CONCURRENCY ANALYSIS BASED ON FRACTIONAL PERMISSION SYSTEM

by

Yang Zhao

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

(Engineering)

at the

UNIVERSITY OF WISCONSIN - MILWAUKEE

August 2007

CONCURRENCY ANALYSIS BASED ON FRACTIONAL PERMISSION SYSTEM

by

Yang Zhao

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

(Engineering)

at the

UNIVERSITY OF WISCONSIN - MILWAUKEE

August 2007

Major Professor:	________________	Date:	__________
Graduate School Approval:	________________	Date:	__________

Abstract

CONCURRENCY ANALYSIS BASED ON FRACTIONAL PERMISSION SYSTEM

Yang Zhao
The University of Wisconsin-Milwaukee, 2007
Under the supervision of Prof. John Boyland

Concurrent programs are hard to write and debug because of the inherent concurrency and indeterminism. The most common runtime errors in concurrent programs are data races and deadlocks.

This thesis presents a "fractional permission" type system for a Java-style shared-memory programs. A permission is a linear value associated with some piece of state in a program. Fractions are used to distinguish reads from writes and the permission nesting is used to indicate that some permissions may be nested in some others. Within the permission analysis, each expression in the program will be checked to determine whether it is permitted to be executed under the granted permissions.

Permissions come from the design intents expressed by field and method annotations. Besides the traditional pointer annotations (uniqueness, nullity ...), a field may be attached a protector, such that any access to this field should be in the synchronized block holding that protector object. The protector could be either the receiver, or some formal parameters appearing in the class definition. Method annotations are not only the traditional “`reads`”, “`writes`” effects, but also some lock usage including “`requires`”, “`uses`” and some partial order among locks.

We provide the fractional permission type system as well as the operational semantics for a simple object-oriented language. A consistency property between the static permission environment and the dynamic runtime state is established, with which we show the soundness as well.

The novel technical features of this approach include: (1) A unified permission form is created to represent all annotations in multithreaded programs including uniqueness, nullity, method effects, lock protected state etc. (2) The permission type system is extended to programs with unstructured parallelism and synchronization; (3) Fields and maybe their pointed-to objects are attached some protection mechanism; (4) Lock objects could be ordered based on some levels; (5) The permission nesting is used to simulate the protection mechanism between fields (data groups) and their guards. (6) Formal rules for permission typing, transformation and consistency.

Major Professor: ______________________ Date: ____________

To

my parents and my wife

Table of Contents

List of Figures

List of Tables

Acknowledgements

I’d like to thank ...

Chapter 1 Introduction

1.1 Why Concurrency?

In computer science, concurrency is a property of systems in which several computational processes are executing at the same time, and potentially interacting with each other [1].

Moore's Law is the observation that the amount you can do on a single chip doubles every two years. Historically, Moore's Law has delivered ever faster computing power to more and more demanding audiences. But it is about to break down–there's a limit to how many interconnections you can create on a chip. Rather than producing faster and faster processors, companies such as Intel and AMD are instead producing multi-core devices: single chips containing two, four, or even more processors [2]. In this case, the concurrent processes may be executing truly simultaneously, that is, your software should take advantage of that extra power to run on separate processors. If your programs aren't concurrent, they'll only run on a single processor at a time which may cause your code slow comparing to others.

Even for a single hardware processor, multithreading enables a program to make the best use of available CPU cycles, thus allowing very efficient programs to be written. For example, multithreading is a natural choice for handling event-driven code, which is so common in today's highly distributed, networked, GUI-based environments [3]. This kind of programs spends a great deal of time waiting for outside events. To respond in time, people may use time slicing. That is, one-processor executes only a single thread in any given time slice. Operating systems simulate doing many things at once by rapidly time-slicing between threads.

The concurrent programming is becoming a mainstream programming practice, but it is difficult and error prone. The difference between a sequential system and a concurrent system is the fact that multiple parallel threads may interact with each other nondeterministically which makes it difficult to extend traditional program analysis techniques developed for sequential systems to concurrent systems. There are two common runtime errors: *data races* and *deadlocks*. A data race occurs when two threads concurrently access the same data without synchronization, and at least one of the accesses is a write. Data races involving shared resources can result in unpredictable system behavior or some severe runtime exceptions.

Example 1: Assuming there is an account object `a` and its `balance` field is $100 originally. If we in parallel deposit money within different threads, then the field `balance` will be updated in parallel at the same time (the `||` is used to indicate there are two concurrent computations):

```
a.balance = a.balance + 10 || a.balance = a.balance + 20
```

This piece of code may exhibit unexpected behaviors. Because of the nondeterministic interference between these two threads, there are more than one execution paths. Figure 1.1 lists some of them. For instance, the (a) shows such an execution path: the first thread may load the `balance` field of the account object and add a constant `10`, then just before storing the new value to the field, it yields to the second thread to finish the whole deposit process. After that, the first thread continues to do the store operation. Using this path, the final value for the balance field is $110, which only reflects one deposit operation. The (b) in Figure 1.1 shows another path ending with $120 instead.

Neither (a) nor (b) gives a satisfied result, since we are supposed to get a \$130 result ((c) shows such a path).

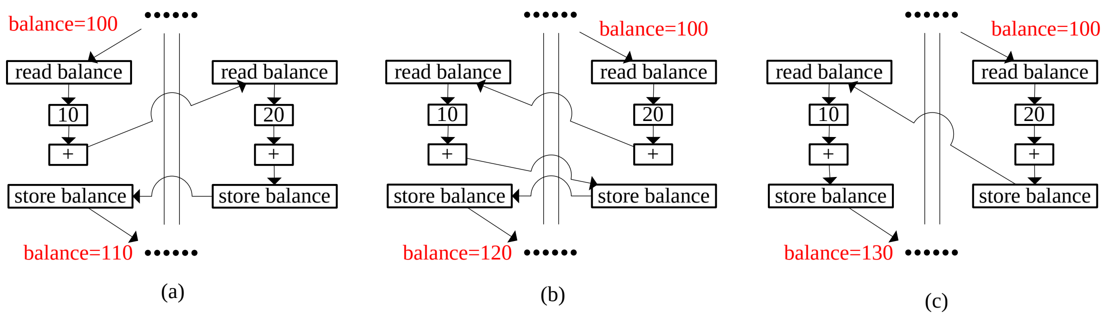


Figure 1.1: Data races.

This kind of problems is caused by the date race among different threads in parallel which could be easily fixed by using synchronization:

```
synchronized a do { a.balance = a.balance + 10 }
  || synchronized a do { a.balance = a.balance + 20 }
```

Either thread that trying to execute the deposit operation has to acquire the lock of the account object first. With the help of mutually exclusive lock, only one thread is able to hold the lock at any time, therefore the synchronization ensures that the two deposit operations are not interrupted in unexpected ways. Figure 1.2 demonstrates two possible paths with using synchronization. In other words, the usage of synchronization eliminates some "bad" paths.

A deadlock appears when multiple concurrent threads are holding some resources but waiting for more that are currently held by some others. It is possible that none of the parallel threads can make progress. For example: thread t_1 holds resource r_1 waiting for r_2, t_2 holds r_2 waiting for r_3, ..., t_n holds r_n waiting for r_1. All of them are blocked and the whole program gets stuck. There is a classic deadlock example in computer science: *Dining philosophers problem* which is retold by Tony Hoare [4].

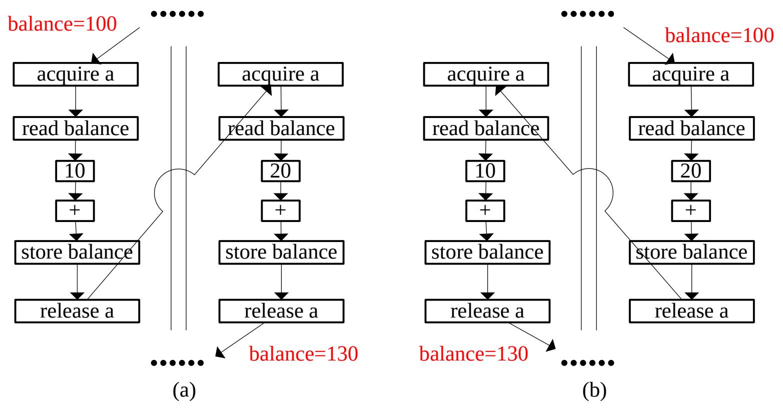


Figure 1.2: Race free.

Example 2: For a transfer operation that withdraw money from a checking account and deposit the same amount to the savings account, it's safe to lock both checking and savings accounts before this transfer and there are two possibilities that either lock the checking account before the savings account, or the other way around. If two transfers with different locking sequence are executed in parallel, then there may be a problem.

```
synchronized checking do {synchronized savings do {...}}
  || synchronized savings do {synchronized checking do {...}}
```

It is race-free, and it may be ok sometimes (see the execution path in Figure 1.3.(a)), but it may also encounter a deadlock (see the execution path in Figure 1.3.(b)): After the `checking` is locked by the first thread, it yields to the second thread to lock `savings`. But when the second thread tries to lock the `checking`, it fails, since the `checking` is held by the first thread who is currently expecting the `savings`. Neither of the two parallel threads could execute further more and the whole program gets stuck.

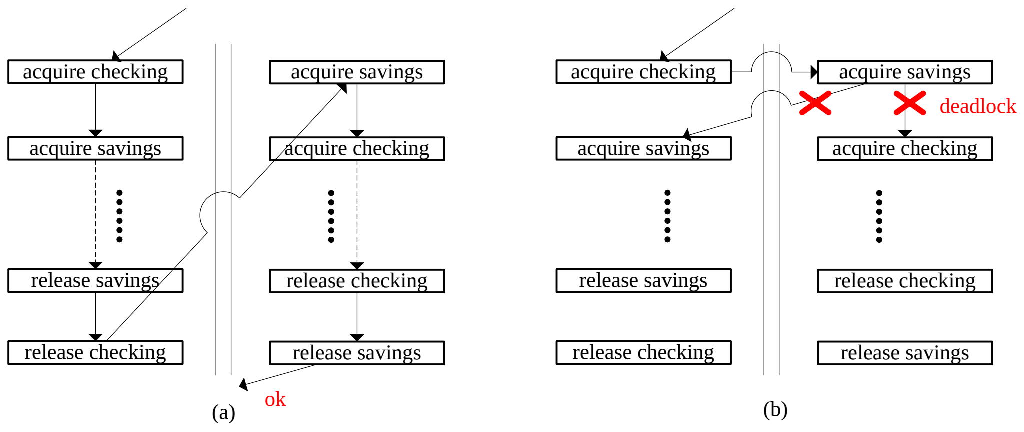


Figure 1.3: Deadlock.

1.2 Concurrency Analysis

The concurrency analysis is an analysis for multithreaded programs that can be used to detect parallel programming errors or enable optimizations.

Numerous static and dynamic analysis techniques are designed to ensure the concurrency programs are free of race conditions and deadlocks. The difference between static and dynamic analysis is that the former does not need to run the program while the latter does. Moreover, the dynamic analysis may not discover all the errors since it is difficult (or even impossible) to go through all the possible execution paths. As far as we know, most concurrency analysis techniques are static although they may additionally require annotations or use approximations and may not precisely simulate the actual runtime behaviors. Roughly, there are mainly two directions: the fixed-point analysis and the type system.

The fixed-point analysis can be either flow-sensitive [26, 31, 32] or flow-insensitive [33, 34]. The former combines existing sequential analyses with the interference information between parallel threads, while the latter ignores the flow of control and represents programs as a set of statements which can be executed multiple times in any possible order. Hence the flow-insensitive analysis may automatically model all the possible interleaving of statements from different threads, however it is less precise than its flow-sensitive counterparts normally [35].

Numerous static analysis techniques are designed to ensure the parallel programs are free of data races [26, 27, 36], and some of them are based on the programming language's type system with annotations [15, 13, 9, 10, 20]. For example, Boyapati and others [9, 10] introduce a variant of ownership types with which all fields are protected by their owners and any access to an object needs to have its owner (root-owner) in the context. Flanagan and others [15, 13] require every field to have a guard object which must be held whenever that field access happens. Guava [12] is a dialect of Java whose rules statically guarantee that parallel threads access shared data only through synchronized methods. Kobayashi and others [20, 23] use type systems for analyzing behavior of concurrent processes to reason about deadlock-freedom and safe usage of locks.

In previous work, Boyland introduces a fractional permission type system to check all the possible interference in a simple non-synchronizing imperative language with structural parallelism [5]. He uses fractions to distinguish reads from writes and shows that the permission system enables parallelism to proceed with deterministic results. After that, a nesting relation in the permission system is designed to connect effects and uniqueness in an object-oriented language without parallelism [7].

In this thesis, we further extend the fractional permission to multithreaded programs with synchronization. Our main concerns are the data race and deadlock detection. In our permission system, field and method definitions need to be attached some non-executable annotations representing the design intents in the code. High-level annotations could be translated into their permission representation and then every expression is checked to determine whether it is permitted to be executed under given permissions. A method is well permission-formed if its body can be checked using its declared permission type.

1.3 Outline

The remainder of this document is organized as follows. Chapter 2 provides several static concurrency analyses that are closely related to our concurrent permission system. Our analysis borrows some techniques from them. Chapter 3 defines a simple object-oriented language including all kinds of program annotations. The operational semantics is given as well. The following chapters focus on permissions: Chapter 4 first introduces the permission syntax step by step, then shows how to translate all kinds of annotations into their permission representation; Chapter 5 provides all permission checking rules and transformation rules as well; Chapter 6 builds the consistency between the static permission environment and the dynamic runtime state with some flattening rules, then the soundness property is established. In Chapter 7, We briefly introduce some issues about the implementation and provide several examples to demonstrate the permission checking. Finally, we discuss some open issues in Chapter 8 and conclude in Chapter 9.

Chapter 2

Related Concurrency Analysis

In this chapter, we go though three kinds of concurrency analysis that are closely related to our permission system. Most of modern programming languages are equipped with type systems, which help reasoning about program behavior and early finding bugs. Among all kinds of type-based concurrency analysis, the typical ones include Boyapati's ownership system and Flanagan's type-based `rccjava`. Greenhouse's lock analysis is an annotation-based approach for expressing design intents in multithreaded programs with some useful "mechanical" properties.

2.1 Owners-as-Locks

Ownership is a recognized alias control technique. With ownership, each object has another object as its *owner*. The root of the ownership hierarchy is often called "world." Some researchers propose an owners-as-dominators model: any reference to an object must pass that object's owner [39, 40, 41, 10].

Boyapati et al. [11, 9] extend ownership to concurrent programs. Programmers associate every object with a protection mechanism that ensures the accesses to the object never create data races. The specified protection mechanism for each object act as part of the type of the variables that point to that object.

First, each object is owned by another object, by itself, or by a special per-thread owner called `thisThread` which is implicitly held by the local thread. Objects owned by `thisThread`, either directly or transitively, are local to the corresponding thread and cannot be accessed by any other thread. Figure 2.1 shows the ownership relation among some objects. The arrow shows an owner relation such that the source is the owner of the sink. For example, the `thisThread` is the owner for both `o1` and `o3`, while the `o4` is self-owned. It is explicitly required that any object has a unique owner. The multiple ownership has not been supported yet.

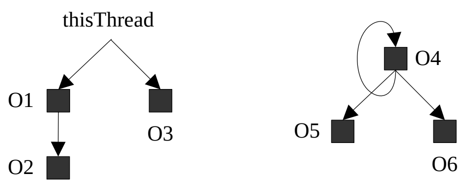


Figure 2.1: Ownership relation.

Second, the requirement for a thread to access an object is that the thread must hold the lock on the root of the ownership tree (root-owner for short) that the object belongs to. It's clear that any thread has already held the `thisThread` at the very beginning. For example, it's safe to access `o1`, `o2` and `o3` since all of them has the same `thisThread` as their root-owner, while the `o4` must be held in order to access itself, `o5` or `o6`. Since any access to an object must go through its root-owner, it is enough to guarantee that objects are protected by mutual exclusion locks. Moreover, thread-local objects can be accessed without synchronization.

Figure 2.2 shows a concrete example from Boyapati's Race-free Java [10]. For a code segment shown below, it creates an ownership relation tree in Figure 2.3.

```
TStack<thisThread, thisThread> s1 = new TStack<thisThread, thisThread>;
TStack<thisThread, self> s2 = new TStack<thisThread, self>;
```

A `TStack` is a stack of `T` objects and is implemented using a linked list. A class definition in Race-Free Java is parameterized by a list of owners, where the first one is always the owner of the current instance. This parameterization helps programmers write

```
class TStack<thisOwner, TOwner> {
  TNode<this, TOwner> head = null;

  T<TOwner> pop() accesses (this) {
    if (head == null) return null;
    T<TOwner> value = head.value();
    head = head.next();
    return value;
  }
  ......
}

class T<thisOwner> { int x=0; }
```

```
class TNode<thisOwner, TOwner> {
  T<TOwner> value;
  TNode<thisOwner, TOwner> next;

  T<TOwner> value() accesses (this)
  { return value; }

  TNode<thisOwner, TOwner> next()
  accesses (this)
  { return next; }
  ...
}
```

Figure 2.2: Stack of T Objects in Race-Free Java

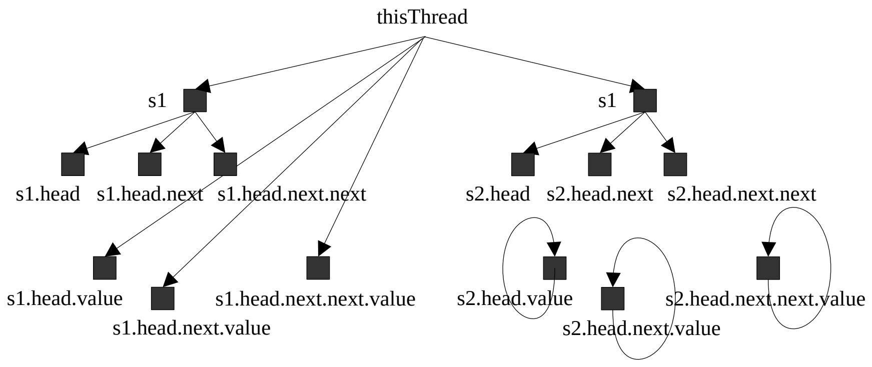


Figure 2.3: Ownership Relation for TStacks s1 and s2.

generic code to implement a class, but create different objects of the class with different protection mechanism.

Methods may be attached some effects showing the requirements of its body. The "`accesses`" annotation shows that its target object may be accessed (read or written) in the body. From the protection mechanism mentioned above, this effect implicitly indicates that the root-owner of the target object should be held at the method entry, otherwise it will be unsafe. In the example, the `value` and `next` methods in the `TNode` class both assume that the callers hold the lock on the root-owner of their receiver object. Without the "`accesses`" clause, the two methods would not have been well-typed.

Immutable objects and unique pointed-to objects are accessible without synchronization. Unique pointers are very important in concurrent programs, since they are useful to support object migration between threads, thus some optimizations, such as removing unnecessary synchronizations, are possible.

Boyapati and others [10] further add the deadlock detection into their ownership system. Objects that own themselves are locks. Each lock object may belong to some lock level, which forms a partial order. When a thread tries to acquire a new lock `l`, the levels of all the locks that the thread currently holds are greater than the level of `l`. In other words, locks must be obtained in descending order. A thread may also acquire a lock that it already holds. The lock acquire operation will be redundant in this case.

Ownership types provide a statically enforceable way of specifying object encapsulation and they are useful for preventing data races because the lock that protects an object can also protect its encapsulated objects. However, the ownership type system is very restrictive. For instance, it does not distinguish the *reads* from *writes*. Both of them are considered as *accesses*. Moreover, different fields of an object have to be protected by a same root-owner of the object.

2.2 Type-based rccjava

`Rccjava` is a type-based race detection tool used to check multithreaded Java programs developed by Flanagan and others [13, 15, 18].

In `Rccjava`, a class may be parameterized by external locks, which allows its fields to be protected by some locks external to the class. Each field must have an annotation "`guarded_by` l", such that any access (either read or write) to this field is required to have the l been held by the current thread.

A class definition contains a (possibly empty) sequence of formal parameters or *ghost* variables. These ghost variables are only used by the type system to verify that the program is race free, thus they won't affect the run-time behavior of the program. Figure 2.4 shows an example of a dictionary that maps keys to values. In this example,

```
class Node<ghost Dictionary d> {
  String key guarded by d = null
  Object value guarded by d = null
  Node<d> next guarded by d = null

  void init(String k, Object v,
      Node<d> n)  requires d {
    node.key = k; node.value = v;
    node.next = n
  }

  void update(String k, Object v)
  requires d {
    if (this.key.equals(k)) {
      this.value = v
    }else if (this.next != null) {
      this.next.update(k,v)
    }
  }
  ......
}
```

```
class Dictionary {
  Node<this> head guarded by this = null

  void put(String k, Object v) {
    synchronized this in {
      if (this.contains(k)) {
        this.head.update(k,v)
      }else {
        let Node<this> node =
               new Node<this> in {
          node.init(k,v,this.head);
          this.head = node
        }
      }
    }
  }
  ......
}
```

Figure 2.4: A synchronized dictionary.

the dictionary is represented as an object containing a linked list of `Node` objects, where each `Node` contains a `key`, a `value`, and a `next` reference pointing to the next. The class `Node` is parameterized by the enclosing dictionary `d` which guards all the fields of `Node`; and each method of `Node` requires that the `d` is held on the entry. Each method of `Dictionary` first acquires the receiver lock and then proceeds with the appropriate manipulation of the linked list. Since all fields of the linked list are protected by the dictionary lock, the type system verifies that this program is well typed and race free.

A class may be annotated as "`thread_local`", such that any instance of that class is local to a particular thread and thus is safely accessible without synchronization.

This type-based race detection tool partially improves Boyapati's *owners-as-locks* model by specifying each field with a guard object, which makes the type system more flexible and powerful. However, `rccjava` enforces every field to have a guard and it does not have anything about deadlocks. Furthermore, this type-based analysis relies on programmer-inserted type annotations that describe the locking discipline, it limits `rccjava`'s applicability to large, legacy systems. Flanagan and others [19, 14] further develop the annotation inference techniques to achieve practical static race detection for large programs,

2.3 Concurrency Assurance in Fluid

Greenhouse and others [27, 28] explore the costs and benefits of a new annotation-based approach for expressing design intents. They use annotations to express "mechanical" properties such as lock-state associations, uniqueness of references, and encapsulation of state into named aggregations.

A fundamental requirement for assuring the correct use of shared state is the identification of state that is shared, where state may encompass multiple variables and span multiple objects. There are several kinds of annotations used in their system including regions, effects, lock-state association, lock usage and concurrency policy.

Regions are names for extensible groups of fields (Java instance and class variables). Publicly visible regions represent the abstract data manipulated by the abstract operations of the class.

There are two techniques for aggregating state from many objects into a single region, where the first is aggregation through uniqueness and the second is achieved by parameterizing class definitions by regions. That is, any class may be attached a formal parameter that indicating the region into which the instance of itself is aggregated. The code segment in Figure 2.5 shows an example of this situation. The `CachedThread` is defined with a formal parameter `Backbone` that aggregates the instance of class `CachedThread`, thus the `freelist` is aggregated into the region `this.Threads` in class `ThreadCache`.

```
class CachedThread /*@<region Backbone>*/ extends Thread {
  ......
}

public class ThreadCache {
  protected Cachedthread /*@<this.Threads>*/ freelist;
}
```

Figure 2.5: Aggregation through parameterization.

Effect annotations basically shows that (1) which state is affected and (2) how it is affected. There are two effect annotations: "`reads`" and "`writes`" that are able to show the different effects that may happen in the body.

Programmers are able to identify shared regions and describe how they are to be protected by introducing lock annotations into classes, for example:

$$\texttt{/* lock } mutex \texttt{ is } field \texttt{ protects } region * /$$

This indicates that the shared *region* can only be accessed when the `lock` is held, where the `lock` is referenced by $field$ with an abstract name $mutex$. Moreover, the $field$ can only be either the receiver or an immutable field. This requirement prevents the identity of the lock from changing.

There are two lock usage annotations.

$$\texttt{/* requires } mutex_1, ...mutex_n \texttt{ */}$$

Callers of such a method must acquire the named locks before invoking it.

In addition, locks can be method return values, which is generally specified using

`/* returns lock` *mutex* `*/`

The concurrency policy of a class implementation specifies which methods have potential executions that can be safely interleaved. They distinguish between two uses of concurrency policy: guiding policy that restricts the implementation of a class, and client policy that restricts the use of particular implementation.

The *guiding concurrency policy* of a class sets an upper bound on the extent of interleaving for the methods of a class and its subclasses. That is, the guiding concurrency policy defines safe or unsafe concurrency for a class implementation. The *client policy* specifies pairs of methods that clients of the class are (and are not) allowed to invoke concurrently, constraining the design decisions of clients. The following annotations to the implementation of a method m

`/*@ safe with` $method_1, ..., method_n$ `*/`

declares that methods m, $method_1$, ..., $method_n$ may be invoked concurrently by clients. The client policy thus describes to clients of a class which potentially unsafe method interactions they avoid. A potential race condition exists if a conservative analysis cannot assure consistent regard to the policy, i.e., that unsafe method pairs are not used concurrently by a client.

Greenhouse's concurrency assurance has some advantages over previous ones. For example, both ESC/Java [29, 30] and `rccjava` [13] associate fields directly with locks, while this assurance associates locks with abstract regions, enabling retention of encapsulation and support for program evolution and subclassing. Furthermore, neither ESC/Java nor `rccjava` is able to represent unique pointer, but this assurance uses uniqueness to aggregate state.

2.4 Summary

In this chapter, we go though three static concurrency analyses that are closely related to our work. Among the three, both Boyapati's ownership system and Flanagan's `rccjava` are type-based approaches, and Greenhouse's assurance primarily resemble type systems. Boyapati's system deal with both data races and deadlocks, while Flanagan's and Greenhouse's only concern the data races. Furthermore, the Boyapati's owners-as-locks system is pretty neat and powerful, but more restrictive than the other two.

From the next chapter, we start to introduce the permission system. At first, we define a simple object-oriented language with annotations and then show the evaluation based on the operational semantics.

Chapter 3

Formal

We use a simple object-oriented language extended from ClassicJava and ConcurrentJava [13, 10, 42] with some additional annotations [28] and parameterized classes. In this chapter, we give the language syntax and its operational semantics.

3.1 Syntax

We give the syntax of this language in Figure 3.1. A shared-memory program consists of several class definitions *defn* which further includes level, field and method definitions. Each class is parameterized by zero or more formal guard objects g which may be used in the annotations for fields and methods. A class may include a "`thread_local`" modifier which means the instances of this class are only accessed in the thread that creates it.

There are two kinds of fields: *regular fields* and *data groups* (also called *regions*). The data groups are names for extensible groups of fields. Publicly visible data groups represent the abstract data manipulated by the abstract operations of the class. One data group may be "`in`" another. That is to say, data groups can be nested.

There are several field annotations:

- "`final`": This field is not changeable after its first assignment.
- "`guarded_by` *guard*": This field (or data group) is protected by the *guard*. In other words, any access to this field (or data group) should be inside of a synchronized block holding the *guard* which could be either the *self* object or a formal guard parameter in the class definition. It's obvious that the *guard* is immutable.

$$
\begin{array}{l}
P ::= \overline{\mathit{defn}} \\
\mathit{defn} ::= [\texttt{thread\_local}]\ \texttt{class}\ \mathit{cn}\ \texttt{extends}\ \mathit{cn}'\{\overline{\mathit{level}}\ \overline{\mathit{field}}\ \overline{\mathit{meth}}\} \\
\mathit{level} ::= \texttt{Level}\ \mathit{lv} \\
\mathit{field} ::= [\mathit{pa}]^*\ \mathit{cn}\ f\ \ [\mathit{fa}]^* \mid \texttt{group}\ \ G\ [\mathit{fa}]^* \\
\mathit{meth} ::= \mathit{cn}\ m(\overline{\mathit{arg}})\ \ [\mathit{ma}]^*\ \{e\} \\
\mathit{pa} ::= \texttt{unique} \mid \texttt{shared} \mid \texttt{nonNull} \mid \texttt{maybeNull} \\
\mathit{fa} ::= \texttt{final} \mid \texttt{guarded\_by}\ \mathit{guard} \mid \texttt{in}\ \ G \mid\ \ \texttt{lessThan}\ \mathit{lv} \mid\ \ \texttt{greaterThan}\ \mathit{lv} \\
\mathit{ma} ::= \texttt{reads}\ \ (\overline{e}) \mid \texttt{writes}\ \ (\overline{e}) \mid \texttt{requires}\ \ (\overline{\mathit{lock}}) \mid \texttt{uses}\ \ (\overline{\mathit{lock}}) \mid \texttt{from}(\overline{e}) \mid \mathit{lock}\texttt{<}\mathit{lock} \\
\mathit{guard} ::= \texttt{this} \mid g \\
\mathit{lock} ::= f\ \mid \mathit{guard} \\
\mathit{cn} ::= C\texttt{<}\overline{g}\texttt{>} \\
\mathit{arg} ::= [\mathit{pa}]\, \mathit{cn}\ x \\
e ::= \texttt{new}\ \ C \mid x \mid e.f\ \mid e.f\,\texttt{=}e \mid e\,\texttt{;}\,e \mid \texttt{let}\ \ x\texttt{=}e\ \ \texttt{in}\ \ e \mid \texttt{if}\ \ e\texttt{==}e\ \ \texttt{then}\ \ e\ \ \texttt{else}\ \ e \\
\qquad \mid e\ \ \mathit{op}\ \ e \mid e.m(\overline{e}) \mid e.C\#m(\overline{e}) \mid \texttt{synch}\ \ e\ \ \texttt{do}\ \ e \mid \texttt{fork}\ \ (\overline{x})\, e \mid \texttt{skip} \mid \mathit{ie} \\
\mathit{ie} ::= \texttt{hold}\ \ o\ \ \texttt{do}\ \ e \mid e||...||e
\end{array}
$$

$C \in$ class names
$lv \in$ level names
$f\ \in$ field names
$m \in$ method names
$o \in$ object references

Figure 3.1: Syntax

- "`in` G": This field (or group) is nested in a group G.

- "`lessThan` lv" and "`greaterThan` lv": The object that pointed to by the field has a lower level and a higher level than the level lv respectively.

The pointer annotations apply to any pointer, not only reference fields, but also formal parameters and the receiver object:

- "`unique`": This pointer is either the null pointer or the only reference to the pointed-to object.

- "`shared`": This pointer is either the null pointer or it is not the only reference to its pointed-to object.

- "`nonnull`": This pointer is not null, but a `nonnull` annotated field reference can only be assumed nonnull after the constructor call.
- "`maybenull`": This pointer may be null or not.

Different pointer annotations may be intermixed.

Method annotations indicate the effects and requirements of the method invocation:

- "`reads` $(\overline{e})$" and "`writes` $(\overline{e})$": These two are the classic method effects indicating the possible read and write accesses to the $\overline{e}$ in the method body respectively.
- "`requires` $(\overline{lock})$": The sequence of $\overline{lock}$ are required to be held at the method entry.
- "`uses` $(\overline{lock})$": The sequence of $\overline{lock}$ may be acquired by synchronization expressions inside of the method body.
- "$lock < lock'$ ": The object *lock* has a lower level than the $lock'$.
- "`from(e)`": This represents a return value that *borrows from* the named effect when the method invocation ends.

Expressions include pure allocation[1], variable read, field read, field write, sequential composition, local declaration, conditional [2], arithmetic operation, method dispatch, static method call, synchronization, thread spawn [3] and skip. The hold expression and parallel composition are internal expressions that will be used not in regular code, but in internal evaluation.

[1]A regular allocation is implemented by a pure allocation followed by a constructor invocation.

[2]The condition part of the conditional expression is a comparison of two expressions whether or not they are pointing to the same location.

[3]The expression `fork` $(\overline{x})\, e$ spawns a new thread with arguments $\overline{x}$ to evaluate e. The evaluation is performed only for its effect; the result of e is never used. Note that the Java mechanism of starting threads using code of the form `Thread t=...; t.start();` can be expressed equivalently in here as `fork (t) t.start();`.

3.2 Operational Semantics

Annotations are only used to do analysis, thus they are non-executable and hence have no effect on the fundamental compiler and runtime environment. Therefore, our evaluation is only based on the original code without any annotation. Here, we use an operational semantics to demonstrate the expression evaluation.

The thread-local operational semantics without thread spawn is given in terms of a small-step evaluation which has a form:

$$(\mu; e) \xrightarrow{p} (\mu'; e')$$

Given a memory μ, an expression e can be one-step evaluated to e' in a thread p while side-effecting memory to μ'. Here, the p is maintained as an integer acting as a thread number and the memory is defined as a mapping from a location (a pair of object address and field name) to the value at that location:

$$\mu \in M = (O \times F) \rightharpoonup O$$

In addition, we suppose the (allocated and unallocated) object space is partitioned by class such that "class(o)" always gives the precise type for any object reference o and we assume an unlimited supply of objects of any type.

The evaluation rules are partitioned into two groups: non-concurrency and concurrency related. They are in Figure 3.2 and 3.4 respectively.

3.2.1 Non-concurrency Related Evaluation

The rules that simply move the evaluation to a subexpression are collected into E-Common using an evaluation context $\mathcal{E}[\bullet]$ that shows which subexpression will be evaluated next [4].

[4]The \$0 used in E-Skip represents a null pointer.

The auxiliary rules for fields(C) and mbody(C, m) are defined in Figure 3.3 where the high-level annotations are removed. The CT is a class table that takes a class name and then returns its definition. α is the method types in permission (we get back to this in the next chapter).

$$\begin{array}{lcl}\mathcal{E}[\bullet] & ::= & \bullet.f \mid \bullet.f\texttt{=}e \mid o.f\texttt{=}\bullet \mid \bullet\texttt{;}e \mid \texttt{let } r\texttt{=}\bullet\texttt{ in } e \mid \texttt{if } \bullet\texttt{==}e\texttt{ then } e\texttt{ else } e \\ & & \mid \texttt{if } o\texttt{==}\bullet\texttt{ then } e\texttt{ else } e \mid \bullet\ op\ e \mid o\ op\ \bullet \mid \bullet.m(\overline{e}) \mid o.m(\overline{o}, \bullet, \overline{e}) \\ & & \mid \bullet.C\#m(\overline{e}) \mid o.C\#m(\overline{o}, \bullet, \overline{e}) \mid \texttt{synch } \bullet\texttt{ do } e \mid \texttt{hold } o\texttt{ do } \bullet\end{array}$$

E-Common
$$\frac{(\mu; e) \xrightarrow{p} (\mu'; e')}{(\mu; \mathcal{E}[e]) \xrightarrow{p} (\mu'; \mathcal{E}[e'])}$$

E-New
$$\frac{\forall f \in \text{fields}(C).(o, f) \notin \text{Dom}(\mu)}{(\mu; \texttt{new } C) \xrightarrow{p} (\mu[(o, f) \mapsto \$0 \mid f \in \text{fields}(C)]; o)}$$

E-Arithmetic
$$\frac{o_3 = o_1\ op\ o_2}{(\mu; o_1\ op\ o_2) \xrightarrow{p} (\mu; o_3)}$$

E-Read
$$\frac{o' = \mu(o, f)}{(\mu; o.f) \xrightarrow{p} (\mu; o')}$$

E-Write
$$\frac{\mu' = \mu[(o_1, f) \mapsto o_2]}{(\mu; o_1.f\texttt{=}o_2) \xrightarrow{p} (\mu'; o_2)}$$

E-Dispatch
$$\frac{\text{mbody}(\text{class}(o_0), m) = (\overline{x}, e, \alpha) \qquad |\overline{x}| = |\overline{o}|}{(\mu; o_0.m(\overline{o})) \xrightarrow{p} (\mu; e[\texttt{this} \mapsto o_0, \overline{x \mapsto o}])}$$

E-Call
$$\frac{\text{class}(o_0) \preccurlyeq C \qquad \text{mbody}(C, m) = (\overline{x}, e, \alpha) \qquad |\overline{x}| = |\overline{o}|}{(\mu; o_0.C\#m(\overline{o})) \xrightarrow{p} (\mu; e[\texttt{this} \mapsto o_0, \overline{x \mapsto o}])}$$

E-Local
$$(\mu; \texttt{let } x\texttt{=}o_1\texttt{ in } e_2) \xrightarrow{p} (\mu; [x \mapsto o_1]e_2)$$

E-IfTrue
$$\frac{o_1 = o_2}{(\mu; \texttt{if } o_1\texttt{==}o_2\texttt{ then } e_3\texttt{ else } e_4) \xrightarrow{p} (\mu; e_3)}$$

E-IfFalse
$$\frac{o_1 \neq o_2}{(\mu; \texttt{if } o_1\texttt{==}o_2\texttt{ then } e_3\texttt{ else } e_4) \xrightarrow{p} (\mu; e_4)}$$

E-Seq
$$(\mu; o_1\texttt{;}e_2) \xrightarrow{p} (\mu; e_2)$$

E-Skip
$$(\mu; \texttt{skip}) \xrightarrow{p} (\mu; \$0)$$

Figure 3.2: Non-concurrency-related evaluation rules.

$$\text{fields}(\texttt{Object}) = \{\}$$

$$\frac{CT(C) = \texttt{class}\ C\ \texttt{extends}\ C'\{\overline{\mathit{field}}, \overline{\mathit{meth}}\} \qquad \text{fields}(C') = \{\overline{f}\}}{\text{fields}(C) = \{\overline{f}, \text{fieldnames}(\overline{\mathit{field}})\}}$$

$$\frac{CT(C) = \texttt{class}\ C\ \texttt{extends}\ C'\{\overline{\mathit{field}}, \overline{\mathit{meth}}\} \qquad (C\ m(\overline{x})\ \{e\}) \in \overline{\mathit{meth}}}{\text{mbody}(C, m) = (\overline{x}, e, \alpha)}$$

$$\frac{CT(C) = \texttt{class}\ C\ \texttt{extends}\ C'\{\overline{\mathit{field}}, \overline{\mathit{meth}}\} \qquad m \text{ is not defined in } \overline{\mathit{meth}}}{\text{mbody}(C, m) = \text{mbody}(C', m)}$$

Figure 3.3: Auxiliary definition.

3.2.2 Concurrency Related Evaluation

This group includes rules for thread spawn, parallel composition as well as lock acquirement and release.

Any thread can lock or unlock objects, but one object can only be locked by at most one thread at a time, thus an object is free to be locked only when it is in *unlocked* status. In order to model this feature, we add an implicit field "Lock" to every object to represent whether this object is locked or not. If the Lock field of an object is a null pointer, then it is in the *unlocked* state, otherwise it is in the *locked* state and the pointed-to object by the Lock field is the thread that is currently holding it.

For a synchronization expression `synch` e_1 `do` e_2, it is required to evaluate the expression e_1 to get the lock object o_1 before entering into the synchronized block e_2. In fact, there may be three possibilities for the acquirement operation on the lock o_1:

- The o_1 is in the *unlocked* state with a null pointer Lock field, thus it is free to be locked;
- The o_1 is in the *locked* state and it is held with the Lock field pointing to the

current thread object;

- The o_1 is in the *locked* state but it is held by some thread other than the current one. In this case, this thread will be blocked until the lock is released.

The last situation does not fit for any evaluation rule, in other works, it is stuck. A deadlock condition shows up if all the parallel threads get stuck.

The E-Acquire only applies for the first case: it updates the Lock field from null to the thread object. The "Thread" serves as a function that takes a thread number and returns the thread object back. After acquiring the lock, we continue to evaluate the synchronized block by E-Hold which witnesses the lock o_1 is held by the current thread [5]. The lock should be released when exiting the synchronization and the rule E-Release does reverse the operations of the E-Acquire.

Basically, the E-Re-Acquire will: (1) first make sure the lock is held by the current thread; (2) then directly evaluate the synchronized block without any effect on the lock (neither acquire nor release). This process matches to Java's re-entrant monitors, but we avoid the need to count multiple entrances because the evaluation rule drops two monitor actions.

The rule E-Fork together with the E-Par are not thread local any more. They are global and we need to list all the parallel threads that are currently active at runtime by a n parallel composition $e_1||...||e_n$. E-Fork applies to a thread spawn expression `fork` $(\overline{x})\, e$ inside of the thread i that in parallel with some others. It will be replaced as a `skip` expression in its original thread to hold a place and spawn a new thread to evaluate the e with remembering its parent thread by a field "Parent". Therefore, the total thread number will be increased by one.

[5]As mentioned before, the `hold` o_1 `do` e is an internal expression that cannot be used by programmers.

Evaluating the global parallel composition $e_1||...||e_n$ without a thread spawn is non-deterministic: any thread may be evaluated one step further, based on which we use the rule E-Par. Since we pick the unstructured parallelism model, we don't have some rule about thread *join* or *elimination*.

E-Acquire

$$\frac{\mu(o_1, \text{Lock}) = \$0 \qquad \text{Thread}(p) = o_T \qquad \mu' = \mu[(o_1, \text{Lock}) \mapsto o_T]}{(\mu; \texttt{synch}\ o_1\ \texttt{do}\ e_2) \xrightarrow{p} (\mu'; \texttt{hold}\ o_1\ \texttt{do}\ e_2)}$$

E-Re-Acquire

$$\frac{\mu(o_1, \text{Lock}) = \text{Thread}(p)}{(\mu; \texttt{synch}\ o_1\ \texttt{do}\ e_2) \xrightarrow{p} (\mu; e_2)}$$

E-Hold

$$\frac{\mu(o_1, \text{Lock}) = \text{Thread}(p) \qquad (\mu; e_2) \xrightarrow{p} (\mu'; e_2')}{(\mu; \texttt{hold}\ o_1\ \texttt{do}\ e_2) \xrightarrow{p} (\mu'; \texttt{hold}\ o_1\ \texttt{do}\ e_2')}$$

E-Release

$$\frac{\mu(o_1, \text{Lock}) = o_T \qquad \text{Thread}(p) = o_T \qquad \mu' = \mu[(o_1, \text{Lock}) \mapsto \$0]}{(\mu; \texttt{hold}\ o_1\ \texttt{do}\ o_2) \xrightarrow{p} (\mu'; o_2)}$$

E-Fork

$$\frac{\mu' = \mu[(o_{T_{n+1}}, \text{Parent}) \mapsto o_{T_i}]}{(\mu; e_1||...||\mathcal{E}[\texttt{fork}\ (\overline{x})\, e]||...||e_n) \xrightarrow{i} (\mu'; e_1||...||\mathcal{E}[\texttt{skip}]||...||e_n||e)}$$

E-Par

$$\frac{(\mu; e_i) \xrightarrow{i} (\mu'; e_i')}{(\mu; e_1||...||e_i||...||e_n) \xrightarrow{i} (\mu'; e_1||...||e_i'||...||e_n)}$$

Figure 3.4: Concurrency-related evaluation rules.

3.3 Summary

This chapter first provides the syntax of a simple object-orient language with some annotations, then explains the meanings of each annotation. The operational semantics is included to demonstrate the expression evaluation.

Chapter 4
Permission

4.1 Permission

A *permission* (Π) is a token associated with some piece of state in a program and permissions are granted to permit certain operations [5, 6, 7]. "Fractions" are attached to distinguish reads from writes, and the "nesting" relation depicts one piece of permission is logically nested in another piece of permission. We introduce different permission elements according to different purposes.

4.1.1 Empty Permission

An *empty permission* does not grant any permission and prohibit most of the operations in programs.

$$\Pi ::= \emptyset \mid \ldots$$

4.1.2 Unit Permission

Assuming we are able to access a field f of an object ρ which is currently pointing to another object ρ', then we need a *unit permission*:

$$\Pi ::= \ldots \mid \rho.f \rightarrow \rho' \mid \ldots$$

There is at most one unit permission associated with every field in the heap. A unit permission gives right to access the state, either reading or writing. A read operation to this field is not able to affect this unit permission, but a write may update the pointed-to object to a new one ρ'':

$$\{\rho.f \ \rightarrow \rho'\} \Longrightarrow \boxed{\text{some write operation}} \Longrightarrow \{\rho.f \ \rightarrow \rho''\}$$

Property 4.1.2.1 *It is not permitted to update a field value unless the corresponding unit permission for that field is granted.*

4.1.3 Compound Permission

We use the comma operation (,) to combine different permissions. A *compound permission* Π, Π' is constituted from two sub-permissions and gives one all the rights associated with either of them being compounded.

$$\Pi ::= \ldots \mid \Pi, \Pi \mid \ldots$$

It's worth a mention that not all permissions can be combined to constitute a compound one, for example, a unit permission $\rho.f \ \rightarrow \rho'$ cannot be combined with another $\rho.f \ \rightarrow \rho''$ if the ρ' and ρ'' are different. In other words, they are "conflict". We have consistency rules in Chapter 6 to decide whether two permissions can be combined or not. Moreover, the comma operator (,) is commutative and associative with using the empty permission $\emptyset$ as the identity.

$$\Pi, \emptyset \equiv \emptyset, \Pi \equiv \Pi$$

4.1.4 Fractional Permission

Although a unit permission permits both read and write operations, it's better to distinguish them since the two operations play different roles in multithreaded programs. For instance, a non-synchronized write operation in one thread must "invalidate" any access (both read and write) in its parallel threads, otherwise a data race happens.

Our fractional permission system is intuitively good at distinguishing reads from writes. This is one of the most important advantages over other type systems. We build the *fractional permission* which scales the permission by a fraction ξ:

$$\Pi ::= \ldots \mid \xi\Pi \mid \ldots$$

The ξ represents a positive fraction (rational number) from zero exclusively to one inclusively and the usage of fraction together with the compound permission gives an elegant way to split permissions. Let's take the below transfer for example:

$$\Pi \equiv \texttt{1/2}\Pi, \texttt{1/2}\Pi \equiv \texttt{1/2}\Pi, (\texttt{1/4}\Pi, \texttt{1/4}\Pi) \equiv \texttt{3/4}\Pi, \texttt{1/4}\Pi$$

A permission Π can be split into two fractional permissions both of which are associated with a fraction `1/2`. Then one of them split again to get two fractional permissions with a smaller fraction (`1/4`). This is demonstrated in Figure 4.1 by treating the original Π as a disk.

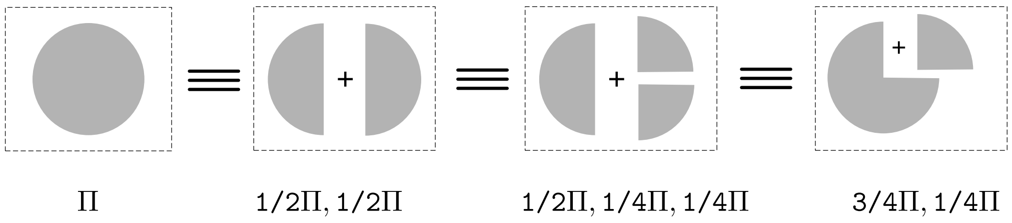


Figure 4.1: Fractional Permission.

With the fractional permission, we use the unit permission to act as a write permission, while a fractional unit permission (fractional permission for short) represents a read permission. For example, the $\rho.f \rightarrow \rho'$ and $\xi\rho.f \rightarrow \rho'$ are the write and read permission to access the field f of object ρ respectively.

It's worthwhile to mention that both of the two fractional permissions $\texttt{1/2}\rho.f \rightarrow \rho'$ and $\texttt{1/4}\rho.f \rightarrow \rho'$ grant a read permission to access the $\rho.f$, however they need different counterparts to recover a write permission: the former needs a $\texttt{1/2}\rho.f \rightarrow \rho'$, while the latter requires the $\texttt{3/4}\rho.f \rightarrow \rho'$.

Property 4.1.4.1 *A unit permission acts as a write permission and it can be split into several read permissions with the same key but different fractions. On the other side, several read permissions with the same key are able to be combined to recover a write permission if the sum of their fractions is one.*

4.1.5 Existential Permission

Since the permission analysis is totally static, we are not able to decide the actual object locations before the runtime. Thus, we use existential variables to represent unknown objects. For example, if we want to say that we have a write permission to $\rho.f$ as well as a write permission to the field f' (assuming a null pointer) of its pointed-to object, then we give a permission using an existential variable: $\exists r\,.\,(\rho.f \rightarrow r, r.f' \rightarrow \$0)$. This is called *existential permission.*

$$\Pi ::= \ldots \mid \exists r\,.\,(\Pi) \mid \ldots$$

An existential permission can be unpacked in the standard way (using skolemization).

4.1.6 Fact and Conditional Permission

A *conditional permission* $(\Gamma)?(\Pi) : (\Pi')$ shows that either Π or Π' is present depending on whether the Γ is true or not.

$$\Pi ::= \ldots \mid (\Gamma)?(\Pi) : (\Pi') \mid \ldots$$

Here, the Γ is represented by boolean formulae to show a *fact* which could be

- standard boolean logic: true, $\neg(\Gamma)$ and $\Gamma \wedge \Gamma$;

- type assertions ($\rho \in C$): ρ has the type C or its subtype;
- reference equalities ($\rho = \rho$): two references are pointing to the same location;
- orders between lock objects ($\rho < \rho'$): lock object ρ has a lower level than ρ';
- orders between lock objects and levels ($\rho < \rho'.lv$ or $\rho > \rho'.lv$): lock object ρ has a lower (or higher) level than the level lv of the ρ';
- nesting facts ($\Pi \prec \rho.f$): permission Π is nested in a unit permission for the $\rho.f$;
- object type predicates $p(\overline{\rho})$: The class invariant with regarding to a sequence of references ρ.

Most of them are straightforward except the last two which will be discussed later. In our system, facts are also considered as permissions.

$$\Pi ::= \ldots \mid \Gamma \mid \ldots$$

Given the truth value of the Γ part, a conditional permission can be "reduced" to either its true- or false-clause. For example:

$$\begin{array}{lcl} \Gamma, (\Gamma)?(\Pi) : (\Pi') & \equiv & \Gamma, \Pi \\ \neg(\Gamma), (\Gamma)?(\Pi) : (\Pi') & \equiv & \neg(\Gamma), \Pi' \end{array}$$

4.1.7 Linear Implication

A *linear implication* $\Pi_1 \multimap \Pi_2$ indicates that one has the rights of the consequent Π_2, except for the ones of the antecedent Π_1.

$$\Pi ::= \ldots \mid \Pi_1 \multimap \Pi_2 \mid \ldots$$

In order to answer why we use this kind of permission, we introduce two concepts first: permission nesting and carving. The nesting is one of the most important hallmarks of our permission system.

Definition 4.1.7.1 (Permission Nesting) *A permission can be nested in a unit permission and this nesting relation is expressed as a fact (formula):*

$$\Pi \prec \rho.f$$

Permission nesting indicates that anyone who has the nester permission *also has the* nested permission*.*

The nesting relation basically has two functions:

- It intuitively indicates the protection relation. Since the nested permissions are not available unless the nester permission is granted, we may consider the nester permission as a lock to protect some critical nested permissions.
- It is able to show the encapsulation capability. The nester may be thought as a data group [43] to hide the nested permissions which should not be directly exposed to clients.

Assuming that the Π and $\rho.f \rightarrow \rho'$ are the nested and nester permissions respectively, then after performing a nesting:

$$\Pi, \rho.f \rightarrow \rho' \rightsquigarrow (\Pi \prec \rho.f), \rho.f \rightarrow \rho'$$

the original Π is "gone", but a nesting fact is present. Actually, the "same" unit permissions (acting as a nester) before and after the nesting operation are different. The latter one has a "bigger size" than the former since it is "enlarged" by containing a new nested permission. The first transfer in Figure 4.2 shows this situation.

After the nesting, the nested permission is not directly present any more. In order to get it to perform some operations, we need to carve it out:

$$(\Pi \prec \rho.f), (\rho.f \rightarrow \rho') \equiv (\Pi \prec \rho.f),\ \Pi,\ \Pi \mathrel{-\!\!+} (\rho.f \rightarrow \rho')$$

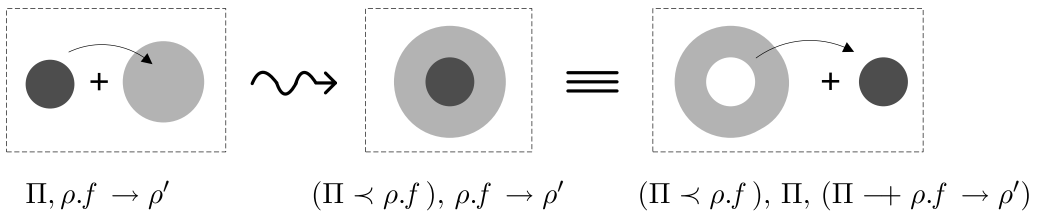

+
+
$\Pi, \rho.f \rightarrow \rho'$
$(\Pi \prec \rho.f), \rho.f \rightarrow \rho'$
$(\Pi \prec \rho.f), \Pi, (\Pi \multimap \rho.f \rightarrow \rho')$

Figure 4.2: Permission nesting and carving.

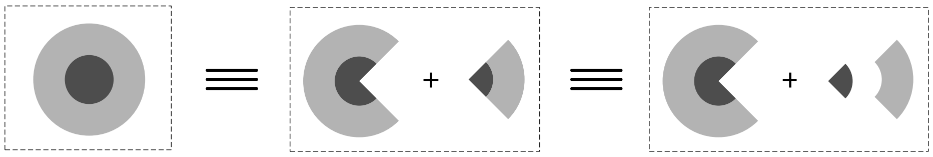
Definition 4.1.7.2 (Carving) *A nested permission* Π *can be carved out from its nester permission* $\rho.f \rightarrow \rho'$ *for some* ρ' *if the nesting fact* $\Pi \prec \rho.f$ *is present and* Π *hasn't been carved out yet.*

The carving is not just a reversed nesting operation. As mentioned before, the nester permission is "enlarged" once a nesting happens, but it cannot shrink after being carved out the nested permission. Instead, there will be a "hole" expressed by a linear implication form of permission: That is to say, the nesting is undoable and the carving cannot be considered as an *anti*-nesting operation (see the one-way arrow in Figure 4.2). However, there does exist an *anti*-carving transfer: the carved out nested permission can be filled back into the implication form of its nester. (see the $\equiv$ notation in Figure 4.2).

Together with the usage of fraction, a fractional nester permission can be carved out a fractional nested permission:

$$(\Pi \prec \rho.f), (\xi\rho.f \rightarrow \rho') \equiv (\Pi \prec \rho.f),\ \xi\Pi,\ \xi\Pi \multimap (\xi\rho.f \rightarrow \rho')$$

Figure 4.3 demonstrates this situation. In the first step, a nester permission is split into two parts, then a carving operation happens on the part with the smaller fraction.



$(\Pi \prec \rho.f\,),\ \rho.f \ \rightarrow \rho'$

$(\Pi \prec \rho.f\,),\ \texttt{3/4}\rho.f \ \rightarrow \rho',$ $\texttt{1/4}\rho.f \ \rightarrow \rho'$

$(\Pi \prec \rho.f\,),\ \texttt{3/4}\rho.f \ \rightarrow \rho',$ $\texttt{1/4}\Pi,\ \texttt{1/4}\Pi \twoheadrightarrow \texttt{1/4}\rho.f \ \rightarrow \rho'$

Figure 4.3: Fractional carving.

4.1.8 Formal Syntax

The formal syntax of the permission type system is given in below:

object reference	ρ	$::=$	$o \mid r$
key	k	$::=$	$\rho.f$
fraction	ξ	$::=$	$\texttt{1} \mid \texttt{1/2} \mid z$
fact	Γ	$::=$	$\text{true} \mid \neg(\Gamma) \mid \Gamma \wedge \Gamma \mid \rho \in \text{C} \mid \rho = \rho \mid \Pi \prec \text{k} \mid \text{p}(\overline{\rho}) \mid \Gamma_<$
order fact	$\Gamma_<$	$::=$	$\rho < \rho \mid \rho < \rho.lv \mid \rho > \rho.lv$
permissions	Π	$::=$	$\emptyset \mid \Gamma \mid k \rightarrow \rho \mid \xi\Pi \mid \exists r\,.\,(\Pi) \mid (\Gamma)?(\Pi) : (\Pi)$
			$\mid \Pi \twoheadrightarrow \Pi \mid \Pi, \Pi$

Here, o and r are used to range literal object addresses and object reference variables respectively.

4.2 Annotation

Annotations are attached to programs to indicate the *design intent* [27]. They are non-executable and hence have no effects on the fundamental compiler and runtime environment. Annotations are considered as high level descriptions of the low level code, thus they may help people (both programmers and maintainers) understand the complicated program. However, high-level annotations cannot be used to do program analysis directly, they must be translated into some machine understandable form. Here, we use the low-level permissions to represent annotations in the code.

4.2.1 Pointer/Field Annotations

We have several kinds of pointer (including field) annotations: *data groups*, *nullity*, *uniqueness*, *immutable*, *lock level*, *guards*.

Data Groups

A "data group" is modeled by a field with an uninteresting type. In our system, we represent it by a special field which is always null. And we use the permission nesting to simulate the abstraction of data groups, where permissions for fields in a data group are nested in the permission for the data group. In addition, we define an `All` data group in the `Object` class that will be inherited by every class. Any field or data group that are not explicitly annotated by an "`in` G" will be implicitly "`in All`" by default.

A code segment in Figure 4.4 shows the usage of data groups. A `Location` data group is defined in the `Point` class. It includes two fields `x` and `y`, and is further included by another data group `Appearance`.

```
class Point {
  group Appearance;
  group Location in Appearance;
  int x in Location;
  int y in Location;
  ...
```

```
class Color3DPoint extends Point {
  group Color in Appearance;
  int color in Color;
  int z in Location;
  ...
```

Figure 4.4: Code segment: data groups.

The permission nesting will be used to model the "`in`" relation either between a field and a data group or between two data groups. There are four permission nesting facts for fields `x` and `y`, data groups `Location` and `Appearance` respectively. Assuming the $r_{\texttt{this}}$ is an object variable for the self object, then

$$\begin{gathered} r_{\texttt{this}}.\texttt{x} \rightarrow \texttt{int} \prec r_{\texttt{this}}.\texttt{Location}, \\ r_{\texttt{this}}.\texttt{y} \rightarrow \texttt{int} \prec r_{\texttt{this}}.\texttt{Location}, \\ r_{\texttt{this}}.\texttt{Location} \rightarrow \$0 \prec r_{\texttt{this}}.\texttt{Appearance}, \\ r_{\texttt{this}}.\texttt{Appearance} \rightarrow \$0 \prec r_{\texttt{this}}.\texttt{All} \end{gathered}$$

Any subclass may extend its superclass by adding new fields and data groups as well, but the existent nesting facts must be inherited. For instance, the subclass is not allowed to make the data groups that defined at its superclass "`in`" a new data group of its own. The `Color3DPoint` is a subclass of the `Point` class with several valid data group annotations. It not only adds a new field `z` into the existent `Location` data group, but also defines a new data group `Color` which is further included by the existent `Appearance` from its superclass. Besides the nesting facts in its superclass `Point`, the additional nesting facts of `Color3DPoint` class include:

$$r_{\texttt{this}}.\texttt{z} \longrightarrow \texttt{int} \prec r_{\texttt{this}}.\texttt{Location},$$
$$r_{\texttt{this}}.\texttt{color} \longrightarrow \texttt{int} \prec r_{\texttt{this}}.\texttt{Color},$$
$$r_{\texttt{this}}.\texttt{Color} \longrightarrow \$0 \prec r_{\texttt{this}}.\texttt{Appearance}$$

Nullity

A "`nonnull`" reference variable is simply expressed by $\neg(r = \$0)$, while the "`maybenull`" annotation will be translated as a conditional permission: $(r = \$0)?(\Pi_1) : (\Pi_2)$ where the Π_1 and Π_2 respectively correspond to the two possibilities whether r is null or not. A "`nonnull`" field cannot be truly non-null until the constructor call ends. A receiver is always assumed non-null.

Uniqueness

As mentioned before, the permission type may use an existential variable to represent the pointed-to object. In general, this pointer will be packaged with facts about or permission to access some or all of the pointed-to object's fields. Different annotations are expressed by packaging different permissions and/or facts.

With linear existentials, it's easy to use permission to store unique pointers in shared state: a unique pointer is one which is packaged along with the permission to access the state pointed to. This representation seems a little weaker than the usual sense of uniqueness in which there are no other pointers to the state in the store, but our permission treatment is safe enough to simulate the uniqueness since other possible references will not be granted permission to access it. The problem is not the *existence* of aliasing pointers, but rather the *access* of the state through these aliasing pointers.

```
class Rectangle {
  unique Point tl;
  unique Point br;
  shared String name;
  ...
```

```
void setPosition( unique Point tl,
unique Point br, shared String name ) {
  this.tl = tl;
  this.br = br;
  this.name = name;
}
```

Figure 4.5: Code segment: uniqueness.

For example, we use three existential permissions to represent the two unique fields and one shared field in Figure 4.5 respectively:

$$\exists r\,.\,(r_{\texttt{this}}.\texttt{tl} \to r, (\neg(r = \$0))?(r.\texttt{All} \to \$0) : (\emptyset))$$
$$\exists r\,.\,(r_{\texttt{this}}.\texttt{br} \to r, (\neg(r = \$0))?(r.\texttt{All} \to \$0) : (\emptyset))$$
$$\exists r\,.\,(r_{\texttt{this}}.\texttt{name} \to r, (\neg(r = \$0))?(r.\texttt{All} \to \$0 \prec r.\texttt{Prot}) : (\emptyset))$$

The variable $r_{\texttt{this}}$ has the same meaning as usual. All classes inherit two distinguished data groups from `Object`: (1) `All` contains all fields (data groups) of the object that are not annotated by "`final`", "`in`" or "`guarded_by`"; (2) `Prot` contains all state "protected" by the object: those state can only be accessed when the object is being locked.

First, $r_{\texttt{this}}.\texttt{tl} \to r$ is the unit permission to write access the field `tl` which is currently a reference pointing to r, where the r is an existential variable and it could be either null or non-null. If the r is not null, then the unit permission $r.\texttt{All} \to \$0$ is also presented because the `tl` is annotated "`unique`" and there is no more permissions available for any other pointers (if any) that also refer to the r. In other words, if we have the unit permission for the $r_{\texttt{this}}.\texttt{tl}$, we will additional have the unit permission for the `All` data group of its pointed-to object (if non-null).

For a "`shared`" pointer, such as the `name` field, the permission for the `All` data group of its pointed-to object (if non-null) is nested in the `Prot` data group of its own. Intuitively, a shared pointer is one whose pointed-to object's state may be accessed from references other than the current one, therefore neither the shared pointer itself nor any other pointers is good enough to take the responsibility for its pointed-to object alone. That is, the pointed-to object has to be responsible to itself: any access to its state should be in a synchronized block holding the object itself.

The fundamental principle is: wherever you get some permission to access a unique pointer, then you have the same amount (fractional) of permission to access the state of its pointed to object (assuming non-null), because the uniqueness property excludes other references to that pointed-to object.

Parameter and return values annotated "`unique`" are handled in a simpler way: the entire permission is passed in but not returned (for a parameter), or only returned (for a return value).

The uniqueness annotations attached to the parameters are handled in the similar way. For example, the parameters of method `setPosition` in Figure 4.5 have their permission representation in these forms:

$$(\neg(r_{\texttt{tl}} = \$0))?(r_{\texttt{tl}}.\texttt{All} \rightarrow \$0) : (\emptyset),$$
$$(\neg(r_{\texttt{br}} = \$0))?(r_{\texttt{br}}.\texttt{All} \rightarrow \$0) : (\emptyset),$$
$$(\neg(r_{\texttt{name}} = \$0))?(r_{\texttt{name}}.\texttt{All} \rightarrow \$0 \prec r_{\texttt{name}}.\texttt{Prot}) : (\emptyset)$$

Here, the $r_{\texttt{tl}}$, $r_{\texttt{br}}$ and $r_{\texttt{name}}$ are the variables for the formal parameters `tl`, `br` and `name` respectively.

Immutable and Lock level

An immutable object may never be written and an immutable reference points to an immutable object.

All immutable state has a fraction that is nested into a special globally known [1]

[1]The $0 is used to express the globally known location.

"immutable" group ($r.\texttt{All} \rightarrow \$0 \prec \$0.\texttt{Immutable}$). Implicitly, every method is passed permission for a (small) fraction of this field. Thus, we do not need to declare method effects for reading immutable state. "`Final`" fields can be modeled by nesting the field permission into the immutable group. For example, the permission for a final field f is:

$$\exists r\,.\,(zr_{\texttt{this}}.f \rightarrow r \prec \$0.\texttt{Immutable})$$

Lock level annotations for fields include "`lessThan`" and "`greaterThan`" which can only be attached to a "`final`" and "`nonnull`" field since its pointed-to object may act as a lock and we don't allow the mutation. The target of these two annotations is restricted to be a level defined in the same class. Levels are invisible to programmers and they are used as intermediates to connect and show the partial order among different locks.

The code segment in Figure 4.6 shows the `checking` object is lower than the level `lv`, while `savings` is higher than `lv`, and both of them are non-null and final fields. Furthermore, with the help of `lv`, it's easy to deduce that the object pointed to by `checking` has a lower level than the one pointed to by `savings`.

To prevent deadlocks, it is required to obtain locks in an ascending order. This is similar as the deadlock detection in Boyapati's ownership type system [10], but we only assign the partial order among locks, thus there is no particular level associated to lock objects.

```
class CombinedAccount {
  final nonnull Account checking < lv;
  final nonnull Account savings > lv;
  Level lv;
  ...
```

Figure 4.6: Code segment: order level.

The permission representation for the `checking` and `savings` fields are:

$$\exists r\,.\,(z_1 r_{\mathtt{this}}.\mathtt{checking} \rightarrow r, \neg(r = \$0), r < r_{\mathtt{this}}.\mathtt{lv}) \prec \$0.\mathtt{Immutable}$$
$$\exists r\,.\,(z_2 r_{\mathtt{this}}.\mathtt{savings} \rightarrow r, \neg(r = \$0), r > r_{\mathtt{this}}.\mathtt{lv}) \prec \$0.\mathtt{Immutable}$$

Guards

Following annotations from Flanagan and others for fields [16, 13, 17], we use a field annotation "`guarded_by` *guard*" to show any access (both read and write) to that field should be protected by the *guard*.

```
class Node<g> {
 Node<g> next guarded_by g;
  ...
```

```
class LinkList {
  Node<this> head guarded_by this;
  ...
```

Figure 4.7: Code segment: guards.

Figure 4.7 shows a code segment for the class `Node` which is parameterized by a formal guard object g. This is called *ghost* by Flanagan [13]. As mentioned before, the permission nesting is intuitively good at expressing a protection relation. Thus, we build a fact here to show that the unit permission to access the `next` field is nested inside of the `Prot` data group of its guard object `g`:

$$\exists r\,.\,(r_{\mathtt{this}}.\mathtt{next} \rightarrow r) \prec r_{\mathtt{g}}.\mathtt{Prot}$$

The $r_{\mathtt{g}}$ is used as a variable for the guard object `g`. Within this nesting fact, any access to this field (either read or write) should be in a synchronized block holding the $r_{\mathtt{g}}$.

Similarly, the permission representation for the "`guarded_by`" annotated `head` field in `LinkList` class is:

$$\exists r\,.\,(r_{\mathtt{this}}.\mathtt{head} \rightarrow r) \prec r_{\mathtt{this}}.\mathtt{Prot}$$

4.2.2 Mixture of multiple pointer/field annotations

We have given each pointer/field annotation a meaning by its permission representation separately. But if more than one annotations are attached to a single field, then we need to compose their permission representation from different annotations. Figure 4.1 and 4.2 provide two tables listing some annotated fields and their permission representation.

The $r_{\texttt{this}}$ is used as the self object variable as usual and $\overline{\texttt{g}}$ are a sequence of the formal guards of the class and they are expressed by a sequence of variables $\overline{r_{\texttt{g}}}$ in the permission. The $C(...)$ is used as an invariant for the class `C` which is treated as a conjunction of some facts that represent the class properties with referring to the self object variable $r_{\texttt{this}}$ and some guard variables $\overline{r_{\texttt{g}}}$. We will reach the class invariant issue in next section.

In order to reduce the number of annotations needed in many common cases, we provide some well-chosen default annotations as well as some restrictions:

- References without uniqueness annotations are considered `unique`.
- References without nullity annotations are considered `maybenull`.
- Neither uniqueness nor nullity annotations are used for groups.
- Fields annotated with "`final`" won't use the "`guarded_by`" or "`in`" annotations any more.

4.2.3 Class Invariants

Field annotations indicate the protection mechanism for each field access and maybe its pointed-to object as well. These conditions and annotations imposed on fields are called *unary field invariants* because they are object instance invariants that involve one field at a time. They are handled by the permission nesting: field and data group permissions (except `Prot`) are always nested in some location. They may express the nullity, the enclosed permission (if any) as well as the protection mechanism.

We combine all the field and group invariants in one class as a named predicate which consists of a conjunction of nesting facts and a type assertion. This is called *class invariant* and it has a form of $C(r_{\mathtt{this}}, \overline{r_{\mathtt{g}}})$, where $r_{\mathtt{this}}$ is the self object variable and $\overline{r_{\mathtt{g}}}$ is a sequence of object variables for the formal parameters used as guards in the class definition.

```
class Node<g> {
  Node<g> next guarded_by g;
  shared Object datum;
  ......
```

```
class LinkList {
  Node<this> head guarded_by this;
  ......
```

Figure 4.8: Code segment: class invariant

Figure 4.8 shows an annotated code segment for `LinkList` and `Node` classes including all their fields. By combining annotations (the `next` and `head` are unique pointers by default), the unary field invariants for the `next` and `datum` fields in class `Node` are:

$$\begin{aligned}\Gamma_{\mathtt{next}} &= \exists r\,.\,(r_{\mathtt{this}}.\mathtt{next} \to r, (\neg(r = \$0))?(r.\mathtt{All} \to \$0, \mathit{Node}(r, r_{\mathtt{g}})) : (\emptyset)) \prec r_{\mathtt{g}}.\mathtt{Prot}\\ \Gamma_{\mathtt{datum}} &= \exists r\,.\,(r_{\mathtt{this}}.\mathtt{datum} \to r, (\neg(r = \$0))?(r.\mathtt{All} \to \$0 \prec r.\mathtt{Prot}, \mathit{Object}(r)) : (\emptyset))\\ &\quad \prec r_{\mathtt{this}}.\mathtt{All}\end{aligned}$$

Both $\mathit{Node}(r, r_{\mathtt{g}})$ and $\mathit{Object}(r)$ showing above are the class invariants with referring the variables in the parentheses. Based on these two unary field invariants, the class invariant for `Node` is defined as:

$$\mathit{Node}(r_{\mathtt{this}}, r_{\mathtt{g}}) = \Gamma_{\mathtt{next}} \wedge \Gamma_{\mathtt{datum}} \wedge r_{\mathtt{this}} \in \mathtt{Node}$$

The last conjunct is the static type assertion for the self object.

The class invariant of `LinkList` has a similar structure except it only depends on the $r_{\mathtt{this}}$ since it does not have any formal class parameter.

$$\mathit{LinkList}(r_{\mathtt{this}}) = \Gamma_{\mathtt{head}} \wedge r_{\mathtt{this}} \in \mathtt{LinkList}$$

where the unary field invariant $\Gamma_{\mathtt{head}}$ is:

$$\Gamma_{\mathtt{head}} = \exists r\,.\,(r_{\mathtt{this}}.\mathtt{head} \to r, (\neg(r = \$0))?(r.\mathtt{All} \to \$0, \mathit{Node}(r, r_{\mathtt{this}})) : (\emptyset)) \prec r_{\mathtt{this}}.\mathtt{Prot}$$

Thread-Local

Large multithreaded programs typically have sections of code that operate on data that is not shared across multiple threads. In particular some objects are only accessible by the thread that create them. In this situation, we may use a "`thread_local`" modifier for this class, such that any access to its fields should only be in the thread that creates this instance. Objects used in this fashion require no synchronization and should not need to have locks guarding their fields.

$$r_{\texttt{this}}.\texttt{All} \rightarrow \$0 \prec r_{\texttt{thisThread}}.\texttt{Prot}$$

We use the above nesting fact to show that the state of this thead_local object is protected by the $r_{\texttt{thisThread}}$ which is a variable for the *current thread*. Here, we require that the object is created in the *current thread*, where the unit permission to the `Prot` data group of the thread object is automatically granted when this thread starts.

Raw and Cooked

The class invariant is an abstract representation indicating instances of this class should hold some properties. However, these properties may not be true unless the object is well established. For example, a field annotated `nonnull` may not actually be a non-null pointer before the end of its constructor call. Especially, our evaluation always treats a regular object allocation as a pure object allocation followed a constructor call. Right after the pure object allocation, all of the reference fields are null pointers (see the evaluation rule E-New in the previous chapter). Therefore, it's necessary to mark whether or not the object is well established or not.

The class invariant must be established by the constructor which is modeled by a method that takes the permissions for each field individually and returns the permission for the `All` data group after establishing the class invariant for its own and all superclasses. The permission to `All` and the class invariant jointly imply that all fields are consistent with their type.

It's forced that the class invariant can only be established when the constructor call is finished. But what happens if there is some other method calls (directly or indirectly) inside of the constructor body and pass a reference whose class invariant it not yet established? Therefore, it may cause trouble if one always assumes that class invariants are true. Particularly, if the requirement on the invariant is an explicit part of the method signature, then the method cannot be overridden in a subclass wanting a strong invariant.

Borrowed from Fähndrich and Leino [44], we use a *raw* type. Instead of treating "raw" as a primitive, we express it (or rather its inverse "cooked") using permissions. A "cooked" pointer is one for which the invariant is true for every dynamic type that the object possesses:

$$\mathit{cooked}(r_{\mathtt{this}}) = (r_{\mathtt{this}} \in C_1)?(\exists \overline{r_{\mathtt{g}}}\,.\,(C_1(r_{\mathtt{this}}, \overline{r_{\mathtt{g}}}))) : ((r_{\mathtt{this}} \in C_2)?(\exists \overline{r_{\mathtt{g}}}\,.\,(C_2(r_{\mathtt{this}}, \overline{r_{\mathtt{g}}}))) : (\ldots))$$

where $r_{\mathtt{this}} \in C_i$ indicates that object $r_{\mathtt{this}}$ is of class C_i or one of its subclasses and $C_i(r_{\mathtt{this}}, \overline{r_{\mathtt{g}}})$ is the class invariant for class C_i with referring the self variable $r_{\mathtt{this}}$ and a sequence of guard variables $\overline{r_{\mathtt{g}}}$. The body of the *cooked* predicate ranges over all classes in the system that have at most this amount of formal guard parameters. Thus, a method can require its receiver to be "*cooked*" and permit an overriding method to use the same predicate to provide a stronger invariant. A reference with a $raw[C]$ annotation only guarantees those from C up the class hierarchy. Therefore,

$$r \in C, \mathit{cooked}(r) \Longrightarrow C(r, \overline{r_{\mathtt{g}}}) \text{ for some guard variables } \overline{r_{\mathtt{g}}}$$

Once the invariant is established after a constructor call, it can be broken by carving a field out of some data group and assigning it a value that does not consistent with the requirement of the invariant. But then the data group permission cannot be restored until the required unary field invariant is restored.

4.2.4 Method Annotations

Method annotations indicate the requirements and effects of the method body. They include not only the annotations for parameters and the receiver, but also the effects and lock usage that take place in the body. All the method annotations will be translated into their permission representation. From the caller side, it must grant these permissions, while on the callee side, it assumes these permissions are present and returns some of them back. Roughly, there are three kinds of method annotations in our system.

Effects

The typical method effects are *read* and *write*, and their target should be a sequence of *e.f* . It's useless to mention variables are read or written since they are always accessible but never updatable [2].

A method is passed permissions to enable it to access state. For example, if a method has an annotation "`writes` (e.*f*)", then there may be a write access to the *e.f* in the method body, thus a unit permission for the *e.f* must be granted to enable that possible write.

```
writes this.x                     | reads this.x
void setX( int newX ) {           | int getX() {
  x = newX;                       |   x;
}                                 | }
```

Figure 4.9: Code segment: effects.

[2]We don't have any expression to assign variables.

There are code examples for the usage of effect annotations in Figure 4.9. The `setX` method needs a $r_{\texttt{this}}.\texttt{x} \rightarrow \texttt{int}$ to update the $r_{\texttt{this}}.\texttt{x}$ field, while the `getX` method only needs a fractional permission $zr_{\texttt{this}}.\texttt{x} \rightarrow \texttt{int}$ where the z is just a fraction variable which is only required to be bigger than zero.

Lock Usage

Besides effects, methods are required to provide the lock annotations. There are two different lock annotations "`requires`" and "`uses`", where the former (also used by Boyapati and Flanagan et al. [9, 13]) shows that the lock should already be held before the method call while the latter indicates that the method body will try to acquire the lock by itself. The targets of "`requires`" and "`uses`" are restricted to: (1) the self object (`this`) or its field (`this.`*f*), (2) formal guard objects (`g`) from the class definition.

How does the "`requires` (*lock*)" affect the permission system? At first, we know this annotation indicates that the method call should be inside of a synchronized block holding the *lock*. Then the permission system indicates that the unit permission for the `Prot` data group of the lock object is only available inside of the synchronized block. Combining these two, the granted permission for this annotation is $r_{lock}.\texttt{Prot} \rightarrow \$0$.

From the permission's point of view, "`requires` (*lock*)" equals to "`writes` (*lock*.`Prot`)", thus it may also be considered as a write effect although the write operation to this data group can never happen. But certain write operations may need corresponding write permissions which are nested inside of the *lock*.`Prot` location. Therefore, once the $r_{lock}.\texttt{Prot} \rightarrow \$0$ is granted, all its nested permissions are available. Let's take the `deposit` method in Figure 4.10 for example, the "`requires (this)`" basically indicates that the $r_{\texttt{this}}.\texttt{Prot} \rightarrow \$0$ is available at the method entry, but what we actually need is the write permission to update the `this.balance` field. Since the `balance` field is annotated by "`guarded_by this`", its write permission is nested inside of the $r_{\texttt{this}}.\texttt{Prot}$. By the permission carving, the write permission to the `balance` field is available consequently. The detailed permission checking is given in next chapter.

```
class Account {
  int balance guarded_by this;
  ...
  void deposit( int x ) requires (this)
  { balance = balance + x; }
  ...
```

```
class LinkList {
  ...
  void insert( shared Object d ) uses (this)
  { sync this do {...} }
  ...
```

Figure 4.10: Code segment: lock usages

On the contrary, `uses` (*lock*) only shows a possible lock acquisition inside of the method body. It does nothing with method effect, but the lock ordering. As mentioned before, the lock acquirements should be in an ascending order. This requirement forces the *lock* to have a higher level than any locks that have already been held, that is, the *lock* should be at least the level of the r_{holding} that passed into this method entry. Here, the *lock* could be the same as or even a lower level than r_{holding}, in which case the target lock has already been held before the method entry and the synchronization for it will be reentrant. We use a conditional permission to represent these possibilities:

$$(r_{\text{holding}} < r_{lock})?(\emptyset) : (r_{lock}.\texttt{All} \rightarrow \$0)$$

Either the r_{lock} has a higher level than the r_{holding}, or the r_{lock} is a reentrant lock in the method body, thus the $r_{lock}.\texttt{Prot} \rightarrow \$0$ is granted at the method entry.

Moreover, the targets of "`requires`" and "`uses`" should be non-null objects, and if it is the form of $\texttt{this.}f$, we additional require to have a read permission for that field.

If the "`uses`" annotation has multiple targets, it's possible that these locks are acquired in arbitrary order. This is unsafe, since a deadlock condition may happen.

We want to control the sequence of lock acquisition and force them to be an ascending order. This design intent may be expressed as a method annotation: "*lock*<*lock*′", such that the former cannot be acquired when the latter one is held already. It is represented as a permission fact $r_{lock} < r_{lock'}$, where the r_{lock} and r_{lock} are the variables for the two lock objects respectively.

In particular, if the "`uses` (*lock*)" is attached to a method as well as "`holdingLock` < *lock*", then it's not necessary to worry whether the *lock* could be a reentrant lock. Therefore, a permission representation $r_{\text{holding}} < r_{lock}$ is enough and we don't use the complicated conditional form mentioned above. Actually, this result is achieved by a transformation:

$$(r_{\text{holding}} < r_{lock})?(\emptyset) : (r_{lock}.\texttt{All} \rightarrow \$0), r_{\text{holding}} < r_{lock} \Rrightarrow r_{\text{holding}} < r_{lock}$$

From Effects

Effect annotations indicate the required permissions to execute certain operations in the method body and they are returned when the method returns. Effects may be in terms of parameters, of the receiver. In some cases, the return value may be annotated as having acquired permissions "from" some of the method effects. When this happens, this kind of permissions will be carved out from the effect permission, and thus the method effects are not returned until the return value is no longer needed.

For example, we make a mutable top-left `Point` object available to the client in this manner in Figure 4.11. This code returns a mutable point, but the permission to mutate the point is made available only by temporarily making the objects location inaccessible (the permission to `this.Location` is rendered inactive).

$$r_{\text{ret}}.\texttt{All} \rightarrow \$0, r_{\text{ret}}.\texttt{All} \rightarrow \$0 \longrightarrow\!\!\!\!+ \, r_{\texttt{this}}.\texttt{Location} \rightarrow \$0$$

```
class Rectangle {
  Point tl in Location;
  ...
  from (this.Location)
  Point getTopLeft() writes (this.Location) {
  ......
```

Figure 4.11: Code segment: `from`.

4.2.5 Method Type

The annotations attached to method definitions can be considered as an abstract specification of the requirements and effects of the method body. Suppose a method has a *percondition* and a *postcondition*, then the method is considered as a mapping:

$$\textit{precondition} \rightarrow \textit{postcondition}$$

In our permission system, method annotations will be translated into permissions to express the pre and postcondition. We assign each method a permission type:

$$(\forall\Delta; \Pi_{\text{in}}) \xrightarrow{r_{\text{holding}}} (\exists\Delta'; \Pi_{\text{out}})$$

where Π_{in} and Π_{out} are the input and output permission respectively. All the variables used in Π_{in} as well as the r_{holding} are bounded in Δ, while Π_{out} may use some new variables in Δ'. The r_{holding} is the most recent lock variable that is held at the method entry.

The input permission Π_{in} indicates the required permissions to permit the body expression, where the output permission Π_{out} displays the retained permissions after a method invocation.

4.2.6 Example 1: Account and CombinedAccount

Figure 4.12 shows two class definitions with annotations (most of them have been mentioned already).

```
class Account {
  int balance guarded_by this;

  Account() {
    balance = 0;
  }
  int balance()
  reads (balance) {
    balance;
  }
  void deposit( int x )
  requires (this) {
    balance = balance + x;
  }
  void withdraw( int x )
  requires (this) {
    balance = balance - x;
  }
}
```

```
class CombinedAccount {
  final nonnull Account checking < lv;
  final nonnull Account savings > lv;
  Level lv;

  CombinedAccount(nonnull Account s,
                  nonnull Account c) {
    savings = s; checking = c;
  }
  void savings2checking(int x)
  uses (this.checking, this.savings) {
   synch checking do {
     synch savings do {
        savings.withdraw( x );
        checking.deposit( x ); } }
  }
  void double_savings(int x) uses (this.savings)
  holdingLock < this.savings {
    let s = savings in {
      fork (s) {synch s do s.deposit(x);};
      fork (s) {synch s do s.deposit(x);}; }
  }
  ......
```

Figure 4.12: Code: Account and CombinedAccount.

Account class has a balance field which is "guarded_by" the self object and its class invariant is:

$$\begin{aligned}
&\mathit{Account}(r_{\texttt{this}}) = \Gamma_{\texttt{balance}} \wedge r_{\texttt{this}} \in \texttt{Account}\\
&\Gamma_{\texttt{balance}} = r_{\texttt{this}}.\texttt{balance} \to \texttt{int} \prec r_{\texttt{this}}.\texttt{Prot}
\end{aligned}$$

CombinedAccount class has two fields: savings and checking, both of which are "final" and they are "nonnull" after the constructor call. As mentioned before, they are also "unique" by default.

$$\begin{aligned}
&\mathit{CombinedAccount}(r_{\texttt{this}}) = \Gamma_{\texttt{savings}} \wedge \Gamma_{\texttt{checking}} \wedge r_{\texttt{this}} \in \texttt{CombinedAccount}\\
&\Gamma_{\texttt{checking}} = \exists r\,.\,(z_1 r_{\texttt{this}}.\texttt{checking} \to r, \neg(r = \$0), z_1 r.\texttt{All} \to \$0, \mathit{Account}(r), r < r_{\texttt{this}}.\texttt{lv})\\
&\qquad\qquad \prec \$0.\texttt{Immutable}\\
&\Gamma_{\texttt{savings}} = \exists r\,.\,(z_2 r_{\texttt{this}}.\texttt{savings} \to r, \neg(r = \$0), z_2 r.\texttt{All} \to \$0, \mathit{Account}(r), r > r_{\texttt{this}}.\texttt{lv})\\
&\qquad\qquad \prec \$0.\texttt{Immutable}
\end{aligned}$$

Figure 4.13 gives the permission represented method types for balance and deposit. The balance method in Account class has a "reads" annotation, then it should be granted a fractional permission when this method starts and be returned after it ends. The additional named predicate $\mathit{Account}(r_{\texttt{this}})$ is to indicate that the class invariant is held with referring to the self object variable $r_{\texttt{this}}$ (no formal guard parameters for this class).

Methods `deposit` and `withdraw` have the same annotation "`requires (this)`" which means both of them can only be called inside of a synchronized block holding the lock `this`. They have the same method type in permission. Although the "`requires this`" explicitly indicates that any call to the `deposit` method must be in a synchronized block holding the $r_{\texttt{this}}$, but we cannot guarantee that the recent holding lock for this call is just the $r_{\texttt{this}}$, therefore we still keep using the r_{holding} variable.

balance	deposit(withdraw)
$\begin{array}{ll} \Delta; & \{r_{\texttt{this}}, r_{\text{holding}}, z\}\,; \\ \Pi_{\text{in}} & \left\{\begin{array}{l} z r_{\texttt{this}}.\texttt{balance} \to \texttt{int}, \\ \mathit{Account}(r_{\texttt{this}}) \end{array}\right. \\ \texttt{->} & r_{\text{holding}} \\ \Delta'; & \{r_{\texttt{ret}}\}\,; \\ \Pi_{\text{out}} & \left\{\begin{array}{l} z r_{\texttt{this}}.\texttt{balance} \to \texttt{int}, \\ \mathit{Account}(r_{\texttt{this}}), \\ r_{\texttt{ret}} = \texttt{int} \end{array}\right. \end{array}$	$\begin{array}{ll} \Delta; & \{r_{\texttt{this}}, r_{\text{holding}}, r_{\texttt{x}}\}\,; \\ \Pi_{\text{in}} & \left\{\begin{array}{l} r_{\texttt{this}}.\texttt{Prot} \to \$0, \\ \mathit{Account}(r_{\texttt{this}}), \\ r_{\texttt{x}} = \texttt{int} \end{array}\right. \\ \texttt{->} & r_{\text{holding}} \\ \Delta'; & \{r_{\texttt{ret}}\}\,; \\ \Pi_{\text{out}} & \left\{\begin{array}{l} r_{\texttt{this}}.\texttt{Prot} \to \$0, \\ \mathit{Account}(r_{\texttt{this}}), \\ r_{\texttt{ret}} = \$0 \end{array}\right. \end{array}$

Figure 4.13: Method type for class `Account`: `balance` and `deposit(withdraw)`.

Method `savings2checking` in the class `CombinedAccount` has some different method annotations. Its permission type is in Figure 4.14. As mentioned before, every method is implicitly passed permission for a small fraction of the $0.`Immutable` (which is intentionally omitted in other methods to save space). In order to get the permissions for the "`uses` " annotation using two "`final`" fields, we have to carve out the corresponding permissions for the two fields from the $0.`Immutable` and unpack their existential forms to make the $r_{\texttt{checking}}$ and $r_{\texttt{savings}}$ explicit.

$$
\begin{array}{ll}
\Delta; & \{r_{\texttt{this}}, r_{\texttt{x}}, r_{\text{holding}}, z_1, z_2\}\,; \\
\Pi_{\text{in}} & \left\{\begin{array}{l}
z_1 r_{\texttt{this}}.\texttt{checking} \rightarrow r_{\texttt{checking}}, \neg(r_{\texttt{checking}} = \$0), z_1 r_{\texttt{checking}}.\texttt{All} \rightarrow \$0, \\
\mathit{Account}(r_{\texttt{checking}}), r_{\texttt{checking}} < r_{\texttt{this}}.\texttt{lv}, \\
z_2 r_{\texttt{this}}.\texttt{savings} \rightarrow r_{\texttt{savings}}, \neg(r_{\texttt{savings}} = \$0), z_2 r_{\texttt{savings}}.\texttt{All} \rightarrow \$0, \\
\mathit{Account}(r_{\texttt{savings}}), r_{\texttt{savings}} > r_{\texttt{this}}.\texttt{lv}, \\
(\exists r\,.\,(z_1 r_{\texttt{this}}.\texttt{checking} \rightarrow r, \neg(r = \$0), z_1 r.\texttt{All} \rightarrow \$0, \mathit{Account}(r), r < r_{\texttt{this}}.\texttt{lv}), \\
\exists r\,.\,(z_2 r_{\texttt{this}}.\texttt{savings} \rightarrow r, \neg(r = \$0), z_2 r.\texttt{All} \rightarrow \$0, \mathit{Account}(r), r > r_{\texttt{this}}.\texttt{lv})) \\
\multimap \$0.\texttt{Immutable} \rightarrow \$0, \\
(r_{\text{holding}} < r_{\texttt{checking}})?(\emptyset) : (r_{\texttt{checking}}.\texttt{Prot} \rightarrow \$0), \\
(r_{\text{holding}} < r_{\texttt{savings}})?(\emptyset) : (r_{\texttt{savings}}.\texttt{Prot} \rightarrow \$0), \\
\mathit{CombinedAccount}(r_{\texttt{this}}), \\
r_{\texttt{x}} = \texttt{int}
\end{array}\right. \\
\texttt{->} & r_{\text{holding}} \\
\Delta'; & \{r_{\texttt{ret}}\}\,; \\
\Pi_{\text{out}} & \left\{\begin{array}{l}
z_1 r_{\texttt{this}}.\texttt{checking} \rightarrow r_{\texttt{checking}}, \neg(r_{\texttt{checking}} = \$0), z_1 r_{\texttt{checking}}.\texttt{All} \rightarrow \$0, \\
\mathit{Account}(r_{\texttt{checking}}), r_{\texttt{checking}} < r_{\texttt{this}}.\texttt{lv}, \\
z_2 r_{\texttt{this}}.\texttt{savings} \rightarrow r_{\texttt{savings}}, \neg(r_{\texttt{savings}} = \$0), z_2 r_{\texttt{savings}}.\texttt{All} \rightarrow \$0, \\
\mathit{Account}(r_{\texttt{savings}}), r_{\texttt{savings}} > r_{\texttt{this}}.\texttt{lv}, \\
(\exists r\,.\,(z_1 r_{\texttt{this}}.\texttt{checking} \rightarrow r, \neg(r = \$0), z_1 r.\texttt{All} \rightarrow \$0, \mathit{Account}(r), r < r_{\texttt{this}}.\texttt{lv}), \\
\exists r\,.\,(z_2 r_{\texttt{this}}.\texttt{savings} \rightarrow r, \neg(r = \$0), z_2 r.\texttt{All} \rightarrow \$0, \mathit{Account}(r), r > r_{\texttt{this}}.\texttt{lv})) \\
\multimap \$0.\texttt{Immutable} \rightarrow \$0, \\
(r_{\text{holding}} < r_{\texttt{checking}})?(\emptyset) : (r_{\texttt{checking}}.\texttt{Prot} \rightarrow \$0), \\
(r_{\text{holding}} < r_{\texttt{savings}})?(\emptyset) : (r_{\texttt{savings}}.\texttt{Prot} \rightarrow \$0), \\
\mathit{CombinedAccount}(r_{\texttt{this}}), \\
r_{\text{ret}} = \$0
\end{array}\right.
\end{array}
$$

Figure 4.14: Method type for class `CombinedAccount`: `savings2checking`.

4.2.7 Example 2: LinkList

Figure 4.15 gives the class definition for class `LinkList` and `Node`. The invariants for these two classes have already been given in previous sections. Here, we instead enumerate some permission types for their methods.

Figure 4.16 shows a permission type for the `setNext` method. The $r_{\texttt{n}}$ represents

```
class Node<g> {
  Node<g> next guarded_by g;
  shared Object datum;

  void setNext( Node<g> n )
  writes (this.next)
  { next = n; }

  from g Node<g> getNext()
  requires g
  { next; }
  ......
```

```
class LinkList {
  Node<this> head guarded_by this;

  void insert( shared Object d )
  uses (this)
  { synch this do {
      let newNode = new Node<this>(d) in {
      newNode.setNext( head );
      head = newNode; } }
  }
  ......
```

Figure 4.15: Code: LinkList and Node

the variable for the formal parameter `n`. Since the parameter `n` is a unique pointer, a unit permission for its `All` data group is included if the `n` is non-null. The "`writes`" annotation is translated as a unit permission packing the corresponding for its pointed-to object using an existential form. The class invariant for the receiver is included by default. All of them will also show up in the output permission except for the permission of the unique parameter.

$$
\begin{array}{ll}
\Delta; & \{r_{\texttt{this}}, r_{\texttt{g}}, r_{\texttt{n}}, r_{\text{holding}}\}\,; \\
\Pi_{\text{in}} & \left\{\begin{array}{l} (\neg(r_{\texttt{n}} = \$0))?(\mathit{Node}(r_{\texttt{n}}, r_{\texttt{g}}), r_{\texttt{n}}.\texttt{All} \to \$0) : (\emptyset), \\ \exists r\,.\,(r_{\texttt{this}}.\texttt{next} \to r, (\neg(r = \$0))?(\mathit{Node}(r, r_{\texttt{g}}), r.\texttt{All} \to \$0) : (\emptyset)), \\ \mathit{Node}(r_{\texttt{this}}, r_{\texttt{g}}) \end{array}\right. \\
\texttt{->} & r_{\text{holding}} \\
\Delta'; & \{r_{\text{ret}}\}\,; \\
\Pi_{\text{out}} & \left\{\begin{array}{l} \exists r\,.\,(r_{\texttt{this}}.\texttt{next} \to r, (\neg(r = \$0))?(\mathit{Node}(r, r_{\texttt{g}}), r.\texttt{All} \to \$0) : (\emptyset)), \\ \mathit{Node}(r_{\texttt{this}}, r_{\texttt{g}}), \\ r_{\text{ret}} = \$0 \end{array}\right.
\end{array}
$$

Figure 4.16: Method type for class `Node`: `setNext`.

Figure 4.17 gives the permission type for method `getNext` which "`requires`" the `g` to be held before this method entry. Thus the unit permission for $r_{\texttt{g}}.\texttt{Prot}$ is granted as well as the class invariant for the self object. Furthermore, this method uses a "`from`" to annotate the returned value. This indicates the $r_{\text{ret}}.\texttt{All} \to \$0$ (if the r_{ret} is not null) is carved out from the $r_{\mathtt{g}}.\mathtt{Prot} \rightarrow \$0$.

$$\begin{array}{ll}
\Delta; & \{r_{\mathtt{this}}\}\,; \\
\Pi_{\text{in}} & \left\{\begin{array}{l} r_{\mathtt{g}}.\mathtt{Prot} \rightarrow \$0, \\ Node(r_{\mathtt{this}}, r_{\mathtt{g}}) \end{array}\right. \\
\texttt{->} & r_{\mathtt{g}} \\
\Delta'; & \{r_{\mathtt{ret}}\}\,; \\
\Pi_{\text{out}} & \left\{\begin{array}{l} (\neg(r_{\text{ret}} = \$0))?(Node(r_{\text{ret}}, r_{\mathtt{g}}), r_{\text{ret}}.\mathtt{All} \rightarrow \$0) : (\emptyset), \\ (\neg(r_{\text{ret}} = \$0))?(Node(r_{\text{ret}}, r_{\mathtt{g}}), r_{\text{ret}}.\mathtt{All} \rightarrow \$0) : (\emptyset) \multimap r_{\mathtt{g}}.\mathtt{Prot} \rightarrow \$0, \\ Node(r_{\mathtt{this}}, r_{\mathtt{g}}) \end{array}\right.
\end{array}$$

Figure 4.17: Method type for class `Node`: `getNext`.

Figure 4.18 shows a method type for the `insert` in class `LinkList`. Besides the class invariant with referring to the $r_{\mathtt{this}}$, this method additionally has a "`shared`" parameter, thus some permissions about that parameter are given as well. The "`uses (this)`" basically grants the conditional permission $(r_{\text{holding}} < r_{\mathtt{this}})?(\emptyset) : (r_{\mathtt{this}}.\mathtt{Prot} \rightarrow \$0)$ as mentioned before.

$$\begin{array}{ll}
\Delta; & \{r_{\mathtt{this}}, r_{\mathtt{d}}, r_{\text{holding}}\}\,; \\
\Pi_{\text{in}} & \left\{\begin{array}{l} (r_{\text{holding}} < r_{\mathtt{this}})?(\emptyset) : (r_{\mathtt{this}}.\mathtt{Prot} \rightarrow \$0), \\ (\neg(r_{\mathtt{d}} = \$0))?(r_{\mathtt{d}}.\mathtt{All} \rightarrow \$0 \prec r_{\mathtt{d}}.\mathtt{Prot}, Object(r_{\mathtt{d}})) : (\emptyset), \\ LinkList(r_{\mathtt{this}}) \end{array}\right. \\
\texttt{->} & r_{\text{holding}} \\
\Delta'; & \{r_{\mathtt{ret}}\}\,; \\
\Pi_{\text{out}} & \left\{\begin{array}{l} (r_{\text{holding}} < r_{\mathtt{this}})?(\emptyset) : (r_{\mathtt{this}}.\mathtt{Prot} \rightarrow \$0), \\ LinkList(r_{\mathtt{this}}), \\ r_{\mathtt{ret}} = \$0 \end{array}\right.
\end{array}$$

Figure 4.18: Method type for class `LinkList`: `insert`.

4.3 Summary

This chapter first introduce the syntax of permissions. The nesting and fractional are the most important properties. Then we show how to translate the design intent expressed by high-level annotations into their low-level permission representation. Fields may be attached some annotations which indicate the protection mechanism for itself as well as the pointed-to object. The permission representation for each field is a unary field invariant and the conjunction of all unary field invariants becomes a class invariant. Method annotations include pointer annotations for its receiver and formal parameters, class invariant for the receiver, effect and lock usage annotations. With these annotations, any method could be treated as a mapping from an input permission to an output permission. From the caller's side, the input permission of a method is the required permission to make a call while the output permission is the retained permission after the method call. From the callee's side, the input permission is the granted permission to its body, while the output permission is required to return to its caller.

Annotations to fields or groups	Permission Representation
$\texttt{C<}\overline{\texttt{g}}\texttt{>}\ f$	$\exists r\,.\,(r_{\texttt{this}}.f \rightarrow r, (\neg(r = \$0))?(C(r,\overline{r_{\texttt{g}}}), r.\texttt{All} \rightarrow \$0) : (\emptyset)) \prec r_{\texttt{this}}.\texttt{All}$
$\texttt{nonnull C<}\overline{\texttt{g}}\texttt{>}\ f$	$\exists r\,.\,(r_{\texttt{this}}.f \rightarrow r, \neg(r = \$0), C(r,\overline{r_{\texttt{g}}}), r.\texttt{All} \rightarrow \$0) \prec r_{\texttt{this}}.\texttt{All}$
$\texttt{shared C<}\overline{\texttt{g}}\texttt{>}\ f$	$\exists r\,.\,(r_{\texttt{this}}.f \rightarrow r, (\neg(r = \$0))?(C(r,\overline{r_{\texttt{g}}}), (r.\texttt{All} \rightarrow \$0 \prec r.\texttt{Prot})) : (\emptyset)) \prec r_{\texttt{this}}.\texttt{All}$
$\texttt{nonnull shared C<}\overline{\texttt{g}}\texttt{>}\ f$	$\exists r\,.\,(r_{\texttt{this}}.f \rightarrow r, \neg(r = \$0), C(r,\overline{r_{\texttt{g}}}), (r.\texttt{All} \rightarrow \$0 \prec r.\texttt{Prot})) \prec r_{\texttt{this}}.\texttt{All}$
$\texttt{C<}\overline{\texttt{g}}\texttt{>}\ f\ \ \texttt{guarded\_by}\ \textit{\textbf{guard}}$	$\exists r\,.\,(r_{\texttt{this}}.f \rightarrow r, (\neg(r = \$0))?(C(r,\overline{r_{\texttt{g}}}), r.\texttt{All} \rightarrow \$0) : (\emptyset)) \prec r_{\textit{guard}}.\texttt{Prot}$
$\texttt{nonnull C<}\overline{\texttt{g}}\texttt{>}\ f\ \ \texttt{guarded\_by}\ \textit{\textbf{guard}}$	$\exists r\,.\,(r_{\texttt{this}}.f \rightarrow r, \neg(r = \$0), C(r,\overline{r_{\texttt{g}}}), r.\texttt{All} \rightarrow \$0) \prec r_{\textit{guard}}.\texttt{Prot}$
$\texttt{shared C<}\overline{\texttt{g}}\texttt{>}\ f\ \ \texttt{guarded\_by}\ \textit{\textbf{guard}}$	$\exists r\,.\,(r_{\texttt{this}}.f \rightarrow r, (\neg(r = \$0))?(C(r,\overline{r_{\texttt{g}}}), (r.\texttt{All} \rightarrow \$0 \prec r.\texttt{Prot})) : (\emptyset)) \prec r_{\textit{guard}}.\texttt{Prot}$
$\texttt{nonnull shared C<}\overline{\texttt{g}}\texttt{>}\ f\ \ \texttt{guarded\_by}\ \textit{\textbf{guard}}$	$\exists r\,.\,(r_{\texttt{this}}.f \rightarrow r, \neg(r = \$0), C(r,\overline{r_{\texttt{g}}}), (r.\texttt{All} \rightarrow \$0 \prec r.\texttt{Prot})) \prec r_{\textit{guard}}.\texttt{Prot}$
$\texttt{C<}\overline{\texttt{g}}\texttt{>}\ f\ \ \texttt{in G}$	$\exists r\,.\,(r_{\texttt{this}}.f \rightarrow r, (\neg(r = \$0))?(C(r,\overline{r_{\texttt{g}}}), r.\texttt{All} \rightarrow \$0) : (\emptyset)) \prec r_{\texttt{this}}.\texttt{G}$
$\texttt{nonnull C<}\overline{\texttt{g}}\texttt{>}\ f\ \ \texttt{in G}$	$\exists r\,.\,(r_{\texttt{this}}.f \rightarrow r, \neg(r = \$0), C(r,\overline{r_{\texttt{g}}}), r.\texttt{All} \rightarrow \$0) \prec r_{\texttt{this}}.\texttt{G}$
$\texttt{shared C<}\overline{\texttt{g}}\texttt{>}\ f\ \ \texttt{in G}$	$\exists r\,.\,(r_{\texttt{this}}.f \rightarrow r, (\neg(r = \$0))?(C(r,\overline{r_{\texttt{g}}}), (r.\texttt{All} \rightarrow \$0 \prec r.\texttt{Prot})) : (\emptyset)) \prec r_{\texttt{this}}.\texttt{G}$
$\texttt{nonnull shared C<}\overline{\texttt{g}}\texttt{>}\ f\ \ \texttt{in G}$	$\exists r\,.\,(r_{\texttt{this}}.f \rightarrow r, \neg(r = \$0), C(r,\overline{r_{\texttt{g}}}), (r.\texttt{All} \rightarrow \$0 \prec r.\texttt{Prot})) \prec r_{\texttt{this}}.\texttt{G}$
$\texttt{group G}$	$r_{\texttt{this}}.\texttt{G} \rightarrow \$0 \prec r_{\texttt{this}}.\texttt{All}$
$\texttt{group G guarded\_by}\ \textit{\textbf{guard}}$	$r_{\texttt{this}}.\texttt{G} \rightarrow \$0 \prec r_{\textit{guard}}.\texttt{Prot}$
$\texttt{group G in G'}$	$r_{\texttt{this}}.\texttt{G} \rightarrow \$0 \prec r_{\texttt{this}}.\texttt{G'}$
$\texttt{int}\ f$	$r_{\texttt{this}}.f \rightarrow \texttt{int} \prec r_{\texttt{this}}.\texttt{All}$
$\texttt{int}\ f\ \ \texttt{guarded\_by}\ \textit{\textbf{guard}}$	$r_{\texttt{this}}.f \rightarrow \texttt{int} \prec r_{\texttt{this}}.\texttt{Prot}$
$\texttt{int}\ f\ \ \texttt{in G}$	$r_{\texttt{this}}.f \rightarrow \texttt{int} \prec r_{\texttt{this}}.\texttt{G}$

Table 4.1: Annotations' permission representation.

Annotations to fields or groups	Permission Representation
$\texttt{final C<}\overline{\texttt{g}}\texttt{>}\ f$	$\exists r\,.\,(zr_{\texttt{this}}.f\ \rightarrow r, z((\neg(r = \$0))?(C(r,\overline{r_{\texttt{g}}}), r.\texttt{All} \rightarrow \$0) : (\emptyset))) \prec \$0.\texttt{Immutable}$
$\texttt{nonnull final C<}\overline{\texttt{g}}\texttt{>}\ f$	$\exists r\,.\,(zr_{\texttt{this}}.f\ \rightarrow r, \neg(r = \$0), C(r,\overline{r_{\texttt{g}}}), zr.\texttt{All} \rightarrow \$0) \prec \$0.\texttt{Immutable}$
$\texttt{shared final C<}\overline{\texttt{g}}\texttt{>}\ f$	$\exists r\,.\,(zr_{\texttt{this}}.f\ \rightarrow r, z((\neg(r = \$0))?(C(r,\overline{r_{\texttt{g}}}), r.\texttt{All} \rightarrow \$0 \prec r.\texttt{Prot}) : (\emptyset))) \prec \$0.\texttt{Immutable}$
$\texttt{nonnull shared final C<}\overline{\texttt{g}}\texttt{>}\ f$	$\exists r\,.\,(zr_{\texttt{this}}.f\ \rightarrow r, \neg(r = \$0), C(r,\overline{r_{\texttt{g}}}), r.\texttt{All} \rightarrow \$0 \prec r.\texttt{Prot}) \prec \$0.\texttt{Immutable}$
$\texttt{nonnull final C<}\overline{\texttt{g}}\texttt{>}\ f\ \ \texttt{< (>)}\ \ lv$	$\exists r\,.\,(zr_{\texttt{this}}.f\ \rightarrow r, \neg(r = \$0), C(r,\overline{r_{\texttt{g}}}), zr.\texttt{All} \rightarrow \$0, r < (>) r_{\texttt{this}}.lv) \prec \$0.\texttt{Immutable}$
$\texttt{nonnull shared final C<}\overline{\texttt{g}}\texttt{>}\ f$	$\exists r\,.\,(zr_{\texttt{this}}.f\ \rightarrow r, \neg(r = \$0), C(r,\overline{r_{\texttt{g}}}), r.\texttt{All} \rightarrow \$0 \prec r.\texttt{Prot}, r < (>) r_{\texttt{this}}.lv) \prec \$0.\texttt{Immutable}$
$\texttt{final int}\ f$	$zr_{\texttt{this}}.f\ \rightarrow \texttt{int} \prec \$0.\texttt{Immutable}$

Table 4.2: Annotations' permission representation (Cont.).

Chapter 5

Permission Type Checking

From the previous chapter, method annotations as well as field (pointer) annotations can be translated into a method type in permission $(\forall\Delta; \Pi_{\text{in}}) \xrightarrow{r_{\text{holding}}} (\exists\Delta'; \Pi_{\text{out}})$. The Π_{in} acts as the granted permissions to the method body, while Π_{out} is the result permissions when the method ends.

Given an environment E and an expression e nested most recently in a synchronized block holding lock ρ_L, if the e is not a parallel composition and can be permission checked using E, then it has a permission type τ with the environment being changed to E'. This process is given as a judgement:

$$E \vdash_{\rho_L} e \Downarrow \tau \dashv E'$$

An environment is composed by two parts: (1) a type context Δ which is a set of object reference variables r and fraction variables z; (2) the granted permission Π. For a well-formed environment $E = (\Delta; \Pi)$, we require that all free variables used in Π are in Δ $(FV(\Pi) \subseteq \Delta)$. For brevity purposes, this restriction is left implicit.

5.1 Permission Typing Rules

5.1.1 Conditional

For a conditional expression, we first check its condition part and get two permission type $\texttt{ptr}(\rho_1)$ and $\texttt{ptr}(\rho_2)$ for two expression e_1 and e_2 respectively. Then we separately permission check different branches with additional facts indicating whether or not $\rho_1 = \rho_2$. The two different output permission Π_3 and Π_4 together with two different permission types respectively will be combined using a conditional permission form.

$$\textsc{If}\quad \frac{\begin{array}{c}\Delta;\Pi \vdash_{\rho_L} e_1 \Downarrow \mathtt{ptr}(\rho_1) \dashv \Delta_1;\Pi_1 \vdash_{\rho_L} e_2 \Downarrow \mathtt{ptr}(\rho_2) \dashv \Delta_2;\Pi_2 \\ \Delta_2;\rho_1=\rho_2,\Pi_2 \vdash_{\rho_L} e_3 \Downarrow \mathtt{ptr}(\rho_3) \dashv \Delta_3;\Pi_3 \\ \Delta_2;\neg(\rho_1=\rho_2),\Pi_2 \vdash_{\rho_L} e_4 \Downarrow \mathtt{ptr}(\rho_4) \dashv \Delta_4;\Pi_4 \\ r \text{ fresh} \qquad \Pi' = (\rho_1=\rho_2)?(r=\rho_3,\Pi_3):(r=\rho_4,\Pi_4)\end{array}}{\Delta;\Pi \vdash_{\rho_L} \mathtt{if}\ e_1\mathtt{==}e_2\ \mathtt{then}\ e_3\ \mathtt{else}\ e_4 \Downarrow \mathtt{ptr}(r) \dashv \Delta_3 \cup \Delta_4;\Pi'}$$

Figure 5.1: Permission type rules: Conditional

5.1.2 Local

$$\textsc{Local}\quad \frac{\begin{array}{c}\Delta;\Pi \vdash_{\rho_L} e_1 \Downarrow \mathtt{ptr}(\rho_1) \dashv \Delta';\Pi' \\ r_x \notin \Delta' \qquad \{r_x\}\cup\Delta';\rho_1=r_x,\Pi' \vdash_{\rho_L} e_2 \Downarrow \mathtt{ptr}(\rho_2) \dashv \Delta'';\Pi''\end{array}}{\Delta;\Pi \vdash_{\rho_L} \mathtt{let}\ x\mathtt{=}e_1\ \mathtt{in}\ e_2 \Downarrow \mathtt{ptr}([r_x \mapsto \rho_1]\rho_2) \dashv \Delta'' \setminus \{r_x\}\,;[r_x \mapsto \rho_1]\Pi''}$$

Figure 5.2: Permission type rules: Local

Local variable declaration will introduce a fresh variable r_x in permissions for the body e_2 such that a fact $\rho_1 = r_x$ is added indicating the r_x is an alias for the object that pointed to by e_1. Since the scope of the local variable is only the body e_2, we need to substitute it back after permission checking the e_2.

5.1.3 Read and Write

$$\textsc{Read}\quad \frac{\Delta;\Pi \vdash_{\rho_L} e \Downarrow \mathtt{ptr}(\rho) \dashv \Delta';\Pi' \qquad \Pi' = \xi\rho.f \rightarrow \rho_f,\Pi''}{\Delta;\Pi \vdash_{\rho_L} e.f \Downarrow \mathtt{ptr}(\rho_f) \dashv \Delta';\Pi'}$$

$$\textsc{Write}\quad \frac{\Delta;\Pi \vdash_{\rho_L} e_1 \Downarrow \mathtt{ptr}(\rho_1) \dashv \Delta';\Pi' \vdash_{\rho_L} e_2 \Downarrow \mathtt{ptr}(\rho_2) \dashv \Delta'';\Pi'' \qquad \Pi'' = \rho_1.f \rightarrow \rho_f,\Pi'''}{\Delta;\Pi \vdash_{\rho_L} e_1.f\mathtt{=}e_2 \Downarrow \mathtt{ptr}(\rho_2) \dashv \Delta'';\rho_1.f \rightarrow \rho_2,\Pi'''}$$

Figure 5.3: Permission type rules: Read and Write

For any read access to a field $e.f$, we first permission check the expression e to get a reference to the ρ object as well as an output permission Π'. Then we require that the output permission Π' to include a fractional permission to that field expressed by $\xi\rho.f \rightarrow \rho_f$ for some ρ_f. This also works to avoid the null pointer exception: if the ρ is \$0, then there is no way for the Π' to include a fractional permission for $\rho.f$.

In order to update a field, we must have a write permission granted to that field. That is the reason we require the Π'', which is the output permission after checking the e_1 and e_2 in turn, to include a unit permission for that field access. It is expressed by a requirement above the line: $\Pi'' = \rho_1.f \rightarrow \rho_f, \Pi'''$ in the rule Write. Furthermore, the unit permission is updated to have a different pointed-to object (ρ_2) after this assignment. The permission type for the whole field assignment will be the same one as its right hand side. Similar as before, the null pointer exception for the field reference $e_1.f$ is avoidable.

5.1.4 New

New

$$\frac{r \text{ fresh} \qquad \forall i \in [1..n].f_i \in \text{fields}(C) \setminus \{\texttt{Prot}\}}{\Delta; \Pi \vdash_{\rho_L} \texttt{new}\ C \Downarrow \texttt{ptr}(r) \dashv \{r\} \cup \Delta; \neg(r = \$0), r \in C, r.f_i \rightarrow \$0, \Pi}$$

Figure 5.4: Permission type rules: New

We assume there is unlimited memory that can be used to allocate objects. For a pure allocation expression, we use the rule New. First, a fresh variable r is picked to represent the reference of the new allocated object. Then all its fields and data groups (except the Prot) are to be initialized as null pointers with presenting unit permissions for them. At last, a type assertion $r \in C$ and a non-null property for r are attached to the output permission as well. The reason not to include a unit permission for the `Prot` data group is that it is not granted by an allocation, but a synchronization.

5.1.5 Synchronization and Hold

Reentrant

$$\frac{\Delta;\Pi \vdash_{\rho_L} e_1 \Downarrow \texttt{ptr}(\rho_1) \dashv \Delta';\Pi' \qquad \Pi' = \rho_1.\texttt{Prot} \rightarrow \$0, \Pi'_1 \qquad \Delta';\Pi' \vdash_{\rho_L} e_2 \Downarrow \texttt{ptr}(\rho_2) \dashv \Delta'';\Pi''}{\Delta;\Pi \vdash_{\rho_L} \texttt{synch}\ e_1\ \texttt{do}\ e_2 \Downarrow \texttt{ptr}(\rho_2) \dashv \Delta'';\Pi''}$$

Synch

$$\frac{\Delta;\Pi \vdash_{\rho_L} e_1 \Downarrow \texttt{ptr}(\rho_1) \dashv \Delta';\Pi' \qquad \Pi' = \rho_L < \rho_1, \Pi'_2 \qquad \Delta';\rho_1.\texttt{Prot} \rightarrow \$0, \Pi' \vdash_{\rho_1} e_2 \Downarrow \texttt{ptr}(\rho_2) \dashv \Delta'';\rho_1.\texttt{Prot} \rightarrow \$0, \Pi''}{\Delta;\Pi \vdash_{\rho_L} \texttt{synch}\ e_1\ \texttt{do}\ e_2 \Downarrow \texttt{ptr}(\rho_2) \dashv \Delta'';\Pi''}$$

Hold

$$\frac{\Delta;\Pi \vdash_{o_1} e_2 \Downarrow \texttt{ptr}(\rho_2) \dashv \Delta'';\Pi'}{\Delta;\Pi \vdash_{o_1} \texttt{hold}\ o_1\ \texttt{do}\ e_2 \Downarrow \texttt{ptr}(\rho_2) \dashv \Delta'';\Pi'}$$

Figure 5.5: Permission type rules: Synchronization

For a synchronization expression, we need to permission check the lock expression e_1 to get an object reference ρ_1 first. Then there are two possibilities:

- This lock has already been held expressed by $\Pi' = \rho_1.\texttt{Prot} \rightarrow \$0, \Pi'_1$, then this is a reentrant lock using rule Reentrant. In this case, the lock acquirement is not needed. We directly permission check the synchronized block using the output permissions from checking the lock expression and the most recent holding lock is kept.

- If the above case does not apply, then we need to make sure that a proper order between the current holding lock and the acquiring lock ($\rho_L < \rho_1$) exists, since it's forced to acquire locks in an ascending order to avoid deadlocks. If it does, then the rule SYNCH applies. We additionally attach a unit permission for the `Prot` data group of the acquired lock to check the synchronized block as well as updating its surrounding lock.

For a non-reentrant lock acquirement, the rule SYNCH guarantees that: the permission $\rho_1.\texttt{Prot} \rightarrow \$0$ as well as the permissions protected by the lock (they are nested in $\rho_1.\texttt{Prot}$) is not available unless the ρ_1 is held. The hold expression is an internal expression and the rule HOLD only ensures that the surrounding lock for a hold expression is the same object explicitly presented in the hold expression.

5.1.6 Fork and Sequence

$$\textsc{Fork}\quad \frac{\overline{x \in \Delta} \qquad \begin{array}{c} \Pi = \Pi_1, \Pi_2 \qquad r_{\text{newThread}} \text{ fresh} \\ \Delta; \Pi_2, r_{\text{newThread}}.\texttt{Prot} \rightarrow \$0 \vdash_{r_{\text{newThread}}} e \Downarrow \texttt{ptr}(\rho) \dashv \Delta'; \Pi_2' \end{array}}{\Delta; \Pi \vdash_{\rho_L} \texttt{fork}\ (\overline{x})\, e \Downarrow \texttt{ptr}(\$0) \dashv \Delta \cup \Delta'; \Pi_1}$$

$$\textsc{Seq}\quad \frac{\Delta; \Pi \vdash_{\rho_L} e_1 \Downarrow \texttt{ptr}(\rho_1) \dashv \Delta'; \Pi' \vdash_{\rho_L} e_2 \Downarrow \texttt{ptr}(\rho_2) \dashv \Delta''; \Pi''}{\Delta; \Pi \vdash_{\rho_L} e_1 ; e_2 \Downarrow \texttt{ptr}(\rho_2) \dashv \Delta''; \Pi''}$$

Figure 5.6: Permission type rules: Fork and Seq

In order to permission check a spawn expression $\texttt{fork}\ (\overline{x})\, e$, we need to split the input permission and flow away some part (Π_2) to check the expression e in a new spawn thread as well as remaining the rest (Π_1) in the current thread. In addition, since the e will be evaluated in a new thread, its surrounding lock will implicitly be the new thread object. It is similar as treating the e as a synchronized block with holding the new thread object acting as a lock. Any permission that is thread local will be available by carving it out from the unit permission for the `Prot` data group of the thread object. We use a fresh variable $r_{\text{newThread}}$ to represent the new spawn thread and make the $r_{\text{newThread}}.\texttt{Prot} \rightarrow \$0$ available at the beginning.

The rule Seq applies to a sequential expression $e_1\texttt{;}e_2$. The two expressions are permission checked in turn, such that the output permission after checking the former becomes the input permission for the latter. The permission type for the e_1 is dropped, while the permission type for the e_2 also works for the whole sequential expression.

5.1.7 Call and Dispatch

Dispatch

$$\frac{\begin{array}{c}\Delta;\Pi \vdash_{\rho_L} e_0 \Downarrow \texttt{ptr}(\rho_0) \dashv \Delta_0;\Pi_0 \\ \Pi_0 = \rho_0 \in C, \Pi_0' \qquad \Delta;\Pi \vdash_{\rho_L} e_0.C\#m(e_1,...,e_n) \Downarrow \texttt{ptr}(r) \dashv \Delta';\Pi'\end{array}}{\Delta;\Pi \vdash_{\rho_L} e_0.m(e_1,...,e_n) \Downarrow \texttt{ptr}(r) \dashv \Delta';\Pi'}$$

Call

$$\frac{\begin{array}{c}\Delta;\Pi \vdash_{\rho_L} e_0 \Downarrow \texttt{ptr}(\rho_0) \dashv E_0 \vdash_{\rho_L} e_1 \Downarrow \texttt{ptr}(\rho_1) \dashv \ldots \vdash_{\rho_L} e_n \Downarrow \texttt{ptr}(\rho_n) \dashv E_n \\ E_0 = \Delta_0;\rho_0 \in C, \Pi_0 \qquad E_n = \Delta_n;\Pi_n \\ \text{mbody}(C,m) = (x^*, e, \forall\Delta_0';\Pi_0' \xrightarrow{r_{\text{holding}}} \exists\Delta_0'';\Pi'') \qquad \sigma_1 : \Delta_0' \rightarrow \Delta_n \qquad \Delta' \text{ fresh} \\ \sigma_2 : \Delta' \rightarrow \Delta_0'' \qquad \Pi'' = \sigma_2\Pi_0'' \qquad \Pi_n = \sigma_1\Pi_0', \Pi' \qquad \forall i \in [1..n].\sigma_1(r_{x_i}) = \rho_i \\ \sigma_1(r_{\texttt{this}}) = \rho_0 \qquad \sigma_1(r_{\text{holding}}) = \rho_L \qquad r' \in \Delta' \qquad \sigma_2(r') = r_{\texttt{ret}}\end{array}}{\Delta;\Pi \vdash_{\rho_L} e_0.C\#m(e_1,...,e_n) \Downarrow \texttt{ptr}(r') \dashv \sigma_1\Pi_0'', \Pi'}$$

Figure 5.7: Permission type rules: Call and Dispatch

Rule Dispatch delegates to a static Call. When a method invocation happens, what is the exact dynamic type for its receiver object? What we can do in permission checking is to find out its static type. For here, we will do the best we can. The type rule will pick some type that the static system knows for the receiver expressed as $\Pi_0 = \rho_0 \in C, \Pi_0'$. It's possible that $\Pi_0' = \rho_0 \in C', \Pi_0''$ which means that the receiver ρ_0 has a polymorphic type since our system allows subtyping. For most precision, one would pick the "best" static type, but safety requires only a possible type. The rule for methods checks overriding to ensure that picking a less precise type does not subvert the permission type system.

In the rule CALL, we first permission check the receiver expression and each actual parameter and get the output permissions Π_n. It's required that Π_n includes a fact of type assertion for the receiver object ρ_0. Then we fetch the procedure type $(\Delta_0'; \Pi_0') \xrightarrow{r_{\text{holding}}} (\Delta_0''; \Pi'')$ according to the method name and receiver's type. The output permission bag Π'' will use the (existentially quantified) variables in Δ_0'' as well as the (universally quantified) variables in Δ_0'. We use the fresh Δ' to act as the first set. The substitution goes in the reverse direction than one might think, and thus Π'' is represented as $\sigma_2\Pi_0''$ where Π_0'' uses the fresh variables as well as the universally quantified variables from Δ_0'. Once all actual parameters are checked, we then form σ_1 to substitute these remaining variables and split the Π_n into two parts: one which matches the substituted input permissions $\sigma_1\Pi_0'$ and one which contains the remaining permissions Π'. The latter are combined with the substituted output permissions $\sigma_1\Pi_0''$ to create the resulting environment.

5.1.8 Others

VARIABLE
$$\frac{r_x \in \Delta}{\Delta; \Pi \vdash_{\rho_L} x \Downarrow \texttt{ptr}(r_x) \dashv \Delta; \Pi}$$

OBJLOC
$$\Delta; \Pi \vdash_{\rho_L} o \Downarrow \texttt{ptr}(o) \dashv \Delta; \Pi$$

SKIP
$$\Delta; \Pi \vdash_{\rho_L} \texttt{skip} \Downarrow \texttt{ptr}(\$0) \dashv \Delta; \Pi$$

ARITHMETIC
$$\frac{\Delta; \Pi \vdash_{\rho_L} e_1 \Downarrow \texttt{ptr(int)} \dashv \Delta'; \Pi' \vdash_{\rho_L} e_2 \Downarrow \texttt{ptr(int)} \dashv \Delta''; \Pi''}{\Delta; \Pi \vdash_{\rho_L} e_1 \text{ op } e_2 \Downarrow \texttt{ptr(int)} \dashv \Delta''; \Pi''}$$

Figure 5.8: Permission type rules: others

Input and output permissions are the same when permission checking a variable, a literal object or a skip expression using rule VARIABLE, OBJLOC and SKIP respectively. The arithmetic expression only accepts the primitive `int` type subexpressions. For simplicity, we still use a pointer type `ptr(int)` for any `int` expression. That is, any `int` type expression can be considered as a reference that pointing to some uninteresting integer object.

5.1.9 Method

METHOD

$$\frac{\begin{array}{c}\Delta_1; r_{\texttt{this}} \in C, \Pi_1 \vdash_{r_{\text{holding}}} e \Downarrow \texttt{ptr}(\rho) \dashv \Delta'; \sigma_2\Pi_2 \qquad \{r_{x_1}, ..., r_{x_n}, r_{\texttt{this}}, r_{\text{holding}}\} \subseteq \Delta_1 \\ \Delta' \cap \Delta_2 = \emptyset \qquad \sigma_2 : \Delta_2 \to \Delta' \qquad r_{\texttt{ret}} \in \Delta_2 \qquad \sigma_2(r_{\texttt{ret}}) = \rho \\ \forall_{C \preccurlyeq C'} \text{mbody}(C', m) = (\overline{x'}, e', \forall\Delta'_1; \Pi'_1 \xrightarrow{r_{\text{holding}}} \exists\Delta'_2; \Pi'_2) \Rightarrow (|\overline{x}| = |\overline{x'}|) \wedge (\Pi'_1 \leadsto \Pi_1) \wedge (\Pi_2 \leadsto \Pi'_2)\end{array}}{\vdash \forall\Delta_1; \Pi_1 \xrightarrow{r_{\text{holding}}} \exists\Delta_2; \Pi_2 \text{ is the type for } mn(\overline{x})\{e\} \text{ of the class } C}$$

Figure 5.9: Permission type rules: method

METHOD permission checks the body of a method using its input permission from its method type. At the end of the method body, the output permission must match a substitution of the output environment. In addition, if the method overrides another method, it should satisfy the standard covariant/contravariant condition, specified using environment transformation. The constructor is a special form of method which calls its superclass's constructor at the beginning and returns the receiver object at the end. Some permission nesting transformations may happen implicitly inside the constructor (see section 5.2 for details).

5.1.10 Parallel Composition

To permission check an internal parallel composition expression $e_1||...||e_n$, such that e_i is nested most recently in a synchronized block holding lock ρ_{L_i}, the input permission Π should be able to be partitioned into n parts, and each of which is the granted permission to permission check one of the parallel branches. The resulting permission type for the whole parallel composition is uninteresting. Here, we use a little different permission judgement, since this parallel composition happens involving different parallel threads at the same time. Actually, there are n permission checking happening within n separate parallel threads and the whole parallel composition cannot be satisfied by the granted permission Π unless all its branches are satisfied by their granted permissions respectively.

$$\textsc{Par}\quad \frac{\Pi = \Pi_1, ..., \Pi_n \qquad \Delta; \Pi_i \vdash_{\rho_{L_i}} e_i \Downarrow \texttt{ptr}(\rho_i) \dashv \Delta'_i; \Pi'_i}{\Delta; \Pi \vdash_{(\rho_{L_1}||...||\rho_{L_n})} e_1||...||e_n \Downarrow \texttt{ptr}(\$0) \dashv \Delta'_1 \cup ... \cup \Delta'_n; \Pi'_1, ..., \Pi'_n}$$

Figure 5.10: Permission type rules: parallel composition

5.2 Permission Transformation

We define permission transformation so that type system can convert permissions from one form, to a more convenient form [8]. In particular:

- one may reorder permissions;
- one may expand or extract a named predicate;
- one may split or merge fractions of the same key;

- one may generate or apply a conditional permission;
- one may open or close an existential type;
- one may carve or replace a nested permission;
- one may perform nesting (but not undo it!);
- one may deduce a new order between objects.

We write $E \rightsquigarrow E'$ to indicate that one environment E can be transformed into another environment E' [1]. We permit two different kinds of transformations: one that is logical implication (written as $\Rrightarrow$) and one that merely permits nesting:

$$\textsc{Tr-Implies}\quad \frac{E \Rrightarrow E'}{E \rightsquigarrow E'} \qquad\qquad \textsc{Tr-Nest}\quad \Delta;\Pi_1,\Pi_2 \rightsquigarrow \Delta;\Pi_1,(\Pi_2 \prec k)$$

The Tr-Nest says that one may give up any permission and receive in its place a "nesting fact". The place of nesting is arbitrary. Adding rules to prevent cyclic nesting is not feasible since nesting can be easily concealed.

Logical implication is defined at the level of permission semantics:

$$\frac{\forall_{\sigma,h\leq\mu,A}(h \models_A \sigma E) \Rightarrow \exists_{\sigma'\supseteq\sigma,h'\leq h}(h' \models_A \sigma' E')}{E \Rrightarrow E'}$$

Here, we borrow some concepts from the next chapter. The $h \models_A \sigma E$ indicates that a fractional heap h can be used to model the E with an assumption A and a substitution σ which will substitute away the variable set in E. The $\sigma' \supseteq \sigma$ means that σ' is the same as σ on the domain of σ. We do not permit the assumption set A to increase, but we do permit the heap to get smaller. In essence, one can "forget" permissions, but not invent new nesting facts.

[1]We simply write $\Pi \rightsquigarrow \Pi'$ if the two environments have the same variable set.

Figure 5.11 and 5.12 list some intuitionistic transformation rules based on logical implication. Transformations about permission scaling are given in Figure 5.13. Here, the $E \equiv E'$ acts as a short hand for $E \Rrightarrow E'$ and $E' \Rrightarrow E$.

$$\textsc{Tr-Subst}\quad \Pi, r = \rho \equiv [r \mapsto \rho]\Pi, r = \rho \qquad \textsc{Tr-Duplicate}\quad \Gamma \equiv \Gamma, \Gamma \qquad \textsc{Tr-Conj}\quad \Gamma_1, \Gamma_2 \equiv \Gamma_1 \wedge \Gamma_2$$

$$\textsc{Tr-Sub}\quad \frac{\Pi_1 \Rrightarrow \Pi_1'}{\Pi_1, \Pi_2 \Rrightarrow \Pi_1', \Pi_2} \qquad \textsc{Tr-Comm}\quad \Pi_1, \Pi_2 \equiv \Pi_2, \Pi_1 \qquad \textsc{Tr-Trans}\quad \frac{\Pi \Rrightarrow \Pi' \qquad \Pi' \Rrightarrow \Pi''}{\Pi \Rrightarrow \Pi''} \qquad \textsc{Tr-Ident}\quad \Pi \equiv \Pi$$

$$\textsc{Tr-Drop}\quad \Pi \Rrightarrow \emptyset \qquad \textsc{Tr-VarNull}\quad \frac{r \notin \Delta}{\Delta; \Pi \Rrightarrow \Delta \cup \{r\}\,; r = \$0, \Pi}$$

Figure 5.11: Transformation rules-1.

$$\textsc{Tr-CondTrue}\quad \Gamma, (\Gamma)?(\Pi_1) : (\Pi_2) \equiv \Gamma, \Pi_1 \qquad \textsc{Tr-CondFalse}\quad \neg(\Gamma), (\Gamma)?(\Pi_1) : (\Pi_2) \equiv \neg(\Gamma), \Pi_2$$

Figure 5.12: Transformation rules-2.

$$\textsc{Tr-FracEmpty}\quad \xi\emptyset \equiv \emptyset \qquad \textsc{Tr-FracFact}\quad \xi\Gamma \equiv \Gamma \qquad \textsc{Tr-FracCond}\quad \xi((\Gamma)?(\Pi_1) : (\Pi_2)) \equiv (\Gamma)?(\xi\Pi_1) : (\xi\Pi_2)$$

$$\textsc{Tr-FracImpl}\quad \xi(\Pi_1 \mathrel{-\!\!+} \Pi_2) \equiv \xi\Pi_1 \mathrel{-\!\!+} \xi\Pi_2 \qquad \textsc{Tr-FracComb}\quad \xi(\Pi_1, \Pi_2) \equiv \xi\Pi_1, \xi\Pi_2 \qquad \textsc{Tr-FracBase}\quad \xi(\xi'\Pi) \equiv (\xi\xi')\Pi$$

Figure 5.13: Transformation rules-3.

The rule Tr-Ident and Tr-Trans ensure that the transformation $\rightsquigarrow$ is a reflexive and transitive relation. Rule Tr-Duplicate shows that facts can be duplicated arbitrarily. Given a particular fact, a conditional permission can be simplified to its then or else part by Tr-CondTrue or Tr-CondFalse respectively.

Tr-Pack

$$\xi k : \texttt{ptr}(\rho), [r \mapsto \rho]\Pi \Rrightarrow \exists r\,.\,(\xi k \to r, \Pi)$$

Tr-Unpack

$$\frac{r' \text{ fresh}}{\Delta; \exists r\,.\,(\xi k \to r, \Pi) \Rrightarrow \{r'\} \cup \Delta; \xi k : \texttt{ptr}(r'), [r \mapsto r']\Pi}$$

Tr-Split

$$\Pi \equiv \texttt{1/2}\Pi, \texttt{1/2}\Pi$$

Tr-OrderGen1

$$r_1 < r.lv, r_2 > r.lv \Rrightarrow r_1 < r.lv, r_2 > r.lv, r_1 < r_2$$

Tr-OrderGen2

$$r_1 < r_2, r_2 < r_3 \Rrightarrow r_1 < r_2, r_2 < r_3, r_1 < r_3$$

Tr-Carve-Fill

$$\frac{\begin{array}{c}\Gamma = \xi k : \tau \prec k' \qquad \Pi_\Gamma = \xi_1 k_1 : \tau_1 \prec k', ..., \xi_n k_n : \tau_n \prec k' \\ \Pi = (\xi_1 k_1 : \tau_1, ..., \xi_n k_n : \tau_n) \multimap k' : \tau' \qquad \forall i \in [1..n].k \neq k_i\end{array}}{\Delta; \Gamma, \Pi_\Gamma, \Pi \equiv \Delta; \Gamma, \Pi_\Gamma, (\xi k : \tau, \xi_1 k_1 : \tau_1, ..., \xi_n k_n : \tau_n) \multimap k' : \tau', \xi k : \tau}$$

Figure 5.14: Transformation rules-4.

Scaling a conditional, an implication or a compound permission could be transformed into scaling their sub-branches by Tr-FracCond, Tr-FracImpl and Tr-FracComb. But scaling a fact does not change the fact by the rule Tr-FracFact.

Figure 5.14 lists some important transformations. Tr-Split shows that any permission could be scaled and split, while the Tr-Pack and Tr-Unpack introduces and eliminates the existential permission packages respectively.

If a level is higher than one object but lower than the other, then the level acts as an intermediate that indicates the former object is directly lower than the latter one by Tr-OrderGen1. The Tr-OrderGen2 shows that the partial order $<$ is transitive.

Carving out (or filling back) a fractional permission uses a complicated rule Tr-Carve-Fill which basically has two requirements above the line:

- There must be a corresponding nesting fact expressed as Γ;

- At least this fractional permission is still nested in its nester permission even though the nester permission may already have been carved out some permissions with different keys.

With the transformations, every permission checking rule permits a transformation around it:

$$\textsc{Trans}\quad \frac{E \rightsquigarrow E' \qquad E' \vdash_{\rho_L} e \Downarrow \tau \dashv E'' \qquad E'' \rightsquigarrow E'''}{E \vdash e \dashv \tau E''' \rho_L}$$

5.3 Summary

This chapter defines the permission checking process including permission typing rules and permission transformation rules. For any expression that nested most recently in a synchronized block holding a lock, if there exists an environment that includes all the required permissions for this expression, then the expression can be permission checked and has a permission type.

Since permissions are given in some complicated form, they may not be able to match the requirements of expressions directly, we use transform them to fit for the requirement by transformation rules. Furthermore, the transformation also demonstrates some important properties of permissions, such as factional and nesting.

Chapter 6

Consistency

To ensure the soundness, it is required that well permission-typed program can never go wrong. That is, neither data races nor deadlocks may happen at runtime for a permission checked program. To verify this property, we need to have the static environments E and dynamic runtime states μ match to each other according to our operational semantics and permission type rules. This is called "consistency".

As mentioned before, a memory μ maps a pair of object and field to another object:

$$\mu \in \text{Memory} = (O \times F) \rightharpoonup O$$

This is very simple and precise. But on the other side, permissions are defined in complicated and indirect forms (fractional, conditional ...), how to match these two? We bridge them by a *fractional heap.*

6.1 Fractional Heap

A fractional heap maps each location to a pair of a positive fraction and an object value (\$0 is a particular object reference represented as null pointer and uses 0 as its fraction):

$$h \in \text{Fractional Heap} = (O \times F) \rightarrow ((\mathbf{Q}^+, \mathbf{O}) \cup \{(\mathbf{0}, \$\mathbf{0})\})$$

Definition 6.1.0.1 (Empty Fractional Heap) *The empty fractional heap (written $\hat{\emptyset}$) maps every address to* $(0, \$0)$*, such that* $\forall l.\hat{\emptyset}(l) = (0, \$0)$.

Two fractional heaps can be combined by adding together the corresponding fractions if the values match and they are combined pointwise.

Definition 6.1.0.2 (Combination of Fractional Heaps) *Given two fractional heaps h_1 and h_2, then for any $l \in Dom(h_1) \cup Dom(h_2)$,*

$$(h_1 \hat{+} h_2)(l) = \begin{cases} h_1(l) & \textit{if } fst(h_2(l)) = 0 \\ h_2(l) & \textit{if } fst(h_1(l)) = 0 \\ (q_1 + q_2, snd(h_1(l))) & \textit{if } snd(h_1(l)) = snd(h_2(l)) \textit{ with } q_i = fst(h_i(l)) \\ \textit{undefined} & \textit{otherwise} \end{cases}$$

h_1 and h_2 are compatible *if their combination $h_1 \hat{+} h_2$ is defined.*

One fractional heap may be included into another.

Definition 6.1.0.3 *We say that one heap is included in another $h_1 \leq h_2$ if for every non-zero fraction in the first, it matches the second with at most that fraction: $\forall l.(q_1 = 0) \vee (q_1 \leq q_2 \wedge o_1 = o_2)$ where $(q_i, o_i) = h_i(l)$.*

Any fractional heap must be consistent with the actual memory.

Definition 6.1.0.4 *A fractional heap h is consistent with memory μ (written $h \leq \mu$) iff $\forall l \in Dom(h).(fst(h(l)) \in [0..1]) \wedge ((fst(h(l)) > 0) \Rightarrow ((l \in Dom(\mu)) \wedge (snd(h(l)) = \mu(l))))$.*

6.2 Flattening

After connecting a memory with a fractional heap ($h \leq \mu$), we turn to another side: connect the fractional heap with permissions.

We use $h \models_A \sigma(\Delta; \Pi)$ to indicate that: given a particular substitution σ, an environment $(\Delta; \Pi)$ can be modeled by a particular fractional heap h with an assumption A. Here, the assumption A is used to give truth value to permission facts.

In general, we need three values to 'witness" facts:

$A_\prec$ Fractional nesting relations assumed true. Used to assign truth value to nesting predicates;

A_{P} Instantiated predicates assumed true. Used to assign truth values to named predicates.

$A_{<}$ Partial order among locks assumed true. Used to assign truth values to partial order predicates.

The above three are combined as a shorthand $A = (A_{\preccurlyeq}, A_{\mathrm{P}}, A_{<})$ with which any fact can be evaluated to get a truth value: $A \vdash \Gamma \Downarrow \text{bool}$ (see Figure 6.1 for all the rules). We use l to range over addresses (o, f).

CB-True
$$A \vdash \mathtt{true} \Downarrow \text{true}$$

CB-Neg
$$\frac{A \vdash \Gamma \Downarrow b}{A \vdash \neg(\Gamma) \Downarrow \neg b}$$

CB-And
$$\frac{A \vdash \Gamma_1 \Downarrow b_1 \qquad A \vdash \Gamma_2 \Downarrow b_2}{A \vdash \Gamma_1 \wedge \Gamma_2 \Downarrow b_1 \wedge b_2}$$

CB-SubType
$$A \vdash o \in C \Downarrow (\text{class}(o) \preccurlyeq C)$$

CB-Equal
$$A \vdash o{=}o' \Downarrow o = o'$$

CB-Axiom
$$\frac{\Gamma \in A}{A \vdash \Gamma \Downarrow \text{true}}$$

CB-Exist
$$\frac{A \vdash [\delta \mapsto X]\Gamma \Downarrow \text{true}}{A \vdash \exists \delta\,.\,(\Gamma) \Downarrow \text{true}}$$

CB-Pred
$$\frac{A \cup \{p(\overline{o})\} \vdash [\overline{r \mapsto o}]P(p) \Downarrow \text{true}}{A \vdash p(\overline{o}) \Downarrow \text{true}}$$

Figure 6.1: Auxiliary rules for facts: $A \vdash \Gamma \Downarrow \text{bool}$.

Definition 6.2.0.5 (Flattening) *Given an assumption A, if a variable-free permission Π with an obligation permission Ψ can be modeled by a fractional heap h, then we say the permission Π can be flattened (written as $h; \Psi \models_A \Pi$), where the obligation Ψ is treated as a restricted permission that can be discharged symbolically from the Π. The detailed flattening rules are in Figure 6.2.*

Both of the empty permission and a true predicate are flattened to the empty fractional heap by CP-Empty and CP-True respectively. For a conditional permission

$$\textsc{CP-Empty}\quad \hat{\emptyset}; \emptyset \models_A \emptyset \qquad \textsc{CP-True}\quad \frac{A \vdash \Gamma \Downarrow \text{true}}{\hat{\emptyset}; \emptyset \models_A \Gamma} \qquad \textsc{CP-Combine}\quad \frac{h_i; \Psi_i \models_A \Pi_i}{h_1 \hat{+} h_2; \Psi_1, \Psi_2 \models_A \Pi_1, \Pi_2} \qquad \textsc{CP-Frac}\quad \frac{\vdash \xi \Downarrow q \qquad h; \Psi \models_A \Pi}{qh; \xi\Psi \models_A \xi\Pi}$$

$$\textsc{CP-TrueCond}\quad \frac{A \vdash \Gamma \Downarrow \text{true} \qquad h; \Psi \models_A \Pi_1}{h; \Psi \models_A (\Gamma)?(\Pi_1) : (\Pi_2)} \qquad \textsc{CP-FalseCond}\quad \frac{A \vdash \neg(\Gamma) \Downarrow \text{true} \qquad h; \Psi \models_A \Pi_2}{h; \Psi \models_A (\Gamma)?(\Pi_1) : (\Pi_2)}$$

$$\textsc{CP-Implication}\quad \frac{h; \Psi', \Psi \models_A \Pi}{h; \Psi' \models_A \Psi -\!\!+ \Pi} \qquad \textsc{CP-Exist}\quad \frac{h; \Psi \models_A [\delta \mapsto X]\Pi}{h; \Psi \models_A \exists\delta\,.\,(\Pi)} \qquad \textsc{CP-Field}\quad \frac{\Pi_{\prec} = \sum_{\Pi \prec o.f \in A} \Pi \qquad h; \Psi \models_A \Pi_{\prec}}{h \hat{+} [o.f \mapsto (1, o')]; \Psi \models_A o.f \rightarrow o'}$$

Figure 6.2: Flattening rules: $h; \Psi \models_A \Pi$.

$(\Gamma)?(\Pi_1) : (\Pi_2)$, we have two rules CP-TrueCond and CP-FalseCond corresponding to the two possibilities for Γ's truth value. CP-Implication applies to the implication permission $\Psi -\!\!+ \Pi$ which moves the Ψ to left-hand-side as an obligation when flattening the Π. CP-Combine shows that if a permission Π and an obligation Ψ can be split into Π_1, Π_2 and Ψ_1, Ψ_2 respectively and h_i is the fractional heap for Π_i with an obligation Ψ_i, then the Π with the obligation Ψ can be flattened into the combination of h_1 and h_2. From the rule CP-Frac, scaling the permission Π and the obligation Ψ with the same amount ξ will cause its fractional heap to be scaled by q, where q is evaluated from ξ. For an existential type, we use the actual objects thus typed to substitute for the object variables in CP-Exist.

To flatten a unit permission, we use CP-Field. which applies to any field. Assuming this field acts as a location that being nested some other permissions, to flatten this unit permission, we need to carve out all the permissions that nested in it and combine them as $\Pi_{\prec}$. If the $\Pi_{\prec}$ can be flattened using the same assumption A and obligation Ψ into h, then the whole unit permission can be flattened into $h \hat{+} [o.f \mapsto (1, o')]$.

With the flattening, the connection between h and environment $E = (\Delta; \Pi)$ is given as a judgement:

$$\frac{h; \emptyset \models_A \sigma\Pi}{h \models_A \sigma(\Delta; \Pi)}$$

6.3 Consistency Checking

With one connection between memory μ and h ($h \leq \mu$) as well as the other connection between h and environment E ($h \models_A \sigma E$), we are able to establish the consistency between μ and E directly which is given as $\mu; A \models \Delta; \Pi$.

We assume there exists a substitution $\sigma : \Delta \to \emptyset$ mapping from object and fraction variables in Δ to absolute addresses or numbers between zero and one respectively. The empty set range means that the result of the substitution uses no variables at all.

With the σ, we are able to flatten the environment $(\Delta; \Pi)$ into a fractional heap h with referring to an assumption A. Moreover, the h should be consistent with memory μ. This is given as a judgement:

$$\frac{\sigma : \Delta \to \emptyset \qquad h \models_A \sigma(\Delta; \Pi) \qquad h \leq \mu}{\mu; A \models \Delta; \Pi}$$

6.4 Soundness

The fundamental soundness of this permission type system depends on a theorem of progress and preservation:

Theorem 6.4.0.6 (Progress and Preservation) *For an expression $e = e_1 || e_2 || \ldots || e_n$ where $1 \leq n$ and assuming that e_i nested most recently in a synchronized block holding o_{L-i} (or $o_{L-i} = o_{\mathit{thisThread}\text{-}i}$ if it is not inside any synchronized block), then if e_i can be checked by a variable-free permission Π_i such that $\emptyset; \Pi_i \vdash_{o_{L-i}} e_i \Downarrow \mathtt{ptr}(\rho_i) \dashv \Delta_i''; \Pi_i''$,*

$\Pi = \Pi_1, \Pi_2, ..., \Pi_n$, $\Delta'' = \bigcup_{i=1}^{n} \Delta''_i$, *and a memory* μ *with an assumption* A *is consistent with* $(\Pi, \widetilde{\Pi})$, *then either* e *is a value of the form* $o_1||o_2||\ldots||o_n$ *or one of its branch can be evaluated one-step further :*

- *either there is no thread spawn and* $(\mu; e_i) \xrightarrow{i} (\mu'; e'_i)$ *happens in some thread* i *and* $e' = e_1||...||e'_i||...||e_n$*; or*
- *a new thread is created* $(\mu; e) \xrightarrow{i} (\mu'; e||e_{n+1})$ *for some thread* i *and* $e' = e||e_{n+1}$*;*

and there exists $\Pi', \widetilde{\Pi}'$, σ *and* A' *such that* σ *will substitute away some of the new type variables (*$\sigma : \Delta \rightarrow \emptyset$ *with* $\Delta \subseteq \Delta''$*),* $\emptyset; \Pi' \vdash e' \Downarrow \texttt{ptr}(\sigma\rho) \dashv \sigma\Delta''; \sigma\Pi''$ *and* μ' *with the assumption* A' *is consistent with* $(\Pi', \widetilde{\Pi}')$ *where* $A \subseteq A'$.

(Sketch) We combine the permission type rules with the operational semantics and prove by the induction on permission checking rules case by case. This will be similar to our previous work [5, 7].

Soundness indicates that a well permission-typed program can never go wrong (free of data races and deadlocks). Here, we say a program is well permission-typed if all its method bodies can be permission checked by their method types in permission.

6.5 Summary

This chapter defines the consistency property of our permission system. Since the permission analysis is totally static, it's necessary to find a way to show that it abstracts the run-time properties properly. We define a fractional heap to bridge the static environment and the dynamic run-time memory. Based on the consistency, we are able to define the progress and preservation theorem and show the soundness of our permission system as well.

Chapter 7 Implementation

We have implemented a prototype tool that embodies the permission analysis techniques described in previous chapters. This tool extends the current permission analysis [William Retert, personal communication] designing for sequential programs with borrowing some annotations from Greenhouse's lock analysis. In this chapter, we briefly show several design issues and then give some permission checking examples.

7.1 Fluid-Level Design

The Fluid Project is focused on creating practicable tools for programmers to assure and evolve real programs [45]. The current permission analysis designed by Retert is developed as part of the Fluid project.

In Fluid, the control-flow graph (CFG) has one node for each atomic action, such that reading a variable, assigning a field, or ignoring a value. For example, the simplified CFG for a code segment

```
this.f1 = this.f2 + x;
```

consists of the straight-line sequence of nodes that

1. reads `this` onto the stack;
2. reads `this` onto the stack;
3. pops off an object reference; pushes its `f2` field onto the stack;
4. reads `x` onto the stack;

5. pops two elements from the stack, adds them and pushes the result back to the stack;

6. swaps the top two elements of the stack; pops off an object reference; copies the new top of the stack into field `f1` of the object;

7. pops the top of the stack.

Actually the CFG in Fluid has many more nodes than the above ones. We ignore them in this document.

The current permission analysis uses a `PermissionLattice` which is composed of four parts:

- A `LocationMap` that maps local variables or location-field pairs to their associated locations;
- A `PermissionSet` that maps location-field pairs to their fractions;
- A `FactSet` that includes all permission facts, such as reference equalities, type assertions, nesting facts and so on;
- A `ValueStack` that includes location values of expressions;

That is:

$$\begin{array}{rcl} \texttt{LocationField} & : & \texttt{Location} \times \texttt{Field} \\ \texttt{LocationMap} & : & (\texttt{Var} \cup \texttt{LocationField}) \rightarrow \texttt{Location} \\ \texttt{PermissionSet} & : & \texttt{LocationField} \rightarrow \texttt{Fraction} \\ \texttt{FactSet} & : & \text{Set of } \texttt{Fact} \\ \texttt{ValueStack} & : & \text{Stack of } \texttt{Location} \\ \texttt{PermissionLattice} & : & \texttt{LocationMap} \times \texttt{PermissionSet} \times \texttt{FactSet} \times \texttt{ValueStack} \end{array}$$

To extend current permission analysis to multithreaded programs, We need to add lock ordering information to the lattice. Besides the new order facts, we additionally attach a `LockStack` to the lattice, where the element of the `LockStack` is a pair of locks: the most recent acquired lock and the most recent holding lock. These two may not be the same at some cases because of the reentrant locks, for example,

```
synch o1 do { synch o2 do { synch o1 do { ...  } } }
```

Assuming the $o1$ has a lower level than the $o2$, then any expression appears inside of the second synchronization of $o1$ has a different recent acquired lock with the most recent holding lock. They are $o1$ and $o2$ respectively.

$$\texttt{ConcurrencyPermissionLattice} : \texttt{PermissionLattice} \times \texttt{LockStack}$$

Figure 7.1 shows how to maintain the `ConcurrencyPermissionLattice` for each node based on the simple language defined in Chapter 3. In practice, they are represented as transfer functions and should be included some assurance about whether the input `ConcurrencyPermissionLattice` is good enough to perform this node.

As mentioned before, a `fork` will spawn a new thread and take away *some* permissions. This is safe enough for the theoretical permission checking, but it is not practical, since we need to know exactly which permissions will be thrown into the new thread and which ones stay. In our prototype tool, we keep all the permission remaining in the current thread, but duplicate facts in new threads. The permission transformation tells us it's safe to arbitrarily duplicate facts. There may be another choice: besides facts, permissions coming with the unique parameters can leave the current thread and go into a new one.

7.2 Examples for Permission Checking

We combine method types, permission checking rules as well as transformations to give the permission checking for several methods step by step based on nodes. For clarity

Nodes	Actions
n	push literal number n on `ValueStack`.
x	read variable x into the `ValueStack`.
_ *op* _	pop two elements from the `ValueStack`, *op* them and push the result back.
_ `==` _	pop two elements from the `ValueStack`, compare them and determine branch with adding different facts to `FactSet`.
_ .f	pop off an object reference from `ValueStack`; push its f field back.
_ .f `=` _	swap the top two elements of the `ValueStack`; pop off an object reference; update the `LocationMap`.
`new` C	create a new object of type C and push it into `ValueStack` and update the `LocationMap`.
_ ;	pop top element of the `ValueStack` and discard it.
`let x =` _	pop top element of the `ValueStack` and update `LocationMap`.
_ .m(_)	pop off n actual parameters and a receiver object from `ValueStack` and then call the method with it; push the return value back.
_ .$C\#m$(_)	pop off n actual parameters and a receiver object from `ValueStack` and then call the method in type C; push the return value back.
`return` _ ;	pop the top element of the `ValueStack` and return it.
`skip`	push the \$0 into the `ValueStack`.
`acquire` _ ;	pop the top element l of the `ValueStack` and peek the top element (l_1, l_2) of the `LockStack`, if l has a higher level than the l_2, then push (l, l) into the `LockStack`; or if $[(l, \texttt{Prot}) \mapsto 1] \in$ `PermissionSet`, then push (l, l_2) into the `LockStack`.
`release`;	pop the top element from the `LockStack`.
`fork`;	push the \$0 into the `ValueStack`.

Table 7.1: CFG nodes and actions

reason, we don't use the `ConcurrencyPermissionLattice` mentioned above, but the traditional permission forms defined in Chapter 4. Therefore, some detailed information may not be included. The first and the last permission in the method body correspond to the input and output permission of this method. To save space, the {...} after an action is equal to the permission before that action. Also, we ignore the action "_ ;" in the code.

7.2.1 Node.setNext

This method shows the usage of method effect. A "`writes`" effect basically gives a unit permission.

`void setNext( Node<g> n ) writes (this.next) {`

$$\left\{\begin{array}{r}(\neg(r_{\mathtt{n}} = \$0))?(\mathit{Node}(r_{\mathtt{n}}, r_{\mathtt{g}}), r_{\mathtt{n}}.\mathtt{All} \rightarrow \$0) : (\emptyset),\\ \exists r\,.\,(r_{\mathtt{this}}.\mathtt{next} \rightarrow r, (\neg(r = \$0))?(r.\mathtt{All} \rightarrow \$0, \mathit{Node}(r, r_{\mathtt{g}})) : (\emptyset)),\\ \mathit{Node}(r_{\mathtt{this}}, r_{\mathtt{g}})\end{array}\right\} \overset{\text{TR-UNPACK}}{\rightsquigarrow}$$

$$\left\{\begin{array}{r}(\neg(r_{\mathtt{n}} = \$0))?(\mathit{Node}(r_{\mathtt{n}}, r_{\mathtt{g}}), r_{\mathtt{n}}.\mathtt{All} \rightarrow \$0) : (\emptyset),\\ r_{\mathtt{this}}.\mathtt{next} \rightarrow r_{\mathtt{next}}, (\neg(r_{\mathtt{next}} = \$0))?(r_{\mathtt{next}}.\mathtt{All} \rightarrow \$0, \mathit{Node}(r_{\mathtt{next}}, r_{\mathtt{g}})) : (\emptyset),\\ \mathit{Node}(r_{\mathtt{this}}, r_{\mathtt{g}})\end{array}\right\}$$

`this` $\Downarrow \mathtt{ptr}(r_{\mathtt{this}})$

{...}

`n` $\Downarrow \mathtt{ptr}(r_n)$

{...}

`_.next = _` $\Downarrow \mathtt{ptr}(r_n)$

$$\left\{\begin{array}{r}(\neg(r_{\mathtt{n}} = \$0))?(\mathit{Node}(r_{\mathtt{n}}, r_{\mathtt{g}}), r_{\mathtt{n}}.\mathtt{All} \rightarrow \$0) : (\emptyset),\\ r_{\mathtt{this}}.\mathtt{next} \rightarrow r_{\mathtt{n}}, (\neg(r_{\mathtt{next}} = \$0))?(r_{\mathtt{next}}.\mathtt{All} \rightarrow \$0, \mathit{Node}(r_{\mathtt{next}}, r_{\mathtt{g}})) : (\emptyset),\\ \mathit{Node}(r_{\mathtt{this}}, r_{\mathtt{g}})\end{array}\right\} \overset{\text{TR-PACK}}{\rightsquigarrow}$$

$$\left\{\begin{array}{r}\exists r\,.\,(r_{\mathtt{this}}.\mathtt{next} \rightarrow r, (\neg(r = \$0))?(r.\mathtt{All} \rightarrow \$0, \mathit{Node}(r, r_{\mathtt{g}})) : (\emptyset)),\\ (\neg(r_{\mathtt{next}} = \$0))?(r_{\mathtt{next}}.\mathtt{All} \rightarrow \$0, \mathit{Node}(r_{\mathtt{next}}, r_{\mathtt{g}})) : (\emptyset),\\ \mathit{Node}(r_{\mathtt{this}}, r_{\mathtt{g}})\end{array}\right\} \overset{\text{TR-DROP}}{\rightsquigarrow}$$

$$\left\{\begin{array}{r}\exists r\,.\,(r_{\mathtt{this}}.\mathtt{next} \rightarrow r, (\neg(r = \$0))?(r.\mathtt{All} \rightarrow \$0, \mathit{Node}(r, r_{\mathtt{g}})) : (\emptyset)),\\ \mathit{Node}(r_{\mathtt{this}}, r_{\mathtt{g}})\end{array}\right\}$$

`null` $\Downarrow \mathtt{ptr}(\$0)$

{...}

```
  return _ ;
```
$$\left\{\begin{array}{r}\exists r\,.\,(r_{\texttt{this}}.\texttt{next}\rightarrow r,(\neg(r=\$0))?(r.\texttt{All}\rightarrow\$0,Node(r,r_{\texttt{g}})):(\emptyset)),\\ Node(r_{\texttt{this}},r_{\texttt{g}}),r_{\texttt{ret}}=\$0\end{array}\right\}$$
```
}
```

7.2.2 Node.Node

A constructor in our system will implicitly call the super's constructor at the beginning and return the receiver object at the end.

```
Node<g>( shared Object d ) {
```
$$\left\{\begin{array}{r}r_{\texttt{this}}\in Node,\\ r_{\texttt{this}}.\texttt{next}\rightarrow\$0,r_{\texttt{this}}.\texttt{datum}\rightarrow\$0,r_{\texttt{this}}.\texttt{All}\rightarrow\$0,\\ (\neg(r_{\texttt{d}}=\$0))?(Object(r_{\texttt{d}}),r_{\texttt{d}}.\texttt{All}\rightarrow\$0\prec r_{\texttt{d}}.\texttt{Prot}):(\emptyset)\end{array}\right\}$$

`this` $\Downarrow \texttt{ptr}(r_{\texttt{this}})$

$\{...\}$

`_ .Object#Object()` $\Downarrow \texttt{ptr}(r_{\texttt{this}})$

$\{...\}$

`this` $\Downarrow \texttt{ptr}(r_{\texttt{this}})$

$\{...\}$

`d` $\Downarrow \texttt{ptr}(r_{\texttt{d}})$

$\{...\}$

`_ .d = _` $\Downarrow \texttt{ptr}(r_{\texttt{d}})$

$$\left\{\begin{array}{r}r_{\texttt{this}}\in Node,\\ r_{\texttt{this}}.\texttt{next}\rightarrow\$0,r_{\texttt{this}}.\texttt{datum}\rightarrow r_{\texttt{d}},r_{\texttt{this}}.\texttt{All}\rightarrow\$0,\\ (\neg(r_{\texttt{d}}=\$0))?(Object(r_{\texttt{d}}),r_{\texttt{d}}.\texttt{All}\rightarrow\$0\prec r_{\texttt{d}}.\texttt{Prot}):(\emptyset)\end{array}\right\}\overset{\textsc{Tr-Pack}}{\leadsto}$$

$$\left\{\begin{array}{r}r_{\texttt{this}}\in Node,\\ r_{\texttt{this}}.\texttt{next}\rightarrow\$0,r_{\texttt{this}}.\texttt{All}\rightarrow\$0,\\ \exists r\,.\,(r_{\texttt{this}}.\texttt{datum}\rightarrow r,(\neg(r=\$0))?(Object(r),r.\texttt{All}\rightarrow\$0\prec r.\texttt{Prot}):(\emptyset))\end{array}\right\}\overset{\textsc{Tr-Nest}}{\leadsto}$$

$$\left\{\begin{array}{r}r_{\texttt{this}}\in Node,\\ r_{\texttt{this}}.\texttt{next}\rightarrow\$0,r_{\texttt{this}}.\texttt{All}\rightarrow\$0,\\ \exists r\,.\,(r_{\texttt{this}}.\texttt{datum}\rightarrow r,(\neg(r=\$0))?(Object(r),r.\texttt{All}\rightarrow\$0\prec r.\texttt{Prot}):(\emptyset))\\ \prec r_{\texttt{this}}.\texttt{All}\end{array}\right\}\overset{\textsc{Tr-VarNull}}{\leadsto}$$

$$\left\{\begin{array}{r} r_{\texttt{this}} \in \mathit{Node}, \\ r_{\texttt{this}}.\texttt{next} \to \$0, r_{\texttt{this}}.\texttt{All} \to \$0, \\ r_{\texttt{next}} = \$0, \\ \exists r\,.\,(r_{\texttt{this}}.\texttt{datum} \to r, (\neg(r = \$0))?(\mathit{Object}(r), r.\texttt{All} \to \$0 \prec r.\texttt{Prot}) : (\emptyset)) \\ \prec r_{\texttt{this}}.\texttt{All} \end{array}\right\} \overset{\text{TR-CONDFALSE}}{\rightsquigarrow}$$

$$\left\{\begin{array}{r} r_{\texttt{this}} \in \mathit{Node}, \\ r_{\texttt{this}}.\texttt{next} \to \$0, r_{\texttt{this}}.\texttt{All} \to \$0, \\ r_{\texttt{next}} = \$0, (\neg(r_{\texttt{next}} = \$0))?(\mathit{Node}(r_{\texttt{next}}, r_{\texttt{g}}), r_{\texttt{next}}.\texttt{All} \to \$0) : (\emptyset), \\ \exists r\,.\,(r_{\texttt{this}}.\texttt{datum} \to r, (\neg(r = \$0))?(\mathit{Object}(r), r.\texttt{All} \to \$0 \prec r.\texttt{Prot}) : (\emptyset)) \\ \prec r_{\texttt{this}}.\texttt{All} \end{array}\right\} \overset{\text{TR-SUBST}}{\rightsquigarrow}$$

$$\left\{\begin{array}{r} r_{\texttt{this}} \in \mathit{Node}, \\ r_{\texttt{this}}.\texttt{next} \to r_{\texttt{next}}, r_{\texttt{this}}.\texttt{All} \to \$0, \\ r_{\texttt{next}} = \$0, (\neg(r_{\texttt{next}} = \$0))?(\mathit{Node}(r_{\texttt{next}}, r_{\texttt{g}}), r_{\texttt{next}}.\texttt{All} \to \$0) : (\emptyset), \\ \exists r\,.\,(r_{\texttt{this}}.\texttt{datum} \to r, (\neg(r = \$0))?(\mathit{Object}(r), r.\texttt{All} \to \$0 \prec r.\texttt{Prot}) : (\emptyset)) \\ \prec r_{\texttt{this}}.\texttt{All} \end{array}\right\} \overset{\text{TR-PACK}}{\rightsquigarrow}$$

$$\left\{\begin{array}{r} r_{\texttt{this}} \in \mathit{Node}, \\ r_{\texttt{this}}.\texttt{All} \to \$0, r_{\texttt{next}} = \$0, \\ \exists r\,.\,(r_{\texttt{this}}.\texttt{next} \to r, (\neg(r = \$0))?(\mathit{Node}(r, r_{\texttt{g}}), r.\texttt{All} \to \$0) : (\emptyset)), \\ \exists r\,.\,(r_{\texttt{this}}.\texttt{datum} \to r, (\neg(r = \$0))?(\mathit{Object}(r), r.\texttt{All} \to \$0 \prec r.\texttt{Prot}) : (\emptyset)) \\ \prec r_{\texttt{this}}.\texttt{All} \end{array}\right\} \overset{\text{TR-NEST}}{\rightsquigarrow}$$

$$\left\{\begin{array}{r} r_{\texttt{this}} \in \mathit{Node}, \\ r_{\texttt{this}}.\texttt{All} \to \$0, r_{\texttt{next}} = \$0, \\ \exists r\,.\,(r_{\texttt{this}}.\texttt{next} \to r, (\neg(r = \$0))?(\mathit{Node}(r, r_{\texttt{g}}), r.\texttt{All} \to \$0) : (\emptyset)) \prec r_{\texttt{g}}.\texttt{Prot}, \\ \exists r\,.\,(r_{\texttt{this}}.\texttt{datum} \to r, (\neg(r = \$0))?(\mathit{Object}(r), r.\texttt{All} \to \$0 \prec r.\texttt{Prot}) : (\emptyset)) \\ \prec r_{\texttt{this}}.\texttt{All} \end{array}\right\} \overset{\text{TR-CONJ}}{\rightsquigarrow}$$

$$\left\{\begin{array}{r} r_{\texttt{this}}.\texttt{All} \to \$0, r_{\texttt{next}} = \$0, \\ \mathit{Node}(r_{\texttt{this}}, r_{\texttt{g}}) \end{array}\right\} \overset{\text{TR-DROP}}{\rightsquigarrow}$$

$$\left\{\begin{array}{r} r_{\texttt{this}}.\texttt{All} \to \$0, \\ \mathit{Node}(r_{\texttt{this}}, r_{\texttt{g}}) \end{array}\right\}$$

`this` $\quad \texttt{ptr}(r_{\texttt{this}})$

$\{\ldots\}$

`return _ ;`

$$\left\{\begin{array}{r} r_{\mathtt{this}}.\mathtt{All} \rightarrow \$0, \\ Node(r_{\mathtt{this}}, r_{\mathtt{g}}), \\ r_{\mathrm{ret}} = r_{\mathtt{this}} \end{array}\right\}$$

}

7.2.3 LinkList.insert

This method may acquire the `this` in its body. The order between the `this` and the surrounding lock expressed as r_{holding} is given as a conditional permission. Its false-part corresponds to the case that the `this` has already been held before the method entry, while the true-part gives an required order [1]. Accordingly, the lock acquirement and release actions are based on the two possibilities and the conditional permission is properly maintained.

`void insert( shared Object d ) uses (this) {`

$$\left\{\begin{array}{r} (r_{\mathrm{holding}} < r_{\mathtt{this}})?(\emptyset) : (r_{\mathtt{this}}.\mathtt{Prot} \rightarrow \$0), \\ (\neg(r_{\mathtt{d}} = \$0))?(r_{\mathtt{d}}.\mathtt{All} \rightarrow \$0 \prec r_{\mathtt{d}}.\mathtt{Prot}, Object(r_{\mathtt{d}})) : (\emptyset), \\ LinkList(r_{\mathtt{this}}) \end{array}\right\}$$

`this` $\Downarrow \mathtt{ptr}(r_{\mathtt{this}})$

{...}

`acquire _`

$$\left\{\begin{array}{r} r_{\mathtt{this}}.\mathtt{Prot} \rightarrow \$0, \\ (\neg(r_{\mathtt{d}} = \$0))?(r_{\mathtt{d}}.\mathtt{All} \rightarrow \$0 \prec r_{\mathtt{d}}.\mathtt{Prot}, Object(r_{\mathtt{d}})) : (\emptyset), \\ LinkList(r_{\mathtt{this}}) \end{array}\right\}$$

`new Node` $\Downarrow \mathtt{ptr}(r)$

$$\left\{\begin{array}{r} \neg(r = \$0), r \in Node, \\ r.\mathtt{next} \rightarrow \$0, r.\mathtt{datum} \rightarrow \$0, r.\mathtt{All} \rightarrow \$0, \\ r_{\mathtt{this}}.\mathtt{Prot} \rightarrow \$0, \\ (\neg(r_{\mathtt{d}} = \$0))?(r_{\mathtt{d}}.\mathtt{All} \rightarrow \$0 \prec r_{\mathtt{d}}.\mathtt{Prot}, Object(r_{\mathtt{d}})) : (\emptyset), \\ LinkList(r_{\mathtt{this}}) \end{array}\right\}$$

`Node<this>(d)` $\Downarrow \mathtt{ptr}(r)$

$$\left\{\begin{array}{r} r.\mathtt{All} \rightarrow \$0, \\ \neg(r = \$0), Node(r, r_{\mathtt{this}}), \\ r_{\mathtt{this}}.\mathtt{Prot} \rightarrow \$0, \\ LinkList(r_{\mathtt{this}}) \end{array}\right\}$$

[1]Although the true-part is empty, the condition part shows this order.

`let newNode = _  {`

$$\left\{\begin{aligned} r_{\texttt{newNode}} = r, \\ r.\texttt{All} \to \$0, \\ \neg(r = \$0), Node(r, r_{\texttt{this}}), \\ r_{\texttt{this}}.\texttt{Prot} \to \$0, \\ LinkList(r_{\texttt{this}}) \end{aligned}\right\}$$

`newNode` $\Downarrow \texttt{ptr}(r_{\texttt{newNode}})$

$\{...\} \overset{\text{TR-CARVE-FILL}}{\rightsquigarrow}$

$$\left\{\begin{aligned} r_{\texttt{newNode}} = r, \\ r.\texttt{All} \to \$0, \\ \neg(r = \$0), Node(r, r_{\texttt{this}}), \\ \exists r'.(r_{\texttt{this}}.\texttt{head} \to r', (\neg(r' = \$0))?(r'.\texttt{All} \to \$0, Node(r', r_{\texttt{this}})) : (\emptyset)), \\ \exists r'.(r_{\texttt{this}}.\texttt{head} \to r', (\neg(r' = \$0))?(r'.\texttt{All} \to \$0, Node(r', r_{\texttt{this}})) : (\emptyset)) \\ -\!\!\!+ r_{\texttt{this}}.\texttt{Prot} \to \$0, \\ LinkList(r_{\texttt{this}}) \end{aligned}\right\} \overset{\text{TR-UNPACK}}{\rightsquigarrow}$$

$$\left\{\begin{aligned} r_{\texttt{newNode}} = r, \\ r.\texttt{All} \to \$0, \\ \neg(r = \$0), Node(r, r_{\texttt{this}}), \\ r_{\texttt{this}}.\texttt{head} \to r_{\texttt{head}}, (\neg(r_{\texttt{head}} = \$0))?(r_{\texttt{head}}.\texttt{All} \to \$0, Node(r_{\texttt{head}}, r_{\texttt{this}})) : (\emptyset), \\ \exists r'.(r_{\texttt{this}}.\texttt{head} \to r', (\neg(r' = \$0))?(r'.\texttt{All} \to \$0, Node(r', r_{\texttt{this}})) : (\emptyset)) \\ -\!\!\!+ r_{\texttt{this}}.\texttt{Prot} \to \$0, \\ LinkList(r_{\texttt{this}}) \end{aligned}\right\}$$

`this` $\Downarrow \texttt{ptr}(r_{\texttt{this}})$

$\{...\}$

`_ .head` $\Downarrow \texttt{ptr}(r_{\texttt{head}})$

$\{...\} \overset{\text{TR-CARVE-FILL}}{\rightsquigarrow}$

$$\left\{\begin{array}{r}
r_{\texttt{newNode}} = r, \\
r.\texttt{All} \rightarrow \$0, \\
\neg(r = \$0), \mathit{Node}(r, r_{\texttt{this}}), \\
r_{\texttt{this}}.\texttt{head} \rightarrow r_{\texttt{head}}, (\neg(r_{\texttt{head}} = \$0))?(r_{\texttt{head}}.\texttt{All} \rightarrow \$0, \mathit{Node}(r_{\texttt{head}}, r_{\texttt{this}})) : (\emptyset), \\
\exists r'.\,(r.\texttt{next} \rightarrow r', (\neg(r' = \$0))?(\mathit{Node}(r', r_{\texttt{this}}), r'.\texttt{All} \rightarrow \$0) : (\emptyset)), \\
(\exists r'.\,(r_{\texttt{this}}.\texttt{head} \rightarrow r', (\neg(r' = \$0))?(r'.\texttt{All} \rightarrow \$0, \mathit{Node}(r', r_{\texttt{this}})) : (\emptyset)), \\
\exists r'.\,(r.\texttt{next} \rightarrow r', (\neg(r' = \$0))?(\mathit{Node}(r', r_{\texttt{this}}), r'.\texttt{All} \rightarrow \$0) : (\emptyset))) \\
-\!\!+\; r_{\texttt{this}}.\texttt{Prot} \rightarrow \$0, \\
\mathit{LinkList}(r_{\texttt{this}})
\end{array}\right\}$$

$\overset{\text{Tr-Subst}}{\rightsquigarrow}$

$$\left\{\begin{array}{r}
r_{\texttt{newNode}} = r, \\
r_{\texttt{newNode}}.\texttt{All} \rightarrow \$0, \\
\neg(r_{\texttt{newNode}} = \$0), \mathit{Node}(r_{\texttt{newNode}}, r_{\texttt{this}}), \\
r_{\texttt{this}}.\texttt{head} \rightarrow r_{\texttt{head}}, (\neg(r_{\texttt{head}} = \$0))?(r_{\texttt{head}}.\texttt{All} \rightarrow \$0, \mathit{Node}(r_{\texttt{head}}, r_{\texttt{this}})) : (\emptyset), \\
\exists r'.\,(r_{\texttt{newNode}}.\texttt{next} \rightarrow r', (\neg(r' = \$0))?(\mathit{Node}(r', r_{\texttt{this}}), r'.\texttt{All} \rightarrow \$0) : (\emptyset)), \\
(\exists r'.\,(r_{\texttt{this}}.\texttt{head} \rightarrow r', (\neg(r' = \$0))?(r'.\texttt{All} \rightarrow \$0, \mathit{Node}(r', r_{\texttt{this}})) : (\emptyset)), \\
\exists r'.\,(r_{\texttt{newNode}}.\texttt{next} \rightarrow r', (\neg(r' = \$0))?(\mathit{Node}(r', r_{\texttt{this}}), r'.\texttt{All} \rightarrow \$0) : (\emptyset))) \\
-\!\!+\; r_{\texttt{this}}.\texttt{Prot} \rightarrow \$0, \\
\mathit{LinkList}(r_{\texttt{this}})
\end{array}\right\}$$

`_ .setNext( _ )` $\Downarrow \texttt{ptr}(\$0)$

$$\left\{\begin{array}{r}
r_{\texttt{newNode}} = r, \\
r_{\texttt{newNode}}.\texttt{All} \rightarrow \$0, \\
\neg(r_{\texttt{newNode}} = \$0), \mathit{Node}(r_{\texttt{newNode}}, r_{\texttt{this}}), \\
r_{\texttt{this}}.\texttt{head} \rightarrow r_{\texttt{head}}, \\
\exists r'.\,(r_{\texttt{newNode}}.\texttt{next} \rightarrow r', (\neg(r' = \$0))?(\mathit{Node}(r', r_{\texttt{this}}), r'.\texttt{All} \rightarrow \$0) : (\emptyset)), \\
(\exists r'.\,(r_{\texttt{this}}.\texttt{head} \rightarrow r', (\neg(r' = \$0))?(r'.\texttt{All} \rightarrow \$0, \mathit{Node}(r', r_{\texttt{this}})) : (\emptyset)), \\
\exists r'.\,(r_{\texttt{newNode}}.\texttt{next} \rightarrow r', (\neg(r' = \$0))?(\mathit{Node}(r', r_{\texttt{this}}), r'.\texttt{All} \rightarrow \$0) : (\emptyset))) \\
-\!\!+\; r_{\texttt{this}}.\texttt{Prot} \rightarrow \$0, \\
\mathit{LinkList}(r_{\texttt{this}})
\end{array}\right\}$$

`this` $\Downarrow \texttt{ptr}(r_{\texttt{this}})$

{...}

$\texttt{newNode} \qquad \Downarrow \texttt{ptr}(r_{\texttt{newNode}})$

$\{...\}$

$\texttt{\_ .head = \_} \qquad \Downarrow \texttt{ptr}(r_{\texttt{newNode}})$

$$\left\{\begin{array}{r} r_{\texttt{newNode}} = r, \\ r_{\texttt{newNode}}.\texttt{All} \rightarrow \$0, \\ \neg(r_{\texttt{newNode}} = \$0), Node(r_{\texttt{newNode}}, r_{\texttt{this}}), \\ r_{\texttt{this}}.\texttt{head} \rightarrow r_{\texttt{newNode}}, \\ \exists r'.(r_{\texttt{newNode}}.\texttt{next} \rightarrow r', (\neg(r' = \$0))?(Node(r', r_{\texttt{this}}), r'.\texttt{All} \rightarrow \$0) : (\emptyset)), \\ (\exists r'.(r_{\texttt{this}}.\texttt{head} \rightarrow r', (\neg(r' = \$0))?(r'.\texttt{All} \rightarrow \$0, Node(r', r_{\texttt{this}})) : (\emptyset)), \\ \exists r'.(r_{\texttt{newNode}}.\texttt{next} \rightarrow r', (\neg(r' = \$0))?(Node(r', r_{\texttt{this}}), r'.\texttt{All} \rightarrow \$0) : (\emptyset))) \\ -\!\!+ r_{\texttt{this}}.\texttt{Prot} \rightarrow \$0, \\ LinkList(r_{\texttt{this}}) \end{array}\right\}$$

$\underset{\rightsquigarrow}{\text{TR-CONDTRUE}}$

$$\left\{\begin{array}{r} r_{\texttt{newNode}} = r, \\ \neg(r_{\texttt{newNode}} = \$0), \\ (\neg(r_{\texttt{newNode}} = \$0))?(r_{\texttt{newNode}}.\texttt{All} \rightarrow \$0, Node(r_{\texttt{newNode}}, r_{\texttt{this}})) : (\emptyset), \\ r_{\texttt{this}}.\texttt{head} \rightarrow r_{\texttt{newNode}}, \\ \exists r'.(r_{\texttt{newNode}}.\texttt{next} \rightarrow r', (\neg(r' = \$0))?(Node(r', r_{\texttt{this}}), r'.\texttt{All} \rightarrow \$0) : (\emptyset)), \\ (\exists r'.(r_{\texttt{this}}.\texttt{head} \rightarrow r', (\neg(r' = \$0))?(r'.\texttt{All} \rightarrow \$0, Node(r', r_{\texttt{this}})) : (\emptyset)), \\ \exists r'.(r_{\texttt{newNode}}.\texttt{next} \rightarrow r', (\neg(r' = \$0))?(Node(r', r_{\texttt{this}}), r'.\texttt{All} \rightarrow \$0) : (\emptyset))) \\ -\!\!+ r_{\texttt{this}}.\texttt{Prot} \rightarrow \$0, \\ LinkList(r_{\texttt{this}}) \end{array}\right\}$$

$\underset{\rightsquigarrow}{\text{TR-PACK}}$

$$\left\{\begin{array}{r} r_{\texttt{newNode}} = r, \\ \neg(r_{\texttt{newNode}} = \$0), \\ \exists r' \,.\, (r_{\texttt{this}}.\texttt{head} \rightarrow r', (\neg(r' = \$0))?(r'.\texttt{All} \rightarrow \$0, Node(r', r_{\texttt{this}})) : (\emptyset)), \\ \exists r' \,.\, (r_{\texttt{newNode}}.\texttt{next} \rightarrow r', (\neg(r' = \$0))?(Node(r', r_{\texttt{this}}), r'.\texttt{All} \rightarrow \$0) : (\emptyset)), \\ (\exists r' \,.\, (r_{\texttt{this}}.\texttt{head} \rightarrow r', (\neg(r' = \$0))?(r'.\texttt{All} \rightarrow \$0, Node(r', r_{\texttt{this}})) : (\emptyset)), \\ \exists r' \,.\, (r_{\texttt{newNode}}.\texttt{next} \rightarrow r', (\neg(r' = \$0))?(Node(r', r_{\texttt{this}}), r'.\texttt{All} \rightarrow \$0) : (\emptyset))) \\ -\!\!\!+ \; r_{\texttt{this}}.\texttt{Prot} \rightarrow \$0, \\ LinkList(r_{\texttt{this}}) \end{array}\right\}$$

$$\overset{\text{Tr-Carve-Fill}}{\rightsquigarrow}$$

$$\left\{\begin{array}{r} r_{\texttt{newNode}} = r, \\ \neg(r_{\texttt{newNode}} = \$0), \\ \exists r' \,.\, (r_{\texttt{newNode}}.\texttt{next} \rightarrow r', (\neg(r' = \$0))?(Node(r', r_{\texttt{this}}), r'.\texttt{All} \rightarrow \$0) : (\emptyset)), \\ \exists r' \,.\, (r_{\texttt{newNode}}.\texttt{next} \rightarrow r', (\neg(r' = \$0))?(Node(r', r_{\texttt{this}}), r'.\texttt{All} \rightarrow \$0) : (\emptyset)) \\ -\!\!\!+ \; r_{\texttt{this}}.\texttt{Prot} \rightarrow \$0, \\ LinkList(r_{\texttt{this}}) \end{array}\right\}$$

$$\overset{\text{Tr-Carve-Fill}}{\rightsquigarrow}$$

$$\left\{\begin{array}{r} r_{\texttt{newNode}} = r, \\ \neg(r_{\texttt{newNode}} = \$0), \\ r_{\texttt{this}}.\texttt{Prot} \rightarrow \$0, \\ LinkList(r_{\texttt{this}}) \end{array}\right\}$$

```
}
```

$$\left\{\begin{array}{r} \neg(r = \$0), \\ r_{\texttt{this}}.\texttt{Prot} \rightarrow \$0, \\ LinkList(r_{\texttt{this}}) \end{array}\right\} \overset{\text{Tr-Drop}}{\rightsquigarrow}$$

$$\left\{\begin{array}{r} r_{\texttt{this}}.\texttt{Prot} \rightarrow \$0, \\ LinkList(r_{\texttt{this}}) \end{array}\right\}$$

```
release
```

$$\left\{\begin{array}{r} (r_{\text{holding}} < r_{\texttt{this}})?(\emptyset) : (r_{\texttt{this}}.\texttt{Prot} \rightarrow \$0), \\ LinkList(r_{\texttt{this}}) \end{array}\right\}$$

```
null      ptr($0)
```

$\{...\}$

`return _ ;`

$$\left\{\begin{array}{r} (r_{\text{holding}} < r_{\texttt{this}})?(\emptyset) : (r_{\texttt{this}}.\texttt{Prot} \to \$0), \\ \mathit{LinkList}(r_{\texttt{this}}), \\ r_{\text{ret}} = \$0 \end{array}\right\}$$

`}`

7.2.4 Account.deposit

A “`requires`” annotation gives the unit permission to the `Prot` data group of the lock object, which should be returned intact.

`void deposit( int x ) requires (this) {`

$$\left\{\begin{array}{r} \mathit{Account}(r_{\texttt{this}}), \\ r_{\texttt{this}}.\texttt{Prot} \to \$0, r_{\texttt{x}} = \texttt{int} \end{array}\right\} \overset{\text{Tr-Carve-Fill}}{\rightsquigarrow}$$

$$\left\{\begin{array}{r} \mathit{Account}(r_{\texttt{this}}), \\ r_{\texttt{this}}.\texttt{balance} \to \texttt{int}, r_{\texttt{this}}.\texttt{balance} \to \texttt{int} \mathrel{-\!\!+} r_{\texttt{this}}.\texttt{Prot} \to \$0, r_{\texttt{x}} = \texttt{int} \end{array}\right\}$$

`this` $\Downarrow \texttt{ptr}(r_{\texttt{this}})$

{...}

`this` $\Downarrow \texttt{ptr}(r_{\texttt{this}})$

{...}

`_ .balance` $\Downarrow \texttt{ptr}(\texttt{int})$

$$\left\{\begin{array}{r} \mathit{Account}(r_{\texttt{this}}), \\ r_{\texttt{this}}.\texttt{balance} \to \texttt{int}, r_{\texttt{this}}.\texttt{balance} \to \texttt{int} \mathrel{-\!\!+} r_{\texttt{this}}.\texttt{Prot} \to \$0, r_{\texttt{x}} = \texttt{int} \end{array}\right\}$$

`x` $\Downarrow \texttt{ptr}(\texttt{int})$

{...}

`_ + _` $\Downarrow \texttt{ptr}(\texttt{int})$

{...}

`_ .balance = _` $\Downarrow \texttt{ptr}(\texttt{int})$

$$\left\{\begin{array}{r} \mathit{Account}(r_{\texttt{this}}), \\ r_{\texttt{this}}.\texttt{balance} \to \texttt{int}, r_{\texttt{this}}.\texttt{balance} \to \texttt{int} \mathrel{-\!\!+} r_{\texttt{this}}.\texttt{Prot} \to \$0, r_{\texttt{x}} = \texttt{int} \end{array}\right\}$$

$$\overset{\text{Tr-Carve-Fill}}{\rightsquigarrow}$$

$$\left\{\begin{array}{r} \mathit{Account}(r_{\texttt{this}}), \\ r_{\texttt{this}}.\texttt{Prot} \to \$0, r_{\texttt{x}} = \texttt{int} \end{array}\right\} \overset{\text{Tr-Drop}}{\rightsquigarrow}$$

$$\left\{\begin{array}{r} \mathit{Account}(r_{\texttt{this}}), \\ r_{\texttt{this}}.\texttt{Prot} \to \$0 \end{array}\right\}$$

```
  null        $\Downarrow \texttt{ptr}(\$0)$
  {...}
  return _ ;
  $\left\{\begin{array}{r} Account(r_{\texttt{this}}), \\ r_{\texttt{this}}.\texttt{Prot} \rightarrow \$0, \\ r_{\text{ret}} = \$0 \end{array}\right\}$
}
```

7.2.5 CombinedAccount.savings2checking

This method "`uses`" two locks which is referred by two fields of the receiver object. Similar as before, two conditional permissions are included at the beginning to show the order relation between them and the most recent holding lock at the method entry. Since the lock `this.checking` is going to be acquired inside of the synchronized block holding `this.savings`, a proper order between them should be granted. This is done by a transformation that uses the `this.lv` as a intermediate to get the direct relation between these two locks.

```
void savings2checking( int x ) uses (this.savings, this.checking) {
  $\left\{\begin{array}{r} z_1 r_{\texttt{this}}.\texttt{checking} \rightarrow r_{\texttt{checking}}, \neg(r_{\texttt{checking}} = \$0), z_1 r_{\texttt{checking}}.\texttt{All} \rightarrow \$0, \\ Account(r_{\texttt{checking}}), r_{\texttt{checking}} < r_{\texttt{this}}.\texttt{lv}, \\ z_2 r_{\texttt{this}}.\texttt{savings} \rightarrow r_{\texttt{savings}}, \neg(r_{\texttt{savings}} = \$0), z_2 r_{\texttt{savings}}.\texttt{All} \rightarrow \$0, \\ Account(r_{\texttt{savings}}), r_{\texttt{savings}} > r_{\texttt{this}}.\texttt{lv}, \\ (\exists r \,.\, (z_1 r_{\texttt{this}}.\texttt{checking} \rightarrow r, \neg(r = \$0), z_1 r.\texttt{All} \rightarrow \$0, Account(r), r < r_{\texttt{this}}.\texttt{lv}), \\ \exists r \,.\, (z_2 r_{\texttt{this}}.\texttt{savings} \rightarrow r, \neg(r = \$0), z_2 r.\texttt{All} \rightarrow \$0, Account(r), r > r_{\texttt{this}}.\texttt{lv})) \\ -\!\!+ \$0.\texttt{Immutable} \rightarrow \$0, \\ (r_{\text{holding}} < r_{\texttt{checking}})?(\emptyset) : (r_{\texttt{checking}}.\texttt{Prot} \rightarrow \$0), \\ (r_{\text{holding}} < r_{\texttt{savings}})?(\emptyset) : (r_{\texttt{savings}}.\texttt{Prot} \rightarrow \$0), \\ CombinedAccount(r_{\texttt{this}}), \\ r_{\texttt{x}} = \texttt{int} \end{array}\right\}$
  this        $\Downarrow \texttt{ptr}(r_{\texttt{this}})$
  {...}
  _ .checking        $\Downarrow \texttt{ptr}(r_{\texttt{checking}})$
  {...}
  acquire _
```

$$\left\{\begin{array}{r} z_1 r_{\texttt{this}}.\texttt{checking} \rightarrow r_{\texttt{checking}}, \neg(r_{\texttt{checking}} = \$0), z_1 r_{\texttt{checking}}.\texttt{All} \rightarrow \$0, \\ \mathit{Account}(r_{\texttt{checking}}), r_{\texttt{checking}} < r_{\texttt{this}}.\texttt{lv}, \\ z_2 r_{\texttt{this}}.\texttt{savings} \rightarrow r_{\texttt{savings}}, \neg(r_{\texttt{savings}} = \$0), z_2 r_{\texttt{savings}}.\texttt{All} \rightarrow \$0, \\ \mathit{Account}(r_{\texttt{savings}}), r_{\texttt{savings}} > r_{\texttt{this}}.\texttt{lv}, \\ (\exists r\,.\,(z_1 r_{\texttt{this}}.\texttt{checking} \rightarrow r, \neg(r = \$0), z_1 r.\texttt{All} \rightarrow \$0, \mathit{Account}(r), r < r_{\texttt{this}}.\texttt{lv}), \\ \exists r\,.\,(z_2 r_{\texttt{this}}.\texttt{savings} \rightarrow r, \neg(r = \$0), z_2 r.\texttt{All} \rightarrow \$0, \mathit{Account}(r), r > r_{\texttt{this}}.\texttt{lv})) \\ \multimap \$0.\texttt{Immutable} \rightarrow \$0, \\ r_{\texttt{checking}}.\texttt{Prot} \rightarrow \$0, \\ (r_{\text{holding}} < r_{\texttt{savings}})?(\emptyset) : (r_{\texttt{savings}}.\texttt{Prot} \rightarrow \$0), \\ \mathit{CombinedAccount}(r_{\texttt{this}}), \\ r_{\texttt{x}} = \texttt{int} \end{array}\right\}$$

$\texttt{this} \quad \Downarrow \texttt{ptr}(r_{\texttt{this}})$

$\{...\}$

$\texttt{\_ .savings} \quad \Downarrow \texttt{ptr}(r_{\texttt{savings}})$

$\{...\} \overset{\text{Tr-OrderGen1}}{\rightsquigarrow}$

$$\left\{\begin{array}{r} z_1 r_{\texttt{this}}.\texttt{checking} \rightarrow r_{\texttt{checking}}, \neg(r_{\texttt{checking}} = \$0), z_1 r_{\texttt{checking}}.\texttt{All} \rightarrow \$0, \\ \mathit{Account}(r_{\texttt{checking}}), r_{\texttt{checking}} < r_{\texttt{this}}.\texttt{lv}, \\ z_2 r_{\texttt{this}}.\texttt{savings} \rightarrow r_{\texttt{savings}}, \neg(r_{\texttt{savings}} = \$0), z_2 r_{\texttt{savings}}.\texttt{All} \rightarrow \$0, \\ \mathit{Account}(r_{\texttt{savings}}), r_{\texttt{savings}} > r_{\texttt{this}}.\texttt{lv}, \\ (\exists r\,.\,(z_1 r_{\texttt{this}}.\texttt{checking} \rightarrow r, \neg(r = \$0), z_1 r.\texttt{All} \rightarrow \$0, \mathit{Account}(r), r < r_{\texttt{this}}.\texttt{lv}), \\ \exists r\,.\,(z_2 r_{\texttt{this}}.\texttt{savings} \rightarrow r, \neg(r = \$0), z_2 r.\texttt{All} \rightarrow \$0, \mathit{Account}(r), r > r_{\texttt{this}}.\texttt{lv})) \\ \multimap \$0.\texttt{Immutable} \rightarrow \$0, \\ r_{\texttt{checking}}.\texttt{Prot} \rightarrow \$0, \\ (r_{\text{holding}} < r_{\texttt{savings}})?(\emptyset) : (r_{\texttt{savings}}.\texttt{Prot} \rightarrow \$0), \\ \mathit{CombinedAccount}(r_{\texttt{this}}), \\ r_{\texttt{x}} = \texttt{int}, \\ r_{\texttt{checking}} < r_{\texttt{savings}} \end{array}\right\}$$

$\texttt{acquire \_}$

$$\left\{\begin{array}{r} z_1 r_{\texttt{this}}.\texttt{checking} \rightarrow r_{\texttt{checking}}, \neg(r_{\texttt{checking}} = \$0), z_1 r_{\texttt{checking}}.\texttt{All} \rightarrow \$0, \\ \mathit{Account}(r_{\texttt{checking}}), r_{\texttt{checking}} < r_{\texttt{this}}.\texttt{lv}, \\ z_2 r_{\texttt{this}}.\texttt{savings} \rightarrow r_{\texttt{savings}}, \neg(r_{\texttt{savings}} = \$0), z_2 r_{\texttt{savings}}.\texttt{All} \rightarrow \$0, \\ \mathit{Account}(r_{\texttt{savings}}), r_{\texttt{savings}} > r_{\texttt{this}}.\texttt{lv}, \\ (\exists r\,.\,(z_1 r_{\texttt{this}}.\texttt{checking} \rightarrow r, \neg(r = \$0), z_1 r.\texttt{All} \rightarrow \$0, \mathit{Account}(r), r < r_{\texttt{this}}.\texttt{lv}), \\ \exists r\,.\,(z_2 r_{\texttt{this}}.\texttt{savings} \rightarrow r, \neg(r = \$0), z_2 r.\texttt{All} \rightarrow \$0, \mathit{Account}(r), r > r_{\texttt{this}}.\texttt{lv})) \\ \multimap \$0.\texttt{Immutable} \rightarrow \$0, \\ r_{\texttt{checking}}.\texttt{Prot} \rightarrow \$0, r_{\texttt{savings}}.\texttt{Prot} \rightarrow \$0, \\ \mathit{CombinedAccount}(r_{\texttt{this}}), \\ r_{\texttt{x}} = \texttt{int}, \\ r_{\texttt{checking}} < r_{\texttt{savings}} \end{array}\right\}$$

$\texttt{this} \quad \Downarrow \texttt{ptr}(r_{\texttt{this}})$

$\{...\}$

$\texttt{\_ .savings} \quad \Downarrow \texttt{ptr}(r_{\texttt{savings}})$

$\{...\}$

$\texttt{x} \quad \Downarrow \texttt{ptr(int)}$

$\{...\} \overset{\text{TR-DUPLICATE}}{\rightsquigarrow}$

$$\left\{\begin{array}{r} z_1 r_{\texttt{this}}.\texttt{checking} \rightarrow r_{\texttt{checking}}, \neg(r_{\texttt{checking}} = \$0), z_1 r_{\texttt{checking}}.\texttt{All} \rightarrow \$0, \\ \mathit{Account}(r_{\texttt{checking}}), r_{\texttt{checking}} < r_{\texttt{this}}.\texttt{lv}, \\ z_2 r_{\texttt{this}}.\texttt{savings} \rightarrow r_{\texttt{savings}}, \neg(r_{\texttt{savings}} = \$0), z_2 r_{\texttt{savings}}.\texttt{All} \rightarrow \$0, \\ \mathit{Account}(r_{\texttt{savings}}), r_{\texttt{savings}} > r_{\texttt{this}}.\texttt{lv}, \\ (\exists r\,.\,(z_1 r_{\texttt{this}}.\texttt{checking} \rightarrow r, \neg(r = \$0), z_1 r.\texttt{All} \rightarrow \$0, \mathit{Account}(r), r < r_{\texttt{this}}.\texttt{lv}), \\ \exists r\,.\,(z_2 r_{\texttt{this}}.\texttt{savings} \rightarrow r, \neg(r = \$0), z_2 r.\texttt{All} \rightarrow \$0, \mathit{Account}(r), r > r_{\texttt{this}}.\texttt{lv})) \\ \multimap \$0.\texttt{Immutable} \rightarrow \$0, \\ r_{\texttt{checking}}.\texttt{Prot} \rightarrow \$0, r_{\texttt{savings}}.\texttt{Prot} \rightarrow \$0, \\ \mathit{CombinedAccount}(r_{\texttt{this}}), \\ r_{\texttt{x}} = \texttt{int}, r_{\texttt{x}} = \texttt{int}, \\ r_{\texttt{checking}} < r_{\texttt{savings}} \end{array}\right\}$$

$\texttt{\_ .withdraw(\_)} \quad \Downarrow \texttt{ptr}(\$0)$

$$\left\{\begin{array}{r}
z_1 r_{\texttt{this}}.\texttt{checking} \to r_{\texttt{checking}}, \neg(r_{\texttt{checking}} = \$0), z_1 r_{\texttt{checking}}.\texttt{All} \to \$0, \\
\mathit{Account}(r_{\texttt{checking}}), r_{\texttt{checking}} < r_{\texttt{this}}.\texttt{lv}, \\
z_2 r_{\texttt{this}}.\texttt{savings} \to r_{\texttt{savings}}, \neg(r_{\texttt{savings}} = \$0), z_2 r_{\texttt{savings}}.\texttt{All} \to \$0, \\
\mathit{Account}(r_{\texttt{savings}}), r_{\texttt{savings}} > r_{\texttt{this}}.\texttt{lv}, \\
(\exists r\,.\,(z_1 r_{\texttt{this}}.\texttt{checking} \to r, \neg(r = \$0), z_1 r.\texttt{All} \to \$0, \mathit{Account}(r), r < r_{\texttt{this}}.\texttt{lv}), \\
\exists r\,.\,(z_2 r_{\texttt{this}}.\texttt{savings} \to r, \neg(r = \$0), z_2 r.\texttt{All} \to \$0, \mathit{Account}(r), r > r_{\texttt{this}}.\texttt{lv})) \\
-\!\!+\ \$0.\texttt{Immutable} \to \$0, \\
r_{\texttt{checking}}.\texttt{Prot} \to \$0, r_{\texttt{savings}}.\texttt{Prot} \to \$0, \\
\mathit{CombinedAccount}(r_{\texttt{this}}), \\
r_{\texttt{x}} = \texttt{int}, \\
r_{\texttt{checking}} < r_{\texttt{savings}}
\end{array}\right\}$$

$\texttt{this} \qquad \Downarrow \texttt{ptr}(r_{\texttt{this}})$

{...}

$\texttt{\_ .checking} \qquad \Downarrow \texttt{ptr}(r_{\texttt{savings}})$

{...}

$\texttt{x} \qquad \Downarrow \texttt{ptr(int)}$

{...}

$\texttt{\_ .deposit(\_)} \qquad \Downarrow \texttt{ptr}(\$0)$

$$\left\{\begin{array}{r}
z_1 r_{\texttt{this}}.\texttt{checking} \to r_{\texttt{checking}}, \neg(r_{\texttt{checking}} = \$0), z_1 r_{\texttt{checking}}.\texttt{All} \to \$0, \\
\mathit{Account}(r_{\texttt{checking}}), r_{\texttt{checking}} < r_{\texttt{this}}.\texttt{lv}, \\
z_2 r_{\texttt{this}}.\texttt{savings} \to r_{\texttt{savings}}, \neg(r_{\texttt{savings}} = \$0), z_2 r_{\texttt{savings}}.\texttt{All} \to \$0, \\
\mathit{Account}(r_{\texttt{savings}}), r_{\texttt{savings}} > r_{\texttt{this}}.\texttt{lv}, \\
(\exists r\,.\,(z_1 r_{\texttt{this}}.\texttt{checking} \to r, \neg(r = \$0), z_1 r.\texttt{All} \to \$0, \mathit{Account}(r), r < r_{\texttt{this}}.\texttt{lv}), \\
\exists r\,.\,(z_2 r_{\texttt{this}}.\texttt{savings} \to r, \neg(r = \$0), z_2 r.\texttt{All} \to \$0, \mathit{Account}(r), r > r_{\texttt{this}}.\texttt{lv})) \\
-\!\!+\ \$0.\texttt{Immutable} \to \$0, \\
r_{\texttt{checking}}.\texttt{Prot} \to \$0, r_{\texttt{savings}}.\texttt{Prot} \to \$0, \\
\mathit{CombinedAccount}(r_{\texttt{this}}), \\
r_{\texttt{checking}} < r_{\texttt{savings}}
\end{array}\right\}$$

`release`

$$\left\{\begin{array}{r} z_1 r_{\texttt{this}}.\texttt{checking} \rightarrow r_{\texttt{checking}}, \neg(r_{\texttt{checking}} = \$0), z_1 r_{\texttt{checking}}.\texttt{All} \rightarrow \$0, \\ \mathit{Account}(r_{\texttt{checking}}), r_{\texttt{checking}} < r_{\texttt{this}}.\texttt{lv}, \\ z_2 r_{\texttt{this}}.\texttt{savings} \rightarrow r_{\texttt{savings}}, \neg(r_{\texttt{savings}} = \$0), z_2 r_{\texttt{savings}}.\texttt{All} \rightarrow \$0, \\ \mathit{Account}(r_{\texttt{savings}}), r_{\texttt{savings}} > r_{\texttt{this}}.\texttt{lv}, \\ (\exists r\,.\,(z_1 r_{\texttt{this}}.\texttt{checking} \rightarrow r, \neg(r = \$0), z_1 r.\texttt{All} \rightarrow \$0, \mathit{Account}(r), r < r_{\texttt{this}}.\texttt{lv}), \\ \exists r\,.\,(z_2 r_{\texttt{this}}.\texttt{savings} \rightarrow r, \neg(r = \$0), z_2 r.\texttt{All} \rightarrow \$0, \mathit{Account}(r), r > r_{\texttt{this}}.\texttt{lv})) \\ \multimap \$0.\texttt{Immutable} \rightarrow \$0, \\ r_{\texttt{checking}}.\texttt{Prot} \rightarrow \$0, \\ (r_{\text{holding}} < r_{\texttt{savings}})?(\emptyset) : (r_{\texttt{savings}}.\texttt{Prot} \rightarrow \$0), \\ \mathit{CombinedAccount}(r_{\texttt{this}}), \\ r_{\texttt{checking}} < r_{\texttt{savings}} \end{array}\right\}$$

`release`

$$\left\{\begin{array}{r} z_1 r_{\texttt{this}}.\texttt{checking} \rightarrow r_{\texttt{checking}}, \neg(r_{\texttt{checking}} = \$0), z_1 r_{\texttt{checking}}.\texttt{All} \rightarrow \$0, \\ \mathit{Account}(r_{\texttt{checking}}), r_{\texttt{checking}} < r_{\texttt{this}}.\texttt{lv}, \\ z_2 r_{\texttt{this}}.\texttt{savings} \rightarrow r_{\texttt{savings}}, \neg(r_{\texttt{savings}} = \$0), z_2 r_{\texttt{savings}}.\texttt{All} \rightarrow \$0, \\ \mathit{Account}(r_{\texttt{savings}}), r_{\texttt{savings}} > r_{\texttt{this}}.\texttt{lv}, \\ (\exists r\,.\,(z_1 r_{\texttt{this}}.\texttt{checking} \rightarrow r, \neg(r = \$0), z_1 r.\texttt{All} \rightarrow \$0, \mathit{Account}(r), r < r_{\texttt{this}}.\texttt{lv}), \\ \exists r\,.\,(z_2 r_{\texttt{this}}.\texttt{savings} \rightarrow r, \neg(r = \$0), z_2 r.\texttt{All} \rightarrow \$0, \mathit{Account}(r), r > r_{\texttt{this}}.\texttt{lv})) \\ \multimap \$0.\texttt{Immutable} \rightarrow \$0, \\ (r_{\text{holding}} < r_{\texttt{checking}})?(\emptyset) : (r_{\texttt{checking}}.\texttt{Prot} \rightarrow \$0), \\ (r_{\text{holding}} < r_{\texttt{savings}})?(\emptyset) : (r_{\texttt{savings}}.\texttt{Prot} \rightarrow \$0), \\ \mathit{CombinedAccount}(r_{\texttt{this}}), \\ r_{\texttt{checking}} < r_{\texttt{savings}} \end{array}\right\}$$

$\overset{\textsc{Tr-Drop}}{\rightsquigarrow}$

$$\left\{\begin{array}{r} z_1 r_{\texttt{this}}.\texttt{checking} \rightarrow r_{\texttt{checking}}, \neg(r_{\texttt{checking}} = \$0), z_1 r_{\texttt{checking}}.\texttt{All} \rightarrow \$0,\\ \mathit{Account}(r_{\texttt{checking}}), r_{\texttt{checking}} < r_{\texttt{this}}.\texttt{lv},\\ z_2 r_{\texttt{this}}.\texttt{savings} \rightarrow r_{\texttt{savings}}, \neg(r_{\texttt{savings}} = \$0), z_2 r_{\texttt{savings}}.\texttt{All} \rightarrow \$0,\\ \mathit{Account}(r_{\texttt{savings}}), r_{\texttt{savings}} > r_{\texttt{this}}.\texttt{lv},\\ (\exists r\,.\,(z_1 r_{\texttt{this}}.\texttt{checking} \rightarrow r, \neg(r = \$0), z_1 r.\texttt{All} \rightarrow \$0, \mathit{Account}(r), r < r_{\texttt{this}}.\texttt{lv}),\\ \exists r\,.\,(z_2 r_{\texttt{this}}.\texttt{savings} \rightarrow r, \neg(r = \$0), z_2 r.\texttt{All} \rightarrow \$0, \mathit{Account}(r), r > r_{\texttt{this}}.\texttt{lv}))\\ \multimap \$0.\texttt{Immutable} \rightarrow \$0,\\ (r_{\text{holding}} < r_{\texttt{checking}})?(\emptyset) : (r_{\texttt{checking}}.\texttt{Prot} \rightarrow \$0),\\ (r_{\text{holding}} < r_{\texttt{savings}})?(\emptyset) : (r_{\texttt{savings}}.\texttt{Prot} \rightarrow \$0),\\ \mathit{CombinedAccount}(r_{\texttt{this}}) \end{array}\right\}$$

```
null      ⇓ ptr($0)
{...}
return _ ;
```

$$\left\{\begin{array}{r} z_1 r_{\texttt{this}}.\texttt{checking} \rightarrow r_{\texttt{checking}}, \neg(r_{\texttt{checking}} = \$0), z_1 r_{\texttt{checking}}.\texttt{All} \rightarrow \$0,\\ \mathit{Account}(r_{\texttt{checking}}), r_{\texttt{checking}} < r_{\texttt{this}}.\texttt{lv},\\ z_2 r_{\texttt{this}}.\texttt{savings} \rightarrow r_{\texttt{savings}}, \neg(r_{\texttt{savings}} = \$0), z_2 r_{\texttt{savings}}.\texttt{All} \rightarrow \$0,\\ \mathit{Account}(r_{\texttt{savings}}), r_{\texttt{savings}} > r_{\texttt{this}}.\texttt{lv},\\ (\exists r\,.\,(z_1 r_{\texttt{this}}.\texttt{checking} \rightarrow r, \neg(r = \$0), z_1 r.\texttt{All} \rightarrow \$0, \mathit{Account}(r), r < r_{\texttt{this}}.\texttt{lv}),\\ \exists r\,.\,(z_2 r_{\texttt{this}}.\texttt{savings} \rightarrow r, \neg(r = \$0), z_2 r.\texttt{All} \rightarrow \$0, \mathit{Account}(r), r > r_{\texttt{this}}.\texttt{lv}))\\ \multimap \$0.\texttt{Immutable} \rightarrow \$0,\\ (r_{\text{holding}} < r_{\texttt{checking}})?(\emptyset) : (r_{\texttt{checking}}.\texttt{Prot} \rightarrow \$0),\\ (r_{\text{holding}} < r_{\texttt{savings}})?(\emptyset) : (r_{\texttt{savings}}.\texttt{Prot} \rightarrow \$0),\\ \mathit{CombinedAccount}(r_{\texttt{this}}),\\ r_{\text{ret}} = \$0 \end{array}\right\}$$

```
}
```

7.2.6 CombinedAccount.double_savings

This method includes two thread spawn inside. The permission coming to the `fork` action will be split into two parts: one remains in the current thread and the other flows into the new thread. The remaining permission shows up right after the spawn expression. We give another process to show the permission checking in the new thread. The two thread spawn presenting in this method have the same expression body as well as the same incoming permission.

```
void double_savings(int x) uses (this.savings)
                   holdingLock < this.savings {
```

$$\left\{\begin{array}{r}
z_2 r_{\texttt{this}}.\texttt{savings} \rightarrow r_{\texttt{savings}}, \neg(r_{\texttt{savings}} = \$0), z_2 r_{\texttt{savings}}.\texttt{All} \rightarrow \$0,\\
\mathit{Account}(r_{\texttt{savings}}), r_{\texttt{savings}} > r_{\texttt{this}}.\texttt{lv},\\
\exists r\,.\,(z_2 r_{\texttt{this}}.\texttt{savings} \rightarrow r, \neg(r = \$0), z_2 r.\texttt{All} \rightarrow \$0, \mathit{Account}(r), r > r_{\texttt{this}}.\texttt{lv})\\
\mathrel{-\!\!+} \$0.\texttt{Immutable} \rightarrow \$0,\\
r_{\text{holding}} < r_{\texttt{savings}},\\
\mathit{CombinedAccount}(r_{\texttt{this}}),\\
r_{\texttt{x}} = \texttt{int}
\end{array}\right\}$$

`this` $\Downarrow \texttt{ptr}(r_{\texttt{this}})$

$\{\ldots\}$

`_ .savings` $\Downarrow \texttt{ptr}(r_{\texttt{savings}})$

$\{\ldots\}$

`let s = _`

$$\left\{\begin{array}{r}
r_{\texttt{s}} = r_{\texttt{savings}},\\
z_2 r_{\texttt{this}}.\texttt{savings} \rightarrow r_{\texttt{savings}}, \neg(r_{\texttt{savings}} = \$0), z_2 r_{\texttt{savings}}.\texttt{All} \rightarrow \$0,\\
\mathit{Account}(r_{\texttt{savings}}), r_{\texttt{savings}} > r_{\texttt{this}}.\texttt{lv},\\
\exists r\,.\,(z_2 r_{\texttt{this}}.\texttt{savings} \rightarrow r, \neg(r = \$0), z_2 r.\texttt{All} \rightarrow \$0, \mathit{Account}(r), r > r_{\texttt{this}}.\texttt{lv})\\
\mathrel{-\!\!+} \$0.\texttt{Immutable} \rightarrow \$0,\\
r_{\text{holding}} < r_{\texttt{savings}},\\
\mathit{CombinedAccount}(r_{\texttt{this}}),\\
r_{\texttt{x}} = \texttt{int}
\end{array}\right\}$$

$\overset{\textsc{Tr-Subst}}{\rightsquigarrow}$

$$\left\{\begin{array}{r}
r_{\texttt{s}} = r_{\texttt{savings}},\\
z_2 r_{\texttt{this}}.\texttt{savings} \rightarrow r_{\texttt{s}}, \neg(r_{\texttt{s}} = \$0), z_2 r_{\texttt{s}}.\texttt{All} \rightarrow \$0,\\
\mathit{Account}(r_{\texttt{s}}), r_{\texttt{s}} > r_{\texttt{this}}.\texttt{lv},\\
\exists r\,.\,(z_2 r_{\texttt{this}}.\texttt{savings} \rightarrow r, \neg(r = \$0), z_2 r.\texttt{All} \rightarrow \$0, \mathit{Account}(r), r > r_{\texttt{this}}.\texttt{lv})\\
\mathrel{-\!\!+} \$0.\texttt{Immutable} \rightarrow \$0,\\
r_{\text{holding}} < r_{\texttt{s}},\\
\mathit{CombinedAccount}(r_{\texttt{this}}),\\
r_{\texttt{x}} = \texttt{int}
\end{array}\right\}$$

$$\overset{\textsc{Tr-Duplicate}}{\rightsquigarrow}$$

$$\left\{\begin{array}{r}
r_{\mathtt{s}} = r_{\mathtt{savings}}, \\
z_2 r_{\mathtt{this}}.\mathtt{savings} \rightarrow r_{\mathtt{s}}, \neg(r_{\mathtt{s}} = \$0), z_2 r_{\mathtt{s}}.\mathtt{All} \rightarrow \$0, \\
\mathit{Account}(r_{\mathtt{s}}), r_{\mathtt{s}} > r_{\mathtt{this}}.\mathtt{lv}, \\
\exists r\,.\,(z_2 r_{\mathtt{this}}.\mathtt{savings} \rightarrow r, \neg(r = \$0), z_2 r.\mathtt{All} \rightarrow \$0, \mathit{Account}(r), r > r_{\mathtt{this}}.\mathtt{lv}) \\
-\!\!+ \$0.\mathtt{Immutable} \rightarrow \$0, \\
r_{\text{holding}} < r_{\mathtt{s}}, \\
\mathit{CombinedAccount}(r_{\mathtt{this}}), \\
r_{\mathtt{x}} = \mathtt{int}, \\
\neg(r_{\mathtt{s}} = \$0), \mathit{Account}(r_{\mathtt{s}}), r_{\mathtt{x}} = \mathtt{int}
\end{array}\right\}$$

$$\mathtt{fork} \qquad \Downarrow \mathtt{ptr}(\$0)$$

$$\left\{\begin{array}{r}
r_{\mathtt{s}} = r_{\mathtt{savings}}, \\
z_2 r_{\mathtt{this}}.\mathtt{savings} \rightarrow r_{\mathtt{s}}, \neg(r_{\mathtt{s}} = \$0), z_2 r_{\mathtt{s}}.\mathtt{All} \rightarrow \$0, \\
\mathit{Account}(r_{\mathtt{s}}), r_{\mathtt{s}} > r_{\mathtt{this}}.\mathtt{lv}, \\
\exists r\,.\,(z_2 r_{\mathtt{this}}.\mathtt{savings} \rightarrow r, \neg(r = \$0), z_2 r.\mathtt{All} \rightarrow \$0, \mathit{Account}(r), r > r_{\mathtt{this}}.\mathtt{lv}) \\
-\!\!+ \$0.\mathtt{Immutable} \rightarrow \$0, \\
r_{\text{holding}} < r_{\mathtt{s}}, \\
\mathit{CombinedAccount}(r_{\mathtt{this}}), \\
r_{\mathtt{x}} = \mathtt{int}
\end{array}\right\}$$

$$\overset{\textsc{Tr-Duplicate}}{\rightsquigarrow}$$

$$\left\{\begin{array}{r}
r_{\mathtt{s}} = r_{\mathtt{savings}}, \\
z_2 r_{\mathtt{this}}.\mathtt{savings} \rightarrow r_{\mathtt{s}}, \neg(r_{\mathtt{s}} = \$0), z_2 r_{\mathtt{s}}.\mathtt{All} \rightarrow \$0, \\
\mathit{Account}(r_{\mathtt{s}}), r_{\mathtt{s}} > r_{\mathtt{this}}.\mathtt{lv}, \\
\exists r\,.\,(z_2 r_{\mathtt{this}}.\mathtt{savings} \rightarrow r, \neg(r = \$0), z_2 r.\mathtt{All} \rightarrow \$0, \mathit{Account}(r), r > r_{\mathtt{this}}.\mathtt{lv}) \\
-\!\!+ \$0.\mathtt{Immutable} \rightarrow \$0, \\
r_{\text{holding}} < r_{\mathtt{s}}, \\
\mathit{CombinedAccount}(r_{\mathtt{this}}), \\
r_{\mathtt{x}} = \mathtt{int}, \\
\neg(r_{\mathtt{s}} = \$0), \mathit{Account}(r_{\mathtt{s}})
\end{array}\right\}$$

$$\mathtt{fork} \qquad \Downarrow \mathtt{ptr}(\$0)$$

$$\left\{\begin{array}{r} r_{\mathtt{s}} = r_{\mathtt{savings}}, \\ z_2 r_{\mathtt{this}}.\mathtt{savings} \rightarrow r_{\mathtt{s}}, \neg(r_{\mathtt{s}} = \$0), z_2 r_{\mathtt{s}}.\mathtt{All} \rightarrow \$0, \\ \mathit{Account}(r_{\mathtt{s}}), r_{\mathtt{s}} > r_{\mathtt{this}}.\mathtt{lv}, \\ \exists r\,.\,(z_2 r_{\mathtt{this}}.\mathtt{savings} \rightarrow r, \neg(r = \$0), z_2 r.\mathtt{All} \rightarrow \$0, \mathit{Account}(r), r > r_{\mathtt{this}}.\mathtt{lv}) \\ \multimap \$0.\mathtt{Immutable} \rightarrow \$0, \\ r_{\text{holding}} < r_{\mathtt{s}}, \\ \mathit{CombinedAccount}(r_{\mathtt{this}}) \end{array}\right\}$$

```
}
```

$$\left\{\begin{array}{r} z_2 r_{\mathtt{this}}.\mathtt{savings} \rightarrow r_{\mathtt{savings}}, \neg(r_{\mathtt{savings}} = \$0), z_2 r_{\mathtt{savings}}.\mathtt{All} \rightarrow \$0, \\ \mathit{Account}(r_{\mathtt{savings}}), r_{\mathtt{savings}} > r_{\mathtt{this}}.\mathtt{lv}, \\ \exists r\,.\,(z_2 r_{\mathtt{this}}.\mathtt{savings} \rightarrow r, \neg(r = \$0), z_2 r.\mathtt{All} \rightarrow \$0, \mathit{Account}(r), r > r_{\mathtt{this}}.\mathtt{lv}) \\ \multimap \$0.\mathtt{Immutable} \rightarrow \$0, \\ r_{\text{holding}} < r_{\mathtt{savings}}, \\ \mathit{CombinedAccount}(r_{\mathtt{this}}) \end{array}\right\}$$

`null` $\Downarrow \mathtt{ptr}(\$0)$

{...}

`return _ ;`

$$\left\{\begin{array}{r} z_2 r_{\mathtt{this}}.\mathtt{savings} \rightarrow r_{\mathtt{savings}}, \neg(r_{\mathtt{savings}} = \$0), z_2 r_{\mathtt{savings}}.\mathtt{All} \rightarrow \$0, \\ \mathit{Account}(r_{\mathtt{savings}}), r_{\mathtt{savings}} > r_{\mathtt{this}}.\mathtt{lv}, \\ \exists r\,.\,(z_2 r_{\mathtt{this}}.\mathtt{savings} \rightarrow r, \neg(r = \$0), z_2 r.\mathtt{All} \rightarrow \$0, \mathit{Account}(r), r > r_{\mathtt{this}}.\mathtt{lv}) \\ \multimap \$0.\mathtt{Immutable} \rightarrow \$0, \\ r_{\text{holding}} < r_{\mathtt{savings}}, \\ \mathit{CombinedAccount}(r_{\mathtt{this}}) \\ r_{\text{ret}} = \$0 \end{array}\right\}$$

```
}
```

Additional permission checking for the thread spawn: `fork (s)` . . .

`fork (s) {`

$$\left\{\begin{array}{c} \neg(r_{\mathtt{s}} = \$0), \mathit{Account}(r_{\mathtt{s}}), r_{\mathtt{x}} = \mathtt{int}, \\ r_{\text{thisThread}} < r_{\mathtt{s}}, r_{\text{thisThread}}.\mathtt{Prot} \rightarrow \$0 \end{array}\right\}$$

`s` $\Downarrow \mathtt{ptr}(r_{\mathtt{s}})$

{...}

`acquire _`

$$\left\{\begin{array}{r}\neg(r_{\mathtt{s}} = \$0), \mathit{Account}(r_{\mathtt{s}}), r_{\mathtt{x}} = \mathtt{int},\\ r_{\text{thisThread}} < r_{\mathtt{s}}, r_{\text{thisThread}}.\mathtt{Prot} \rightarrow \$0,\\ r_{\mathtt{s}}.\mathtt{Prot} \rightarrow \$0\end{array}\right\}$$

s $\Downarrow \mathtt{ptr}(r_{\mathtt{s}})$

{...}

x $\Downarrow \mathtt{ptr}(\mathtt{int})$

{...}

_ .deposit(_) $\Downarrow \mathtt{ptr}(\$0)$

$$\left\{\begin{array}{r}\neg(r_{\mathtt{s}} = \$0), \mathit{Account}(r_{\mathtt{s}}),\\ r_{\text{thisThread}} < r_{\mathtt{s}}, r_{\text{thisThread}}.\mathtt{Prot} \rightarrow \$0,\\ r_{\mathtt{s}}.\mathtt{Prot} \rightarrow \$0\end{array}\right\}$$

release

$$\left\{\begin{array}{r}\neg(r_{\mathtt{s}} = \$0), \mathit{Account}(r_{\mathtt{s}}),\\ r_{\text{thisThread}} < r_{\mathtt{s}}, r_{\text{thisThread}}.\mathtt{Prot} \rightarrow \$0\end{array}\right\}$$

}

7.3 Summary

This chapter shows some implementation issues about our permission system. Our prototype tool is based on the current permission analysis in Fluid project by extending the lattice and transfer functions. We provide the detailed permission checking for some methods that mentioned in previous chapters.

Chapter 8

Discussion

8.1 Ownership & Permission Nesting

Ownership is a recognized alias control technique. With ownership, each object has another object as its *owner*. The root of the ownership hierarchy is often called "world." Ownership types provide a statically enforceable way of specifying object encapsulation. Clarke and others propose an *owners-as-dominators* model: any reference to an object must pass that object's owner [46, 47]. This encapsulation property prevents any access to an object from objects outside its owner.

Boyapati and others varies the previous owners-as-dominators model into a new one: *owners-as-locks*, such that any reference to an object must guarantee that its owner (or the root-owner) is held by the current thread.

As mentioned before, the ownership basically indicates an encapsulation relation. This is very similar as the nesting relation in our permission system. We can nest some permission of any object into its owner permission. For the owners-as-dominators model, this is expressed as:

$$r.\texttt{All} \prec r_{\text{owner}}.\texttt{Owned}$$

where the r_{owner} is the owner of the object r and the `Owned` data group [1] holds the state of objects owned by the current object.

For the Boyapati' owners-as-locks model, this ownership protection relation looks

[1]Similar as the `All`, the `Owned` is a special data group inherited from Object.

like:

$$r.\texttt{All} \prec r_{\text{rootowner}}.\texttt{Prot}$$

where the $r_{\text{rootowner}}$ is the root-owner of the object r. The `Prot` data group has the same meaning used as usual and the unit permission to r.`All` is only granted inside of the synchronized block holding its root-owner object, where the r.`All`, by default, is considered as the state of object r. This is done by nesting all the unit permissions of fields (data groups) into the `All` data group. Figure 8.1 shows an example. The ownership tree is on the left, such that the o_1 is the root. The unit permission for o_1.`Prot` is represented as a box that includes the unit permission for o_2.`All`, o_3.`All`, o_4.`All` as well as o_1.`All` since it is self owned.

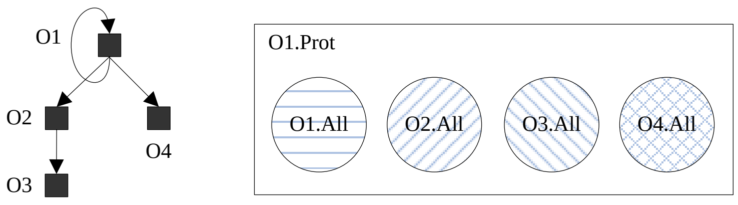


Figure 8.1: Permission representation for ownership

In the owners-as-locks model, the whole state of an object has to be protected by a single lock (its root-owner). This is a little bit restrictive. Our permission system, however, is more flexible and able to have different guards that protect different parts of an object's state. For a code segment in Figure 8.2, class `C` has several fields such that the `f1` and `f2` are "`guarded_by`" different guards.

From Chapter 4, annotations will be translated into permissions. Thus, we get three different field invariants in permission:

$$\begin{aligned}
\Gamma_{\texttt{f1}} &= r_{\texttt{this}}.\texttt{f1} \rightarrow \texttt{int} \prec r_{\texttt{g1}}.\texttt{Prot} \\
\Gamma_{\texttt{f2}} &= r_{\texttt{this}}.\texttt{f2} \rightarrow \texttt{int} \prec r_{\texttt{g2}}.\texttt{Prot} \\
\Gamma_{\texttt{f3}} &= r_{\texttt{this}}.\texttt{f3} \rightarrow \texttt{int} \prec r_{\texttt{this}}.\texttt{All}
\end{aligned}$$

This permission representation shows that the different fields in this class has different

```
class C<g1, g2> {
  int f1 guarded_by g1;
  int f2 guarded_by g2;
  int f3;
  ......
}
```

Figure 8.2: Code segment of a class definition.

nester permission. See Figure 8.3, assuming `o1` is an instance of this class and we use a disk to represent its state, then the `g1.Prot`, `g2.Prot` and `o1.All` separately "owns" different parts of `o1`'s state. Therefore, we split object's state based on fields and they

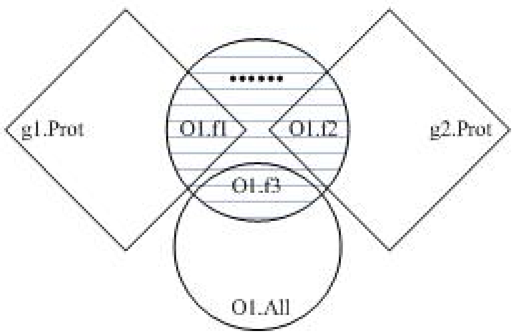


Figure 8.3: Multiple nesters for object state based on fields

are associated different protectors (nesters).

In the current permission system, if a field is "`guarded_by` *guard*", any access to this field should be in the protection of the *guard*, both reads and writes. In this case, the whole unit permission of the field access is nested into the `Prot` data group of its *guard* object. One of our future directions is to apply a new field annotation "`write_guarded_by` *guard*", such that only the write access to this field is required to protected by the *guard*. This annotation is first introduced by Flanagan and Qadeer in their atomicity paper [16].

The intuitive idea is to only nest fractional permission of the field access into the `Prot` data group of its *guard* object, while remaining the rest into the `All` data group of the receiver object. For example, if a f with a primitive `int` type is "`write_guarded_by` *guard*", then its field invariant is given as:

$$\Gamma_f = (\texttt{1/2}r_{\texttt{this}}.f \rightarrow \texttt{int}) \prec r_{guard}.\texttt{Prot} \wedge (\texttt{1/2}r_{\texttt{this}}.f \rightarrow \texttt{int}) \prec r_{\texttt{this}}.\texttt{All}$$

Assuming the class defined in Figure 8.2 has an additional formal class parameter $g3$ as well as a field $f4$ such that the latter is "`write_guarded_by`" the former, then Figure 8.4 shows another multi-protector graph for the object's state with using the new "`write_guarded_by`".

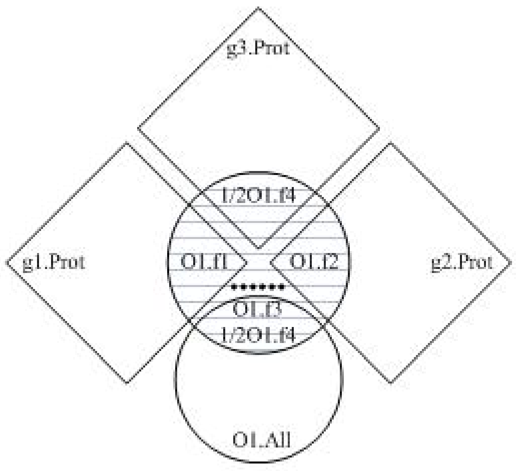


Figure 8.4: Multiple nesters for object state based on fields and protection mechanism.

The annotation "`write_guarded_by`" does help to build some parallel interference patterns, especially the *unique thread write* which only allows a unique thread to write a particular field of an object. In order to achieve race-free for the *unique thread write* pattern among parallel interference, there are two choices.

- The whole unit permission to access that field is granted to a thread, while other parallel threads get nothing. Figure 8.5.(a) demonstrates an example of this case. We use the **1** to show the unit permission indicating the possible write access, while a **0** shows no access at all.

- If this field is annotated as "`write_guarded_by` *guard*", such that a fractional (`1/2`) permission is protected by (nested into) the *guard*, while the rest is staying in the `All` data group of its object. Figure 8.5.(b) shows an example of this case. The **1/2** is used to represent the rest fractional (`1/2`) permission that nested in the `All` of its object. It goes into one thread and is promoted to a whole unit permission (**1/2+1/2**) when entering some particular synchronized block with holding its guard object. Other parallel threads are not able to access this field except in a synchronized block holding its guard, where only read is allowed.

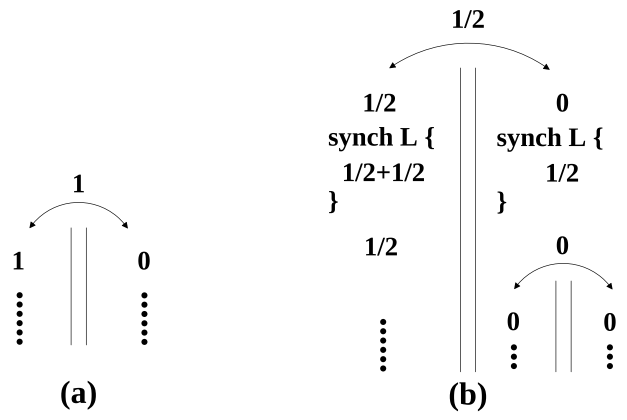


Figure 8.5: Unique thread write patterns.

8.2 Reader-Writer Lock

If multiple threads concurrently execute code that writes to or modifies a resource, then obviously the resource must be protected with a thread synchronization lock to ensure that the resource doesn't get corrupted. However, it is common to have a resource that is occasionally written to but frequently read from. If multiple threads are concurrently reading a resource, using a mutual exclusive lock hurts performance significantly because only one thread at a time is allowed to read from the resource. It's far more efficient to allow all the threads to read the resource simultaneously, and it's fine to do this if all of the threads treat the resource as read-only and do not attempt to write to or modify it.

A reader-writer synchronization lock can and should be used to improve performance and scalability. A reader-writer lock ensures that only one thread can write to a resource at any one time and it allows multiple threads to read from a resource simultaneously as long as no thread is writing at the same time.

We use `synch some` e `do` e to express a reader lock synchronization, while keeping the original one as the writer lock synchronization. Figure 8.9 shows an example using this syntax. With the reader lock synchronization, multiple `get()` calls could be in parallel without blocking.

Our original permission type system allows the synchronization to have some choices.

- If the lock has already been held, we simply remove the two actions for lock acquirement and release (see REENTRANT);
- If the lock hasn't been held and it is higher than the surrounding holding lock, then we acquire the lock as well as updating the surrounding holding lock (see SYNCH).

Based on the two choices mentioned above, there are two additional permission typing rules for the reader lock acquirement in Figure 8.6. The original ones in Figure 5.5 still work for the writer lock acquirement.

In order to acquire a reader lock:

- If the lock has already been held (either reader or writer), we remove the two actions for lock acquirement and release (see REENTRANT-READER);

- If the lock hasn't been held and it is higher than the surrounding holding lock, then we acquire the reader lock as well as updating the surrounding holding lock (see SYNCH-READER). After acquiring a reader lock ρ, a *fractional* permission to access the ρ.Prot is granted. This is similar as before, except the writer lock acquirement will grant a unit permission.

$$\textsc{Reentrant-Reader}\quad \frac{\begin{array}{c}\Delta;\Pi \vdash_{\rho_L} e_1 \Downarrow \texttt{ptr}(\rho_1) \dashv \Delta';\Pi' \\ \Pi' = \xi\rho_1.\texttt{Prot} \to \$0, \Pi'_1 \qquad \Delta';\Pi' \vdash_{\rho_L} e_2 \Downarrow \texttt{ptr}(\rho_2) \dashv \Delta'';\Pi''\end{array}}{\Delta;\Pi \vdash_{\rho_L} \texttt{synch some } e_1 \texttt{ do } e_2 \Downarrow \texttt{ptr}(\rho_2) \dashv \Delta'';\Pi''}$$

$$\textsc{Synch-Reader}\quad \frac{\begin{array}{c}\Delta;\Pi \vdash_{\rho_L} e_1 \Downarrow \texttt{ptr}(\rho_1) \dashv \Delta';\Pi' \\ \Pi' = \rho_L < \rho_1, \Pi'_2 \qquad \Delta';\xi\rho_1.\texttt{Prot} \to \$0, \Pi' \vdash_{\rho_1} e_2 \Downarrow \texttt{ptr}(\rho_2) \dashv \Delta'';\xi\rho_1.\texttt{Prot} \to \$0, \Pi''\end{array}}{\Delta;\Pi \vdash_{\rho_L} \texttt{synch some } e_1 \texttt{ do } e_2 \Downarrow \texttt{ptr}(\rho_2) \dashv \Delta'';\Pi''}$$

Figure 8.6: Permission type rules: Reader Lock Synchronization

Accordingly, we may need to add a "`requires some` *lock*" or "`uses some` *lock*" to indicate the reader *lock* is acquired before or after the method entry. Their permission representation are $zr_{lock}.\texttt{Prot} \to \$0$ and $(r_{\text{holding}} < r_{lock})?(\emptyset) : (zr_{lock}.\texttt{Prot} \to \$0)$ respectively.

8.3 Volatile

In the current Java memory model [37], a write of a *volatile* variable in one thread T_1 followed by a read in a different thread T_2 functions similarly to a monitor in that all writes to state by T_1 before writing the volatile variable are visible to T_2 after it reads the variable. However there is no guarantee that any particular write will be visible by another thread—no blocking, no atomicity with regard to other state changes. The reading thread might read the volatile variable multiple times between writes, or not at all. Thus it seems that no (linear) permissions can be transmitted through volatile variables. However, it is entirely possible to transmit *non-linear* information.

With nesting, it is easy to use linear types in an imperative programming language. In nesting terms, the fact that a nesting has taken place and that one permission is available inside another, can be transmitted non-linearly. Then in some other part of a program where the nester permission is already available, the nested permission can be recovered. The price to paid is that the nesting fact, once established, cannot be negated.

The basic idea is to use nesting to model transfer of permission through volatile field reads and writes. Nesting works invisibly—when a permission Π is nested in a location, Π is "immediately" transferred to any part of the program has (fractional) access to that location. However, nesting cannot be put into effect until the *knowledge* of nesting is received.

The traditional technique ("standard practice") for protecting mutable state is to designate a protecting object for each piece of mutable state (one object may protect many others) and ensure that all accesses to the state occur dynamically only within a synchronization on the protecting object. For example, see class `Traditional` in Figure 8.8 (the auxiliary code is in Figure 8.7); the bodies of the methods `get()` and `inc()` include synchronizations around the *reads* and *writes* of the mutable state respectively. This approach is safe, but may not be very efficient sometimes, especially when the `get()` calls are frequent and we wish to permit them to operate in parallel.

Figure 8.9 also shows how a volatile field can substitute for synchronization. The reading method can simply access the nodes directly using a volatile field read (and then traverse the list without synchronization). The incrementing method must be careful to copy the structure before modifying it. Furthermore, the entire operation is synchronized to ensure that two increments are not carried out in parallel. However, it is legal to interleave the `get()` and `inc()` calls since the `inc()` method never updates any state the `inc()` method can see, except for the volatile field.

```
class Node {
  Node next;
  Node()
  { next = null; }

  Node getNext()
  { next; }

  int count()
  { if ( this == null ) then 0;
    else { 1 + next.count(); } }
```

```
  Node copy()
  { if ( this == null ) then null;
    else new Node( next.copy() ); }

  void add1()
  { this.nap( new Node() ); }

  Node nap( Node n )
  { if ( this == null ) then n;
    else { next = next.nap(n); this; }; }
}
```

Figure 8.7: A Node class definition

```
class Race {
  Node nodes;
  Race() { }
  int get()
  { nodes.count(); }
  void inc()
  { nodes = nodes.add1(); }
}
```

```
class Traditional {
  Node nodes;
  Traditional() { }
  int get()
  { synch this do nodes.count(); }

  void inc()
  { synch this do
      nodes = nodes.add1(); }
}
```

Figure 8.8: Two classes with unprotected and protected field respectively.

The usage of volatile field challenges our permission system since calling the `get()` and `inc()` of a same `UsingVolatile` receiver in parallel is considered safe (race-free). The thread that makes the `get()` calls should at least have some fractional permission to access the `nodes` field, but its parallel thread that calls the `inc()` needs to have a whole unit permission inside of the synchronized block. Even with the new introduced "`write_guarded_by`" annotation, this is still impossible.

```
class ReaderWriter {
Node nodes;
  ReaderWriter() { }
  int get()
  { synch some (this) do
      nodes.count(); }

  void inc()
  { synch this do
      nodes = nodes.add1(); }
}
```

```
class UsingVolatile {
  volatile Node nodes;
  UsingVolatile() { }
  int get()
  { nodes.count(); }

  void inc()
  { synch this do
      nodes = nodes.copy().add1(); }
}
```

Figure 8.9: Two approaches to protect state.

The permission system needs to be extended to have the ability that allows the volatile field to be able to read and synchronized write in parallel. And this ability only applies to the volatile fields. A promising way is to make some permission "shared", such that they are duplicated at the thread spawn and go to different threads simultaneously. We may add a permission syntax $\Omega^n(\Pi)$ to show the shared permission based on Π where the n is the depth of the sharing. Figure 8.10 demonstrates a process that how does a permission become to shared at a thread spawn. Any permission could be shared, such

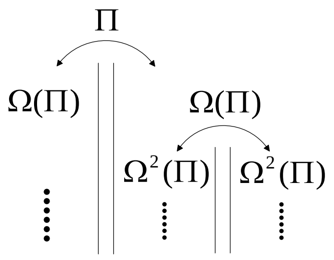


Figure 8.10: Fractional Permission.

that

Tr-Shared1 $\qquad$ Tr-Shared1

$$\Pi \equiv \Omega^0(\Pi) \qquad \Omega^n(\Pi) \rightsquigarrow \Omega^{n+1}(\Pi), \Omega^{n+1}(\Pi)$$

For any read and write to a regular field, the permission checking keeps unchanged. But for the read and write to a volatile field, there should be some new rules :

$$\textsc{ReadV}\quad \frac{\Delta; \Pi \vdash_{\rho_L} e \Downarrow \mathtt{ptr}(\rho) \dashv \Delta'; \Pi' \\ \Pi' = \Omega^n(\xi \exists r\,.\,(\rho.f \ \rightarrow r, \Gamma)), \Pi'' \qquad \rho \neq r \qquad r' \notin \Delta \qquad f \in Fv}{\Delta; \Pi \vdash_{\rho_L} e.f \ \Downarrow \mathtt{ptr}(r') \dashv \{r'\} \cup \Delta'; [r \mapsto r']\Gamma, \Pi'}$$

$$\textsc{WriteV}\quad \frac{\Delta; \Pi \vdash_{\vdash} e_1 \Downarrow \mathtt{ptr}(\rho_1) \dashv \Delta'; \Pi'_{\rho_L} e_2 \Downarrow \mathtt{ptr}(\rho_2) \dashv \Delta''; \Pi'' \\ \Psi = \exists r\,.\,(\rho_1.f \ \rightarrow r, \Gamma) \qquad \rho_1 \neq r \\ \Pi'' = \Omega^{n_1}(q_1\Psi), ..., \Omega^{n_m}(q_m\Psi), [r \mapsto \rho_2]\Gamma, \Pi''' \qquad q_1 + ... + q_m = 1 \qquad f \in Fv}{\Delta; \Pi \vdash_{\rho_L} e_1.f\, \mathtt{=} e_2 \Downarrow \mathtt{ptr}(\rho_2) \dashv \Delta''; \Pi''}$$

Figure 8.11: Permission type rules: Read and Write volatile fields

They permit any depth of sharing $\Omega^n(...)$, including none (n = 0); WriteV allows several different shared permissions to be combined. The volatile field must have existential type so that its actual value can be changed without requiring a change in the shared permission. The (non-linear) information Γ existentially closed along with the shared permission is duplicated when reading and overwritten when writing.

8.4 Eliminate Reentrant Synchronization

Reentrant locks present the simplest form of unnecessary synchronization [38]. A lock is reentrant if a synchronization on it is already inside of a synchronized block holding itself. It is safe to remove the inner synchronization since the outer one has already guaranteed that no other thread could synchronize on the lock at the same time. The reentrant lock detection is intuitively supported by our permission system. The permission checking rule Reentrant copes with this situation. Assuming there is a synchronization on lock ρ, if the input permission includes the unit permission $\rho.\mathtt{Prot} \rightarrow \$0$, then this ρ is a reentrant lock and the current synchronization could be eliminated.

Chapter 9
Conclusion

The concurrent programming is becoming a mainstream programming practice, since it greatly increases the performance not only on multi-core devices, but also on single-processor machines. However, it also brings non-determinism and requires programmers to have specific knowledge and discipline.

Multithreaded programs synchronize operations on shared mutable data to ensure that the operations execute atomically. Failure to correctly synchronize such operations can lead to *data races* or *deadlocks*. Because of the nondeterministic interference among parallel threads, traditional analysis techniques that built for sequential programs cannot be easily extended to multithreaded programs.

Boyland [5, 7] introduces a fractional permission system which uses fractions to distinguish write from read. With fractions, it's able to show non-interference between two concurrent computations. Nesting (an extension of adoption [48]) is used to model state encapsulation and protection mechanism. With nesting, one permission can be "nested" inside another permission.

This thesis shows how to extend the fractional permissions to annotated programs using unstructured parallelism and synchronizations. Annotations include uniqueness, nullity, immutability, method effects and lock protected state etc. Fields are given annotations indicating how they can be accessed, while method annotations show the requirements and effects of the method calls. All these high-level annotations will be translated into a unified low-level permission representation.

Permissions are granted to permit certain operations in programs. For example, a read (fractional) permission only allows the read access, while a write (unit) permission allows both read and write accesses. Each field has its own protection mechanism, such that any access to that field must satisfy the requirement. This protection relation is built by permission nesting, since the nested permission is always protected by the nester permission. To avoid deadlocks, we assign partial order among lock objects and force the lock acquirements to be an ascending order.

In this thesis, a simple object-oriented language is defined to demonstrate the usage of annotations and how they can be translated into permissions. We show the consistency property between static permissions and runtime state. Combining the operational semantics, permission typing rules with the consistency property, we establish the progress and preservation theorem for this permission system. Well permission-typed programs in our system are guaranteed to be free of data races and deadlocks.

Possible future directions for this work include:

- Applying the prototype tool to real Java language with cases studies;
- Adding "`write_guarded_by`" annotation which indicates only the write access needs to be protected by its guard;
- Introducing into the reader-writer lock;
- Operational semantics and permission rules for volatile fields;
- Eliminating unnecessary synchronizations.

Bibliography

[1] Roscoe, A. W., *The Theory and Practice of Concurrency*, Prentice Hall. ISBN 0-13-674409-5, 1997.

[2] Joe Armstrong, *Programming Erlang: Software for a Concurrent World*, First Edition, Pragmatic Bookshelf, July 2007.

[3] Herbert Schildt, *The Art of C++*, McGraw-Hill/Osborne, 2004; ISBN: 0072255129.

[4] http://en.wikipedia.org/wiki/Dining_philosophers_problem.

[5] John Boyland, *Checking Interference with Fractional Permissions*, In R. Cousot, editor, Static Analysis: 10th International Symposium, volume 2694 of Lecture Notes in Computer Science, pages 55–72, Berlin, Heidelberg, New York, 2003. Springer.

[6] John Boyland, *Why We Should Not Add `readonly` to Java, yet*, 7th ECOOP Workshop on Formal Techniques for Java-like Programs, 2005

[7] John Boyland and William Retert, *Connecting Effects and Uniqueness with Adoption*, Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Long Beach, California, USA, January 12 - 14, 2005

[8] John Boyland, *Semantics of Fractional Permissions with Nesting*, submitted to POPL'08.

[9] Chandrasekhar Boyapati and Martin Rinard, *A parameterized type system for race-free Java programs*, Proceedings of the 16th ACM SIGPLAN conference on

Object oriented programming, systems, languages, and applications, Tampa Bay, FL, Oct. 2001

[10] Chandrasekhar Boyapati, Robert Lee and Martin Rinard, *Ownership Types for Safe Programming: Preventing Data Races and Deadlocks* Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, November 04-08, 2002, Seattle, Washington, USA

[11] Chandrasekhar Boyapati, *SafeJava: A Unified Type System for Safe Programming*, Phd thesis, MIT, 2004.

[12] David F. Bacon, Robert E. Strom and Ashis Tarafdar, *Guava: a dialect of Java without data races*, Proceedings of the 15th Annual Conference on Object-Oriented Programming Systems, Languages and Applications, Portland, OR, Oct. 2000.

[13] Cormac Flanagan and Stephen N. Freund, *Types-based Race Detection for Java*, Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation, p.219-232, June 18-21, 2000, Vancouver, British Columbia, Canada

[14] Cormac Flanagan and Stephen N. Freund, *Detecting Race Conditions in Large Programs*, Proceedings of the ACM Workshop on Program Analysis for Software Tools and Engineering, 2001, pp. 90-96

[15] Cormac Flanagan and Martin Abadi, *Types for Safe Locking*, Proceedings of the 8th European Symposium on Programming Languages and Systems, p.91-108, March 22-28, 1999

[16] Cormac Flanagan and Shaz Qadeer, *A Type and Effect System for Atomicity*, Proceedings of the ACM SIGPLAN 2003.

[17] M. Abadi and C. Flanagan and S. N. Freund, *Types for Safe Locking: Static Race Detection for Java*, ACM Transactions on Programming Languages and Systems, page 207-255, March, 2006.

[18] Martin Abadi, Cormac Flanagan, Stephen N. Freund, *Types for Safe Locking: Static Race Detection for Java*, ACM Transactions on Programming Languages and Systems (TOPLAS), Volume 28 Issue 2, Pages: 207-255, ACM Press New York, NY, USA

[19] Cormac Flanagan and Stephen N. Freund, *Type Inference Against Races*, Static Analysis Symposium, 2004, pp. 116-132.

[20] Naoki Kobayashi, *Type Systems for Concurrent Programs*, Proceedings of UNU/IIST 10th Anniversary Colloquium, LNCS 2757, pp.439-453

[21] Naoki Kobayashi, *A Type System for Lock-free Processes*, Information and Computation, vol. 177, pp.122-159, 2002

[22] Naoki Kobayashi, *A New Type System for Deadlock-Free Processes*, Proceedings of CONCUR 2006, Springer LNCS 4137, pp.233-247, 2006

[23] Atsushi Igarashi and Naoki Kobayashi, *A generic type system for the Pi-calculus*, Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (2001), pp. pp.128-141.

[24] Robin Milner, *Communicating and Mobile Systems: the Pi-Calculus* Cambridge University Press, 1999

[25] R. Milner, J. Parrow and D. Walker, *A Calculus of Mobile Processes, Part I/II*, Information and Computation, 1992

[26] Stephen Brookes, *A Semantics for Concurrent Separation Logic*, In CONCUR'04: 15th International Conference on Concurrency Theory, volume 3170 of Lecture Notes in Computer Science, pages 16–34, London, August 2004

[27] Aaron Greenhouse and William L. Scherlis, *Assuring and Evolving Concurrent Programs: Annotations and Policy*, Proceedings of the 24th International Conference on Software Engineering, Pages: 453-463, 2002

[28] Aaron Greenhouse, *A Programmer-Oriented Approach to Safe Concurrency*, Ph.D. Thesis. Carnegie Mellon University School of Computer Science, Pittsburgh, PA. May 2003. Technical Report CMU-CS-03-135.

[29] D. L. Detlefs, K. R. M. Leino, G. Nelson, and J. B. Saxe, *Extended static checking*, Research Report 159, Compaq SRC, Dec. 1998.

[30] K. R. M. Leino, G. Nelson, and J. B. Saxe, *ESC/Java user's manual*, Technical Note 2000-002, Compaq SRC, Oct. 2000.

[31] R. Rugina and M. C. Rinard, *Pointer Analysis for Structured Parallel Programs*, ACM Transactions on Programming Languages and Systems, 25(1):70C116, January 2003.

[32] Richard Bornat, Cristiano Calcagno, Peter O'Hearn and Matthew Parkinson, *Permission Accounting in Separation Logic* Proceedings of POPL, pages 259C270. ACM Press, 2005.

[33] L. O. Andersen, *Program Analysis and Specialization for the C Programming Language*, PhD thesis, University of Copenhagen, 1994.

[34] B. Steensgaard, *Points-to Analysis in Almost Linear Time* Proceedings of the 23rd Annual ACM Symposium on the Principles of Programming Lanugages, pages 32-41. ACM Press, January 1996

[35] B. Ryder, W. Landi, P. Stocks, S. Zhang and R. Altucher, *A Schema for Interprocedural Modification Side-effect Analysis with Pointer Aliasing*, ACM Trans. Prog. Lang. Syst., 2001.

[36] Dawson Engler and Ken Ashcraft, *RacerX: Effective, Static Detection of Race Conditions and Deadlocks*, Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles, 2003

[37] Jeremy Manson, William Pugh and Sarita V. Adve, *The Java Memory Model*, Proceedings of the 32nd ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, pages 378-391. ACM Press, 2005

[38] Jonathan Aldrich, Craig Chambers, Emin Gun Sirer, and Susan Eggers. *Static Analyses for Eliminating Unnecessary Synchronization from Java Programs*, Proceedings of Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '99), Denver, Colorado, 1-5 November 1999.

[39] David G. Clarke and John M. Potter and James Noble, *Ownership Types for Flexible Alias Protection*, Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, Vancouver, British Columbia, Canada, 1998.

[40] David Clarke, *Object Ownership & Containment*, PhD thesis, University of New South Wales, Australia.

[41] Dave Clarke and Sophia Drossopoulou, *Ownership, Encapsulation and the Disjointness of Types and Effects*, Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA'02).

[42] Matthew Flatt, Shriram Krishnamurthi and Matthias Felleisen, *Classes and mixins*, Conference Record of the Twenty-fifth Annual ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages, pages 171C183. ACMPress, January 1998.

[43] K. Rustan M. Leino, *Data Groups: Specifying the Modification of Extended State*, Proceedings of the 1998 ACM SIGPLAN Conference on ObjectOriented Programming, Systems, Languages, and Applications (OOPSLA '98), volume 33, number 10 in SIGPLAN Notices, pages 144–153. ACM, October 1998.

[44] Manuel Fähndrich and K. Rustan M. Leino, *Declaring and Checking Non-Null Types in an Object-Oriented Language*, Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications, 2003.

[45] http://www.fluid.cs.cmu.edu

[46] Dave Clarke, *Object Ownership and Containment*, PhD thesis, University of New South Wales, 2001.

[47] Dave Clarke, John Potter and James Noble, *Ownership Types for Flexible Alias*

Protection, OOPSLA '98 Conference Proceedings, volume 33(10) of ACM SIGPLAN Notices, pages 48–64. ACM, October 1998.

[48] M. Fähndrich and R. DeLine, *Adoption and Focus: Practical Linear Types for Imperative Programming*, Proceedings of the SIGPLAN'02 Conference on Programming Language Design and Implementation, 2002.

CURRICULUM VITAE

Yang Zhao

Place of birth: Nanjing, China

Education

- **University of Wisconsin-Milwaukee** Milwaukee, WI, USA
  Ph.D., Computer Science, 2007 (expected)
  Minor area in Mathematical Science
- **Nanjing University** Nanjing, China
  M.S., Computer Science, 2003
- **Nanjing University of Posts & Telecommunications** Nanjing, China
  B.S., Computer Science, 2000
  Minor in Management Engineering

Dissertation:Concurrency Analysis Based on Fractional Permission System

Major Professor: ______________________ Date: ____________