7-10-2012

Open Effects

Yuheng Long
Iowa State University, csgzlong@iastate.edu

Open Effects

Abstract
Open world assumption is an important design decision for modern object-oriented languages --- it allows extensibility in program design. Type-and-effect systems are also valuable for these languages, e.g. they can help reason about concurrent OO programs. Open world assumption, however, makes the design of a type-and-effect system challenging for an OO language. Main problem is with the computation of the effects of a dynamically dispatched method call, because all possible dynamic types are not known in advance. Previous research has proposed asking programmers for effect annotations that give an upper bound on the effects of a dynamically dispatched method call. This work describes an easier approach for programmers, albeit with some runtime overhead compared to previous work, which is based on the novel notion of open effects, effects that are optimistically assumed to satisfy the effect-based property of interest. We describe a sound type-and-effect system with open effects which has two parts: a static part that takes effects of dynamically dispatched calls with certain special references as an open effect; and a dynamic part that manages dynamic effects as these special references change and verifies that the optimistic assumptions about open effects hold. This system is implemented in the OpenJDK compiler and its utility is tested by applying it to verify non(interference) of concurrent tasks.

Keywords
type-and-effect, open effects, optimistic concurrency

Disciplines
Programming Languages and Compilers

Open Effects

Yuheng Long and Hridesh Rajan
TR #12-02
Initial Submission: July 10, 2012

Abstract: This is the Technical Report version of the 2013 POPL submission by the same title. It includes the POPL version verbatim, followed by an appendix containing omitted contents and proofs.

Keywords: type-and-effect, open effects, optimistic concurrency

CR Categories:
D.1.3 [*Concurrent Programming*] Parallel programming
D.1.5 [*Programming Techniques*] Object-Oriented Programming
D.2.2 [*Design Tools and Techniques*] Modules and interfaces,Object-oriented design methods
D.2.3 [*Coding Tools and Techniques*] Object-Oriented Programming
D.2.4 [*Software/Program Verification*] Validation
D.2.10 [*Software Engineering*] Design
D.3.1 [*Formal Definitions and Theory*] Semantics,Syntax
D.3.1 [*Language Classifications*] Concurrent, distributed, and parallel languages,Object-oriented languages
D.3.3 [*Programming Languages*] Concurrent programming structures, Language Constructs and Features - Control structures
D.3.4 [*Processors*] Compilers

Department of Computer Science
226 Atanasoff Hall
Iowa State University
Ames, Iowa 50011-1041, USA

Contents

Open Effects

Yuheng Long and Hridesh Rajan
Dept. of Computer Science, Iowa State University
{csgzlong,hridesh}@iastate.edu

Abstract

We present an optimistic effect system for enabling safe concurrency in modern object-oriented languages with an open world assumption. New to our effect system is the notion of *open effects*. An *open effect* is a placeholder effect. It is produced by method calls when the dynamic type of the receiver object is unknown. An open effect is assumed to be blank (i.e., noninterfering effect) statically but verified to be truly so when the dynamic type of the receiver is known. An open effect-based analysis has several benefits. It is modular and so it allows analysis of partial programs and libraries. It is more precise than a comparable static analysis. It also has a small annotation overhead, and does not require specification on super type methods to restrict overriding in subclasses. We have formalized our analysis and proven that it is sound and that it enables deterministic semantics. We have also extended the OpenJDK Java compiler with support for open effects and tested its effectiveness on several reusable library classes where it shows only about 0.13-7.65% overhead and good speedup.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory—Semantics

General Terms Languages, Theory, Verification

Keywords type-and-effect, open effects, optimistic concurrency

1. Introduction

Both static [2, 5, 6, 8, 17, 30, 33, 40] and dynamic [7, 16, 18, 38] analyses have been proposed to help programmers write correct concurrent software. In essence, these techniques compute sets of computational effects [**? ? ?**] of concurrent tasks to determine whether these tasks interfere with each other and thus could lead to unexpected program behavior.

1.1 The Problems and Their Importance

For soundness, such static analysis for an object-oriented (OO) language must conservatively handle features such as dynamic dispatch [17, pp.222]. Consider a method call, if the exact runtime type of the call's receiver is not known statically, then a static analysis has two options. First option is to compute the sets of effects, e.g. {reads field f}, produced by all overriding implementations of that method and take the set of potential effects of that call to be the union of these sets. The second option is to require specifying a method's implementation in the supertype and use that specification to compute an upper bound on the potential effects produced by all overriding implementations of that method.

To illustrate consider the class `ArrayList` in Figure 1, inspired from its namesake in the Java Development Kit (JDK). This class provides an operation `applyall` that iterates over each element in the list and calls method `run` with argument `c` of type `Command` as the receiver object. Two separate applications, PROGRAM and PROGRAM' (among others) use this `ArrayList` class. Each application provides separate and distinct implementation of the command interface. PROGRAM computes a prefix sum, which, as implemented, is not a commutative operation [34]. PROGRAM' computes a 32-bit hash of array elements, which is an independent operation for each element in the array list.

Now consider a concurrent implementation of the method `applyall` in `ArrayList`. If the effect of each iteration of `run` on line 6 does not interfere with any other iteration, then this method may be safely parallelized. Unfortunately, computing the set of effects produced by all implementations of method `run` may not be feasible. In fact, many such overriding implementations, such as those in PROGRAM and PROGRAM' may not even be available during compilation of the library class `ArrayList`.

The second option is to ask programmers to annotate the method `run` such that the effect annotations provide an upper bound on the potential effects produced by all overriding implementations of `run`. Computing such upper bound is difficult primarily due to the variety of, and often unanticipated, usage of library classes such as `ArrayList`. Even if we are able to anticipate all such usage and compute an upper bound, such a bound may be overly conservative. For example, based on the code of the method `run` in PROGRAM, one may conclude that parallelization of the `applyall` method would be unsafe. Whereas, in reality there may be several subclasses of `Command`, such as class `Hash` in PROGRAM', for which `applyall` method can be safely parallelized.

Dynamic analyses can help; however, they provide incomplete, post-deployment detection of concurrency-related defects, whereas we seek preventative detection and defect avoidance [24, pp. 6:2]. For example, we would like to avoid unsafe parallelization of `applyall` in PROGRAM instead of detecting an unsafe trace. Also, majority of current proposals for sound, software-based dynamic analysis cause a major slowdown in programs (see [18, Table 1]), e.g., STM-based solutions are reported to have 2X overhead [39], Goldilocks that requires a custom virtual machine (VM) is reported to have 2X overhead [16] and even higher for production VM [18]. Most of this overhead is due to conflict detection and state buffering (for roll back) mechanisms.

To summarize, OO languages e.g., Java, C#, allow separate compilation and testing of libraries and frameworks that is not supported by static, whole program analyses; asking a developer to annotate methods with effects such that these annotations give an upper bound on effects of all overriding methods could be challenging; and purely dynamic analysis could be expensive [18].

```
1 class ArrayList {
2   int[] elements;
3   int size;
4   void applyall(Command c) {
5     for(int i=0; i<size; i++)
6       c.run(i, elements[i]);
7   }
8 }
```

```
9 class Command {
10   void run(int index, int o) { }
11 }
```

LIBRARY

```
12 class Prefix extends Command {
13   int sum = 0;
14   int[] eles;

16   void run(int index, int o){
17     int i = o;
18     sum += i; //conflicts on sum
19     eles[index] = sum; //! commutative
20   }
21 }
```

PROGRAM

```
22 class Hash extends Command {
23   int[] data;
24   void run(int index, int o){
25     int key = o;
26     key = ~key + (key << 15);
27     ... // Hash computation
28     data[index] = key; //writes to different slot
29   }
30 }
```

PROGRAM'

Figure 1. A library class ArrayList and two separate applications that make use of it. The method of interest is `applyall`.

1.2 Contributions to the State-of-the-Art: *Open* Effects

A promising idea is for the programmer to optimistically assert that method calls, with certain special references as receivers, will produce concurrency-safe effects, and the compiler to trust the programmer statically and generate parallel code, but also emit code to verify programmer's assertion at runtime. We present a new optimistic effect analysis that takes this idea and blends static effect analysis with dynamic effect verification, producing an analysis that has many of the advantages of both static and dynamic analyses, but suffers from none of the limitations described above.

Our effect system has two kinds of effects: *open* and *concrete* effects. An open effect is produced by a method call, whose receiver's dynamic type is unknown, but its static type is qualified with an annotation `@open`. An open effect is assumed to be blank (i.e., noninterfering) statically, but it is filled in at runtime. Concrete effects include reads and writes to memory regions [**?**].

Like static analyses, we compute effects at compile-time. However, unlike static approaches that make conservative approximations when the exact dynamic type is unavailable, we use placeholder *open* effects. Like dynamic approaches, a parallelization technique based on our analysis may treat each parallelization opportunity as optimistically parallel, if *concrete* effects of parallel tasks do not interfere. However, unlike dynamic approaches that detect conflicts after-the-fact, we require verifying that *open* effects are noninterfering prior to forking off parallel tasks.

```
1 class ArrayList {
2   int[] elements;
3   int size;
4   void applyall(@open Command c) {
5     for(int i:size)
6       c.run(i, elements[i]);  //! produces open effect
7   }
8 }
9 class Command {
10   void run(int index, int o){ }
11 }
```

LIBRARY

Figure 2. `ArrayList` with an *Open* Parameter. Thus, method `applyall` has *open* effects. Applications remain the same.

To illustrate, imagine that the programmer optimistically marked the argument `c`'s type in the class `ArrayList` as *open* as on line 4 in Figure 2. The static part of our analysis would then trust the programmer by taking the effect of a method call on this reference as an *open effect*, *i.e. an effect that could be extended at runtime but is blank statically*. So the effect of method call `c.run(..)` on line 6 would be taken as an *open effect*, because the dynamic type of `c` is unknown. Thus, the effect of an iteration of the for loop on line 5 would be reading the *ith* element of the array `elements` and an *open effect*. An *open effect* is treated as a blank effect statically, so an iteration of this for loop would be treated (trustingly) as parallel since it is independent of other iterations.

The dynamic part of our analysis fills in, or *concretizes*, open effects when references marked with `@open` annotations, such as `c`, are assigned. Then, if the concretized (previously open) effects of the method call `c.run(..)` do not interfere, the for loop on line 5 could be run in parallel, else it must be run sequentially.

```
12 class PROGRAM {
13   void main() {
14     ArrayList al = new ArrayList();
15     //add elements to the arraylist al
16     Command p = new Prefix();
17     al.applyall(p);
18   }
19 }
```

Figure 3. An example use of the ArrayList library.

On a different day, the developer of PROGRAM imports the class `ArrayList`. At runtime, PROGRAM creates an instance `al` of `ArrayList`, and passes an instance `p` of class `Prefix` on line 17 in Figure 3 as argument `c`. This assignment to the argument `c` concretizes the effect of an iteration of the for loop, because the effect of this for loop contains an *open* effect (method call effect on an unknown reference `c`). Note that the receiver object `c` is now an alias of the instance `p` of class `Prefix`. So, the extended effect (the original effect union with the concrete effect of the method `run` of the class `Prefix`) of the loop iteration is now reading and writing to the field `sum` of instance `Prefix` and writing to different slot of an array. As a result, iterations of the for loop now has a loop carried dependence (on the field `sum`, line 18 in Figure 1). Thus, the for loop on line 5 is run sequentially when method `applyall` is called on line 17 in PROGRAM in Figure 3.

On yet another day, developer of PROGRAM' imports the `ArrayList` class. PROGRAM' also instantiates `ArrayList`, but passes an instance of class `Hash` in Figure 1 as argument `c`,

which concretizes the effects of method call `c.run(..)` on line 6. These concrete effects are writing to different slot of an array. The effect of an iteration of the for loop is similarly enlarged. As a result, iterations of the loop are still independent. Thus, the same for loop on line 5 is run in parallel in PROGRAM'.

Thus, our optimistic effect analysis can help expose safe concurrency in this case whereas purely static analysis, analyzing only this library and its dependencies, would conservatively label the for loop on line 5 as a sequential loop. Alternatively, such static analysis could also ask users for effect annotations that will specify an upper bound on effects for all subclasses of the `Command` class [6, 33]. A benefit of such analysis would be that it will not incur any dynamic overhead, whereas a drawback is that writing effect annotations by hand, when only partial code is available, can be difficult. Our analysis also has following benefits.

- *It is modular* and so it allows analysis of libraries and frameworks, which is important for software reuse and maintenance. Here "modular" means that the analysis can be done using only the code in question and the interface of the static types used in the code. For example, static analysis of `ArrayList` relies only on code for `ArrayList` and the interface of the `Command` classes, but not necessarily on its implementation. This would be essential for analyzing `ArrayList` without requiring PROGRAM or PROGRAM' to also be present. This benefit of *OpenEffectJ* is critical for libraries, which are analyzed and compiled once, but reused often. Previous work show that most concurrency is exposed via libraries [6, 25].
- It *does not require annotating methods* in a supertype to specify upper bounds on the effect of all subtypes, e.g. the `run` method in type `Command` (Figure 2, line 10).
- Parallelization based on our effect analysis will *never require rollback*, rather an operation is attempted in parallel if and only if the *open* effect assertions hold.
- For our use cases, it had a *small annotation overhead*, e.g. one annotation was needed in the `ArrayList`.
- *User annotations cannot break soundness*, in the worst case they can create extra overhead (and only when effects are unknown statically).
- It is *more precise than a comparable static analysis*, but would have some runtime overhead. Our evaluation shows that these *overheads are negligible*. For example, *OpenEffectJ* was able to distinguish between effects of the method call `run` in PROGRAM (with `Prefix` class) and PROGRAM' (with `Hash` class) designed by two different programmers at two different times. This allows the for loop in the `ArrayList` class to be optimistically parallelized. Main benefits of this parallelization are reaped by PROGRAM', where the implementation of `run` method is safe to parallelize. However, PROGRAM also does not suffer significantly because *OpenEffectJ* preemptively detects conflicts and does not require rollbacks.

These benefits make an *open effects*-based analysis an interesting point in the design space between a fully static and a fully dynamic effect analysis. Since the annotation `@open` is explicit, programmers can control the optimism in the analysis and thus the overhead of its dynamic part.

In summary, main contributions of this work are:

- a language design with *open* effects that facilitate important patterns of optimistic and deterministic parallelism in object-oriented programs in Section 3 and examples in Section 2;
- a static semantics with open effects in Section 4. The novelty lies in the integration of the *open* effects with standard effects;
- a dynamic semantics with open effect concretization and open effect-based concurrency decisions in Section 5;
- a prototype compiler based on the OpenJDK Compiler in Section 6 that shows only about 0.13-7.65% overhead;
- a rigorous proof that *OpenEffectJ* ensures determinism – thus, users are guaranteed to avoid many complex concurrency issues in Section 5.4. The soundness proof is challenging compared to the static analyses, because the effect of a task could change at runtime, due to the *open* effects; and
- a comparative analysis with related ideas in Section 7.

2. Optimistic Effect Analysis for Reusable Code

We anticipate that *OpenEffectJ* is useful for exposing safe and optimistic concurrency in libraries and frameworks, which could be extended with possibly concurrency-unsafe code by clients, e.g the `ArrayList` class in Section 1. Here, we present further assessment on several representative algorithms.

2.1 Using Open Effects in Sorting Algorithms

We now study an implementation of merge sort. The code below is adapted from the package `java.util`. It uses a divide-and-conquer technique on line 16 and line 17 and combines it with an insertion sort as a base case for small arrays on lines 6-14.

To allow clients of this library to extend sorting by implementing application-specific comparisons, this library is designed to use an abstract class `Comparator`.

```
1 class Arrays {
2   final Object[] mergeSort(@open Comparator c,
3                Object[] src, int low, int high) {
4     int size = high - low;
5     Object[] dest = new Object[size];
6     if (size < THRESHOLD){//Use insertion sort
7       System.arraycopy(src, low, dest, 0, size);
8       for (int i=0; i<size; i++){
9         for (int j=i; j>0 &&
10              c.compare(src[j-1],src[j])>0; j--)
11          this.swap(dest,j,j-1);
12      }
13      return dest;
14    }
15    int mid = (low + high) / 2;
16    Object[] d1 = this.mergeSort(c, src, low, mid);
17    Object[] d2 = this.mergeSort(c, src, mid, high);
18    /* the details of merge omitted.*/ }
19 }
20 class Comparator extends Object {
21   int compare(Object o1, Object o2){}
22 }
```

The method `mergeSort` in the library uses an instance of the class `Comparator` on line 10 to compare two objects in the array.

It is almost a universal belief that the comparators are *pure* methods and thus this parallelization can be done safely. However, programmers may or may not subscribe to this belief. For example, in OpenJDK itself, the class `RuleBasedCollator` (RBC) in package `java.text` is-a `Comparator`, but the method `compare` has side effects. So the parallelization of merge sort may have heap conflicts if an instance of this class is used as a `Comparator` and would result in incorrect output[1]. We can annotate the class `Comparator` to require that method `compare` be `pure`; however, such annotation could make it unnecessarily difficult to implement certain comparators, e.g. RBC.

However, most comparators are side effect free. Thus, it would be nice to parallelize merge sort for these cases. We can do so in

[1] The original code in `RuleBasedCollator` is thread safe though.

OpenEffectJ by declaring the parameter `c` in method `mergeSort` as *open* on line 2. Nothing else changes!

To illustrate how this declaration facilitates safe, optimistic concurrency, consider an example client code below.

```
23 Comparator cPure = new...//Pure comparator
24 Comparator cDirty = new...//Comparator w/ effects
25 Arrays a = new Arrays ();

27 // Following will do a parallel sort.
28 Object[] d1 = a.mergeSort(cPure,src,0,src.length);

30 //Following will do a sequential sort.
31 Object[] d2 = a.mergeSort(cDirty,src,0,src.length);
```

In the code above, there are two comparators: `cPure` that does not have side-effects and `cDirty` that does. The *open* parameter `c` is bound to `cPure` on line 28. Due to this binding, potential side-effects of the call `a.mergeSort` are updated. Since the method `compare`, if called with the receiver object `cPure`, would have no side-effect, the call `a.mergeSort` also has no externally visible side-effects. So, the sort initiated on line 28 can be parallelized.

Later, the same *open* parameter `c` is bound to `cDirty` on line 31. Due to this, potential side-effects of the method call `a.mergeSort` are updated. Since the method `compare`, if called with the receiver object `cDirty`, would have side-effects, the method call `a.mergeSort` may also have externally visible side-effects. So, the sort initiated on line 31 is done sequentially.

This further illustrates main benefits of *OpenEffectJ* that it facilitated optimistic and safe parallelization of a reusable implementation of `mergeSort`. This was done without requiring access to the complete inheritance hierarchy of the class `Comparator`, which adheres to the open-world assumption. Furthermore, no restrictions were imposed on the inheritance hierarchy of this class. Last but not least, only one annotation was required to accomplish this task.

2.2 Using Open Effects in Search Algorithms

Next, we study a representative search algorithm, depth-first search (DFS). DFS is typically formulated as a graph traversal as in the listing below [23]. For extensibility and reuse, this library is designed to use abstract implementation of classes `Node` and `Goal`. A client of this library would extend the class `Node` on line 17 to create an application-specific node and extend the class `Goal` on line 18 to implement application-specific search objectives.

```
1 class DFS {
2   final boolean dfs(@open Goal goal, Node curr) {
3     if (mark.contains(curr)) return false;
4     else mark.add(curr);
5     boolean found = false;
6     if(goal.satisfied(curr)) {
7       rs.add(curr);
8       found = true;
9     }
10    for(int i=0; i<curr.children.length; i++)
11      found |= this.dfs(goal,curr.children[i]);
12    return found;
13  }
14  HashSet mark = new HashSet();
15  HashSet rs = new HashSet();
16 }
17 class Node { Node[] children; }
18 class Goal {
19   boolean satisfied(Node node){ return false; }
20 }
```

The algorithm could be parallelized by executing the recursive `dfs` concurrently, on line 10. But this parallelization may not be safe if the call `satisfied` on `goal` on line 6 is concurrency unsafe. However, developers of the library class `DFS` have neither access to, nor control over the implementation of subclasses of the class `Goal` in client code.

So we declare the parameter `goal` *open*, shown on line 2. The method `satisfied` is pure when `goal` points to an instance of side-effect free goal tester and the depth-first search can be safely parallelized. At the same time, if clients of the DFS library need to extend the class `Goal` with a subclass whose `satisfied` method has side-effects, they are free to do so. As we discuss in Section 5, an implementation of *open* effects can produce a warning to alert such clients that they may be missing out on potential concurrency.

2.3 Using Open Effects with Map-Reduce Framework

We now illustrate the usage of open effects in safe parallelization of programs that use the MapReduce framework [1]. In this framework, first step is *map*, i.e partitioning the problem and distributing it to worker, and second step is *reduce*, i.e. combining results from workers. Our code below is inspired by JSR166 [1]. For extensibility and reuse, this library is designed to use abstract implementation of classes `Mapper` and `Reducer`. These classes are extended by clients to implement application-specific functionality.

```
1 class Mapper { int map(int a); }
2 class Reducer { int reduce(int a, int b); }
3 class MapReduce {
4   @open Mapper mapper;          //An open field
5   @open Reducer reducer;
6   int[] arr;
7   void setMapper(Mapper m){ this.mapper = m; }
8   void setReducer(Reducer r){ this.reducer = r; }
9   final int compute(int low, int high) {
10    if(high - low <= THRESHOLD) {
11      int x = 0;
12      for (int i = low; i < high; ++i) {
13        int temp = mapper.map(arr[i]);
14        x = reducer.reduce(x, temp);
15      }
16      return x;
17    }
18    int mid = (low + high) >>> 1;
19    int r1 = this.compute(low, mid);
20    int r2 = this.compute(mid, high);
21    return reducer.reduce(r1, r2);
22  }
23 }
```

When the input range is small, the base case applies, on lines 32-39, where the `map` method is applied on each element of the array on line 35. The results from the `Mapper` are combined, on line 36, by the `Reducer`. The algorithm divides the array into non overlapping subarrays and recursively applies `compute` on these subarrays, on lines 41-42.

This algorithm could be parallelized by executing the `compute` on the subarrays concurrently, lines 41-42. Since we may not know about the effect of each subclass of the `Mapper` and `Reducer`, we declare them as *open* fields, lines 26-27, to ensure concurrency safety. The method calls on these fields result in two *open* effects. The method `compute` can be safely parallelized when `mapper` and `reducer` point to instances of concurrency safe subclasses of `Mapper` and `Reducer`, respectively.

2.4 Using Open Effects with Numerical Integration

We now illustrate the usage of open effects in safe parallelization of programs that Guassian Quadrature for numerical integration. In this application, first step is *map*, i.e partitioning the problem and distributing it to worker, and second step is *reduce*, i.e. combining results from workers. Our code below is inspired by an application from the Fork/Join Framework [1], which, in turn, is inspired by a filaments program [**?**]. For extensibility and reuse, this library is

designed to use abstract implementation of classes `Mapper` and `Reducer`. These classes are extended by clients to implement application-specific functionality.

```
1 class Function { double compute(double x); }
2 class Integrate {
3   @open Mapper mapper;          //An open field
4   @open Reducer reducer;
5   int[] arr;
6   void setMapper(Mapper m){ this.mapper = m; }
7   void setReducer(Reducer r){ this.reducer = r; }
8   final int compute(int low, int high) {
9     if(high - low <= THRESHOLD) {
10      int x = 0;
11      for (int i = low; i < high; ++i) {
12        int temp = mapper.map(arr[i]);
13        x = reducer.reduce(x, temp);
14      }
15      return x;
16    }
17    int mid = (low + high) >>> 1;
18    int r1 = this.compute(low, mid);
19    int r2 = this.compute(mid, high);
20    return reducer.reduce(r1, r2);
21  }
22 }
```

When the input range is small, the base case applies, on lines 32-39, where the `map` method is applied on each element of the array on line 35. The results from the `Mapper` are combined, on line 36, by the `Reducer`. The algorithm divides the array into non overlapping subarrays and recursively applies `compute` on these subarrays, on lines 41-42.

This algorithm could be parallelized by executing the `compute` on the subarrays concurrently, lines 41-42. Since we may not know about the effect of each subclass of the `Mapper` and `Reducer`, we declare them as *open* fields, lines 26-27, to ensure concurrency safety. The method calls on these fields result in two *open* effects. The method `compute` can be safely parallelized when `mapper` and `reducer` point to instances of concurrency safe subclasses of `Mapper` and `Reducer`, respectively.

Summary We applied *OpenEffectJ* to 4 representative examples, 3 of which are library classes from OpenJDK. For each case *OpenEffectJ* gracefully assists the programmer in parallelization of reusable libraries and/or frameworks. Here the libraries or frameworks could be extended by the clients. Thus, *OpenEffectJ* optimistically provides safe concurrency opportunities. In each case, at most two annotations were needed to safely parallelize the library class under consideration. Finally, in each case *OpenEffectJ* did not require the entire client code for effect analysis.

3. A Concurrent Object-oriented Calculus

This section introduces *OpenEffectJ*, an expression language based on Classic Java [19]. The grammar is shown in Figure 4. The grammar include two interim expressions for semantics: *loc* that represents locations and **`yield`** that models concurrency.

The notation over-bar denotes a finite ordered sequence ($\overline{a}$ stands for $a_1 \ldots a_n$). The notation `[a]` means that `a` is optional. The examples in Section 2 used an extended language with integers and boolean constants and operations, e.g., `int` is used as a shorthand for `Integer`, `i < j` is for `i.lt(j)`, etc. As in Fork/Join framework [25], we desugar a for loop to the code below.

```
1 //Sugar: for(int i=l; i<h; i++) be
2 final void loop (int l, int h) {
3   if(h == l) be
4   else fork (loop (l,h-l/2); loop (h-l/2,h))
5 }
```

prog ::= $\overline{\text{decl}}$ e
decl ::= **`class`** c **`extends`** d { $\overline{\text{field}}$ $\overline{\text{meth}}$ }
field ::= [**`@open`**] c f;
meth ::= t m ($\overline{\text{arg}}$) { e }
t ::= c | **`void`**
arg ::= c var, where var $\neq$ **`this`**
e ::= **`new`** c () | var | **`null`** | e.m ($\overline{\text{e}}$) | e.f | e.f = e
| arg = e ; e | e ; e | **`fork`** (e ; e)

where

c	$\in$	$\mathscr{C}$, a set of class names
d	$\in$	$\mathscr{C} \cup \{Object\}$, superclass names
f	$\in$	$\mathscr{F}$, a set of field names
m	$\in$	$\mathscr{M}$, a set of method names
var	$\in$	$\mathscr{V} \cup \{$**`this`**$\}$, $\mathscr{V}$ is a set of variable names

Figure 4. The Grammar for *OpenEffectJ*.

OpenEffectJ provides one construct for expressing parallelism: the **`fork`** expression. This expression has the form **`fork`**$(e_0;e_1)$. It runs e_0 and e_1 in two concurrent tasks. Concurrent execution of these tasks is dependent upon whether they may conflict. If these tasks do not conflict they are run concurrently, else sequentially.

A novelty in *OpenEffectJ*'s semantics is that this parallelization decision is based upon the information *available at the time of the evaluation of a* **`fork`** *expression*. Delaying parallelization decision allows the use of useful dynamic information. This is similar to Best *et al.*'s synchronization via scheduling [5], but differs from static [33] and dynamic analysis [18] (more in Section 7).

A programmer writing parallel code in reusable classes, e.g. the class `ArrayList` in Figure 2, typically knows that a reference such as `c` in that class may point to concrete objects of different types at runtime, and if a method is called on such reference, it may result in different effects. Some of these effects may be concurrency safe while the others may not, which will affect the parallelization decision. In such cases, they can mark the type of these references as **`open`**, using the annotation **`@open`** (e.g. line 4 in Figure 2).

To focus our attention on the essence of **`@open`** effects and to make the presentation tractable, we have formalized a subset of *OpenEffectJ*. In the core language, only fields can be annotated **`@open`**, but not other references, e.g., variables and parameters. In Section 5.5, we will discuss how we could extend the formalism to model other **`@open`** references. So all the examples presented work out nicely as they are.

4. Type and Static Effect Computation

OpenEffectJ's type-and-effect system has both a static and a dynamic part. The purpose of the static part is to compute the effects of every method. The statically computed effects include standard read/write effects and bottom effect[2].

Main novelty of this effect system is the notion of *open* effects. An *open* effect is produced as a result of evaluating method call expressions with *open* fields as receiver object. The format is **`@open`** $f\ m\ \rho$. It contain the name of the *open* field f, the method m being invoked, and a placeholder for concrete effects ρ.

The placeholder for concrete effects in an open effect is used by the dynamic part of our type-and-effect system. At runtime, the dynamic part concretizes the *open* effects by filling in this placeholder with actual effects. These concretization points happen whenever f

[2] We simplify the presentation by avoiding tracking object instances in read/write effects. In the implementation, *OpenEffectJ* tracks object instances and uses a sound intra-procedural aliasing analysis [20] and a purity analysis [35] to improve the precision of the static effect analysis.

is set. *OpenEffectJ* uses these concretized effects to make concurrency decisions, when evaluating the **fork** expressions.

4.1 Notations and Conventions

As Figure 5 shows, we represent type attributes for expressions as (t,ρ), the type t of an expression and its effects ρ.

$\theta ::= \text{OK}$	"program/decl/body types"
$\mid (t_1 \times \ldots \times t_n \to t,\rho)$ in c	"method types"
$\mid (t,\rho)$	"expression types"
$\rho ::= \{\varepsilon_j\}_{j\in\mathbb{N}}$	"program effects"
$\varepsilon ::= \texttt{read}\ f$	"read effect"
$\mid \texttt{write}\ f$	"write effect"
$\mid \texttt{open}\ f\ m\ \rho$	"*open* effect"
$\mid \bot$	"bottom effect"
$\Pi ::= \{var_j \mapsto t_j\}_{j\in\mathbb{N}}$	"type environments"

Figure 5. Type and effect attributes.

The notation $t' <: t$ means t' is a subtype of t. It is the standard reflexive-transitive closure of the declared subclass relationship [19]. We will use the notation $\rho_0 \sharp \rho_1$ to mean that effects ρ_0 and ρ_1 do not interfere and $\rho_0 \not\sharp \rho_1$ to mean that they do interfere.

We state the type checking rules using a fixed class table (list of declarations *CT* [19]). The typing rules for expressions use a type environment Π, which is a finite partial mapping from variable names *var* to a type t. Each method in the class table (CT) contains its effect ρ, computed by *OpenEffectJ*'s static type-and-effect system, in its signature.

The rules for top-level declarations are fairly standard. The typings and effects rules for OO expressions that do not produce effects are also standard. These include object creation, variable reference and declaration, null reference and sequence (Section A.1 contains the rules for programs, classes and the standard OO rules).

4.2 Type-and-Effect Rules for Method Declaration

The (T-METHOD) rule says that a method m type checks in class c, in which m is declared, if the body has type u and latent effect ρ.

$$\text{(T-METHOD)}\quad \frac{\begin{array}{c} override(m,c,(t_1 \times .. \times t_n \to t)) \\ \forall i \in \{1..n\},\ isClass(t_i) \qquad isType(t) \\ (var_1 : t_1,..,var_n : t_n, \texttt{this} : c) \vdash e : (u,\rho) \qquad u <: t \end{array}}{\vdash\ t\ m(t_1\ var_1,..,t_n\ var_n)\{e\} : (t_1 \times .. \times t_n \to t,\rho)\ \text{in}\ c}$$

This rule uses a function *override* (below). Unlike some static type-and-effect systems for concurrency [33], *OpenEffectJ* does not require that method overiding implies effect containment.

$$\frac{\begin{array}{c} CT(c) = \texttt{class}\ c\ \texttt{extends}\ d\ \{\overline{field}\ meth_1 \ldots meth_p\} \\ \nexists i \in \{1..p\}\ \text{s.t.}\ meth_i = t\ m(t_1\ var_1,\ldots,t_n\ var_n)\{e\} \\ override(m,d,(t_1 \times \ldots \times t_n \to t)) \end{array}}{override(m,c,(t_1 \times \ldots \times t_n \to t))}$$

$$\frac{(d,t,m\ (t_1\ var_1,\ldots,t_n\ var_n)\ \{e\},\rho') = findMeth(c,m)}{override(m,c,(t_1 \times \ldots \times t_n \to t))}$$

$$override(m,Object,(t_1 \times \ldots \times t_n \to t,\rho))$$

The function *findMeth*(used by *override*) looks up the method m, starting from the class c, looking in superclasses if necessary.

4.3 Open Effects of Polymorphic Method calls

The rules for method call are one of the central new rules. The typings for these rules are standard, however, for effects we distinguish based on the kind of the receiver object. We first discuss the pessimistic case, in which the receiver of the call is not an *open* field.

$$\frac{\begin{array}{c} CT(c) = \texttt{class}\ c\ \texttt{extends}\ d\ \{\overline{field}\ meth_1 \ldots meth_p\} \\ \exists i \in \{1\ldots p\}\ \text{s.t.}\ meth_i = (t,\rho,m(\overline{t\ var})\{e\}) \end{array}}{findMeth(c,m) = (c,t,m(\overline{t\ var})\{e\},\rho)}$$

$$\frac{\begin{array}{c} CT(c) = \texttt{class}\ c\ \texttt{extends}\ d\ \{\overline{field}\ meth_1 \ldots meth_p\} \\ \nexists i \in \{1\ldots p\}\ \text{s.t.}\ meth_i = (t,\rho,m(\overline{t\ var})\{e\}) \qquad findMeth(d,m) = l \end{array}}{findMeth(c,m) = l}$$

$$\text{(T-CALL)}\quad \frac{\begin{array}{c} e_0 \neq \texttt{this}.f \vee (e_0 = \texttt{this}.f \wedge typeOfF(f) \neq (c,\texttt{@open}\ c_0))) \\ findMeth(c_0,m) = (c_1,t,m\ (t_1\ var_1,\ldots,t_n\ var_n)\ \{e_{n+1}\},\rho) \\ \Pi \vdash e_0 : (c_0,\rho_0) \qquad (\forall\ i \in \{1..n\}\ ::\ \Pi \vdash e_i : (t_i',\rho_i)\ \wedge\ t_i' <: t_i) \end{array}}{\Pi \vdash e_0.m(e_1,\ldots,e_n) : (t,\{\bot\})}$$

In this case, statically we may not know which method will be invoked due to dynamic dispatch, nor its exact effect. Thus, the effect of this call is taken as a bottom effect[3]. In the implementation, *OpenEffectJ* relaxes this restriction by applying the following sound and modular optimizations. If a method is declared private or final; or its enclosing class is declared final; or if we know the exact type of the receiver (by applying a sound *intra-procedural* aliasing analysis [20]), the exact callee will be known; then we safely inline the effect of the callee in place of the call expression.

The optimistic case (T-CALL-OPEN) applies when a method m is called with an *open* field f as a receiver object.

$$\text{(T-CALL-OPEN)}\quad \frac{\begin{array}{c} \Pi \vdash \texttt{this}.f : (c_0,\rho_0) \qquad typeOfF(f) = (d,\texttt{@open}\ c_0) \\ findMeth(c_0,m) = (c_1,t,m\ (t_1\ var_1,\ldots,t_n\ var_n)\ \{e_{n+1}\},\rho) \\ (\forall\ i \in \{1..n\}\ ::\ \Pi \vdash e_i : (t_i',\rho_i)\ \wedge\ t_i' <: t_i) \end{array}}{\Pi \vdash \texttt{this}.f.m(e_1,\ldots,e_n) : (t,\{\texttt{open}\ f\ m\ \emptyset\} \cup \bigcup_{i=0}^{n} \rho_i)}$$

In this case, statically we assume that this method call will have no effect (represented by $\emptyset$ in **open** $f\ m\ \emptyset$).

To illustrate the implication of this assumption, let e_0 and e_1 be two expressions in two potential tasks id_0 and id_1, and their corresponding effects be ρ_0 and ρ_1. If neither ρ_0 nor ρ_1 contains any *open* effect (i.e., $\nexists f,m,\rho$ s.t.$\forall i \in \{0,1\}$ :: **@open** $f\ m\ \rho \notin \rho_i$), *OpenEffectJ* acts just like a static type-and-effect system. At compile time, it advises a parallelizing compiler to parallelize id_0 and id_1 if ρ_0 and ρ_1 do not interfere, i.e. $\rho_0 \sharp \rho_1$ [33]. If ρ_0 and ρ_1 interfere ($\rho_0 \not\sharp \rho_1$), it suggests sequential execution.

Consider the scenario where *open* effect exists in ρ_0 and/or ρ_1 (i.e., $\exists f,m,\rho$ s.t.$\exists i \in \{0,1\}$ :: **@open** $f\ m\ \rho \in \rho_i$). Let $|\rho|$ be the concrete effects in set ρ, i.e., $|\rho| = \{\varepsilon | (\varepsilon \in \rho) \wedge (\varepsilon \neq \texttt{@open}\ f\ m\ \rho')\}$. At compile time, *OpenEffectJ* advises a parallelizing compiler to parallelize id_0 and id_1 if $|\rho_0| \sharp |\rho_1|$, but to insert an effect check right before parallelizing them (e.g., before evaluating **fork** $(e_0;e_1)$). In Section 5, we will illustrate one such effect checking technique via the semantics of the **fork** expression.

4.4 Type-and-effects for Field Related Expressions

The typings and effects for the field access rules (T-GET) and (T-SET) are standard. The auxiliary function *typeOfF*, uses the class

[3] Using bottom effect in this context bears a conscious analogy to writing no annotation in the static approaches [33], which means that the method may read and write the entire heap. Thus, it presents an opportunity to explore the idea of combining *open* effect with the specification approaches that specify the effect ρ of the method m. In that case, the bottom effect will be substituted by the effect ρ.

table CT to find the type of a field f, the class in which f is declared and the *open* annotation information, for the input field f.

$$\text{(T-GET)}\quad \frac{\Pi \vdash e : (c,\rho) \qquad \mathit{typeOfF}(f) = (d, [\texttt{@open}]\ t) \qquad c <: d}{\Pi \vdash e.f : (t, \rho \cup \{\texttt{read}\ f\})}$$

$$\text{(T-SET)}\quad \frac{\Pi \vdash e : (c,\rho) \qquad c <: c' \qquad \mathit{typeOfF}(f) = (c', t) \qquad \Pi \vdash e' : (t',\rho') \qquad t' <: t}{\Pi \vdash e.f = e' : (t', \rho \cup \rho' \cup \{\texttt{write}\ f\})}$$

$$\text{(T-SET-OPEN)}\quad \frac{\Pi \vdash e : (c,\rho) \qquad c <: c' \qquad \mathit{typeOfF}(f) = (c', \texttt{@open}\ t) \qquad \Pi \vdash e' : (t',\rho') \qquad t' <: t}{\Pi \vdash e.f = e' : (t', \{\bot\})}$$

Using only field to denote read/write effect here is somewhat conservative. However, it helps create an efficient dynamic effect management system, which is crucial.

The (T-SET-OPEN) rule denotes a concretization point, which may change the static precomputed effects of other methods. *OpenEffectJ* gives it a bottom effect to maintain soundness. In practice, this can be relaxed using an aliasing analysis to determine the set of concrete objects that an open field can point to.

$$\text{(T-YIELD)}\quad \frac{\Pi \vdash e : (t,\rho)}{\Pi \vdash \texttt{yield}\ e : (t,\rho)} \qquad \text{(T-FORK)}\quad \frac{\Pi \vdash e : (t,\rho') \qquad \Pi \vdash e' : (t',\rho')}{\Pi \vdash \texttt{fork}\ (e : e') : (\texttt{void}, \rho \cup \rho')}$$

Concurrency expressions The (T-YIELD) rule says that a `yield` expression has the same type and effect as the expression e. The (T-FORK) first type checks its two subexpressions. It has the type **void**. Its effect is the union of the effects of the subexpressions.

5. Dynamic Semantics with Open Effects

Here we give a small-step operational semantics for *OpenEffectJ*. To the best of our knowledge, this is the first work to integrate dynamic effect management with an object-oriented semantics, to enable safe, optimistic concurrency.

5.1 Notations and Conventions

The small steps taken in the semantics are defined as transitions from one configuration (Σ in Figure 6) to another. Some rules use an implicit attribute, the class table CT.

Evaluation relation: $\hookrightarrow : \Sigma \dashrightarrow \Sigma$

Intermediate expressions :

e	::=	$loc \mid \texttt{yield}\ e$	where $loc \in \mathscr{L}$, a set of locations

Domains:

Σ	::=	$\langle \psi, \mu \rangle$	"Program Configurations"
ψ	::=	$\langle e, \tau \rangle + \psi \mid \bullet$	"Task Queue"
τ	::=	$(id, \{id_j \mid j \in \mathbb{N}\})$	"Task Dependencies"
		where $id,\ id_j \in \mathbb{N}$,	
μ	::=	$\{loc \mapsto o_j\}_{j \in \mathbb{N}}$	"Store"
o	::=	$[c\,.\,F\,.\,E]$	"Object Records"
F	::=	$\{f_j \mapsto v_j\}_{j \in \mathbb{N}}$	"Field Maps"
v	::=	$\texttt{null} \mid loc$	"Values"
E	::=	$\{m_j \mapsto \rho_j\}_{j \in \mathbb{N}}$	"Effect Maps"

Evaluation contexts:

$$\mathbb{E} ::= - \mid \mathbb{E}.m(\overline{e}) \mid v.m(\overline{v}, \mathbb{E}, \overline{e}) \mid \mathbb{E}.f{=}e \mid v.f{=}\mathbb{E} \mid \mathbb{E};e \mid \mathbb{E}.f \mid t\ var{=}\mathbb{E};e$$

Figure 6. Notations Used in the Dynamic Semantics.

A configuration consists of a task queue ψ and a global store μ. A store maps locations to object records. An object record $o = [c.F.E]$ contains the concrete type c of the object, a field map F, and a dynamic effect map E (which is new).

An effect map E is a function from a method name to its runtime effects. The task queue ψ consists of a list of tasks $\langle e, \tau \rangle$. Each task consists of an expression e and the corresponding task dependencies τ. The expression e serves as the remaining evaluation for the task. The task dependencies τ are used to record the identity of the current task (*id*) and identities of tasks (children set) that it waits on.

We present the semantics as a set of evaluation contexts $\mathbb{E}$ and an one-step reduction relation that acts on the position in the overall expression identified by the evaluation context [19]. This avoids the need for writing out standard recursive rules and clearly presents the order of evaluation. The language uses a call-by-value evaluation strategy. The initial configuration of a program with a main expression e is $\Sigma_\star = \langle \langle e, (0, \emptyset) \rangle, \bullet \rangle$. The operator $\oplus$ is an overriding operator for finite functions, i.e. if $\mu' = \mu \oplus \{loc \mapsto o\}$, then $\mu'(loc') = o$ if $loc' = loc$, otherwise $\mu'(loc') = \mu(loc')$.

5.2 Dynamic Effect Management in OO Expressions

The semantics rules for standard OO expressions are shown below (Section B contains omitted auxiliary functions). Compared to traditional dynamic semantics for OO expressions [19], there are two main differences. First, the `yield` expression is used in the resulting configuration to relinquish control to other tasks. Second, some of the rules manipulate the dynamic effect map E.

$$\text{(NEW)}\quad \frac{\begin{array}{c} loc \notin dom(\mu) \\ \mu' = \{loc \mapsto [c.\{f \mapsto \texttt{null} \mid f \in \mathit{fields}(c)\} \\ .\{m \mapsto \rho \mid m \mapsto \rho \in \mathit{methE}(c)\}]\} \oplus \mu \end{array}}{\langle \langle \mathbb{E}[\texttt{new}\ c()], \tau \rangle + \psi, \mu \rangle \hookrightarrow \langle \langle \mathbb{E}[\texttt{yield}\ loc], \tau \rangle + \psi, \mu' \rangle}$$

The (NEW) rule uses a function *methE* (below) to initialize the effect map of the new instance. This function searches the class table CT for all the methods declared in class c and all its super classes. Its result is a map E that contains each method m found in previous step and its statically computed effects ρ. The type-and-effect rules in Section 4 are used to compute the effects ρ.

$$\begin{array}{l} \mathit{methE}(c) = E \oplus \bigcup_{i=0}^{n} \{m_i \mapsto \rho_i\} \\ \quad \textbf{where}\ CT(c) = \texttt{class}\ \ c\ \texttt{extends}\ d\ \{\overline{\mathit{field}}\ \mathit{meth}_1 \ldots \mathit{meth}_n\} \\ \quad \textbf{and}\ \mathit{methE}(d) = E \\ \quad \textbf{and}\ (\forall\ i \in \{1..n\}\ ::\ \mathit{findMeth}(c, m_i) = (c, t_i, m_i(\overline{t\ var}), \rho_i)) \end{array}$$

$$\text{(GET)}\quad \frac{\mu(loc) = [c.F.E] \qquad v = F(f)}{\langle \langle \mathbb{E}[loc.f], \tau \rangle + \psi, \mu \rangle \hookrightarrow \langle \langle \mathbb{E}[\texttt{yield}\ v], \tau \rangle + \psi, \mu \rangle} \qquad \text{(SET)}\quad \frac{[c.F.E] = \mu(loc) \qquad \mu_0 = \mu \oplus (loc \mapsto [c.(F \oplus (f \mapsto v)).E]) \qquad \mu' = \mathit{update}(\mu_0, loc, f, v)}{\langle \langle \mathbb{E}[loc.f = v], \tau \rangle + \psi, \mu \rangle \hookrightarrow \langle \langle \mathbb{E}[\texttt{yield}\ v], \tau \rangle + \psi, \mu' \rangle}$$

The semantics of field get is standard, whereas that of field set is new. If a field is declared *open*, assigning a value to it may change the effect of those methods that access it. The function *update* shown below implements this.

$$\begin{array}{l} \mathit{update}(\mu, loc, f, v) = \mu \quad \textbf{where}\ \mu(loc) = [c.F.E] \\ \quad \textbf{and}\ E = \mathit{updateEff}(\mu, f, v, E) \\ \mathit{update}(\mu, loc, f, v) = \mu'' \ \textbf{where}\ \mu(loc) = [c.F.E]\ \textbf{and}\ E' \neq E \\ \quad \textbf{and}\ E' = \mathit{updateEff}(\mu, f, v, E) \quad \textbf{and}\ \mathit{reverse}(\mu, loc) = \kappa \\ \quad \textbf{and}\ \mu' = \{loc \mapsto [c\,.\,F\,.\,E']\} \oplus \mu\ \textbf{and}\ \mathit{fixPoint}(\mu', loc, \kappa) = \mu'' \end{array}$$

The inputs to *update* are the current store μ, the object reference *loc*, the field f, and the R-value v. The output is a modified store. This function first updates the effect (by calling *updateEff*) of the object pointed to by *loc* (by the effect of an object o, we mean

the effects of all the methods of o). If the effects of o remain unchanged, the algorithm stops. Otherwise, the effect of an object o', which has some *open* field pointing to o, should also be changed. Effects are further propagated using the function *fixPoint* until a fixed point is reached.

$$reverse(\mu, loc) = \bigcup_{i=1}^{n} S_i \ \textbf{where} \ \forall i \in \{1..n\} \text{ s.t. } loc_i \in dom(\mu) :: S_i = \{\langle loc_i, f\rangle \,|\, F(f) = loc \wedge \mu(loc_i) = [c.F.E]\}$$

The function *reverse*, searches the input store μ for objects loc_i and field f_i pair that is pointing to the current object *loc*. In practice, reverse pointers could be used to optimize this update [5].

$$updateEff(\mu, f, v, \overline{\langle m, \rho\rangle}) = \overline{\langle m, \rho'\rangle}$$
$$\textbf{where} \ \forall\, i \in \{1..n\} \rho_i = \{\varepsilon_k | 1 \le k \le p\} \ \textbf{and} \ \rho'_i = \{\varepsilon'_k | 1 \le k \le p\}$$
$$\textbf{and} \ \forall\, j \in \{1..p\} \ :: \ \varepsilon_j \in \rho_i : concretize(\mu, f, v, \varepsilon_j) = \varepsilon'_j$$

Each object contains a map E of effects. The *updateEff* function concretizes the effects in E one by one, by calling *concretize*.

$$\begin{array}{l} concretize(\mu, f, v, \varepsilon) = \textbf{match}\ \varepsilon\ \textbf{with} \\ \quad |\ \texttt{@open}\ \ f'\ m\ \rho \to \textbf{match}\ f'\ \textbf{with} \\ \quad\quad |\ f \to \textbf{match}\ v\ \textbf{with} \\ \quad\quad\quad |\ \texttt{null} \to \texttt{@open}\ \ f\ m\ \emptyset \\ \quad\quad\quad |\ loc \to \textbf{open}\ f\ m\ \rho'\ \ \textbf{where}\ [c.F.E] = \mu(loc), \\ \quad\quad\quad\quad\quad \textbf{and}\ \rho' = \cup\bigcup_{i=1}^{n} \rho_i \ \textbf{and}\ E(m) = \{\varepsilon_i | 1 \le i \le n\} \\ \quad\quad\quad\quad\quad \textbf{and}\ \ \forall\, i \in \{1..n\} :: cp(\varepsilon_i) = \rho_i \\ \quad\quad |\ \_ \to \varepsilon \\ \quad |\ \_ \to \varepsilon \end{array}$$

The function *concretize* changes the concrete effects in the placeholder inside an *open* effect. Note that when the *open* field f is set (in the (T-SET) rule), only the *open* effects that have f as receiver are concretized, i.e., `@open` $f\ m\ \rho$.

$$\begin{array}{ll} cp(\varepsilon) = \textbf{match}\ \varepsilon\ \textbf{with} & |\ \textbf{open}\ f\ m\ \rho \to \rho \\ & |\ \_ \to \varepsilon \end{array}$$

The function *cp* is used by the function *concretize* to retrieve the concrete effects, i.e., the effects of the R-value v are copied to fill the placeholder effects of the *open* effect of *loc* in the (SET) rule.

The (CALL) rule is standard. It acquires the method signature via the function *findMeth* (Section A.1) that uses dynamic dispatch[19].

(CALL)
$$\frac{(c', t, m(t_1\ var_1, \ldots, t_n\ var_n)\{e\}, \rho) = findMeth(c, m) \quad [c.F.E] = \mu(loc) \quad e' = [loc/\textbf{this}, v_1/var_1, \ldots, v_n/var_n]e}{\langle \mathbb{E}[loc.m(v_1, \ldots, v_n)], \tau\rangle + \psi, \mu\rangle \hookrightarrow \langle \langle \mathbb{E}[\textbf{yield}\ e'], \tau\rangle + \psi, \mu\rangle}$$

To summarize, in *OpenEffectJ*'s semantics, object creation is augmented to initialize the effect map; and field assignment to open fields updates these effect maps.

5.3 Safe Optimistic Concurrency using *Open* Effect

We now describe how open effects are used by a `fork` expression. Recall from Section 4 that *OpenEffectJ*'s static effect system would advise a parallelizing compiler to parallelize `fork` $(e_0\,;\,e_1)$ if and only if statically computed effects of e_0 and e_1 do not interfere.

Interference checks for open effects were deferred to allow optimistic parallelism. This deferred check is included in the dynamic semantics of the fork expression, which checks if the two subexpressions e_0 and e_1 could run in parallel. To do so, we use the effect judgments in Figure 7 that recursively computes the effects of subexpressions. An effect judgement of the form $\mu \vdash e : \rho$ means that an expression e has static effect ρ with respect to a store μ. In practice, expressions e_0 and e_1 can be wrapped into two compiler-generated methods m_0 and m_1; *OpenEffectJ* can then retrieve the effects of e_0 and e_1 from the effect map E of the `this` object, thus eliminating the need for this dynamic effect computation.

(E-NEW) $\mu \vdash \textbf{new}\ c() : \emptyset$ (E-VAR) $\mu \vdash var : \emptyset$ (E-NULL) $\mu \vdash \texttt{null} : \emptyset$ (E-LOC) $\mu \vdash loc : \emptyset$

(E-CALL-OPEN)
$$\frac{\begin{array}{c}\mu \vdash loc.f : \rho_0 \quad \mu(loc) = [c.F.E] \\ E = \{m_i \mapsto \rho_i | 1 \le i \le n\} \\ \exists i \text{ s.t. } (\exists \varepsilon = \texttt{@open}\ \ f\ m\ \rho \text{ s.t. } \varepsilon \in \rho_i) \\ typeOfF(f) = (d, \texttt{@open}\ \ c_0) \\ (\forall\, i \in \{1..n\}\ ::\ \mu \vdash e_i : \rho_i)\end{array}}{\mu \vdash loc.f.m(\overline{e}) : \{\textbf{open}\ f\ m\ \rho\} \cup \bigcup_{i=0}^{n} \rho_i}$$

(E-CALL-LOC)
$$\frac{\begin{array}{c}\mu(loc) = [c.F.E] \\ E(m) = \rho_0 \\ (\forall\, i \in \{1..n\}\ ::\ \mu \vdash e_i : \rho_i)\end{array}}{\mu \vdash loc.m(\overline{e}) : \bigcup_{i=0}^{n} \rho_i}$$

(E-CALL) $\mu \vdash e_0.m(\overline{e}) : \bot$

(E-GET)
$$\frac{\mu \vdash e : \rho}{\mu \vdash e.f : \rho \cup \{\textbf{read}\ f\}}$$

(E-SET-OPEN)
$$\frac{typeOfF(f) = (c, \texttt{@open}\ \ c_0)}{\mu \vdash e.f = e' : \{\bot\}}$$

(E-SET)
$$\frac{\mu \vdash e' : \rho' \quad \mu \vdash e : \rho \quad typeOfF(f) = (c, t)}{\mu \vdash e.f = e' : \rho \cup \rho' \cup \{\textbf{write}\ f\}}$$

(E-YIELD)
$$\frac{\mu \vdash e : \rho}{\mu \vdash \textbf{yield}\ e : \rho}$$

(E-FORK)
$$\frac{\mu \vdash e : \rho \quad \mu \vdash e' : \rho'}{\mu \vdash \textbf{fork}\ (e : e') : \rho \cup \rho'}$$

Figure 7. Effect judgment for expressions.

The dynamic rules in Figure 7 are similar to the static rules in Section 4, except for method calls on *open* fields (E-CALL-OPEN). The *open* effect now has a concrete part ρ, instead of $\emptyset$, because we know the concrete object that an *open* field is pointing to.

After computing the effects, the rules for fork verify that the effects of e_0 and e_1 do not interfere (written $\varepsilon_0 \sharp \varepsilon_1$). Read effects do not interfere; read/write and write/write pairs conflict if they access the same field; *open* effect `@open` $f\ m\ \rho$ conflicts with another effect ε if any effect ε' in ρ conflicts with ε; bottom effect $\bot$ conflicts with any effect.

If e_0 and e_1's effects do conflict, the (FORK-SEQUENTIAL) rule applies; otherwise the (FORK-PARALLEL) rule will be used. *OpenEffectJ* makes its concurrency decision at this point, which allows the usage of more accurate effect information than a pure static analysis. A pure dynamic analysis will optimistically execute the tasks in concurrent. But they may rollback when conflicts do happen (see Section 7 for more discussion). The (FORK-PARALLEL) rule creates 2 concurrent children tasks id_0 and id_1 and put them into the queue ψ. The current task is suspended until id_0 and id_1 are done. This is done by puting the id_0 and id_1 in the children set $(\tau' = (id, \{id_0, id_1\}))$. The previous children set I in the current forking task may be safely dropped, since the current forking task can not resume until its children are done.

(FORK-PARALLEL)
$$\frac{\begin{array}{c}\mu \vdash e_0 : \{\varepsilon_1, .., \varepsilon_n\} \\ \mu \vdash e_1 : \{\varepsilon'_1, .., \varepsilon'_{n'}\} \quad \forall i \in \{0..n\}, j \in \{0..n'\} \text{ s.t. } (\varepsilon_i \sharp \varepsilon_j) \\ id_0 = fresh() \quad id_1 = fresh() \quad \tau = (id, I) \\ \tau' = (id, \{id_0, id_1\}) \quad \psi' = \psi + \langle e_0, (id_0, \emptyset)\rangle + \langle e_1, (id_1, \emptyset)\rangle\end{array}}{\langle \langle \mathbb{E}[\textbf{fork}\ (e_0\,;\,e_1)], \tau\rangle + \psi, \mu\rangle \hookrightarrow \langle \langle \mathbb{E}[\textbf{yield}\ \texttt{null}], \tau'\rangle + \psi', \mu\rangle}$$

The (FORK-SEQUENTIAL) rule constructs an expression $e_0; e_1; \texttt{null}$, which sequentializes the `fork` expression to prevent data races. An alternative may be to signal an exception when the effects conflict [16, 18], which could be useful for debugging and reasoning about concurrency during program development.

(FORK-SEQUENTIAL)

$$\frac{\mu \vdash e_0 : \{\varepsilon_1,..,\varepsilon_n\} \qquad \mu \vdash e_1 : \{\varepsilon'_1,..,\varepsilon'_{n'}\} \quad \exists i \in \{0..n\}, j \in \{0..n'\} \text{ s.t. } (\varepsilon_i \sharp \varepsilon_j) \qquad e = e_0; e_1; \texttt{null}}{\langle\langle \mathbb{E}[\texttt{fork}\ (e_0\texttt{;}\ e_1)\ ], \tau\rangle + \psi, \mu\rangle \hookrightarrow \langle\langle \mathbb{E}[\texttt{yield}\ e], \tau\rangle + \psi, \mu\rangle}$$

The (YIELD) rule puts the current task to the end of the task queue and evaluates the next active task in this queue ψ.

(YIELD)

$$\frac{\langle e', \tau'\rangle + \psi' = active(\psi + \langle \mathbb{E}[e], \tau\rangle)}{\langle\langle \mathbb{E}[\texttt{yield}\ e], \tau\rangle + \psi, \mu\rangle \hookrightarrow \langle\langle e', \tau'\rangle + \psi', \mu\rangle}$$

(TASK-END)

$$\frac{\langle e', \tau'\rangle + \psi' = active(\psi)}{\langle\langle v, \tau\rangle + \psi, \mu\rangle \hookrightarrow \langle\langle e', \tau'\rangle + \psi', \mu\rangle}$$

Finding an active task is done by the function *active* (not shown). It returns the top most task in ψ that can be run. A task is ready to run if all the tasks in its children set are done (evaluated to a value v). The (TASK-END) rule says that the current task is done, thus it is removed from ψ and the next active task is scheduled.

5.4 Key Properties of *OpenEffectJ*

The key formal properties of *OpenEffectJ* are: effect preservation, type preservation, and determinism. The proof of type preservation uses the standard subject reduction argument [19]. It is contained in our report [**?**], which also contains detailed proof for effect preservation and determinism.

5.4.1 Effect Preservation

The effect preservation property is that the dynamic effect, i.e. heap accesses, of each concurrent task refines the static effect of that task computed when it is forked off. First, we define dynamic effects.

A dynamic effect η of a task *id* can be a read effect $(\texttt{rd}, loc, f, id)$ or a write effect $(\texttt{wt}, loc, f, id)$. A dynamic effect η refines a static effect ρ, written $\eta \propto \rho$, if either $\eta = (\texttt{rd}, loc, f, id) \wedge (\texttt{read}\ f) \in \rho$; or $\eta = (\texttt{wt}, loc, f, id) \wedge (\texttt{write}\ f) \in \rho$.

The dynamic effect of a task *id* is a dynamic trace $\chi = \overline{\eta}$, a sequence of dynamic effects.

Informally, effect preservation will hold if the dynamic effects of a task *id* and dynamic effects of all the child tasks spawned by *id* together refine the static effects of *id*. Since our dynamic semantics does not maintain the parent-child relationship between tasks nor maintain dynamic effects, we introduce an instrumented semantics *dyn*, which augments dynamic semantics in Section 5 with dynamic effects and an additional parent-child relationship Υ, which is a map $\{id_j \mapsto I_j\}_{j \in \mathbb{N}}$. Here, *id* is a task's identity and I is a set of its children tasks' identities.

The function *dyn* (below), records the dynamic memory footprint for each task [13, 33], which helps us understand the relation between the static effects and their corresponding dynamic effects. It is trivial to see that this instrumented semantics retains formal properties of the original dynamic semantics.

In the following, $dyn(\Sigma, \chi, \Upsilon) = dyn(\Sigma', \chi', \Upsilon')$ and $\Sigma \hookrightarrow \Sigma'$.	
Σ	**Side Conditions**
$\langle\langle \mathbb{E}[loc.f], (id, I)\rangle + \psi, \mu\rangle$	$\chi' = \chi + (\texttt{rd}, loc, f, id), \Upsilon' = \Upsilon$
$((\mathbb{E}[loc.f = v], (id, I)) + \psi, \mu)$	$\chi' = \chi + (\texttt{wt}, loc, f, id), \Upsilon' = \Upsilon$
$((\mathbb{E}[\texttt{fork}\ (e_0\texttt{;}\ e_1)\ ], (id, I)) + \psi, \mu)$	$\Sigma' = ((e', (id, I')) + \psi', \mu), \chi' = \chi$ $\Upsilon' = \{id \mapsto (\Upsilon(id) \cup I')\} \oplus \Upsilon$
Other cases	$\chi' = \chi, \Upsilon' = \Upsilon$

The novelty of *OpenEffectJ* is that it stores the effects ρ in object records for methods. These methods effects will be used by the `fork` expression at runtime. Therefore, two invariants (Definition 5.3), for these method effects ρ, are necessary to maintain the *effect preservation* property. These invariants include: the placeholder effect ρ_0, of an *open* effect `@open` $f\ m\ \rho_0$, should be supereffect $\supseteq$ of the effect $E'(m)$ for the method m of the object f is pointing to (Definition 5.1), i.e., $E'(m) \subseteq \rho_0$, and the effect $E(m)$ of a method m stored in the object record should be supereffect of the effect ρ of the body e of m (Definition 5.2). For example, in Figure 3, after the *open* parameter `c` is bound to the instance `p`, on line 17, the *open* effect, of the `for` loop of the `ArrayList` instance `a`, is supereffect of the effect of the method `run` of `p`;

DEFINITION 5.1. *[Well-formed object] An object record $o = [c.F.E]$ is a well-formed object in μ, written $\mu \vdash o$, if for all* open *effect* `@open` $f\ m\ \rho_0 \in \rho \in rng(E)$, *either* $(F(f) = loc) \wedge (\mu(loc) = [c'.F'.E']) \wedge (E'(m) \subseteq \rho_0)$*; or* $(F(f) = \texttt{null}) \wedge (\rho_0 = \emptyset)$.

DEFINITION 5.2. *[Well-formed location] A location loc is well-formed in μ, written $\mu \vdash loc$, if either $\mu(loc) = [c.F.E]$, $\forall m \in dom(E)$ s.t. $findMeth(c,m) = (c', t, m(\overline{t\ var})\{e\}, \rho') \wedge \mu \vdash [loc/\texttt{this}]e : \rho$, then $\rho \subseteq E(m)$; or $\mu(loc) = \texttt{null}$.*

DEFINITION 5.3. *[Well-formed store] A store μ is well-formed, written $\mu \vdash \diamond$, if $\forall o \in rng(\mu)$ s.t. $\mu \vdash o$ and $\forall loc \in dom(\mu)$ s.t. $\mu \vdash loc$.*

THEOREM 5.4. [Effect preservation] *Let the program configuration be $\Sigma = \langle\langle e, (id, I)\rangle + \psi, \mu\rangle$. If it transits to another configuration $\Sigma \hookrightarrow \langle\langle e', (id, I')\rangle + \psi', \mu'\rangle$, the store is well-formed $\mu \vdash \diamond$, $\mu \vdash e : \rho$, and $(\bot) \notin \rho$, then there is some ρ', χ s.t.(a) $\mu' \vdash e' : \rho'$, and $\rho' \subseteq \rho$; and (b) $(dyn(\Sigma, \chi, \Upsilon) = (\Sigma', \chi + \eta, \Upsilon')) \Rightarrow (\eta \propto \rho)$*

Proof Sketch: The essence of Theorem 5.4 is that during program execution, the subsequent expression e' has a subeffect $\rho' \subseteq \rho$ of the previous expression e, with the effect judgment $\mu \vdash e : \rho$ (Figure 7). We prove that the dynamic effect η in each step refines the static ρ of the original expression e, $\eta \propto \rho$. Thus with (a), η refines the effect ρ_0 of the expression e_0 when the task was forked off, with the heap μ_0, e.g., `fork` $(e_0\texttt{;}\ e_1)$ and $\mu_0 \vdash e_0 : \rho_0$. Unlike the static approaches, which compute ρ_0 at compile-time, *OpenEffectJ* computes ρ_0 before evaluating the fork expression.

5.4.2 Determinism

We prove the determinism of *OpenEffectJ* programs by showing that the concurrent tasks do not have dynamic effect interference and therefore a well-typed *OpenEffectJ* program produces the same result given the same input.

To prove that the tasks do not have heap interference, we introduce the accumulated dynamic effects function *dynE*. These effects, produced by a concurrent task *id* and all its descendants *id'* ($id' \leqslant_\Upsilon id$), refine the static effect computed when *id* is forked off.

The function *dynE* is: $dynE(\Sigma, \chi, \Upsilon, n) = dynE(\Sigma', \chi', \Upsilon', n-1)$, if $dyn(\Sigma, \chi, \Upsilon) = (\Sigma', \chi', \Upsilon')$ and $dynE(\Sigma, \chi, \Upsilon, 0) = dyn(\Sigma, \chi, \Upsilon')$.

A task id' is a descendant of a task *id*, with Υ, written as $id' \leqslant_\Upsilon id$, if $id' \in desc(id, \Upsilon)$. Here, $desc(id, \Upsilon) = I \cup \bigcup_{i=0}^{n} I_i$, $\Upsilon(id) = I = \{id_0, \ldots, id_n\}$, and $\forall id_i \in I, desc(id_i, \Upsilon) = I_i$.

To reason about the interaction between the two concurrent tasks, we define the newly generated queue ψ', which is formed by these two tasks, i.e., we write $\Sigma \twoheadrightarrow \psi'$, if $\Sigma \hookrightarrow \Sigma'$, $\Sigma = \langle\langle \mathbb{E}[\texttt{fork}\ \ (e; e')\ ], \tau\rangle + \psi, \mu\rangle$, and $\Sigma' = \langle\langle e_0, \tau'\rangle + \psi + \psi', \mu\rangle$.

To say that concurrent tasks do not interfere (heap accesses), we define the noninterference relation for dynamic effects ($\eta \sharp \eta'$) as follows: reads effects do not conflict with each other; read-write and write-write pairs do not conflict if the location or the field is different. A set of dynamic effects $\varphi = \{\eta_1..\eta_n\}$ do not conflict with another set $\varphi' = \{\eta'_1..\eta'_p\}$ if $\forall i \in \{1..n\}, j \in \{1..p\}$ s.t. $\eta_i \sharp \eta'_j$.

THEOREM 5.5. [Deterministic semantic] *Let $\mu \vdash \diamond$, $\Sigma \hookrightarrow \Sigma'$, and $\Sigma \twoheadrightarrow \psi'$, where $\Sigma = \langle\langle \mathbb{E}[\texttt{fork}\ (e_1; e_2)\ ], \tau\rangle + \psi, \mu\rangle$. For the two tasks $\langle e_1, (id_1, I_1)\rangle \in \psi'$ and $\langle e_2, (id_2, I_2)\rangle \in \psi'$, $\mu \vdash e_1 : \rho_1$, $\mu \vdash e_2 : \rho_2$, if $\rho_1 \sharp \rho_2$, then $\forall n \in \mathbb{N}$ s.t. $dynE(\Sigma', \bullet, \emptyset, n) = (\Sigma_n, \chi, \Upsilon)$, $dynESet(id_1, \chi, \Upsilon) \sharp dynESet(id_2, \chi, \Upsilon)$. Here $\{\eta \in \chi \mid (\eta = (\ldots, id)) \vee (\eta = (\ldots, id') \wedge id' \in desc(id, \Upsilon))\} = dynESet(id, \Upsilon, \chi)$.*

Proof Sketch: The essence of the theorem is that for the 2 concurrent tasks id_1 and id_2, generated by a **fork** expression, if the store is well-formed $\mu \vdash \diamond$, the effect judgments give them effect ρ_1 and ρ_2 respectively and they do not interfere $\rho_1 \sharp \rho_2$, then their dynamic effects ($dynESet(id_1, \chi)$ returns the dynamic traces by the task id_1) do not interfere $dynESet(id_1, \chi, \Upsilon) \sharp dynESet(id_2, \chi, \Upsilon)$. The map Υ records the children set (Section 5.1) of a task *id*. We need Υ because the effect by a child task of *id* should count as *id*'s effect.

Proving the above soundness theorems are non-trivial compared to static effect approaches [13, 33], in which the exact effect of a task is known statically. A technical challenge for proving the soundness of *OpenEffectJ* is that the effects of the concurrent tasks may change due to the **open** effect, i.e., the effect concretization.

5.5 Open References

As discussed in Section 3, other *open* references can be allowed. Here, we discuss how we can extend *OpenEffectJ* to enable *open* variable and parameter. At compiler-time, *OpenEffectJ*'s type-and-effect system will generate an *open* effect **@open** *var m* $\emptyset$ for a method call *var.m*(...). At runtime, the concretization of the *open* effect happens in the `fork` rule. At runtime, if *var* is bound to a location *loc*, the (E-CALL-LOC) rule in Figure 7, takes effect. It transits from the placeholder effect **@open** *var m* $\emptyset$ to the concrete effect ρ_0 (the second item in (E-CALL-LOC)). With this concretized effect, the `fork` rules become more optimistic and all the examples follow directly from this extension.

6. Adding Open Effects to OpenJDK

We have extended the OpenJDK Java compiler to add support for *open* effects. An overview of the compiler is presented in Figure 8.

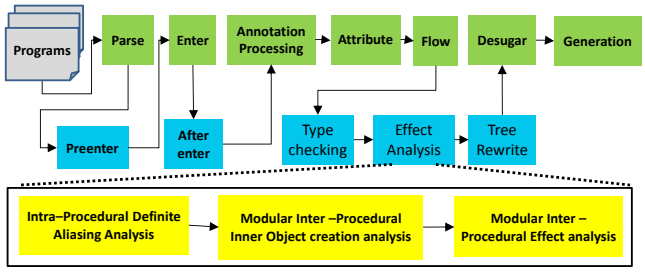

Figure 8. Overview of *OpenEffectJ* Compiler.

Apart from modifications to support the `@open` annotation and the `fork` expression, parsing remains unchanged. Type checking (the Attribute and Flow phases in the OpenJDK compiler) are modified to implement new constraints specified in Section 4. This phase is also extended with an effect analysis, which implements the effect system discussed in Section 4 augmented with modular analyses to further improve precision. These include an intra-procedural definite alias analysis [20], purity analysis [35], and modular inter-procedural analysis that detect temporary objects. This phase attributes each AST node with static effects for each method, which is used by tree rewriting phase to generate code for runtime effect manipulation.

6.1 Effect Storage and Maintenance

Application classes are instrumented to contain dynamic effects. Concrete effects are stored as a static member array, to avoid duplication, and open effects are stored as an instance field array.

We noted in Section 5 that if the effects of an object o changes, the effects of an object, o' which has some *open* field pointing to o, should also be changed. In the semantics, we implemented this change using the function *update*. In the implementation, we maintain a reverse pointer from o to o' for efficiency reasons. This reverse pointer is maintained as a weak reference, which does not prevent o' from being garbage-collected. It is only needed for classes that have open fields. If a class has no open fields, the effect of all of its method will be concrete effects and will not change.

When an *open* field f is assigned a value, concrete effects of the methods in the object that contains f may change. In our example in Section 2.3 when the `mapper` of an instance of class `MapReduce` changed, concrete effect of method `compute` will change also.

We generate a method `cascade` to implement this functionality. The method first checks whether the effect is actually enlarged by this *open* field assignment, i.e. whether it has reached a fixpoint. If so, the algorithm stops propagating the changes (Section 5). Otherwise, it calls the `cascade` method of all its reverse pointers.

7. Comparative Analysis with Related Work

We now compare *OpenEffectJ* with closely related ideas.

7.1 Overview of Closely Related Ideas

Like *OpenEffectJ*, **Synchronization via Scheduling (SVS)** [5] computes conflicts between potentially concurrent tasks right before forking them off. SvS supports a C like language. It compares reachable objects graph (OG) of tasks to determine if they may conflict [31]. Compared to SVS, *OpenEffectJ* supports a full OO language with support for overriding and dynamic dispatch, which makes conflict detection much more challenging [17]. Furthermore, using effects sets instead of reachable OG may be more precise for OO features, e.g., in every example in Section 1 and Section 2, the OG for all the tasks are the same (all of them access the same receiver object of the method call on the *open* references) and thus overlap with each other; therefore, SvS will recommend sequential execution for all of them, whereas *OpenEffectJ* allows parallelism.

In **type, regions and effect-based approaches** [6, 8, 11, 22, 33] programmers specify the footprint (region) of concurrent tasks. By reasoning that two regions are disjoint, programmers conclude that the tasks do not depend on each others. These approaches are pure static, whereas *OpenEffectJ* uses a hybrid approach. This allows greater optimism in exposing safe parallelization opportunities compared to static approaches that also operate within the same constraints. *OpenEffectJ*'s analysis is modular and does not require effect annotations in supertypes. Here by *modular*, we mean that to analyze a piece of code, the analysis requires only the code in question and the interface of static types used in the code.

DPJ framework [6] uses effect parameters [**?**] and effect constraint to reason about the correctness of the client code. Effect constraint is used to restrict the effect of the user-supplied subclass. There are two main differences. First, OpenEffectJ requires no annotations on super classes to restrict overriding subclasses, whereas DPJ does. Second, if a subclass does not refine its superclass specifications DPJ signals compilation error, whereas if a subclass has interfering effects, OpenEffectJ runs relevant tasks serially.

There is a large body of work on **dynamic analysis for concurrency** [7, 16, 18, 38]. In essence, they monitor memory footprints of tasks and signal when conflicts are detected. In contrast, *OpenEffectJ* detects conflicts just before forking off parallel tasks.

Transactional memory [21, 26, 36, 39] optimistically executes tasks concurrently, but monitors memory accesses. It rollbacks side effects when conflicts happen. There are TM-like approaches [4, 15, 41] that provide sequential consistency (DTM) by enforcing a deterministic commit order, instead of rolling back non-deterministically on conflict. In *OpenEffectJ*, state buffering is not needed, because conflicts are detected before parallel code.

In **concurrent revisions** [9, 10] programmers know that tasks conflict on shared objects and annotate these objects. Eash task has a local copy of the objects to avoid data races. In *OpenEffectJ*,

however, when a concurrent library class c is developed, the presence/absence of conflicts may be unknown, because c could be extended with concurrency-unsafe code by clients. *OpenEffectJ* exposes safe, optimistic concurrency for the library.

Gradual Typing [37] and **Hybrid Type Checking** [24] blend the advantages of static and dynamic type checking, whereas *OpenEffectJ* blends the advantages of static and dynamic effect analysis.

7.2 Criteria and Analysis Results

The comparison criteria and the results are summarized below:

Work	SM	OO	DS	IC	DT	OPT
OpenEffectJ	**Yes**	**Yes**	**Yes**	**No**	**Hybrid**	**Fork Point**
DPJ [6, 33]	**Yes**	**Yes**	**Yes**	**Yes**	**Static**	**Static**
Galois [27]	**Yes**	**Yes**	**No**	**No**	**Dynamic**	**Complete**
SvS [5]	**Yes**	**No**	**Yes**	**No**	**Hybrid**	**Fork Point**
Ownership [11, 13]	**Yes**	**Yes**	**Yes**	**No**	**Static**	**Static**
Actor [3]	**No**	**Yes**	**No**	**No**	**Static**	**Static**
TM [21, 26, 36, 39]	**Yes**	**Yes**	**No**	**No**	**Dynamic**	**Complete**
DTM etc. [4, 15, 41]	**Yes**	**Yes**	**Yes**	**No**	**Dynamic**	**Complete**
FastTrack [18], CP [38]	**Yes**	**Yes**	**Yes**	**No**	**Dynamic**	**Complete**
Goldilocks [16]	**Yes**	**Yes**	**Yes**	**No**	**Hybrid**	**Complete**
Revision [9, 10]	**Yes**	**Yes**	**Yes**	**No**	**Dynamic**	**Complete**
X10 [12]	**Yes**	**Yes**	**No**	**No**	**Dynamic**	**Complete**

In **shared memory (SM)** systems, tasks communicate via accessing the shared memory space, while in the distributed memory systems, tasks communicate via messages. X10 is classified as shared memory because places can share heap objects. As the main stream languages (C++, Java and C#) adopt the shared memory and object-oriented (OO) models, it may require more intellectual efforts to use distributed models [28], and/or the non-OO models.

In a programming model with **deterministic semantics (DS)**, a given input will produce the same result. Programmers generally find it easier to reason about deterministic programs. The actor model, due to its asynchronous nature, does not provide determinstic semantics, since the order of the arrival of the messages is arbitrary [28]. Galois targets applications that do not require determinism. Normally, TM has no determinism guarantee due to its nondeterministic commit order, except for deterministic TMs.

By **inheritance constraint (IC)** we mean that the subclass `C` must obey certain rule due to the specification on its superclasses, e.g., `C` has subeffect of its superclasses [6]. It facilitates reasoning, but requires extra efforts developing superclass, especially when subclasses may not be anticipated easily.

The last two columns, **deployment time (DT) and optimism level (OPT)** show when the systems are activated and how optimistic they are in concurrency. A static approach does reasoning at compile time, has least runtime information and is least optimistic. A hybrid analysis, like *OpenEffectJ*, uses static information to facilitate the runtime analysis and is more optimistic than a static one. A dynamic approach reasons about correctness completely at runtime and is the most optimistic. The actor model does not do any static/dynamic analysis, and is put in the static category. Goldilocks is hybrid because it could apply static analysis to reduce runtime overhead, which requires whole program analysis.

7.3 Scope and Applicability of Open Effects

Here, we compare *OpenEffectJ* with static and dynamic approaches for their scope and applicability.

The best scenario for static analyses is when using compile-time knowledge, we could soundly conclude that the tasks either always or never conflict, e.g., if a task c is a consumer of a producer task p, then c should not be executed until p is done; or if two tasks are pure computations. In such cases, a static analysis wins hands down with no runtime overhead. However, if a static analysis makes use of many conservative approximations, because accurate type information is not available, an optimistic approach such as *OpenEffectJ* or TM should be a more desirable model.

The best scenario for dynamic approaches is when the parallel section has alternate paths, e.g., p_1 and p_2, some of which p_1 have data races, but these are not the hot paths in program. The others p_2 have no side effect and are frequently executed. TM will win, because 1) p_1 will indeed be executed, and so all the sound models which make decisions before the parallel section must indicate that it is not safe; and 2) p_2, is more frequently used.

There are at least two scenarios when *OpenEffectJ* outperforms the other two. First is when barriers for memory access can be removed when adequate runtime information is aquired before the parallel section, but not enough at compile time. In the second scenario, there are three tasks a, b and c. Task a and b do not conflict, but both conflict with c. The static approach may sequentialize all of them. *OpenEffectJ* will indicate that c should be run after a and b are done and a and b can be run concurrently. TM will most likely run all of them concurrently and c will be rolled back [5].

8. Conclusion and Future Work

Two reasons severely hinder the safe parallelization of object-oriented libraries: overriding and dynamic dispatch. The developer of a library class, when writing that class, has little knowledge about the behavior and side effects of its clients' code. The side effects of clients' code can affect the correctness of the parallelization of the library code. We have developed a new effect system in *OpenEffectJ* that solves this problem. This effect system employs an optimistic strategy that blends modular static effect analysis with dynamic effect verification. It has many benefits. It is modular and so it allows analysis of libraries and frameworks. It does not require annotations on methods in a supertype to specify upper bounds on the effect of all subtypes. It is more precise compared to a static analysis with same properties, but would have some runtime overhead. It is less precise compared to a pure dynamic analysis, but would detect conflicts before they occur. Thus, state buffering for rollbacks is not necessary and overhead is less. In future, it would be sensible to explore a logical extreme, where every reference is implicitly *open* and a static analysis is used to systematically eliminate `@open` references. Combining effect specifications and *open* effects would also be interesting.

Acknowledgements This work was supported in part by the NSF under grants CCF-08-46059, and CCF-11-17937.

References

[1] JSR-166y Specification Request for Java 7. `http://gee.oswego.edu/dl/concurrency-interest/`.

[2] M. Abadi, C. Flanagan, and S. N. Freund. Types for safe locking: Static race detection for Java. *TOPLAS*, 28:207–255, March 2006.

[3] G. Agha and C. Hewitt. Concurrent programming using actors: Exploiting large-scale parallelism. In *FSTTCS*, pages 19–41. Springer, 1985.

[4] E. D. Berger, T. Yang, T. Liu, and G. Novark. Grace: safe multithreaded programming for c/c++. In *OOPSLA*, pages 81–96, 2009.

[5] M. J. Best, S. Mottishaw, C. Mustard, M. Roth, A. Fedorova, and A. Brownsword. Synchronization via scheduling: techniques for efficiently managing shared state. In *PLDI*, pages 640–652, 2011.

[6] R. L. Bocchino and V. S. Adve. Types, regions, and effects for safe programming with object-oriented parallel frameworks. In *ECOOP*, 2011.

[7] M. D. Bond, K. E. Coons, and K. S. McKinley. Pacer: proportional detection of data races. In *PLDI*, pages 255–268, 2010.

[8] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: preventing data races and deadlocks. In *OOPSLA*, 2002.

[9] S. Burckhardt, A. Baldassin, and D. Leijen. Concurrent programming with revisions and isolation types. In *OOPSLA*, pages 691–707, 2010.

[10] S. Burckhardt, D. Leijen, C. Sadowski, J. Yi, and T. Ball. Two for the price of one: a model for parallel and incremental computation. In *OOPSLA*, 2011.

[11] N. R. Cameron, S. Drossopoulou, J. Noble, and M. J. Smith. Multiple ownership. In *OOPSLA*, pages 441–460, 2007.

[12] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA*, pages 519–538, October 2005.

[13] D. Clarke and S. Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *OOPSLA*, pages 292–310, 2002.

[14] C. Clifton and G. T. Leavens. MiniMAO$_1$: Investigating the semantics of proceed. *SCP*, 63(3):321–374, 2006.

[15] C. Ding, X. Shen, K. Kelsey, C. Tice, R. Huang, and C. Zhang. Software behavior oriented parallelization. In *PLDI*, pages 223–234, 2007.

[16] T. Elmas, S. Qadeer, and S. Tasiran. Goldilocks: a race and transaction-aware Java runtime. In *PLDI*, pages 245–255, 2007.

[17] C. Flanagan and S. N. Freund. Type-based race detection for Java. In *PLDI*, pages 219–232, 2000.

[18] C. Flanagan and S. N. Freund. FastTrack: efficient and precise dynamic race detection. In *PLDI*, pages 121–133, 2009.

[19] M. Flatt, S. Krishnamurthi, and M. Felleisen. A Programmer's Reduction Semantics for Classes and Mixins. In *Formal Syntax and Semantics of Java*, chapter 7, pages 241–269. Springer, 1999.

[20] D. Goyal. An improved intra-procedural may-alias analysis algorithm. Technical report, New York, NY, USA, 1999.

[21] M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. In *ISCA*, pages 289–300, 1993.

[22] I. T. Kassios. Dynamic frames: Support for framing, dependencies and sharing without restrictions. In *FM*, pages 268–283, 2006.

[23] J. Kleinberg and E. Tardos. *Algorithm Design*. Addison Wesley, 2005.

[24] K. Knowles and C. Flanagan. Hybrid type checking. *TOPLAS*, 32: 6:1–6:34, February 2010.

[25] D. Lea. A Java Fork/Join Framework. In *Java Grande*, 2000.

[26] M. Lesani and J. Palsberg. Communicating memory transactions. In *POPL*, pages 157–168, 2011.

[27] M. Kulkarni *et al.* Optimistic parallelism requires abstractions. In *PLDI*, 2007.

[28] P. Mackay. Why has the actor model not succeeded? Technical Report 2, Imperial College, Dept. of Comput., 1997.

[29] D. Marino, A. Singh, T. Millstein, M. Musuvathi, and S. Narayanasamy. Drfx: a simple and efficient memory model for concurrent programming languages. In *PLDI*, pages 351–362, 2010.

[30] M. Naik and A. Aiken. Conditional must not aliasing for static race detection. In *POPL*, pages 327–338, 2007.

[31] F. Pluquet, S. Langerman, and R. Wuyts. Executing code in the past: efficient in-memory object graph versioning. In *OOPSLA*, pages 391–408, 2009.

[32] P. Prabhu, G. Ramalingam, and K. Vaswani. Safe programmable speculative parallelism. In *PLDI*, pages 50–61, 2010.

[33] R. Bocchino *et al.*. A type and effect system for deterministic parallel Java. In *OOPSLA*, pages 97–116, 2009.

[34] M. C. Rinard and P. C. Diniz. Commutativity analysis: a new analysis framework for parallelizing compilers. In *PLDI*, pages 54–67, 1996.

[35] R. D. Salcianu and M. C. Rinard. Purity and side effect analysis for Java programs. In *VMCAI*. Springer-Verlag, 2005.

[36] N. Shavit and D. Touitou. Software transactional memory. In *PODC*, pages 204–213, 1995.

[37] J. Siek and W. Taha. Gradual typing for objects. In *ECOOP*, ECOOP '07, pages 2–27, 2007.

[38] Y. Smaragdakis, J. M. Evans, C. Sadowski, Y. Jaeheon, and C. Flanagan. Sound predictive race detection in polynomial time. In *POPL*, 2012.

[39] T. Shpeisman *et al.* Enforcing isolation and ordering in STM. In *PLDI*, pages 78–88, 2007.

[40] J. W. Voung, R. Jhala, and S. Lerner. RELAY: static race detection on millions of lines of code. In *ESEC-FSE*, pages 205–214, 2007.

[41] A. Welc, S. Jagannathan, and A. Hosking. Safe Futures for Java. In *OOPSLA*, pages 439–453, 2005.

[42] B. P. Wood, L. Ceze, and D. Grossman. Data-race exceptions have benefits beyond the memory model. In *MSPC*, pages 30–36, 2011.

A. Type-and-effect System: Omitted Details

This section presents type-and-effect rules that were omitted in the main text for brevity.

A.1 Type-and-Effect Rules for Declarations

The rules for top-level declarations are fairly standard. Below, the (T-PROGRAM) rule says that the entire program type checks if all the declarations type check and the expression e has any type t and any effect ρ.

$$\frac{\forall decl_i \in \overline{decl} \vdash decl_i : \text{OK} \qquad \vdash e : (t,\rho)}{\vdash \overline{decl}\ e : (t,\rho)} \text{ (T-PROGRAM)}$$

The (T-CLASS) rule says that a class declaration type checks if all the following constraints are satisfied. First, all the newly declared fields are not fields of its super class (this is checked by the auxiliary function *validF*). Next, its super class d is defined in the Class Table (this is checked by the auxiliary function *isClass*). Finally, all the declared methods type check.

$$\frac{\forall field_k \in \overline{field} : validF(field_k, d) \quad isClass(d) \qquad \forall meth_j \in \overline{meth} \vdash meth_j : (t_j, \rho_j) \text{ in } c}{\vdash \texttt{class}\ c\ \texttt{extends}\ d\ \{\overline{field}\ \ \overline{meth}\} : \text{OK}} \text{ (T-CLASS)}$$

The function *validF* and *isClass* check if a field is valid and a class is declared, respectively, which are standard.

$$\frac{CT(c) = \texttt{class}\ c\ \texttt{extends}\ d\ \{field_1 \ldots field_n\ \overline{meth}\} \quad \nexists i \in \{1..n\} \text{ s.t. } field_i = [\texttt{@open}]\ t\ f; \qquad validF(f,d)}{validF(f,c)}$$

$$validF(f, Object) \qquad \frac{\texttt{class}\ c\ \texttt{extends}\ d\ \{\overline{field}\ \overline{meth}\} \in CT}{isClass(c)}$$

$$\frac{isClass(t) \vee (t = \texttt{void})}{isType(t)}$$

A.2 Type-and-Effect Rules for Expressions

The rules for OO expressions are standard, except for the effects in type attributes.

$$\frac{isClass(c)}{\Pi \vdash \texttt{new}\ c() : (c, \emptyset)} \text{ (T-NEW)} \qquad \frac{\Pi(var) = t}{\Pi \vdash var : (t, \emptyset)} \text{ (T-VAR)} \qquad \frac{isType(t)}{\Pi \vdash \texttt{null} : (t, \emptyset)} \text{ (T-NULL)}$$

$$\frac{isClass(c) \qquad \Pi \vdash e_1 : (t_1, \rho) \qquad \Pi, var : c \vdash e_2 : (t_2, \rho') \qquad t_1 <: c}{\Pi \vdash c\ var = e_1; e_2 : (t_2, \rho \cup \rho')} \text{ (T-DEFINE)}$$

$$\frac{\Pi \vdash e_1 : (t_1, \rho) \qquad \Pi \vdash e_2 : (t_2, \rho')}{\Pi \vdash e_1; e_2 : (t_2, \rho \cup \rho')} \text{ (T-SEQ)}$$

The (T-NEW) rule ensures that the class c being instantiated was declared. This expression has empty effect. The (T-VAR) rule checks that *var* is in the environment. The (T-NULL) rule says that the **null** expression could be of any valid type. The declaration expression (T-DEFINE) rule ensures that the initial expression should be a subtype of the type of the new variable. Also, the subsequent expression e_2 types check if the type of the variable is placed in the environment. The (T-SEQUENCE) rule states that the sequence expression has same type as the last expression and its effects are the union of the two expressions. The sequence expression type checks if both left and right expressions type check.

The auxiliary function *typeOfF* (used in the rules in Section 4), uses CT to find the type of a field f, the class in which f is declared and the *open* annotation information, for the input field f.

$$typeOfF(f) = (c,t)$$
where s.t. $CT(c) = \texttt{class}\ c\ \texttt{extends}\ d\{field_1 \ldots field_n\ \overline{meth}\}$
and $\exists i \in \{1..n\}\ ::\ \exists t\ ::\ fieldOf(field_i) = (f,t)$

$$fieldOf(\texttt{@open}\ c\ f) = (f, \texttt{@open}\ c)$$
$$fieldOf(c\ f) = (f,c)$$

B. Dynamic Semantics: Omitted Details

This section presents auxiliary functions that were omitted in Section 5 for brevity.

The *fields* function, used in the (NEW) rule, returns all the fields declared in the class and its super classes (it uses the *fieldOf* function defined in Section A.2).

$$fields(c) = Fs \cup \{f_1 \ldots f_n\}$$
where $CT(c) = \texttt{class}\ c\ \texttt{extends}\ d\ \{field_1 \ldots field_n\ \overline{meth}\}$
and $fields(d) = Fs$ **and** $\forall i \in 1..n\ ::\ fieldOf(field_i) = (f_i, t_i)$

The function *fixPoint*, used in the (SET) rule is shown below. It calls the *update* function in Section 5.2 until the store μ, or more specifically the effects in the store, does not change. The *update* is called on all the loc_i and field f pairs that are pointing to the *loc*, whose effects have been changed. The effects of loc_i are changed by calling the *update* function.

$$fixPoint(\mu, loc, \kappa) = \mu_n \quad \textbf{where } \kappa = \{\langle loc_i, f_i \rangle \mid 1 \le i \le n\}$$
and $update(\mu, loc_1, f_1, loc) = \mu_1$
and $\forall\ i \in \{2..n\}\ ::\ update(\mu, loc_{i-1}, f_{i-1}, loc) = \mu_i$

The function *active* (below) returns the top most task in ψ that can be run. A task is ready to run if all the tasks in its children set are done (evaluated to a single value v).

$$active(\langle e, \tau \rangle + \psi) = \langle e, \tau \rangle + \psi \qquad \textbf{if } intersect(\tau, \psi) = false$$
$$active(\langle e, \tau \rangle + \psi) = active(\psi + \langle e, \tau \rangle) \quad \textbf{if } intersect(\tau, \psi) = true$$

The function *intersect*, used by the *active* function, checks whether there is still any task in the dependent (children) set of the current set, i.e., to check either the dependent set is empty or all the tasks in the dependent set are done and thus deleted from the queue ψ.

$$intersect((id, \emptyset), \psi) = false$$
$$intersect((id, \{id_1, .., id_n\}), \psi) = \bigvee b_i$$
where $\forall\ i \in \{1..n\}\ ::\ inQueue(id_i, \psi) = b_i$

The function *inQueue*, used by the *intersect* function, searches whether there is a task *id* matching the input value *n*.

$$inQueue(n, \bullet) = false$$
$$inQueue(n, \langle e, \langle n, \{id_j \mid j \in \mathbb{N}\}\rangle\rangle + \psi) = true$$
$$inQueue(n, \langle e, \langle n', \{id_j \mid j \in \mathbb{N}\}\rangle\rangle + \psi) = inQueue(n, \psi) \quad \textbf{if } n \neq n'$$

This function gets the tasks from the task queue ψ and matches the input *n* with the *id* of the tasks. If one task matches, the function returns true. Otherwise it continues searching the rest of the tasks in the queue until one of them matches or none of the tasks matches.

C. Proof of Key Properties

We now prove the key properties of *OpenEffectJ*: Effect and Type Preservation, and Determinism. Some of the definitions, descriptions and proof sketchs are also in Section 5.4. We write all these for the sake of clarity.

We have proven the soundness of *OpenEffectJ*'s type system (Section C.4 contains proof that use the standard subject reduction argument [19]). The Effect preservation property is that the dynamic effect (heap accesses) of each concurrent task *id* refines the static effect computed when *id* is forked off. We prove this in Section C.2. We prove the determinism of *OpenEffectJ* programs by showing that the concurrent tasks do not have dynamic effect interference and therefore a well-typed *OpenEffectJ* program produces the same result given the same input in Section C.3. Proving effect soundness is non-trivial compared to static effect approaches [13, 33], in which the exact effect of a task is known statically. A technical challenge for proving the soundness of *OpenEffectJ* is that the effects of the concurrent tasks may change due to the **open** effect, i.e., the effect concretization.

C.1 Preliminary Definitions

We now give some preliminary definitions used in the proofs for *OpenEffectJ*'s properties. A standard approach to show determinism, for multi-tasking systems, is to prove that the heap accesses of the tasks do not interfere [33]. To record the heap accessed for each task, we define dynamic trace χ, which contains a sequence of dynamic effects (heap accesses) by the tasks. Later we will show that the dynamic effects of each task *id* refines the static effect ρ computed when *id* is forked off, s.t. if the static effects do not interfere, the dynamic effects will not interfere [13, 33] (Section C.3).

DEFINITION C.1. *[Dynamic Trace] A dynamic trace (χ) consists of a sequence of dynamic effects ($\overline{\eta}$), where η can be a read effect* $(\mathtt{rd}, loc, f, id)$ *or write effect* $(\mathtt{wt}, loc, f, id)$.

Next, we will introduce a relationship Υ. It records the children tasks of a task *id*. The relationship Υ will be used to prove that the dynamic traces produced by a child task refine the static effect of its parent task *id*.

DEFINITION C.2. *[Relationship] A relationship Υ for tasks is a map* $\{id_j \mapsto I_j\}_{j\in\mathbb{N}}$. *Here id is a task's identity and I is a set of its children tasks' identities.*

The function *dyn*, defined in Figure 9, records the dynamic memory footprint for each task. With it, we can prove that the dynamic effects of each task *id* refines the static effect ρ computed when it is forked off.

In the following, $dyn(\Sigma,\chi,\Upsilon) = dyn(\Sigma',\chi',\Upsilon')$ and $\Sigma \hookrightarrow \Sigma'$.	
Σ	**Side Conditions**
$\langle\langle \mathbb{E}[loc.f], (id,I)\rangle + \psi, \mu\rangle$	$\chi' = \chi + (\mathtt{rd}, loc, f, id), \Upsilon' = \Upsilon$
$(\langle\mathbb{E}[loc.f = v], (id,I)\rangle + \psi, \mu)$	$\chi' = \chi + (\mathtt{wt}, loc, f, id), \Upsilon' = \Upsilon$
$(\langle\mathbb{E}[\mathtt{fork}\ (e_0; e_1)\,], (id,I)\rangle + \psi, \mu)$	$\Sigma' = (\langle e', (id,I')\rangle + \psi', \mu), \chi' = \chi$ $\Upsilon' = \{id \mapsto (\Upsilon(id) \cup I')\} \oplus \Upsilon$
Other cases	$\chi' = \chi, \Upsilon' = \Upsilon$

Figure 9. Dynamic Effect function *dyn*.

Here, we define what it means by dynamic effects refine the static effects, s.t. the non-interference of the static effects implies the non-interference of the dynamic effects.

DEFINITION C.3. *[Static effect inclusion] An effect ε is included in an effect set $\rho = \{\varepsilon_i | 1 \le i \le n\}$, where $(\bot) \notin \rho$, written $\varepsilon \in \rho$, if either: $\exists\ \varepsilon_i$ s.t. $\varepsilon = \varepsilon_i$; or $\exists\ \varepsilon_i$ s.t. $(\varepsilon_i = \mathtt{@open}\ f\ m\ \rho') \wedge (\rho' = \{\varepsilon'_j | 1 \le j \le n'\}) \wedge (\varepsilon = \varepsilon'_j)$.*

This definition says that an effect ε is included in an effect set ρ if it is one of the elements in ρ; or there is an *open* effect $\mathtt{@open}\ f\ m\ \rho'$ in ρ and ε is an element of ρ'.

DEFINITION C.4. *[Dynamic effect refines static effect] A dynamic effect η refines a static effect ρ, where $(\bot) \notin \rho$, written $\eta \propto \rho$, if either $\eta = (\mathtt{rd}, loc, f, id) \wedge (\mathtt{read}\ f) \in \rho$; or $\eta = (\mathtt{wt}, loc, f, id) \wedge (\mathtt{write}\ f) \in \rho$.*

In Section C.2, we will show that during the evaluation, for concurrent tasks, the effect ρ of an expression e, is refined by the effect ρ' of its subsequent expression e', i.e., if $\langle\langle e, (id,I)\rangle + \psi, \mu\rangle \hookrightarrow \langle\langle e', (id,I')\rangle + \psi', \mu'\rangle$, $\mu \vdash e : \rho$ and $\mu' \vdash e' : \rho'$, then $\rho' \subseteq \rho$. This guarantees that the static effect, computed when a task is forked off, is a sound approximation of the effects of all subsequent expressions. Here we define how an effect ρ' refines another effect ρ.

DEFINITION C.5. *[Static effect refinement] An effect set ρ' refines another effect set ρ if $\rho' \subseteq \rho \wedge (\bot) \notin \rho$.*

During the evaluation of concurrent tasks, the store keeps changing, and we want to ensure that the same expression has the same static effect in the presence of task interleaving (Theorem C.2). To do so, we define effect equivalent stores (Definition C.6) and prove that these stores give the same effects for a same expression.

DEFINITION C.6. *[Effect equivalent stores] Two stores μ and μ' are effect equivalent, written $\mu \cong \mu'$, if both conditions hold: $dom(\mu) \subseteq dom(\mu')$; and $\forall loc$ if $\mu(loc) = [c.F.E]$, then $\mu'(loc) = [c.F'.E]$, for some F'.*

This definition says that two stores are effect equivalent if they have the same effects for all common locations.

Except for the method call expression, proving that an expression has static effects that are refined by their subsequent expression is standard [13, 33]. The novelty is that effects for method calls are new in this work and *OpenEffectJ* needs to maintain proper effects for methods (Definition C.7 and Definition C.8). To prove that a method call on *open* field has static effects that are refined by their subsequent expression, we introduce well-formed object.

DEFINITION C.7. *[Well-formed object] An object record $o = [c.F.E]$ is a well-formed object in μ, written $\mu \vdash o$, if for all* open *effect $\mathtt{@open}\ f\ m\ \rho_0 \in \rho \in rng(E)$, either $(F(f) = loc) \wedge (\mu(loc) = [c'.F'.E']) \wedge (E'(m) \subseteq \rho_0)$; or $(F(f) = \mathtt{null}) \wedge (\rho_0 = \emptyset)$.*

This definition says that an object record is well-formed, if all of its *open* effect ($\mathtt{@open}\ f\ m\ \rho$) is supereffect ($\supseteq$) of the effect of the method m of the object the field f is pointing to.

To prove that a method call on a location *loc* have static effects that are refined by their subsequent expression, we introduce well-formed location (Definition C.8).

DEFINITION C.8. *[Well-formed location] A location loc is well-formed in μ, written $\mu \vdash loc$, if either $\mu(loc) = [c.F.E]$, $\forall m \in dom(E)$ s.t. findMeth(c,m) $= (c', t, m(\overline{t\ var})\{e\}, \rho') \wedge \mu \vdash [loc/\mathtt{this}]e : \rho$, then $\rho \subseteq E(m)$; or $\mu(loc) = \mathtt{null}$.*

A location *loc* is well-formed in a store μ, if the effect, of each method m of the object *loc* is pointing to, is supereffect ($\supseteq$) of the effect given by the effect judgment of the body e of the method m.

Finally, to prove the effect preservation theorem (Theorem C.10), we need to prove an invariant of any *OpenEffectJ* program, i.e., the store is well-formed (Definition C.9). With a well-formed store, it is ready to show that the effect of a method call has expression that is refined by its subsequent expression.

DEFINITION C.9. *[Well-formed store] A store μ is well-formed, written $\mu \vdash \diamond$, if $\forall o \in rng(\mu)$ s.t. $\mu \vdash o$ and $\forall loc \in dom(\mu)$ s.t. $\mu \vdash loc$.*

The definition says that the store is well-formed, if all the locations and object records are well-formed.

C.2 Effect Preservation

In this section, we prove *OpenEffectJ*'s effect preservation property. This involves three invariants during the evaluation of the concurrent tasks. First, the effect ρ produced by an expression e is refined by the effect ρ', by its subsequence expression e' ($\rho' \subseteq \rho$); and second, the dynamic effect η, produced by the reduction, if any, refines ρ ($\eta \propto \rho$); finally, the store remains effect equivalent ($\mu \cong \mu'$).

THEOREM C.10. [Effect preservation] *Let the program configuration* $\Sigma = \langle\langle e, (id, I)\rangle + \psi, \mu\rangle$. *If it transits to another configuration* $\Sigma \hookrightarrow \langle\langle e', (id, I')\rangle + \psi', \mu'\rangle$, *the store is well-formed* $\mu \vdash \diamond$, $\mu \vdash e : \rho$, *and* $(\bot) \notin \rho$, *then there is some* ρ', χ *s.t.*

(a) $\mu \cong \mu'$;
(b) $\mu' \vdash e' : \rho'$, *and* $\rho' \subseteq \rho$;
(c) $(dyn(\Sigma, \chi, \Upsilon) = (\Sigma', \chi + \eta, \Upsilon')) \Rightarrow (\eta \propto \rho)$.

Proof: The proof is by cases on the reduction step applied. We first state two useful lemmas.

C.2.1 Replacement with Subeffect

The following lemma says that two effect equivalent stores give the same effect ρ to the same expression e. Because we will prove that during the concurrent execution of the tasks, all the stores are effect equivalent $\mu \cong \mu'$. Therefore, it suffices to prove that the effect of the subsequent expression e' refines the effect of the expression e, given by effect equivalent stores.

LEMMA C.11. [Stationary effect] *Let e be an expression,* μ *and* μ' *two stores s.t.* $\mu \cong \mu'$. *If* $\mu \vdash e : \rho$ *and* $(\bot) \notin \rho$, *then* $\mu' \vdash e : \rho$.

Proof: Proof is by induction on the structure of the expression e. We prove it case by case on the rule used to generate the effect ρ. In each case we show that $\mu \vdash e : \rho$ implies that $\mu' \vdash e : \rho$, and thus the claim holds by the induction hypothesis (IH). The base cases include (NEW), (NULL), (LOC), and (VAR). These cases are obvious: $\rho' = \rho = \emptyset$. The remaining cases cover the induction step. The IH is that the claim of the lemma holds for all sub-derivations of the derivation being considered. The case for (SET-OPEN) and (CALL) hold because in these cases $\bot \in \rho$.

The cases for (DEFINE), (SEQUENCE), (GET), (SET), (YIELD), and (FORK) follow directly from the induction hypothesis. We show the case for (FORK) and other cases are similar.

(FORK) The last type derivation step is:

$$\frac{\mu \vdash e_0 : \rho_0 \qquad \mu \vdash e_1 : \rho_1}{\mu \vdash \texttt{fork}\ (e_0; e_1) : \rho_0 \cup \rho_1} \qquad \frac{\mu' \vdash e_0 : \rho_0' \qquad \mu' \vdash e_1 : \rho_1'}{\mu' \vdash \texttt{fork}\ (e_0; e_1) : \rho_0' \cup \rho_1'}$$

By the IH, $\rho_0' = \rho_0$ and $\rho_1' = \rho_1$. Therefore $\rho' = \rho_0' \cup \rho_1' = \rho_0 \cup \rho_1 = \rho$, and the claim holds.

(CALL-OPEN) Here $e = loc.f.m(e_1, \ldots, e_n)$. The last type derivation step has the following form:

$$\frac{\begin{array}{c}\mu(loc) = [c.F.E] \qquad \mu \vdash loc.f : \rho_0 \\ E = \{m_i \mapsto \rho_i \| 1 \le i \le n\} \\ \exists i \text{ s.t. } (\exists \varepsilon = \texttt{@open}\ f\ m\ \rho'\ \in \rho_i) \\ typeOfF(f) = (c, \texttt{@open}\ c_0) \\ (\forall\, i \in \{1..n\}\ ::\ \mu \vdash e_i : \rho_i)\end{array}}{\mu \vdash loc.f.m(\overline{e}) : \{\texttt{open}\ f\ m\ \rho'\} \cup \bigcup_{i=0}^{n} \rho_i} \qquad \frac{\begin{array}{c}\mu'(loc) = [c.F'.E] \qquad \mu' \vdash loc.f : \rho_0' \\ E = \{m_i \mapsto \rho_i \| 1 \le i \le n\} \\ \exists i \text{ s.t. } (\exists \varepsilon = \texttt{@open}\ f\ m\ \rho'\ \in \rho_i) \\ typeOfF(f) = (c, \texttt{@open}\ c_0) \\ (\forall\, i \in \{1..n\}\ ::\ \mu' \vdash e_i : \rho_i')\end{array}}{\mu' \vdash loc.f.m(\overline{e}) : \{\texttt{open}\ f\ m\ \rho'\} \cup \bigcup_{i=0}^{n} \rho_i'}$$

Clearly, $\rho_0 = \rho_0' = \{\texttt{read}\ f\}$. Since $\mu \cong \mu'$, the effect maps E are the same. By the IH, $\forall\ i \in \{1..n\} :: \rho_i' = \rho_i$. Thus $\rho' = \{\texttt{open}\ f\ m\ \rho'\} \cup \bigcup_{i=0}^{n} \rho_i' = \{\texttt{open}\ f\ m\ \rho'\} \cup \bigcup_{i=0}^{n} \rho_i = \rho$, and the claim holds.

(CALL-LOC) Here $e = loc.m(e_1, \ldots, e_n)$. The last type derivation step has the following form:

$$\frac{\mu(loc) = [c.F.E] \qquad E(m) = \rho_0 \qquad (\forall\, i \in \{1..n\}\ ::\ \mu \vdash e_i : \rho_i)}{\mu \vdash loc.m(\overline{e}) : \bigcup_{i=0}^{n} \rho_i} \qquad \frac{\mu'(loc) = [c.F'.E] \qquad E(m) = \rho_0' \qquad (\forall\, i \in \{1..n\}\ ::\ \mu' \vdash e_i : \rho_i')}{\mu' \vdash loc.m(\overline{e}) : \bigcup_{i=0}^{n} \rho_i'}$$

Since $\mu \cong \mu'$, the effect maps E are the same and $\rho_0 = \rho_0'$. By the induction hypothesis, $\forall\, i \in \{1..n\} :: \rho_i' = \rho_i$. Thus $\rho' = \bigcup_{i=0}^{n} \rho_i' = \bigcup_{i=0}^{n} \rho_i = \rho$, and the claim holds.

Thus, for all possible derivations of $\mu \vdash e : \rho$ and $\mu' \vdash e : \rho'$, we see that $\rho' = \rho$. ∎

The following lemma says that given two effect equivalent stores, and the same evaluation context, if the effect of the subsequent expression e' refines the original expression e, then the effect of the entire subsequent expression $\mathbb{E}[e']$ refines the entire original expression $\mathbb{E}[e]$. With this lemma, it suffices to show that the effect of the subsequent subexpression e' refines the origianl subexpression e.

LEMMA C.12. [Replacement with subeffect] *If* $\mu \vdash \diamond$, $\Sigma \hookrightarrow \Sigma'$, $\Sigma = \langle\langle \mathbb{E}[e], (id, I)\rangle + \psi, \mu\rangle$, $\Sigma' = \langle\langle \mathbb{E}[e'], (id, I')\rangle + \psi', \mu'\rangle$, $\mu \vdash \mathbb{E}[e] : \rho$, $\mu \vdash e : \rho_0$, $\mu \vdash e' : \rho_1$, $\mu \cong \mu'$, *and* $\rho_1 \subseteq \rho_0$, *then* $\mu \vdash \mathbb{E}[e'] : \rho' \wedge \rho' \subseteq \rho$.

Proof: Proof is by induction on the size of the evaluation context $\mathbb{E}$. Size of the $\mathbb{E}$ refers to the number of recursive applications of the syntactic rules necessary to create $\mathbb{E}$. In the base case, $\mathbb{E}$ has size zero, $\mathbb{E} = -$, and $\rho' = \rho_1 \subseteq \rho_0 = \rho$. For the induction step we divide the evaluation context into two parts such that $\mathbb{E}[e_1] = \mathbb{E}_1[\mathbb{E}_2[e_2]]$, and $\mathbb{E}_2$ has size one. The induction hypothesis (IH) is that the lemma holds for all evaluation contexts, which is smaller than the one ($\mathbb{E}_1$) considered in the induction step. We prove it case by case on the rule used to generate $\mathbb{E}_2$. In each case we show that $\mu \vdash \mathbb{E}_2[e] : \rho$ implies that $\mu' \vdash \mathbb{E}_2[e'] : \rho'$, for some $\rho' \subseteq \rho$, and thus the claim holds by the IH.

The cases for (E-GET), (E-DEFINE), and (E-SEQ) follow directly from the induction hypothesis.

The cases for (E-SET-OPEN) and (E-CALL) hold because in these cases $\bot \in \rho$. $\bot$ is the maximum, any effect $\rho' \subseteq \bot$.

Case $-.f = e_2$ The last step for $\mathbb{E}_2[e]$ should be (E-SET):

$$\frac{\mu \vdash e : \rho_0 \qquad typeOfF(f) = (c, t) \qquad \mu \vdash e_2 : \rho_2}{\mu \vdash \mathbb{E}_2[e] : \rho_0 \cup \rho_2 \cup \{\texttt{write}\ f\}}$$

By the definition of field lookup, $typeOfF(f)$ remains unchanged, i.e. $typeOfF(f) = (c, t)$. Thus, by (E-SET), $\mu' \vdash \mathbb{E}_2[e'] : \rho_1 \cup \rho_2 \cup \{\texttt{write}\ f\}$;

Case $v_0.f = -$ The last step for $\mathbb{E}_2[e]$ should be (E-SET):

$$\frac{typeOfF(f) = (c, t) \qquad \mu \vdash e : \rho_0}{\mu \vdash \mathbb{E}_2[e] : (u, \rho_0 \cup \{\texttt{write}\ f\})}$$

So $\mu' \vdash \mathbb{E}_2[e'] : \rho_1 \cup \{\texttt{write}\ f\}$;

Case $-.m(e_1, \ldots, e_n)$ The last step in the effect derivation for $\mathbb{E}_2[e]$ should be (E-CALL-OPEN): $e = loc.f$

$$\frac{\begin{array}{c}e = loc.f \qquad \mu(loc) = [c.F.E] \qquad E = \{m_i \mapsto \rho_i \| 1 \le i \le n\} \\ \exists i \text{ s.t. } (\exists \varepsilon = \texttt{@open}\ f\ m\ \rho''\ \in \rho_i) \qquad typeOfF(f) = (c, \texttt{@open}\ c_0) \\ \mu \vdash loc.f : \rho_0' \qquad (\forall\, i \in \{1..n\}\ ::\ \mu \vdash e_i : \rho_i')\end{array}}{\Pi \vdash \mathbb{E}_2[e] : \{\texttt{open}\ f\ m\ \rho''\} \cup \bigcup_{i=0}^{n} \rho_i'}$$

By (GET), $e' = loc'$:

$$\frac{e' = loc' \qquad \mu(loc') = [c'.F'.E'] \qquad E(m) = \rho_0'' \qquad (\forall\, i \in \{1..n\}\ ::\ \mu \vdash e_i : \rho_i'')}{\Pi \vdash \mathbb{E}_2[e'] : \bigcup_{i=0}^{n} \rho_i''}$$

Because $\mu \vdash \diamond$, by Definition C.9 and Definition C.7, we have $\rho_0'' \subseteq \rho''$. $\forall i \in \{1..n\} e_i$ does not change, thus $\rho_i' = \rho_i''$. Therefore, the claim holds.

Case $loc.m(v_1,\ldots,v_{p-1},-,e_{p+1},\ldots,e_n)$ Here $p \in \{1..n\}$. The last step for $\mathbb{E}_2[e]$ must be (E-CALL-LOC):

$$\frac{\mu(loc)=[c.F.E] \quad E(m)=\rho'' \quad \mu(e)=\rho_0 \quad (\forall\, i \in \{(p+1)..n\} \;::\; \mu \vdash e_i : \rho_i'')}{\mu \vdash \mathbb{E}_2[e] : \rho_0 \cup \rho'' \cup \bigcup_{i=(p+1)}^{n} \rho_i}$$

By (E-CALL-LOC), $\mu' \vdash \mathbb{E}_2[e'] : \rho_1 \cup \rho'' \cup \bigcup_{i=(p+1)}^{n} \rho_i$.

Using the lemmas To prove Theorem C.2, in each reduction case, let $e = \mathbb{E}[e_0]$, $e' = \mathbb{E}[e_1]$, $\mu \vdash e_0 : \rho_0$ and $\mu' \vdash e_1 : \rho_1$. Given that (a) $\mu \cong \mu'$, by Lemma C.12 and Lemma C.11, to prove (b), it suffices to prove $\rho_1 \subseteq \rho_0$. We divide the cases into 3 categories: in the first category, some variables (var) will be replaced by actual values (v), in Section C.2.2; the cases, in the second category, access the store, in Section C.2.3; and the other cases are listed right below.

Here all the rules leave no dynamic trace, and (c) holds. All the cases, other than the New Object, do not change the store, i.e., $\mu' = \mu$, therefore $\mu \cong \mu'$ and (a) holds.

New Object: Here $e = \mathbb{E}[new\ c()]$, $e' = \mathbb{E}[\texttt{yield}\ loc]$, where $loc \notin dom(\mu)$, $\mu' = \{loc \mapsto [c.\{f \mapsto \texttt{null} \mid f \in fields(c)\}.\{m \mapsto \rho \in methE(c)\}]\} \oplus \mu$. Because this rule does not change any object, $\mu \cong \mu'$. Also $\mu \vdash \texttt{new}\ c() : \emptyset$ and $\mu \vdash \texttt{yield}\ loc : \emptyset$, and (b) holds.

Yield: Here $e = \mathbb{E}[\texttt{yield}\ e_1]$, $e' = \mathbb{E}[e_1]$. Notice that *id* remains unchanged in the statement of the theorem, i.e., the reduction step does not switch control to other task. Therefore $\mu' = \mu$, $\mu \cong \mu'$ and the claim holds.

Fork Sequential: Here $e = \mathbb{E}[\texttt{fork}\ (e_0';e_1')]$, $e' = \mathbb{E}[\texttt{yield}\ e_2']$ and $e_2' = e_0';e_1';\texttt{null}$. Let $\mu \vdash e_0' : \rho_0$, and $\mu \vdash e_1' : \rho_1$. By (E-FORK), $\mu \vdash \texttt{fork}\ (e_0';e_1') : \rho_0 \cup \rho_1$. We have $\mu \vdash e_0';e_1';\texttt{null} : \rho_0 \cup \rho_1$. Therefore, (b) holds.

Fork Parallel: Here $e = \mathbb{E}[\texttt{fork}\ (e_0';e_1')]$, and $e' = \mathbb{E}[\texttt{yield null}]$. Since $\mu \vdash \texttt{null} : \emptyset$, (b) holds.

C.2.2 Substituting Variables with Values

Here all the rules leave no dynamic trace, and (c) holds. Neither do they change the store, i.e., $\mu = \mu'$ and $\mu \cong \mu'$, thus (b) holds. We state a lemma for substituting the variables var for the actual values v, which indicates that the static effect ρ' after the substitution refines the one before the substitution ρ. This lemma is useful for method calls and definitions, where parameters and local variables, respectively, will be substituted by values.

LEMMA C.13. [Substitution effect] *If $\mu \vdash e : \rho$, then there is some ρ', such that $\mu \vdash [v_1/var_1,\ldots,v_n/var_n]e : \rho'$, for all values v_i and free variables var_i, and $\rho' \subseteq \rho$.*

Proof: To simplify the notations, let $[\overline{v/var}] = [v_1/var_1,...,v_n/var_n]$. We prove it by structural induction on the derivation of $\mu \vdash e : \rho$ and by cases, based on the last step in that derivation. The base cases include (E-NEW), (E-NULL), (E-LOC), and (E-VAR). The first three of these cases are obvious: e has no variables, $\rho' = \rho = \emptyset$. In the (E-VAR) case, $\mu \vdash v : \emptyset$ and $\mu \vdash var = \emptyset$. Thus, it holds.

The remaining cases cover the induction step. The induction hypothesis (IH) is that the claim of the lemma holds for all sub-derivations of the derivation being considered.

The cases for (E-GET), (E-DEFINE), (E-YIELD), and (E-SEQ) follow directly from the induction hypothesis.

The case for (E-SET-OPEN) and (E-CALL) hold because in these cases $\bot \in \rho$. $\bot$ is the maximum, any effect $\rho' \subseteq \bot$.

(E-CALL-OPEN) Here $e = loc.f.m(e_1,\ldots,e_n)$. The last effect derivation step has the following form:

$$\frac{\begin{array}{c}\mu(loc)=[c.F.E] \quad E=\{m_i \mapsto \rho_i \| 1 \le i \le n\} \\ \exists i \text{ s.t. } (\exists \varepsilon = \texttt{@open}\ f\ m\ \rho'' \text{ s.t. } \varepsilon \in \rho_i) \quad typeOfF(f) = (d, \texttt{@open}\ c_0) \\ \mu \vdash loc.f : \rho_0 \quad (\forall\, i \in \{1..n\} \;::\; \mu \vdash e_i : \rho_i)\end{array}}{\mu \vdash loc.f.m(e_1,\ldots,e_n) : \{\texttt{open}\ f\ m\ \rho''\} \cup \bigcup_{i=0}^{n} \rho_i}$$

Let $e_i' = [\overline{v/var}]e_i$ for $i \in \{1..n\}$, $[\overline{v/var}]e = loc.f.m(\overline{e'})$. We show that $\mu \vdash [\overline{v/var}]e : \{\texttt{open}\ f\ m\ \rho''\} \cup \bigcup_{i=0}^{n} \rho_i'$, where $\forall i \in \{0..n\}$ $\rho_i' \subseteq \rho_i$. Because $loc.f$ has no free variable, $\rho_0' = \rho_0$ and $\{\texttt{open}\ f\ m\ \rho''\}$ are unchanged. Also by IH $\forall i \in \{1..n\} \;::\; \mu \vdash e_i' : \rho_i'$ and $\rho_i' \subseteq \rho_i$. Thus the claim holds.

(E-CALL-LOC) Here $e = loc.m(\overline{e})$. The last step is:

$$\frac{\mu(loc)=[c.F.E] \quad E(m)=\rho_0 \quad (\forall\, i \in \{1..n\} \;::\; \mu \vdash e_i : \rho_i)}{\mu \vdash loc.m(e_1,\ldots,e_n) : \bigcup_{i=0}^{n} \rho_i}$$

Let $e_i' = [\overline{v/var}]e_i$ for $i \in \{1..n\}$, then $[\overline{v/var}]e = loc.m(\overline{e'})$. We show that $\mu \vdash [\overline{v/var}]e : \bigcup_{i=0}^{n} \rho_i'$, where $\forall i \in \{0..n\}$ $\rho_i' \subseteq \rho_i$. Clearly, $\rho_0' = \rho_0$. By IH $\forall i \in \{1..n\} :: \mu \vdash e_i' : \rho_i'$ and $\rho_i' \subseteq \rho_i$.

(E-SET) Here $e = e_0.f = e_1$. The last derivation step is:

$$\frac{\mu \vdash e_0 : \rho_0 \quad typeOfF(f) = (c,t) \quad \mu \vdash e_1 : \rho_1}{\mu \vdash e_0.f = e_1 : \rho_0 \cup \rho_1 \cup \{\texttt{write}\ f\}}$$

Now $[\overline{v/var}]e = ([\overline{v/var}]e_0.f = [\overline{v/var}]e_1)$. By IH, $\mu \vdash [\overline{v/var}]e_0 : \rho_0'$, and $\mu \vdash [\overline{v/var}]e_1 : \rho_1'$, where $\rho_0' \subseteq \rho_0$ and $\rho_1' \subseteq \rho_1$. By the definition of *typeOfF*, the result of $typeOfF(f)$ remains unchanged, i.e. $typeOfF(f) = (c,t)$. Therefore $\Pi \vdash [\overline{v/var}]e : \rho_0' \cup \rho_1' \cup \{\texttt{write}\ f\}$, and it holds.

(E-FORK) The last effect derivation step is:

$$\frac{\mu \vdash e_0 : \rho_0 \quad \mu \vdash e_1 : \rho_1}{\mu \vdash \texttt{fork}\ (e_0;e_1) : \rho_0 \cup \rho_1}$$

Now $[\overline{v/var}]e = \texttt{fork}\ ([\overline{v/var}]e_0; [\overline{v/var}]e_1)$. By IH, $\mu \vdash [\overline{v/var}]e_0 : \rho_0'$, and $\mu \vdash [\overline{v/var}]e_1 : \rho_1'$, where $\rho_0' \subseteq \rho_0$, and $\rho_1' \subseteq \rho_1$. Therefore, $\mu \vdash [\overline{v/var}]e : \rho_0' \cup \rho_1'$, and the claim holds.

Thus, for all possible derivations of $\mu \vdash e : \rho$ we see that $\mu \vdash [\overline{v/var}]e : \rho'$ for some $\rho' \subseteq \rho$. ∎

Using the lemma We now present the case for method call and local declaration.

Method Call: Here $e = \mathbb{E}[loc.m(\overline{v})]$, $(u', t_m, m(\overline{t\ var})\{e_2\}, \rho_m) = findMeth(u,m)$, $e' = \mathbb{E}[\texttt{yield}\ e_1]$, $e_1 = [loc/\texttt{this}, \overline{v/var}]e_2$, $\mu(loc) = [u.F.E]$. Let $\mu \vdash loc.m(\overline{v}) : \rho_0$, i.e., $E(m) = \rho_0$. Let $e_3 = [loc/\texttt{this}]e_2$, $\mu \vdash e_3 : \rho_3$ and $\mu \vdash e_1 : \rho_1$. By Lemma C.13, $\rho_1 \subseteq \rho_3$. By $\mu \vdash \diamond$, Definition C.8 and Definition C.9, $\rho_3 \subseteq \rho_0$, thus $\rho_1 \subseteq \rho_0$.

Local Declaration: Here $e = \mathbb{E}[t\ var = v; e_1]$, and $e' = \mathbb{E}[\texttt{yield}\ e_1']$, where $e_1' = [v/var]e_1$. Let $\mu \vdash e_1 : \rho_0$, by (E-DEFINE), $\mu \vdash t\ var = v; e_1 : \rho_0$. $\mu \vdash [v/var]e_1 : \rho_1$, for some $\rho_1 \subseteq \rho_0$, by Lemma C.13.

C.2.3 Fields Access

In this subsection, we first state a lemma for the effect relationship between an expression and its subexpression.

The following lemma says that the effect ρ of subexpression e is a subset $\subseteq$ of the effect ρ' of its entire expression $\mathbb{E}[e]$.

LEMMA C.14. [Subexpression effect containment] *If $\mu \vdash e : \rho$ and $\mu \vdash \mathbb{E}[e] : \rho'$, then $\rho \subseteq \rho'$.*

Proof: By the effect rule for each expression, the effect of any direct subexpression is a subset of the entire expression.

Using the lemma We now prove cases for field accesses.

Field Get: Here $e = \mathbb{E}[loc.f]$, $e' = \mathbb{E}[\texttt{yield } v]$, where $\mu(loc) = [u.F.E]$, $F(f) = v$, $\mu' = \mu$ and $\mu \cong \mu'$. Because $\mu \vdash loc.f : \{\texttt{read } f\}$, and $\mu' \vdash \texttt{yield } v : \emptyset$, (b) holds. Finally, $\eta = (\texttt{rd}, loc, f, id)$, and $\eta \propto \{\texttt{read } f\} \subseteq \rho$, by Lemma C.14.

Field Set: Here $e = \mathbb{E}[loc.f = v]$, $e' = \mathbb{E}[\texttt{yield } v]$, $\mu' = \mu \oplus (loc \mapsto o)$, and $o = [u.F \oplus (f \mapsto v).E]$, where $\mu(loc) = [u.F.E]$ and $typeOfF(f) = (c,t)$ for some t. The field is not an *open* field, and by the function *update*, it does not update any effect, and $\mu \cong \mu'$. To see $\mu \vdash \mathbb{E}[v] : \rho' \subseteq \rho$, we have $\mu \vdash loc.f = v : \{\texttt{write } f\}$, and $\mu \vdash v : \emptyset$, thus $\rho' \subseteq \rho$. Finally, $\eta = (\texttt{wt}, loc, f, id)$, and $\eta \propto \{\texttt{write } f\} \subseteq \rho$, by Lemma C.14.

Field Set Open: Here $e = \mathbb{E}[loc.f = v]$, $e' = \mathbb{E}[\texttt{yield } v]$, where $\mu_0 = \mu \oplus (loc \mapsto [c.(F \oplus (f \mapsto v)).E])$, and $\mu' = update(\mu_0, loc, f, v)$. Impossible, since $(\bot) \notin \rho$. ∎

C.3 Deterministic Semantics

The goal of this section is to show that the tasks generated by a **fork** expression do not interfere (Theorem C.25) and therefore *OpenEffectJ* produces deterministic results. First we show that the expressions, in later configurations of a task, refine the expressions in earlier configurations [33]. We define multistep reduction (Definition C.15), which relates the later configurations with earlier configurations. To reason about the interaction between the two concurrent tasks, we define the newly generated queue, in Definition C.17, which is formed by these two tasks. To prove that the tasks do not have heap interference, we introduce the accumulated dynamic effect in Definition C.16 (accumulated heap effects) and show that these accumulated effects, produced by a concurrent task *id* and all its descendants (Definition C.18), refine the static effect computed when *id* is forked off.

DEFINITION C.15. *[Multiple reduction steps] We write $\Sigma \hookrightarrow^0 \Sigma'$ if $\Sigma \hookrightarrow \Sigma'$, and $\Sigma \hookrightarrow^n \Sigma'$ if $(\Sigma \hookrightarrow^{n-1} \Sigma_0) \wedge (\Sigma_0 \hookrightarrow \Sigma')$. We write $\Sigma \hookrightarrow^* \Sigma'$ if $\exists n \geq 0$ s.t. $\Sigma \hookrightarrow^n \Sigma'$.*

DEFINITION C.16. *[Multi step effect] The multi steps effect function dynE is: $dynE(\Sigma, \chi, \Upsilon, n) = dynE(\Sigma', \chi', \Upsilon', n-1)$, if $dyn(\Sigma, \chi, \Upsilon) = (\Sigma', \chi', \Upsilon')$ and $dynE(\Sigma, \chi, \Upsilon, 0) = dyn(\Sigma, \chi, \Upsilon')$.*

Here, the function *dyn* produces the dynamic effect and the relationship Υ for an one step transition from a configuration to another. The function *dynE* accumulates the dynamic effects and the relationship for n transistion steps.

DEFINITION C.17. *[Queue for parallel fork] We said that ψ' is a queue for parallel fork tasks and we write $\Sigma \twoheadrightarrow \psi'$, if $\Sigma \hookrightarrow \Sigma'$, $\Sigma = \langle\langle \mathbb{E}[\textbf{fork } (e; e')], \tau\rangle + \psi, \mu\rangle$, and $\Sigma' = \langle\langle e_0, \tau'\rangle + \psi + \psi', \mu\rangle$.*

The queue ψ' for parallel fork contains the two concurrent tasks generated by the **fork** expression.

To say that concurrent tasks do not interfere (heap accesses), we define the noninterference relation for dynamic effects ($\eta \sharp \eta'$) as follows: the noninterference relation is symmetric; reads effects do not conflict with each other; read-write and write-write pairs do not conflict if the location or the field is different. A set of dynamic effects $\varphi = \{\eta_1 .. \eta_n\}$ do not conflict with another set $\varphi' = \{\eta'_1 .. \eta'_p\}$ if $\forall i \in \{1..n\}, j \in \{1..p\}$ s.t. $\eta_i \sharp \eta'_j$.

DEFINITION C.18. *[Descendant] A task id' is a descendant of a task id, with Υ, written as $id' \leqslant_\Upsilon id$, if $id' \in desc(id, \Upsilon)$. Here, $desc(id, \Upsilon) = I \cup \bigcup_{i=0}^{n} I_i$, $\Upsilon(id) = I = \{id_0, \ldots, id_n\}$, and $\forall id_i \in I, desc(id_i, \Upsilon) = I_i$.*

The descendant is a recursive relationship. The descendant of a task id, includes the children tasks id' of id and all the descendant of id'.

To show that the effect preservation of the subsequent expressions, we need to ensure that the store is well-formed throughout the program execution (Lemma C.20). To facilitate the description, we introduce the initial configuration $\Sigma_\star$ that starts the program.

DEFINITION C.19. *[Initial configuration] The initial configuration, with a main expression e, is $\Sigma_\star = \langle\langle e, (0, \emptyset)\rangle, \bullet\rangle$.*

LEMMA C.20. [Stores preservation] *If $\Sigma_\star \hookrightarrow^n \Sigma$ and $\Sigma = \langle \psi, \mu \rangle$, then $\mu \vdash \diamond$.*

In Theorem C.25, we will prove that the tasks, created by the **fork** expression, do not interfere, and thus *OpenEffectJ* provides deterministic semantics [33]. We first prove a simpler theorem (Theorem C.23): for the 2 tasks id_1 and id_2, in ψ, generated by a **fork** expression ($\Sigma \twoheadrightarrow \psi$), when the queue is empty $\bullet$, if the store is well-formed $\mu \vdash \diamond$, the effect judgments give them effect ρ_1 and ρ_2 respectively and they do not interfere $\rho_1 \sharp \rho_2$, then their dynamic effects (given by the function *dynESet*) do not interfere. To show this, we need to prove, as an intermediate step, that during the evaluation of the tasks, the effect ρ of an expression e, is refined by the effect ρ' of its immediate subsequent expression e', with task interleaving, in Lemma C.24. To reason about the effect relationship between an expression e and its immediate subsequent expression e' of a task, we define the local reduction of a task, in Definition C.21. Because the dynamic effect of a task *id* should include the effect of any of its child task id' [33], we show that the effect of id' refines *id* in Lemma C.22.

DEFINITION C.21. *[Local reduction] A reduction $\Sigma \hookrightarrow^* \Sigma'$, where $\Sigma = \langle\langle e, (id, I)\rangle + \psi, \mu\rangle$ and $\langle\langle e', (id, I')\rangle + \psi', \mu'\rangle = \Sigma'$, is called a task local reduction, denoted as $\Sigma \rightarrowtail \Sigma'$, if $\nexists e'', I'', \mu'' \psi''$ s.t. $\Sigma \hookrightarrow^* \langle\langle e'', (id, I'')\rangle + \psi'', \mu''\rangle \hookrightarrow^* \Sigma'$.*

The local reduction says that an expression e' of the immediate subsequent expression of e for the same task *id*, disregard the interleaving of other tasks.

LEMMA C.22. [Child effects refine parent effects] *If $\mu \vdash \diamond$, $\Sigma = \langle(\mathbb{E}[e], \tau) + \psi, \mu\rangle$, $e = \textbf{fork } (e_0; e_1)$, $\Sigma \twoheadrightarrow \psi'$, $\mu \vdash e : \rho$, $\langle e_0, \tau_0\rangle \in \psi'$ and $\mu \vdash e_0 : \rho'$ then $\rho' \subseteq \rho$.*

Proof: Immediately follows from the definition of (T-FORK) and (FORK-PARALLEL) rules. ∎

THEOREM C.23. [Noninterference] *Let $\Sigma \twoheadrightarrow \psi$, $\mu \vdash \diamond$, $\Sigma \hookrightarrow \Sigma'$ and $\Sigma = \langle\langle \mathbb{E}[\textbf{fork } (e_1; e_2)], \tau\rangle + \bullet, \mu\rangle$. For the two tasks $\langle e_1, (id_1, I_1)\rangle \in \psi$ and $\langle e_2, (id_2, I_2)\rangle \in \psi$, if $\mu \vdash e_1 : \rho_1$, $\mu \vdash e_2 : \rho_2$, and $\rho_1 \sharp \rho_2$, then $\forall n \in \mathbb{N}$ s.t. $dynE(\Sigma', \bullet, \emptyset, n) = (\Sigma_n, \chi, \Upsilon)$, $dynESet(id_1, \chi, \Upsilon) \sharp dynESet(id_2, \chi, \Upsilon)$. Here $\{\eta \in \chi \mid (\eta = (\ldots, id)) \vee (\eta = (\ldots, id') \wedge id' \in desc(id, \Upsilon))\} = dynESet(id, \Upsilon, \chi)$.*

Proof: Let $\Sigma' = \langle\langle \mathbb{E}[\texttt{yield null}], \tau'\rangle + \psi, \mu\rangle$. Without loss of generality, assume that the task id_1 finishes before id_2. Let N be the smallest integer such that both tasks are done, i.e., $\Sigma_N = \langle\langle v, (id_2, I_N)\rangle + \psi_N, \mu_N\rangle$ and $\Sigma' \hookrightarrow^N \Sigma_N$. It suffices to show that $\forall n \leq N$, the claim is true: once a task is done, it cannot produce any store access.

Here we first prove an equivalent lemma: (1) the dynamic effect η, produced by a reduction, refines the static effect ρ of the original expression of that reduction, i.e., $\eta \propto \rho$; and (2) the static effect ρ of a subsequent expression refines the static effect ρ' of the original expression, i.e., $\rho \subseteq \rho'$. Observe that, with (1) and (2), we know that a dynamic trace, produced by a task *id*, refines the static effect, computed when *id* was created, i.e., $\eta \propto \rho \subseteq \rho'$. Observe that, Theorem C.23 directly follows from Lemma C.24.

LEMMA C.24. *If $\Sigma_n = \langle\langle \mathbb{E}[e'_n], (id'_n, I'_n)\rangle + \psi_n, \mu_n\rangle$, $0 \leq n < N$, $\Sigma' \hookrightarrow^n \Sigma_n$ and $\mu_n \vdash \mathbb{E}[e'_n] : \rho'_n$, then*

(a) $(\bot) \notin \rho'_n$;

(b) if $dyn(\Sigma_n, \chi_n, \Upsilon_n) = (\Sigma_{n+1}, \chi_n + \eta, \Upsilon'_n)$, *then* $\eta \propto \rho'_n$. *Also* $\mu_n \cong \mu_{n+1}$, *where* $\Sigma_{n+1} = \langle \psi_{n+1}, \mu_{n+1} \rangle$;

(c) if $\exists k < n$ *s.t.* $(\Sigma_k = \langle \langle \mathbb{E}[e'_k], (id'_k, I'_k) \rangle + \psi_k, \mu_k \rangle) \wedge (\Sigma_k \rightarrowtail \Sigma_n)$, $\mu_k \vdash e'_k : \rho'_k$ *and* $\mu_n \vdash e'_n : \rho'_n$, *then* $\rho'_n \subseteq \rho'_k$;

We will prove it by induction on n. The base case $n = 0$. Note that $\Sigma' = \langle \langle \mathbb{E}[\texttt{yield null}], \tau' \rangle + \psi, \mu \rangle$ and when the **fork** expression is being evaluated, the queue is empty ($\bullet$). According to the (YIELD) rule, the next configuration must schedule a task id_j in ψ, by the *active* function, i.e. $\Sigma_0 = \langle \langle e_j, (id_j, I_j) \rangle + \psi_0, \mu \rangle$ and $\langle e_j, (id_j, I_j) \rangle \in \psi$. As stated, for id_1 and id_2, $\rho_1 \sharp \rho_2$, thus (a) holds. We have $\mu \vdash \diamond$, by Theorem C.10, (b) holds. (c) holds, because this is the first time the task id_j makes progress, i.e., $\nexists \Sigma_x$ s.t. $(\Sigma_x \rightarrowtail \Sigma_0)$.

For the induction step, we have $\forall j$ s.t. $0 \leq j \leq i$ all the conditions of the lemma are true, and it suffices to prove that when $j = i + 1$, the lemma is true.

Let $\Sigma_j = \left\langle \left\langle \mathbb{E}[e'_j], (id'_j, I'_j) \right\rangle + \psi_j, \mu_j \right\rangle$, $\Sigma_{j+1} = \left\langle \left\langle \mathbb{E}[e'_{j+1}], (id'_{j+1}, I'_{j+1}) \right\rangle + \psi_{j+1}, \mu_{j+1} \right\rangle$ and $\Sigma_{j+2} = \left\langle \left\langle \mathbb{E}[e'_{j+2}], (id'_{j+2}, I'_{j+2}) \right\rangle + \psi_{j+2}, \mu_{j+2} \right\rangle$. By Lemma C.20, $\mu_j \vdash \diamond$, $\mu_{j+1} \vdash \diamond$ and $\mu_{j+2} \vdash \diamond$:

1. if $id'_j = id'_{j+1}$, by Theorem C.10, (a) holds;
2. $id'_j \neq id'_{j+1}$, then $e'_j = \texttt{yield}\ e''_j$ for some e''_j. By the (YIELD) rule, $\mu_{j+1} = \mu_j$. By IH, 1) if $\exists k$ s.t. $\Sigma_k \rightarrowtail \Sigma_{j+1}$, then $\rho'_{j+1} \subseteq \rho'_k$, $(\bot) \notin \rho'_k$, thus $(\bot) \notin \rho'_{j+1}$; 2) otherwise, as stated, for the tasks id_1 and id_2, $\rho_1 \sharp \rho_2$, therefore, (a) holds.

For (b), if $id'_{j+1} = id'_{j+2}$, then by Theorem C.10, it holds; otherwise, $e'_{j+1} = \texttt{yield}\ e''$ for some e'', $\mu_{j+2} = \mu_{j+1}$ and it leaves no dynamic trace, thus it holds. If $\exists k < n$ s.t. $(\Sigma_k = \langle \langle \mathbb{E}[e'_k], (id'_k, I'_k) \rangle + \psi_k, \mu_k \rangle)$, then $e'_k = \texttt{yield}\ e''$ and $e'_j = e''$ for some e''. By IH, $\mu_k \cong \mu_n$. By Lemma C.11, (c) holds.

For all steps $n < N$, we see that $\eta \propto \rho'_n$ and for each local reduction $\Sigma_k \rightarrowtail \Sigma_n$, $\rho'_n \subseteq \rho'_k$. By Lemma C.22, the effect of a child task refines the corresponding **fork** expression of its parent task. Thus, $dynESet(id_1, \chi, \Upsilon) \sharp dynESet(id_2, \chi, \Upsilon)$, if $\rho_1 \sharp \rho_2$. ∎

THEOREM C.25. [Deterministic semantic] *Let* $\mu \vdash \diamond$, $\Sigma \hookrightarrow \Sigma'$, $\Sigma \twoheadrightarrow \psi'$ *and* $\Sigma = \langle \langle \mathbb{E}[\texttt{fork}\ (e_1; e_2)], \tau \rangle + \psi, \mu \rangle$. *For the two tasks* $\langle e_1, (id_1, I_1) \rangle \in \psi'$ *and* $\langle e_2, (id_2, I_2) \rangle \in \psi'$, $\mu \vdash e_1 : \rho_1$, $\mu \vdash e_2 : \rho_2$, *if* $\rho_1 \sharp \rho_2$ *then* $\forall n \in \mathbb{N}$ *s.t.* $dynE(\Sigma', \bullet, \emptyset, n) = (\Sigma_n, \chi, \Upsilon)$, $dynESet(id_1, \chi, \Upsilon) \sharp dynESet(id_2, \chi, \Upsilon)$.

Proof: Note that the difference between this theorem and Theorem C.23 is that ψ may or may not be $\bullet$. Observe that there exists $\Sigma_x = \langle \langle \mathbb{E}[\texttt{fork}\ (e_x; e'_x)], \tau_x \rangle + \bullet, \mu_x \rangle$ and $k > 0$ such that $\Sigma_x \hookrightarrow^k \Sigma$. Without loss of generality, assume that the task id_1 finishes before id_2. Let N be the smallest integer such that both tasks id_1 and id_2 are done, i.e., $\Sigma_N = \langle \langle v, (id_2, I_N) \rangle + \psi_N, \mu_N \rangle$ and $\Sigma' \hookrightarrow^N \Sigma_N$. By Lemma C.24, $\forall 0 \leq j < (k + N)$, if $\Sigma_x \hookrightarrow^j \Sigma_j$, then $\mu_x \cong \mu_j$. Observe that all the conditions of Lemma C.24 are still correct in this theorem and the claim holds. ∎

C.4 Type Soundness

In this section, we prove the standard type preservation property. Type rules omitted in Section 4 are in Figure 10. To prove the type preservation, we extend the type environment, which maps variables and locations to types.

Before proving the type preservation theorem, we define the consistency between a type environment and a store [19], which is standard. One difference between concurrent tasks application and serial application is context switching. We will show in the type

$$\text{(T-GET-OPEN-LOC)}\quad \frac{\Pi(loc) = c \qquad c <: d \qquad typeOfF(f) = (d, \texttt{@open}\ t)}{\Pi \vdash loc.f : (t, \{\texttt{read}\ f\})}$$

$$\text{(T-SET-OPEN-LOC)}\quad \frac{\Pi(loc) = c \qquad c <: d \qquad typeOfF(f) = (d, \texttt{@open}\ t) \qquad \Pi \vdash e' : (t', \rho') \qquad t' <: t}{\Pi \vdash loc.f = e' : (t', \{\bot\})}$$

$$\text{(T-LOC)}\quad \frac{\Pi(loc) = t}{\Pi \vdash loc : (t, \emptyset)}$$

$$\text{(T-CALL-OPEN-LOC)}\quad \frac{\Pi \vdash loc.f : (c_0, \rho_0) \qquad typeOfF(f) = (d, \texttt{@open}\ c_0) \qquad findMeth(c_0, m) = (c_1, t, m\ (\overline{t\ var})\ \{e_{n+1}\}, \rho) \qquad (\forall i \in \{1..n\}\ ::\ \Pi \vdash e_i : (t'_i, \rho_i) \wedge t'_i <: t_i)}{\Pi \vdash loc.f.m(\overline{e}) : (t, \{\texttt{open}\ f\ m\ \emptyset\} \cup \bigcup_{i=0}^{n} \rho_i)}$$

$\Pi ::= \{\iota_j \mapsto t_j\}_{j \in \mathbb{N}}$ "type environments"
where $\iota \in (\mathscr{L} \cup \{\texttt{this}\} \cup \mathscr{V})$

Figure 10. Type and effect rules for loc.

preservation theorem that after the control comes back to the current task, the remaining expression has the same type before yielding control. This is mainly proven by Lemma C.31, i.e., each task in the reduction do not change the type of locations in the type environment and the type environment keeps extending (Definition C.27, i.e. $\Pi \lessdot \Pi'$). Also, we need to ensure that during the evaluation, all the tasks have proper types, i.e., all the tasks in the queue ψ are well-typed (Definition C.28 and Definition C.29). Finally, we state the stardard lemmas [14, 19] (Lemma C.30, Lemma C.31, Lemma C.32 and Lemma C.33).

DEFINITION C.26. *[Environment-store consistency] A store* μ is consistent with *a type environment* Π, *written* $\mu \approx \Pi$, *if all of the following hold:*

1. $\forall loc$ *s.t.* $\mu(loc) = [t.F.E]$,
   (a) $\Pi(loc) = t$ *and*
   (b) $dom(F) = dom(fields(t))$ *and*
   (c) $rng(F) \subseteq dom(\mu) \cup \{\texttt{null}\}$ *and*
   (d) $\forall f \in dom(F)$ *s.t.* $F(f) = loc'$, $\mu(loc') = [t'.F'.E']$ *and* $typeOfF(f) = (c, [\texttt{@open}\ ]\ u) \Rightarrow t' <: u$
2. $loc \in dom(\Pi) \Rightarrow loc \in dom(\mu)$

DEFINITION C.27. *[Environment enlargement] Let* Π *and* Π' *be two type environments. We write* $\Pi \lessdot \Pi'$, *if* $dom(\Pi) \subseteq dom(\Pi')$ *and* $\forall a \in dom(\Pi)$, *if* $\Pi(a) = t$, *then* $\Pi'(a) = t$.

This definition says taht an environment Π' enlarges another environment Π, if the domain of Π is a subset of Π' and, they give the same type for the common location. This definition will be used to show that during the evaluation of any *OpenEffectJ* program, we can use an ever increasing type environment to type check the expressions.

DEFINITION C.28. *[Well-typed queue] A queue* ψ is well-typed *in* Π, *written* $\Pi \vdash \psi$, *if* $\forall \langle e, \tau \rangle \in \psi$, $\Pi \vdash e : (t, \rho)$ *for some type t and effect* ρ.

A queue is well-typed, if the expression in each task in the queue has proper type. This definition will be used to prove that after the control go back to the original expression, due to thread interleaving, it has the same type given the update-to-date type environment.

DEFINITION C.29. *[Well-typed configuration] A configuration* $\Sigma = \langle \psi, \mu \rangle$ is well-typed *in* Π, *written* $\Pi \vdash \Sigma$, *if* $\Pi \vdash \psi$ *and* $\mu \approx \Pi$.

LEMMA C.30. [Substitution] *If* $\Pi, \overline{var : t} \vdash e : (t, \rho)$ *and* $\forall i \in \{1..n\}$, $\Pi \vdash v_i : (s_i, \emptyset)$ *where* $s_i <: t_i$ *then* $\Pi \vdash [\overline{v/var}]e : (s, \rho')$ *for some* $s <: t$ *and some* ρ'.

Proof: To simplify the notations, we let $\Pi' = \Pi, \overline{var : t}$. We prove it by structural induction on the derivation of $\Pi \vdash e : (t, \rho)$ and by cases, based on the last step in that derivation. The base cases include (T-NEW), (T-NULL), (T-LOC), and (T-VAR). The first three of these cases are obvious: e has no variables, $s = t$. In the (T-VAR) case, $e = var$, and there are two subcases. If $var \notin \{var_1, ..., var_n\}$, then $\Pi'(var) = \Pi(var) = t$ and the claim holds. Otherwise, suppose $var = var_k$. Then $[\overline{v/var}]e = v_k$ and, by the assumptions of the lemma, $\Pi \vdash [\overline{v/var}]e : (s_k, \emptyset)$ and $s_k <: t_k = t$.

The remaining cases cover the induction step. The induction hypothesis (IH) is that the claim of the lemma holds for all subderivations of the derivation being considered.

The cases for (T-YIELD), (T-DEFINE), and (T-SEQ) follow directly from the induction hypothesis. (T-FORK) holds because its type is **void** before and after the substitution.

(T-CALL-OPEN) Here $e = \mathbf{this}.f.m(\overline{e'})$. The last type derivation step has the following form:

$$\frac{\begin{array}{c} e'_0 = \mathbf{this}.f \\ typeOfF(f) = (d, \mathtt{@open}\ c_0) \qquad \Pi' \vdash e'_0 : (c_0, \rho_0) \\ findMeth(c_0, m) = (c_1, t, m\ (u_1\ var_1, \ldots, u_n\ var_n)\ \{e_{n+1}\}, \rho_2) \\ (\forall\, i \in \{1..n\}\ ::\ \Pi' \vdash e'_i : (u'_i, \rho_i)\ \wedge\ u'_i <: u_i) \end{array}}{\Pi' \vdash e'_0.m(e'_1, \ldots, e'_n) : (t, \{\mathbf{open}\ f\ m\ \emptyset\} \cup \bigcup_{i=0}^{n} \rho_i)}$$

Let $e''_i = [\overline{v/var}]e'_i$ for $i \in \{0..n\}$, then $[\overline{v/var}]e = e''_0.m(\overline{e''})$. We show that $\Pi \vdash [\overline{v/var}]e : (t, \rho')$. By IH, $\Pi \vdash e''_0 = (c2, \rho''_0)$, where $c2 <: c_0$. If $findMeth(c_0, m) = (c_1, t, m\ (\overline{u\ var})\ \{e_{n+1}\}, \rho_2)$ and $findMeth(c_2, m) = (c_3, t_2, m\ (\overline{u\ var})\ \{e'_{n+1}\}, \rho_3)$, by the definitions of *findMeth* and *override*, $t_2 = t$. Also, by IH, $\forall i \in \{1..n\} :: \Pi \vdash e''_i : (u''_i, \rho_i)$ and $u''_i <: u'_i$. Finally, $\forall i \in \{1..n\}\ ::\ u''_i <: u_i$, by transitivity, the claim holds.

(T-CALL) Here $e = e'_0.m(\overline{e'})$. The last type derivation step has the following form:

$$\frac{\begin{array}{c} (c_1, t, m\ (u_1\ var_1, \ldots, u_n\ var_n)\ \{e_{n+1}\}, \rho'') = findMeth(u'_0, m) \\ \Pi' \vdash e'_0 : (u'_0, \rho_0) \\ (\forall\, i \in \{1..n\}\ ::\ \Pi' \vdash e'_i : (u'_i, \rho_i)\ \wedge\ u'_i <: u_i) \end{array}}{\Pi' \vdash e'_0.m(e'_1, \ldots, e'_n) : (t, \bot)}$$

Let $e''_i = [\overline{v/var}]e'_i$ for $i \in \{0..n\}$, then $[\overline{v/var}]e = e''_0.m(\overline{e''})$. We show $\Pi \vdash [\overline{v/var}]e : (t, \rho')$, for some ρ'. By IH, $\Pi \vdash e''_0 : (u''_0, \rho'_0)$, where $u''_0 <: u'_0$. By the definitions of *findMeth* and *override*, if $findMeth(u'_0, m) = (c_1, t, m(\overline{u\ var})(e'_{n+1}), \rho'')$, and $findMeth(u''_0, m) = (c_2, t_2, m(\overline{t\ var})(e''_{n+1}), \rho''')$, then $t_2 = t$. Also by IH $\forall i \in \{1..n\} :: \Pi \vdash e''_i : (u''_i, \rho'_i)$ and $u''_i <: u'_i$. Finally, $\forall i \in \{1..n\} :: u''_i <: u_i$, by transitivity the claim holds.

(T-CALL-OPEN-LOC) Here $e = loc.f.m(\overline{e'})$. The last type derivation step has the following form:

$$\frac{\begin{array}{c} typeOfF(f) = (d, \mathtt{@open}\ c_0) \qquad \Pi' \vdash loc.f : (c_0, \rho_0) \\ findMeth(c_0, m) = (c_1, t, m\ (u_1\ var_1, \ldots, u_n\ var_n)\ \{e_{n+1}\}, \rho'') \\ (\forall\, i \in \{1..n\}\ ::\ \Pi' \vdash e'_i : (u'_i, \rho_i)\ \wedge\ u'_i <: u_i) \end{array}}{\Pi' \vdash loc.f.m(e'_1, \ldots, e'_n) : (t, \{\mathbf{open}\ f\ m\ \emptyset\} \cup \bigcup_{i=0}^{n} \rho_i)}$$

Let $e''_i = [\overline{v/var}]e'_i$ for $i \in \{1..n\}$, then $[\overline{v/var}]e = loc.f.m(\overline{e''})$. We show that $\Pi \vdash [\overline{v/var}]e : (t, \rho')$ for some ρ'. By IH, $\forall i \in \{1..n\} :: \Pi \vdash e''_i : (u''_i, \rho'_i)$ and $u''_i <: u'_i$. Finally, $\forall i \in \{1..n\}\ ::\ u''_i <: u_i$, by transitivity and thus the claim holds.

(T-GET) Here $e = e'.f$. The last derivation step is:

$$\frac{\Pi' \vdash e' : (c, \rho') \qquad typeOfF(f) = (d, t) \qquad c <: d}{\Pi' \vdash e'.f : (t, \rho_0 \cup \{\mathbf{read}\ f\})}$$

Now $[\overline{v/var}]e = [\overline{v/var}]e'.f$. By IH, $\Pi \vdash [\overline{v/var}]e' : (u', \rho_1)$, where $u' <: u$. By the definition of *typeOfF*, $typeOfF(f)$ does not change. Therefore $\Pi \vdash [\overline{v/var}]e : (t, \rho_1 \cup \{\mathbf{read}\ f\})$ and the claim holds.

(T-GET-OPEN) Here $e = \mathbf{this}.f$. The last step is:

$$\frac{\Pi'(this) : c \qquad typeOfF(f) = (d, \mathtt{@open}\ t) \qquad c <: d}{\Pi' \vdash \mathbf{this}.f : (t, \{\mathbf{read}\ f\})}$$

There are two subcases: 1) if $\mathbf{this} \notin \{var_1, ..., var_n\}$, then $\Pi' \vdash e = \Pi \vdash e$ and the claim holds. Otherwise, suppose $\mathbf{this} = var_k$. Then $[\overline{v/var}]e = v_k.f$ and, by (T-GET-OPEN-LOC), $\Pi \vdash [\overline{v/var}]e : (t, \{\mathbf{read}\ f\})$.

(T-GET-OPEN-LOC) Here $e = loc.f$. This expression has no free variable, the claim holds.

(T-SET) Here $e = e'_1.f = e'_2$. The last derivation step is:

$$\frac{\begin{array}{cc} \Pi' \vdash e'_1 : (c, \rho_1) & typeOfF(f) = (d, u) \\ c <: d \qquad \Pi' \vdash e'_2 : (t, \rho_2) & t <: u \end{array}}{\Pi' \vdash e'_1.f = e'_2 : (t, \rho_1 \cup \rho_2 \cup \{\mathbf{write}\ f\})}$$

Now $[\overline{v/var}]e = ([\overline{v/var}]e'_1.f = [\overline{v/var}]e'_2)$. By IH, $\Pi \vdash [\overline{v/var}]e'_1 : (u''_1, \rho'_1)$, where $u''_1 <: u'_1$; $\Pi \vdash [\overline{v/var}]e'_2 : (u'_2, \rho'_2)$, where $u'_2 <: t$. By the definition of *typeOfF*, its result does not change. By transitivity $t' = u'_2 <: t <: u$. Therefore $\Pi \vdash [\overline{v/var}]e : (t', \rho'_1 \cup \rho'_2 \cup \{\mathbf{write}\ f\})$, $t' <: t$. The claim holds.

(T-SET-OPEN) Here $e = \mathbf{this}.f = e'$. The last step is:

$$\frac{\begin{array}{cc} \Pi'(\mathbf{this}) = c & typeOfF(f) = (d, \mathtt{@open}\ u) \\ c <: d \qquad \Pi' \vdash e' : (t, \rho_0) & t <: u \end{array}}{\Pi' \vdash \mathbf{this}.f = e' : (t, \{\bot\})}$$

Now $[\overline{v/var}]e = ([\overline{v/var}]\mathbf{this}.f = [\overline{v/var}]e')$. By IH, $\Pi \vdash [\overline{v/var}]e' : (u'_2, \rho_1)$, where $u'_2 <: t$. By the definition of *typeOfF*, its result does not change. By transitivity $t' = u'_2 <: t <: u$. There are two subcases: 1) if $\mathbf{this} \notin \{var_1, ..., var_n\}$, then nothing changes. Otherwise, suppose $\mathbf{this} = var_k$. Then $[\overline{v/var}]e = v_k.f = [\overline{v/var}]e'$ and, by (T-SET-OPEN-LOC), $\Pi \vdash [\overline{v/var}]e : (t', \{\bot\})$, $t' <: t$ and the claim holds.

(T-SET-OPEN-LOC) Here $e = loc.f = e'$. The last step is:

$$\frac{\begin{array}{cc} \Pi'(loc) = c & typeOfF(f) = (d, \mathtt{@open}\ u) \\ c <: d \qquad \Pi' \vdash e' : (t, \rho') & t <: u \end{array}}{\Pi' \vdash loc.f = e' : (t, \{\bot\})}$$

Now $[\overline{v/var}]e = loc.f = [\overline{v/var}]e')$. By IH, $\Pi \vdash [\overline{v/var}]e' : (u'_2, \rho'_2)$, where $u'_2 <: t$. By the definition of *typeOfF*, its result does not change. By transitivity $t' = u'_2 <: t <: u$. Therefore $\Pi \vdash [\overline{v/var}]e : (t', \{\bot\})$, $t' <: t$ and the claim holds.

Thus, for all possible derivations of $\Pi' \vdash e : (t, \rho)$ we see that $\Pi \vdash [\overline{v/var}]e : (t', \rho)$ for some $t' <: t$. ∎

LEMMA C.31. [Environment extension] *If* $\Pi \vdash e : (t, \rho)$ *and* $a \notin dom(\Pi)$, *then* $(\Pi, a : t') \vdash e : (t, \rho)$.

Proof: Observe that the effect does not depend on the typing environment and it suffices to prove the typing relationship. The proof is by a structural induction on the derivation of $\Pi \vdash e : (t, \rho)$. The base cases are (T-NEW), (T-NULL), (T-LOC), and (T-VAR). In (T-NEW) and (T-NULL), the type environment does not appear in the hypotheses of the judgment, so the claim holds. For the (T-VAR) case,

$e = var$ and $\Pi(var) = t$. But $a \notin dom(\Pi)$, so $var \neq a$. Therefore $(\Pi, a : t')(var) = t$ and the claim holds for this case. The (T-LOC) case is similar. The remaining rules cover the induction step. By the induction hypothesis, changing the type environment to $\Pi, a : t'$ does not change the types and effects assigned by any hypotheses of each rule. Therefore, the types and effects assigned by each rule are also unchanged and the claim holds. ∎

LEMMA C.32. [Replacement] *If* $\Sigma = \langle\langle \mathbb{E}[e], (id, I)\rangle + \psi, \mu\rangle$, $\Sigma' = \langle\langle \mathbb{E}[e'], (id, I')\rangle + \psi', \mu'\rangle$, $\Sigma \hookrightarrow \Sigma'$, $\Pi \vdash \mathbb{E}[e] : (t, \rho)$, $\Pi \vdash e : (t', \rho')$ *and* $\Pi \vdash e' : (t', \rho'_0)$, *then* $\Pi \vdash \mathbb{E}[e'] : (t, \rho_0)$, *for some* ρ_0.

Proof: Proof is by induction on the size of the evaluation context $\mathbb{E}$ [4]. Size of the $\mathbb{E}$ refers to the number of recursive applications of the syntactic rules necessary to create $\mathbb{E}$. In the base case, $\mathbb{E}$ has size zero, $\mathbb{E} = -$, and $t' = u' <: u = t$. For the induction step we divide the evaluation context into two parts such that $\mathbb{E}[e_1] = \mathbb{E}_1[\mathbb{E}_2[e_2]]$, and $\mathbb{E}_2$ has size one. The induction hypothesis (IH) is that the lemma holds for all evaluation contexts, which is smaller than the one ($\mathbb{E}_1$) considered in the induction step. We prove it case by case on the rule used to generate $\mathbb{E}_2$. In each case we show that $\Pi \vdash \mathbb{E}_2[e] : (s, \rho)$ implies that $\Pi \vdash \mathbb{E}_2[e'] : (s, \rho')$, for some ρ', and thus the claim holds by IH.

The cases for $(loc_0.f = -)$, $(-; e_2)$, and $(c\ var = -; e_2)$ follow directly from the induction hypothesis.

Case $-.m(\overline{e})$ The last step for $\mathbb{E}_2[e]$ could be
<1> (T-CALL):

$$\frac{(c_1, t, m\ (\overline{t\ var})\ \{e_{n+1}\}, \rho_0''') = \mathit{findMeth}(u, m) \qquad \Pi \vdash e : (u, \rho_0) \qquad (\forall\, i \in \{1..n\}\ ::\ \Pi \vdash e_i : (t'_i, \rho''_i) \wedge t'_i <: t_i)}{\Pi \vdash \mathbb{E}_2[e] : (t, \{\bot\})}$$

Here $\mathit{findMeth}(u', m) = (c_2, t, m\ (\overline{t\ var})\ \{e'_{n+1}\}, \rho_1''')$, by the definitions of *override* and *findMeth*, where $c_2 <: c_1$, so (T-CALL) gives $\Pi \vdash \mathbb{E}_2[e'] : (t, \rho')$; or
<2> (T-CALL-OPEN):

$$\frac{e = \texttt{this}.f \qquad \mathit{typeOfF}(f) = (d, \texttt{@open}\ t') \qquad (c_1, t, m\ (t_1\ var_1, \ldots, t_n\ var_n)\ \{e_{n+1}\}, \rho_0''') = \mathit{findMeth}(t', m) \qquad \Pi \vdash e : (t', \rho') \qquad (\forall\, i \in \{1..n\}\ ::\ \Pi \vdash e_i : (t'_i, \rho''_i) \wedge t'_i <: t_i)}{\Pi \vdash \mathbb{E}_2[e] : (t, \{\texttt{open}\ f\ m\ \emptyset\} \cup \rho' \cup \bigcup_{i=1}^{n} \rho''_i)}$$

It must be the case that $e' = loc.f$. From the statement of the lemma, we have $\Pi \vdash loc.f : (t', \rho'_0)$. Also the results of *typeOfF* and *findMeth* does not change, therefore, by (T-CALL-OPEN-LOC), the type of the expression is t.
<3> (T-CALL-OPEN-LOC):

$$\frac{e = loc.f \qquad \mathit{typeOfF}(f) = (d, \texttt{@open}\ t') \qquad (c_1, t, m\ (t_1\ var_1, \ldots, t_n\ var_n)\ \{e_{n+1}\}, \rho_0''') = \mathit{findMeth}(t', m) \qquad \Pi \vdash e : (t', \rho') \qquad (\forall\, i \in \{1..n\}\ ::\ \Pi \vdash e_i : (t'_i, \rho''_i) \wedge t'_i <: t_i)}{\Pi \vdash \mathbb{E}_2[e] : (t, \{\texttt{open}\ f\ m\ \emptyset\} \cup \rho' \cup \bigcup_{i=1}^{n} \rho''_i)}$$

It must be the case that $e' = loc'$. From the statement of the lemma, we have $\Pi \vdash loc' : (t', \rho'_0)$. Also the results of *findMeth* does not change, therefore, by (T-CALL), the type of the expression is t.

[4] Formulation of the proof is similar Flatt's work [19]

Case $v.m(v_1, \ldots, v_{p-1}, -, e_{p+1}, \ldots, e_n)$ The last step in the type derivation for $\mathbb{E}_2[e]$ must be (T-CALL):

$$\frac{\Pi \vdash v : (u, \emptyset) \qquad (c_1, t, m\ (t_1\ var_1, \ldots, t_n\ var_n)\ \{e_{n+1}\}, \rho_0''') = \mathit{findMeth}(u, m) \qquad (\forall\, i \in \{1..(p-1)\}\ ::\ \Pi \vdash v_i : (t'_i, \emptyset) \wedge t'_i <: t_i) \qquad (\forall\, j \in \{(p+1)..n\}\ ::\ \Pi \vdash e_j : (t'_j, \rho''_j) \wedge t'_j <: t_j) \qquad \Pi \vdash e_p : (t', \rho') \wedge t' <: t_p)}{\Pi \vdash \mathbb{E}_2[e] : (t, \{\bot\})}$$

We have $\Pi \vdash e' : (t', \rho'_0)$ and $t' <: t_p$ and other parts of condictions do not change. The claim holds.

Case $-.f = e_2$ The last step for $\mathbb{E}_2[e]$ must be
<1> (T-SET):

$$\frac{\Pi \vdash e : (c, \rho') \qquad \mathit{typeOfF}(f) = (d, t) \qquad c <: d \qquad \Pi \vdash e_2 : (t_2, \rho_2) \qquad t_2 <: t}{\Pi \vdash \mathbb{E}_2[e] : (t_2, \rho' \cup \rho_2 \cup \{\texttt{write}\ f\})}$$

By the definition of field lookup, $\mathit{typeOfF}(f)$ does not change. Thus, by (T-SET), $\Pi \vdash \mathbb{E}_2[e'] : (t_2, \rho_0)$; or
<2> (T-SET-OPEN):

$$\frac{e = \texttt{this} \qquad \Pi(\texttt{this}) : t' \qquad \mathit{typeOfF}(f) = (d, \texttt{@open}\ t) \qquad t' <: d \qquad \Pi \vdash e_2 : (t_2, \rho_2) \qquad t_2 <: t}{\Pi \vdash \mathbb{E}_2[e] : (t_2, \{\bot\})}$$

The only possibility is that $e' = loc$, for some loc. By the statement of this lemma $\Pi \vdash e' : (t', \rho'_0)$, i.e., $\Pi \vdash loc : (t', \rho'_0)$, thus by (T-SET-OPEN-LOC), the claim holds.

Case $-.f$ The last step for $\mathbb{E}_2[e]$ could be
<1> (T-GET):

$$\frac{\Pi \vdash e : (c, \rho_1) \qquad \mathit{typeOfF}(f) = (d, t) \qquad c <: d}{\Pi \vdash \mathbb{E}_2[e] : (t, \rho_1 \cup \{\texttt{read}\ f\})}$$

The result of *typeOfF* does not change. Thus, by (T-GET), $\Pi \vdash \mathbb{E}_2[e'] : (t, \rho_0)$; or
<2> (T-GET-OPEN):

$$\frac{e = \texttt{this} \qquad \Pi(e) = c \qquad \mathit{typeOfF}(f) = (d, \texttt{@open}\ t) \qquad c <: d}{\Pi \vdash \mathbb{E}_2[e] : (t, \{\texttt{read}\ f\})}$$

e' must be loc, for some loc. By the statement $\Pi \vdash loc : (c, \emptyset)$. The result of *typeOfF* does not change. Thus, by (T-GET-OPEN-LOC), $\Pi \vdash \mathbb{E}_2[e'] : (t, \{\texttt{read}\ f\})$. ∎

LEMMA C.33. [Replacement with subtyping] *If* $\Sigma \hookrightarrow \Sigma'$, $\Sigma = \langle\langle \mathbb{E}[e], (id, I)\rangle + \psi, \mu\rangle$, $\Sigma' = \langle\langle \mathbb{E}[e'], (id, I')\rangle + \psi', \mu'\rangle$, $\Pi \vdash \mathbb{E}[e] : (t, \rho)$, $\Pi \vdash e : (u, \rho_0)$, *and* $\Pi \vdash e' : (u', \rho_1)$ *and* $u' <: u$, *then* $\Pi \vdash \mathbb{E}[e'] : (t', \rho')$ *where* $t' <: t$.

Proof: Proof is by induction on the size of the evaluation context $\mathbb{E}$ [5]. Size of the $\mathbb{E}$ refers to the number of recursive applications of the syntactic rules necessary to create $\mathbb{E}$. In the base case, $\mathbb{E}$ has size zero, $\mathbb{E} = -$, and $t' = u' <: u = t$. For the induction step we divide the evaluation context into two parts such that $\mathbb{E}[e_1] = \mathbb{E}_1[\mathbb{E}_2[e_2]]$, and $\mathbb{E}_2$ has size one. The induction hypothesis (IH) is that the lemma holds for all evaluation contexts, which is smaller than the one ($\mathbb{E}_1$) considered in the induction step. We prove it case by case on the rule used to generate $\mathbb{E}_2$. In each case we show that $\Pi \vdash \mathbb{E}_2[e] : (s, \rho)$ implies that $\Pi \vdash \mathbb{E}_2[e'] : (s', \rho')$, for some $s' <: s$, and the claim holds by IH. The cases for $(loc_0.f = -)$, $(-; e_2)$, and $(c\ var = -; e_2)$ follow directly from IH.

[5] Formulation of the proof is similar to Clifton's work [14] and Flatt's work [19]

Case $-.m(\overline{e})$ The last step for $\mathbb{E}_2[e]$ could be
<1> (T-CALL):

$$\frac{\Pi \vdash e : (t', \rho') \qquad (c_1, t, m\ (\overline{t\ var})\ \{e_{n+1}\}, \rho_1) = \mathit{findMeth}(t', m) \qquad (\forall\, i \in \{1..n\}\ ::\ \Pi \vdash e_i : (t_i', \rho_i'') \wedge t_i' <: t_i)}{\Pi \vdash \mathbb{E}_2[e] : (t, \{\bot\})}$$

We have $\mathit{findMeth}(t', m) = (c_2, t, m\ (\overline{t\ var})\ \{e'_{n+1}\}, \rho_1)$, by the definitions of *override* and *findMeth*, $c_2 <: c_1$, so (T-CALL) gives $\Pi \vdash \mathbb{E}_2[e'] : (t, \{\bot\})$; or
<2> (T-CALL-OPEN):

$$\frac{e = \mathtt{this}.f \qquad \mathit{typeOfF}(f) = (d, \mathtt{@open}\ u) \qquad (c_1, t, m\ (\overline{t\ var})\ \{e_{n+1}\}, \rho_1) = \mathit{findMeth}(u, m) \qquad \Pi \vdash e : (u, \rho_2) \qquad (\forall\, i \in \{1..n\}\ ::\ \Pi \vdash e_i : (t_i', \rho_i'') \wedge t_i' <: t_i)}{\Pi \vdash \mathbb{E}_2[e] : (t, \{\mathtt{open}\ f\ m\ \emptyset\} \cup \rho_2 \cup \bigcup_{i=1}^{n} \rho_i'')}$$

It must be the case that $e' = \mathit{loc}.f$. From the statement of the lemma, we have $\Pi \vdash \mathit{loc}.f : (u', \rho_1)$, where $u' <: u$. By the definitions of *override* and *findMeth*, $\mathit{findMeth}(t', m) = (c_2, t, m\ (\overline{t\ var})\ \{e'_{n+1}\}, \rho_1')$. The result of *typeOfF* does not change, so the type of the expression is t, by (T-CALL-OPEN-LOC);
<3> (T-CALL-OPEN-LOC):

$$\frac{e = \mathit{loc}.f \qquad \mathit{typeOfF}(f) = (d, \mathtt{@open}\ t') \qquad (c_1, t, m\ (\overline{t\ var})\ \{e_{n+1}\}, \rho_0') = \mathit{findMeth}(t', m) \qquad \Pi \vdash e : (u, \rho_0) \qquad (\forall\, i \in \{1..n\}\ ::\ \Pi \vdash e_i : (t_i', \rho_i'') \wedge t_i' <: t_i)}{\Pi \vdash \mathbb{E}_2[e] : (t, \{\mathtt{open}\ f\ m\ \emptyset\} \cup \rho_0 \cup \bigcup_{i=1}^{n} \rho_i'')}$$

It must be the case that $e' = \mathit{loc}'$. From the statement of the lemma, we have $\Pi \vdash \mathit{loc}' : (u', \rho_0')$, where $u' <: u$. By the definitions of *override* and *findMeth*, $\mathit{findMeth}(u', m) = (c_2, t, m\ (\overline{t\ var})\ \{e'_{n+1}\}, \rho_1')$. Therefore, by (T-CALL), the type of the expression is t.

Case $v_0.m(v_1, \ldots, v_{p-1}, -, e_{p+1}, \ldots, e_n)$ Here $p \in \{1..n\}$. The last step for $\mathbb{E}_2[e]$ must be (T-CALL):

$$\frac{\Pi \vdash v_0 : (u_0, \emptyset) \qquad (c, t, m\ (\overline{t\ var})\ \{e_{n+1}\}, \rho''') = \mathit{findMeth}(u_0, m) \qquad (\forall\, i \in \{1..(p-1)\}\ ::\ \Pi \vdash v_i : (u_i', \emptyset)) \qquad (\forall\, i \in \{(p+1)..n\}\ ::\ \Pi \vdash e_i : (u_i', \rho_i'')) \qquad \Pi \vdash e : (u, \rho_0) \qquad (\forall\, i \in \{1..n\} \setminus \{p\}\ ::\ u_i' <: t_i) \qquad u <: t_p}{\Pi \vdash \mathbb{E}_2[e] : (t, \{\bot\})}$$

Now $u' <: u <: t_p$, so by (T-CALL), $\Pi \vdash \mathbb{E}_2[e'] : (t, \{\bot\})$.

Case $-.f = e_2$ The last step for $\mathbb{E}_2[e]$ could be
<1> (T-SET):

$$\frac{\Pi \vdash e : (u, \rho_0) \qquad \mathit{typeOfF}(f) = (d, t_0) \qquad u <: d \qquad \Pi \vdash e_2 : (t, \rho_2) \qquad t <: t_0}{\Pi \vdash \mathbb{E}_2[e] : (t, \rho_0 \cup \rho_2 \cup \{\mathtt{write}\ f\})}$$

Now $u' <: u <: d$. The result of $\mathit{typeOfF}(f)$ does not change. Thus, by (T-SET), $\Pi \vdash \mathbb{E}_2[e'] : (t, \rho')$; or
<2> (T-SET-OPEN):

$$\frac{e = \mathtt{this} \qquad \Pi(\mathtt{this}) : u \qquad \mathit{typeOfF}(f) = (d, \mathtt{@open}\ t') \qquad u <: d \qquad \Pi \vdash e_2 : (t, \rho_2) \qquad t <: t'}{\Pi \vdash \mathbb{E}_2[e] : (t, \{\bot\})}$$

The only possibility is that $e' = \mathit{loc}$, for some *loc*. By the statement of this lemma $\Pi \vdash e' : (u', \rho_1)$, where $u' <: u <: d$. Thus by (T-SET-OPEN-LOC), the type is t.

Case $-.f$ The last step for $\mathbb{E}_2[e]$ could be
<1> (T-GET):

$$\frac{\Pi \vdash e : (c, \rho_1) \qquad \mathit{typeOfF}(f) = (d, t) \qquad c <: d}{\Pi \vdash \mathbb{E}_2[e] : (t, \rho_1 \cup \{\mathtt{read}\ f\})}$$

The result of *typeOfF* does not change. Thus, by (T-GET), $\Pi \vdash \mathbb{E}_2[e'] : (t', \rho_0)$; or
<2> (T-GET-OPEN):

$$\frac{e = \mathtt{this} \qquad \Pi(e) = c \qquad \mathit{typeOfF}(f) = (d, \mathtt{@open}\ t) \qquad c <: d}{\Pi \vdash \mathbb{E}_2[e] : (t, \{\mathtt{read}\ f\})}$$

Here e' must be *loc*, for some *loc*. By the statement $\Pi \vdash \mathit{loc} : (c', \emptyset)$, and $c' <: c$. The result of *typeOfF* does not change. Thus, by (T-GET-OPEN-LOC), $\Pi \vdash \mathbb{E}_2[e'] : (t, \{\mathtt{read}\ f\})$. ∎

THEOREM C.34. [Type preservation] *If* $\Pi \vdash \Sigma$, *where* $\Sigma = \langle\langle e, (id, I)\rangle + \psi, \mu\rangle$, $\Sigma \rightarrowtail \langle\langle e', (id, I')\rangle + \psi', \mu'\rangle$, *and* $\Pi \vdash e : (t, \rho)$, *then there is some* Π', t' *and* ρ' *such that*

(a) $(\mu' \approx \Pi') \wedge (\Pi' \vdash \psi')$, *i.e.* $\Pi' \vdash \Sigma'$;
(b) $\Pi \lessdot \Pi'$; *and*
(c) $\Pi' \vdash e' : (t', \rho') \wedge (t' <: t)$.

Proof: The proof is by cases on the reduction step applied. We prove the first seven cases where the reduction takes only one task local step. Then, we prove the case for yielding controls to other tasks. For all the base cases (except for the **fork**-parallel rule), the queue ψ does not change, so by Lemma C.31 (Environment extension) and Definition C.28, if $\Pi \vdash \psi$, then $\Pi' \vdash \psi'$.

New Object Here $e = \mathbb{E}[\mathit{new}\ c()]$ and $e' = \mathbb{E}[\mathtt{yield}\ \mathit{loc}]$, where $\mathit{loc} \notin \mathit{dom}(\mu)$, and $\mu' = \{\mathit{loc} \mapsto [c.\{f \mapsto \mathtt{null} \mid f \in \mathit{fields}(c)\}.\{(m \mapsto \rho) \in \mathit{methE}(c)\}]\} \oplus \mu$. Let $\Pi' = \Pi, \mathit{loc} : c$, then $\Pi \lessdot \Pi'$. We now show that $\Pi' \approx \mu'$. Because $\mathit{loc} \notin \mathit{dom}(\mu)$, $(\Pi \approx \mu) \Rightarrow (\mathit{loc} \notin \mathit{dom}(\Pi))$ by Definition C.26. Thus part 1 of the definition for $\Pi' \approx \mu'$ holds for all $\mathit{loc}' \neq \mathit{loc}$. Now $\mu'(\mathit{loc}) = [c.F.E]$, $\Pi'(\mathit{loc}) = c$, $\mathit{dom}(F) = \mathit{dom}(\mathit{fields}(c))$, $\mathit{rng}(F) = \{\mathtt{null}\} \subseteq \mathit{dom}(\mu) \cup \{\mathtt{null}\}$, and 1(d) holds vacuously. So part 1 of $\Pi' \approx \mu'$ holds. Part 2 holds because $\Pi \approx \mu$, $\mathit{loc} \in \mathit{dom}(\Pi')$, $\mathit{loc} \in \mathit{dom}(\mu')$.
By Lemma C.31 (Environment extension) and $\mathit{loc} \notin \mathit{dom}(\Pi)$, we have $\Pi' \vdash \mathbb{E}[\mathtt{new}\ c()] : (t, \rho)$. Now $\Pi' \vdash \mathtt{new}\ c() : (c, \emptyset)$ and $\Pi' \vdash \mathtt{yield}\ \mathit{loc} : (c, \emptyset)$, so by Lemma C.32, $\Pi' \vdash \mathbb{E}[\mathtt{yield}\ \mathit{loc}] : (t, \rho')$.

Field Get In this case $e = \mathbb{E}[\mathit{loc}.f]$, $e' = \mathbb{E}[\mathtt{yield}\ v]$ (where $\mu(\mathit{loc}) = [u.F.E]$ and $F(f) = v$), and $\mu' = \mu$. Let $\Pi' = \Pi$. Clearly $\Pi' \approx \mu'$, and $\Pi \lessdot \Pi'$.

We now show that $\Pi \vdash \mathbb{E}[\mathtt{yield}\ v] : (t', \rho')$ for some $t' <: t$ and some ρ'. We have $\Pi \vdash \mathit{loc}.f : (s, \{\mathtt{read}\ f\})$. The last step in this derivation must be (T-GET) or (T-GET-OPEN-LOC). By the first hypothesis of (T-GET), (T-GET-OPEN-LOC) and by (T-LOC), and by $\Pi \approx \mu$, we have $\Pi(\mathit{loc}) = u$. By the second hypothesis of (T-GET), $\mathit{typeOfF}(f) = (c, s)$. By the second hypothesis of (T-GET-OPEN-LOC), $\mathit{typeOfF}(f) = (c, \mathtt{@open}\ s)$. Also by $\Pi \approx \mu$, if (a) $\mu(v) = [u'.F'.E']$, then $\Pi(v) = u'$ and $u' <: s$; otherwise (b) $\mu(v) = \mathtt{null}$. In both cases, the type of v is subtype of s, by Lemma C.33 (Replacement with subtyping), $\Pi \vdash E[\mathtt{yield}\ v] : (t', \rho')$.

Field Set Here $e = \mathbb{E}[\mathit{loc}.f = v]$, $e' = \mathbb{E}[\mathtt{yield}\ v]$, $\mu_0 = \mu \oplus (\mathit{loc} \mapsto [u.F \oplus (f \mapsto v)])$, $\mu' = \mathit{update}(\mu_0, \mathit{loc}, f, v)$, and $\mu(\mathit{loc}) = [u.F.E]$. Let $\Pi' = \Pi$, thus $\Pi \lessdot \Pi'$. We now show that $\Pi \approx \mu'$. Observe that the *update* function changes the effect mapping E in each of the object record, but not the fields F, which have no impact on $\Pi \approx \mu'$, by Definition C.26. Here $\mu'(\mathit{loc}) = [c.F \oplus (f \mapsto v).E']$, for some E'. For part 1(a) $\Pi(\mathit{loc}) = u$, since $\mu(\mathit{loc}) = [u.F.E]$ and $\Pi \approx \mu$. For part 1(b) $\mathit{dom}(F \oplus (f \mapsto v) = \mathit{dom}(\mathit{fields}(u))$, since $\mathit{loc}.f = v$ is

well-typed. For part 1(c), $rng(F \oplus (f \mapsto v)) \subseteq rng(F) \cup \{v\}$. Now since $loc.f = v$ is well-typed, then $v \in dom(\Pi)$ or $v = \texttt{null}$. In the former case, by $\Pi \approx \mu$, then $v \in dom(\mu)$. $v \in dom(\mu)$ implies $v \in dom(\mu')$. In either case $rng(F) \cup \{v\} \subseteq dom(\mu') \cup \{\texttt{null}\}$. Part 1(d) holds for all $f \in dom(F)$, $f' = f$. Part 1(d) holds vacuously for f if $v = \texttt{null}$. Otherwise, $(F \oplus (f \mapsto v))(f) = v$, and by (T-SET) or (T-SET-OPEN-LOC) and (T-LOC), $\Pi(v) <: s'$, where $fields(u) = (c, s')$ and $u <: c$. Parts 2 holds since $dom(\mu') = dom(\mu)$.

To see $\Pi \vdash e' : (t, \rho)$, let $\Pi \vdash loc.f = v : (s, \rho_0)$. By (T-SET) or (T-SET-OPEN-LOC), $\Pi \vdash v : (s, \emptyset)$ and Lemma C.32 (Replacement), $\Pi \vdash \mathbb{E}[\texttt{yield } v] : (t, \rho_1)$.

Method Call Here $e = \mathbb{E}[loc.m(\overline{v})]$, $e' = \mathbb{E}[\texttt{yield } e_1]$, $\mu(loc) = [u\,.\,F\,.\,E]$, $(findMeth(u,m) = (u', t, m(\overline{t\ var})\{e_2\}, \rho_0)$, $\mu' = \mu$ and $e_1 = [loc/\texttt{this}, \overline{v/var}]e_2)$. Let $\Pi' = \Pi$. Clearly $\Pi' \approx \mu'$, and $\Pi \lessdot \Pi'$.

We now show that $\Pi \vdash e' : (t', \rho')$ for some $t' <: t$ and some ρ'. $\Pi \vdash e : (t, \rho)$ implies that $loc.m(\overline{v})$ and all its subterms are well-typed in Π. By part 1(a) of $\Pi \approx \mu$, $\Pi \vdash loc : (u, \emptyset)$. By the definition of *findMeth*, $u <: u'$. Let $\Pi \vdash v_i : (u_i, \emptyset)$ $\forall i \in \{1..n\}$ and let $\Pi \vdash loc.m(\overline{v}) : (t_m, \rho_m)$. This last judgment must be (T-CALL), with $(u', t_m, m(\overline{t\ var})\{e_2\}, \rho_m) = findMeth(u,m)$, where $\forall i \in \{1..n\}$ :: $u_i <: t_i$. By the definition of the function *findMeth*, rules (T-METHOD) and *override*, $(\overline{var : t}, \texttt{this} : u') \vdash e_2 : (u'_m, \rho_1)$, and $u'_m <: t_m$. By Lemma C.31 (Environment extension) (and appropriate alpha conversion of free variables in e_2), $\Pi, \overline{var : t}, \texttt{this} : u' \vdash e_2 : (u'_m, \rho_1)$. By Lemma C.30 (Substitution), $\Pi \vdash [loc/\texttt{this}, \overline{v/var}]e_2 : (u'', \rho_1)$, for some $u'' <: u'_m <: t_m$. Finally, Lemma C.33 (Replacement with subtyping) gives $\Pi \vdash e' : (t', \rho')$ for some $t' <: t$.

Local Declaration In this case $e = \mathbb{E}[t\ var = v; e_1]$, $e' = \mathbb{E}[\texttt{yield } e'_1]$, where $e'_1 = [v/var]e_1$ and $\mu' = \mu$. Let $\Pi' = \Pi$. Obviously $\Pi' \approx \mu'$, and $\Pi \lessdot \Pi'$. We show $\Pi \vdash \mathbb{E}[\texttt{yield } e'_1] : (t', \rho')$, for some $t' <: t$. $\Pi \vdash e : (t, \rho)$ implies that $t\ var = v; e_1$ and all its subterms are well typed in Π, let $\Pi \vdash t\ var = v; e_1 : (s, \rho_0)$. By (T-DEFINE), $\Pi, var : t \vdash e_1 : (s, \rho_0)$. By Lemma C.30 (Substitution), $\Pi \vdash [v/var]e_1 : (s', \rho_1)$, for some $s' <: s$. Finally, Lemma C.33 (Replacement with subtyping) gives $\Pi \vdash e' : (t', \rho')$ for some $t' <: t$.

Fork-Sequential Here $e = \mathbb{E}[\texttt{fork}\,(e_2; e_3)]$, $e' = \mathbb{E}[\texttt{yield } e_0]$, $\mu' = \mu$ and $e_0 = e_2; e_3; \texttt{null}$. Let $\Pi' = \Pi$. Clearly $\Pi' \approx \mu'$, and $\Pi \lessdot \Pi'$.

We now show that $\Pi \vdash e' : (t', \rho')$ for some $t' <: t$ and some ρ'. By (T-FORK), $\Pi \vdash \texttt{fork}\ (e_2; e_3) : (\texttt{void}, \rho_0)$. Because the expression is well type, e_2 and e_3 are well-typed. We know $\Pi \vdash \texttt{null} : (\texttt{void}, \emptyset)$, so by (T-SEQ), $\Pi \vdash (\texttt{yield } e_0) : (u, \rho_1)$, for any valid type u and some ρ_1. Since the type of `null` is subtype of any type, Lemma C.33 (Replacement with subtyping) gives $\Pi \vdash e' : (\texttt{void}, \rho')$.

Fork-Parallel Here $e = \mathbb{E}[\texttt{fork}\ (e_0; e_1)]$, $e' = \mathbb{E}[\texttt{yield null}]$ and $\mu' = \mu$. Let $\Pi' = \Pi$. Clearly $\Pi' \approx \mu'$ and $\Pi \lessdot \Pi'$.

We now show that $\Pi \vdash e' : (t', \rho')$ for some $t' <: t$ and some ρ'. By (T-FORK), $\Pi \vdash \texttt{fork}\ (e_0; e_1) : (\texttt{void}, \rho_0)$. We know $\Pi \vdash \texttt{null} : (\texttt{void}, \emptyset)$, $\Pi \vdash (\texttt{yield null}) : (u, \emptyset)$, for any valid type u and some ρ_1. Since the type of `null` is subtype of any type, Lemma C.33 (Replacement with subtyping) gives $\Pi \vdash e' : (t', \rho')$ for some $t' <: t$.

Next, we show that $\Pi \vdash \psi$. Because the expression is well type, e_2 and e_3 are well-typed.

Yield In this case, $e = \mathbb{E}[\texttt{yield } e_1]$, $e' = \mathbb{E}[e_1]$. There are two cases: (a) there is no reduction step happens for other tasks during this reduction; (b) there are reduction steps happen for other tasks during this reduction.

In the first case, $\psi' = \psi$ and $\mu' = \mu$. Let $\Pi' = \Pi$, then $\Pi' \approx \mu'$, $\Pi \lessdot \Pi'$, and $\Pi' \vdash \psi'$. Let $\Pi \vdash (\texttt{yield } e_1) : (t_0, \rho_0)$, then $\Pi \vdash e_1 : (t_0, \rho_0)$, thus by Lemma C.32 (Replacement), $\Pi \vdash e' : (t, \rho)$.

For the second case, we already showed that for the other eight basic steps above all the conditions hold. For each of the reductions that do not reduce the current task, there is a Π'' such that all the condition holds. By Definition C.28 and Definition C.27, if $\Pi \vdash (\texttt{yield } e_1) : (t, \rho)$, by Lemma C.31 (Environment extension), $\Pi' \vdash e_1 : (t, \rho)$, thus by Lemma C.32 (Replacement), $\Pi \vdash e' : (t, \rho)$ and all other conditions hold. ∎