Trust, but verify: Optimistic Effect Analysis for Reusable Code

Yuheng Long  
Iowa State University, csgzlong@iastate.edu

Hridesh Rajan  
Iowa State University

Follow this and additional works at: http://lib.dr.iastate.edu/cs_techreports

Part of the Programming Languages and Compilers Commons, and the Software Engineering Commons

Recommended Citation

http://lib.dr.iastate.edu/cs_techreports/212

This Article is brought to you for free and open access by the Computer Science at Digital Repository @ Iowa State University. It has been accepted for inclusion in Computer Science Technical Reports by an authorized administrator of Digital Repository @ Iowa State University. For more information, please contact hinefuku@iastate.edu.
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

Copyright (c) 2012, Yuheng Long, and Hridesh Rajan.

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

1 Introduction ........................................ 1
   1.1 The Problems and Their Importance .......... 1
   1.2 Contributions to the State-of-the-Art: Open Effects .. 2

2 Optimistic Effect Analysis for Reusable Code .......... 3
   2.1 Using Open Effects in Sorting Algorithms .... 3
   2.2 Using Open Effects in Search Algorithms .... 4
   2.3 Using Open Effects with Map-Reduce Framework .. 4

3 A Concurrent Object-oriented Calculus ................. 5

4 Type and Static Effect Computation .................. 5
   4.1 Notations and Conventions ................. 5
   4.2 Type-and-Effect Rules for Method Declaration .... 5
   4.3 Open Effects of Polymorphic Method calls .... 6
   4.4 Type-and-efferts for Field Related Expressions ... 6

5 Dynamic Semantics with Open Effects ................. 6
   5.1 Notations and Conventions ................. 6
   5.2 Dynamic Effect Management in OO Expressions ... 7
   5.3 Safe Optimistic Concurrency using Open Effect ... 8
   5.4 Key Properties of OpenEffectJ ............ 8
   5.5 Open References . ..................... 9

6 Adding Open Effects to OpenJDK .................... 9
   6.1 Effect Storage and Maintenance ............ 10
   6.2 Performance Evaluation ................. 10

7 Comparative Analysis with Related Work ............ 10
   7.1 Overview of Closely Related Ideas ........ 10
   7.2 Criteria and Analysis Results ............ 11
   7.3 Scope and Applicability of Open Effects .... 11

8 Conclusion and Future Work ......................... 11

References ........................................ 12

A Type-and-effect System: Omitted Details ........ 13
   A.1 Type-and-Effect Rules for Declarations .... 13
   A.2 Type-and-Effect Rules for Expressions ... 13

B Dynamic Semantics: Omitted Details ............... 13

C Proof of Key Properties ................................ 14
   C.1 Preliminary Definitions .................. 14
   C.2 Effect Preservation ..................... 15
   C.3 Deterministic Semantics ............... 17
   C.4 Type Soundness ......................... 18
Open Effects

Yuheng Long and Hridesh Rajan
Dept. of Computer Science, Iowa State University
(csg@long.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, 12, 34, 42] and dynamic [7, 16, 18, 39] analyses have been proposed to help programmers write correct concurrent software. In essence, these techniques compute sets of computational effects [20, 29, 41] for each statement of a program to determine whether they 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 [35]. 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 [35 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 Table 1), e.g., STM-based solutions are reported to have 2X overhead [40], 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].
1.2 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 [41].

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 of the method call c.run(…) on line 6 could be run in parallel, else it must be run sequentially. Concrete effects include reads and writes to memory regions [41].

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 [5] 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.

On a different day, the developer of PROGRAM imports the ArrayList class. At runtime, PROGRAM creates an instance al of ArrayList, and passes an instance p of class Prefix on line [1] 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 writes 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 [1] in Figure 1). Thus, the for loop on line [5] is run sequentially when method applyall is called on line [1] 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][34]. 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 shows that most concurrency is exposed via libraries [6][23].

- It does not require annotating methods in a supertype 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 5 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.

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 5, 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.

```java
class Arrays {
    final Object[] mergeSort(@open Comparator c, Object[] src, int low, int high) {
        int size = high - low;
        Object[] dest = new Object[size];
        if (size < THRESHOLD){ //Use insertion sort
            System.arraycopy(src, low, dest, 0, size);
            for (int i=0; i<size; i++){
                for (int j=i; j>0 &&
                    c.compare(src[j-1], src[j])>0; j--)
                this.swap(dest, j, j-1);
            }
        }
        int mid = (low + high) / 2;
        Object[] d1 = this.mergeSort(c, src, low, mid);
        Object[] d2 = this.mergeSort(c, src, mid, high);
        /* the details of merge omitted. */
    }
}
```

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. 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.

```java
Comparator cPure = new...//Pure comparator
Comparator cDirty = new...//Comparator w/ effects
Arrays a = new Arrays();
// Following will do a parallel sort.
Object[] d1 = a.mergeSort(cPure,src,0,src.length);
// Following will do a sequential sort.
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 18. Due to this binding, potential side-effects of the call to `mergeSort` are updated. Since the method `mergeSort`, if called with the receiver object `cPure`, would have no side-effect, the call to `mergeSort` also has no externally visible side-effects. So, the sort initiated on line 14 can be parallelized.

Later, the same `open` parameter \( c \) is bound to `cDirty` on line 19. Due to this, potential side-effects of the method call to `mergeSort` are updated. Since the method `mergeSort`, if called with the receiver object `cDirty`, would have side-effects, the method call to `mergeSort` may also have externally visible side-effects. So, the sort initiated on line 22 can be parallelized.

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 [24]. For extensibility and reuse, this library is designed to use abstract implementation of `Goal` and `Node`. 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 16 to implement application-specific search objectives.

```java
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                  for (int i = 0; i < curr.children.length; i++)
10                      if (goal.satisfied(curr.children[i]))
11                          return found;
12              }
13              HashSet mark = new HashSet();
14              HashSet rs = new HashSet();
15          }
16          @open class Node {
17              final Goal[] children;
18          }
19          @open class Goal {
20              boolean satisfied(Node node){ return false; }
21          }
22      }

When the input range is small, the base case applies, on lines 10-19 where the map method is applied on each element of the array on line 13. The results from the Mapper are combined, on line 20 by the Reducer. The algorithm divides the array into non overlapping subarrays and recursively applies compute on these subarrays, on lines 19-20.

This algorithm could be parallelized by executing the compute on the subarrays concurrently, line 19-20. Since we may not know about the effect of each subclass of the Mapper and Reducer, we declare them as `open` fields, lines 4-5 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 OpenDK. 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: \textit{loc} that represents locations and \textit{yield} that models concurrency.

\begin{align*}
\text{prog} &::= \text{decl} e \\
\text{decl} &::= \text{class} c \text{ extends} d \{ \text{field} \ \text{meth} \} \\
\text{field} &::= \{ \text{open} c; f \} \\
\text{meth} &::= \text{t} m \{ \text{arg} \} \{ e \} \\
\text{t} &::= \text{c} \text{ void} \\
\text{arg} &::= \text{var} \ {\text{null}} \ | \ \text{arg} = e \ ; \ e \ ; \ e \\
\text{e} &::= \text{var} \ | \ \text{int} \ | \ \text{fork} (e; e) \ | \ \text{loc} \ {\text{yield}} \ e
\end{align*}

where \(c \in C\), a set of class names \(d \in \mathcal{D} \cup \{\text{Object}\}\), superclass names \(f \in \mathcal{F}\), a set of field names \(m \in \mathcal{M}\), a set of method names \(\varnothing \in \mathcal{V} \cup \{\text{this}\}\), \(\mathcal{V}\) is a set of variable names

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure4.png}
\caption{The Grammar for OpenEffectJ.}
\end{figure}

The notation over-bar denotes a finite ordered sequence (\(\pi\) 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., \text{int} is used as a shorthand for \text{Integer}, \(i < j\) is for \text{if} \(j()\), etc. As in Fork/Join framework [26], we desugar a for loop to the code below.

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

OpenEffectJ provides one construct for expressing parallelism: the \textit{fork} expression. This expression has the form \textit{fork} (\text{e}_0, \text{e}_1). It runs \text{e}_0 and \text{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 [26]‘s semantics is that this parallelization decision is based upon the information available at the time of the evaluation of a \textit{fork} expression. Delaying parallelization decision allows the use of useful dynamic information. This is similar to Best et al.’s synchronisation via scheduling [5], but differs from static [34] and dynamic analysis [13] (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 a 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 \textit{open}, using the annotation \[\text{open}\] (e.g. line 2 in Figure 2).

To focus our attention on the essence of \[\text{open}\] effects and to make the presentation tractable, we have formalized a subset of OpenEffectJ. In the core language, only fields can be annotated \[\text{open}\], 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 \[\text{open}\] 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 \[\text{open}\ f m \rho\]. It contain the name of the open field \(f\), the method \(m\) being invoked, and a placeholder for concrete effects \(\rho\).

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 \(f\) is set. OpenEffectJ uses these concretized effects to make concurrency decisions, when evaluating the \textit{fork} expressions.

4.1 Notations and Conventions

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

\begin{align*}
\theta &::= \text{OK} & \text{program/decl/body types} \\
| (t_1 \times \ldots \times t_n \rightarrow t, \rho) \in c & \text{method types} \\
| (t, \rho) & \text{expression types} \\
\rho &::= \{e_j\}_{j \in \mathbb{N}} & \text{program effects} \\
\epsilon &::= \text{read } f & \text{read effect} \\
| \text{write } f & \text{write effect} \\
| \text{open } f m \rho & \text{open effect} \\
| \bot & \text{bottom effect} \\
\Pi &::= \{\{\text{var}_j \mapsto t_j\}_{j \in \mathbb{N}}\} & \text{type environments}
\end{align*}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure5.png}
\caption{Type and effect attributes.}
\end{figure}

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 \(\rho_0 \triangleright \rho_1\) to mean that effects \(\rho_0\) and \(\rho_1\) do not interfere and \(\rho_0 \triangleright \rho_1\) 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 \(\Pi\), which is a finite partial mapping from variable names \(\text{var}\) to a type \(\rho\). Each method in the class table \(CT\) contains its effect \(\rho\), 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 4.1 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 \(\rho\).

\begin{align*}
\text{(T-METHOD)} &\quad \text{override}(m, c, (t_1 \times \ldots \times t_n \rightarrow t)) \\
&\quad \forall i \in \{1..n\}, \text{isClass}(t_i) \Rightarrow \text{isType}(t) \\
&\quad \{\text{var}_1 : t_1, \ldots, \text{var}_n : t_n \text{ this} : c\} \vdash e' : (u, \rho) \quad u < t \\
&\quad \{\text{t} m(\text{var}_1, \ldots, \text{var}_n) [e] : (t_1 \times \ldots \times t_n \rightarrow t, \rho) \in c
\end{align*}

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 [21] and a purity analysis [25] to improve the precision of the static effect analysis.
This rule uses a function `override` (below). Unlike some static type-and-effect systems for concurrency [34], OpenEffectJ does not require that method overriding implies effect containment.

\[
CT(c) = \text{class } c \text{ extends } d \{ \text{field meth}_1, \ldots, \text{meth}_p \} \\
\exists i \in \{1, \ldots, p\} \text{ s.t. } \text{meth}_i = t(m(t_1 \text{var}_1, \ldots, t_n \text{var}_n)\{e\}) \\
\quad \text{override}(m,d,(t_1 \times \ldots \times t_n \rightarrow t)) \\
\bar{d} \in \{1, \ldots, p\} \text{ s.t. } \text{meth}_\bar{d} = t(m(t_1 \text{var}_1, \ldots, t_n \text{var}_n)\{e\}) \\
\quad \text{override}(m,c,(t_1 \times \ldots \times t_n \rightarrow t)) \\
\bar{e}, \bar{p}' = \text{findMeth}(c,m) \\
\quad \text{override}(m,c,(t_1 \times \ldots \times t_n \rightarrow t))
\]

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

\[
CT(c) = \text{class } c \text{ extends } d \{ \text{field meth}_1, \ldots, \text{meth}_p \} \\
\exists i \in \{1, \ldots, p\} \text{ s.t. } \text{meth}_i = (t,p.m(T\text{var})\{e\}) \\
\quad \text{findMeth}(c,m) = (t,p.m(T\text{var})\{e\},p)
\]

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.

\[
(T\text{-CALL}) \\
e_0 \neq \text{this}.f \lor (e_0 = \text{this}.f \land \text{typeOf} f(e_0) \neq (\text{void}, \text{open } c_0)) \\
\quad \text{findMeth}(c_0,m) = (t_1, t.m(T\text{var})\{e_0\}, \rho) \\
\quad \Pi \vdash e_0 : (c_0, \rho_0) \\
\quad \forall i \in \{1, \ldots, n\} : \Pi \vdash e_i : (t'_i, \rho_0) \land t'_i < t_i \\
\Pi \vdash e_0, e_1, \ldots, e_n : (t, \{\})
\]

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 [21]), 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\text{-CALL-OPEN}) applies when a method `m` is called with an open field `f` as a receiver object.

\[
(T\text{-CALL-OPEN}) \\
\Pi \vdash \text{this}.f : (c_0, \rho_0) \\
\quad \text{typeOf} f(e_0) = (d, \text{open } c_0) \\
\quad \text{findMeth}(c_0,m) = (t_1, t.m(T\text{var})\{e_0\}, \rho) \\
\quad \forall i \in \{1, \ldots, n\} : \Pi \vdash e_i : (t'_i, \rho_0) \land t'_i < t_i \\
\Pi \vdash \text{this}.f.m(e_1, \ldots, e_n) : (t, \{\text{open } f \text{ m } 0\} \cup \bigcup_{i=0}^{n} \rho_i)
\]

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 \( \rho_0 \) and \( \rho_1 \). If neither \( \rho_0 \) nor \( \rho_1 \) contains any open effect (i.e., \( \exists f, m, \rho \text{ s.t. } \forall i \in \{0,1\} : \{\text{open } f \text{ m } \rho \} \notin \rho \)), 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 \( \rho_0 \) and \( \rho_1 \) do not interfere, i.e. \( \rho_0 \parallel \rho_1 \). If \( \rho_0 \) and \( \rho_1 \) interfere (\( \rho_0 \not\parallel \rho_1 \)), it suggests sequential execution.

Consider the scenario where open effect exists in \( \rho_0 \) and/or \( \rho_1 \) (i.e., \( \exists f, m, \rho \text{ s.t. } \forall i \in \{0,1\} : \{\text{open } f \text{ m } \rho \} \)). Let \( \rho \) be the concrete effects in set \( \rho \), i.e., \( \rho = \{\{e \in \rho \} \wedge (e \neq \{\text{open } f \text{ m } \rho \}) \} \). At compile time, OpenEffectJ advises a parallelizing compiler to parallelize `id_0` and `id_1` if \( \rho_0 \not\parallel \rho_1 \), 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\text{-GET}) and (T\text{-SET}) are standard. The auxiliary function `typeOf\text{F}` uses the class 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`.

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

\[
(T\text{-SET}) \\
\Pi \vdash e : (c, \rho) \\
\quad \text{typeOfF}(f) = (c', t) \\
\Pi \vdash e' : (t', \rho') \\
\Pi \vdash e.f = e' : (t', \rho' \cup \{\text{write } f\})
\]

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\text{-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.

\[
(T\text{-YIELD}) \\
\Pi \vdash e : (t, \rho) \\
\Pi \vdash \text{yield} : (t, \rho) \\
\Pi \vdash \text{fork} (e' : (\text{void}, \rho' \cup \{\})
\]

Concurrence expressions: The (T\text{-YIELD}) rule says that a yield expression has the same type and effect as the expression `e`. The (T\text{-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 \( \Sigma \) in Figure 6 to another. Some rules use an implicit attribute, the class table `CT`.
Evaluation relation: $\ni \rightarrow: \Sigma \rightarrow \Sigma$

Domains:

$\Sigma ::= \langle \psi, \mu \rangle$ "Program Configurations"
$\psi ::= \langle e, \tau \rangle + \psi | \bullet$ "Task Queue"
$\tau ::= (id, (id_j | j \in \mathbb{N}))$ "Task Dependencies"
where $id, id_j \in \mathbb{N}$
$\mu ::= \langle loc \mapsto \tau \rangle_{\eta \in \mathbb{N}}$ "Store"
$\sigma ::= [c, \mathbf{F}, \mathbf{E}]$ "Object Records"
$F ::= \langle f_j \mapsto v_j | j \in \mathbb{N} \rangle$ "Field Maps"
$\nu ::= \text{null} | \text{loc}$ "Values"
$E ::= \langle m_j \mapsto \rho_j \rangle_{\eta \in \mathbb{N}}$ "Effect Maps"

Evaluation contexts:

$E ::= - | E, m(\tau) | v, m(\tau, E, \pi) | E.f = e | v, f = E | E.f; e | E.f / t var = E; e$

Figure 6. Notations Used in the Dynamic Semantics.

A configuration consists of a task queue $\psi$ and a global store $\mu$. A store maps locations to object records. An object record $\sigma = [c, \mathbf{F}, \mathbf{E}]$ 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 $\psi$ consists of a list of tasks $(e, \tau)$. Each task consists of an expression $e$ and the corresponding task dependencies $\tau$. The expression $e$ serves as the remaining evaluation for the task. The task dependencies $\tau$ 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 $E$ 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 $\Sigma_s = \langle e, (0, 0) \rangle \bullet$. The operator $\ominus$ is an overriding operator for finite functions, i.e. if $\mu' = \mu \ominus \langle loc \mapsto o \rangle$, 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 3 contains omitted auxiliary functions). Compared to traditional dynamic semantics for OO expressions [19], there are two main differences. First, the $\text{yield}$ expression is used in the resulting configuration to relinquish control to other tasks. Second, some of the rules manipulate the dynamic effect map $E$.

\begin{align*}
\text{(NEW)}
loc \notin \text{dom}(\mu) \\
\mu' &= \langle loc \mapsto [c, \{f \mapsto \text{null} | f \in \text{fields}(c)\}] \rangle \ominus \mu \\
\end{align*}

\begin{align*}
\langle E | \text{new} c() | \tau \rangle + \psi, \mu' &\hookrightarrow \langle E | \text{yield} \text{loc}, \tau \rangle + \psi, \mu' \\
\end{align*}

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

\begin{align*}
\text{methE}(c) &= E \cup \bigcup_{i=0}^{n} \langle m_i \mapsto \rho_i \rangle \\
\text{where } CT(c) &= \text{class } c \text{ extends } d \{ \text{field meth1 ... methn} \} \\
\text{and } \text{methE}(d) &= E \\
\text{and } \forall i \in \{1...n\} :: \text{findMeth}(c, m_i) = (\{ c.t, m_i(\text{var}), \rho_i \})
\end{align*}

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 $\text{update}$ shown below implements this.

\begin{align*}
\text{(GET)}
\mu(loc) &= [c,F,E] \\
\nu &= F(f) \\
\mu' &= \text{update}(\mu, loc, \nu, E) \\
\begin{cases}
\langle E | loc.f(v), \tau \rangle + \psi, \mu' &\hookrightarrow \langle E | \text{yield} v, \tau \rangle + \psi, \mu' \\
\end{cases}
\end{align*}

\begin{align*}
\text{(SET)}
\mu(loc) &= [c,F,E] \\
\mu_0 &= \mu \oplus \langle loc \mapsto [c,F \oplus (f \mapsto v)], E \rangle \\
\mu' &= \text{update}(\mu_0, loc, \nu, E) \\
\begin{cases}
\langle E | loc.f(v), \tau \rangle + \psi, \mu' &\hookrightarrow \langle E | \text{yield} v, \tau \rangle + \psi, \mu' \\
\end{cases}
\end{align*}

The function $\text{update}$ updates the effect (by calling $\text{updateEff}$) 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 updated and propagated using the function $\text{fixPoint}$ until a fixed point is reached.

\begin{align*}
\text{reverse}(\mu, loc) &= \bigcup_{i=1}^{n} S_i \\
\text{where } \forall i \in \{1...n\} \text{ s.t. } loc_i \in \text{dom}(\mu) :: \\
S_i &= \{ (loc_i, f) | F(f) = loc \land \mu(loc_i) = [c,F,E] \}
\end{align*}

The function $\text{reverse}$ searches the input store $\mu$ of objects $loc$, and field $f$, that is pointing to the current object $loc$ in practice, reverse pointers could be used to optimize this update [5].

\begin{align*}
\text{updateEff}(\mu, f, v, \rho) &= (m, \rho') \\
\text{where } \forall i \in \{1...n\} \rho_i = [e_i | 1 \leq k \leq p] \land \rho'_i = [e'_i | 1 \leq k \leq p]
\end{align*}

Each object contains a map $E$ of effects. The $\text{updateEff}$ function concretizes the effects in $E$ one by one, by calling $\text{concretize}$.

\begin{align*}
\text{concretize}(\mu, f, v, \epsilon) &= \text{match } \epsilon \text{ with} \\
| \text{open } f' & m \rho \rightarrow \text{match } f' \text{ with} \\
| f & \rightarrow \text{match } v \text{ with} \\
| \text{null} & \rightarrow \text{open } f \ m \emptyset \\
| \text{loc} & \rightarrow \text{open } m \rho' \text{ where } [c,F,E] = \mu(loc), \\
\text{and } \rho' &= \bigcup_{i=1}^{n} \rho_i, \text{ and } E(m) = [e_i | 1 \leq i \leq n] \\
\text{and } \forall i \in \{1...n\} :: \text{cp}(e_i) = \rho_i \\
| \_ & \rightarrow \epsilon
\end{align*}

The function $\text{concretize}$ 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., \text{open } f m \rho.

\begin{align*}
\text{cp}(e) &= \text{match } \epsilon \text{ with} \\
| \text{open } f m \rho & \rightarrow \rho \\
| \_ & \rightarrow \epsilon
\end{align*}

The function $\text{cp}$ is used by the function $\text{concretize}$ to retrieve the concrete effects, i.e., the effects of the $\text{R-value}$ $v$ 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 5.1) that uses dynamic dispatch[19].

\[
\text{(CALL)} \quad (c', \lambda.M(t_1 \ldots t_n)(e), \rho) = \text{findMeth}(c.m)
\]

\[
E.M = \mu(\text{loc}) \quad e' = [\mu(\text{this})v_1 \ldots v_n][e_n] \quad (E(M[v_1 \ldots v_n]), \tau) + \psi, \mu \iff ([E(\text{yield} e'), \tau] + \psi, \mu)
\]

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 \( \rho \) with respect to a store \( \mu \). 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.

![Figure 7. Effect judgment for expressions.](image-url)

The dynamic rules in Figure 7 are similar to the static rules in Section 5 except for method calls on open fields (E-CALL-OPEN). The open effect now has a concrete part \( \rho \), 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 \( \text{e_0}\text{e_1} \)). Read effects do not interfere; read/write and write/read pairs conflict if they access the same field; open effect \( \text{open f m} \) conflicts with another effect e if any effect e' in \( \rho \) conflicts with e; 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 4 for more discussion). The (FORK-PARALLEL) rule creates 2 concurrent children tasks id_0 and id_1 and put them into the queue \( \psi \). The current task is suspended until id_0 and id_1 are done. This is done by putting 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.

![Figure 8. Effect behavior of fork expression.](image-url)

(FORK-PARALLEL) \( \mu \vdash e_0 : \{e_1, \ldots, e_n\} \)

\[
\mu \vdash e_0 : \{e_1, \ldots, e_n\} \quad \forall i \in \{0..n\}, f \in \{0..n'\} \text{ s.t. } e_i(f) \quad \text{id}_0 = \text{fresh}(\text{id}) \quad \text{id}_1 = \text{fresh}(\text{id}) \quad \tau' = \{id, \{id_0, id_1\}\}
\]

\[
([E(\text{fork}(e_0; e_1)), \tau] + \psi, \mu) \iff ([E(\text{yield null}), \tau'] + \psi, \mu)
\]

The (FORK-SEQUENTIAL) rule constructs an expression e_0; e_1; null, which serializes 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.

![Figure 9. Effect behavior of fork expression.](image-url)

(FORK-SEQUENTIAL) \( \mu \vdash e_0 : \{e_1, \ldots, e_n\} \)

\[
\mu \vdash e_0 : \{e_1, \ldots, e_n\} \quad \exists i \in \{0..n\}, f \in \{0..n'\} \text{ s.t. } e_i(f) \quad \text{id}_0 = \text{fresh}(\text{id}) \quad \text{id}_1 = \text{fresh}(\text{id}) \quad \tau' = \{id, \{id_0, id_1\}\}
\]

\[
([E(\text{fork}(e_0; e_1)), \tau] + \psi, \mu) \iff ([E(\text{yield null}), \tau'] + \psi, \mu)
\]

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

![Figure 10. Effect behavior of fork expression.](image-url)

(YIELD) \( (e', \tau') + \psi' = \text{active} (\psi + \text{E}[e], \tau) \)

\[
(\langle e', \tau' \rangle + \psi' = \text{active} (\psi + \text{E}[e], \tau) \iff (\langle e', \tau' \rangle + \psi', \mu)
\]

Finishing an active task is done by the function active (not shown). It returns the top most task in \( \psi \) 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 \( \psi \) 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 [23], 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 \( \eta \) of a task \( id \) can be a read effect \( \text{rd loc, f, id} \) or a write effect \( \text{wt loc, f, id} \). A dynamic effect \( \eta \) refines a static effect \( \rho \), written \( \eta \preceq \rho \), if either \( \eta = \text{rd loc, f, id} \land \text{read f} \in \rho \); or \( \eta = \text{wt loc, f, id} \land \text{write f} \in \rho \).
The dynamic effect of a task id is a dynamic trace $\chi = \eta$, 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 $\gamma$, which is a map $[\{id_j \rightarrow I\}]_{j \in J}$. 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][24], 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.

<table>
<thead>
<tr>
<th>In the following, $\text{dyn}(\Sigma, \chi, \psi, \mu)$</th>
<th>Side Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>$[(E; \text{loc} \cdot f),(id,I)] + \psi, \mu$</td>
<td>$\chi = \chi + (\text{cd}, \text{loc}, f, \text{id}, \psi, \mu)$</td>
</tr>
<tr>
<td>$[(E; \text{fork}(e_{0}) \cdot e_{1}),(id,I)]$</td>
<td>$\chi = \chi + \psi, \mu$</td>
</tr>
<tr>
<td>$\Sigma \Rightarrow (\text{id} \rightarrow (\text{Y}((\text{id} \cup \emptyset)) \cup \Gamma)) + \psi, \mu$</td>
<td>$\chi = \chi, \psi, \mu$</td>
</tr>
</tbody>
</table>

The novelty of OpenEffectI is that it stores the effects $\rho$ in object records for methods. These effects will be used by the $\text{fork}$ expression at runtime. Therefore, two invariants (Definition 5.3), for these method effects $\rho$, are necessary to maintain the effect preservation property. These invariants include: the placeholder effect $\rho_0$ of an open effect $\text{open}$ f m $\rho_0$ should be superstructure $\subseteq$ 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 superstructure of the effect $\rho$ 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 superstructure of the effect of the method run of p:

**Definition 5.1.** [Well-formed object] An object record $\alpha = \langle E:F.E \rangle$ is a well-formed object in $\mu$, written $\mu \vdash \alpha$, if for any open effect $\text{open}$ f m $\rho_0 \in \rho$ in $\alpha$, either $(F(f) = \alpha) \land \mu(\alpha) = \langle \langle e', id'(I) \rangle + \psi, \mu \rangle \land E(m \subseteq \rho_0)$; or $(F(f) = \text{null}) \land (\rho_0 = \emptyset)$.

**Definition 5.2.** [Well-formed location] A location loc is well-formed in $\mu$, written $\mu \vdash \text{loc}$, if either $\mu(\text{loc}) = \langle \langle e', id'(I) \rangle + \psi, \mu \rangle \land E(m \subseteq \rho_0)$; or $(F(f) = \text{null}) \land (\rho_0 = \emptyset)$.

**Definition 5.3.** [Well-formed store] A store $\mu$ is well-formed, written $\mu \vdash \varnothing$, if $\forall \psi \in \rho \oplus \rho_0 \land \rho : \mu \vdash \text{loc}$.

**Theorem 5.4.** [Effect preservation] Let the program configuration be $\Sigma \Rightarrow c \cdot (e, (id,I)) + \psi, \mu$. If it transits to another configuration $\Sigma \Rightarrow c \cdot (e', (id',I')) + \psi, \mu'$, the store is well-formed $\mu \vdash \varnothing$, $\mu \vdash e \cdot \rho$, and $\Gamma \vdash \mu$, then there is some $\mu', \mu' \vdash e \cdot \rho$, and $\rho' \subseteq \rho$; and (b) $\text{dyn}(\Sigma, \chi, \psi, \mu) = (\Sigma, \chi', \psi', \mu')$.

**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 $\mu \vdash e \cdot \rho$ (Figure 7). We prove that the dynamic effect $\eta$ in each step refines the static $\rho$ of the original expression $e$, $\eta \subseteq \rho$. Thus with (a), $\eta$ refines the effect $\rho_0$ of the expression $e_0$ when the task was forked off, with the heap $\mu_0$, e.g., $\text{fork}(e_0; e_1)$ and $\mu_0 \vdash e_0; \rho_0$. Unlike the static approaches, which compute at compile-time, OpenEffectI computes $\rho_0$ before evaluating the fork expression.

### 5.4.2 Determinism

We prove the determinism of OpenEffectI programs by showing that the concurrent tasks do not have dynamic effect interference and therefore a well-typed OpenEffectI 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' $<_{T} id$), refine the static effect computed when id is forked off. The function dynE is: $\text{dynE}(\Sigma, \chi, \psi, \mu) = \text{dynE}(\Sigma, \chi', \psi', \mu', \mu)$, if dynE(\Sigma, \chi, \psi, \mu), (\Sigma, \chi', \psi') and dynE(\Sigma, \chi, \psi, \mu') = dynE(\Sigma, \chi', \psi', \mu', \mu).

A task id is a descendant of a task id', with $\psi$, written as $id' <_{T} id$, if $id' \in \text{desc}(id, \psi)$. Here, $\text{desc}(id, \psi) = I \cup \{id_0, \ldots, id_n\}$, and $\text{Id}_{i} \in I, \text{desc}(id, \psi) = I$.

To reason about the intersection between the two concurrent tasks, we define the newly generated queue $\psi'$, which is formed by these two tasks, i.e., we write $\Sigma \rightarrow \psi'$, if $\Sigma \rightarrow \psi', \Sigma = \langle \langle E, (\psi'), \tau \rangle + \psi, \mu \rangle$, and $\Sigma = \langle \langle E, (\psi'), \tau \rangle + \psi, \mu \rangle$. To say that concurrent tasks do not interfere (heap accesses), we define the noninterference relation for dynamic effects $(\eta \gamma \eta')$ 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 $\varphi = \langle \eta_1, \eta_2 \rangle$ do not conflict with another set $\varphi' = \langle \eta'_1, \eta'_2 \rangle$ if $\forall i \in \{1..\}, j \in \{1..\}$. s.t. $\eta_i \gamma \eta'_i$.

**Theorem 5.5.** [Deterministic semantic] Let $\mu \vdash \varnothing, \Sigma \rightarrow \psi', \Sigma = \langle \langle E, (\psi'), \tau \rangle + \psi, \mu \rangle$. For the two tasks $(e_1, (id_1, I)) \in \psi'$ and $(e_2, (id_2, I)) \in \psi'$, $\psi, \mu \vdash e_1; \mu \vdash e_2; \rho_2$ if $\rho_1 \oplus \rho_2$, and $\forall \eta \in \text{N}, \text{dynE}(\Sigma, \mu) = (\psi, \mu), \text{dynE}(\Sigma, \psi, \mu, (id_1, \chi), \text{dynE}(\Sigma, \psi, \mu, (id_2, \chi), \chi)$. Here $(\eta \in \chi) \land (\eta \gamma (\ldots, \eta, \ldots)) \land (\eta \gamma (\ldots, \eta, \ldots)) \land (\eta \gamma (\ldots, \eta, \ldots)) = \text{dynE}(\Sigma, \psi, \mu, (id, \chi), \chi)$.

**Proof Sketch:** The essence of the theorem is that for the 2 concurrent tasks id1 and id2, generated by a fork expression, if the store is well-formed $\mu \vdash \varnothing$, the effect judgments give them effect $\rho_1$ and $\rho_2$ respectively and they do not interfere $\rho_1 \oplus \rho_2$, then their dynamic effects (dynE(id1, \chi) returns the dynamic traces by the task id1) do not interfere dynE(id2, \chi), dynE(id2, \chi, \chi). The map $\chi$ records the children set (Section 5.1) of a task id. We need $\chi$ because the effect of a child task id should count as id’s effect.

Proving the above soundness theorems are non-trival compared to static effect approaches [13][24], in which the exact effect of a task is known statically. A technical challenge for proving the soundness of OpenEffectI 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 OpenEffectI to enable open variable and parameter. At compiler-time, OpenEffectI’s type-and-effect system will generate an open effect $\text{open} \varnothing \text{var} \mu \text{var} \varnothing$ for a method call $\text{var} \text{var} \ldots$ (3). At runtime, the concretization of the open effect happens in the fork rule. At runtime, if $\text{var}$ is bound to a location loc, the (E-CALL-Loc) rule in Figure 7 takes effect. It transits from the placeholder effect $\text{open} \varnothing \text{var} \mu \text{var}$ to the concrete effect $\rho_0$ (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. Apart from modifications to support the $\text{open}$ annotation and the fork expression, parsing remains unchanged. Type checking
(the Attribute and Flow phases in the OpenDK 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 [21], purity analysis [36], 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 the 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 4 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.

6.2 Performance Evaluation
We have conducted an initial evaluation of OpenEffectJ’s prototype compiler using the library classes described in Section 1 and Section 2. These are array list with hash computation (Hash), array list with slightly heavier computation (Heavy), merge sort algorithm (MergeSort), depth-first search algorithm (Search), and a map-reduce application (MapReduce). All experiments were run on a system with a total of 4 cores (Intel Core2 chips 2.40GHz) running Fedora GNU/Linux. For each of the experiments, an average of the results over ten runs was taken.

The Heavy variant of array list computes the following formula $\text{Math.sqrt}(2 \times \text{Math.pow}((\text{double})o, 2))$ for each element $o$ in the list. For both Hash and Heavy variant, array list contained 20 million elements. The merge sort algorithm sorts a list of 10 million randomly generated integers. The search algorithm searches for solutions to an n-queens problem, where $n$ is 11. In the map-reduce algorithm, map step computes the formula $\text{Math.sqrt}(2 \times \text{Math.pow}((\text{double})o, 2))$ for the element $o$ and reduce step is simply addition. It is applied to 100 Million integers.

The performance results of running these applications are shown in Figure 9. In this table, the columns marked Sequential shows the time taken by the single-threaded version of the application. The columns marked Manual shows the time taken by (and speedup of) the manually parallelized version, which does not manage effects (and thus could be unsafe). The columns marked OpenEffectJ shows the time taken by (and speedup of) a version of our compiler, which does effect comparison prior to forking off each task. The columns marked OpenEffectJ' shows the time taken by (and speedup of) an improved version of our compiler, which does an intra-procedural analysis to determine that effects will not change between one fork and next (and thus first effect check implies the second).

The manual (unsafe) version showed decent to good speedup for all of these applications. The OpenEffectJ and OpenEffectJ' version also showed comparable speedup. For the array list, results for both OpenEffectJ and OpenEffectJ' are the same. The overhead for this example was also small. For the MergeSort, Search and MapReduce examples, however, the overhead for OpenEffectJ increased significantly. This was because each of these problems use recursive, parallel algorithm. This was precisely where OpenEffectJ' shines, because this version compiler is able to eliminate the effect check for the nested fork expressions. Thus, this version outperformed our previous version on all of the recursive cases.

These results show that support for open effects can be provided in an industrial strength compiler such as the OpenDK compiler at a reasonable cost (single digit overheads). To put these overheads in perspective, a dynamic analysis can sometimes incur 2X overhead [15] Table 1. More attention to OpenEffectJ’s compiler prototype will help discover simple and more clever optimizations that will help decrease overheads even further.

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 [33]. 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] [24] [34] 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 [29] 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] [10] [15] [19]. In essence, they monitor memory footprints of tasks and signals when conflicts are detected. In contrast, OpenEffectJ detects conflicts just before forking off parallel tasks.

Transactional memory [22] [27] [28] [40] optimistically executes tasks concurrently, but monitors memory accesses. It rolls back side effects when conflicts happen. There are TM-like approaches [4] [15] [43] that provide sequential consistency (DTM) by enforcing a deterministic commit order, instead of rolling back non-deterministically. 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. Each 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 [38] and Hybrid Type Checking [25] 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:

<table>
<thead>
<tr>
<th>Work</th>
<th>Sequential</th>
<th>Manual (No effects/Unsafe)</th>
<th>OpenEffectJ (Improved)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>time (in secs)</td>
<td>time in (secs)</td>
<td>speedup</td>
</tr>
<tr>
<td>ArrayList(Hash)</td>
<td>0.15</td>
<td>0.10</td>
<td>1.34</td>
</tr>
<tr>
<td>ArrayList(Heavy)</td>
<td>1.09</td>
<td>0.29</td>
<td>3.76</td>
</tr>
<tr>
<td>MergeSort</td>
<td>2.65</td>
<td>1.10</td>
<td>2.41</td>
</tr>
<tr>
<td>Depth First Search</td>
<td>12.77</td>
<td>14.58</td>
<td>1.91</td>
</tr>
<tr>
<td>MapReduce</td>
<td>5.93</td>
<td>2.16</td>
<td>2.75</td>
</tr>
</tbody>
</table>

---

Figure 9. Experimental Results of Compiling and Running Example Applications using the Current OpenEffectJ Compiler.

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., p1 and p2, some of which p1 have data races, but these are not the hot paths in program. The others p2 have no side effect and are frequently executed. TM will win, because 1) p1 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) p2, 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 acquired 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 explicitly 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


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.

\[(T\text{-PROGRAM}) \quad \forall decl \in decl \vdash decl : OK \quad \vdash e : (t, \rho)\]

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.

\[(T\text{-CLASS}) \quad \forall fields \in field : validF(fields, d) \quad isClass(d) \quad \forall meth \in meth \vdash meth : (t_1, \rho_1) \text{ in } c \quad \vdash c \ extends d \ \{\text{field} \ meth\} : OK\]

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

The function typeOfF (used in the rules in Section 5.4) 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.

\[\text{typeOfF}(f) = (c, t) \quad \text{where s.t. } CT(c) = \text{class } c \ extends d \ \{\text{field}_1, \ldots, \text{field}_n \ \text{meth}\} \quad \text{and } \exists i \in \{1..n\} \ : \ \exists \ : \ fieldOf(field)_i = (f, t)\]

\[\text{fieldOf}(@open c f) = (f, @open c) \quad \text{fieldOf}(c f) = (f, c)\]

B. Dynamic Semantics: Omitted Details

This section presents auxiliary functions that were omitted in Section 5.3 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).

\[\text{fields}(c) = F_s \cup \{f_1, \ldots, f_n\} \quad \text{where } CT(c) = \text{class } c \ extends d \ \{\text{field}_1, \ldots, \text{field}_n, \ \text{meth}\} \quad \text{and } \text{fields}(d) = F_s \quad \text{and } \forall i \in \{1..n\} \ : \ fieldOf(field)_i = (f_i, t_i)\]

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 locᵢ and field fᵢ pairs that are pointing to the locᵢ whose effects have been changed. The effects of locᵢ are changed by calling the update function.

\[\text{fixPoint}(\mu, loc, \kappa) = \mu_0 \quad \text{where } \kappa = \{(loc_i, f_i) \mid 1 \leq i \leq n\} \quad \text{and } \text{update}(\mu, loc, f_i) = \mu_i \quad \text{and } \forall i \in \{2..n\} \ : \ \text{update}(\mu, loc_{i-1}, f_{i-1}, loc_i) = \mu_i\]

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 ν).

\[\text{active}(e, \tau) + \psi = (e, \tau) + \psi \quad \text{if } \text{intersect}(\tau, \psi) = \text{false} \quad \text{active}(e, \tau) + \psi = \text{active}(\psi + (e, \tau)) \quad \text{if } \text{intersect}(\tau, \psi) = \text{true}\]

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.

\[\text{intersect}((id, \emptyset), \psi) = \text{false} \quad \text{intersect}((id, \{id_1, \ldots, id_n\}), \psi) = \emptyset \quad \text{where } \forall i \in \{1..n\} \ : \ \text{inQueue}(id_i, \psi) = b_i \]

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

\[\text{inQueue}(n, *) = \text{false} \quad \text{inQueue}(n, \langle e, \langle n, \{id_j \mid j \in N\} \rangle \rangle + \psi) = \text{true} \quad \text{inQueue}(n, \langle e, \langle n', \{id_j \mid j \in N\} \rangle \rangle + \psi) = \text{inQueue}(n, \psi) \quad \text{if } n \neq n'\]

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 $\text{OpenEffectJ}$: Effect and Type Preservation, and Determinism. Some of the definitions, descriptions and proof sketches are also in Section C.4. We write all these for the sake of clarity.

We have proven the soundness of $\text{OpenEffectJ}$’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 $\text{OpenEffectJ}$ programs by showing that the concurrent tasks do not have dynamic effect interference and therefore a well-typed $\text{OpenEffectJ}$ 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, 34], in which the exact effect of a task is known statically. A technical challenge for proving the soundness of $\text{OpenEffectJ}$ 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 $\text{OpenEffectJ}$’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 [34]. To record the heap accessed for each task, we define dynamic trace $\chi$, 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 $\rho$ computed when $id$ is forked off, s.t. if the static effects do not interfere, the dynamic effects will not interfere $\rho'$ (Section C.4).

DEFINITION C.1. [Dynamic Trace] A dynamic trace ($\chi$) consists of a sequence of dynamic effects ($\forall \eta$), where $\eta$ can be a read effect (rd, loc, f, id) or write effect (wt, loc, f, id).

Next, we will introduce a relationship $\Upsilon$. It records the children tasks of a task $id$. The relationship $\Upsilon$ 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 $\Upsilon$ for tasks is a map $\{id \rightarrow I\}_{\eta \in \mathcal{E}}$. Here $id$ is a task’s identity and $I$ is a set of its children tasks’ identities.

The function $\text{dyn}$, defined in Figure 10, 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 $\rho$ computed when it is forked off.

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.1). 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.3. [Static effect inclusion] An effect $\varepsilon$ is included in an effect set $\rho$ if it is one of the elements in $\rho$; or there is an open effect $\@open f \ f p'$ in $\rho$ and $\varepsilon$ is an element of $p'$.

DEFINITION C.4. [Dynamic effect refines static effect] A dynamic effect $\eta$ refines a static effect $\rho$, where $(\bot, \eta, \rho)$, written $\eta \prec \rho$, if either $\eta = (\text{rd}, \text{loc}, f, id) \wedge \mathcal{E}(f) \in \rho$; or $\eta = (\text{wt}, \text{loc}, f, id) \wedge \mathcal{E}(f) \in \rho$.

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

In the following, $\mathcal{E}(f)$ needs to be maintained proper effect (2) of the effect of the method $m$ of the object the field $f$ is pointing to.
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 $\rho$ produced by an expression $e$ is refined by the effect $\rho'$, by its subexpression expression $e'$ ($\rho' \subseteq \rho$); and second, the dynamic effect $\eta$, produced by the reduction, if any, refines $\rho (\eta \sim \rho)$; finally, the store remains effect equivalent ($\mu \equiv \mu'$).

**Theorem C.10.** [Effect preservation] Let the program configuration $\Sigma = \langle \epsilon, (id, I) \rangle$. If it transforms to another program configuration $\Sigma' = \langle \epsilon', (id, I') \rangle$, the store is well-formed $\mu \vdash o$, $\mu \vdash e : \rho$, and $\bot \notin \rho$, then there is some $\rho', \chi$ s.t.

(a) $\mu \equiv \mu'$;
(b) $\mu \vdash e' : \rho'$, and $\rho' \subseteq \rho$;
(c) $(dyn(\Sigma, \chi, Y) = (\Sigma', \chi + \eta, Y')) \Rightarrow (\eta \sim \rho)$.

**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 $\rho$ to the same expression $e$. Because we will prove that during the concurrent execution of the tasks, all the stores are effect equivalent $\mu \equiv \mu'$. Therefore, it suffices to prove that the effect of the subexpression $e'$ refines the effect of the expression $e$, given by effect equivalent stores.

**Lemma C.11.** [Stationary effect] Let $e$ be an expression, $\mu$ and $\mu'$ two stores s.t. $\mu \equiv \mu'$. 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 $\rho$. 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 holds for all sub-derivations of the derivation being considered. The case for (SET-OPEN) and (CALL) hold because in these cases $\bot \notin \rho$.

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:

$$
\mu \vdash \epsilon_0 : \rho_0; \mu' \vdash \epsilon_1 : \rho_1
$$

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

**Call-Open**

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

<table>
<thead>
<tr>
<th>$\mu(\text{loc}) = {c.F.E}$</th>
<th>$E(m) = \rho_0$</th>
<th>$\mu'(\text{loc}) = {c'.F'.E'}$</th>
<th>$E(m) = \rho_1'$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\forall i \in {1, n} : \mu^i \vdash e_i : \rho_i$</td>
<td>$\mu \vdash \text{loc}(\tau) : \bigcup_{i=0}^{n} \rho_i$</td>
<td>$\forall i \in {1, n} : \mu^i \vdash e_i : \rho_i'$</td>
<td></td>
</tr>
</tbody>
</table>

Clearly, $\rho_0 = \rho_0' \equiv \{\text{read } f\}$. Since $\mu \equiv \mu'$, the effect maps $E$ are the same. By the IH, $\forall i \in \{1, n\} : \epsilon_i = \epsilon_i$. Thus $\rho' = \{\text{open } f \ m \ \rho_i'\} \subseteq \bigcup_{i=0}^{n} \rho_i', \forall i \in \{1, n\} \epsilon_i \neq \epsilon_i$ does not change, thus $\rho_1' = \rho_1'$. Therefore, the claim holds.

**Call-Loc**

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

$$
\mu(\text{loc}) = \{c.F.E\} \\
E(m) = \rho_0 \\
\forall i \in \{1, n\} : \mu^i \vdash e_i : \rho_i
$$

Since $\mu \equiv \mu'$, the effect maps $E$ are the same and $\rho_0 = \rho_1'$. By the induction hypothesis, $\forall i \in \{1, n\} : \epsilon_i = \epsilon_i$. Thus $\rho' = \bigcup_{i=0}^{n} \rho_i' = \rho_1$, 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 subexpression $e'$ refines the original expression $e$, then the effect of the entire subexpression $E[e']$ refines the entire original expression $E[e]$. With this lemma, it suffices to show that the effect of the subexpression $e'$ refines the original subexpression $e$.

**Lemma C.12.** [Replacement with subeffect] If $\mu \vdash o$, $\Sigma \rightarrow \Sigma' = \langle E[e], (id, I) \rangle + \psi, \mu \rangle$, $\Sigma' = \langle E[e'], (id, I') \rangle + \psi', \mu' \rangle$, $\Sigma = \langle E[e], (id, I') \rangle + \psi, \mu \rangle$, $\Sigma' = \langle E[e'], (id, I') \rangle + \psi', \mu' \rangle$, then $\mu \equiv \mu'$.

**Proof:** Proof is by induction on the size of the evaluation context $E$. Size of $E$ refers to the number of recursive applications of the syntactic rules necessary to create $E$. In the base case, $E$ has size zero, $E = \epsilon$, and $\rho' = \rho \subseteq \rho$. For the induction step we divide the evaluation context into two parts such that $E[e_1] = E_1[E_2[e_2]]$. $E_1$ has size one. The induction hypothesis (IH) is that the lemma holds for all evaluation contexts, which is smaller than the one ($E_2$) considered in the induction step. We prove it case by case on the rule used to generate $E_2$. In each case we show that $\mu \vdash E_2[e'] : \rho'$ implies that $\mu' \vdash E_2[e'] : \rho'$, 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 $\bot \in \rho$. The maximum, any effect $\rho' \subseteq \bot$.

**Case** $- f = e_2$

The last step for $E_2[e_2]$ should be (E-SET):

$$
\mu' \vdash e : \rho_0
$$

By the definition of field lookup, typeOfF$E$ remains unchanged, i.e. typeOfF$E' = (c.t)$. Thus, by (E-SET), $\mu' \vdash E_2[e'] : \rho_1 \cup \{[\text{write } f]\}$

**Case** $v_0.f = $

The last step for $E_2[e']$ should be (E-SET):

$$
\mu' \vdash e : \rho_0
$$

So $\mu' \vdash E_2[e'] : \rho_1 \cup \{[\text{write } f]\}$

**Case** $- m(e_1, \ldots, e_n)$

The last step in the effect derivation for $E_2[e]$ should be (E-CALL-OPEN): $e = loc.f$

$$
\exists \text{s.t. } (\exists = \{\text{open } f \ m \ \rho_i'\} \subseteq \rho_i)
$$

By (GET), $e' = loc.c'`

$$
\epsilon' = \text{loc.c} \leftarrow \mu(c')(\{c.F'.E'\} ) = \rho_1', \forall i \in \{1, n\} \epsilon_i \neq \epsilon_i
$$

Because $\mu \vdash o$, by Definition C.9 and Definition C.7 we have $\rho_0' \subseteq \rho'$. If $\epsilon_1 \neq \epsilon_i$. The claim holds.
Case loc.m(v_1, ..., v_{p-1}, - , e_{p+1}, ..., e_n) Here \( p \in \{1..n\} \). The last step for \( E_2[e] \) must be (E-CALL-LOC):

\[
\begin{align*}
\mu(\text{loc}) &= [e, F] \\
E(m) &= p^\mu \\
\mu(e) &= \rho_0 \\
\forall i \in \{p+1..n\} : \mu[\varepsilon_i] = \rho_i^\mu \\
\mu \vdash E_2[e] \colon \rho_0 \cup \rho_i^\mu \cup \bigcup_{i=p+1}^{n} \rho_i
\end{align*}
\]

By (E-CALL-LOC), \( \mu \vdash E_2[e'] : \rho_0 \cup \rho_i^\mu \cup \bigcup_{i=p+1}^{n} \rho_i \).

Using the lemmas To prove Theorem 4 in each reduction case, let \( e' = E_2[e], e'' = E_1[e] \), \( \mu = \rho_0 \) and \( \mu[e'] = \rho_1 \). Given that (a) \( \mu \equiv \mu' \), by Lemma 12 and Lemma 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 2.2.2, the cases in the second category, access the store, in Section 2.2.3 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 \( \mu \equiv \mu' \) and (a) holds.

New Object: Here \( e = E_{\text{new}}(c)[], e' = E_{\text{yield loc}} \), where \( \text{loc} \notin \text{dom}(\mu) \), \( \mu' = \rho_0 \) and \( \mu[e'] = \rho_1 \). Given that \( \mu \equiv \mu' \), by Lemma 12 and Lemma 11, the last step for \( \text{loc} \) holds.

Yield: Here \( e = E_{\text{yield}}(e_1), e' = E_{\text{yield}}(e_1) \).

Fork Sequential: Here \( e = E_{\text{fork}}(e_0', e_1'), e' = E_{\text{yield}}(e_1') \), and \( e_0' = e_0; e_1' = e_1 \). Let \( \mu[e_0'] = \rho_0 \) and \( \mu[e_1'] = \rho_1 \). By (E-FORK), \( \mu \vdash E_{\text{fork}}(e_0', e_1') : \rho_0 \cup \rho_1 \).

Fork Parallel: Here \( e = E_{\text{fork}}(e_0', e_1'), e' = E_{\text{yield null}} \).

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 \( \mu \equiv \mu' \), thus (b) holds. We state a lemma for substituting the variables var for the actual values v, which indicates that the static effect \( \rho' \) after the substitution refines the one before the substitution \( \rho \). This lemma is useful for method calls and definitions, where parameters and local variables, respectively, will be substituted by values.

Lemma 13. [Substitution effect] If \( \mu \vdash e : \rho \), then there is some \( \rho' \), such that \( \mu \vdash [v_1/\text{var}_1, ..., v_n/\text{var}_n] e : \rho' \), for all values \( v_1 \) and free variables \( \text{var}_1, ..., \text{var}_n \), and \( \rho' \subseteq \rho \).

Proof: To simplify the notations, let \( [v/\text{var}] = [v_1/\text{var}_1, ..., v_n/\text{var}_n] \). 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, \( \rho' = \emptyset \). In the (E-VAR) case, \( \mu \vdash e : \emptyset \) and \( \mu \vdash \text{var} : \emptyset \). Thus, it holds.

The remaining cases cover the induction step. The induction hypothesis (IH) is that the claim of the lemma holds for all subderivations 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 \( \emptyset \subseteq \rho \). \( \emptyset \) is the maximum, any effect \( \rho' \subseteq \emptyset \).

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

\[
\begin{align*}
\mu(\text{loc}) &= [e, F] \\
E(m) &= p^\mu \\
\mu(e_i) &= \rho_i^\mu \\
\forall i \in \{1..n\} : \mu[\varepsilon_i] = \rho_i \\
\mu \vdash E_2[e] : \rho_0 \cup \rho_i^\mu \cup \bigcup_{i=1}^{n} \rho_i
\end{align*}
\]

Let \( e' = [\text{var}/e]_i \) for \( i \in \{1..n\} \), \( [\text{var}/e] = loc.f.m(e') \). We show that \( \mu \vdash [\text{var}/e] : \rho_0 \cup \rho_i^\mu \cup \bigcup_{i=1}^{n} \rho_i \), where \( \forall i \in \{0..n\} \rho_i^\mu \subseteq \rho_i \). Because \( loc.f \) has no free variable, \( \rho_0^\mu = \rho_0 \) and \( \rho_i^\mu \cup \bigcup_{i=1}^{n} \rho_i \) are unchanged. Also by IH, \( \forall i \in \{1..n\} : \mu[\varepsilon_i] = \rho_i, \rho_i^\mu \subseteq \rho_i \).

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

\[
\begin{align*}
\mu(\text{loc}) &= [e, F] \\
E(m) &= p^\mu \\
\forall i \in \{1..n\} : \mu[\varepsilon_i] = \rho_i
\end{align*}
\]

Now \( [\text{var}/e] = \rho_0 \cup \rho_i^\mu \cup \bigcup_{i=1}^{n} \rho_i \). By IH, \( \forall i \in \{0..n\} : \rho_i^\mu \subseteq \rho_i \). Clearly, \( \rho_0^\mu = \rho_0 \). Hence \( \forall i \in \{1..n\} : \mu[\varepsilon_i] = \rho_i, \rho_i^\mu \subseteq \rho_i \).

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

\[
\begin{align*}
\mu \vdash e_0 : \rho_0 \\
\mu[\text{var}] = \rho_1 \\
\mu \vdash \text{write } f : \rho_0 \cup \rho_1 \cup \{\text{write } f\}
\end{align*}
\]

Now \( [\text{var}/e] = \rho_0 \cup \rho_1 \cup \{\text{write } f\} \).

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

Method Call: Here \( e = E_{\text{loc.m}(\overline{e})}, (e_i, t_m, m(\overline{e})) = \text{findMethod}(u, m), e_1 = E_{\text{yield}}(e_1), e = [\text{loc}/\text{this}(e_0/e_1)] \).

Let \( \mu \vdash \text{loc.m}(\overline{e}) : \rho_0 \), i.e., \( E(m) = \rho_0 \). Let \( e_0 = \text{loc}/\text{this}(e_0/e_1) \), and \( e_1 = e_1 : \rho_1, \mu[\text{var}] = \rho_0 \cup \rho_1 \), and the claim holds.

Thus, for all possible derivations of \( \mu[\text{var}] = \rho' \) we see that \( \mu[\text{var}] = \rho' \) for some \( \rho' \subseteq \rho \).

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 shows that the effect \( \rho \) of subexpression \( e \) is a subset \( \subseteq \) of the effect \( \rho' \) of its entire expression \( [e] \).

Lemma 14. [Subexpression effect containment] If \( \mu \vdash e : \rho \) and \( \mu \vdash [e] : \rho' \), 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 $e = \text{E}[\text{loc}, f], e' = \text{E}[\text{yield}, v]$, where $\mu(\text{loc}) = [u, F, E], F(f) = v, \mu' = \mu$ and $\mu \not\equiv \mu'$. Because $\mu \vdash \text{loc} : f \rightarrow {\text{read}}$, and $\mu' \vdash \text{yield} = v : \emptyset$, (b) holds. Finally, $\eta = \langle \text{read, loc, f, id}, \gamma \rangle$, and $\eta \not\succ \text{read} \not\succ f \not\succ \rho$, by Lemma C.14.

**Field Set:** Here $e = \text{E}[\text{loc}, f = v], e' = \text{E}[\text{yield}, v], \mu' = \mu \cup \text{loc} \rightarrow \alpha$, and $\alpha = [u, F \oplus (f \rightarrow v), E]$, where $\mu(\text{loc}) = [u, F, E]$ and typeOf$(f) = (c, t)$ for some $\tau$. The field is not an open field, and by the function update, it does not update any effect, and $\mu \not\equiv \mu'$. To see $\mu \vdash \text{E}[v] : \rho' \subseteq \rho$, we have $\mu \vdash \text{loc} : f = v : \text{write} f$, and $\mu' \vdash v : \emptyset$, thus $\rho' \subseteq \rho$. Finally, $\eta = \langle \text{wt, loc, f, id}, \gamma \rangle$, and $\eta \not\succ \text{write} \not\succ f \not\succ \rho$, by Lemma C.14.

**Field Set Open:** Here $e = \text{E}[\text{loc}, f = v], e' = \text{E}[\text{yield}, v], \mu_0 = \mu \cup \text{loc} : f \rightarrow [\text{E}(\text{loc, f, id})], \text{f.}, \mu' = \text{update}(\mu_0, \text{loc}, f, v)$, Impossible, since $\langle \, \rangle \not\succ \rho$.

### 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 [14]. 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 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 $\Sigma \rightarrow^{0} \Sigma'$ if $\Sigma \rightarrow \Sigma'$, and $\Sigma \rightarrow^{0} \Sigma'$ if $\Sigma \rightarrow^{n} \Sigma'$ (where $l_0 \rightarrow l_0$). We write $\Sigma \rightarrow^{r} \Sigma'$ if $\exists n \geq 0. s.t. \Sigma \rightarrow^{n} \Sigma'$.

**Definition C.16.** [Multi step effect] The multi step effect function $\text{dynE}$ is: $\text{dynE}(\Sigma, \chi, \tau, n) = \text{dynE}(\Sigma', \chi', \tau', n - 1)$, if $\text{dyn}(\Sigma, \chi, \tau) = (\Sigma', \chi', \tau', 0)$.

Here, the function $\text{dyn}$ produces the dynamic effect and the relationship $\Omega$ for an one step transition from a configuration to another. The function $\text{dynE}$ accumulates the dynamic effects and the relationship for n transition steps.

**Definition C.17.** [Queue for parallel fork] We said that $\psi'$ is a queue for parallel fork tasks and we write $\Sigma \rightarrow \psi'$, if $\Sigma \rightarrow \Sigma'$, $\Sigma = \langle (\text{E}(\text{fork} (e, e'))), \psi, \mu, \Psi \rangle$, and $\Sigma' = \langle (e_0, \tau'), \psi' + \psi', \mu' \rangle$.

The queue $\psi'$ 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 ($\eta \trianglelefteq \eta'$) 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 $\varphi = \{\eta_1, \eta_2, \ldots\}$ do not conflict with another set $\varphi' = \{\eta'_1, \eta'_2, \ldots\}$ if $\forall i \in \{1, n\}, j \in \{1, p\}$ s.t. $\eta_i \trianglelefteq \eta'_j$.

**Definition C.18.** [Descendant] A task id' is a descendant of a task id, with $\chi$, written as $id' \leq_{T} id$, if id' $\in$ desc(id, $\chi$). Here, $\text{desc}(id, \chi) = \{id, \ldots, id_{n}\}$, and $\text{desc}(id, \chi) = \{id, \ldots, id_{n}\}$, and $\forall id_{i} \in \text{desc}(id, \chi) = \{id_{i}\}$.

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 $\Sigma_{i}$ that starts the program.

**Definition C.19.** [Initial configuration] The initial configuration, with a main expression e, is $\Sigma_{i} = (e, (0, 0), \emptyset)$.

**Lemma C.20.** [Stores preservation] If $\Sigma_{i} \rightarrow^{n} \Sigma$ and $\Sigma = (\psi, \mu)$, then $\psi \not\succ \mu$.

In Theorem C.25 we will prove that the tasks, created by the **fork** expression, do not interfere, and thus OpenEffectJ provides deterministic semantics [14]. We first prove a simpler theorem (Theorem C.23), for the 2 tasks $id_1$ and $id_2$, in $\psi$, generated by a **fork** expression ($\Sigma \rightarrow \psi$), when the queue is empty $\emptyset$, if the store is well-formed $\mu \not\succ \psi$, the effect judgments give them effect $p_1$ and $p_2$ respectively and they do not interfere $p_1 \not\succ p_2$, 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 $\rho$ of an expression e, is refined by the effect $\rho'$ 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', we show that the effect of id' refines id in Lemma C.22.

**Definition C.21.** [Local reduction] A reduction $\Sigma \rightarrow^{r} \Sigma'$, where $\Sigma = \langle \{e, (id, f) : \psi, \mu\} \rangle$ and $\langle \{e', (id', f') : \psi', \mu'\} \rangle \rightarrow \Sigma'$, is called a task local reduction, denoted as $\Sigma \rightarrow^{t} \Sigma'$, if $\exists \rho : \psi' \not\succ \mu \not\succ \rho$ then $\rho' \subseteq \rho$.

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 $\mu \not\succ \psi, \Sigma = \langle \{e, \tau : \psi, \mu\} \rangle$ and $\langle \{e', \tau' : \psi', \mu'\} \rangle \rightarrow \Sigma'$, then $\psi \not\succ \mu$ and $\mu \not\succ \rho'$ then $\rho'' \subseteq \rho$.

**Proof:** Immediately follows from the definition of (T-**fork** and (FORK-**parallel** rules).

**Theorem C.23.** [Noninterference] Let $\Sigma \rightarrow \psi, \mu \not\succ \psi, \Sigma \rightarrow \Sigma'$ and $\Sigma = \langle \{\text{E}(\text{fork} (e_1, e_2)), \psi, \mu\} \rangle$. For the two tasks $\langle e_1, (id_1, f_1) \rangle < \psi$ and $\langle e_2, (id_2, f_2) \rangle < \psi$, if $\mu \not\succ \mu_1 : \rho_1, \mu \not\succ \mu_2 : \rho_2$, then $\forall n < N$, dynESet(id, $\chi$) = dynESet(id, $\chi$) where $\chi = \chi \cup \{e_1, e_2\}$. It suffices to show that $\forall n < N$, 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 $\eta$ produced by a reduction, refines the static effect $\rho$ of the original expression of that reduction, i.e., $\eta \not\succ \rho$; and (2) the static effect $\rho$ of a subsequent expression refines the static effect $\rho'$ of the original expression, i.e., $\rho \not\succ \rho'$. 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., $\eta \not\succ \rho$.

**Theorem C.24.** [directly follows from Lemma C.24]
We will prove it by induction on \( n \). The base case \( n = 0 \). Note that \( \Sigma' = (\Sigma' \land \text{yieldnull}, \tau') + \psi + \mu \) 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 \( id_1 \) in \( \psi \), by the active function, i.e., \( \Sigma_0 = (\langle e_1, (id_1, I_1) \rangle + \psi, \mu) \) and \( (e_1, (id_1, I_1)) \in \psi \). As stated, for id1 and id2, \( \rho_1 ^ \circ \Sigma_0 = (id_1, \rho_1) \), thus (a) holds. We have \( \mu \vdash \circ \), by Theorem C.10 (b) holds. (c) holds, because this is the first time the task id makes progress, i.e., \( \exists \Sigma_1. s.t. (id, \Sigma_1) \to (id, \Sigma) \).

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 \( \Sigma_j = \langle (\Sigma'[j], (id_1', I_1')) + \psi, \mu \rangle \), \( \Sigma_{j+1} = \langle (\Sigma'[j+1], (id_1'+1, I_1'+1)) + \psi_1 + \mu_1 \rangle \), \( j \in \mathbb{N} \), \( j \geq 1 \) and \( j \neq 0 \).

1. If \( id_1' = id_1 \), by Theorem C.10 (a) holds;
2. \( id_1' \neq id_1 \), then \( \psi_1 \subseteq \psi \) for some \( \psi_1 \). By the (yield) rule, \( \mu_1 = \mu \). By IH, \( \exists \Sigma_{j+1} : (id_1, \Sigma) \to (id_1, \Sigma') \).

For (b), if \( id_1' = id_1 \), then by Theorem C.10 it holds; otherwise, \( \psi_1 \subseteq \psi \), for some \( \psi_1 \). By IH, \( \exists \Sigma_{j+2} : (id_1, \Sigma) \to (id_1, \Sigma') \).

By Lemma C.11 (c) holds.

For all steps \( n < N \), we see that \( \eta \vdash ^\circ \rho \) and for each local reduction \( \Sigma \to \Sigma' \), \( \rho \vdash ^\circ \rho' \). By Lemma C.12 the effect of a child task refines the corresponding fork expression of its parent task. Thus, \( \text{dynEset}(\Sigma, \psi, \chi, \pi) \subseteq \text{dynEset}(\Sigma, \psi, \chi, \pi) \).

THEOREM C.25. [Deterministic semantic] Let \( \mu \vdash \circ, \Sigma \to \Sigma' \), \( \psi \subseteq \chi \) and \( (\Sigma, \psi, \chi, \mu) \). For the two tasks \( (e_1, (id_1, I_1)) \in \psi \) and \( (e_2, (id_2, I_2)) \in \chi \), \( \psi' \subseteq \phi \), \( \psi' \subseteq \psi_2 \), \( \psi_2 \subseteq \phi \), \( \phi \subseteq \phi_2 \) then \( \forall \Sigma. s.t. \text{dynEset}(\Sigma, \psi, \chi, \mu) \).

Proof: Note that the difference between this theorem and Theorem C.23 that is \( \mu \) may or may not be \( \psi \). Observe that there exists \( \Sigma' = (\Sigma, \psi_1, \mu_1) \) and \( k > 0 \) such that \( \Sigma' \to^k \Sigma \).

We will show the type 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. \( \Pi \subseteq \Pi' \)). Also, we need to ensure that during the evaluation, all the tasks have proper types, i.e., all the tasks in the queue \( \psi \) are well-typed (Definition C.28 and Definition C.29). Finally, we state the standard lemmas \( \text{[Well-typed queue]} \) (Lemma C.30 Lemma C.31 Lemma C.32 and Lemma C.33).

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

1. \( (\forall \text{loc } \mu(\text{loc}) = [t.F.E.] \)
2. \( (\forall \text{dom}(F) = \text{dom}(\text{fields}(t)) \text{ and } \forall \text{range}(F) \subseteq \text{loc}(\text{mu}) \text{ and } \forall \text{null}\) )
3. \( (\forall \text{find}(c, m) = t \text{ and } \text{typeOf}(f) = (d, \text{open } c_0) \text{ and } \text{find}(c, m) = t \) )

Figure 11. Type and effect rules for loc.

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

This definition says that an environment \( \Pi' \) enlarges another environment \( \Pi \), if the domain of \( \Pi' \) is a subset of \( \Pi \) and, they give the same type for the common location. This definition will be used to show that during the evaluation of any OpenEffectU program, we can use an even increasing type environment to type check the expressions.

DEFINITION C.28. [Well-typed queue] A queue \( \psi \) is well-typed in \( \Pi \), written \( \Pi \vdash \psi \), if \( (\forall \epsilon, \psi \in \psi, \Pi \vdash \epsilon : (t, \rho) \text{ for some type } t \text{ and effect } \rho) \).

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 \( \Sigma = (\psi, \mu) \) is well-typed in \( \Pi \), written \( \Pi \vdash \Sigma \), if \( \Pi \vdash \psi \) and \( \mu \approx \Pi \).
LEMMA C.30. [Substitution] If \( \overline{\Pi}, \overline{\alpha} : t \vdash e : (t, \rho) \) and \( \forall i \in \{1..n\}, \Pi \vdash v_i : (s_i, \theta) \) where \( s_i < t_i \) then \( \overline{\Pi}, \overline{\alpha} : (s, \rho') \) for some \( s < t \) and \( s' \).

**Proof:** To simplify the notations, we set \( \Pi' = \overline{\Pi}, \overline{\alpha} : t \). We prove it by structural induction on the derivation of \( \Pi' : e : (t, \rho) \) 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 \( \var \in \{ \var_1, \ldots, \var_n \} \), then \( \Pi'(\var) = \Pi(\var) = t \) and the claim holds. Otherwise, suppose \( \var = \var_k \). Then \( \overline{\Pi'}(\var) = v_k \) and, by the assumptions of the lemma, \( \Pi' \vdash \overline{\Pi'}(\var) : (s_k, \theta) \) 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 sub-derivations 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 \( e = \textbf{this}.f.m(\vec{\sigma}) \). The last type derivation step has the following form:

\[
\begin{align*}
\overline{\Pi'} &\vdash e_0' : (t, \rho_0) \\
\text{typeOf}(f) &\equiv (d, @open) \quad \Pi' \vdash e_0' : (c_0, \rho_0) \\
&\text{findMeth}(c_0, m) = (c_1, m, u_1, v_1, v_0) = (e_0', \rho_2) \\
&\forall i \in \{1..n\} : \Pi' \vdash e_i' : (u_i, \rho_1) \land u_i < u_i \\
&\overline{\Pi'} \vdash e_0'm(e_1', \ldots, e_n') : (t, \{ \text{open } f \text{ m } \theta \} \cup \bigcup \rho_i)
\end{align*}
\]

Let \( e_0' = \overline{\Pi'}(\var) : (t, \rho_0) \), \( \Pi' \vdash e_0' : (c_0, \rho_0) \) and \( \text{findMeth}(c_0, m) = (c_1, m, u_1, v_1, v_0) = (e_0', \rho_2) \) for some \( i \in \{1..n\} \). By the definition of \( \text{findMeth} \) and \( \overline{\Pi'}(\var) = v_k \), \( v_k < u_k \). Finally, \( \forall i \in \{1..n\} : u_i < u_i \) by transitivity, the claim holds.

**T-CALL** Here \( e = e_0'm(\vec{\sigma}) \). The last type derivation step has the following form:

\[
\begin{align*}
\overline{\Pi'} &\vdash e_0' : m(c_1', \ldots, c_n') : (t, \{ \text{open } f \text{ m } \theta \} \cup \bigcup \rho_i) \\
\text{typeOf}(f) &\equiv (d, @open) \quad \Pi' \vdash e_0' : (c_0, \rho_0) \\
&\text{findMeth}(c_0, m) = (c_1, m, u_1, v_1, v_0) = (e_0', \rho_2) \\
&\forall i \in \{1..n\} : \Pi' \vdash e_i' : (u_i, \rho_1) \land u_i < u_i \\
&\overline{\Pi'} \vdash e_0'm(c_1', \ldots, c_n') : (t, \rho_0)
\end{align*}
\]

Let \( e_0' = \overline{\Pi'}(\var) : (t, \rho_0) \), \( \Pi' \vdash e_0' : (c_0, \rho_0) \) and \( \text{findMeth}(c_0, m) = (c_1, m, u_1, v_1, v_0) = (e_0', \rho_2) \) for some \( i \in \{1..n\} \). By the definition of \( \text{findMeth} \) and \( \overline{\Pi'}(\var) = v_k \), \( v_k < u_k \). Finally, \( \forall i \in \{1..n\} : u_i < u_i \) by transitivity, the claim holds.

**T-CALL-OPEN-LOC** Here \( e = \text{loc}.f.m(\vec{\sigma}) \). The last type derivation step has the following form:

\[
\begin{align*}
\text{typeOf}(f) &\equiv (d, @open) \quad \Pi \vdash f : (c_0, \rho_0) \\
\text{findMeth}(c_0, m) &\equiv (c_1, \{ \text{open } f \text{ m } \theta \} \cup \bigcup \rho_i) \\
&\forall i \in \{1..n\} : \Pi \vdash e_i' : (u_i, \rho_1) \land u_i < u_i \\
&\Pi \vdash f : (t, \rho_0)
\end{align*}
\]

**T-SET** Here \( e = e_1', e_2' : f \). The last type derivation step is:

\[
\begin{align*}
\Pi' &\vdash e_1' : (c_1, \rho_1) \\
\Pi' &\vdash e_2' : (t, \rho_2) \\
&\Pi' \vdash e : (t, \rho_1 \cup \{ \text{write } f \})
\end{align*}
\]

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

\[
\begin{align*}
\Pi' &\vdash e' : (c_1, \rho_1) \\
&\Pi' \vdash e' : (t, \rho_2) \\
&\Pi' \vdash e : (t, \rho_1 \cup \{ \text{read } f \})
\end{align*}
\]

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

\[
\begin{align*}
\Pi'(\text{this}) &\vdash c \\
\Pi' &\vdash e' : (d, @open) \\
&\Pi' \vdash e' : (t, \rho_1 \cup \{ \text{read } f \})
\end{align*}
\]

**T-GET-OPEN-LOC** Here \( e = \text{loc}.f \). The last step is:

\[
\begin{align*}
\Pi'(\text{loc}) &\vdash c \\
\Pi' &\vdash e' : (t, \rho_2) \\
&\Pi' \vdash f : (t, \rho_1 \cup \{ \text{write } f \})
\end{align*}
\]

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

\[
\begin{align*}
\Pi'(\text{this}) &\vdash c \\
\Pi' &\vdash e' : (d, @open) \\
&\Pi' \vdash e' : (t, \rho_2)
\end{align*}
\]

**T-SET-OPEN-LOC** Here \( e = \text{loc}.f = e' \). The last step is:

\[
\begin{align*}
\Pi'(\text{loc}) &\vdash c \\
\Pi' &\vdash e' : (t, \rho_1) \\
&\Pi' \vdash f : (t, \rho_1 \cup \{ \text{write } f \})
\end{align*}
\]

**T-GET-OPEN-LOC** Here \( e = \text{loc}.f = e' \). The last step is:

\[
\begin{align*}
\Pi'(\text{loc}) &\vdash c \\
\Pi' &\vdash e' : (t, \rho_1) \\
&\Pi' \vdash f : (t, \rho_1 \cup \{ \text{write } f \})
\end{align*}
\]

**LEMMA C.31. [Environment extension]** If \( \Pi \vdash e : (t, \rho) \) and \( a \notin \text{dom}(\Pi) \), then \( \Pi, a' : (t, \rho') \) for some \( t' \leq t \).

**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 \( \Pi : e : (t, \rho) \). 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, $e \equiv var$ and $\Pi(var) = t$. But $a \not\in dom(\Pi)$, 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 $\Sigma = ((\Sigma[e], (id, I)) + \psi, \mu)$, $\Sigma' = ((\Sigma[e'], (id, I')) + \psi', \mu')$, $\Sigma \Rightarrow \Sigma'$, $\Pi \vdash E[e] : (t, \rho)$, $\Pi \vdash e : (t', \rho')$ and $\Pi \vdash e' : (t', \rho'_0)$, then $\Pi \vdash E[e'] : (t, \rho_0)$, for some $\rho_0$.

**Proof:** Proof is by induction on the size of the evaluation context $E$. Size of the $E$ refers to the number of recursive applications of the syntactic rules necessary to create $E$. In the base case, $E$ has size zero, $E = -$, and $t' = u' < u = t$. For the induction step we divide the evaluation context into two parts such that $E[e_1] = E_1[2\Sigma_2[e_2]]$, and $E_2$ has size one. The induction hypothesis (IH) is that the lemma holds for all evaluation contexts, which is smaller than the one ($E_1$) considered in the induction step. We prove it case by case on the rule used to generate $E_2$. In each case we show that $\Pi \vdash E_2[e] : (s, \rho)$ implies that $\Pi \vdash E_2[e'] : (s, \rho')$, for some $\rho'$, and thus the claim holds by IH.

The cases for (loc, $f = -$), (loc, $e'$), and (var, $e = -; e_2$) follow directly from the induction hypothesis.

**Case $-m(\overline{e})$** The last step for $E_2[e]$ could be

<table>
<thead>
<tr>
<th>Case</th>
<th>Type of $E_2[e]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Pi \vdash e : (u, \rho_0)$</td>
<td>$\Pi \vdash E_2[e] : (t, \rho_0)$</td>
</tr>
</tbody>
</table>

Here $\pi_1$ is the first projection function $\pi_1(t, m \rightarrow n) = m$, and $\pi_2$ is the second projection function $\pi_2(t, m \rightarrow n) = n$, by the definitions of typeOfF and findMeth, where $c \ll c_1$, so (T-CALL) gives $\Pi \vdash E_2[e] : (t', \rho')$; or

**Case $\langle\langle t \rangle\rangle$**

<table>
<thead>
<tr>
<th>Case</th>
<th>Type of $E_2[e]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Pi \vdash e : (t, \rho'_0)$</td>
<td>$\Pi \vdash E_2[e] : (t, \rho'_0)$</td>
</tr>
</tbody>
</table>

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 $t$.

**Case $\langle\langle t \rangle\rangle$**

<table>
<thead>
<tr>
<th>Case</th>
<th>Type of $E_2[e]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Pi \vdash e : (t, \rho'_0)$</td>
<td>$\Pi \vdash E_2[e] : (t, \rho'_0)$</td>
</tr>
</tbody>
</table>

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 findMeth does not change, therefore, by (T-CALL), the type of the expression is $t$.

**Case $\langle\langle t \rangle\rangle$**

<table>
<thead>
<tr>
<th>Case</th>
<th>Type of $E_2[e]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Pi \vdash e : (t, \rho'_0)$</td>
<td>$\Pi \vdash E_2[e] : (t, \rho'_0)$</td>
</tr>
</tbody>
</table>

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 findMeth does not change, therefore, by (T-CALL), the type of the expression is $t$.

**Case $\langle\langle t \rangle\rangle$**

<table>
<thead>
<tr>
<th>Case</th>
<th>Type of $E_2[e]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Pi \vdash e : (t, \rho'_0)$</td>
<td>$\Pi \vdash E_2[e] : (t, \rho'_0)$</td>
</tr>
</tbody>
</table>

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 findMeth does not change, therefore, by (T-CALL), the type of the expression is $t$.

**Case $\langle\langle t \rangle\rangle$**

<table>
<thead>
<tr>
<th>Case</th>
<th>Type of $E_2[e]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Pi \vdash e : (t, \rho'_0)$</td>
<td>$\Pi \vdash E_2[e] : (t, \rho'_0)$</td>
</tr>
</tbody>
</table>

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 findMeth does not change, therefore, by (T-CALL), the type of the expression is $t$.

**Case $\langle\langle t \rangle\rangle$**

<table>
<thead>
<tr>
<th>Case</th>
<th>Type of $E_2[e]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Pi \vdash e : (t, \rho'_0)$</td>
<td>$\Pi \vdash E_2[e] : (t, \rho'_0)$</td>
</tr>
</tbody>
</table>

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 findMeth does not change, therefore, by (T-CALL), the type of the expression is $t$.

<table>
<thead>
<tr>
<th>Case</th>
<th>Type of $E_2[e]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Pi \vdash e : (t, \rho'_0)$</td>
<td>$\Pi \vdash E_2[e] : (t, \rho'_0)$</td>
</tr>
</tbody>
</table>

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 findMeth does not change, therefore, by (T-CALL), the type of the expression is $t$.

**Case $\langle\langle t \rangle\rangle$**

<table>
<thead>
<tr>
<th>Case</th>
<th>Type of $E_2[e]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Pi \vdash e : (t, \rho'_0)$</td>
<td>$\Pi \vdash E_2[e] : (t, \rho'_0)$</td>
</tr>
</tbody>
</table>

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 findMeth does not change, therefore, by (T-CALL), the type of the expression is $t$.

**Case $\langle\langle t \rangle\rangle$**

<table>
<thead>
<tr>
<th>Case</th>
<th>Type of $E_2[e]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Pi \vdash e : (t, \rho'_0)$</td>
<td>$\Pi \vdash E_2[e] : (t, \rho'_0)$</td>
</tr>
</tbody>
</table>

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 findMeth does not change, therefore, by (T-CALL), the type of the expression is $t$.

**Case $\langle\langle t \rangle\rangle$**

<table>
<thead>
<tr>
<th>Case</th>
<th>Type of $E_2[e]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Pi \vdash e : (t, \rho'_0)$</td>
<td>$\Pi \vdash E_2[e] : (t, \rho'_0)$</td>
</tr>
</tbody>
</table>

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 findMeth does not change, therefore, by (T-CALL), the type of the expression is $t$.

**Case $\langle\langle t \rangle\rangle$**

<table>
<thead>
<tr>
<th>Case</th>
<th>Type of $E_2[e]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Pi \vdash e : (t, \rho'_0)$</td>
<td>$\Pi \vdash E_2[e] : (t, \rho'_0)$</td>
</tr>
</tbody>
</table>

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 findMeth does not change, therefore, by (T-CALL), the type of the expression is $t$.  

\[ \text{Formulation of the proof is similar Flatt's work} \]  

\[ \text{2012/7/21} \]
Case \( \neg \text{m}(\pi) \) The last step for \( \mathcal{E}_2[e] \) could be
\[
\Pi \vdash e : (t', \rho') \quad (c_1, t, m (\text{Var}) (e_{n+1}), (e_1), \rho_1) = \text{findMeth}(t', m) \\
\quad \forall i \in [1..n] \quad : \Pi \vdash e_i : (t'_i', \rho''_i) \wedge t'_i < t_i \\
\Pi \vdash \mathcal{E}_2[e] : (t, \{ \})
\]
We have \( \text{findMeth}(t', m) = (c_2, t, m (\text{Var}) (e'_{n+1}), (e_1), \rho_1) \), by the definitions of \text{override} and \text{findMeth}, \( c_2 < c_1 \), so (T-CALL) gives
\( \Pi \vdash \mathcal{E}_2[e] : (t, \{ \}) \).

Case \( \neg \cdot f \) The last step for \( \mathcal{E}_2[e] \) could be
\[
\Pi \vdash e : (c, \rho_1) \quad \text{typeOff}(f) = (t, \rho_1) \quad e < c \\
\Pi \vdash \mathcal{E}_2[e] : (t, \{ \}) \quad \{ \text{read } f \}
\]
The result of \text{typeOff} does not change. Thus, by (T-GET), \( \Pi \vdash \mathcal{E}_2[e'] : (t', \rho_0) \).

Theorem C.34. [Type preservation] If \( \Pi \vdash \Sigma \), where \( \Sigma = \langle \psi, \mu \rangle \), \( \psi \rightarrow (t', \rho') \), and \( t < c \). The result of \text{typeOff} does not change. Thus, by (T-GET-LOC-P), \( \Pi \vdash \mathcal{E}_2[e'] : (t', \rho') \).

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 \( \psi \) does not change. Thus by Lemma C.31 (Environment extension) and Definition C.26 if \( \Pi \vdash \psi \), then \( \Pi \vdash \psi' \).

New Object Here \( e = [\text{new } c] \) and \( e' = [\text{yield } v] \) loc where \( \text{loc} \in \text{dom}(\mu) \) and \( \mu' = \{ \text{loc} \rightarrow c \} \) \& \( \text{fields}(c) \) \& \( \{ \text{loc} \rightarrow \rho \} \in \text{meth}(E(c)) \in \mu \). Then \( \Pi' = \Pi \cup \text{loc} : c \) \& \( \mu' \). We now show that \( \Pi' \equiv \mu' \). Because \( \text{loc} \not\equiv \text{dom}(\mu) \), \( \text{loc} \not\equiv \text{loc}(\mu) \) by Definition C.26. Thus part 1 of the definition for \( \Pi' \equiv \mu' \) holds for all \( \text{loc} \not\equiv \text{loc} \). Now \( \mu' (\text{loc}) = \{ c \} \). By dom \( (\text{fields}(c)) \), \( \text{ref}(F) = \{ \} \) \& \( \text{null} \rightarrow \text{null} \) \& \( \text{null} \rightarrow \text{null} \) \& \( \text{null} \rightarrow \text{null} \). Also by \( \Pi' \equiv \mu' \), we have \( \Pi' \equiv \text{new } c : [t', \rho'] \). Now \( \Pi' \equiv \text{new } c : [t', \rho'] \). By Lemma C.32 (Environment extension) and dom \( (\text{loc}) \), we have \( \Pi' \equiv \text{yield } loc : (t', \rho') \). Now \( \Pi' \equiv \text{new } c : [t', \rho'] \).

Field Get In this case \( e = E[\text{loc}, f] \), \( e' = [\text{yield } v] \) loc \( \mu' = \{ \text{loc} \rightarrow c \} \) \& \( \text{fields}(c) \) \& \( \{ \text{loc} \rightarrow \rho \} \). Then \( \mu' = \{ \text{loc} \rightarrow c \} \). Clearly \( \Pi' \equiv \mu' \), and \( \Pi' \equiv \mu' \). We now show that \( \Pi' \equiv \text{yield } v : (t', \rho') \). Now \( \Pi' \equiv \text{loc} : f : [t, \{ \text{read } f \}] \). The last step in this derivation must be (T-GET) or (T-GET-OPEN-LOC). By the first hypothesis of (T-GET), \( \text{GET-OPEN-LOC} \) and by \( \Pi' \equiv \mu' \), we have \( \Pi' (\text{loc}) = u \). By the second hypothesis of (T-GET), \( \text{typeOff}(f) = (t, \rho) \). By the second hypothesis of (T-GET-LOC-P), \( \text{typeOff}(f) = (t, \rho) \). Also by \( \Pi' \equiv \mu' \), if (a) \( \mu' (v) = u', \rho' \), then \( \mu' (v) = u' \) \& \( \rho' \); otherwise (b) \( \mu' (v) = \text{null} \). In both cases, the type of \( v \) is subtype of \( s \), by Lemma C.33 (Replacement with subtyping), \( \Pi' \equiv [\text{yield } v] : (t', \rho') \).

Field Set Here \( e = E[\text{loc}, f] \), \( e' = [\text{yield } v] \), \( \mu_0 = \mu \); \( \text{loc} \rightarrow \{ t : f \rightarrow v \} \). Then \( \mu' = \mu' \). Let \( \Pi' = \Pi \). Now we show that \( \Pi' \equiv \mu' \). Observe that the update function changes the effect mapping \( E \) in each of the object record, but not the fields \( F \). Thus, by Definition C.26 (Here \( \mu' (\text{loc}) = \{ c \} \rightarrow (f \rightarrow v) \)). For part (a) \( \Pi' \equiv \mu_0 \equiv \mu \). Let \( \Pi' = \Pi \equiv \mu' \). For part (b) \( \Pi' \equiv \mu_0 \equiv \mu \). Let \( \Pi' = \Pi \equiv \mu' \). Observe that the update function changes the effect mapping \( E \) in each of the object record, but not the fields \( F \). Thus, by Definition C.26 (Here \( \mu' (\text{loc}) = \{ c \} \rightarrow (f \rightarrow v) \)). For part (a) \( \Pi' \equiv \mu_0 \equiv \mu \). Let \( \Pi' = \Pi \equiv \mu' \). For part (b) \( \Pi' \equiv \mu_0 \equiv \mu \). Let \( \Pi' = \Pi \equiv \mu' \).
well-typed. For part 1(c), \(\text{rng}(F \oplus (f \mapsto v)) \subseteq \text{rng}(F) \cup \{v\}\). Now since \(\text{loc}.f = v\) is well-typed, then \(v \in \text{dom}(\Pi)\) or \(v = \text{null}\). In the former case, by \(\Pi \approx \mu\), then \(v \in \text{dom}(\mu)\). \(v \in \text{dom}(\mu)\) implies \(v \in \text{dom}(\mu')\). In either case \(\text{rng}(F) \cup \{v\} \subseteq \text{dom}(\mu')\) or \(\{\text{null}\}\). Part 1(d) holds for all \(f \in \text{dom}(F)\), \(f' = f\). Part 1(d) holds vacuously for \(f\) if \(v = \text{null}\). Otherwise, \((F \oplus (f \mapsto v))(f) = v\), and by (T-SET) or (T-SET-OPEN-LOC) and (T-LOC), \(\Pi(v) < \psi\), where fields\((u) = (e, x')\) and \(u < c\). Parts 2 holds since \(\text{dom}(\mu') = \text{dom}(\mu)\).

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

**Method Call**

Here \(e = \text{yield}\ e_1\), \(\mu(\text{loc}) = [u . F . \emptyset]\), \((\text{findMethod}(u, m) = (u', t, \text{map}\ \var)\{e_2\}, \rho_0)\), \(\mu' = \mu\) and \(e_1 = [\text{this}.\var\map v]\). Let \(\Pi' = \Pi\). Clearly \(\Pi' \approx \mu'\), and \(\Pi \ll \Pi'\).

We now show that \(\Pi \vdash e' : (t', \rho')\) for some \(t' < t\) and some \(\rho'\).

\(\Pi \vdash e : (t, \rho)\) implies that \(\text{loc}.m(\var)\) and all its subterms are well-typed in \(\Pi\). By part 1(a) of \(\Pi \approx \mu\), \(\Pi \vdash \text{loc}.(u, \emptyset)\).

By the definition of \(\text{findMethod}\), \(u : \var < u'\). Let \(\Pi \vdash v : (u, \emptyset)\) and \(\Pi \vdash \text{loc}.(\var)\{e_2\}, \rho_0\). This last judgment must be \(\text{(T-CALL)}\), with \(\{u', t', \text{map}\ \var\}\{e_2\}, \rho_0\) = \(\text{findMethod}(u, m)\), \(\forall v \in \{1, \ldots, n\} : u_v < t_v\). By the definition of the function \(\text{findMethod}\), rules \(\text{(T-METHOD)}\) and \(\text{override}\), \(\{\text{var}\map t, \text{this} : u'\} \vdash e_2 : (u_m, \rho_1)\), and \(u_m < t_m\).

By Lemma C.31 (Environment extension) (and appropriate alpha conversion of free variables in \(e_2\)), \(\Pi \vdash \var\map t, \text{this} : u'\map e_2 : (u_m, \rho_1)\).

By Lemma C.30 (Substitution), \(\Pi' \vdash \text{loc}\var\map t, \text{this} : e_2 : (u', \rho_1)\), 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 \(e = \text{yield}\ e_1\), \(e' = \text{yield}\ e_1')\), where \(e_1' = [v\map\var]e_1\) and \(\mu' = \mu\). Let \(\Pi' = \Pi\). Obviously \(\Pi' \approx \mu'\), and \(\Pi \ll \Pi'\). We show \(\Pi \vdash \text{yield}\ e_1' : (t', \rho')\) for some \(t' < t\). \(\Pi \vdash e : (t, \rho)\) implies that \(t\ \var = v\). \(e_1\) and all its subterms are well-typed in \(\Pi\), let \(\Pi \vdash t\ \var = v : (s, \rho_0)\). By (T-DEFINE), \(\Pi \vdash t\ \var = e_1' : (s, \rho_0)\).

By Lemma C.30 (Substitution), \(\Pi' \vdash \var\map e_1' : (s', \rho_1)\), for some \(s' < s\). Finally, Lemma C.33 (Replacement with subtyping) gives \(\Pi' \vdash e' : (t', \rho')\) for some \(t' < t\).

**Fork-Sequential**

Here \(e = \text{fork}(e_2; e_3)\), \(e' = \text{yield}\ e_0\), \(\mu' = \mu\) and \(e_0 = e_2; e_3\). \(\Pi' = \Pi\). Clearly \(\Pi' \approx \mu'\), and \(\Pi \ll \Pi'\).

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

**Fork-Parallel**

Here \(e = \text{fork}(e_0; e_1)\), \(e' = \text{yield}\ \text{null}\) and \(\mu' = \mu\). Let \(\Pi' = \Pi\). Clearly \(\Pi' \approx \mu'\) and \(\Pi \ll \Pi'\).

We now show that \(\Pi \vdash e' : (t', \rho')\) for some \(t' < t\) and some \(\rho'\). By (T-FORK), \(\Pi \vdash \text{fork}(e_0; e_1) : (\text{void}, \rho_0)\). We know \(\Pi \vdash \text{null} : (\text{void}, \emptyset)\), \(\Pi \vdash \text{yield}\ \text{null} : (u, \emptyset)\), for any valid type \(u\) and some \(\rho_1\). Since the type of \(\text{null}\) 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, \(e = \text{yield}\ e_1\), \(e' = \text{yield}\ e_1\) 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 \(\Pi' \approx \mu'\), \(\Pi \ll \Pi'\), and \(\Pi \vdash \psi\).

Let \(\Pi \vdash \text{yield}\ e_1 : (t_0, \rho_0)\), then \(\Pi \vdash e_1 : (t_0, \rho_0)\), thus by Lemma C.32 (Replacement), \(\Pi \vdash e' : (t, \rho)\).