require the correct $fact$ before a word of memory can be accessed. Therefore the resource cannot be used if no lock or the wrong lock is acquired. When the lock is released, the v_{resource} is handed back to the lock and the old copy is invalid (because it is linear) so the v_{resource} can no longer be passed to access functions for the resource.

The "forget to release" error causes deadlock and is not solved by the abstract machine. A programming language such as Clay, which implements the abstract machine could notice when a function ends abandoning a linear variable and produce an error message at compile time. The error is, however, not a memory safety violation.

The "re-entrant locks" problem can also cause deadlock and is not solved by the abstract machine. As with the "forget to release" error, it could be solved by

implementing lock numbering in Clay. Lock numbering can be used to tell when a lock is doubly acquired. A wrapper around the lock functions which implements lock numbering would solve this problem.

To release a lock, a valid v_{resource} must be handed to the *release* expression. Since the v_{resource} is only valid when the lock is held, it cannot be released when it is already free. If a duplicate resource is created and handed to the release function, it will block until the lock is acquired.

Together, *lock*, *create_lock*, *acquire*, and *release* allow creation of a nonlinear lock on a linear resource. Copies of the lock may be passed around but only one lock copy may access the resource at a time. A locked resource cannot be accessed by acquiring the wrong lock or no lock at all. An unheld lock is not able to be released. This means that locks in my abstract syntax cannot cause memory safety violations by improperly locking shared data. Standardly, locks implemented with acquire and release functions, instead of a lock clause, can be risky if a lock is released in the middle of a conditional expression [3]. However, an expression which uses a lock after it is released will not type check in my abstract machine. Therefore, the machine can safely use the more powerful acquire and release implementation.

3.5.5 Lock Discussion

One lock for multiple copies of a resource

The abstract machine allows a given lock to control only one copy of a resource even though an operating system may have several copies of a resource to place under one lock. This section discusses an alternate lock specification to the one presented in the abstract syntax.

A basic lock controlling one copy of a resource is also known as a bit semaphore. The lock bit needs 1 bit to encode on and off. An integer semaphore which encodes numbers controls multiple identical copies of a resource thus allowing code to increase

the semaphore by releasing resources into the lock and decrease the semaphore by acquiring resources from the lock. When the lock integer is 0, the acquire function blocks until some other code releases a resource into the lock.

```
lock = createlock(lock_int, lock_mem); // empty! no resources yet.
release(lock_int, lock, resource1);    // now it has a resource
release(lock_int, lock, resource2);    // 2 resources checked in
release(lock_int, lock, resource3);    // 3 resources checked in
resource = acquire(lock_int, lock);    // back to 2 resources
```

In the abstract syntax, unheld resources can be stored in R, the resource environment. In the multi-resource implementation, releasing an unheld lock adds a resource to the lock instead of causing an error.

Acquire blocks until a copy of the resource is available and then decrements the lock integer. The test and decrement operation needs to be atomic to prevent overlapping acquire calls which decrement the lock below 0. Release increments the lock integer. There is no upper limit on the number of unheld resources allowed in this implementation.

The memory and resource environments for a lock integer i currently guarding 3 unheld resources:

$$M[i] = 3$$

$$R[i] = \langle fact, resource, resource, resource \rangle$$

$$\Theta[i] = \langle Mem[I, \mathrm{Int}(3)], t_{resource} \rangle$$

The memory environment M is unchanged. The resource environment R contains a tuple of the lock integer $fact$ and all the unheld resources. The resource type environment is unchanged because all the resource copies have the same type.

If desired, a maximum number of checked-in resources could be set. In this case, the release function blocks if the maximum number of resources is already checked

in.

My abstract machine uses one lock per resource because the device driver did not required any multi-resource locks.

One lock for one copy of a multi-copy resource

The ($E\_RELEASE1$) evaluation rule in the abstract syntax tests $M(i)$ and $R(i)$ to make sure the lock is held before releasing it. Without that test, ($E\_RELEASE1$) will release any same typed resource with the lock for that type. If the lock is meant to manage only one copy of the resource and falsely assumes only one copy exists, it could release the wrong copy. For example, in Clay's syntax:

$$@type0\ Foo[type\ T]\ =\ exists\ [intI]\ Mem[I, T]$$

$Foo[T]$ is a linear size 0 state type which takes a type parameter T with kind *type* and tracks a location of this T in memory. There can be easily several values of type $Foo[T]$ in existence at one time for a given T (T could be the singleton integer 3). If there are two values of type $Foo[T]$; $resource\_one$ and $resource\_two$:

$$lock\ =\ create\_lock(lock\_bit,\ lock\_bit\_Mem,\ resource\_one);$$
$$resource\_one\ =\ acquire(lock,\ lock\_bit);$$
$$release(lock,\ lock\_bit,\ resource\_two);$$
$$release(lock,\ lock\_bit,\ resource\_one);$$

The first call to release passes in the wrong resource but succeeds. The second call would cause havoc as $resource\_two$ is over-written or $resource\_one$ is lost or the lock bit fails to increment. Normally this problem cannot exist because only one copy of a resource with that exact type exists. A previously released resource is invalid so it can't be re-released. In this case, there are multiple instances of the same typed

resource.

There are several ways to handle this problem:

1. Release blocks if the lock is already unheld. This could be useful for process communication. In a system with two communicating threads, if $thread_2$ needs to know that $thread_1$ has begun a task, it can release an unheld lock. When $thread_1$ acquires the lock, it immediately releases leaving $thread_1$ with a resource copy and leaving $thread_2$ with the knowledge that $thread_1$ has started the task. $Thread_2$ can then acquire the lock, retrieving its copy of the resource, and both threads can continue.

2. Acquire returns both the resource and a special tag. If release takes a resource and the tag then different threads could not release each others held locks. This still means the wrong resource could be released by the thread holding the lock but a release would show up as an error as the tag would be invalidated on the first release.

3. The release expression takes the lock, the resource, a success expression, and a failure expression. This would require a more complicated typing rule for the release expression

The simplest of these choices was the first. Since there are reasons to pick all three choices, the simplest won out. A lock which blocks indefinitely on release causes deadlock, which is unwanted but safe behavior. This choice does not violate the progress property because it allows a release expression to continually progress to itself. The decision was largely arbitrary as all three options were viable.

3.6 Soundness Proofs with Concurrency and Locks

The Soundness proofs for the abstract syntax with the addition of locks are shown in Appendix A. The addition of the R and Θ environments changed the proof

statements as follows:

Preservation: If $\Psi_e; \Theta; \Delta; \Gamma \vdash e : \tau$, $\Psi_{spare}, \Psi_e; \Theta; \Delta; \Gamma \vdash (R, M, e : \tau)$ and $(R, M, e) \rightarrow (R', M', e')$, then $\Psi'_e; \Theta'; \Delta; \Gamma \vdash e' : \tau$ and $\Psi_{spare}, \Psi'_e; \Theta'; \Delta; \Gamma \vdash (R', M', e' : \tau)$.

Progress: If (R, M, e) is closed and well-typed ($C \vdash (R, M, e : \tau)$ for some τ and $C = \Psi; \Theta; \varnothing; \varnothing$), then either e is a value or else there is some (R', M', e') so that $(R, M, e) \rightarrow (R', M', e')$.

Erasure: If (R, M, e) is closed and well-typed ($C \vdash (R, M, e : \tau)$ for some τ and $C = \Psi; \Theta; \varnothing; \varnothing$) then the following holds:

1. If $(R, M, e) \mapsto (R', M', e')$ then $\text{erase}((R, M, e)) \overset{?}{\mapsto} \text{erase}((R', M', e'))$.
2. If $\text{erase}((R, M, e)) \mapsto (L', d')$, then $(R, M, e) \overset{+}{\mapsto} (R', M', e')$ and $\text{erase}((R', M', e')) = (L', d')$.
3. If $\text{erase}(e)$ is a value, then $\text{erase}((R', M', e')) = (L', d')$ and $\text{erase}((R, M, e)) = \text{erase}((R, M, v))$.

The major change from the original soundness proofs discussed in Section 2.3 is the addition of locks and the resource environment (R and its type environment Θ). The resource environment is only affected by operations on locks. The new soundness proofs are shown in Appendix A.

4 Device Driver Abstractions

Since the study done by Chou et al. [4] found that drivers are the source of most operating system bugs, a safe Ethernet adapter is a starting point for a safe OS kernel. Ethernet adapters rely on interrupts, FIFO queues, and system state. The safety abstractions of a driver can be guaranteed using state and configuration types.

Ethernet adapters pass packets between a machine and its Ethernet cable. The packets are held in two FIFO queues: one to hold packets before they are transmitted onto the network, and one to hold packets as they are received from the network. Depending on whether the adapter uses direct memory access or programmed IO (PIO), the FIFO queues may be located in kernel memory or on the device itself. Adapters also have registers to hold status information. The adapter communicates with the operating system through a driver.

This section explains the safety abstractions of a programmed IO Ethernet adapter and then shows the formal types I use to guarantee the abstractions. Given some basic starting assumptions (e.g. the driver is passed the correct data structure) an Ethernet driver that implements the abstractions is not able to violate memory safety or perform unsafe IO operations on the adapter. Section 5 describes my implementation of the locks of the abstract machine and the formal types needed by an adapter to make sure my 3Com Etherlink III 3c509 Ethernet driver implementation handles the safety invariants of both the shared data structures and the adapter.

4.0.1 The 3c509 EtherLink III NIC adapter

The 3c509 network card is an example of an Ethernet adapter that uses programmed IO. The FIFO queues are located on the adapter. The processor sends and receives from the FIFOs a word at a time. The driver must take care not to overflow the transmit FIFO by pushing more words than will fit or underflow the receive FIFO by popping non-existent words. The 3Com Etherlink III technical manual [5] de-

Port offset	Write function: High byte	Write function: Low byte	Read function: High byte	Read function: Low byte
0E	Command		Window 0-7	Status
0C	EEPROM data		EEPROM data	
0A	EEPROM command		EEPROM command	
08	Resource configuration		Resource configuration	
06	Address Configuration		Address Configuration	
04	Configuration Control		Configuration Control	
02			Product ID	
00			Manufacturer ID	

Figure 21: 3c509 window 0 registers [5]

scribes the abstractions a 3c509 driver is expected to support. These abstractions are assembled into a collection of invariants and rules on the interaction between the the 3c509 adapter and the operating system.

Accessing the Adapter

The 3c509 adapter is accessed through IO reads and writes to its registers which are separated into seven 8-word windows. Each window contains a related group of registers. One register can be accessed in any window. Writes to this register are commands issued to the adapter. Reads access the status of the adapter. The other registers require the adapter to be their specific windows. Each register has a set of invariants it expects are true when it is accessed. For example, some registers can only be read not written and others require statistics to be turned off, etc. A safe 3c509 adapter needs to guarantee the set of access invariants for each register.

Window 0 registers

Window 0 is used to initialize the adapter. Adapter interrupts are disabled when window 0 is the current register window. This window contains the EEPROM data/command registers which access stored adapter information such as the de-

Port offset	Write function: High byte	Write function: Low byte	Read function: High byte	Read function: Low byte
0E	Command		Window 0-7	Status
0C			Free transmit bytes	
0A	TX Status		TX Status	Timer
08			RX Status	
06				
04				
02	TX PIO data write		RX PIO data read	
00	TX PIO data write		RX PIO data read	

Figure 22: 3c509 window 1 registers [5]

fault IO address and available cable connections. Only the EEPROM can be accessed before the adapter is initialized. The other registers in this window mirror the EEPROM configurations and can only be accessed after initialization. Each of these register is 16 bits in length. The registers for product and manufacturing ID can only be read. All of the other window 0 registers can be written as well as read.

Window 1 registers

Window 1 is the standard operating window. The registers in this window include transmitter and receiver status, free bytes in the transmitter FIFO queue rounded down to a double word, and the PIO data registers which read from and write to the FIFO queues. The transmitter status register and PIO data registers may be read or written. The other registers in this window may only be read. As seen in Figure 22, only 8 bits can be read from the TX status or timer registers.

The PIO register IO operations can transmit and receive one byte, one word or two words at a time. The packet must be padded to a double word before transmitting or receiving. Any buffers on the operating system side need extra space for this padding.

The RX status register contains the length of the head received packet in the

RX FIFO. It is unsafe to read more than this length from the RX FIFO. The length shown in the RX status register in incremented as packet bytes are received and does not reflect how much of the packet has been read by the driver. For safety, the driver should check packet status after the whole packet has been received and the adapter status register shows an RX complete interrupt. Then the driver needs to read the packet and issue an RX discard command to go to the next RX packet's status.

It is unsafe to write to the TX status register unless the TX complete interrupt is triggered. Other writes risk making packet status inconsistent. The free transmit bytes register takes an extra IO cycle to update after packet transmission is completed. A safe driver will make some other IO operation between transmitting a packet and reading the free transmit bytes register.

Window 2 registers

Window 2 holds the station address. It must be read from the EEPROM and written here before the receiver is enabled.

Window 3 registers

Window 3 manages the transmit and receive FIFO queues. The actual number of free bytes in each queue is available here. As with free transmit bytes, free receive bytes should not be used to calculate RX packet length because of the overhead involved.

Window 4 registers

Window 4 contains diagnostic registers and the available controller types (coax, twisted pair, or a combination).

Window 5 registers

Window 5 contains current masks and thresholds. These registers are set through commands and do not need to be accessed.

Window 6 registers

Window 6 holds the statistics registers. Statistics must be disabled before these are read.

The command register

The command register accepts 16 bit IO writes. Bits 11-15 of a command are the command itself. Bits 0-10 hold arguments to the command or 0 if there are no arguments. If a command takes longer than one I/O cycle to complete, the driver must poll the adapter status register for a non-busy signal before continuing with further IO operations. Each command is listed below with a short explanation and its list of calling invariants.

Global Reset Resets the adapter and clears the FIFO queues. Requires a 1ms delay where no other IO operations are allowed.

Select Window Selects the window given as the argument. Windows 0-7 are legal even though there is no window 7.

Start Coax Turns on the coax transceiver. This command requires an 800us delay.

RX Disable Disables packet receiving.

RX Enable Enables packet receiving.

RX Reset Disables the receiver and empties the receiving FIFO queue.

RX Discard Discards the head packet in the receiving FIFO queue and its status. This command makes the adapter busy for an indeterminate length of time and the driver must poll the status register for a non-busy signal before proceeding.

TX Enable Enables the transmitter.

TX Disable Disables the transmitter.

TX Reset Disables the transmitter, empties TX FIFO queue, resets the available TX space, and restarts the transmitter.

Request Interrupt Requests the adapter to set an interrupt.

Acknowledge Interrupt Acknowledges and clears specific interrupt triggers. The possible interrupts are:

- Interrupt Latch - The actual interrupt.
- Adapter Failure - This occurs with any errors not shown by other interrupt triggers. The adapter will not proceed until a global reset command is issued. This interrupt is triggered by safety violations but is not, itself, unsafe.
- TX Complete - Packet transmission has completed.
- TX Available - There is room in the TX FIFO for a requested number of packet bytes.
- RX Complete - A packet has been completely received.
- RX Early - The first part of a packet has arrived.
- Interrupt Requested - An interrupt was requested.
- Update Statistics - The statistics are full.

Set Interrupt Mask Chooses which interrupt triggers to allow. The masked triggers can still be read but won't generate an interrupt on their own.

Set Read Zero Mask Masks interrupt triggers so that their status register bit is always a 0.

Set RX Filter Sets the receiving filter to individual, multicast, broadcast, or all.

Set RX Early Threshold Sets the amount of a packet that should be received before a RX early interrupt is generated. The threshold can be any multiple of

4 between 0 and 1792. If the threshold is greater than 1792, RX early is disabled. The adapter truncates arguments to a 2-word multiple. If the threshold is set to less than 60, there may be packet collisions. Therefore, any threshold value greater that or equal to 60 and less than or equal to 1792 is safe.

Set TX Available Threshold Sets the number of free bytes in the TX FIFO queue needed to transmit. The threshold can be any multiple of 4 between 0 and 2,044. If the threshold is greater than 1792, TX available is disabled.

Set TX Start Threshold Sets the number bytes in TX FIFO queue before the packet starts to transmit. The threshold can be a multiple of 4 from 0 to 1792. The adapter truncates arguments to a 2-word multiple. If the threshold is greater than 1792, early transmission is disabled and the packet transmits when all of it has reached the TX FIFO queue. Therefore, any threshold value less than or equal to 1792 is safe.

Statistics Enable Enables statistics.

Statistics Disable Disables statistics. Statistics must be disabled before reading them. The adapter still collects statistics while they are disabled; leaving them disabled for very long will cause some statistics to be incorrectly updated.

Stop Coax Stops the coax transceiver. This command requires a 800us delay.

These commands affect several adapter states. Global Reset, Start Coax, and Stop Coax require a known waiting period before further commands can be issued. The Select Window command changes the current register window. Statistics Enable and Disable enable or disable statistics. RX Reset, TX Reset, Global Reset, and RX Discard affect the amount of free space in the FIFO queues. RX Discard makes the adapter busy for an unspecified amount of time and requires polling the status register for a non-busy signal before any further IO operations may occur. The driver needs to correctly update state-type values for these states when commands

are issued.

The status register

The 16 bit status register allows the driver to access several adapter state conditions. This is the only register that can be safely accessed when the adapter is busy. The status bits include the interrupts, the bit which signals if the adapter is busy, and bits for the current window.

The Packet Receiver

When an RX Complete interrupt is triggered, a full packet is waiting in the receive FIFO. Its length and any errors are shown by the RX status register in window 1. If there were errors the driver must issue an RX Discard command. Otherwise, the length needs to be padded up to a double word before the packet is read from the RX PIO data register. After receiving the packet, an RX Discard command needs to be issued to remove the packet from the receive FIFO and update the RX status register.

The Packet Transmitter

Packet transmission requires three steps. First a packet information word is written to the TX PIO data register. The word contains the actual packet length, whether an interrupt is wanted after complete transmission, and the CRC which determines who pads the packet (driver or adapter). The CRC bit is usually 0 so the adapter does any necessary padding. In the second step, a word equal to 0 is written to the TX PIO data register. In the third step, the packet is written to the TX PIO data register a double word at a time. The packet must still be rounded up to a the nearest double word before it is sent even though the adapter handles the actual padding values.

4.0.2 Driver Functionality

The driver consists of setup, start, transmit, receive, stop, statistics, and multicast functions. The receive function is interrupt driven. The driver shares a device data structure with the operating system. I added a lock bit to the end of this data structure. Each function called by the operating system acquires the lock on the data structure before proceeding.

The setup function locates an inactivated 3c509 adapter if there is one and activates it. Setup uses contention select to pick an adapter if more than one 3c509 is present. The setup function allocates space for a device data structure if the operating system has not already allocated space for it. It then actives the adapter and initializes it with the base IO address and the interrupt request line. Setup initializes the fields of the device data structure and puts pointers to the start, transmit, stop, statistics and multicast functions into the structure. This function leaves the adapter in window 1, the normal operating window.

The start function takes the adapter from activated to enabled. It first resets transmitter and receiver FIFO queues. This clears the queues and their status and free space registers. The start function then calls an interrupt setup function with the irq and a pointer to the interrupt handling function. It sends an activation IO call to the adapter in case it has been de-activated and then starts the coax transceiver or twisted pair. After that it clear all of the adapter statistics and enables them. It enables the transmitter and receiver and finally resets the wanted interrupt triggers and clears any leftover interrupts. At this point, the adapter is ready to be used.

When a incoming packet has completely reached the adapter, it generates an interrupt which is passed to the driver's interrupt function. The interrupt function determines if the interrupt heralds a received packet or an error and either calls the receive function or updates a statistics data structure.

The receive function reads in packets from the receive FIFO on the adapter. It

checks the length of the head packet in the RX FIFO queue. If the RX Status register indicates that the packet was received correctly, the receive function creates a buffer to hold the packet and reads it into the buffer 2 words at a time from the RX PIO data register. When the packet is read, an RX discard command is issued to the adapter. This removes the head packet form the RX FIFO and update the RX Status register to reflect the status of the next packet. The receive function handles packets until the queue is empty of completed packets. Incorrectly received packets are discarded.

The stop function disables the adapter without deactivating it. It turns off adapter statistics, disables the transmitter and receiver, turns off the coax or twisted pair transceiver and removes the irq from the operating system's collection of interrupts. After the stop function is called, the adapter is not usable until the start function is called again.

The multicast function allows the operating system to update the type of packets it wants to receive without disabling the adapter. The operating system chooses a combination of direct, multicast, broadcast and promiscuous modes. Multicast implies broadcast and promiscuous implies all three other modes.

The statistics function updates a statistics data structure and returns a pointer to it. To do this, it disables adapter statistics, changes to window 6, reads all statistics and updates the data structure, and then re-enables adapter statistics and changes back to window 1.

The driver also includes various support functions. These include a transmitter timeout function, a statistics updating function and EEPROM reading functions. The timeout function is called by the transmit function if the adapter has not had room in its TX FIFO for a new packet for too long. Timeout clears the interrupts and resets the transmitter. This discards all packets waiting in the TX FIFO and restarts the transmitter. The statistics update function reads each adapter statistic and

updates the relevant OS statistic in the statistics data structure. It disables adapter statistics before reading them and enables them again afterwords. The EEPROM functions allow the driver to read the EEPROM before and after activation. Both send the desired EEPROM data location to the EEPROM command register, wait a required amount of time, and then read the data from the EEPROM data register.

4.1 Operations on a PIO Ethernet Adapter

An Ethernet adapter supports a limited set of operations which allow it to initialize, send and receive packets, generate interrupts, track adapter/queue status, and collect statistics. An Ethernet adapter is accessed through IO operations on its registers which are separated into different windows. The FIFO queues for a PIO adapter are located on the device and are also accessed through IO operations. A type-safe Ethernet driver has a safe interface for each operation.

The initialization operations only access the set of IO addresses the device might listen on. When the adapter is initialized, a base IO address is picked and all future IO operations use that base address. The operations can be grouped into a few categories based on their safety requirements.

The first category of IO operations are oprtaions on the FIFO queues. Operations on the send and receive FIFO queues are only done when the adapter is in an initialized and ready state. The read and write operations access data in allowed increments and the overall amount of data accessed is padded to a specified boundary. The driver only reads data that is actually in the receive queue. During the read operation, the adapter is in the correct window for the receive FIFO and isn't busy. This data is read into a waiting buffer that is large enough to hold it. Likewise, the driver only writes data from an existing buffer into the send queue when the amount of data does not overflow the buffer or queue. This operation also requires a specific window and an non-busy adapter. These are the only safe operations that can be

done on a PIO adapter FIFO queue.

The second category of IO operations are the status operations. The data read by status operations includes triggered interrupts, free bytes in the send queue, amount of data available in the receive queue, and the current window. These operations read an allowed amount from a register in the correct window. Some status operations are allowed when the adapter is busy. (This lets the driver poll to see when the device is no longer busy). Some status registers allow writes to update their information. The act of writing triggers the adapter to update these status registers.

The third category of IO operations are commands to the adapter. Commands are IO writes of a specific size (e.g. 16 bits) to one command register. Commands might be delayed or ignored if issued while the adapter is busy so the adapter needs to be non-busy unless it is known that delayed/dropped commands cannot cause safety violations. Some commands cause the adapter to become busy and for safety, the driver waits until the adapter is free again.

The final category of IO operations are the general read and write operations. These test and set the adapter settings. These operations only take place when the adapter is in the correct window. Some operations come in read/write pairs and some are only a read or only a write. Each general operation accesses at most its maximum number of allowed bytes (generally 1,2, or 4). Some operations are only performed after adapter statistics are turned off. Like the commands, the general operations only occur when the device is non-busy.

The operations described above are the only operations in a safe PIO Ethernet driver. To formalize an interface for these operations, I first extend some type concepts from Section 3 and then create types for the IO operation interfaces.

4.2 Extending Types to Support Operations

This section focuses on extending type concepts to provide more support for OS and device abstractions. Starting from $\text{Mem}(I, \tau)$ I create types to enforce abstractions on collections of memory, state in an operating system, and the IO interface to Ethernet adapters.

4.2.1 Collections of Memory

The $\text{Mem}(I, \tau)$ type is used to track the state of a memory word. A $\text{Mem}(I, \tau)$ can therefore be called a memory state type. The concept of memory state types can be extended to track the state of a collection of memory words.

A type for such a collection of memory words can be built from the existing types in the abstract syntax: $Pair(I, X, Y) =^{\text{lin}} \langle \text{Mem}(I, X), \text{Mem}(I+1, Y) \rangle :^{\text{lin}}_{0}$. The *Pair* type is the memory state-type for a pair of consecutively data.

4.2.2 State Types

Operating systems keep track of a lot more state than just the type of memory words. For example, an operating system might need to track whether interrupts are on or off, or if the Ethernet adapter is up and running. I extended the concept of "memory state types" to "state types" which are linear size 0 and track states throughout the operating system. The generalized state types are still linear and size 0 so they take up no space in memory and have no associated run time cost.

As an example, the abstract syntax for an interrupt state type could be $intr : Intr(I) :^{\text{lin}}_{0}$ where I is 0 or 1 depending on whether interrupts are turned on or off. If a single copy of *intr* exists in the operating system, it can be passed around and used to track the current state of interrupts. The enforcer expressions associated with it are those expressions which require interrupts to be on or off or which enable or disable interrupts. As Mem enforces the abstraction of linearity on a word of memory, $Intr$

enforces abstractions on the sections of OS code concerned with interrupts.

The abstractions of safe IO operations can be tracked by state types. Given the IO operation safety criteria, the driver needs to use state types for:

- The current window
- The base IO address
- Adapter statistics status
- Adapter busy status
- TX FIFO free space
- RX FIFO used space

These six states track all of the critical information in an Ethernet adapter. Writes to and from a buffer and the PIO TX and RX registers additionally use a buffer state-type to ensure that the driver does not overflow or underflow an operating system buffer.

Each state needed by the driver is expressed as a formal state type:

$$IOAddressState(i_{\text{Adapter}}, i_{\text{Address}}) :^{\text{lin}} 0$$

$$WindowState(i_{\text{Adapter}}, i_{\text{Win}}) :^{\text{lin}} 0$$

$$BusyState(i_{\text{Adapter}}, i_{\text{Busy}}) :^{\text{lin}} 0$$

$$StatisticsState(i_{\text{Adapter}}, i_{\text{Stats}}) :^{\text{lin}} 0$$

$$QueueState(i_{\text{Adapter}}, i_{\text{Queue}}, i_{\text{Space}}) :^{\text{lin}} 0$$

All of these states are tied to a specific adapter in case more than one network card is present. The $IOAddressState$ tracks the current base IO address. The state prevents IO operations outside the legal range of addresses. $WindowState$ tracks the currently used register window. The state ensures that register are only

accessed while the adapter is in the correct window. *BusyState* indicates whether the adapter is busy. Only the adapter status register is read when this state shows that the adapter is busy. *StatisticsState* tracks whether adapter statistics are currently turned on or off. This state needs to indicate off before statistics can be read from window 6. The driver needs two values of type *QueueState* to track the amount of known free space in the TX FIFO and the amount of known used space in the RX FIFO. As long as the known amounts are conservative (there may be more actual free space in the TX FIFO), this state prevents overflow and underflow of the queues. Together these state track all safety concerns of the adapter.

One additional state type is needed to guarantee safety on the operating system side. Packets in the operating system are stored in buffers. The driver needs to make sure it does not overflow or underflow these buffers:

$$\mathit{BufferState}(i_{\text{HeadAddress}}, i_{\text{DataStart}}, i_{\text{DataEnd}}, i_{\text{TailAddress}}) :^{\text{lin}} 0$$

The *BufferState* type shows the current address boundaries for a buffer. This buffer follows the convention for an SK buffer in a Linux adapter driver. $HeadAddress$ points the the first byte in the buffer. *DataStart* points to the first byte of actual data. Unneeded packet headers may fill the space between $HeadAddress$ and *DataStart*. *DataEnd* points to the byte after the end of the data. Similarly, $TailAddress$ points the the byte after the end of the buffer. Overall, $i_{\text{HeadAddress}} \leq i_{\text{DataStart}} \leq i_{\text{DataEnd}} \leq i_{\text{TailAddress}}$. If this relationship is preserved, the buffer cannot overflow or underflow.

4.2.3 Configuration Types

Operating system abstractions have sets of static rules which describe the allowed relationships between state-type variables. These configuration rules describe the

values that a set of state-type variables must have in order to perform an action, such as sending a packet to an Ethernet adapter. I further extended the concept of state types to configuration types which are constant. Since configuration type variables are constant, they do not need to be linear but they are still size 0.

An IO function specification that allows an Ethernet adapter to communicate with its driver needs to ensure that several different states are in agreement. For example, a safe IO call to send a word of memory to a specific port on the device might need to know the following:

1. The register is legal.
2. The register can be written.
3. The device is either not busy or it doesn't matter.

The third need can be expressed as a state type: $DeviceBusy(I_{Device}, I_{BusyState}) :^{\text{lin}} 0$. which expresses whether the device with identifier I_{Device} is busy or not. The identifier is a singleton integer unique to the device. $I_{BusyState}$ is 0 or 1 depending on the state of the adapter. A type that can relate different state types is needed to package all three safety needs together. This type specifies the different acceptable configurations of state types and is named a configuration type.

Three basic configuration types are needed to enforce the abstractions on the base IO address, commands written to the adapter, and general register reads and writes. IO packet data read and write operations and status read and write operations are possible on such a limited set of ports that it makes sense to write individual function interfaces for these operations. The remaining IO operations use the configuration types described below to share a smaller set of function interfaces.

$$IOAddressConfig(i_{\text{Bus}}, i_{\text{LowAddress}}, i_{\text{HighAddress}}) :^{\text{non}} 0$$

The $IOAddressConfig$ type states the acceptable base IO address range for a

given bus. In the case of the ISA bus, this range is 0x200 to 0x3E0, inclusive. This configuration type is used to set the base IO address to a legal value.

$$CommandConfig(i_{\text{Command}}, i_{\text{TakesArguements}}, i_{\text{WaitTime}}, i_{\text{ChangesStats}}, i_{\text{ChangesBusy}}) : \overset{\text{non}}{0}$$

The $CommandConfig$ type lists the constraints on writes to the command register. The first parameter is the command itself. This allows an IO expression to match a $CommandConfig$ type value to a singleton integer command. The second parameter indicates whether the command takes arguments. Arguments are bit masks and are added to the command before it is sent to the adapter. The IO expression checks that the arguments, if any, fit into the given 11 argument bits. The wait time indicates the time in μ seconds that the driver must wait after issuing the command. This allows the driver to handle commands which take more than one IO cycle to complete. An IO expression matches this parameter to a singleton integer time. The fourth parameter indicates if the command causes adapter statistics to turn on or off. Three values (0,1,2) could determine if it turns statistics on, off, or has no affect on statistics. Commands which change statistics cause a statistics state type to change. The last parameter indicates if the command may be issued while the adapter is busy. Some commands such as RX Discard take an unknown amount of time to complete and cause a busy state type to change.

$$RegisterConfig(i_{\text{Window}}, i_{\text{Bytes}}, i_{\text{Read}}, i_{\text{Write}}, i_{\text{IsRangedPort}}, i_{\text{LowPort}}, i_{\text{HighPort}}, i_{\text{StatsOnOk}}, i_{\text{AdapterBusyOk}}, i_{\text{ChangeQueue}}) : \overset{\text{non}}{0}$$

The $RegisterConfig$ state type list the configuration of states and values necessary to access a particular register. The first parameter is the window of the register. The second parameter is the number of bytes that are allowed to be accessed at once.

If bytes is 4, as in the case of the TX PIO data register, the register may be accessed by the byte, word, or double word. The read and write parameters indicate which type of access is allowed. Some registers allow both reads and writes so it was simpler to encode this as two parameters. The fifth parameter indicates whether the port actually occupies a range of registers. The station address is an example of a ranged port. A ranged port allows one $RegisterConfig$ to control access to the six consecutive station address ports. The lowest and highest ports of a ranged port are the next two parameters. If the port is not ranged, these should both be the actual port. The stats parameter indicates whether adapter statistics may be turned on when this register is accessed. The statistics registers require that adapter statistics be turned off. The busy parameter indicates if the adapter is allowed to be busy. Only the status register may be accessed when the device is busy. The last parameter indicates whether accessing the register changes the free space in a FIFO queue. For example, writes to the TX PIO data register decrease the free space in the TX FIFO queue. Together, these parameters list all the configuration rules related to accessing a register.

4.2.4 A Typed Function Specification

The formal interface for a PIO FIFO data read uses most of the state and configuration types. Data is read in repeated accesses a few bytes at a time. The PIO data

read operation takes care of the repetition.

$$
\begin{array}{l}
(T\_PIO\_data\_read) \\
\\
C_1 \vdash i_{\text{BaseAddress}} : Int(I_{\text{BaseAddress}}) \\
i_{BufferAddress} : Int(I_{BufferAddress}) \quad i_{\text{DataSize}} : Int(I_{\text{DataSize}}) \\
RegisterConfig(I_{\text{Window}}, I_{\text{Bytes}}, I_{\text{Read}}, I_{\text{Write}}, \\
I_{\text{IsRangedPort}}, I_{\text{LowPort}}, I_{\text{HighPort}}, \\
I_{\text{StatsOnOk}}, I_{\text{AdapterBusyNotOk}}, I_{\text{ChangeQueue}}) \\
C_2 \vdash e_{\text{buffer\_state}} : BufferState(I_{\text{HeadAddress}}, I_{\text{DataStart}}, I_{\text{DataEnd}}, I_{\text{TailAddress}}) \\
C_3 \vdash e_{\text{address\_state}} : IOAddressState(I_{\text{Adapter}}, I_{\text{BaseAddress}}) \\
C_4 \vdash e_{\text{window\_state}} : WindowState(I_{\text{Adapter}}, I_{\text{Window}}) \\
C_5 \vdash e_{\text{busy\_state}} : BusyState(I_{\text{Adapter}}, I_{\text{NotBusy}}) \\
C_6 \vdash e_{\text{queue\_state}} : QueueState(I_{\text{Adapter}}, I_{\text{RXQueue}}, I_{\text{UsedSpace}}) \\
\text{where } I_{\text{DataStart}} <= I_{BufferAddress},\ I_{BufferAddress} <= I_{\text{TailAddress}}, \\
I_{\text{DataSize}} > 0,\ I_{BufferAddress} + I_{\text{DataSize}} + 3 <= I_{\text{DataEnd}}, (3 \text{ for padding}), \\
I_{\text{UsedSpace}} \geq I_{\text{DataSize}},\ I_{\text{Read}} = 1 \text{ and } I_{\text{Write}} = 0 \\
\hline
C_1, C_2, C_3, C_4, C_5, C_6, \vdash PIO\_data\_read(i_{\text{BaseAddress}}, i_{BufferAddress}, i_{\text{DataSize}}, \\
e_{\text{buffer\_state}}, e_{\text{address\_state}}, e_{\text{window\_state}}, e_{\text{busy\_state}}, e_{\text{queue\_state}}) \\
: \langle BufferState(I_{\text{HeadAddress}}, I_{\text{DataStart}}, I_{\text{DataEnd}}, I_{\text{TailAddress}}), \\
IOAddressState(I_{\text{Adapter}}, I_{\text{BaseAddress}}), \\
WindowState(I_{\text{Adapter}}, I_{\text{Window}}), \\
BusyState(I_{\text{Adapter}}, I_{\text{NotBusy}}), \\
QueueState(I_{\text{Adapter}}, I_{\text{RXQueue}}, I_{\text{UsedSpace}} - I_{\text{DataSize}}) \rangle
\end{array}
$$

This IO operation only reads an existing number of bytes from the RX FIFO queue and places the bytes into an existing buffer that is large enough to hold them. The interface guarantees that the window is correct and the adapter is not busy.

(This port is not ranged, statistics do not matter, and read does change the used space in the RX FIFO queue.)

The various state-type values are set by IO status operations and by values in the shared data structure of the driver. The adapter adapter status register contains information on whether the device is currently busy and the data sturcture has a field for the IO address. Function interfaces for the status read and IO address read set the state types to their correct values.

5 Implementation

I implemented locks and a safe 3Com Etherlink III 3c509 driver in Clay. The driver depends on locks to safely access a shared data structure. Using an OS starter kit, the OSKit[9], I implemented a small multi-threaded operating system, Shale, in C. The driver compiles to C++ and then to an object file, which is linked with the C code. My 3c509 driver is guaranteed to only make safe IO calls and safe buffer reads/writes. It is able to send and receive packets. This section discusses my lock implementation, my 3c509 driver implementation, and the small operating system. The next section discusses benchmarking my driver.

5.1 Locks

This implementation of locks is written in Clay's syntax and matches the abstract syntax mentioned in Appendix A.

Locks are built from three basic types.

```
type Int[int I] = native
@type0 Mem[int I, int A] = native
typedef Ptr[int I, int T] = @[Int[I], Mem[I,T]]
```

$Int[I]$ is a singleton integer and a pointer to memory location I. Mem is the memory State Type which tracks a word of linear memory. $Mem[I,T]$ means that memory location I contains a value of type $Int[T]$. $Ptr[I,T]$ is a linear pair of a pointer to linear memory and its associated Mem. The pointer cannot be used without its Mem so it is a convenience to handle them together.

Locks come in two related types:

```
type0 Lock0[int Bit, @type0 Resource] = native
typedef Lock[int Bit, @type0 Resource] =
    .[Lock0[Bit, Resource], Int[Bit]]
```

The first type is the lock itself. This type is nonlinear and size 0 so that copies of it can be passed around to all code sections which might need the lock; no extra memory will be needed for it at run time. The second type is a pair of the lock and a pointer to the lock bit. The "." before the pair brackets ([ ]) states that the pair is nonlinear. Since all accesses to the lock will require the lock bit, it makes sense to bundle them together. Keeping the size 0 lock type separate allows flexibility in declaring the location of the lock bit. The bit location could be separately declared or could be in a chosen place in a locked data structure (e.g. the first bit). The type checker verifies that the lock bit pointer matches the *Lock*0 it's bundled with.

Two functions are used to create locks:

```
native Lock0[B,R] create_lock0 [@type0 R, int B]
  (R resource, Mem[B,0] bit) limited[0];

Lock[B,R] create_lock [@type0 R, int B] (R resource, Ptr[B,0] bit_p)
{
  let (bit_addr, bit_mem) = bit_p;
  let lock0 = create_lock0(resource, bit_mem);
  return .(lock0, bit_addr);
}
```

The first function takes the resource state type value, the lock bit pointer, and its memory state type value and returns a *Lock* for the resource. The memory state type and resource state type values are not returned, thus preventing access to the resource until the lock is acquired. This function calls a limited native function which takes the memory state type and resource state type values and coerces them to a *Lock*0 value. The native *create_lock*0 function is limited which means it handles only size 0 types and has no need of a matching C function. The *Lock*0 value gets bundled with the lock bit pointer and returned as the *Lock*. In this implementation,

the lock bit is allocated and initialized to 0 before the create function is called. The type checker will verify that the lock bit value is actually 0.

The function to acquire a lock is native. It takes a *Lock* (a pair of a *Lock*0 and its lock bit pointer) and goes to C to acquire the lock:

```
native R acquire [int B, @type0 R] (Lock[B,R] lock);
native {
  // test_and_set_bit is atomic x86 assembly btsl on bit 0
  inline void acquire(unsigned long lock) {
    while (test_and_set_bit(0, (void *) lock)==1);
  }
}
```

The C *acquire* function uses an atomic swap operation to block until the lock bit value is 0 and then sets it to 1 claiming the lock. The Clay function header coerces the resource state type out of the *Lock* and returns it.

The release function is also native. It takes a *Lock* and its resource state type value and goes to C to release the lock:

```
native void release [int B, @type0 R] (Lock[B,R] lock, R resource);
native {
  // test_and_clear_bit is atomic x86 assembly btcl on bit 0
  inline void release(unsigned long lock){
    while (test_and_clear_bit(0, (void *) lock)==0);
  }
}
```

The C *release* function uses an atomic swap operation to block until the lock bit value is 1 and then sets it to 0 releasing the lock. The resource state type value is not returned and must once again be acquired to use it.

Locks in action

To demonstrate the Clay lock implementation, a collection of memory state types is formed and locked. Unsafe operations on this lock are caught by the Clay compiler. The memory to be locked is passed in as array of consecutive Mem. The array type follows:

```
@type0 LArrayS[int I, int J, @type0<-(int) F] = struct
{
  F[I] data;
  LArray[I+1,J,F] next;
}
typedef LArray[int I, int J, @type0<-(int) F]
  = LIf[I!=J,LArrayS[I,J,F]]
```

$LArrayS$ is a structure containing linear size 0 data and an array of more data. $LArray$ is an array of linear size 0 data with the same basic type. An array of type $LArray[I, J, F]$ contains linear type 0 data with the indices I through $J-1$. F is a function that maps an index to its data. LIf is a union type that contains a linear size 0 type if a given condition is satisfied. In this case the LIf contains the next $LArrayS$ structure as long as there is another one in the array ($I \neq J$).

A stack can be represented as a linear array of Mems (assuming you want only 32-bit integers). Such a stack has the type

$$LArray[0,\ Max,\ fun[int\ I]\ exists[int\ A]\ Mem[Base + (4 * I),\ A]$$

where the stack addresses run from $base : Int[Base]$ to $max : Int[Base\ +\ 4 * (Max - 1)]$. Mems can be allocated from the stack using an allocation function. The recursive type fun needed to create an $LArray$ is described in [14] but not presented in the abstract machine.

```
@[Ptr[Base+4*I,N], Int[Base+4*(I+1)],
  LArray[I+1,J,fun[int I] exists[u32 A] Mem[Base+4*I,A]]]
alloc[u32 I, u32 J, u32 Base, u32 N, I<J]
(LArray[I,J,fun[int I] exists[u32 A] Mem[Base+4*I,A]] stack,
 Int[Base+4*I] ptr, Int[N] value);
```

The function is not relevant to this example and calls several other Clay functions so only its header is shown here. *alloc* takes a stack of linear memory, a pointer to the next free memory location, and an initial value. The function returns a pointer to the allocated and initialized memory (a pair of its $Mem[Base + 4 * I, N]$ and its address $Int[Base + 4 * I]$) as well as the updated *next* pointer and the, now smaller, stack.

The collection of memory that is locked below is a consecutively stored pair of integers that add to 100. The memory state type which tracks this pair is show here:

```
typedef AddedPair[int I] =
    exists [s32 N, s32 M; M==100-N] @[Mem[I,N],Mem[I+4, M]]
AddedPair[I] create_addedpair [int I, s32 A, s32 B; B==100-A]
    (Mem[I,A] x_mem, Mem[I+4,B] y_mem)
{
  return @(x_mem, y_mem);
}
```

Locking an *AddedPair* state-type prevents access to the pair of integers it tracks.

Given a stack of linear memory and a pointer to its base:

```
LArray[0,200,fun[int I] exists[int A] Mem[Base+4*I,A]] stack
Int[Base] base
```

We can dynamically allocate some linear integers and a lock bit (also a linear integer) from the stack and create an *AddedPair*:

```
let (x_ptr,   base  stack) = alloc(stack, base, 4);
```

```
let (y_ptr,   base, stack) = alloc(stack, base, 96);
let (bit_ptr, base, stack) = alloc(stack, base, 0);
let (x_addr, x_mem) = x_ptr;
let (y_addr, y_mem) = y_ptr;
let resource = create_addedpair(x_mem, y_mem);
```

x_ptr, *y_ptr*, and *bit_ptr* are allocated from the linear memory stack and initialized to the passed in values. They are all of type *Ptr* meaning they are a linear pair of a pointer and its *Mem*. The fourth and fifth lines of code separate each *Ptr* into its component address and *Mem*. This invalidates each original linear *Ptr*. The resource is then created from the two memory state types, invalidating them. Invalidating used linears makes the following code mistakes impossible:

```
let resource2 = create_addedpair(x_mem, y_mem); // error caught now
let resource2 = resource;                       // error caught later
```

The first mistake attempts to create a second *AddedPair* and this line fails the type checker because *x_mem* and *y_mem* were invalidated upon creating the first *AddedPair*. The second mistake attempts to make an alias to *resource*. The line of code succeeds but it invalidates the original *resource* so any future attempts to use *resource* with fail the type checker because *resource* is invalid. These caught errors show that Clay correctly handles linear values and does not allow them to be aliased or used after they become invalid.

The following code locks the *resource*, and duplicates the nonlinear lock.

```
let lock = create_lock(resource, bit_ptr); // linear resource invalid
let lock2 = lock;                          // lock is still valid
let lock3 = lock;                          // lock = lock2 = lock3
```

The *create_lock* function reads the linear *resource* and the *Mem* component of the *bit_ptr* and does not return them. This invalidates the *resource* and the lock bit *Mem* so only the lock has control over *resource* and only the lock can load or store

the lock bit. If we attempted to read *resource* after this, the type checker would catch it:

```
let [ ] (x_mem,y_mem) = resource;          // read resource..fails
```

This mistake attempts to access the locked, but not acquired, *resource*. The type checker catches it because *resource* has become invalid.

The next section of code acquires the lock, separates *resource* into its components, and prints the values it was guarding.

```
let resource = acquire(lock);              // linear resource valid
let [ ] (x_mem,y_mem) = resource;          // linear resource invalid
let (x_mem, x_data) = load(x_addr, x_mem); // load value of x (4)
let (y_mem, y_data) = load(y_addr, y_mem); // load value of y (100-4)
print_int(x_data);                         // prints 4
print_int(y_data);                         // prints 100-4
```

The lock is now held so the resource can be accessed.

To release the lock, the *AddedPair* is re-formed and passed to the release function:

```
let resource = create_addedpair(x_mem, y_mem);// linear resource valid
let lock = release(lock, resource);           // then invalid
```

The lock is now unheld and *resource* is invalid.

If the values of x and y no longer added to 100 or an integer in a different memory location was substituted, the type checker will catch the error:

```
let y_mem = store(y_addr, y_mem,10);         // y = 10 (10 != -4+100)
let resource = create_addedpair(x_mem, y_mem); // bad addition..fails
let resource = create_addedpair(x_mem, z_mem); // bad location..fails
```

The type checker find errors when creating this added pair. It will also catch errors where the wrong integer is placed into the resource. The memory locations of x and z are not consecutive so the *create_addedpair* function fails to type check.

Attempts to re-release the lock will also fail because *resource* becomes invalid on the first call to *release*:

```
let lock = release(lock, resource);        // re-release..fails
```

The type checker catches this re-releasing error because the linear *resource* is invalid. If the lock is re-released using a different, but valid, *AddedPair*, the lock will block until another thread acquires it.

Because the lock is nonlinear, copies can be made and used interchangeably even though only one *resource* may ever be acquired at a time:

```
let resource = acquire(lock2);             // uses lock 2
let lock4 = release(lock3, resource);      // lock = lock2 = lock3...
```

Prevented lock problems

This lock implementation matches the locks in the abstract syntax. Of the four lock problems, the abstract syntax and implementation for locks provides compile time prevention of "forget to acquire". Additionally, locks may not be released without a matching resource which is a partial solution to "release unheld lock". The "re-entrant lock" problem could be solved through lock numbering and Clay is intended to catch dropped linear values at the end of a function[2] which would provide an implementation solution to the "forget to release" problem.

Minimizing trips to C++

All types and functions declared to be native must be defined in C++. The C++ code is not guaranteed to be safe so minimizing trips to C++ is important.

When the Clay lock code is compiled, it produces extern statements for all non-limited native functions. These functions must be defined in C++.

```
extern void acquire(unsigned long lock);
```

[2]Clay does not currently notice dropped linear values at the end of a function but this could be added in the future.

```
extern void release(unsigned long lock);
```

Both functions contain only 1 line of C++ code and the atomic functions they depend on contain 1 line of assembly code. The functions are short enough that inspection gives reasonable assurances of safety.

The following shows a compilation and run of the lock code example described above. For testing and demonstration purposes, print lines have been included which show the memory location of the lock bit and its value as the lock is acquired and released.

```
#prompt) clay lock.clay
Successful compilation! 3.885883 seconds.
#prompt) g++ lock.cc
#prompt) a.out
lock at 162988048
  lock before acquire = 0
  lock after acquire = 1
4
96
lock at 162988048
  lock before release = 1
  lock after release = 0
lock at 162988048
  lock before acquire = 0
  lock after acquire = 1
lock at 162988048
  lock before release = 1
  lock after release = 0
```

If the above were run without the C function printlines, the output would simply

print the values of x and y in the example. 4 and 96 are the values assigned to the pair of integers.

It would be possible to write the *acquire* and *release* functions in Clay by substituting the while loops with a special type of recursive for-loop that allows type parameters to change as the loop cycles. The calls to the atomic assembly instructions need to remain native since there is currently no Clay equivalent to C's assembly calls.

Allocating different sizes of memory

The lock example showed only 32-bit integers. The Mem type and *alloc* function could be easily changed to allow allocation of any memory size.

```
@type0 Mem1[int I, type T] = native
typedef Mem8[int I, type T] = @[Mem1[I,T], Mem1[I+1,T], Mem1[I+2,T],
                                Mem1[I+3,T], Mem1[I+4,T],
                                Mem1[I+5,T], Mem1[I+6,T], Mem1[I+7,T]]
typedef Mem16[int I, type T] = @[Mem8[I,T], Mem8[I+8,T]]
typedef Mem32[int I, type T] = @[Mem8[I,T], Mem8[I+8,T],
                                 Mem8[I+16,T], Mem8[I+24,T]]
```

These memory state-types track memory by the bit, the smallest unit available. In this implementation, Mem ($Mem32$) is a pair of consecutive smaller Mems. I could also have defined it as two $Mem16$s or 32 $Mem1$s.

A separate allocation function would be needed for each size of allocation. For example, the allocation function for unsigned 16-bit integers:

```
@[Ptr16[Base+I,N], Int[Base+I+16],
  LArray[I+16,J,fun[int I] exists[int A] Mem1[Base+I,A]]]
alloc16 [u32 Base, u32 I, u32 J, u16 N; I+16<J]
(LArray[I,J,fun[int I] exists[int A] Mem1[Base+I,A]] stack,
```

```
 Int[Base+I] ptr, Int[N] value);
```

This function allocates 16 1-bit *Mem*s and returns the *Ptr* to the allocated memory, and the updated stack and next pointer.

5.2 The 3c509 Adapter

This section discusses my implementation of the 3c509 driver. I talk about implementation choices I made based on an existing 3c509 driver and the architecture in my computer. The benchmark performance of the driver is discussed in Section 6.

5.2.1 Implementation Choices

My safe 3c509 driver is based on Donald Becker's 3c5x9 driver version 1.18 [1]. I ported the driver to Clay using the state and configuration types designed in Section 3. This driver handles several different 3Com network adapters and several bus types (EISA, ISA, and MCA). The x86 architecture running Shale includes the ISA bus and the 3c509 adapter so the safe driver only includes code for the 3c509 running on a ISA bus. Donald Becker's implementation differs slightly from the 3Com manual [5]. Since the 3c5x9 driver is already known to work, the safe driver mimics this driver rather than the exact steps outlined by the 3Com manual. This allows general speed comparisons between the 3c5x9 driver and the safe driver.

5.2.2 Clay Types

This section discusses some of the types used in my driver implementation. They are closely related to the state and configuration types mentioned in section 4. The configuration types include configurations for command and register IO operations as well as a configuration to set the base IO address.

```
type0 BaseConf [int Bus, int Low, int High] = native
type0 CmdConf  [int C, int Wait, int Stats, int Busy] = native
```

```
type0 PortConf [int Win, int Bytes, int Read, int Write,
                int Port] = native
type0 RangedPortConf [int W, int B, int R, int T, int LowP,
                      int HighP] = native
```

These *BaseConf* type states that a given bus allows base IO addresses between *Low* and *High*.

```
BaseConf[ISA_Base, 0x200, 0x3F0]       isa;
```

My 3c509 driver uses an ISA bus and sets the base IO address using the above configuration-type value.

Command operations come with an indication of the wait time they require and whether they change statistics or make the adapter busy.

```
CmdConf[StartCoax_CMD,800,0,0,0]              start_coax;
CmdConf[StatsEnable_CMD,0,StatsEnabled+1,0,0] stats_enable;
```

The above command configuration-types state that the start coax command needs an 800 micro-second wait and the Statistics Enable command changes statistics to enabled.

The configuration types for a register operation shows the necessary window, the number of bytes to access, and the port for the register.

```
PortConf[1,2,1,0,RxStatus_REG]    rx_status;
```

The RX status register is located in window 1, handles up to 2 word reads, and doesn't allow writes. *RxStatus_REG* is defined to be 0x08. For ease of use, I split the ranged port and single port configuration-type into two configuration types. Only 2 registers actually use a range of ports and this move shortened the type information checked at each IO interface for the other register operations.

The state types track the various states of the adapter.

```
@type0 BaseState   [int Device, int Base] = native
@type0 BusyState   [int Device, int Busy] = native
```

```
@type0 StatsState  [int Device, int Stats] = native
@type0 QState      [int Device, int Queue, int Space] = native
@type0 WinState    [int Device, int Win] = native
```

The driver uses one of each state type (and one state type for each FIFO queue).

```
@type0 DevStateS
[int This, int W, int B, int D, int S, int T, int R] = struct {
  WinState   [This,    W]                window;
  BaseState  [This,    B]                io_addr_base;
  BusyState  [This,    D]                busy;
  StatsState [This,    S]                statistics;
  QState [This, TX_FREE, T] tx_free;
  QState [This, RX_USED, R] rx_used;
}
```

When the driver enters a function (e.g. transmit), it does not know the current values of these states and must gather them before Clay will let it perform operations which depend on them. This is accomplished through reading a stored base address and a status register read operation. Together, they set the current window, base address, and adapter busy state. The *QState*s are set as needed by TX and RX status operations. The statistics state is set by enabling or disabling statistics.

Certain registers such as the status register require a unique IO interface so I didn't give them a port configuration-type and instead wrote an IO function which only uses their port offset.

```
native exists [u1 B2, u16 V, u3 W2]
@[Int[V], Int[W2], Int[B2], WinState[D,W2],
  BaseState[D,A], BusyState[D,B2]]
el3_inwStatus
[u32 D, u32 A, int B1, int W1]
```

```
(Int[A] addr,
 WinState[D,W1] win, BaseState[D,A] base, BusyState[D,B1] busy);
```

The above function header shows the interface to the status register. The only incoming requirement is that the base address is the stored one. The status register returns two bytes of data ($Int[V]$ where V is the complete status data). Among the data bits are the current window ($Int[W2]$, the window bits) and the busy state of the adapter ($Int[B2]$, the busy bit). The status function returns these and then updates state-type values so that further IO operations are possible.

5.2.3 Prevented Violations

My implementation of the 3c509 driver prevents several categories of errors assuming the function input invariants are not violated.

- Two driver functions will never hold the device data structure lock at once or interleave their reads and writes to the adapter.
- The driver will never read a register in the wrong window.
- The driver will not read or write more bytes than a register allows.
- The driver will read only the adapter status register while the adapter is busy.
- The driver will not read more data from the receive FIFO than it contains.
- The driver will not overflow the buffer it places a received packet in.
- The driver will only send a packet to the adapter if there is room in the transmit FIFO.
- The driver will not under-write or over-write its device data structure.

This list of prevented errors includes all the abstraction invariants from section 4. In particular, the third point means that the following scenario happens safely: One thread is in the interrupt function when another interrupt is generated. The first thread holds the device data structure lock. The second interrupt thread will wait

for the lock while the first proceeds. The lock mean that two interrupt threads will not be able to process 3c509 interrupts at the same time.

Clay guarantees the specification matches the implementation. There may still be specification bugs. If the specification is correct, a badly written driver may not send or receive any packets but it will not be able to overrun or underrun the FIFO queues, cause memory errors, mishandle shared data structures.

5.3 A Toy Operating System

To show the practical application of the additions to Clay's abstract syntax described in Section 3, I used the additions to re-write portions of a toy operating system. The toy OS, Shale, is a kernel built using parts from the Flux OSKit [9] developed at the University of Utah. Shale is similar to Linux although much smaller to make it manageable for research purposes.

5.3.1 The OSKit

The Flux OSKit [9] was designed to make operating system building and interactions on an x86 architecture much simpler. It consists of libraries written in C which handle all of the standard tasks an operating system or interface to one might need. Using the OSKit, the "hello world" OS requires less than 20 lines of additional code.

Several systems have been written using the OSKit including the Fluke OS and ported versions of ML, SR, and Java [9].

- The Fluke OS was built using the OSKit and consists mostly of OSKit code, drastically cutting down on the code that needed to be specifically written for the OS.
- ML/OS is a version of SML/NJ ported to run on a PC. Using the OSKit, a small team of masters and undergraduate students completed the project in a

few months. The libraries in the OSKit took care of the x86 low level details so that most of the time was spent understanding the details of SML.

- SR/OS is a port of SR from Unix to a more platform-neutral version. It was accomplished by a graduate student in a matter of weeks.
- Java/PC is similar to Sun's JavaOS but took significantly less time to build; a few weeks of work by a graduate student.

The Flux OSKit allows operating systems to be created very quickly by providing functions for basic OS tasks. The functions used in these tasks can be rewritten or replaced to create a specific operating system.

5.3.2 Functionality

Shale, the toy OS, is written in C, assembly, and Clay using pieces of the OSKit. The OS uses the memory interface, networking, interrupt, thread, and driver support sections of the OSKit. It runs three kernel level applications: arp, ping, and a firewall. Shale uses standard linux device drivers including the 3Com 3c509 network adapter described in Section 4. The OS takes advantage of the OSKit's thread support to run multi-threaded using a round robin scheduler. All threads have the same priority.

The arp and ping applications behave like their standard Unix counterparts. They send out data packets and report the responses they get. The operating system can also respond to arp and ping requests sent from other machines on the internal or external network.

The firewall separates an internal network from an external one and expects two Ethernet connections, one to the internal network and one to the external. In protecting the internal network, the firewall allows only specified packet types to travel between the internal and external networks. Either network can arp or ping the firewall and vice versa. Currently only HTTP packets are allowed to cross between networks. A standard firewall might allow ssh and sftp but disallow telnet

and ftp. This firewall uses IP masquerading to hide the IP address of machines on the internal network. Packets sent through to the external network show the firewall machine's IP address regardless of which internal machine sent the packet. Arp allows the firewall to support IP routing and masquerading.

The interrupt code spawns a new thread for certain interrupts including those from the 3c509. This prevents a deadlock possibility: One thread calls the transmit function and acquires the device data structure lock. While it is in the transmit function, the adapter receives a packet and generates an interrupt. The interrupt code then spawns a new thread calling the 3c509 interrupt function. The interrupt thread needs to acquire the device data structure lock so it will block until the transmit thread releases the lock. Interrupts are only disabled long enough to spawn a new interrupt thread so the transmit function will be able to proceed and will not deadlock. If the interrupt did not spawn a new thread and left the old interrupts disabled, the process scheduler would not switch processes and the transmit thread would never release its lock causing a deadlock situation.

6 Measures of Success

A type-safe language for operating systems can be judged by several measures. The first concerns how safe the language actually is. What errors can it prevent? The second concern is how much complicated information must be added to the code to achieve the safety. The third measure is how well code sections written in the language work with legacy code sections and whether the legacy code can be automatically translated into the type-safe language. Does it play well with legacy code? The fifth measure is how easy the language is to use. Does it take longer than expected to write int eh language? Can other programmers read your code? The fifth measure is whether the type-safe code is slower than a legacy version of the same code. The fifth measure concerns how extensible the language is. Will the safety features of this language be enough for other applications? I performed some stress tests on the adapter to check its behavior under large loads and found that it ran safely as specified in the Clay implementation.

6.1 Safety

Clay was already able to prevent aliasing errors, buffer overflows, null pointer dereferences and many other errors. My research prevents some lock problems and enables better abstraction enforcement through special types. The abstract syntax and matching Clay implementation of locks can prevent the "forget to acquire" errors. Clay should be able to catch the "forget to release" problem at the end of function calls. That leaves the "re-entrant locks" problem unsolved by Clay's implementation or the lock abstract syntax. State and configuration types are extensions of concepts already present in my abstract machine. State types allow code to track operating system without impacting run-time. Configuration types enable the programmer to add invariants from abstractions directly to the code. By checking state-type values

against the configuration-type values in function headers, the program guarantees that the encoded abstractions are being enforced.

6.2 Additions To Code

In order to implement the abstractions required by the device driver, a significant amount of type information was added to the function headers and several IO functions had to be duplicated to easily handle different calling invariants. Relatively little had to be done to the actual function calls other than small syntax changes (adding "let") and passing or returning state and configuration-type values in the functions calls. As intended, most of the safety burden is on the few function headers rather than the many function calls. The extra type information ballooned the 3c509 code from 945 lines to over 4000 lines of code.

6.3 Legacy Code Interaction

Interactions between Clay and C++ are simpler than those between Clay and C. Function calls from Clay to C++ require a header declaration with a *native* marker in the Clay code and wrapper functions to coerce the types output by Clay to those expected by the legacy C++ code. The Shale operating system is written in C and the interface between Clay and C requires *extern "C"* statements and wrapper functions. It is difficult to pass function pointers between C and Clay since Clay expects a more exact argument type.

6.4 Ease of Use

Clay is not an easy language to use. Some of the syntax differences between C and Clay are subtle but important in the types they produce (e.g. $int\ x\ =\ 3;$ vs $let\ x\ =\ 3;$). Writing Clay code with driver specifications includes time spent turning notes

on the specification into mathematical comparisons (e.g. "The index is legal for the array bounds" becomes $[int\ N,\ type\ T,\ int\ I;\ 0 <= I\ \&\&\ I < N]$ for an array of type $Array[N, T]$ and an index of type $Int[I]$). From personal experience understanding uncommented Clay code is often very difficult. An informal survey of C programmers indicated than the format of commented Clay code was readable but the types were indecipherable without an explanation. Clay programmers who understood the format and possibilities of Clay types should have less problem reading Clay code.

I have both ported some code from Clay to C (the driver) and written some code directly in Clay (the lock code). In order to port the driver to Clay, I throughly read the 3c509 manual and took notes on the specification it described. These notes are presented in section 4.0.1. I also took notes on any specification implied by the code and comments of Donald Becker's 3c5x9 driver. Using my notes on the adapter functionality, I created state types for all of the states the driver might need. This included the "adapter busy" state and the port access configurations. Some of these types are static and will not change as the driver runs. These types became the nonlinear configuration types.

The driver consists mainly of memory operations on the shared data and IO operations on the adapter. The memory operations were ported using Mems and the load and store operations. Since the shared data structure consists of integers, shorts, characters, arrays, and pointers, I created several different types of Mems for each size and type of data. Each type of Mem needed its own set of load and store interfaces. Donald Becker's driver uses six assembly IO calls. I split these into several different interfaces for the status checks, packet data accesses, commands, and general IO operations. The interfaces for the commands were then split into interfaces which changed various adapter states and generic command interfaces. All total, the six assembly calls from C became about 20 wrapped assembly interfaces in Clay. In order to share both the data structure and access to the adapter, I created a

structure of all the adapter *Mem*s and the state-type and configuration-type values. The overall structure was linear and size 0 so it could be acquired from a lock. I added one integer to the end of the actual shared data structure for the lock bit. All code which uses this data structure assumes it has size *sizeof(struct device)* so adding an integer to the end of the structure should not cause problems elsewhere. The next step was to port the 3c509 portions of the 3c5x9 driver line by line. I had to re-arrange some code sections in order to use the functional style required by linear types. Conditional statements, for instance, require that the validity of a linear value be the same before and after the statement, so all branches must leave the linear with the same validity. When the driver was written, I moved on to type checking and fixing mismatches between the specification and implementation. Most of the mismatches involved typographical errors and dropped linears.

Porting the 3c509 driver to Clay took longer than it likely took to write the original C driver. Once the code type checked, it ran with only one error. The error was an incorrect data structure offset and was fixed within a few hours of discovering it. In this case, it was a specification error (my C-to-Clay interface specified the wrong data offset).

About half the porting time was spent writing a specification in mathematical terms. This included describing all of the IO interfaces with arithmetic constraints and integer type-parameters. The other half of the time was spent directly translating C syntax to Clay syntax. Because Clay implements abstractions not specifically mentioned in C code, it is not possible to automatically translate all C to Clay and produce a safe operating system that follows all the OS safety abstractions. However, it would be possible to translate some C syntax to Clay syntax and add obvious type information. This would speed up translation of operating system sections. Such a translator does not currently exist but it could be a future project. The remainder of the porting time was divided between creating my specification from the 3c509

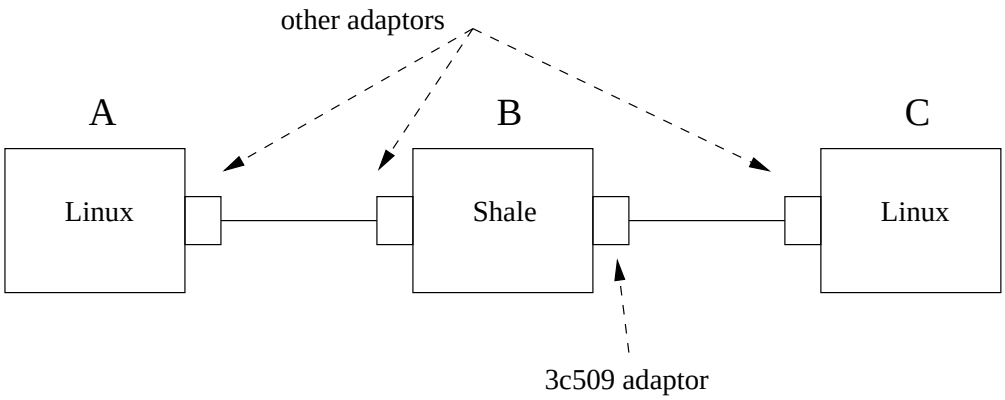

Figure 23: Network setup of my operating system and the 3c509 adapter

manual and tracking down invalid linear values.

Unlike porting C code to Clay, implementing locks in Clay did not take much longer than implementing them in C. The lock specification in the abstract syntax ported swiftly and easily to Clay.

6.5 Speed

I tested the ping program on 4 different versions of the operating system. All versions run multi-threaded but some do not spawn a new thread for 3c509 interrupts. The four versions are:

1. C 3c509 driver and same-thread 3c509 interrupts.
2. Clay 3c509 driver and same-thread 3c509 interrupts.
3. C 3c509 driver and spawned 3c509 interrupts.
4. Clay 3c509 driver and spawned 3c509 interrupts.

Figure 23 shows the setup of my testing network. Computer B runs the Shale operating system and one 3c509 adapter. Computers A and C are connected only through computer B. Computer C is protected by Shale's firewall and IP masquerading so all access to C must originate from B or C.

		Pings			
		B to A	B to C	C to B	C to A
1	C driver	1.387 ms	1.605 ms	.528 ms	.371 ms
2	Clay driver	1.387 ms	1.649 ms	.391 ms	.567 ms
3	C driver, spawn	1.399 ms	1.880 ms	5.42 ms	6.73 ms
4	Clay driver, spawn	2.118 ms	1.389 ms	4.4 ms	3.6 ms

Table 1: Timing test ping results.

For each version of Shale I ran 4 sets of 10 pings. The pings tested the connections from B to A and C and from C to A and B. The connection from B to A is a control since it does not involve the 3c509 adapter. For each set of 10 pings, I recorded the quickest round trip time. I also recorded the slowest trip time but this information is less useful because external factors could slow down a ping packet immeasurably while factors inside Shale and the adapters limit the fastest possible round trip time.

Table 1 shows the fastest round trip ping packets between each computer using the different versions of Shale. All trip times are in milliseconds. The versions of Shale that spawn a new thread for each 3c509 interrupt are noticeably slower than the versions which do not spawn on interrupt. The difference between round trips with the C and Clay 3c509 drivers is negligible by comparison.

The Clay implementation of the 3c509 required several extra IO operations per function and eliminated several run-time bounds checks. The extra IO operations verified the current state-type values. In practice, these values are the expected values so this approach erred on the side of caution. Therefore is it not unexpected that the C and Clay versions of the 3c509 driver have similar trip times. The bounds checks on data input from C code could also be eliminated if the rest of Shale was written in Clay, further speeding up the Clay 3c509 driver.

The obvious conclusion is that spawning a new thread comes with substantial overhead but a Clay implementation on its own does not. The Clay 3c509 driver

on a non-spawning Shale can become deadlocked if a 3c509 interrupt comes in while a packet is being sent. The packet function holds the data structure lock and the interrupt function will block forever to acquire it. A compromise between always spawning a thread on a 3c509 interrupt and never spawning one is to test the data structure lock and only spawn a thread if the lock was already held. This way, a Clay 3c509 interrupt only spawns a new thread when it could otherwise cause deadlock.

6.6 Stress Tests

Although Clay theoretically prevents many errors, stress testing shows that the system does not exhibit these errors under harsh conditions. I ran 2 types of stress tests on the Clay 3c509 driver.

The first test spawned 18 threads on computer B. Each thread sent 3 ping requests to computer C. Each interrupt generated by ping response packets spawned another thread. This meant that a total of 54 threads were spawned. Any ping request packets which did not fit into the transmit FIFO queue were dropped and not transmitted. The read and write operations (ping request and ping response) became interleaved as the process scheduler cycled through all the processes. The ping packets sent out data varying in size from 32 to 828 bytes per packet. The threads shared the driver data structure and did not cause adapter failure. No packets were dropped.

The second test spawned 18 threads, again on computer B. Each thread sent 10 ping requests to computer C. The rest of the setup was the same as in the first test. Three packets were not transmitted as a result of a full transmit FIFO. The average round trip time per packet was about 10 times longer than packets sent singly without all the other threads running simultaneously. This test was repeated with 15 ping requests per thread and the number of dropped packets went up to 27. Several ping response packets were received during the same interrupt.

Overall, the adapter behaved as it is intended to under extreme load.

6.7 Extending this approach

My approach provides safety for locks, shared memory and system state, and IO operations. It should easily apply to other operating system drivers which use the above features. Its system-state tracking ability is limited to state which can be expressed by boolean or integer arguments. State requiring more complex expression methods would need a different form of state type. The configuration types are also limited to integer and boolean arguments. At this time I have not seen any OS code which would require such complex state or configuration types but I would not rule out the possibility. Most drivers communicate with their device through some kind of channel (IO calls or some other channel). The interfaces to the 3c509 IO calls should generalize to interfaces on other IO calls or other channels.

Some low-level devices use memory-mapped control registers. These memory locations are not under the control of the stack and thus do not have Mems and cannot be accessed by *load* and *store* operations. They would need their own capability type and access functions. This capability type would likely be similar to the $PortConfig$ configuration type used by my 3c509 driver. The ports to access 3c509 registers cannot be accessed as standard memory and and required they own capability types and IO access functions.

This approach should scale well with a few limitations. If large segments of code are written in Clay with function and type specifications then each function can make more assumptions about its inputs and fewer run-time checks will be necessary (For instance, I currently make a run-time check for null data structure pointers on the boundary of C OS code and Clay driver code. If more OS code was written in Clay, these checks could be eliminated). On the downside, some linear values including interrupt state would be used all over a multi-threaded OS written in Clay. Without

global values, most functions would need to take and return this value. This problem could be solved by a form of global storage which preserved linear access but acted like a global.

7 Related Work

Related work on better operating systems can be separated into three categories: expressive type-safe languages, which use type systems to prevent safety bugs; correctness and safety proofs, which prove an operating system or application is free of certain safety or implementation bugs; and automated debuggers, which help programmers find safety and implementation bugs.

7.1 Expressive Type-Safe Languages

Type-safe languages strive to prevent bugs that cause safety violations such as data sharing errors, buffer overflows, pointer aliasing errors, and memory leaks. The languages mentioned below all prevent a subset of safety violations.

Vault The Vault programming language [7] is an extended form of C that embeds resource management protocols in the source code's types. The protocols prevent dangling references, memory leaks, race conditions, and other errors that can occur if operations on resources happen out of order. The type checker automatically enforces all protocols at compile time.

Vault's "type guards" provide a way to specify protocols for temporal events. The protocols can specify that events occur in a certain order, which operations must occur before a data object is accessed, and that an operation must eventually occur. For example, the type guard on a pointer might specify that the pointer must be non-NULL before it is dereferenced.

There are several commands that can be used with a guard:

- Create a guard and a set of values
- Associate a guard with a variable
- Check the value of the guard
- Set the value of the guard

- Add its flag to a global set of accessible guards
- Remove its flag from the set

A guarded value can only be accessed if its flag is in the global set and the guard value matches given conditions. For example, a guarded pointer is created with a guard value of NULL. When the pointer is set, the guard value changes to non-NULL. Later the pointer can only be dereferenced if its flag is in the global set and the guard value is non-NULL.

Flags can be removed from the global set to grant exclusive access to the guarded resource. When exclusive access is no longer needed, the flag can be added to the global set again. Flag removal and replacement provide a compile-time solution to some aliasing problems that would otherwise require expensive reference counting.

Additional type information needed for the guards is used all over Vault programs but is brief in each location. Like Clay, Vault provides compile-time safety checks.

Cyclone The Cyclone programming language [17, 15] is another extended form of C which adds types to guarantee conditions about pointers and arrays and inserts run time checks where needed if no compile time check is present. Additionally, Cyclone adds support for convenient programing features including parametric polymorphism and pattern matching.

Cyclone provides safety guarantees about pointer usage. It prevents the cast of an integer to a pointer because this would allow code to overwrite a random memory location. It also disallows pointer arithmetic on a * pointer unless special types are used which can guarantee the pointer won't be incremented out of bounds. Cyclone inserts a NULL check with every * pointer dereference unless a special type has been used guaranteeing the pointer is non-NULL. Together, these features mean that a Cyclone program cannot crash due to * pointer errors.

Cyclone has several special pointer types that lessen the number of run time checks its compiler inserts into code. These include @fat, @notnull, and @zeroterm

pointers. @fat pointers keep track of bounds information and allow pointer arithmetic with compile time error checks. @notnull pointers must be initialized when they are declared and can be dereferenced safely without a NULL check. @zeroterm pointers are used with C style char * strings. This type indicates that the string is terminated by a 0 and can be combined with the @fat and @notnull types. When an @zerterm (and @fat) pointer is initialized, the length of the sequence is computed once and saved and the array indices can be checked statically. These three pointer types allow a Cyclone programmer to move most pointer related safety checks from run time to compile time.

Cyclone provides support for programming features often seen in ML or Haskell programs but missing from C. Some of these features, such as parametric polymorphism, allow Cyclone to type check several unsafe portions of C (involving void *). Other features, such as pattern matching, permit a concise way to pull small parts out of a larger object. This is achieved with Haskell style let declarations and with more powerful switch statements. The added features make programming easier and provide ways to safely port C code to Cyclone.

Overall, Cyclone provides a safer version of C with many checks done at compile time. Missing safety checks are inserted by the compiler. Work is in progress on an application to port C to Cyclone and on a safe lock system resembling Java's synchronized [15, 10] .

Modula-3 for Extensible Operating Systems Hsieh et al. [13] added support for extensible operating systems to Modula-3. Operating system extensions need to handle pointer casting to avoid unnecessary data copies and make calls to and from untrusted code. Safe extensions can directly access system services via system calls and system data via load and store operations. Writing these extensions in a type-safe language avoids many costly address-space switches. This research is similar to mine in that it studies the needs of an operating system component and moves

costly run-time steps to compile-time.

Typed Assembly Language Typing information is often lost during program compilation. Morrisett et al. [19, 18] created a type system for assembly language to preserve type information from higher level languages. The TAL assembler can type-check programs written in a type-safe language and compiled to TAL. With typed assembly, only one type checker is needed to check programs written in a variety of languages. TALx86 is a version of TAL for the Intel Pentium. When the TAL papers were published, a type-safe version of C called popcorn and part of Scheme could be compiled to TAL.

7.2 Correctness and Safety Proofs

The OS verification approach to safety is very accurate but every time a change or addition is made to the OS, the previous proofs are no longer valid and the process needs to be done all over again. For this reason, the correctness proofs for an OS are often not re-done after the OS is extended or changed.

A verified OS Kernel Bevier [2] wrote a small (10 page) OS kernel (KIT) in assembly language. He translated the machine code for the kernel to Boyer-Moore logic and used the Boyer-Moore theorem prover to prove its correctness. During this process, many bugs were discovered and fixed.

Despite its small size, the kernel implements several distributed communicating processes and provides many now-verified services including process scheduling, CPU time allocation, error handling, message passing, and an interface to asynchronous devices.

Proof Carrying Code Some verification methods put the burden on the programmer. An example of this is the Proof Carrying Code (PCC) of Necula and Lee [20]. To use PCC, the OS kernel defines a safety policy and makes it public. Applications use the policy to provide binaries with a proof that the code obeys the

policy and the kernel validates the proofs during compile time. Proof generation is difficult and a new proof must be generated with each version of the application; but an automated proof generator has recently been created for Java programs.

Safe Machine Code Sirer et al. [28] discovered the difficulty of keeping type information when integrating one language with another. This can cause verification problems when a program is written in several languages or code written in one language is run on an OS written in another language. (e.g. Modula-3 code and C code in Sirer's SPIN operating system).

To avoid multi-language problems, Xu et al. [33] developed a method of safety checking machine code that only requires some type information and linear constraints from the programmer. The safety checker takes the desired safety properties of an OS and checks that the machine code obeys these properties. This method allows code additions to a program or operating system to be written in any language.

Coalgebraic Class Specification Language Tews [26] translated the memory management portion of the Fiasco microkernel into the Coalgebraic Class Specification programming language and created a set of properties that should be true for the chosen code portion. The translated code and properties were run through the PVS theorem prover and the properties were verified or rejected. Tews states that the project took 3 months and much more time would be required to verify a full OS because there is no automated method of translating C++ (the language of Fiasco) into a language accepted by PVS.

Currently, Tews et al. [27] are verifying the whole Fiasco microkernel (about 15,000 lines of code). When finished, this will be one of the first fully-verified real operating systems.

7.3 Automated Debuggers

Since there are so many types of debuggers, only two are mentioned here. Both locate bugs on a large scale and are therefore suited for use on operating systems.

Bug Inference There are many debuggers currently available and significant advances are being made towards new automated debugging methods that put little burden on the programmer. One relatively recent advance is the heuristic bug checker of Engler et al. [8]. This checker takes a few basic facts such as a dereferenced pointer is non-NULL and a program and statically looks through the code to find inconsistencies. It develops beliefs about the code without programmer input and finds the points where beliefs are contradictory. If the code contains many points where a lock is acquired and then released, the checker will develop the belief that acquired locks are later released. If one lock is acquired twice in a row, this will contradict the belief and the checker will report it as a possible error.

This debugging method is able to catch bugs that rarely manifest themselves and are thus difficult to locate. However, bugs in actions that are taken only a few times may not be caught unless the checker is specifically looking for them. Missed bugs could include a pointer access followed by a Null check if few Null checks are done. Other types of bugs including array bounds violations are completely out of the range of this checker.

Engler et al. ran the bug inference system on the Linux and OpenBSD operating systems and discovered hundreds of bugs, many of which have since been patched. Their findings are published as Chou et al. [4]

Redundancy Checking Xi and Engler [32] recently showed that redundant code operations correlate with errors and provided a tool to find such redundancy. Their tool searches for idempotency errors ($x = x,\ x/x,\ x\|x$), redundant assignments that are written but never read, code sections that can never run, and several other error types. When testing it on Linux code, they found a few idempotency errors

that could have devastating results. They also found over 2000 possible redundant assignments; 129 of the 155 they inspected turned out to be real errors. The majority of the dead-code errors inspected also turned out to be real errors. Their tool can be used to find inconsistencies between the specification and implementation of a program as well as logical inconsistencies in the control flow of a specification.

8 Future Work

The main contributions of this research are concurrency and locks in an abstract syntax with a proof of soundness, extended type concepts to better enforce abstractions, and an implementation to show it actually works and provides safety guarantees to a vulnerable system.

There are many directions for possible future work in this area of language theory. The configuration types are used statically but there are no guarantees that their types cannot be changed. Such changes cause unsafe code behavior. There are also lock issues that the abstract machine does not handle. Looking at implementation issues, Donald Becker's 3c5x9 driver was written within the limits of C. A safe 3c509 driver written to follow the manual rather than an existing C driver might have fewer errors and better speed if it was written to take advantage of Clay's flexibility. A safe operating system depends on more than just its drivers. Other safety-critical system parts are likely to have needs not addressed by this research.

8.1 Static Configuration Types

The configuration type values needs by a network driver are all constant. The type parameters do not change while the system is being run. Since functions that rely on configuration types expect a given configuration type value to be static, changes in a configuration type could cause unsafe behavior. Configuration types do not currently have guarantees that they are static.

8.2 Re-writing Rather Than Porting The 3c509 Driver

Re-writing the 3c509 driver using the manual as a base rather than an existing driver would have several benefits. Clay uses the syntax and rules of my abstract machine to make many run-time safety checks possible at compile-time. A driver written to

have as few run-time safety checks as possible is likely to be faster than the current porting of the 3c5x9 driver to Clay. Errors propagated from driver to driver using cut and paste techniques would also be eliminated.

8.3 Problems Remaining for Locks

The abstract syntax and rules of locks has moved some of the burden or correctness from all uses of locks to the lock types and functions. A programmer using these locks cannot produce "forget to acquire" and "release unheld lock" errors because the type checker will catch them at compile time. This moves a large part of the burden off of the numerous creation, acquiring, and releasing calls and onto the single implementation of these functions.

The Clay compiler could catch "forget to release" errors by noticing linear values abandoned at the end of a code block. A better solution would allow the abstract machine to catch these errors. This solution might involve checks on the current state of a lock. While the "re-entrant lock" problem can be solved by lock numbering, there may be a better solution through types.

8.4 Improved Safer Locks

The native C functions are a continuing source of un-safety. The acquire and release functions both currently call C functions and therefore contain chances for errors. One way to limit the number of C functions that locks depend on is to rely on a single swap function so that both acquire and release can use the same C function to set the lock bit.

8.5 Other Safety-critical OS Components

An operating system depends on the safety of many parts. These include interrupts, the process scheduler, and inter-process communication. State and configuration types may not handle all the needs of these safety-critical OS components. A safe OS kernel would require further research into the needs of each important component.

A Clay/Locks: Formal Definition

This section presents the complete syntax and rules for the abstract machine $\lambda^{concurrent}$ which includes linear memory, concurrency support, and support for locks. $\lambda^{concurrent}$ is a subset of λ^{low}, described in [14], with additions for concurrency and locks and a few corrections. It presents rules for evaluation of $\lambda^{concurrent}$programs along with rules for type well-formedness and type checking. Since the Clay compiler strips extraneous types from the code while compiling to C, this section also shows rules for type erasure and subsequent untyped evaluation rules.

A.1 Abstract Syntax

This section defines the abstract syntax of $\lambda^{concurrent}$. The letter x is used to indicate a value variable, while α is used to indicate a type variable. As usual, we consider expressions and types that differ only in bound variable names to be equivalent.

linearity

$$\phi = \text{non} \mid \text{lin}$$

kinds

$$K = \overset{\phi}{i} \mid \text{int}$$

arithmetic

$$i = \cdots \mid -2 \mid -1 \mid 0 \mid 1 \mid 2 \mid \cdots$$

$$b = \text{true} \mid \text{false}$$

$$I = \alpha \mid i$$

types

$$\tau = \tau_1 \xrightarrow{\phi} \tau_2 \;\mid\; \phi\langle\overrightarrow{\tau}\rangle \;\mid\; I \;\mid\; \forall\alpha : K.\tau \;\mid\; \exists\alpha : K.\tau$$

$$\mid\; \mathrm{Int}(I) \;\mid\; \mathrm{Mem}(I,\tau) \;\mid\; \mathrm{Lock}(I,\tau) \;\mid\; \mathrm{bool}$$

expressions

$$e = i \;\mid\; b \;\mid\; x \;\mid\; e_1\, e_2 \;\mid\; e\tau \;\mid\; \phi\langle\overrightarrow{e}\rangle \;\mid\; \lambda x : \tau \xrightarrow{\phi} e \;\mid\; \Lambda\alpha : K.v \;\mid\; \mathrm{let}\,\langle\overrightarrow{x}\rangle = e_1 \,\mathrm{in}\, e_2$$

$$\mid\; \mathrm{if}\, e_1 \,\mathrm{then}\, e_2 \,\mathrm{else}\, e_3 \;\mid\; \mathrm{pack}[\tau_1, e] \,\mathrm{as}\, \exists\alpha : K.\tau_2 \;\mid\; \mathrm{unpack}\,\alpha, x = e_1 \,\mathrm{in}\, e_2$$

$$\mid\; \mathrm{load}(e_{\mathrm{ptr}}, e_{\mathrm{Mem}}) \;\mid\; \mathrm{store}(e_{\mathrm{ptr}}, e_{\mathrm{Mem}}, e_{\mathrm{v}}) \;\mid\; \mathrm{release}(e_{\mathrm{Lock}}, e_{\mathrm{ptr}}, e_{\mathrm{resource}})$$

$$\mid\; \mathrm{create\_lock}(e_{\mathrm{ptr}}, e_{\mathrm{Mem}}, e_{\mathrm{resource}}) \;\mid\; \mathrm{acquire}(e_{\mathrm{Lock}}, e_{\mathrm{ptr}}) \;\mid\; \mathrm{fix}\, x : \tau.\mathrm{v} \;\mid\; \mathrm{fact} \;\mid\; \mathrm{lock}$$

values

$$v = i \;\mid\; b \;\mid\; \Lambda\alpha : K.v \;\mid\; \mathrm{pack}[\tau_1, v] \,\mathrm{as}\, \exists\alpha : K.\tau_2 \;\mid\; \lambda x : \tau \xrightarrow{\phi} e \;\mid\; \phi\langle\overrightarrow{v}\rangle \;\mid\; \mathrm{fact} \;\mid\; \mathrm{lock}$$

untyped expressions

$$d = i \;\mid\; b \;\mid\; x \;\mid\; d_1\, d_2 \;\mid\; \langle\overrightarrow{d}\rangle \;\mid\; \lambda x \longrightarrow d \;\mid\; \mathrm{let}\,\langle\overrightarrow{x}\rangle = d_1 \,\mathrm{in}\, d_2$$

$$\mid\; \mathrm{if}\, d_1 \,\mathrm{then}\, d_2 \,\mathrm{else}\, d_3 \;\mid\; \mathrm{fix}\, x.u \;\mid\; \mathrm{load}(d_{\mathrm{ptr}}, d_{\mathrm{Mem}}) \;\mid\; \mathrm{store}(d_{\mathrm{ptr}}, d_{\mathrm{Mem}}, d_{\mathrm{v}})$$

$$\mid\; \mathrm{create\_lock}(d_{\mathrm{ptr}}, d_{\mathrm{Mem}}, d_{\mathrm{resource}}) \;\mid\; \mathrm{acquire}(d_{\mathrm{Lock}}, e_{\mathrm{ptr}}) \;\mid\; \mathrm{release}(d_{\mathrm{Lock}}, d_{\mathrm{ptr}}, d_{\mathrm{resource}})$$

untyped values

$$u = i \;\mid\; b \;\mid\; \lambda x \longrightarrow d \;\mid\; \langle\overrightarrow{u}\rangle$$

expressions for substitution

$$s = v \;\mid\; \mathrm{fix}\, x : \tau.v$$

environments

$$M = \{1 \mapsto v_1, \cdots, n \mapsto v_n\}$$

$$R = \{1 \mapsto \langle \overrightarrow{v_1} \rangle, \cdots, n \mapsto \langle \overrightarrow{v_n} \rangle\}$$

$$C = \Psi; \Theta; \Delta; \Gamma$$

$$\Psi = \{1 \mapsto \tau_1, \cdots, n \mapsto \tau_n\}$$

$$\Theta = \{1 \mapsto \langle \overrightarrow{\tau_1} \rangle, \cdots, n \mapsto \langle \overrightarrow{\tau_n} \rangle\}$$

$$\Delta = \{\alpha_1 \mapsto K_1, \cdots, \alpha_n \mapsto K_n\}$$

$$\Gamma = \{x_1 \mapsto \tau_1 \cdots, x_n \mapsto \tau_n\}$$

untyped environments

$$L = \{1 \mapsto u_1, \cdots, n \mapsto u_n\}$$

judgments

$$\Delta \vdash \tau : K$$

$$C \vdash e : \tau$$

$$\Psi_{spare}; \Theta; C \vdash (R, M, e : \tau)$$

$$(R, M, e) \rightarrow (R', M', e')$$

abbreviations

$$\text{let}\, x = e_1 \,\text{in}\, e_2 \triangleq \text{let}\, \langle x \rangle =^{\text{lin}} \langle e_1 \rangle \,\text{in}\, e_2$$

Notes on environments

We treat the environments Ψ, Θ, Δ, Γ as sets, so the order of elements does not matter: $\{x_1 \mapsto \tau_1, x_2 \mapsto \tau_2\} = \{x_2 \mapsto \tau_2, x_1 \mapsto \tau_1\}$.

The environments Ψ, Θ, Δ, Γ must be well-formed functions (and the definitions in this thesis apply only to well-formed functions):

$(i \mapsto \tau_1 \in \Psi) \wedge (i \mapsto \tau_2 \in \Psi) \Rightarrow \tau_1 = \tau_2$

$(i \mapsto \langle \overrightarrow{\tau_1} \rangle \in \Theta) \wedge (i \mapsto \langle \overrightarrow{\tau_2} \rangle \in \Theta) \Rightarrow \langle \overrightarrow{\tau_1} \rangle = \langle \overrightarrow{\tau_2} \rangle$

$(\alpha \mapsto K_1 \in \Delta) \wedge (\alpha \mapsto K_2 \in \Delta) \Rightarrow K_1 = K_2$

$(x \mapsto \tau_1 \in \Gamma) \wedge (x \mapsto \tau_2 \in \Gamma) \Rightarrow \tau_1 = \tau_2$

For Ψ, Θ, Δ, Γ we use the usual function application notation:

$\Gamma(x) = \tau \Leftrightarrow x \mapsto \tau \in \Gamma$.

Environment splitting

These definitions split environments into two parts, where linear elements must go into exactly one of the parts:

$$\Psi = \Psi_1, \Psi_2 \quad \Leftrightarrow \quad (\Psi = \Psi_1 \cup \Psi_2) \wedge (\mathit{domain}(\Psi_1) \cup \mathit{domain}(\Psi_2) = \varnothing)$$

$$\begin{aligned} \Delta \vdash \Gamma = \Gamma_1, \Gamma_2 \quad \Leftrightarrow \quad \forall x, \tau. \quad & (\Delta \vdash \tau : \overset{\text{non}}{i} \Rightarrow \quad ((\Gamma(x) = \tau) \Leftrightarrow (\Gamma_1(x) = \tau)) \\ & \qquad \wedge ((\Gamma(x) = \tau) \Leftrightarrow (\Gamma_2(x) = \tau))) \\ & \wedge (\Delta \vdash \tau : \overset{\text{lin}}{i} \Rightarrow \quad ((\Gamma(x) = \tau) \Rightarrow (\Gamma_1(x) = \tau) \\ & \qquad \mathit{xor} (\Gamma_2(x) = \tau))) \\ & \wedge (\Delta \vdash \tau : \overset{\text{lin}}{i} \Rightarrow \quad ((\Gamma(x) = \tau) \Leftarrow (\Gamma_1(x) = \tau) \\ & \qquad \vee (\Gamma_2(x) = \tau))) \end{aligned}$$

This defines how the entire set of enviromnents splits into two parts.

$\Psi; \Theta; \Delta; \Gamma = (\Psi_1; \Theta; \Delta; \Gamma_1), (\Psi_2; \Theta; \Delta; \Gamma_2) \Leftrightarrow (\Psi = \Psi_1, \Psi_2) \wedge (\Delta \vdash \Gamma = \Gamma_1, \Gamma_2)$

Environment extension

These add new elements to environments:

$\Psi, i \mapsto \tau \triangleq \Psi \cup \{i \mapsto \tau\}$, where $i \notin domain(\Psi)$

$\Theta, i \mapsto \langle \overrightarrow{\tau} \rangle \triangleq \Theta \cup \{i \mapsto \langle \overrightarrow{\tau} \rangle\}$, where $i \notin domain(\Theta)$

$\Delta, \alpha \mapsto K \triangleq \Delta \cup \{\alpha \mapsto K\}$, where $\alpha \notin domain(\Delta)$

$\Gamma, x \mapsto \tau \triangleq \Gamma \cup \{x \mapsto \tau\}$, where $x \notin domain(\Gamma)$

$(\Psi; \Theta; \Delta; \Gamma), i \mapsto \tau \triangleq (\Psi, i \mapsto \tau; \Theta; \Delta; \Gamma)$,

$(\Psi; \Theta; \Delta; \Gamma), i \mapsto \langle \overrightarrow{\tau} \rangle \triangleq (\Psi; \Theta, i \mapsto \langle \overrightarrow{\tau} \rangle; \Delta; \Gamma)$,

$(\Psi; \Theta; \Delta; \Gamma), \alpha \mapsto K_\alpha \triangleq (\Psi; \Theta; \Delta, \alpha \mapsto K_\alpha; \Gamma)$

where α does not appear anywhere in $(\Psi; \Theta; \Delta; \Gamma)$.

$(\Psi; \Theta; \Delta; \Gamma), x \mapsto \tau \triangleq (\Psi; \Theta; \Delta; \Gamma, x \mapsto \tau)$,

where x does not appear anywhere in $(\Psi; \Theta; \Delta; \Gamma)$ and $\Delta \vdash \tau :\overset{\phi}{i}$.

Nonlinear environments

The $^{\text{non}}$ operator removes any linearity from an environment. We use it to prohibit linearity in some of the type checking rules.

$\overset{\text{non}}{\Gamma}(\Delta) \triangleq \{x \mapsto \tau | (x \mapsto \tau \in \Gamma) \wedge (\Delta \vdash \tau : \overset{\text{non}}{i})\}$

If $C = \Psi; \Theta; \Delta; \Gamma$, then $\overset{\text{non}}{C} \triangleq \varnothing; \Theta; \Delta; \overset{\text{non}}{\Gamma}(\Delta)$.

A.2 Evaluation Rules

Definitions

- $(R_1, M_1, e_1) \overset{?}{\mapsto} (R_2, M_2, e_2)$ means (R_1, M_1, e_1) progresses in zero or one steps to (R_2, M_2, e_2).
- $(R_1, M_1, e_1) \overset{*}{\mapsto} (R_2, M_2, e_2)$ means (R_1, M_1, e_1) progresses in zero or more steps to (R_2, M_2, e_2).
- $(R_1, M_1, e_1) \overset{+}{\mapsto} (R_2, M_2, e_2)$ means (R_1, M_1, e_1) progresses in one or more steps to (R_2, M_2, e_2).

- $(R_1, M_1, e_1) \overset{?,(evaluation-rule)}{\mapsto} (R_2, M_2, e_2)$ means

  (R_1, M_1, e_1) progresses to (R_2, M_2, e_2) by applying the given evaluation rule zero or one times.

- $(R_1, M_1, e_1) \overset{*,(evaluation-rule)}{\mapsto} (R_2, M_2, e_2)$ and

  $(R_1, M_1, e_1) \overset{+,(evaluation-rule)}{\mapsto} (R_2, M_2, e_2)$ are defined in the same way as the above definition.

Rules

R has been omitted from the evaluation rules which do not affect it. M has been similarly omitted.

$$(E\_LOAD) \quad (R, M, \text{load}(i, fact)) \rightarrow (RM, {}^{\text{lin}}\langle M(i), fact\rangle)$$

$$(E\_STORE) \quad (R, M, \text{store}(i, fact, v)) \rightarrow (R, [i \mapsto v]M, fact)$$

$$(E\_CREATELOCK) \quad (R, M, \text{create\_ lock}(i, \text{fact}, v_{\text{resource}})) \rightarrow ([i \mapsto \langle \text{fact}, v_{\text{resource}}\rangle]R, M, \text{lock})$$

$$(E\_ACQUIRE1) \quad (R, M, \text{acquire}(\text{lock}, i)) \rightarrow ([i \mapsto \langle \text{fact}\rangle]R, [i \mapsto 1]M, v_{\text{resource}}) \text{ where } R(i) = \langle \text{fact}, v_{\text{resource}}\rangle \text{ and } M(i) = 0$$

$$(E\_ACQUIRE2) \quad (R, M, \text{acquire}(\text{lock}, i)) \rightarrow (R, M, \text{acquire}(\text{lock}, i)) \text{ where } R(i) = \langle \text{fact}\rangle \text{ and } M(i) = 1$$

$$(E\_RELEASE1) \quad (R, M, \text{release}(\text{lock}, i, v_{\text{resource}})) \rightarrow ([i \mapsto \langle \text{fact}, v_{\text{resource}}\rangle]R, [i \mapsto 0]M, \langle\rangle) \text{ where } R(i) = \langle \text{fact}\rangle \text{ and } M(i) = 1$$

$$(E\_RELEASE2) \quad (R, M, \text{release}(\text{lock}, i, v_{\text{resource}}))$$
$$\rightarrow ((R, M, \text{release}(\text{lock}, i, v_{\text{resource}})))$$
$$\text{where } R(i) = \langle \text{fact}, v_{\text{resource}} \rangle \text{ and } M(i) = 0$$

$$(E\_ABSAPP) \quad (\lambda x : \tau \xrightarrow{\phi} e_1)v_2 \rightarrow [x \mapsto v_2]e_1$$

$$(E\_TABSTAPP) \quad (\Lambda\alpha : K.v)\tau \rightarrow [\alpha \mapsto \tau]v$$

$$(E\_LET) \quad \text{let } \langle x_1, \cdots, x_n \rangle = \phi\langle v_1, \cdots, v_n \rangle \text{ in } e \rightarrow [x_1 \mapsto v_1, \cdots, x_n \mapsto v_n]e$$

$$(E\_IF1) \quad \text{if } true \text{ then } e_1 \text{ else } e_2 \rightarrow e_1$$

$$(E\_IF2) \quad \text{if } false \text{ then } e_1 \text{ else } e_2 \rightarrow e_2$$

$$(E\_UNPACK) \quad \text{unpack } \alpha, x = (\text{pack}[\tau_1, v_1] \text{ as } \tau_2) \text{in } e_2 \rightarrow [\alpha \mapsto \tau_1, x \mapsto v_1]e_2$$

$$(E\_FIX) \quad \text{fix } x : \tau.\text{v} \rightarrow [x \mapsto \text{fix } x : \tau.\text{v}]\text{v}$$

Rather than writing each of the congruence rules separately ($\frac{e_2 \rightarrow e_2'}{v_1\ e_2 \rightarrow v_1\ e_2'}$, etc.), we use E to indicate an expression with one (shallowly dug) hole in it, and $E[e]$ to indicate the expression with the hole replaced by e, so that a single evaluation rule covers all the cases. This is only for notational convenience.

$$E[e] = e\tau \ \mid \ \text{pack}[\tau_1, e] \text{ as } \exists\alpha : K.\tau_2 \ \mid \ \text{unpack } \alpha, x = e \text{ in } e_2 \ \mid \ ee_2 \ \mid \ v_1 e$$

$$\mid \ \phi\langle e_i, \cdots, e_{k-1}, e, e_{k+1}, \cdots, e_n \rangle \ \mid \ \text{let } \langle \overrightarrow{x} \rangle = e \text{ in } e_2 \ \mid \ \text{if } e \text{ then } e_2 \text{ else } e_3$$

$$\mid \ \text{load}(e, e_{\text{Mem}}) \ \mid \ \text{load}(v_{\text{ptr}}, e)$$

$$\mid \ \text{store}(e, e_{\text{Mem}}, e_{\text{v}}) \ \mid \ \text{store}(v_{\text{ptr}}, e, e_{\text{v}}) \ \mid \ \text{store}(v_{\text{ptr}}, v_{\text{Mem}}, e)$$

$$\mid \ \text{create\_ lock}(e, e_{\text{Mem}}, e_{\text{resource}}) \ \mid \ \text{create\_ lock}(v_{\text{ptr}}, e, e_{\text{resource}})$$

$$\mid \text{create\_lock}(v_{\text{ptr}}, v_{\text{Mem}}, e) \mid \text{acquire}(e, e_{\text{ptr}}) \mid \text{acquire}(v_{\text{Lock}}, e)$$

$$\mid \text{release}(e, e_{\text{ptr}}, e_{\text{resource}}) \mid \text{release}(v_{\text{Lock}}, e, e_{\text{resource}}) \mid \text{release}(v_{\text{Lock}}, v_{\text{ptr}}, e)$$

$$(congruence\, rule)\frac{(R, M, e) \to (R', M', e')}{(R, M, E[e]) \to (R', M', E[e'])}$$

$$\frac{e \to e'}{(R, M, e) \to (R, M, e')}$$

A.3 Type Well-formedness Rules

$$(K\_IVAR) \quad \Delta \vdash i : \text{int}$$

$$(K\_TVAR) \quad \Delta, \alpha : K \vdash \alpha : K$$

$$(K\_BOOL) \quad \Delta \vdash bool : \overset{\text{non}}{1}$$

$$(K\_ALL) \quad \frac{\Delta, \alpha : K \vdash \tau : \overset{\phi}{i}}{\Delta \vdash \forall \alpha : K.\tau : \overset{\phi}{i}}$$

$$(K\_SOME) \quad \frac{\Delta, \alpha : K \vdash \tau : \overset{\phi}{i}}{\Delta \vdash \exists \alpha : K.\tau : \overset{\phi}{i}}$$

$$(K\_ABS) \quad \frac{\Delta \vdash \tau_1 : \overset{\phi_1}{i_1} \quad \Delta \vdash \tau_2 : \overset{\phi_2}{i_2}}{\Delta \vdash \tau_1 \xrightarrow{\phi} \tau_2 : \overset{\phi}{1}}$$

$$(K\_NLTUPLE) \quad \frac{\forall j.(\Delta \vdash \tau_j : \overset{\text{non}}{i_j})}{\Delta \vdash^{\text{non}} \langle \overrightarrow{\tau} \rangle : \overset{\text{non}}{i}} \text{ where } (i = \sum_j i_j)$$

$$(K\_LTUPLE) \quad \frac{\forall j.(\Delta \vdash \tau_j : \overset{\phi_j}{i_j})}{\Delta \vdash^{\text{lin}} \langle \overrightarrow{\tau} \rangle : \overset{\text{lin}}{i}} \text{ where } (i = \sum_j i_j)$$

$$(K\_INT) \quad \frac{\Delta \vdash I : \text{int}}{\Delta \vdash \text{Int}(I) : \overset{\text{non}}{1}}$$

$$(K\_MEM) \quad \frac{\Delta \vdash I : \text{int} \quad \Delta \vdash \tau : \overset{\text{non}}{1}}{\Delta \vdash \text{Mem}(I, \tau) : \overset{\text{lin}}{0}}$$

$$(K\_LOCK) \quad \frac{\Delta \vdash I : \text{int} \quad \Delta \vdash \tau : \overset{\text{lin}}{0}}{\Delta \vdash \text{Lock}(I, \tau) : \overset{\text{non}}{0}}$$

Valid Lock States

$$(T\_FREE) \quad \text{Int}(0); \langle \tau_{Mem}, \tau_{resource} \rangle \vdash \langle \tau_{Mem}, \tau_{resource} \rangle$$

$$(T\_HELD) \quad \text{Int}(1); \langle \tau_{Mem}, \tau_{resource} \rangle \vdash \langle \tau_{Mem} \rangle$$

A.4 Type Checking Rules

$$(T\_RMe) \quad \frac{\begin{array}{c}\Psi = \Psi_{spare}, \Psi_e, \Psi_{R1}, \cdots, \Psi_{Rn} \quad \Psi_e; \Theta; \Delta; \Gamma \vdash e : \tau \\ \forall i \in dom(\Psi).(\varnothing; \Theta; \varnothing; \varnothing \vdash M(i) : \Psi(i)) \\ \forall i \in dom(\Theta).(\Psi_{Ri}; \Theta; \varnothing; \varnothing \vdash R(i) : \tau_i \text{ and } \Psi(i); \Theta(i) \vdash \tau_i)\end{array}}{\Psi; \Theta; \Delta; \Gamma \vdash (R, M, e : \tau)}$$

$$(T\_VAR) \quad \overset{\text{non}}{C}, x : \tau \vdash x : \tau$$

$$(T\_FACT) \quad \overset{\text{non}}{C}, I \mapsto \tau \vdash \text{fact} : \text{Mem}(I, \tau)$$

$$(T\_LOCK) \quad \varnothing; \Theta, I \mapsto \langle \text{Mem}(I, \text{Int}(J)), \tau_2 \rangle; \Delta; \overset{\text{non}}{\Gamma} \vdash \text{lock} : \text{Lock}(I, \tau_2)$$
$$\text{where } J = 0 \text{ or } J = 1$$

$$(T\_BOOL) \quad \overset{\text{non}}{C} \vdash b : \text{bool}$$

$$(T\_INT) \quad \overset{\text{non}}{C} \vdash i : \text{Int}(i)$$

$$(T\_LOAD) \quad \frac{C_1 \vdash e_{\text{ptr}} : \text{Int}(I) \quad C_2 \vdash e_{\text{Mem}} : \text{Mem}(I, \tau)}{C_1, C_2 \vdash \text{load}(e_{\text{ptr}}, e_{\text{Mem}}) : \overset{\text{lin}}{} \langle \tau, \text{Mem}(I, \tau) \rangle}$$

$$(T\_STORE) \quad \frac{\begin{array}{c}C_2 \vdash e_{\text{Mem}} : \text{Mem}(I, \tau_1) \quad C_3 \vdash e_{\text{v}} : \tau_2 \\ C_1 \vdash e_{\text{ptr}} : \text{Int}(I) \quad C_1, C_2, C_3 \vdash \tau_2 : \overset{\text{non}}{1}\end{array}}{C_1, C_2, C_3 \vdash \text{store}(e_{\text{ptr}}, e_{\text{Mem}}, e_{\text{v}}) : \text{Mem}(I, \tau_2)}$$

$$(T\_TABS) \quad \frac{C, \alpha : K \vdash v : \tau}{C \vdash \Lambda\alpha : K.v : \forall\alpha : K.\tau}$$

$$(T\_TAPP) \quad \frac{C \vdash e : \forall\alpha : K.\tau_1 \quad C \vdash \tau_2 : K}{C \vdash e\tau_2 : [\alpha \mapsto \tau_2]\tau_1}$$

$$(T\_TUPLE) \quad \frac{C = \overset{\text{non}}{C}, C_1, \cdots, C_n \quad \forall i.(C_i \vdash e_i : \tau_i) \quad C \vdash \phi\langle\overrightarrow{\tau}\rangle : K}{C \vdash \phi\langle\overrightarrow{e}\rangle : \phi\langle\overrightarrow{\tau}\rangle}$$

$$(T\_ABS) \quad \frac{\Delta \vdash \tau_1 : K \qquad \overset{\phi}{\Psi}; \Theta; \Delta; \overset{\phi}{\Gamma}, x : \tau_1 \vdash e : \tau_2}{\overset{\phi}{\Psi}; \Theta; \Delta; \overset{\phi}{\Gamma} \vdash \lambda x : \tau_1 \xrightarrow{\phi} e : \tau_1 \xrightarrow{\phi} \tau_2}$$

$$(T\_APP) \quad \frac{C_1, C_2 = \Psi; \Theta; \Delta; \Gamma \quad C_1 \vdash e_1 : \tau_a \xrightarrow{\phi} \tau_b \;\; C_2 \vdash e_2 : \tau_a}{C_1, C_2 \vdash e_1 e_2 : \tau_b}$$

$$(T\_LET) \quad \frac{C_a \vdash e_a : \langle\overrightarrow{\tau}\rangle \quad C_b, \overrightarrow{x : \tau} \vdash e_b : \tau_b}{C_a, C_b \vdash \text{let}\langle\overrightarrow{x}\rangle = e_a \,\text{in}\, e_b : \tau_b}$$

$$(T\_FIX) \quad \frac{C \vdash \tau : \overset{\phi}{i} \quad C, x : \tau \vdash \text{v} : \tau}{C \vdash (\text{fix}\, x : \tau.\text{v}) : \tau}$$

$$(T\_PACK) \quad \frac{C \vdash \tau_1 : K \quad C \vdash \exists\alpha : K.\tau_2 : \overset{\phi}{i} \quad C \vdash e : [\alpha \mapsto \tau_1]\tau_2}{C \vdash \text{pack}[\tau_1, e] \,\text{as}\, \exists\alpha : K.\tau_2 : \exists\alpha : K.\tau_2}$$

$$(T\_UNPACK) \quad \frac{C_1 \vdash e_1 : \exists\alpha : K.\tau_1 \quad C_2, \alpha : K, x : \tau_1 \vdash e_2 : \tau_2}{C_1, C_2 \vdash \text{unpack}\,\alpha, x = e_1 \,\text{in}\, e_2 : \tau_2}$$

$$(T\_IF) \quad \frac{C_a \vdash e_1 : \text{bool} \quad C_b \vdash e_2 : \tau \quad C_b \vdash e_3 : \tau}{C_a, C_b \vdash \text{if}\, e_1 \,\text{then}\, e_2 \,\text{else}\, e_3 : \tau}$$

$$(T\_CREATELOCK) \quad \frac{C_1 \vdash e_{\text{ptr}} : \text{Int}(I) \quad C_2 \vdash e_{\text{Mem}} : \text{Mem}(I, \text{Int}(0)) \quad C_3 \vdash e_{\text{resource}} : \tau : \overset{\text{lin}}{0}}{C_1, C_2, C_3 \vdash \text{create\_ lock}(e_{\text{ptr}}, e_{\text{Mem}}, e_{\text{resource}}) : \text{Lock}(I, \tau)}$$

$$(T\_ACQUIRE) \quad \frac{C \vdash e_{\text{Lock}} : \text{Lock}(I, \tau) \quad e_{\text{ptr}} : \text{Int}(I)}{C \vdash \text{acquire}(e_{\text{Lock}}, e_{\text{ptr}}) : \tau}$$

$$(T\_RELEASE) \quad \frac{C_1 \vdash e_{\text{Lock}} : \text{Lock}(I, \tau) \quad e_{\text{ptr}} : \text{Int}(I) \quad C_2 \vdash e_{\text{resource}} : \tau}{C_1, C_2 \vdash \text{release}(e_{\text{Lock}}, e_{\text{ptr}}, e_{\text{resource}}) :^{\text{non}} \langle\rangle}$$

A.5 Type Erasure Rules

The erasure rules are defined only for well-typed terms: to erase types, we require a derivation of some judgment $C \vdash (R, M, e : \tau)$ to be given, and we use the derivation to annotate subexpressions inside (R, M, e) with types. Specifically, if $e = ...e_1...$, and the derivation of $C \vdash (R, M, e : \tau)$ contains the judgment $C_1 \vdash e_1 : \tau_1$, then we annotate e with τ_1 as $e = ...(e_1 : \tau_1)...$ when convenient. We annotate types with kinds in a similar fashion. These annotations guide the erasure rules.

$$(ER\_RMe) \quad \text{erase}((R, M, e)) \;=\; (\,\text{erase}(M),\, \text{erase}(e))$$

$$(ER\_M) \quad \text{erase}(M) \;=\; \{1 \mapsto \text{erase}(v_1), \cdots, n \mapsto \text{erase}(v_n)\}$$

$$(ER\_i) \quad \text{erase}(i) \;=\; i$$

$$(ER\_b) \quad \text{erase}(b) \;=\; b$$

$$(ER\_x) \quad \text{erase}(x) \;=\; x$$

$$(ER\_APP) \quad \text{erase}(e_1\, e_2) \;=\; \text{erase}(e_1)\, \text{erase}(e_2)$$

$$(ER\_APPT) \quad \text{erase}(e\, \tau) \;=\; \text{erase}(e)$$

$$(ER\_TUPLE) \quad \text{erase}(\phi\langle e_1, \ldots, e_n\rangle) \;=\; \langle\, \text{erase}(e_1), \ldots, \text{erase}(e_n)\rangle$$

$$(ER\_FUN) \quad \lambda x : \tau \xrightarrow{\phi} e \;=\; \lambda x \longrightarrow \text{erase}(e)$$

$$(ER\_TFUN) \quad \text{erase}(\Lambda\alpha : K.v) \;=\; \text{erase}(v)$$

$$(ER\_LET) \quad \text{erase}(\text{let}\, \langle \overrightarrow{x} \rangle = e_1 \,\text{in}\, e_2) \;=\; \text{let}\, \langle \overrightarrow{x} \rangle = \text{erase}(e_1) \,\text{in}\, \text{erase}(e_2)$$

$$(ER\_IF) \quad \text{erase}(\text{if}\, e_1 \,\text{then}\, e_2 \,\text{else}\, e_3) \;=\; \text{if erase}(e_1) \,\text{then erase}(e_2) \,\text{else erase}(e_3)$$

$$(ER\_PACK) \quad \text{erase}(\text{pack}[\tau_1, e] \,\text{as}\, \exists\alpha : K.\tau_2) \;=\; \text{erase}(e)$$

$$(ER\_UNPACK) \quad \text{erase}(\text{unpack}\,\alpha, x = e_1 \,\text{in}\, e_2) \;=\; \text{let}\, x = \text{erase}(e_1) \,\text{in erase}(e_2)$$

$$(ER\_FIX) \quad \text{erase}(\text{fix}\, x : t :\overset{\phi}{i}\, .v) \;=\; \text{fix}\, x.\, \text{erase}(v) \,\text{where}\, i > 0$$

$$(ER\_FIX0) \quad \text{erase}(\text{fix}\, x : t :\overset{\phi}{0}\, .v) \;=\; \langle\rangle$$

$$(ER\_FACT)\, \text{erase}(\text{fact}) \;=\; \langle\rangle$$

$$(ER\_LOAD) \quad \text{erase}(\text{load}(e_{\text{ptr}}, e_{\text{Mem}})) \;=\; \text{load}(\,\text{erase}(e_{\text{ptr}}),\, \text{erase}(e_{\text{Mem}}))$$

$$(ER\_STORE) \quad \text{erase}(\text{store}(e_{\text{ptr}}, e_{\text{Mem}}, e_{\text{v}}))$$
$$= \;\text{store}(\,\text{erase}(e_{\text{ptr}}),\, \text{erase}(e_{\text{Mem}}),\, \text{erase}(e_{\text{v}}))$$

$$(ER\_LOCK) \quad \text{erase}(\text{lock}) \;=\; \langle\rangle$$

$$(ER\_CREATELOCK) \quad \text{erase}(\text{create\_ lock}(e_{\text{ptr}}, e_{\text{Mem}}, e_{\text{resource}}))$$
$$= \;\text{create\_ lock}(\,\text{erase}(e_{\text{ptr}}),\, \text{erase}(e_{\text{Mem}}),\, \text{erase}(e_{\text{resource}}))$$

$$(ER\_ACQUIRE) \quad \text{erase}(\text{acquire}(e_{\text{Lock}}, e_{\text{ptr}})) \;=\; \text{acquire}(\,\text{erase}(e_{\text{Lock}}),\, \text{erase}(e_{\text{ptr}}))$$

$$(ER\_RELEASE) \quad \text{erase}(\text{release}(e_{\text{Lock}}, e_{\text{ptr}}, e_{\text{resource}}))$$
$$= \;\text{release}(\,\text{erase}(e_{\text{Lock}}),\, \text{erase}(e_{\text{ptr}}),\, \text{erase}(e_{\text{resource}}))$$

A.6 Untyped Evaluation Rules

Some of the evaluation rules will work for both typed and untyped evaluation. These new rules apply to untyped evaluation when the standard evaluation rules cannot.

$$(D\_LOAD) \quad (L, \text{load}(i, \langle\rangle)) \rightarrow (L, \langle L(i), \langle\rangle\rangle)$$

$$(D\_STORE) \quad (L, \text{store}(i, \langle\rangle, u)) \rightarrow ([i \mapsto u]L, \langle\rangle)$$

$$(D\_ABSAPP) \quad (\lambda x \longrightarrow d_1)u_2 \rightarrow [x \mapsto u_2]d_1$$

$$(D\_FIX) \quad \text{fix}\, x.u \rightarrow [x \mapsto \text{fix}\, x.u]u$$

$$(D\_CREATELOCK) \quad (L, \text{create\_ lock}(i, \langle\rangle, \langle\rangle)) \rightarrow (L, \langle\rangle) \text{ where } L(i) = 0$$

$$(D\_ACQUIRE1) \quad (L, \text{acquire}(\langle\rangle, i)) \rightarrow ([i \mapsto 1]L, \langle\rangle) \text{ where } L(i) = 0$$

$$(D\_ACQUIRE2) \quad (L, \text{acquire}(\langle\rangle, i)) \rightarrow (L, \text{acquire}(\langle\rangle, i)) \text{ where } L(i) = 1$$

$$(D\_RELEASE1) \quad (L, \text{release}(\langle\rangle, i, \langle\rangle)) \rightarrow ([i \mapsto 0]L, \langle\rangle) \text{ where } L(i) = 1$$

$$(D\_RELEASE2) \quad (L, \text{release}(\langle\rangle, i, \langle\rangle)) \rightarrow (L, \text{release}(\langle\rangle, i, \langle\rangle)) \text{ where } L(i) = 0$$

B Clay/Locks: Soundness Proof

This section provides a proof of soundness (type safety) for $\lambda^{concurrent}$. Soundness means that well-typed programs will never get stuck. The proof consists of a proof of preservation and a proof of progress, in the syntactic style of [30] (also see [22] for an general introduction to syntactic approaches to semantics and types). Preservation says that a well-typed program steps to another well-typed program. Progress says that a well-typed program can always step unless it is in an acknowledged end state, a value. This chapter also presents a proof that the extraneous types can be erased without upsetting the run-time behavior. Supporting lemmas stated here without proof are proven in [14].

Formally stated, the theorems proved here are:

Preservation: If $\Psi_e; \Theta; \Delta; \Gamma \vdash e : \tau$ and $\Psi_{spare}, \Psi_e; \Theta; \Delta; \Gamma \vdash (R, M, e : \tau)$ and $(R, M, e) \rightarrow (R', M', e')$, then $\Psi'_e; \Theta'_e; \Delta; \Gamma \vdash e' : \tau$ and $\Psi_{spare}, \Psi'_e; \Theta'; \Delta; \Gamma \vdash (R', M', e' : \tau)$.

Progress: If (R, M, e) is closed and well-typed ($C \vdash (R, M, e : \tau)$ for some τ and $C = \Psi; \Theta; \varnothing; \varnothing$), then either e is a value or else there is some (R', M', e') so that $(R, M, e) \rightarrow (R', M', e')$.

Erasure: If (R, M, e) is closed and well-typed ($C \vdash (R, M, e : \tau)$ for some τ and $C = \Psi; \Theta; \varnothing; \varnothing$) then the following hold:

1. If $(R, M, e) \rightarrow (R', M', e')$ then $\text{erase}((R, M, e)) \stackrel{?}{\rightarrow} \text{erase}((R', M', e'))$.
2. If $\text{erase}((R, M, e)) \rightarrow (L', d')$, then $(R, M, e) \stackrel{+}{\rightarrow} (R', M', e')$ and $\text{erase}((R', M', e')) = (L', d')$.
3. If $\text{erase}(e)$ is a value, then $\text{erase}((R', M', e')) = (L', d')$ and $\text{erase}((R, M, e)) = \text{erase}((R, M, v))$.

B.1 Preservation Lemmas

LEMMA [WEAKENING FOR TERMS]

$(C_1 \subseteq C_2) \wedge (C_1 \vdash e : \tau) \Rightarrow (C_2 \vdash e : \tau)$.

LEMMA [WEAKENING FOR TYPES]

$(C_1 \subseteq C_2) \wedge (C_1 \vdash \tau : K) \Rightarrow (C_2, C_3 \vdash \tau : K)$.

LEMMA [SPLIT SUBSET]

If $C = C_1, C_2$ and $C_1 \stackrel{\text{non}}{\subseteq} C_1'$, then there is some C_2' such that $C_2 \stackrel{\text{lin}}{\subseteq} C_2'$ and $C_1, C_2 \stackrel{\text{non}}{\subseteq} C_1', C_2'$.

If $C = C_1, C_2$ and $C_1 \stackrel{\text{non}}{\subseteq} C_1'$, then there is some C_2' such that $C_2 \subseteq C_2'$ and $C_1, C_2 \stackrel{\text{non}}{\subseteq} C_1', C_2'$.

LEMMA [TYPE SUBSTITUTION]

If $C, \overrightarrow{\alpha : K_\alpha} \vdash \tau : K$, and $C \vdash \tau_{\alpha_i} : K_{\alpha_i}$, then $C \vdash [\overrightarrow{\alpha \mapsto \tau_\alpha}]\tau : K$.

(Corollary, via the type environment substitution lemma: if $C, \overrightarrow{\alpha : K_\alpha} \vdash \tau : K$, and $C \vdash \tau_{\alpha_i} : K_{\alpha_i}$, then $[\overrightarrow{\alpha \mapsto \tau_\alpha}]C \vdash [\overrightarrow{\alpha \mapsto \tau_\alpha}]\tau : K$.)

LEMMA [TERM SUBSTITUTION]

If we make the following definitions and assumptions:

- We define a substitution $[s] = [\overrightarrow{\alpha \mapsto \tau_\alpha}, \overrightarrow{x \mapsto s_x}, \overrightarrow{y \mapsto s_y}, \overrightarrow{z \mapsto s_z}]$
- $C = \Psi; \Theta; \Delta; \Gamma$
- No α_i appears free in $\Psi; \Theta; \Delta$ (note: typically, this condition can be satisfied by alpha-renaming α_i before invoking this lemma).
- $[\overrightarrow{\alpha \mapsto \tau_\alpha}]C = \Psi; \Theta; \Delta; [\overrightarrow{\alpha \mapsto \tau_\alpha}]\Gamma$ is syntactically well-formed (this assumption is needed because substitutions of non-integers into integers I does not always produce integers).
- $C, \overrightarrow{\alpha : K_\alpha}, \overrightarrow{x : \tau_x}, \overrightarrow{y : \tau_y} \vdash e : \tau$
- $z_i \notin domain(\Gamma)$
- $[\overrightarrow{\alpha \mapsto \tau_\alpha}]\ \overset{\text{non}}{C} \vdash s_{x_i} : [\overrightarrow{\alpha \mapsto \tau_\alpha}]\tau_{x_i}$ (where τ_{x_i} are nonlinear types)
- $[\overrightarrow{\alpha \mapsto \tau_\alpha}](\overset{\text{non}}{C}, C_{y_i}) \vdash s_{y_i} : [\overrightarrow{\alpha \mapsto \tau_\alpha}]\tau_{y_i}$ (where τ_{y_i} are linear types)
- $[\overrightarrow{\alpha \mapsto \tau_\alpha}](\overset{\text{non}}{C}, C_{z_i}) \vdash s_{z_i} : [\overrightarrow{\alpha \mapsto \tau_\alpha}]\tau_{z_i}$ (where τ_{z_i} are linear types)
- $C \vdash \tau_{\alpha_i} : K_{\alpha_i}$

then we can conclude the following:

- $[\overrightarrow{\alpha \mapsto \tau_\alpha}](C, \overrightarrow{C_{y_i}}) \vdash [s]e : [\overrightarrow{\alpha \mapsto \tau_\alpha}]\tau$

Context abreviations

Some abreviations used in the preservation proof:

$C_e = \Psi_e; \Theta_e; \Delta; \Gamma$

$C = \Psi_{spare}, \Psi_e; \Theta; \Delta; \Gamma$

$C'_e = \Psi'_e; \Theta'_e; \Delta; \Gamma$

$C' = \Psi_{spare}, \Psi'_e; \Theta'; \Delta; \Gamma$

LEMMA [FINAL-STEP]

This lemma is the last step common to all cases of the preservation proof. In all proofs where R' and M' are well typed, $\forall i \in dom(\Psi).(C_\varnothing \vdash M'(i) : \Psi(i))$, $\forall i \in dom(\Theta).(C_\varnothing \vdash \Theta(i) = \langle \tau_{Mem}, \tau_{resource} \rangle).M(i) : \text{Int}(0)$ or $M(i) : \text{Int}((1))$ If $M(i) : \text{Int}(0)$, then $R(i) : \langle \tau_{Mem}, \tau_{resource} \rangle$. If $M(i) : \text{Int}(1)$, then $R(i) : \langle \tau_{Mem} \rangle$. $C'_e \vdash e' : \tau$ so by (T_RMe), $C' \vdash (R', M', e' : \tau)$.

B.2 Proof of Preservation

If $\Psi_e; \Theta; \Delta; \Gamma \vdash e : \tau$, $\Psi_{spare}, \Psi_e; \Theta; \Delta; \Gamma \vdash (R, M, e : \tau)$ and $(R, M, e) \rightarrow (R', M', e')$, then $\Psi'_e; \Theta'; \Delta; \Gamma \vdash e' : \tau$ and $\Psi_{spare}, \Psi'_e; \Theta'; \Delta; \Gamma \vdash (R', M', e' : \tau)$.

Prove by induction on the type derivation. The cases below omit most of the congruence rule cases, because the proofs for these all look more or less alike. See the load, store, and createlock cases for examples of the proofs for congruence rule cases.

1. Case (T_LOAD): $C \vdash (R, M, e : \tau)$ and $C_e \vdash e : \tau = C_{ptr}, C_{Mem} \vdash \text{load}(e_{\text{ptr}}, e_{\text{Mem}}) : \text{lin}\langle \tau_1, \text{Mem}(I, \tau_1) \rangle$ where $C_{ptr} = \varnothing; \varnothing; \Delta; \overset{\text{non}}{\Gamma}_{e_{\text{ptr}}} \vdash e_{\text{ptr}} : \text{Int}(I)$ and $C_{Mem} = \{I \mapsto \tau_1\}; \varnothing; \Delta; \overset{\text{non}}{\Gamma}_{e_{\text{Mem}}} \vdash e_{\text{Mem}} : \text{Mem}(I, \tau_1)$. The evaluation rules that let $e \rightarrow e'$ are (E_LOAD) and the congruence rules for load.

   Using (E_LOAD), $C_{ptr} \vdash i : \text{Int}(I)$, $C_{Mem} \vdash \text{fact} : \text{Mem}(I, \tau_1)$, and

$(R', M', e') = (R, M, \text{lin}\langle M(i), \text{fact}\rangle)$. By Weakening for Terms $C_e = C_{ptr}, C_{Mem}$ and $C_{Mem} \subseteq C_e$ and $C_{Mem} \vdash \text{fact} : \text{Mem}(I, \tau_1)$ so $C_e = \{I \mapsto \tau_1\}; \varnothing; \Delta; \Gamma \vdash \text{fact} : \text{Mem}(I, \tau_1)$. Since M is well typed, $C_e \vdash M(i) : \Psi_e(i)$ so $C_e \vdash M(i) : \tau_1$. Ψ_e and Θ are unchanged by (E_LOAD) so $C'_e \vdash \text{lin}\langle M(i), \text{fact}\rangle : \text{lin}\langle \tau_1, \text{Mem}(I, \tau_1)\rangle$ so $C'_e \vdash e' : \tau$. R and M are unchanged so (FINAL-STEP) applies and $C' \vdash (R', M', e' : \tau)$.

Using $(congruence\, rule) E[e] = \text{load}(e, e_{\text{Mem}})$, $(R, M, e_{\text{ptr}}) \rightarrow (R', M', e'_{\text{ptr}})$. Since $C_{ptr} = \varnothing; \varnothing; \Delta; \Gamma_{e_{\text{ptr}}} \vdash e_{\text{ptr}} : \text{Int}(I)$, $\Psi_{spare}; \Theta; \Delta; \Gamma_{e_{\text{ptr}}} \vdash (R, M, e_{\text{ptr}} : \text{Int}(I))$. By induction, $C'_{ptr} = \Psi'_{e_{\text{ptr}}}; \Theta'; \Delta; \Gamma_{e_{\text{ptr}}} \vdash e'_{\text{ptr}} : \text{Int}(I)$ and $\Psi'_{spare}, \Psi'_{e_{\text{ptr}}}; \Theta'; \Delta; \Gamma_{e_{\text{ptr}}} \vdash (R', M', e'_{\text{ptr}} : \text{Int}(I))$. Since $C = C_{ptr}, C_{Mem}$, and $C_{ptr} \subseteq C'_{ptr}$, by the split subset lemma, there is a C'_{Mem} such that $C_{Mem} \subseteq C'_{Mem}$ and $C_{ptr}, C_{Mem} \subseteq C'_{ptr}, C'_{Mem} = C'_e$. By Weakening for Terms $C'_{Mem} \vdash e_{\text{Mem}} : \text{Mem}(I, \tau_1)$. By (T_LOAD) $C'_e \vdash \text{load}(e'_{\text{ptr}}, e_{\text{Mem}}) : \text{lin}\langle \tau_1, \text{Mem}(I, \tau_1)\rangle$ so $C'_e \vdash e' : \tau$. R' and M' are well typed so (FINAL-STEP) applies and $C' \vdash (R', M', e' : \tau)$.

Using $(congruence\, rule) E[e] = \text{load}(v_{\text{ptr}}, e)$, $(R, M, e_{\text{Mem}}) \rightarrow (R', M', e'_{\text{Mem}})$. $C_2 = \{I \mapsto \tau_1\}; \varnothing; \Delta; \Gamma_{e_{\text{Mem}}} \vdash e_{\text{Mem}} : \text{Mem}(I, \tau_1)$ and $\Psi_{spare}, \{I \mapsto \tau_1\}; \Theta; \Delta; \Gamma_{e_{\text{Mem}}} \vdash (R, M, e_{\text{Mem}} : \text{Mem}(I, \tau_1))$. By induction, $C'_{mem} = \Psi'_{e_{\text{Mem}}}; \Theta'; \Delta; \Gamma_{e_{\text{Mem}}} \vdash e'_{\text{Mem}} : \text{Mem}(I, \tau_1)$ and $\Psi'_{spare}, \Psi'_{e_{\text{Mem}}}; \Theta'; \Delta; \Gamma_{e_{\text{Mem}}} \vdash (R', M', e'_{\text{Mem}} : \text{Mem}(I, \tau_1))$. Since $C = C_{ptr}, C_{Mem}$, and $C_{Mem} \subseteq C'_{Mem}$, by the split subset lemma, there is a C'_{ptr} such that $C_{ptr} \subseteq C'_{ptr}$ and $C_{ptr}, C_{Mem} \subseteq C'_{ptr}, C'_{Mem} = C'_e$. By Weakening for Terms $C'_{ptr} \vdash v_{\text{ptr}} : \text{Int}(I)$. By (T_LOAD) $C'_e \vdash \text{load}(v_{\text{ptr}}, e'_{\text{Mem}}) : \text{lin}\langle \tau_1, \text{Mem}(I, \tau_1)\rangle$ so $C'_e \vdash e' : \tau$. R' and M' are well typed so (FINAL-STEP) applies and $C' \vdash (R', M', e' : \tau)$.

2. Case (T_STORE): $C \vdash (R, M, e : \tau)$ and $C \vdash e : \tau = C_{ptr}, C_{Mem} C_v \vdash \text{store}(e_{\text{ptr}}, e_{\text{Mem}}, e_{\text{v}}) : \text{Mem}(I, \tau_2)$ where

$C_{ptr} = \Psi_{e_{\text{ptr}}}; \Theta; \Delta; \Gamma_{e_{\text{ptr}}} \vdash e_{\text{ptr}} : \text{Int}(I)$, $C_{Mem} = \Psi_{e_{\text{Mem}}}; \Theta; \Delta; \Gamma_{e_{\text{Mem}}} \vdash e_{\text{Mem}} : \text{Mem}(I, \tau_1)$, $C_{3v} = \Psi_{e_{\text{v}}}; \Theta; \Delta; \Gamma_{e_{\text{v}}} \vdash e_{\text{v}} : \tau_2$, and $\Delta \vdash \tau_2 \overset{\text{non}}{:} 1$. The evaluation rules that let $e \rightarrow e'$ are (E_STORE) and the congruence rules for store.

Using (E_STORE), $C_{ptr} = \varnothing; \varnothing; \Delta; \overset{\text{non}}{\Gamma}_{e_{\text{ptr}}} \vdash i : \text{Int}(I)$,

$C_{Mem} = \{i \mapsto \tau_1\}; \varnothing; \Delta; \Gamma_{e_{\text{Mem}}} \vdash \text{fact} : \text{Mem}(I, \tau_1)$, $C_v = \varnothing; \varnothing; \Delta; \Gamma_{e_{\text{v}}} \vdash v : \tau_2$, and $(R', M', e') = (R, [i \mapsto v]M, \text{fact})$. $C'_e = \{i \mapsto \tau_2\}; \varnothing; \Delta; \Gamma$. By (T_FACT) $\{i \mapsto \tau_2\}; \varnothing; \Delta; \Gamma \vdash \text{fact} : \text{Mem}(I, \tau_2)$ so $C'_e \vdash e' : \tau$. $R' = R$, and M' is still well typed so (FINAL-STEP) applies and $C' \vdash (R', M', e' : \tau)$.

Using $(congruence\, rule) E[e] = \text{store}(e, e_{\text{mem}}, e_{\text{v}})$, $(R, M, e_{\text{ptr}}) \rightarrow (R', M', e'_{\text{ptr}})$, and $(R, M, e) \rightarrow (R', M', e')$. By Weakening for Terms, $C_e \vdash e_{\text{ptr}} : \text{Int}(I)$ and $C \vdash (R, M, e_{\text{ptr}} : \text{Int}(I))$. By induction, $C'_{ptr} = \Psi'_{e_{\text{ptr}}}; \Theta'; \Delta; \Gamma_{e_{\text{ptr}}} \vdash e'_{\text{ptr}} : \text{Int}(I)$ and $\Psi'_{spare}, \Psi'_{e_{\text{ptr}}}; \Theta'; \Delta; \Gamma_{e_{\text{ptr}}} \vdash (R', M', e'_{\text{ptr}} : \text{Int}(I))$. Since $C = C_{ptr}, C_{Mem/v}$, and $C_{ptr} \subseteq C'_{ptr}$, by the split subset lemma, there is a $C'_{Mem/v}$ such that $C_{Mem/v} \subseteq C'_{Mem/v}$ and $C_{ptr}, C_{Mem/v} \subseteq C'_{ptr}, C'_{Mem/v} = C'_e$. By Weakening for Terms $C'_{Mem/v} \vdash e_{\text{Mem}} : \text{Mem}(I, \tau_1)$ and $C'_{Mem/v} \vdash e_{\text{v}} : \tau_2$. By (T_STORE) $C'_e \vdash \text{store}(e'_{\text{ptr}}, e_{\text{Mem}}, e_{\text{v}}) : \text{Mem}(I, \tau_2)$ so $C'_e \vdash e' : \tau$. R' and M' are well typed so (FINAL-STEP) applies and $C' \vdash (R', M', e' : \tau)$.

Using $(congruence\, rule) E[e] = \text{store}(v_{\text{ptr}}, e, e_{\text{v}})$, $(R, M, e_{\text{Mem}}) \rightarrow (R', M', e'_{\text{Mem}})$, and $(R, M, e) \rightarrow (R', M', e')$. By induction, $C'_{Mem} = \Psi'_{e_{\text{Mem}}}; \Theta'; \Delta; \Gamma_{e_{\text{Mem}}} \vdash e'_{\text{Mem}} : \text{Mem}(I, \tau_1)$ and $\Psi'_{spare}, \Psi'_{e_{\text{Mem}}}; \Theta'; \Delta; \Gamma_{e_{\text{Mem}}} \vdash (R', M', e'_{\text{Mem}} : \text{Mem}(I, \tau_1))$. Since $C = C_{ptr/v}, C_{Mem}$, and $C_{Mem} \subseteq C'_{Mem}$, by the split subset lemma, there is a $C'_{ptr/v}$ such that $C_{ptr/v} \subseteq C'_{ptr/v}$ and $C_{ptr/v}, C_{Mem} \subseteq C'_{ptr/v}, C'_{Mem} = C'_e$. By Weakening for Terms $C'_{ptr/v} \vdash v_{\text{ptr}} : \text{Int}(I)$ and $C'_{ptr/v} \vdash e_{\text{v}} : \tau_2$. By (T_STORE) $C'_e \vdash \text{store}(e_{\text{ptr}}, e'_{\text{Mem}}, e_{\text{v}}) : \text{Mem}(I, \tau_2)$ so $C'_e \vdash e' : \tau$. R' and M' are well typed so (FINAL-STEP) applies and $C' \vdash (R', M', e' : \tau)$.

Using $(congruence\, rule) E[e] = \text{store}(v_{\text{ptr}}, \text{fact}, e)$, $(R, M, e_{\text{v}}) \rightarrow (R', M', e'_{\text{v}})$,

and $(R, M, e) \rightarrow (R', M', e')$. By induction, $C'_v = \Psi'_{e_\mathrm{v}}; \Theta'; \Delta; \Gamma_{e_\mathrm{v}} \vdash e'_\mathrm{v} : \tau_2$ and $\Psi'_{spare}, \Psi'_{e_\mathrm{v}}; \Theta'; \Delta; \Gamma_{e_\mathrm{v}} \vdash (R', M', e'_\mathrm{v} : \tau_2)$. Since $C = C_{ptr/Mem}, C_v$, and $C_v \subseteq C'_v$, by the split subset lemma, there is a $C'_{ptr/Mem}$ such that $C_{ptr/Mem} \subseteq C'_{ptr/Mem}$ and $C_{ptr/Mem}, C_v \subseteq C'_{ptr/Mem}, C'_v = C'_e$. By Weakening for Terms $C'_{ptr/Mem} \vdash v_\mathrm{ptr} : \mathrm{Int}(I)$ and $C'_{ptr/Mem} \vdash \mathrm{fact} : \mathrm{Mem}(I, \tau_1)$. By (T_STORE) $C'_e \vdash \mathrm{store}(e_\mathrm{ptr}, e_\mathrm{Mem}, e'_\mathrm{v}) : \mathrm{Mem}(I, \tau_2)$ so $C'_e \vdash e' : \tau$. R' and M' are well typed so (FINAL-STEP) applies and $C' \vdash (R', M', e' : \tau)$.

3. Case $(T\_CREATELOCK)$: $C \vdash (R, M, e : \tau)$ and

   $C_e \vdash e : \tau = C_{ptr}, C_{Mem}, C_{resource} \vdash \mathrm{create\_lock}(e_\mathrm{ptr}, e_\mathrm{Mem}, e_\mathrm{resource}) : \mathrm{Lock}(I, \tau_1)$ where $C_{ptr} = \varnothing; \Theta; \Delta; \Gamma_{e_\mathrm{ptr}} \vdash e_\mathrm{ptr} : \mathrm{Int}(I)$ and

   $C_{Mem} = \{I \mapsto \mathrm{Int}(0)\}; \varnothing; \Delta; \Gamma_{e_\mathrm{Mem}} \vdash e_\mathrm{Mem} : \mathrm{Mem}(I, \mathrm{Int}(0))$ and $C_{resource} \vdash e_\mathrm{resource} : \tau_1 \overset{\mathrm{lin}}{:} 0$. The evaluation rules that let $e \rightarrow e'$ are $(E\_CREATELOCK)$ and the congruence rules for create_lock.

   Using $(E\_CREATELOCK)$, $C_{ptr} \vdash i : \mathrm{Int}(I)$, $C_{Mem} \vdash \mathrm{fact} : \mathrm{Mem}(I, \mathrm{Int}(0))$, $C_{resource} \vdash v_r : \tau_1$, and $(R', M', e') = ([i \mapsto \langle \mathrm{fact}, v_r \rangle]R, M, \mathrm{lock})$. $C'_e = \{I \mapsto \mathrm{Int}(0)\}; \{I \mapsto \langle \mathrm{Mem}(I, \mathrm{Int}(0)), \tau_1 \rangle\}, \Theta; \Delta; \Gamma$. By (T_LOCK1), $C'_e \vdash \mathrm{lock} : \mathrm{Lock}(I, \tau_1)$ so $C'_e \vdash e' : \tau$. R' is still well typed and $M = M'$ so (FINAL-STEP) applies and $C' \vdash (R', M', e' : \tau)$.

   Using $(congruence\, rule) E[e] = \mathrm{create\_lock}(e, e_\mathrm{Mem}, e_\mathrm{resource})$,

   $(R, M, e_\mathrm{ptr}) \rightarrow (R', M', e'_\mathrm{ptr})$. $\varnothing; \varnothing; \Delta; \Gamma_{e_\mathrm{ptr}} \vdash e_\mathrm{ptr} : \mathrm{Int}(I)$ and

   $\Psi_{spare}; \Theta; \Delta; \Gamma_{e_\mathrm{ptr}} \vdash (R, M, e_\mathrm{ptr} : \mathrm{Int}(I))$. By induction,

   $C'_{ptr} = \Psi'_{e_\mathrm{ptr}}; \Theta'; \Delta; \Gamma_{e_\mathrm{ptr}} \vdash e'_\mathrm{ptr} : \mathrm{Int}(I)$ and $\Psi'_{spare}, \Psi'_{e_\mathrm{ptr}}; \Theta'; \Delta; \Gamma_{e_\mathrm{ptr}} \vdash (R', M', e'_\mathrm{ptr} : \mathrm{Int}(I))$. Since $C = C_{ptr}, C_{Mem/resource}$, and $C_{ptr} \subseteq C'_{ptr}$, by the split subset lemma, there is a $C'_{Mem/resource}$ such that $C_{Mem/resource} \subseteq C'_{Mem/resource}$ and $C_{ptr}, C_{Mem/resource} \subseteq C'_{ptr}, C'_{Mem/resource} = C'_e$. By Weakening for Terms $C'_{Mem/resource} \vdash e_\mathrm{Mem} : \mathrm{Mem}(I, \tau_1)$ and $C'_{Mem/resource} \vdash e_\mathrm{resource} : \tau_1$.

   By $(T\_CREATELOCK)$ $C'_e \vdash \mathrm{create\_lock}(e'_\mathrm{Ptr}, e_\mathrm{Mem}, e_\mathrm{resource}) : \mathrm{Lock}(I, \tau_1)$

so $C'_e \vdash e' : \tau$. R' and M' are well typed so (FINAL-STEP) applies and $C' \vdash (R', M', e' : \tau)$.

Using $(congruence\, rule) E[e] = \text{create\_ lock}(v_{\text{ptr}}, e, e_{\text{resource}})$, $(R, M, e_{\text{Mem}}) \rightarrow (R', M', e'_{\text{Mem}})$. $\{I \mapsto \text{Int}(0)\}; \varnothing; \Delta; \Gamma_{e_{\text{Mem}}} \vdash e_{\text{Mem}} : \text{Mem}(I, \text{Int}(0))$ and $\Psi_{spare}, \{I \mapsto \text{Int}(0)\}; \Theta; \Delta; \Gamma_{e_{\text{Mem}}} \vdash (R, M, e_{\text{Mem}} : \text{Mem}(I, \text{Int}(0)))$. By induction, $C'_{Mem} = \Psi'_{e_{\text{Mem}}}; \Theta'; \Delta; \Gamma_{e_{\text{Mem}}} \vdash e'_{\text{Mem}} : \text{Mem}(I, \text{Int}(0))$ and $\Psi'_{spare}, \Psi'_{e_{\text{Mem}}}; \Theta'; \Delta; \Gamma_{e_{\text{Mem}}} \vdash (R', M', e'_{\text{Mem}} : \text{Mem}(I, \text{Int}(0)))$. Since $C = C_{ptr/resource}, C_{Mem}$, and $C_{Mem} \subseteq C'_{Mem}$, by the split subset lemma, there is a $C'_{ptr/resource}$ such that $C_{ptr/resource} \subseteq C'_{ptr/resource}$ and $C_{ptr/resource}, C_{Mem} \subseteq C'_{ptr/resource}, C'_{Mem} = C'_e$. By Weakening for Terms $C'_{ptr/resource} \vdash v_{\text{ptr}} : \text{Int}(I)$ and $C'_{ptr/resource} \vdash e_{\text{resource}} : \tau_1$. By $(T\_CREATELOCK)$ $C'_e \vdash \text{create\_ lock}(e_{\text{Ptr}}, e'_{\text{Mem}}, e_{\text{resource}}) : \text{Lock}(I, \tau_1)$ so $C'_e \vdash e' : \tau$. R' and M' are well typed so (FINAL-STEP) applies and $C' \vdash (R', M', e' : \tau)$.

Using $(congruence\, rule) E[e] = \text{create\_ lock}(v_{\text{ptr}}, v_{\text{Mem}}, e)$, $(R, M, e_{\text{resource}}) \rightarrow (R', M', e'_{\text{resource}})$. $\Psi_{e_{\text{resource}}}; \Theta; \Delta; \Gamma_{e_{\text{resource}}} \vdash e_{\text{resource}} : \tau_1$ and $\Psi_{spare}, \Psi_{e_{\text{resource}}}; \Theta; \Delta; \Gamma_{e_{\text{resource}}} \vdash (R, M, e_{\text{resource}} : \tau_1)$. By induction, $C'_{resource} = \Psi'_{e_{\text{resource}}}; \Theta'; \Delta; \Gamma_{e_{\text{resource}}} \vdash e'_{\text{resource}} : \tau_1$ and $\Psi'_{spare}, \Psi'_{e_{\text{resource}}}; \Theta'; \Delta; \Gamma_{e_{\text{resource}}} \vdash (R', M', e'_{\text{resource}} : \tau_1)$. Since $C = C_{ptr/Mem}, C_{resource}$, and $C_{resource} \subseteq C'_{resource}$, by the split subset lemma, there is a $C'_{ptr/Mem}$ such that $C_{ptr/Mem} \subseteq C'_{ptr/Mem}$ and $C_{ptr/Mem}, C_{resource} \subseteq C'_{ptr/Mem}, C'_{resource} = C'_e$. By Weakening for Terms $C'_{ptr/Mem} \vdash v_{\text{ptr}} : \text{Int}(I)$ and $C'_{ptr/Mem} \vdash e_{\text{Mem}} : \text{Mem}(I, \text{Int}(0))$. By $(T\_CREATELOCK)$ $C'_e \vdash \text{create\_ lock}(e_{\text{Ptr}}, e_{\text{Mem}}, e'_{\text{resource}}) : \text{Lock}(I, \tau_1)$ so $C'_e \vdash e' : \tau$. R' and M' are well typed so (FINAL-STEP) applies and $C' \vdash (R', M', e' : \tau)$.

4. Case $(T\_ACQUIRE)$: $C \vdash (R, M, e : \tau)$ and $C_e \vdash e : \tau = \text{acquire}(e_{\text{Lock}}, e_{\text{ptr}}) : \tau$

where $C_e \vdash e_{\text{Lock}} : \text{Lock}(I, \tau)$ and $C_e \vdash e_{\text{ptr}} : \text{Int}(I)$. The evaluation rules that let $e \rightarrow e'$ are ($E\_ACQUIRE1$), ($E\_ACQUIRE2$), and the congruence rules for acquire.

Using ($E\_ACQUIRE1$), $R(i) = \langle \text{fact}, v_{\text{resource}} \rangle$, $M(i) = 0$,

$C_e = \{I \mapsto \text{Int}(0)\}; \{I \mapsto \langle \text{Mem}(I, \text{Int}(0)), \tau \rangle\}; \Delta; \Gamma_{e_{\text{Lock}}} \vdash \text{lock} : \text{Lock}(I, \tau)$, $C_e \vdash i : \text{Int}(I)$, and

$(R', M', e') = ([i \mapsto \langle \text{fact} \rangle]R, [i \mapsto 1]M, v_{\text{resource}})$. Because R and M were well typed, by (T_RMe) $C_e \vdash v_{\text{resource}} : \tau$.

$C'_e = \{I \mapsto \text{Int}(1)\}; \{I \mapsto \langle \text{Mem}(I, \text{Int}(1)), \tau \rangle\}; \Delta; \Gamma \vdash v_{\text{resource}} : \tau$ so $C'_e \vdash e' : \tau$.

R' and M' are well typed so (FINAL-STEP) applies and $C' \vdash (R', M', e' : \tau)$.

Using ($E\_ACQUIRE2$), $(R', M', e') = (R, M, e)$. $C_e \vdash e : \tau$ so $C_e = C'_e \vdash e' : \tau$.

R' and M' are well typed so (FINAL-STEP) applies and $C' \vdash (R', M', e' : \tau)$.

The proof for the congruence rules follows the same format as the congruence rule proofs shown in the load, store, and createlock cases.

5. Case ($T\_RELEASE$): $C \vdash (R, M, e : \tau)$ and

   $C_e \vdash e : \tau = C_1, C_2 \vdash \text{release}(e_{\text{Lock}}, e_{\text{ptr}}, e_{\text{resource}}) :^{\text{non}} \langle\rangle$, $C_1 \vdash e_{\text{Lock}} : \text{Lock}(I, \tau_1)$, $C_1 \vdash e_{\text{ptr}} : \text{Int}(I)$, and $C_2 \vdash e_{\text{resource}} : \tau_1$. The evaluation rules that let $e \rightarrow e'$ are ($E\_RELEASE1$), ($E\_RELEASE2$) and the congruence rules for release.

   Using ($E\_RELEASE1$), $R(i) = \langle \text{fact} \rangle$, $M(i) = 1$,

   $C_1 = \{i \mapsto \text{Int}(1)\}; \{i \mapsto \langle \text{Mem}(I, \text{Int}(1)), \tau_1 \rangle\}; \Delta; \overset{\text{non}}{\Gamma}_1 \vdash \text{lock} : \text{Lock}(I, \tau_1)$, $i : \text{Int}(I)$, and $C_2 = \Psi_2; \Theta; \Delta; \Gamma_2 \vdash v_r : \tau_1$, and

   $(R', M', e') = ([i \mapsto \langle \text{fact}, v_r \rangle]R, [i \mapsto 0]M, {}^{\text{non}}\langle\rangle)$. $C_\varnothing \vdash^{\text{non}} \langle\rangle :^{\text{non}} \langle\rangle$ so $C'_e \vdash e' : \tau$. $\Psi'_{spare} = \{i \mapsto \text{Int}(1)\}, \Psi_{spare}$, $\Theta' = \{i \mapsto \langle \text{Mem}(I, \text{Int}(1)), \tau_1 \rangle\}$, R' and M' are well typed so (FINAL-STEP) applies and $C' \vdash (R', M', e' : \tau)$.

   Using ($E\_RELEASE2$), $(R', M', e') = (R, M, e)$. $C_e \vdash e : \tau$ so $C_e = C'_e \vdash e' : \tau$.

   R' and M' are well typed so (FINAL-STEP) applies and $C' \vdash (R', M', e' : \tau)$.

   The proof for the congruence rules follows the same format as the congruence

rule proofs shown in the load, store, and createlock cases.

6. Case (T_VAR), (T_FACT), (T_LOCK), (T_INT), (T_TABS), (T_ABS), (T_PACK): $C \vdash (R, M, e : \tau)$ and $C_e \vdash e : \tau$. There are no evaluation rules for e.

7. Case (T_TAPP): $C \vdash (R, M, e : \tau)$ and $C_e \vdash e : \tau = C_e \vdash e_1\tau_2 : [\alpha \mapsto \tau_2]\tau_1$ where $C_e \vdash e_1 : \forall\alpha : K.\tau_1$ and $C_e \vdash \tau_2 : K$. The evaluation rules that let $e \rightarrow e'$ are ($E\_TAPPTABS$) and ($congruence\,rule$)$E[e] = e\tau$.

   Using ($E\_TAPPTABS$), $e_1 = \Lambda\alpha : K; B.v_1 : \forall\alpha : K.\tau_1$ and $e' = [\alpha \mapsto \tau_2]v_1$. By Inversion (TABS), $C_e, \alpha : K \vdash v_1 : \tau_1$. By Term Substiution, $\Psi_e; \Theta; \Delta; [\alpha \mapsto \tau_2]\Gamma \vdash [\alpha \mapsto \tau_2]v_1 : [\alpha \mapsto \tau_2]\tau_1$. We can use α-renaming to ensure that $\alpha$is not found in Γ. Therefore $\Psi_e; \Theta; \Delta; \Gamma \vdash [\alpha \mapsto \tau_2]v_1 : [\alpha \mapsto \tau_2]\tau_1 = \tau'$ and $\tau = \tau'$ so $C'_e \vdash e' : \tau$. M and R are unchanged so (FINAL-STEP) applies and $C' \vdash (R', M', e' : \tau)$.

   The proof for the congruence rule follows the same format as the congruence rule proofs shown in the load, store, and createlock cases.

8. Case (T_TUPLE). Only congruence rules apply and they follow the same format as the congruence rule proofs shown in the load, store, and createlock cases.

9. Case (T_APP): $C \vdash (R, M, e : \tau)$ and $C_e \vdash e : \tau = C_1, C_2 \vdash e_1e_2 : \tau$ where $C_1 \vdash e_1 : \tau_1 \xrightarrow{\phi} \tau$ and $C_2 \vdash e_2 : \tau_1$. The evaluation rules that let $e \rightarrow e'$ are (E_APPABS) and the congruence rules for app.

   Using (E_APPABS), $C_1 \vdash \lambda x : \tau_1 \xrightarrow{\phi} e_{11} : \tau_1 \xrightarrow{\phi} \tau$, $C_2 \vdash v : \tau_1$, $(R', M', e') = (R, M, [x \mapsto v]e_{11})$. By Term Substitution, $C_e, x : \tau_1 \vdash e_{11} : \tau$. $C_e, x : \tau_x \vdash [x \mapsto v]e_{11} : [\tau_1 \mapsto \tau_1]\tau$. This does not change τ so $C'_e \vdash e' : \tau$. M and R are unchanged so (FINAL-STEP) applies and $C' \vdash (R', M', e' : \tau)$.

   The proof for the congruence rules follows the same format as the congruence rule proofs shown in the load, store, and createlock cases.

10. Case (T_LET): $C \vdash (R, M, e : \tau)$ and $C_e \vdash e : \tau = C_1, C_2 \vdash \text{let } \langle \overrightarrow{x_i} \rangle = e_a \text{ in } e_b : \tau$ where $C_1 \vdash e_a : \langle \overrightarrow{\tau_i} \rangle$ and $C_2, \overrightarrow{x : \tau} \vdash e_b : \tau$. The evaluation rules that let $e \rightarrow e'$ are (E_LET) and the congruence rules for let.

    Using (E_LET), $C_1 \vdash \langle v_1, \ldots, v_n \rangle : \langle \tau_1, \ldots, \tau_n \rangle$ and $(R', M', e') = (R, M, [x_1 \mapsto v_1, \ldots, x_n \mapsto v_n]e_b)$. By Term Substitution, $C'_e, \overrightarrow{x : \tau} \vdash [x_1 \mapsto v_1, \ldots, x_n \mapsto v_n]e_b : [\tau_1 \mapsto \tau_1, \ldots, \tau_n \mapsto \tau_n]\tau$. This does not change τ so $C'_e \vdash e' : \tau$. M and R are unchanged so (FINAL-STEP) applies and $C' \vdash (R', M', e' : \tau)$.

    The proof for the congruence rules follows the same format as the congruence rule proofs shown in the load, store, and createlock cases.

11. Case (T_FIX): $C \vdash (R, M, e : \tau)$ and $C_e \vdash e : \tau = (\text{fix } x : \tau.v) : \tau$ where $C_e \vdash \tau :\overset{\phi}{i}$ and $C_e, x : \tau \vdash v : \tau$. The evaluation rule that lets $e \rightarrow e'$ is (E_FIX). Using (E_FIX), $(R', M', e') = (R, M, [x \mapsto \text{fix } x : \tau.v]v)$. By Term Substitution, $C'_e, x : \tau \vdash [x \mapsto \text{fix } x : \tau.v]v : [\tau \mapsto \tau]\tau$. This does not change τ so $C'_e \vdash e' : \tau$. M and R are unchanged so (FINAL-STEP) applies and $C' \vdash (R', M', e' : \tau)$.

12. Case (T_UNPACK): $C \vdash (R, M, e : \tau)$ and $C_e \vdash e : \tau = C_1, C_2 \vdash \text{unpack } \alpha, x = e_1 \text{ in } e_2 : \tau$ where $C_1 \vdash e_1 : \exists \alpha : K.\tau_1$ and $C_2, \alpha : K, x : \tau_1 \vdash e_2 : \tau$. The evaluation rules that let $e \rightarrow e'$ are (E_UNPACK) and the congruence rules for unpack.

    Using (E_UNPACK), $C_1 \vdash (\text{pack}[\tau_2, v_1] \text{ as } \exists \alpha : K.\tau_1) : \exists \alpha : K.\tau_1$ and $(R', M', e') = (R, M, [\alpha \mapsto \tau_2, x \mapsto v_1]e_2)$. $C_1 \vdash \tau_2 : K$ $C_1 \vdash v_1 : [\alpha \mapsto \tau_2]\tau_1$, $C_1 \vdash \exists \alpha : K.\tau_1 :\overset{\phi}{i}$. By Term Substitution, $[\alpha \mapsto \tau_2, x \mapsto v_1]C'_e, x : \tau_1 \vdash [\alpha \mapsto \tau_2, x \mapsto v_1]e_2 : [\alpha \mapsto \tau_2, x \mapsto v_1]\tau$ becomes $C'_e, x : [\alpha \mapsto \tau_2]\tau_1 \vdash [\alpha \mapsto \tau_2, x \mapsto v_1]e_2 : [\alpha \mapsto \tau_2]\tau$. By alpha-renaming, we can ensure $\alpha$is not found in τso $C'_e \vdash [\alpha \mapsto \tau_2, x \mapsto v_1]e_2 : \tau$ and $C'_e \vdash e' : \tau$. M and R are unchanged so (FINAL-STEP) applies and $C' \vdash (R', M', e' : \tau)$.

    The proof for the congruence rules follows the same format as the congruence

rule proofs shown in the load, store, and createlock cases.

13. Case (T_IFE): $C \vdash (R, M, e : \tau)$ and $C_e \vdash e : \tau = C_1, C_2 \vdash \text{if}\, e_1 \,\text{then}\, e_2 \,\text{else}\, e_3 : \tau$ where $C_1 \vdash e_1 : \text{bool}$ and $C_2, \text{true} \vdash e_2 : \tau$, and $C_3, \text{false} \vdash e_2 : \tau$. The evaluation rules that let $e \rightarrow e'$ are (E_IF1), (E_IF2) and the congruence rules for if.

    Using (E_IF1), $C_1 \vdash e_1 = \text{true} : \text{bool}$ and $(R', M', e') = (R, M, e_2)$. By Weakening for Terms $C_e = C_1, C_2, C_3$ and $C_2 \subseteq C_e$ and $C_2 \vdash e_2 : \tau$ so $C_e = \Psi_e; \varnothing; \Delta; \Gamma \vdash e_2 : \tau$ so $C'_e \vdash e' : \tau$. M and R are unchanged so (FINAL-STEP) applies and $C' \vdash (R', M', e' : \tau)$.

    Using (E_IF2), $C_1 \vdash e_1 = \text{false} : \text{bool}$ and $(R', M', e') = (R, M, e_3)$. By Weakening for Terms $C_e = C_1, C_2, C_3$ and $C_3 \subseteq C_e$ and $C_3 \vdash e_3 : \tau$ so $C_e = \Psi_e; \varnothing; \Delta; \Gamma \vdash e_3 : \tau$ so $C'_e \vdash e' : \tau$. M and R are unchanged so (FINAL-STEP) applies and $C' \vdash (R', M', e' : \tau)$.

    The proof for the congruence rules follows the same format as the congruence rule proofs shown in the load, store, and createlock cases.

B.3 Progress Lemmas

LEMMA [CANONICAL FORMS]

Suppose v is a closed, well-typed expression: $C \vdash v : \tau \,\text{for some}\, \tau$, $C = \Psi; \Theta; \varnothing; \varnothing$. The first 6 cases only differ from [14] by the elimination of $B; \mathit{limit}$.

1. If $C \vdash v : \tau_1 \xrightarrow{\phi} \tau_2$, then $v = \lambda x : \tau_1 \xrightarrow{\phi} e$.
2. If $C \vdash v : \langle \tau_1, \ldots \tau_n \rangle$, then $v = \langle v_1, \ldots, v_n \rangle$.
3. If $C \vdash v : \forall \alpha : K.\tau$, then $v = \Lambda \alpha : K.v_0$.
4. If $C \vdash v : \exists \alpha : K.\tau_2$, then $v = \text{pack}[\tau_1, v_0] \,\text{as}\, \exists \alpha : K.\tau_2$.
5. If $C \vdash v : \text{Int}(I)$, then $v = i$.

6. If $C \vdash v : \mathrm{Mem}(I, \tau)$ then $v = \mathrm{fact}$.

7. If $C \vdash v : \mathrm{Lock}(I, \tau)$ then $v = \mathrm{lock}$.

Proof:

Only one type checking rule can prove $C \vdash v : \mathrm{Lock}(I, \tau)$. By (*T_Lock*), we know $v = \mathrm{lock}$.

B.4 Proof of Progress

If (R, M, e) is closed and well-typed ($C_{RMe} \vdash (R, M, e : \tau)$ for some τ and $C_{RMe} = \Psi_{RMe}; \Theta_{RMe}; \varnothing; \varnothing$), then either e is a value or else there is some (R', M', e') so that $(R, M, e) \rightarrow (R', M', e')$.

Observe that by the rule for typing (R, M, e), we know that $\Psi_{RMe} = \Psi_{spare}, \Psi$ and $\Theta_{RMe} = \Theta$ and $C \vdash e : \tau$, where $C = \Psi; \Theta; \varnothing; \varnothing$. Now prove that (R, M, e) steps, by induction on the derivation of $C \vdash e : \tau$ (holding M, Ψ_{RMe}, and Θ_{RMe} fixed throughout the induction as C, e, and τ vary). Several cases have not changed since [14] and have been omitted. Store, load, and the congruence rules are mostly unchanged and are shown here for consistency.

Proof :

First, if $e = E[e_0]$ where e_0 is not a value, then by inspection of the type checking rules, $C_0 \vdash e_0 : \tau_0$, where $C_0 = \Psi_0; \Theta_0; \varnothing; \varnothing$ and $\Psi_{Me} = \Psi_{0-spare}, \Psi_0$ $\Theta_{Me} = \Theta_0$, so by induction, $(R, M, e_0) \rightarrow (R', M', e_0')$, so $(R, M, e) = (R, M, E[e_0]) \rightarrow (R', M', E[e_0'])$. For all other cases where e is not a value:

1. $e = \mathrm{load}(v_{\mathrm{ptr}}, v_{\mathrm{Mem}})$,

   where (*T_LOAD*) $\quad \frac{C_1 \vdash v_{\mathrm{ptr}} : \mathrm{Int}(I) \quad C_2 \vdash v_{\mathrm{Mem}} : \mathrm{Mem}(I,\tau)}{C_1, C_2 \vdash \mathrm{load}(v_{\mathrm{ptr}}, v_{\mathrm{Mem}}) : {}^{\mathrm{lin}}\langle \tau, \mathrm{Mem}(I,\tau)\rangle}$.

   By canonical forms, $v_{\mathrm{ptr}} = i_{ptr}$, and $v_{Has} = \mathrm{fact}$. By inversion, $C_1 \vdash i_{ptr} : Int(i_{ptr})$ and $C_2 = C_2', i_{has} \mapsto \tau \vdash \mathrm{fact} : \mathrm{Mem}(i_{has}, \tau)$. Together, these imply $i_{ptr} = i_{has}$. Since $\Psi(i_{has}) = \tau$ and memory M is well-typed under $C_{Me} =$

$\Psi_{Me}; ...$, where $\Psi_{Me} = \Psi_{spare}, \Psi$, we know $i_{has} \mapsto \tau \in \Psi_{Me}$, so $M(i_{has})$ exists, so $M(i_{ptr})$ exists, so e steps by (E_LOAD).

2. $e = \text{store}(v_{\text{ptr}}, v_{\text{Mem}}, v_{\text{v}})$,

   where (T_STORE) $\dfrac{C_2 \vdash v_{\text{Mem}} : \text{Mem}(I, \tau_1) \quad C_3 \vdash v_{\text{v}} : \tau_2 \quad C_1 \vdash v_{\text{ptr}} : \text{Int}(I) \quad C_1, C_2, C_3 \vdash \tau_2 : \overset{\text{non}}{1}}{C_1, C_2, C_3 \vdash \text{store}(v_{\text{ptr}}, v_{\text{Mem}}, v_{\text{v}}) : \text{Mem}(I, \tau_2)}$.

   By canonical forms, $v_{\text{ptr}} = i_{ptr}$, and $v_{Has} = \text{fact}$, so e steps by (E_STORE).

3. $e = \text{create\_ lock}(v_{\text{ptr}}, v_{\text{Mem}}, v_{\text{resource}})$,

   where $(T\_CREATELOCK)\frac{C_1 \vdash e_{\text{ptr}} : \text{Int}(I) \quad C_2 \vdash e_{\text{Mem}} : \text{Mem}(I, \text{Int}(0)) \quad C_3 \vdash e_{\text{resource}} : \tau : \overset{\text{lin}}{0}}{C_1, C_2, C_3 \vdash \text{create\_ lock}(e_{\text{ptr}}, e_{\text{Mem}}, e_{\text{resource}}) : \text{Lock}(I, \tau)}$

   By canonical forms, $v_{\text{ptr}} = i : \text{Int}(I)$, and $v_{\text{Mem}} = \text{fact} : \text{Mem}(I, \text{Int}(0))$ so e steps by $(E\_CREATELOCK)$.

4. $e = \text{acquire}(v_{\text{Lock}}, v_{\text{ptr}})$,

   where $(T\_ACQUIRE)\frac{C \vdash e_{\text{Lock}} : \text{Lock}(I, \tau), e_{\text{ptr}} : \text{Int}(I)}{C \vdash \text{acquire}(e_{\text{Lock}}, e_{\text{ptr}}) : \tau}$

   By canonical forms, $v_{\text{Lock}} = \text{lock} : \text{Lock}(I, \tau)$, and $v_{\text{ptr}} = i : \text{Int}(I)$. Since R and M are well typed under C_{RMe}, $\Theta_{RMe}(I) = \langle \text{Mem}(I, \text{Int}(J)), \tau \rangle$ and $\Psi_{RMe}(I) = \text{Int}(J)$ where J equals 0 or 1. If $J = 0$, then $R(I) = \langle \text{fact}, v_{\text{resource}} \rangle$ and $M(I) = 0$ where $v_{\text{resource}} : \tau$. Therefore e steps by $(E\_ACQUIRE1)$. If $J = 1$, then $R(I) = \langle \text{fact} \rangle$ and $M(I) = 1$. Therefore e steps by $(E\_ACQUIRE2)$.

5. $e = \text{release}(v_{\text{Lock}}, v_{\text{ptr}}, v_{\text{resource}})$,

   where $(T\_RELEASE)\frac{C_1 \vdash e_{\text{Lock}} : \text{Lock}(I, \tau) \quad e_{\text{ptr}} : \text{Int}(I) \quad C_2 \vdash e_{\text{resource}} : \tau}{C_1, C_2 \vdash \text{release}(e_{\text{Lock}}, e_{\text{ptr}}, e_{\text{resource}}) : \text{non}\langle\rangle}$

   By canonical forms, $v_{\text{Lock}} = \text{lock} : \text{Lock}(I, \tau)$, $v_{\text{ptr}} = i : \text{Int}(I)$, and $v_{\text{resource}} : \tau$. Since R and M are well typed under C_{RMe}, $\Theta_{RMe}(I) = \langle \text{Mem}(I, \text{Int}(J)), \tau \rangle$ and $\Psi_{RMe}(I) = \text{Int}(J)$ where J equals 0 or 1. If $J = 0$, then $R(I) = \langle \text{fact}, v_{\text{resource}} \rangle$ where $v_{\text{resource}} : \tau$ and $M(I) = 0$. Therefore e steps by $(E\_RELEASE2)$. If $J = 1$, then $R(I) = \langle \text{fact} \rangle$ and $M(I) = 1$. Therefore e steps by $(E\_RELEASE1)$.

B.5 Erasure Lemmas

LEMMA [ERASE-TERM-SUBSTITUTION]

$\text{erase}([x \mapsto v]e) = [x \mapsto \text{erase}(v)]\,\text{erase}(e)$. Proof by induction on the expression e.

LEMMA [ERASE-TYPE-SUBSTITUTION]

$\text{erase}([\alpha \mapsto \tau]e) = \text{erase}(e)$. Proof by induction on the expression e.

LEMMA [VALUE-ERASE-VALUE]

A value will always erase to a value. Proof by induction on the values. The proof is the same as in[14] with the addition of a lock case and the changed tuple case.

1. $\text{erase}(\text{lock}) = \langle\rangle$ by (ER_LOCK).
2. $\text{erase}(\phi\langle v_1, \cdots, v_n\rangle) = \langle v_m, \cdots, v_n\rangle$ for $v_j : t :\overset{\phi}{i}, i > 0$ by (ER_TUPLE). By induction, each remaining value erases to another value.

All the value erase rules produce either values or the erase of another value.

LEMMA [ZERO-ERASE-VALUE]

If $C \vdash v : \tau$then $\text{erase}(v : t :\overset{\phi}{0}) = \langle\rangle$. A value with kind $\overset{\phi}{0}$ will erase to $\langle\rangle$. Proof by induction on the values. The proof is the same as in [14] with the addition of a lock case and the changed tuple case.

1. By (ER_LOCK) $\text{erase}(\text{fact}) = \langle\rangle$.
2. By (ER_TUPLE) $\text{erase}(\phi\langle v_1, \cdots, v_n\rangle)$ (where $v_i : t :\overset{\phi}{0}$) $= \langle\rangle$.

LEMMA [TYPE-SUBSTITUTION-VALUE]

$[\alpha \mapsto \tau]v$ is a value. Proof by induction on the values.

LEMMA [TERM-SUBSTITUTION-VALUE]

$[x \mapsto v]v$ and $[x \mapsto \text{fix}\, x.v]v$ are values. Proof by induction on the values.

c Set

c is the set of expressions e which erase to a value.

$$c = i \mid b \mid c\tau \mid \phi\langle \overrightarrow{c} \rangle \mid \lambda x : \tau \xrightarrow{\phi} e$$

$$\mid \Lambda\alpha : K.v \mid \text{pack}[\tau_1, c] \,\text{as}\, \exists\alpha : K.\tau_2 \mid \text{fix}\, x : \tau \overset{\phi}{:} 0 \,.\text{v} \mid \text{fact} \mid \text{lock}$$

LEMMA [C-ERASE-VALUE]

$\text{erase}(c) = u$. Proof by induction on c. The only new cases are lock and tuple. The other cases are shown in [14].

1. By (ER_LOCK), $\text{erase}(\text{lock}) = \langle\rangle$.
2. By (ER_TUPLE), $\text{erase}(\phi\langle \overrightarrow{c} \rangle) = \langle \overrightarrow{v} \rangle$. By induction all the terms of the erased tuple are either values or another $\text{erase}(c)$.

$\text{erase}(c)$ is either a value or another $\text{erase}(c)$.

LEMMA [NON-C-ERASE-NON-VALUE]

If $e \neq c$ then, $\text{erase}(e) \neq u$. Proof by induction on the expressions. The new cases are createlock, acquire, and release. The other cases are shown in [14].

1. By $(ER\_CREATELOCK)$, $\text{erase}(\text{create\_ lock}(e_{\text{ptr}}, e_{\text{Mem}}, e_{\text{resource}})) =$
   $\text{create\_ lock}(\text{erase}(e_{\text{ptr}}), \text{erase}(e_{\text{Mem}}), \text{erase}(e_{\text{resource}}))$
   which is not a value.
2. By $(ER\_ACQUIRE)$, $\text{erase}(\text{acquire}(e_{\text{Lock}}, e_{\text{ptr}})) =$
   $\text{acquire}(\text{erase}(e_{\text{Lock}}), \text{erase}(e_{\text{ptr}}))$ which is not a value.

3. By $(ER\_RELEASE)$, $\text{erase}(\text{release}(e_{\text{Lock}}, e_{\text{ptr}}, e_{\text{v}})) =$ $\text{release}(\text{erase}(e_{\text{Lock}}), \text{erase}(e_{\text{ptr}}), \text{erase}(e_{\text{v}}))$ which is not a value.

All expressions which are not in c erase to non values (or do not erase).

csize function

The cases of csize are unchanged from [14]. Only the parts relevant to $\lambda^{concurrent}$ are shown here. Define $\text{csize}(c)$ as:

- $\text{csize}(c\tau) = 1 + \text{csize}(c)$
- $\text{csize}(\phi\langle\overrightarrow{c}\rangle) = sum(\text{csize}(c_i))$
- $\text{csize}(\text{pack}[\tau_1, c] \text{ as } \exists\alpha : K.\tau_2) = 1 + \text{csize}(c)$
- $\text{csize}(\text{fix}\, x : \tau \overset{\phi}{:0} .\text{v}) = 1$
- $\text{csize}(v) = 0$

LEMMA [C-SIZE-DECREASES]

If $C \vdash c : \tau$, where C has an empty Δ and Γ, and (M, c) steps, then there is some c' so that $(M, c) \xrightarrow{+} (M, c')$ and $\text{csize}(c) > \text{csize}(c')$. Proof by induction on the possible evaluation rules. In all cases we reach c' by taking only one step. The cases are unchanged from [14]. Only the parts relevant to $\lambda^{concurrent}$ are shown here.

1. $(\Lambda\alpha : K.v)\tau \xrightarrow{(E\_TAPPTABS)} [\alpha \mapsto \tau]v$. $\text{csize}((\Lambda\alpha : K.v)\tau) = 1$. By Type-Substitution-Value, $[\alpha \mapsto \tau]v$ is a value so $\text{csize}([\alpha \mapsto \tau]v) = 0$.
2. $\text{fix}\, x : t \overset{\phi}{:0} .v \xrightarrow{(E\_FIX)} [x \mapsto \text{fix}\, x : t \overset{\phi}{:0} .v]v$. By Term-Substitution-Value, $[x \mapsto \text{fix}\, x : t \overset{\phi}{:0} .v]v$ is a value. $\text{csize}(\text{fix}\, x : t \overset{\phi}{:0}) = 1 > 0$
3. $c \xrightarrow{(congruence\, rule) c = E[c_1]} c'$. By induction, $(M, c_1) \rightarrow (M, c_1')$ and $\text{csize}(c_1) > \text{csize}(c_1')$ so $\text{csize}(E[c_1]) > \text{csize}(E[c_1'])$.

LEMMA [C-*STEP-VALUE]

If $C \vdash c : \tau$, where C has an empty Δ and Γ, then $(M, c) \xrightarrow{*} (M, v)$. The proof is shown in [14] and makes use of the C-Size-Decreases Lemma.

LEMMA [UNTYPED-NON-VALUES-STEP]

If $C \vdash (R, M, e : \tau)$ where C has an empty Δ and Γ and erase(e) is a non value, then erase((R, M, e)) $\rightarrow (L', d')$. Proof by induction on the expressions. The original proof is shown in [14]. Cases for lock, createlock, acquire, and release are shown here.

1. $e =$ lock, erase(e) is a value.

2. $e =$ create_ lock($e_{\text{ptr}}, e_{\text{Mem}}, e_{\text{resource}}$) | acquire($e_{\text{Lock}}, e_{\text{ptr}}$)
   | release($e_{\text{Lck}}, e_{\text{ptr}}, e_{\text{resource}}$).
   The proofs for all of these are similar so we will only show one.
   $e =$ acquire($e_{\text{Lock}}, e_{\text{ptr}}$).
   If e_{Lock} is a value, erase(acquire($e_{\text{Lock}}, e_{\text{ptr}}$)) = acquire(erase($e_{\text{Lock}}$), erase($e_{\text{ptr}}$)) by
   ($ER\_ACQUIRE$). If $e_{\text{ptr}} = c$, by C-Erase-Value erase(e_{ptr}) = u_{ptr} so erase(e) = acquire(u_{ptr},) and erase(e) can step by ($D\_ACQUIRE1$) or ($D\_ACQUIRE2$).
   If $e_{\text{ptr}} \neq c$ by Non-C-Erase-Non-Value erase(e_{ptr}) $\neq u$. By induction,
   erase((R, M, e_{ptr})) $\rightarrow (L', d_{\text{ptr}})$ and erase(e) steps by the acquire congruence rule.
   If e_{Lock} is a not a value, erase(acquire($e_{\text{Lock}}, e_{\text{ptr}}$)) =
   let $x =$ erase(e_{Lock}) in acquire(erase(e_{ptr}),) by ($ER\_ACQUIREe0$). By induction, erase((R, M, e_{Lock})) $\rightarrow (L', d_{\text{Lock}})$ and erase(e) steps by the let congruence rule.

B.6 Proof of Progress after Type Erasure

If e is a closed, well-typed expression ($C \vdash e : \tau$ for some τ and $C = \Psi; \Theta; \varnothing; \varnothing$), then the following statements hold:

1. (Erasure-Progress-1) If $(R, M, e) \rightarrow (R', M', e')$ then
   $\text{erase}((R, M, e)) \stackrel{?}{\rightarrow} \text{erase}((R', M', e'))$.
2. (Erasure-Progress-2) If $\text{erase}(e)$ is a value then
   $(R, M, e) \stackrel{*}{\rightarrow} (R', M', v)$, $\text{erase}((R, M, e)) = \text{erase}((R', M', v))$.
3. (Erasure-Progress-3) If $\text{erase}((R, M, e)) \rightarrow (L', d')$ then
   $(R, M, e) \stackrel{+}{\rightarrow} (R', M', e')$, $\text{erase}((R', M', e')) = (L', d')$.

In the cases where e does not affect locks, ($R = R'$), $(R, M, e) \rightarrow (R', M', e')$ is abbreviated to $(M, e) \rightarrow (M', e')$. Similarly, in the cases where e does not affect memory, ($M = M'$), $(M, e) \rightarrow (M', e')$ is abreviated to $e \rightarrow e'$. The cases involving create_lock, acquire, and release affect locks and the cases involving loads, stores, and congruence rules have the potential to affect memory. The cases which are unchanged from [14] have been omitted.

THEOREM [Erasure-Progress-1]

$$\frac{C \vdash (R, M, e : \tau) \quad (R, M, e) \rightarrow (R', M', e')}{\text{erase}((R, M, e)) \stackrel{?}{\rightarrow} \text{erase}((R', M', e'))}$$

If (R, M, e) evaluates in one step to (R', M', e'), then $\text{erase}((R, M, e))$ evaluates in zero or one steps to $\text{erase}((R', M', e'))$. Proof by induction on the typed evaluation rules. Most cases do not differ from [14] and have been ommitted. The load and store cases are shown along with the cases for createlock, acquire, and release.

There are several cases for this proof. For each, we list the evaluation rules which follow the case and show one proof. The other proofs in each case can be obtained using a similar proof to the example.

- Case: $(R, M, e) \rightarrow (R', M', e')$ using a typed evaluation rule and $\text{erase}((R, M, e)) \rightarrow \text{erase}((R', M', e'))$ using the corresponding untyped (or typed) evaluation rule.

1. (E_LOAD)If $e \overset{(E\_LOAD)}{\rightarrow} e'$ then $(M, e) = (M, \text{load}(i, \text{fact}))$ and $(M', e') = (M, {}^{\text{lin}}\langle M(i), \text{fact}\rangle)$.
2. (E_STORE) If $e \overset{(E\_STORE)}{\rightarrow} e'$ then $(M, e) = (M, \text{store}(i, \text{fact}, v))$ and $(M', e') = ([i \mapsto v]M, \text{fact})$.
3. $(E\_CREATELOCK)$ If $e \overset{(E\_CREATELOCK)}{\rightarrow} e'$ then $(R, M, e) = (R, M, \text{create\_ lock}(i, \text{fact}, v_{\text{resource}}))$ where $M(i) = 0$ and $(R', M', e') = ([i \mapsto \langle \text{fact}, v_{\text{resource}}\rangle]R, M, \text{lock})$.
4. $(E\_ACQUIRE1)$ If $e \overset{(E\_ACQUIRE1)}{\rightarrow} e'$ then $(R, M, e) = (R, M, \text{acquire}(\text{lock}, i))$ where $R(i) = \langle \text{fact}, v_{\text{resource}}\rangle$ and $M(i) = 0$ and $(R', M', e') = ([i \mapsto \langle \text{fact}\rangle]R, [i \mapsto 1]M, v_{\text{resource}})$.
5. $(E\_ACQUIRE2)$ If $e \overset{(E\_ACQUIRE2)}{\rightarrow} e'$ then $(R, M, e) = (R, M, \text{acquire}(\text{lock}, i))$ where $R(i) = \langle \text{fact}\rangle$ and $M(i) = 1$ and $(R', M', e') = (R, M, \text{acquire}(\text{lock}, i))$.
6. $(E\_RELEASE1)$ If $e \overset{(E\_RELEASE)}{\rightarrow} e'$ then $(R, M, e) = (R, M, \text{release}(\text{lock}, i, v_{\text{resource}}))$ where $R(i) = \langle \text{fact}\rangle$ and $M(i) = 1$ and $(R', M', e') = ([i \mapsto \langle \text{fact}, v_{\text{resource}}\rangle]R, [i \mapsto 0]M, \langle\rangle)$.
7. $(E\_RELEASE2)$ If $e \overset{(E\_RELEASE)}{\rightarrow} e'$ then $(R, M, e) = (R, M, \text{release}(\text{lock}, i, v_{\text{resource}}))$ where $R(i) = \langle \text{fact}, v_{\text{resource}}\rangle$ and $M(i) = 0$ and $(R', M', e') = (R, M, \text{release}(\text{lock}, i, v_{\text{resource}}))$.

Proof :

$(E\_CREATELOCK)$ If $e \overset{(E\_CREATELOCK)}{\rightarrow} e'$ then

$(R, M, e) = (R, M, \text{create\_ lock}(i, \text{fact}, v_{\text{resource}}))$ and

$(R', M', e') = ([i \mapsto \langle \text{fact}, v_{\text{resource}}\rangle]R, M, \text{lock})$. By $(T\_CREATELOCK)$ $M(i) = 0$.

$\text{erase}((R, M, e)) = (\text{erase}(M), \text{erase}(e))$ by (ER_RMe) and (ER_M).

This equals $(L,$ erase(create_ lock$(i,$ fact$, v_{\text{resource}})))$ =

$(L,$ create_ lock(erase(i), erase(fact), erase$(v_{\text{resource}})))$ by $(ER\_CREATELOCK)$ and equals $(L,$ create_ lock$(i, \langle\rangle, \langle\rangle))$ by (ER_i), (ER_FACT), and Zero Erase Value. erase$((R', M', e'))$ = (erase(M'), erase$(e'))$ by (ER_RMe). This equals $(L',$ erase(lock)) = $(L', \langle\rangle)$ by (ER_LOCK) and (ER_M). Finally, $(L,$ create_ lock$(i, \langle\rangle, \langle\rangle)) \overset{(D_CREATELOCK)}{\rightarrow} (L, \langle\rangle)$ where $L(i) = 0$ so erase$((R, M, e)) \rightarrow$ erase$((R', M', e'))$.

THEOREM [ERASURE-PROGRESS-2]

$$\frac{C \vdash (R, M, e : \tau) \text{ where } C \text{ has an empty } \Delta \text{ and } \Gamma \quad \text{erase}(e) \text{ is a value}}{(R, M, e) \xrightarrow{*} (R, M, v),\ \text{erase}((R, M, e)) = \text{erase}((R, M, v))}$$

The proof is unchanged with the addition of locks and is shown in [14]. It makes use of the C-*Step-Value Lemma.

THEOREM [ERASURE-PROGRESS-3]

$$\frac{C \vdash (R, M, e : \tau) \text{ where } C \text{ has an empty } \Delta \text{ and } \Gamma \quad \text{erase}((R, M, e)) \rightarrow (L', d')}{(R, M, e) \xrightarrow{+} (R', M', e'),\ \text{erase}((R', M', e')) = (L', d')}$$

If erase(e) is not a value then, by Untyped-Non-Values-Step, erase$((R, M, e))$ evaluates to (L', d'). Additionally, (R, M, e) evaluates in one or more steps to (R', M', e'), and erase$((R', M', e'))$= (L', d'). Proof by induction on the expressions. Most cases are unchanged from [14] so we only show load and store along with lock, createlock, acquire, and release.

1. Let $e =$ lock

   By Value-Erase-Value, erase$(v) = u$ and Erasure-Progress-3 does not apply.

2. Let $(R, M, e) = (R, M, \text{load}(e_{\text{ptr}}, e_{\text{Mem}}))$ | $(R, M, \text{store}(e_{\text{ptr}}, e_{\text{Mem}}, e_{\text{v}}))$
   | $(R, M, \text{acquire}(e_{\text{Lock}}, e_{\text{ptr}}))$ | $(R, M, \text{create\_ lock}(e_{\text{ptr}}, e_{\text{Mem}}, e_{\text{v}}))$
   | $(R, M, \text{release}(e_{\text{Lock}}, e_{\text{ptr}}, e_{\text{v}}))$

The proofs for these cases are similar. We will only show one of the proofs.

Let $(R, M, e) = (R, M, \text{load}(e_{\text{ptr}}, e_{\text{Mem}}))$.

$\text{erase}((R, M, e)) = \text{erase}((R, M, \text{load}(e_{\text{ptr}}, e_{\text{Mem}})))$

$= (\text{erase}(M), \text{erase}(\text{load}(e_{\text{ptr}}, e_{\text{Mem}}))) = (L, \text{load}(\text{erase}(e_{\text{ptr}}), \text{erase}(e_{\text{Mem}}))$

by (ER_RMe), (ER_M), and (ER_LOAD).

If e_{ptr} and e_{Mem} are in c, $\text{erase}(e_{\text{ptr}})$ and $\text{erase}(e_{\text{Mem}})$ are values so by Erasure-2, $(M, e_{\text{ptr}}) \xrightarrow{*} (M, v_{\text{ptr}})$ where $\text{erase}(v_{\text{ptr}}) = \text{erase}(e_{\text{ptr}})$ and $(M, e_{\text{Mem}}) \xrightarrow{*} (M, v_{\text{Mem}})$ where $\text{erase}(v_{\text{Mem}}) = \text{erase}(e_{\text{Mem}})$. By (T_LOAD), $\text{erase}(e_{\text{ptr}}) = i$ and $\text{erase}(e_{\text{Mem}}) = \langle\rangle$. $\text{erase}((R, M, e)) = (L, \text{load}(i, \langle\rangle)) \overset{(D\_LOAD)}{\rightarrow} (L, \langle L(i), \langle\rangle\rangle) = (L', d')$.

$(R, M, e) =$

$(R, M, \text{load}(e_{\text{ptr}}, e_{\text{Mem}})) \overset{*,(congruence\ rule) E[e]=\text{load}(e,e_2)}{\rightarrow} (R, M, \text{load}(v_{\text{ptr}}, e_{\text{Mem}}))$

$\overset{*,(congruence\ rule) E[e]=\text{load}(v,e)}{\rightarrow} (R, M, \text{load}(v_{\text{ptr}}, v_{\text{Mem}})) = (R, M, \text{load}(i, \text{fact}))$ by Canonical forms and (T_LOAD). $(R, M, \text{load}(i, \text{fact})) \overset{(E\_LOAD)}{\rightarrow} (R, M, {}^{\text{lin}}\langle M(i), \text{fact}\rangle)$

$= (R', M', e')$.

$\text{erase}((R', M', e')) = \text{erase}((R, M, {}^{\text{lin}}\langle M(i), \text{fact}\rangle)) = (L, \langle L(i), \langle\rangle\rangle) = (L', d')$.

If e_{ptr} is in c but e_{Mem} is not, $\text{erase}(e_{\text{ptr}})$ is a value so by Erasure-2, $(M, e_{\text{ptr}}) \xrightarrow{*} (M, v_{\text{ptr}})$ where $\text{erase}(v_{\text{ptr}}) = \text{erase}(e_{\text{ptr}})$. e_{Mem} is not a value so $\text{erase}((M, e_{\text{Mem}})) \rightarrow (L', d_{\text{Mem}})$ by Progress and $(M, e_{\text{Mem}}) \xrightarrow{+} (M', e'_{\text{Mem}})$ and $\text{erase}((M', e'_{\text{Mem}})) = (L', d_{\text{Mem}})$ by induction.

By (T_LOAD), $\text{erase}(e_{\text{ptr}}) = i$. $\text{erase}((R, M, e)) = (L, \text{load}(i, \text{erase}(e_{\text{Mem}}))) \rightarrow (L', \text{load}(i, d_{\text{Mem}})) = (L', d')$.

$(R, M, e) = (R, M, \text{load}(e_{\text{ptr}}, e_{\text{Mem}}))$

$\overset{*,(congruence\ rule) E[e]=\text{load}(e,e_2)}{\rightarrow} (R, M, \text{load}(v_{\text{ptr}}, e_{\text{Mem}}))$

$\overset{+,(congruence\ rule) E[e]=\text{load}(v,e)}{\rightarrow} (R', M', \text{load}(v_{\text{ptr}}, e'_{\text{Mem}}))$

$= (R', M', \text{load}(i, e'_{\text{Mem}}))$ by Canonical forms and (T_LOAD). $= (R', M', e')$.

$\text{erase}((R', M', e')) = \text{erase}((R', M', \text{load}(i, e'_{\text{Mem}}))) = (L', \text{load}(i, d_{\text{Mem}})) = (L', d')$.

If e_{ptr} is not in c, e_{ptr} is not a value so $\text{erase}((M, e_{\text{ptr}})) \rightarrow (L', d_{\text{ptr}})$ by Progress and

$(M, e_{\text{ptr}}) \xrightarrow{+} (M', e'_{\text{ptr}})$ and $\text{erase}((M', e'_{\text{ptr}})) = (L', d_{\text{ptr}})$ by induction.

$\text{erase}((R, M, e)) =$

$(L, \text{load}(\text{erase}(e_{\text{ptr}}), \text{erase}(e_{\text{Mem}}))) \rightarrow (L', \text{load}(d_{\text{ptr}}, \text{erase}(e_{\text{Mem}}))) = (L', d').$

$(R, M, e) =$

$(R, M, \text{load}(e_{\text{ptr}}, e_{\text{Mem}})) \overset{*,(congruence\ rule)E[e]=\text{load}(e,e_2)}{\rightarrow} (R', M', \text{load}(e'_{\text{ptr}}, e_{\text{Mem}}))$

$= (R', M', e').$

$\text{erase}((R', M', e')) =$

$\text{erase}((R', M', \text{load}(e'_{\text{ptr}}, e_{\text{Mem}}))) = (L', \text{load}(d_{\text{ptr}}, \text{erase}(e_{\text{Mem}})) = (L', d').$

C Lock Implementation in Clay

This implementation of locks is done in Clay and matches the abstract syntax in Appendix A.

Included Clay types and functions

```
type Int[I] = native
@type0 Mem[int I, int T] = native
typedef Ptr[int I, int T] = @[Int[I], Mem[I,T]]
```

The above types are needed to implement locks. $Int[I]$ is a singleton integer and a pointer to memory location I. Mem is the memory State Type which tracks a word of linear memory. $Mem[I,T]$ means that memory location I contains a value of type $Int[T]$. $Ptr[I,T]$ is a linear pair of a pointer to linear memory and its associated Mem.

Locks in Clay

The Clay lock code defines the basic lock type and functions and implements the outer lock creation function.

```
// Lock0 is the lock type. Lock is a Lock0 and a lock bit pointer
type0 Lock0[int Bit, @type0 Resource] = native
typedef Lock[int Bit, @type0 Resource] =
  .[Lock0[Bit, Resource], Int[Bit]]
// Lock creation functions
native Lock0[B,R] create_lock0
  [@type0 R, int B] (R resource, Mem[B,0] bit) limited[0];
Lock[B,R] create_lock
  [@type0 R, int B] (R resource, Ptr[B,0] bit_ptr)
```

```
{
  let (bit_addr, bit_mem) = bit_ptr;
  let lock0 = create_lock0(resource, bit_mem);
  return .(lock0, bit_addr);
}
// Lock acquire function
native R acquire [int B, @type0 R] (Lock[B,R] lock);
// Lock release function
native void release [int B, @type0 R] (Lock[B,R] lock, R resoure);
```

Locks in C

The C lock code implements the types and functions declared (unlimited) native in the Clay lock code.

```
// Acquire function sets a lock bit to 1 when it becomes 0.
void acquire(unsigned long lock)
{
  int a = 1;
  while (a == 1)
      a = atomic_test_and_set(*((int *)lock));   // assembly x86 btsl
}
// Release function sets a lock bit to 0 when it becomes 1.
void release(unsigned long lock)
{
  int a = 0;
  while (a == 0)
      a = atomic_test_and_clear(*((int *)lock)); // assembly x86 btcl
}
```

D Built-in Primitives

In my abstract machine, the lock type and access expressions need to be added to the syntax and rules before I prove soundness. A *Lock* is an example of a state type which takes another state type as a parameter. Like Mem, locks needs their own environments R and Θ to hold the linear resources and their types. Further state types also need to be added individually with their access expressions, and environments before soundness can be re-proven.

State types could be expressed through consolidated typing and kinding rules with only 2 new environments. However, the downsides to this approach cause this abstract machine to have just as many evaluation rules as my abstract machine so I decided for simplify to add state types such as locks as needed.

consolidated environments

$$S = \{\sigma_1 \mapsto K_1, \cdots, \sigma_n \mapsto K_n\}$$

$$T = \{\chi_1 \mapsto \tau_1, \cdots, \chi_n \mapsto K_n\}$$

$$C = \Psi; T, \Delta; \Gamma$$

S is the state kind environment which maps state types (such as Mem) to kinds. If Mem was removed from the abstract machine and replaced with a state type, the S environment would show $S(\text{Mem}) = (int, \overset{\text{non}}{1}) \rightarrow \overset{\text{lin}}{0}$ meaning Mem takes a kind int and a kind non-linear size 0 and has type linear size 0.

T is the state type environment which maps state type operations (such as *load*) to types. Keeping the Mem example,

$$T(\text{load}) = \Lambda\alpha : int, \beta : \overset{\text{non}}{1} \,.\, {}^{\text{lin}}\langle \text{Mem}(\alpha, \beta), \text{Int}(\alpha)\rangle \rightarrow^{\text{lin}} \langle \text{Int}(\beta), \text{Mem}(\alpha, \beta)\rangle$$

which means that *load* takes a singleton integer (a pointer) and a matching Mem

and returns a value of type β.

consolidated typing rules

$$(T\_STATE)\ \overset{\text{non}}{C} \vdash \chi : T(\chi)$$

consolidated kinding rules

$$(K\_STATE)\Delta \vdash \sigma : S(\sigma)$$

The above syntax and rules would add state types to the abstract machine. This would allow all state types values to be expressed by one type and aid polymorphism. The same principle could be applied to configuration types. However, the state type operations include any expressions or operations which take state types as arguments. Using these operations would require a list of constants for the operations (*load*, *store*, *set_irq*, ...) and an evaluation rule for each operation ((E_LOAD), (E_STORE), ...). As this would not drastically shorten an abstract machine which includes the many state and configuration types needed by an operating system, this abstract machine does not use the consolidated state and configuration types.

References

[1] Donald Becker. 3com etherlink iii 3c5x9 driver v. 1.18. http://joshua.raleigh.nc.us/docs/linux-2.4.10_html/284303.html, 2000.

[2] William Bevier. *A Verified Operating System Kernel*. PhD thesis, University of Texas at Austin, 1987.

[3] Andrew Birrell. An introduction to programming with threads. In *Research Reports*, number 35. SRC, January 1989.

[4] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An empirical study of operating system errors. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pages 73–88, Banff, Alberta, Canada, 2001.

[5] 3Com Corporation. Etherlink iii parallel tasking isa, eisa, micro channel, and pcmcia adapter drivers technical reference. 1-800-NET-3Com, August 1994.

[6] Russ Cox, Eric Grosse, Rob Pike, Dave Presotto, and Sean Quinlan. Security in plan 9. To appear in Proc. of the 2002 Usenix Security Symposium, San Francisco.

[7] Robert DeLine and Manuel Fähndrich. Enforcing high-level protocols in low-level software. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, Snowbird, UT, United States, June 2001.

[8] Dawson Engler, Davic Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pages 57–72, Banff, Alberta, Canada, 2001.

[9] Bryan Ford, Godmar Back, Greg Benson, Jay Lepreau, Albert Lin, and Olin Shivers. The flux oskit: A substrate for os and language research. In *Proceedings*

of the 16th ACM Symposium on Operating Systems Principles, pages 38–51, Saint-Malo, France, October 1997.

[10] Dan Grossman. Type-safe multithreading in cyclone. In *ACM Workshop on Types in Language Design and Implementation*, pages 13–25, New Orleans, LA, January 2003.

[11] Chris Hawblitzel. Personal communication about the Marianas network of secure coprocessors.

[12] Chris Hawblitzel, Edward Wei, Heng Huang, Erik Krupski, and Lea Wittie. Low-level linear memory management. In *Workshop on Semantics, Program Analysis, and Computing Environments for Memory Management*, Venice, Italy, January 2004.

[13] Wilson Hsieh, Marc Fiuczynski, Charles Garrett, Stefan Savage, David Becker, and Brian Bershad. Language support for extensible operating systems. In *Workshop on Compiler Support for System Software*, February 1996.

[14] Heng Huang, Lea Wittie, and Chris Hawblitzel. Formal properties of linear memory types. Technical report, Dartmouth College, 2003.

[15] Trevor Jim, Greg Morrisett, Dan Grossman, Michael Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of c. In *USENIX Annual Technical Conference*, pages 275–288, Monterey, CA, June 2002.

[16] John Mitchell and Gordon Plotkin. Abstract types have existential types. *toplas*, 10(3):470–502, 1988.

[17] Greg Morrisett. Cyclone project. http://www.eecs.harvard.edu/ greg/cyclone/.

[18] Greg Morrisett, Karl Crary, Neal Glew, Dan Grossman, Richard Samuels, Frederick Smith, David Walker, Stephanie Wierich, and Steve Zdancewic. Talx86: A realistic typed assembly language. In *Proceedings of the 1999 ACM SIGPLAN*

Workshop on Compiler Support for System Software, pages 25–35, Altanta, GA, United States, May 1999.

[19] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From system f to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):528–569, May 1999.

[20] George C. Necula and Peter Lee. Safe kernel extensions without run-time checking. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation*, Seattle, WA, United States, October 1996.

[21] David M. Nicol, Sean W. Smith, Chris Hawblitzel, and Edward A. Feustel. Marianas: Survivable trust for critical infrastructure. Current research.

[22] Benjamin Pierce. *Types and Programming Languages*. The MIT Press, 2002.

[23] William Pugh. The omega project. http://www.cs.umd.edu/projects/omega/.

[24] SecurityFocus. Securityfocus vulnerabilities archive. http://online.securityfocus.com/bid/vendor/, 2002.

[25] John D. Strunk, Garth R. Goodson, Michael L. Scheinholtz, Craig A.N. Soules, and Gregory R. Ganger. Self-securing storage: Protecting data in compromised systems. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation*, San Diego, CA, United States, October 2000.

[26] Hendrik Tews. Case study in coalgebraic specification: Memory management in the fiasco microkernel. Technical report, Technical University of Dresden, 2000. Technical report TPG2/1/2000 on FSB 358.

[27] Hendrik Tews, Hermann Härtig, and Michael Hohmuth. Vfiasco - towards a provably correct microkernel. Technical report, Technical University of Dresden, 2001. Technical report TUD-FI01-1, ISSN 1430-211X.

[28] Emin G un Sirer, Stefan Savage, Przemyslaw Pardyak, Greg DeFouw, Mary Ann Alapat, and Brian Bershad. Writing an operating system using module-3. In

Proceedings of the 1996 ACM SIGPLAN Workshop on Compiler Support for System Software, 1996.

[29] P. Wadler. Linear types can change the world! In M. Broy and C. Jones, editors, *IFIP TC 2 Working Conference on Programming Concepts and Methods, Sea of Galilee, Israel*, pages 347–359. North Holland, 1990.

[30] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

[31] Honwei Xi and Frank Pfenning. Eliminating array bounds checking through dependent types. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, pages 249–257, Montreal, Canada, June 1998.

[32] Yichen Xie and Dawson Engler. Using redundancies to find errors. In *Proceedings of the 2002 ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 51–60, Charleston, South Carolina, United States, November 2002.

[33] Zhichen Xu, Barton P. Miller, and Thomas Reps. Safety checking of machine code. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, pages 70–82, Vancouver, B.C. Canada, June 2000.