7-10-2012

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](http://lib.dr.iastate.edu/cs_techreports)

Part of the [Programming Languages and Compilers Commons](http://lib.dr.iastate.edu/cs_techreports), and the [Software Engineering Commons](http://lib.dr.iastate.edu/cs_techreports)

Recommended Citation

[http://lib.dr.iastate.edu/cs_techreports/212](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 at Iowa State University. It has been accepted for inclusion in Computer Science Technical Reports by an authorized administrator of Digital Repository at Iowa State University. For more information, please contact digirep@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
Abstract

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

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

General Terms Languages, Theory, Verification

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

1. Introduction

Both static [2, 5, 6, 8, 17, 32, 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] of concurrent tasks to determine whether these tasks interfere with each other and thus could lead to unexpected program behavior.

1.1 The Problems and Their Importance

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

To illustrate consider the class ArrayList in Figure 1 inspired from its namesake in the Java Development Kit (JDK). This class provides an operation applyall that iterates over each element in the list and calls method run with argument c of type Command as the receiver object. Two separate applications, PROGRAM and PROGRAM' (among others) use this ArrayList class. Each application provides a 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 a loop iteration is now reading and writing a different slot of an array.

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 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 static part of our analysis would then trust the programmer by taking the effect of a method call on this reference as an open effect, i.e., an effect that could be extended at runtime but is blank statically. So the effect of method call c.run(…) on line 6 would be taken as an open effect, because the dynamic type of c is unknown. Thus, the effect of an iteration of the for loop on line 5 would be reading the ith element of the array elements as a blank 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.

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

To illustrate, imagine that the programmer optimistically marked the argument c’s type in the class ArrayList as open as on line 12 in Figure 2. The static part of our analysis would then trust the programmer by taking the effect of a method call on this reference as an open effect, i.e., an effect that could be extended at runtime but is blank statically. So the effect of method call c.run(…) on line 6 would be taken as an open effect, because the dynamic type of c is unknown. Thus, the effect of an iteration of the for loop on line 5 would be reading the ith element of the array elements and an open effect. An open effect is treated as a blank effect statically, so an iteration of this for loop would be treated (trustingly) as parallel since it is independent of other iterations.

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

Figure 3. An example use of the ArrayList library.

On a different day, the developer of PROGRAM imports the class ArrayList. At runtime, PROGRAM creates an instance al of ArrayList, and passes an instance p of class Prefix on line 13 in Figure 3 as argument c. This assignment to the argument c concretizes the effect of an iteration of the for loop, because the effect of this for loop contains an open effect (method call effect on an unknown reference c). Note that the receiver object c is now an alias of the instance p of class Prefix. So, the extended effect (the original effect union with the concrete effect of the method run of the class Prefix) of the loop iteration is now reading and writing to the field sum of instance Prefix and writing to different slot of an array. As a result, iterations of the for loop now has a loop carried dependence (on the field sum, line 18 in Figure 1). Thus, the for loop on line 13 is run sequentially when method applyall is called on line 12 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 show that most concurrency is exposed via libraries [6] [26].

- 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 3 and examples in Section 2.

- a dynamic semantics with open effect concretization and open effect-based concurrency decisions in Section 5.

- a prototype compiler based on the OpenJDK Compiler in Section 6 that shows only about 0.13-7.65% overhead.

- a rigorous proof that OpenEffectJ ensures determinism – thus, users are guaranteed to avoid many complex concurrency issues in Section 5.4 The soundness proof is challenging compared to the static analyses, because the effect of a task could change at runtime, due to the open effects; and

- a comparative analysis with related ideas in Section 7.

2. Optimistic Effect Analysis for Reusable Code

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

2.1 Using Open Effects in Sorting Algorithms

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

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

```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+1; j>0 && c.compare(src[j-1], src[j]) > 0; j--)
                    dest.swap(dest[j-1], j);
            }
            return dest;
        }
        int mid = (low + high) / 2;
        Object[] dl = this.mergeSort(c, src, low, mid);
        Object[] dh = this.mergeSort(c, src, mid, high);
        /* the details of merge omitted. */
    }
    class Comparator extends Object {
        int compare(Object o1, Object o2) {
    }
}
```

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

1The original code in RuleBasedCollator is thread safe though.
OpenEffectJ by declaring the parameter $c$ in method `mergeSort` as `open` on line 6. 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[] d2 = a.mergeSort(cDirty,src,0,src.length);
// Following will do a sequential sort.
Object[] d1 = a.mergeSort(cPure,src,0,src.length);
```

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

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

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

2.2 Using Open Effects in Search Algorithms

Next, we study a representative search algorithm, depth-first search (DFS). DFS is typically formulated as a graph traversal as in the listing below 24. For extensibility and reuse, this library is designed to use abstract implementation of `DFS`. This algorithm could be parallelized by executing the `compute` on the subarrays concurrently, lines 18-19. Since we may not know about the effect of each subclass of the class `Mapper` and `Reducer`, we declare them as `open` fields, lines 22-23 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.

When the input range is small, the base case applies, on lines 10-11 where the `map` method is applied on each element of the array on line 13. The results from the `Mapper` are combined, on line 14 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, lines 19-20. Since we may not know about the effect of each subclass of the class `Mapper` and `Reducer`, we declare them as `open` fields, lines 22-23 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 includes two interim expressions for semantics: loc that represents locations and yield that models concurrency.

```
prog := decl e
decl := class c extends d { field meth }
field := [open] c f;
method := t m ( arg ) { e }
t := c void
arg := var null arg = e e i < j
   | new c() e | m( )
   | e.f e = e
fork ( e : c )
loc yield e
where
   c ∈ C, a set of class names
   d ∈ D ∪ { Object }, superclass names
   f ∈ F, a set of field names
   m ∈ M, a set of method names
   var ∈ V ∪ { this }, V is a set of variable names
```

Figure 4. The Grammar for OpenEffectJ.

The notation over-bar denotes a finite ordered sequence (π stands for a1...an). The notation [a] means that a is optional. The examples in Section 2 used an extended language with integers and boolean constants and operations, e.g., int is used as a shorthand for Integer, i < j is for i.lt(j), etc. As in Fork/Join framework [26], we desugar a for loop to the code below.

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

OpenEffectJ provides one construct for expressing parallelism: the fork expression. This expression has the form fork(e1,e2). It runs e1 and e2 in two concurrent tasks. Concurrent execution of these tasks is dependent upon whether they may conflict. If these tasks do not conflict they are run concurrently, else sequentially.

A novelty in OpenEffectJ’s semantics is that this parallelization decision is based upon the information available at the time of the evaluation of a fork expression. Delaying parallelization decision allows the use of useful dynamic information. This is similar to Best et al.’s synchronization via scheduling [5], but differs from static [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 open, using the annotation @open (e.g., line 2 in Figure 2).

To focus our attention on the essence of @open effects and to make the presentation tractable, we have formalized a subset of OpenEffectJ. In the core language, only fields can be annotated @open, but not other references, e.g., variables and parameters. In Section 5, we will discuss how we could extend the formalism to model other @open references. So all the examples presented work out nicely as they are.

4. Type and Static Effect Computation

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

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

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 fork expressions.

4.1 Notations and Conventions

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

```
θ := OK [method/decl/body types]
   | { t1 × ... × tn → t, ρ } in c [method types]
   | (t, ρ) [expression types]
ρ := { ρj } j∈N [program effects]
   | read f [read effect]
   | open f m p [open effect]
   | ⊥ [bottom effect]
Π := { varj → tj } j∈N [type environments]
```

Figure 5. Effect and type attributes.

The notation t'<:t means t' is a subtype of t. It is the standard reflexive-transitive closure of the declared subclass relationship [19]. We will use the notation ρ0 ρ1 to mean that effects ρ0 and ρ1 do not interfere and ρ0 ρ1 to mean that they do interfere.

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

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

4.2 Type-and-Effect Rules for Method Declaration

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

```
(T-METHOD)
   override(m,c,(t1 × ... × tn → t))
   ∀ i ∈ {1..n}, isClass(t[i]) isType(t)
   [var1 : t1, ..., varn : tn, this : c] + e : (u,ρ) u < t
   | t m | [var1 : t1, ..., varn : tn] e : [t1 × ... × tn → t, ρ] in c
```

2 We simplify the presentation by avoiding tracking object instances in read/write effects. In the implementation, OpenEffectJ tracks object instances and uses a sound intra-procedural aliasing analysis [21] and a purity analysis [20] 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.

\[ CT(c) = \text{class } c \text{ extends } d \{ \text{field meth}_1 \ldots \text{meth}_p \} \]

\[ \exists i \in \{1 \ldots p\} \text{ s.t. } \text{meth}_i = t(m_1 \text{var}_v \ldots t_n \text{var}_n)\{e\} \]

\[ \text{override}(m, d, (t_1 \times \ldots \times t_n \rightarrow t)) \]

\[ (d, t.m(t_1 \text{var}_v \ldots t_n \text{var}_n)\{e\}, \rho') = \text{findMeth}(c, m) \]

\[ \text{override}(m, c, (t_1 \times \ldots \times t_n \rightarrow t)) \]

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

\[ CT(c) = \text{class } c \text{ extends } d \{ \text{field meth}_1 \ldots \text{meth}_p \} \]

\[ \exists i \in \{1 \ldots p\} \text{ s.t. } \text{meth}_i = (t, \rho.\{ \text{var}\}) \{ e \} \]

\[ \text{findMeth}(c, m) = (e, t.m(\{ \text{var}\})\{e\}, \rho) \]

\[ CT(c) = \text{class } c \text{ extends } d \{ \text{field meth}_1 \ldots \text{meth}_p \} \]

\[ \exists i \in \{1 \ldots p\} \text{ s.t. } \text{meth}_i = (t, \rho.m(\{ \text{var}\})\{e\}) \]

\[ \text{findMeth}(c, m, l) = l \]

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.

\[ \text{T-CALL} \]

\[ e_0 \neq \text{this.f} \lor (e_0 = \text{this.f} \land \text{typeOf}(f) \neq (c, \text{open } c_0)) \]

\[ \forall i \in \{1 \ldots n\} \land \Pi \vdash e_i : (t', \rho.\{ \text{var}\}) \land t_i' < t_i \]

\[ \Pi \vdash e_0.m(e_1, \ldots, e_n) : (t, \{ \text{open } f \ldots m \}) \]

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. 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-CALL-OPEN) applies when a method \( m \) is called with an open field \( f \) as a receiver object.

\[ \Pi \vdash \text{this.f} : (c_0, \rho_0) \text{ typeOf}(f) = (d, \text{open } c_0) \]

\[ \forall i \in \{1 \ldots n\} \land \Pi \vdash e_i : (t', \rho.\{ \text{var}\}) \land t_i' < t_i \]

\[ \Pi \vdash \text{this.f.m(e_1, \ldots e_n)} : (t, \{ \text{open } f \ldots m \} \cup \bigcup_{i=0}^{n} \rho_i) \]

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

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., \( e \neq \emptyset \) \( m \not\in \rho \)), OpenEffectJ acts just like a static type-and-effect system. At compile time, it advises a parallelizer compiler to parallelize id0 and id1 if \( \rho_0 \) and \( \rho_1 \) do not interfere, i.e., \( \rho_0 \cap \rho_1 \not= \emptyset \). If \( \rho_0 \) and \( \rho_1 \) interfere \( (\rho_0 \not\subseteq \rho_1) \), it suggests sequential execution.

Consider the scenario where open effect exists in \( \rho_0 \) and \( \rho_1 \) (i.e., \( e \neq \emptyset \) \( m \not\in \rho \)). Let \( \rho \) be the concrete effects in set \( \rho \), i.e., \( \rho = \{ (e \in \rho) \land (e \neq \text{open } f \ldots m) \} \). At compile time, OpenEffectJ advises a parallelizing compiler to parallelize id0 and id1 if \( \rho_0 \not\subseteq \rho_1 \), but to insert an effect check right before parallelizing them (e.g., before evaluating fork \((e_0, e_1)\)). In Section 5 we will illustrate one such effect checking technique via the semantics of the fork expression.

4.4 Type-and-effects for Field Related Expressions

The typings and effects for the field access rules (T-GET) and (T-SET) are standard. The auxiliary function typeOf, 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 \( f \).

\[ (T-GET) \]

\[ \Pi \vdash c : (c, \rho) \text{ typeOf}(f) = (d, \text{open } t) \]

\[ \Pi \vdash e.f : (t, \rho \cup \{ \text{read } f \}) \]

\[ (T-SET) \]

\[ \Pi \vdash e : (c, \rho) \text{ typeOf}(f) = (c', \text{open } t) \]

\[ \Pi \vdash e' : (t', \rho') \]

\[ (t', \rho \cup \rho' \cup \{ \text{write } f \}) \]

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-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.

\[ (T-YIELD) \]

\[ \Pi \vdash e : (t, \rho) \]

\[ \Pi \vdash \text{yield} : (t, \rho) \]

\[ (T-FORK) \]

\[ \Pi \vdash e : (t, \rho') \]

\[ \Pi \vdash \text{fork} : (e', \{ \text{void}, \rho \}) \]

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

5. Dynamic Semantics with Open Effects

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

5.1 Notations and Conventions

The small steps taken in the semantics are defined as transitions from one configuration (\( \Sigma \) in Figure 6) to another. Some rules use an implicit attribute, the class table \( CT \).
Evaluation relation: $\leadsto \Sigma \rightarrow \Sigma$

Domains:
- $\Sigma ::= (\psi, \mu) \quad \text{"Program Configurations"}$
- $\psi ::= (e, \tau) + \psi \mid \bullet \quad \text{"Task Queue"}$
- $\tau ::= (id, [id_{j} \mid j \in N]) \quad \text{"Task Dependencies"}$
  where $id, id_{j} \in N$
- $\mu ::= \{loc \rightarrow o\}_{j \in N} \quad \text{"Store"}$
- $o ::= \{c, F, E\} \quad \text{"Object Records"}$
- $F ::= \{f_{j} \rightarrow v_{j}\}_{j \in N} \quad \text{"Field Maps"}$
- $v ::= \text{null} \mid \text{loc} \quad \text{"Values"}$
- $E ::= \{m_{j} \rightarrow p_{j}\}_{j \in N} \quad \text{"Effect Maps"}$

Evaluation contexts:
- $E ::= - \mid E.m(\tau) \mid v.m(\tau, E, \pi) \mid E.f=e \mid v.f=E \mid E; e \mid E.f \mid t \vartriangledown E; e$

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 $o = \{c, F, E\}$ contains the concrete type $c$ of the object, a field map $F$, and a dynamic effect map $E$ (which is new).

An effect map $E$ is a function from a method name to its runtime effects. The task queue $\psi$ 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 $\Sigma_{s} = (e, (0, 0), \bullet)$. The operator $\oplus$ is an overriding operator for finite functions, i.e., if $\mu' = \mu \oplus \{loc \rightarrow o\}$, then $\mu'(loc) = o$ if $loc' = loc$, otherwise $\mu'(loc') = \mu(loc')$.

5.2 Dynamic Effect Management in OO Expressions

The semantics rules for standard OO expressions are shown below (Section 3.4 contains omitted auxiliary functions). Compared to traditional dynamic semantics for OO expressions [19], there are two main differences. First, the yield expression is used in the resulting configuration to relinquish control to other tasks. Second, some of the rules manipulate the dynamic effect map $E$.

\begin{align*}
\text{(NEW)} \quad \mu &= \{loc \rightarrow [c, f \mapsto \text{null} \mid f \in \text{fields}(c)]\} \oplus \mu \\
\langle \langle E, \text{new}(c), \tau \rangle + \psi, \mu \rangle &\rightarrow \langle \langle E, \text{yield}(c), \tau \rangle + \psi, \mu \rangle
\end{align*}

The (NEW) rule uses a function $\text{methE}$ (below) to initialize the effect map of the new instance. This function searches the class table $CT$ for all the methods declared in class $c$ and all its super classes. Its result is a map that contains each method $m$ found in previous step and its statically computed effects $\rho$. The type-and-effect rules in Section 4.3 [1] are used to compute the effects $\rho$.

\begin{align*}
\text{methE}(c) &= E \cup \bigcup_{m_{i} \in \text{fields}(c)} \{m_{i} \rightarrow \rho_{i}\} \\
\text{where} \quad CT(c) &= \text{class c extends} \{\text{field meth1 \ldots methn}\} \\
\text{and} \quad \text{methE}(d) &= E \\
\text{and} \quad (\forall i \in \{1 \ldots n\} :: \text{findMeth}(c, m_{i}) = (c, t, m_{i}(\text{var}), \rho_{i}))
\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., $\langle \text{open f m} \rho \rangle$.

\begin{align*}
\text{concretize}(\mu, f, v, \varepsilon) &= \text{match } \varepsilon \text{ with} \\
| \text{@open f' m }& \rightarrow \text{match f' with} \\
| f &\rightarrow \text{match v with} \\
| \text{null }&\rightarrow \text{@open f m } \emptyset \\
| \text{loc }&\rightarrow \text{mq }f \rightarrow \rho' \text{ where } c, F, E = \mu(loc), \\
&\quad \text{and } \rho' = \bigcup_{i \in \{1 \ldots n\}} \rho_{i} \text{ and } E = \{\varepsilon_{1} | 1 \leq i \leq n\} \\
&\quad \text{and } \forall f \in \{1 \ldots n\} :: cp(\varepsilon_{i}) = \rho_{i} \\
| \_ &\rightarrow \varepsilon
\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}(\varepsilon) &= \text{match } \varepsilon \text{ with} \\
| \& \text{open f m }& \rightarrow \rho \rightarrow \varepsilon
\end{align*}

The function $\text{cp}$ is used by the function $\text{concretize}$ to retrieve the concrete effects, i.e., the effects of the R-value $v$ are copied to fill the placeholder effects of the open effect of $loc$ in the (SET) rule.
The (CALL) rule is standard. It acquires the method signature via the function findMeth (Section 4.1) that uses dynamic dispatch[19].

\[
\text{(CALL)} \\
(c', t. m(t | var_1, \ldots, var_n)(e), \rho) = \text{findMeth}(c.m) \\
\left[\text{c.F.E} = \mu(\text{loc}) \quad e' = \lfloor \text{loc}/\text{this} \rfloor v_1/\ldots/v_n/\var_\text{m} \text{e} \right] \\
\langle \langle \text{E.open.m(v_1, \ldots, v_n), } \tau \rangle + \psi, \mu \rangle \mapsto \langle \langle \text{E.yield e'}, \tau \rangle + \psi, \mu \rangle
\]

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

5.3 Safe Optimistic Concurrency using Open Effect

We now describe how open effects are used by a fork expression. Recall from Section 4 that OpenEffectJ’s static effect system would advise a parallelizing compiler to parallelize fork(e0; e1) if and only if statically computed effects of e0 and e1 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 e0 and e1 could run in parallel. To do so, we use the effect judgments in Figure 2 that recursively computes the effects of subexpressions. An effect judgment 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 e0 and e1 can be wrapped into two compiler-generated methods m0 and m1; OpenEffectJ can then retrieve the effects of e0 and e1 from the effect map E of the this object, thus eliminating the need for this dynamic effect computation.

\[
\text{(E-call-loc)} \\
\mu(\text{loc}) = \lfloor \text{c.F.E} \rfloor \\
E(\text{m}) = \rho_0 \\
\langle \forall i \in \{1..n\} : \mu \vdash e_i : \rho_i \rangle \\
\mu \vdash \text{loc.m}(\tau) : \bigcup_{i=0}^n \rho_i
\]

\[
\text{(E-set-open)} \\
\text{typeOf}(f) = (\text{c.@open } e_0) \\
\mu \vdash e.f : e' : (\bot)
\]

\[
\text{(E-set)} \\
\mu \vdash e : \rho \\
\mu \vdash e' : \rho' \\
\text{typeOf}(f) = (c.t) \\
\mu \vdash e.f.e' : \rho \cup \rho' \cup \{\text{read } f\}
\]

\[
\text{(E-call-open)} \\
\mu \vdash e : \rho \\
\mu \vdash f : \rho' \\
\text{typeOf}(f) = (c.t) \\
\mu \vdash e.f.e' : \rho \cup \rho' \cup \{\text{read } f\}
\]

\[
\text{(E-call-loc)} \\
\mu \vdash e_0 : \rho_0 \\
\mu \vdash \text{loc.m}(\tau) : \bigcup_{i=0}^n \rho_i
\]

\[
\text{(E-yield)} \\
\mu \vdash e : \rho \\
\mu \vdash \text{yield } e : \rho
\]

\[
\text{(E-new)} \\
\mu \vdash \text{new } c() : \emptyset \\
\mu \vdash \text{var} : \emptyset \\
\mu \vdash \text{null} : \emptyset \\
\mu \vdash \text{loc} : \emptyset
\]

Figure 7. Effect judgment for expressions.

The dynamic rules in Figure 2 are similar to the static rules in Section 4 except for method calls on open fields (E-call-open). The open effect now has a concrete part \(\rho\), 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 e0 and e1 do not interfere (written \(e_0\|e_1\)). Read effects do not interfere; read/write and write/write pairs conflict if they access the same field; open effect \(\text{open } f\text{ m}\) conflicts with another effect \(e\) if any effect \(e'\) in \(\rho\) conflicts with \(e\); bottom effect \(\bot\) conflicts with any effect.

If e0 and e1’s effects do conflict, the (fork-sequential) rule applies; otherwise the (fork-parallel) rule will be used. OpenEffectJ makes its concurrency decision at this point, which allows the usage of more accurate effect information than a pure static analysis. A pure dynamic analysis will optimistically execute the tasks in concurrent. But they may rollback when conflicts do happen (see Section 7 for more discussion). The (fork-parallel) rule creates 2 concurrent children tasks id0 and id1 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.

\[
\text{(fork-parallel)} \\
\mu \vdash e_0 : \{e_1, .. e_n\} \\
\mu \vdash e_1 : \{e'_1, .. e'_m\} \\
\forall i \in \{0..n\}, j \in \{0..n'\} \text{ s.t. } (e_i \neq e_j) \\
id_0 = \text{fresh}(\tau) \\
id_1 = \text{fresh}(\tau) \\
\tau' = (id, \{id_0, id_1\} \cup \{e_1, id_1, 0\}) \\
\langle \langle \text{E.fork (e0; e1)}, \tau \rangle + \psi, \mu \rangle \mapsto \langle \langle \text{E.yield e}, \tau \rangle + \psi, \mu \rangle
\]

The (fork-sequential) rule constructs an expression \(e_0; e_1; \text{null}\), 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.

\[
\text{(fork-sequential)} \\
\exists i \in \{0..n\}, j \in \{0..n'\} \text{ s.t. } (e_i \neq e_j) \\
e = e_0; e_1; \text{null} \\
\langle \langle \text{E.fork (e0; e1)}, \tau \rangle + \psi, \mu \rangle \mapsto \langle \langle \text{E.yield e}, \tau \rangle + \psi, \mu \rangle
\]

Finding an active task is done by the function active (not shown). It returns the top most task in \(\psi\) 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 \(\eta \prec \rho\), 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 \( \text{dyn} \) which augments dynamic semantics in Section 5 with dynamic effects and an additional parent-child relationship \( \text{id} \) which is a map \( \{ \text{id}_j \rightarrow \text{id}_r \} \). Here, id is a task’s identity and \( \text{id}_r \) is a set of its children tasks’ identities.

The function \( \text{dyn} \) (below), records the dynamic memory footprint for each task \( \text{id} \) 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.

**The novelty of OpenEffectI is that it stores the effects \( \rho \) in object records for methods. These methods 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 supersuperef \( \supseteq \) of the effect \( E'(m) \) for the method \( m \) of the object \( f \) is pointing to (Definition 5.4), i.e., \( E'(m) \subseteq \rho_0 \), and the effect \( E(m) \) of a method \( m \) stored in the object record should be supersuperef of the effect \( \rho \) of the body \( e \) of \( m \) (Definition 5.5). 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 supersuperef of the effect of the method \( \text{run} \) of \( p \).

**DEFINITION 5.1. [Well-formed object] An object record \( o = \langle E, F, E' \rangle \) is a well-formed object in \( \mu \), written \( \mu \vdash o \), if for all open effect \( \text{open} f m \rho_0 \in \text{rng}(E) \), either \( (F(f)) = \text{loc} \wedge (\mu(\text{loc}) = \langle E', E' \rangle) \wedge (E(m) \subseteq \rho_0) \); or \( (F(f) = \text{null}) \wedge (\rho_0 = 0) \).

**DEFINITION 5.2. [Well-formed location] A location loc is well-formed in \( \mu \), written \( \mu \vdash \text{loc} \), if each \( \mu(e) = \langle E, F, E' \rangle, \forall m \in \text{dom}(E) \), s.t. findMeth(c.m) = \langle e', t, m, 1/var \rangle(\text{e}, \rho') \wedge \mu^+\text{loc/this} = \rho, \text{then } \rho \subseteq \mu(\text{loc}); \text{or } \mu(\text{loc}) = \text{null} \).

**DEFINITION 5.3. [Well-formed store] A store \( \mu \) is well-formed, written \( \mu \vdash \), if \( \forall o \in \text{rng}(\mu) \), s.t. \( \mu \vdash o \) and \( Vloc(\mu) \), s.t. \( \mu \vdash l\).

**THEOREM 5.4. [Effect preservation] Let the program configuration be \( \Sigma = \langle \langle e, (id, I) \rangle, \psi, \mu \rangle \). If it transits to another configuration \( \Sigma' = \langle \langle e', (id, I') \rangle, \psi', \mu' \rangle \), the store is well-formed \( \mu \vdash \), \( \mu' \vdash e', \text{and } (\mu', \mu, \psi, \psi') \). If there is some \( p', \chi \) s.t. \( \mu^+ e' = p', \text{and } p' \subseteq \rho; \text{and } b(dyn(\Sigma, \chi, \psi)) \Rightarrow (\eta \propto \rho) \).

**Proof Sketch:** The essence of Theorem 5.4 is that during program execution, the subsequent expression \( e' \) has a subeffect \( p' \subseteq \rho \) of the previous expression \( e \), with the effect judgment \( \mu \vdash e' \). (Figure 7). We prove that the dynamic effect \( \eta \) in each step refines the static \( \rho \) of the original expression \( e \), \( \eta \propto \rho \). Thus with \( \eta \) 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 \( \rho_0 \vdash e_0 \). Unlike the static approaches, which compute \( \rho_0 \) 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 \( \text{dynE} \). These effects, produced by a concurrent task \( i \) and all its descendants \( i' \) (\( i' \leq_{\text{r}} i \)), refine the static effect computed when id is forked off. The function \( \text{dynE} \) is \( \text{dynE}(\Sigma, \chi, \psi, n) = \text{dynE}(\Sigma', \chi', \psi', n', 1) \), if \( \text{dyn}(\Sigma, \chi, \psi, n) = (\Sigma', \chi', \psi', n', 1) \), where \( \Sigma = (\Sigma, \chi, \psi) \) and \( \text{dyn}(\Sigma, \chi, \psi, n) = (\Sigma, \chi, \psi, n') \).

A task \( i \) is a descendant of a task \( i', \text{with } \Sigma \), written as \( i' \leq_{\text{r}} i \), if \( i' \in \text{desc}(id, \Sigma) \). Here, \( \text{desc}(id, \Sigma) = \bigcup_{0}^{n} I, I(id) = I \cup \{0, \ldots, i_0\}, \text{and } \forall i_0 \in I, \text{desc}(id, I) = I \).

To reason about the interaction 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 \( \Sigma \rightarrow \Sigma' \), \( \Sigma = \langle \langle \text{fork}(e', \psi'), \tau, \psi', \mu \rangle, \Sigma' = \langle \langle e_0, \psi, \tau + \psi + \mu \rangle, \rangle \).

To say that concurrent tasks do not interfere (heap accesses), we define the noninterference relation for dynamic effects (\( \eta \eta' \)) 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 \( \psi = \{\eta, \eta'\} \) do not conflict with another set \( \psi' = \{\eta', \eta''\} \) if \( \forall i \in \{1, \ldots, j\}, e \in \{1, \ldots, p\} \), s.t. \( \psi = \{\eta, \eta'\} \).

**THEOREM 5.5. [Deterministic semantic] Let \( \mu \vdash o \), \( \Sigma \rightarrow \Sigma' \), and \( \Sigma = \psi' \), where \( \Sigma = \langle \langle \text{fork}(e_1, e_2) \rangle, \tau, \psi \rangle \). For the two tasks \( \langle e_1, (id_1, I) \rangle \in \psi' \) and \( \langle e_2, (id_2, I') \rangle \in \psi' \), \( \mu \vdash e_1 \); \( \mu \vdash e_2 \); \( \rho_1 \vdash e_1 \); \( \rho_1 \vdash e_2 \); \( \eta, \eta' \) and \( \eta, \eta' \) do not interfere different effect judgments \( \rho_1, \rho_2 \). Then \( \mu \vdash \text{fork}(e_1, e_2) \); \( \mu \vdash \text{null} \); \( \mu \vdash e_1 \); \( \rho_1 \vdash \text{null} \); \( \rho_2 \vdash \text{null} \). Theorem 5.5.1 of a task \( i \). We need \( \Sigma \) because the effect of a child task \( i \) should count as \( i \)'s effect.

Proving the above soundness theorems are non-trivial compared to static effect approaches \( \text{OpenJDK} \), 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 5, 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 \( \text{open} var \) for a method call \( \text{var.m}(...) \). At runtime, the concretization of the open effect happens in the \( \text{fork} \) rule. At runtime, if \( var \) is bound to a location \( loc \), the \( \text{E-CALL-LOC} \) rule in Figure 7 takes effect. It transits from the placeholder effect \( \text{open} var \) to the concrete effect \( \rho_0 \) (the second item in \( \text{E-CALL-LOC} \)). With this concretized effect, the \( \text{fork} \) 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
We have conducted an initial evaluation of OpenEffectJ Compiler.

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 \( \text{update} \). 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, when the mapper of an instance of class MapReduce changed, concrete effect of method compute will change also.

We generate a method \( \text{cascade} \) 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 \( \text{cascade} \) 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

\[
\text{Math.sqrt}(2 * \text{Math.pow}((\text{double})o, 2))
\]

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 8. 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 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 OpenJDK compiler at a reasonable cost (single digit overheads). To put these overheads in perspective, a dynamic analysis can sometimes incur 2X overhead [18]. 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, 23, 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 [2,16,18,19]. In essence, they monitor memory footprints of tasks and signal when conflicts are detected. In contrast, OpenEffectJ detects conflicts just before forking off parallel tasks.

Transactional memory [22,27,37,40] optimistically executes tasks concurrently, but monitors memory accesses. It rollsbacks 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 on conflict. In OpenEffectJ, state buffering is not needed, because conflicts are detected before parallel code.

In concurrent revisions [9,10] programmers know that tasks conflict on shared objects and annotate these objects. 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>SM</th>
<th>OO</th>
<th>DS</th>
<th>IC</th>
<th>DT</th>
<th>OPT</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenEffectJ</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Hybrid</td>
<td>Fork Point</td>
</tr>
<tr>
<td>DPJ [6,34]</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Static</td>
<td>Static</td>
</tr>
<tr>
<td>Galois [16]</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Dynamic</td>
<td>Complete</td>
</tr>
<tr>
<td>Ownership [11,13]</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Hybrid</td>
<td>Fork Point</td>
</tr>
<tr>
<td>Actor [3]</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Static</td>
<td>Static</td>
</tr>
<tr>
<td>TM [22,27,37,40]</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Dynamic</td>
<td>Complete</td>
</tr>
<tr>
<td>FastTrack [18, CP 35]</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Dynamic</td>
<td>Complete</td>
</tr>
<tr>
<td>Goldilocks [16]</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Hybrid</td>
<td>Complete</td>
</tr>
<tr>
<td>Revision [9,16]</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Dynamic</td>
<td>Complete</td>
</tr>
<tr>
<td>X10 [12]</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Dynamic</td>
<td>Complete</td>
</tr>
</tbody>
</table>

In shared memory (SM) systems, tasks communicate via accessing the shared memory space, while in the distributed memory systems, tasks communicate via messages. X10 is classified as shared memory because places can share heap objects. As the main stream languages (C++, Java and C#) adopt the shared memory and object-oriented (OO) models, it may require more intellectual efforts to use distributed models [31] and/or the non-OO models.

In a programming model with deterministic semantics (DS), a given input will produce the same result. Programmers generally find it easier to reason about deterministic programs. The actor model, due to its asynchronous nature, does not provide deterministic semantics, since the order of the arrival of the messages is arbitrary [31]. Galois targets applications that do not require determinism. Normally, TM has no determinism guarantee due to its nondeterministic commit order, except for deterministic TMs.

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

The last two columns, deployment time (DT) and optimism level (OPT) show when the systems are activated and how optimistic they are in concurrency. A static approach does reasoning at compile time, has least runtime information and is least optimistic. A hybrid analysis, like OpenEffectJ, uses static information to facilitate the runtime analysis and is more optimistic than a static one. Dynamic analysis 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 [3].

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 over-
head. It is less precise compared to a pure dynamic analysis, but 
would detect conflicts before they occur. Thus, state buffering for 
rollbacks is not necessary and overhead is less. In future, it would 
be sensible to explore a logical extreme, where every reference is 
implicitly open and a static analysis is used to systematically elim-
inate 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

doeego.edu/dl/concurrency-interest/


ploring large-scale parallelism. In FSTTCS, pages 19–41, Springer, 
1985.


[5] M. J. Best, S. Mottishaw, C. Mustard, M. Roth, A. Fedorova, and 
A. Brownsword. Synchronization via scheduling: techniques for ef-

ciently managing shared state. In ACM SIGPLAN Conference on 

programming with object-oriented parallel frameworks. In European 

detection of data races. In ACM SIGPLAN Conference on Program-

[8] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe pro-


the price of one: a model for parallel and incremental computation. 
In OOPSLA, 2011.


[12] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, 
approach to non-uniform cluster computing. In OOPSLA, pages 519– 
538, October 2005.


ware behavior oriented parallelization. In ACM SIGPLAN Conference on 

transaction-aware Java runtime. In ACM SIGPLAN Conference on 

ACM SIGPLAN Conference on Programming Language Design and 

[18] C. Flanagan and S. N. Freund. FastTrack: efficient and precise dy-
namic race detection. In ACM SIGPLAN Conference on Programming 

tion Semantics for Classes and Mixins. In Formal Syntax and Seman-

ative programming. In The acm conference on LISP and functional 


[23] I. T. Kassios. Dynamic frames: Support for framing, dependencies and 


[25] K. Knowles and C. Flanagan. Hybrid type checking. TOPLAS, 32: 


the ACM SIGPLAN-SIGACT symposium on Principles of program-

Univ., July 2012.

ACM SIGPLAN-SIGACT symposium on Principles of programming 

ACM SIGPLAN Conference on Programming Language Design and 
Implementation (PLDI), 2007.

[31] P. Mackay. Why has the actor model not succeeded? Technical 

[32] M. Naik and A. Aiken. Conditional must not aliasing for static race 
detection. In the ACM SIGPLAN-SIGACT symposium on Principles of 

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

[34] R. Bocchino et al.. A type and effect system for deterministic parallel 

framework for parallelizing compilers. In ACM SIGPLAN Conference on 

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

[37] N. Shavit and D. Touitou. Software transactional memory. In PODC, 

[38] J. Siek and W. Taha. Gradual typing for objects. In European 

[39] Y. Smaragdakis, J. M. Evans, C. Sadowski, Y. Jaeheon, and C. Flana-
gan. Sound predictive race detection in polynomial time. In the ACM 
SIGPLAN-SIGACT symposium on Principles of programming 
languages (POPL), 2012.

[40] T. Shepeisman et al. Enforcing isolation and ordering in STM. In 
ACM SIGPLAN Conference on Programming Language Design and 


[42] J. W. Young, R. Jalha, and S. Lerner. RELAY: static race detection on 

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 \( \rho \).

\[
\frac{\forall decl \in decl \vdash decl \vdash \text{OK}}{\vdash e : (t, \rho)}
\]

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.

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

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

\[
\begin{align*}
\text{CT}(c) &= \text{class } c \text{ extends } d \{ \text{field } \} \in CT \quad \text{isClass}(c) \quad \\text{validF}(f, d) \\
\text{validF}(f, \text{Object}) &= \text{class } c \text{ extends } d \{ \text{field } \} \in CT \\
& \quad \text{isClass}(t) \lor (t = \text{void}) \\
& \quad \text{isType}(t)
\end{align*}
\]

A.2 Type-and-Effect Rules for Expressions

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

\[
\begin{align*}
\text{T-NEW} & : \text{isClass}(c) \quad \Pi [\text{var} = t] \quad \Pi [\text{null} : (t, \emptyset)] \\
\Pi & \vdash \text{new } c() : (c, \emptyset) \\
\text{T-VAR} & : \Pi [\text{var} = t] \\
\Pi & \vdash \text{null} : (t, \emptyset) \\
\text{T-NULL} & : \Pi [\text{null} : (t, \emptyset)] \\
\Pi & \vdash \text{null} : (t, \emptyset)
\end{align*}
\]

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

The auxiliary function typeOfF (used in the rules in Section A.2) 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 \( f \).

\[
\text{typeOfF}(f) = (c, t) \quad \text{where } \forall i \in \{1..n\} : \text{field}(f, i) = (f, t)
\]

B. Dynamic Semantics: Omitted Details

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

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

\[
\begin{align*}
\text{fields}(c) &= \text{Fs} \cup \{ f_1 \ldots f_n \} \\
\text{where } CT(c) &= \text{class } c \text{ extends } d \{ \text{field } \} \\
\text{and } \text{fields}(d) &= \text{Fs} \quad \forall i \in \{1..n\} : \text{field}(f, i) = (f_i, t_i)
\end{align*}
\]

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

\[
\begin{align*}
\text{fixPoint}(\mu, \text{loc}, \kappa) = \mu_s \quad \text{where } \kappa &= \{ \langle \text{loc}_i, f_i \rangle | 1 \leq i \leq n \} \\
\text{and } \text{update}(\mu, \text{loc}_1, f_1, \text{loc}) &= \mu_s \\
\text{and } \forall i \in \{2..n\} : \text{update}(\mu, \text{loc}_{i-1}, f_i, \text{loc}) = \mu_i
\end{align*}
\]

The function active (below) 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 single value \( v \)).

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

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 \( \psi \).

\[
\begin{align*}
\text{intersect}((\text{id}, \emptyset), \psi) &= \emptyset \\
\text{intersect}((\text{id}, \{ \text{id}_1, \ldots, \text{id}_n \}), \psi) &= \emptyset \quad \text{where } \forall i \in \{1..n\} : \text{inQueue}(\text{id}_i, \psi) = b_i
\end{align*}
\]

The function inQueue, used by the intersect function, searches whether there is a task \( \text{id} \) matching the input value \( n \).

\[
\begin{align*}
\text{inQueue}(n, \bullet) &= \emptyset \\
\text{inQueue}(n, \langle e, n, \{ \text{id}_j | j \in \text{N} \} \rangle + \psi) &= \emptyset \\
\text{inQueue}(n, \langle e, n', \{ \text{id}_j | j \in \text{N} \} \rangle + \psi) &= \text{inQueue}(n, \psi) \quad \text{if } n \neq n'
\end{align*}
\]

This function gets the tasks from the task queue \( \psi \) and matches the input \( n \) with the id of the tasks. If one task matches, the function returns true. Otherwise it continues searching the rest of the tasks in the queue until one of them matches or none of the tasks matches.
C. Proof of Key Properties

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

We have proven the soundness of OpenEffectJ’s type system (Section C.1) contains proof that use the standard subject reduction argument [13]. The Effect preservation property is that the dynamic effect (heap accesses) of each concurrent task id refines the static effect computed when id is forked off. We prove this in Section C.2. We prove the determinism of OpenEffectJ programs by showing that the concurrent tasks do not have dynamic effect interference and therefore a well-typed OpenEffectJ program produces the same result given the same input in Section C.3. Proving effect soundness is non-trivial compared to static effect approaches [13, 34], in which the exact effect of a task is known statically. A technical challenge for proving the soundness of OpenEffectJ is that the effects of the concurrent tasks may change due to the open effect, i.e., the effect concretization.

C.1 Preliminary Definitions

We now give some preliminary definitions used in the proofs for OpenEffectJ’s properties. A standard approach to show determinism, for multi-tasking systems, is to prove that the heap accesses of the tasks do not interfere [34]. To record the heap accessed for each task, we define dynamic trace χ, which contains a sequence of dynamic effects (heap accesses) by the tasks. Later we will show that the dynamic effects of each task id refines the static effect ρ computed when id is forked off, s.t. if the static effects do not interfere, the dynamic effects will not interfere [13, 34] (Section C.4).

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

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

DEFINITION C.2. [Relationship] A relationship Υ for tasks is a map [id i → I] ∈ Υ. Here id i is a task’s identity and I is a set of its children tasks’ identities.

The function 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 ρ computed when it is forked off.

![Figure 10. Dynamic Effect function dyn.](image)

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

DEFINITION C.3. [Static effect inclusion] An effect e is included in an effect set ρ if e i is one of the elements in ρ; or there is an open effect \( \text{open} f m \rho \) in ρ and e is an element of \( \rho f \).

This definition says that an effect e is included in an effect set ρ if it is one of the elements in ρ; or there is an open effect \( \text{open} f m \rho \) in ρ and e is an element of \( \rho f \).

DEFINITION C.4. [Dynamic effect refines static effect] A dynamic effect E refines a static effect ρ, where (⊥) ∈ ρ, written \( \eta \vdash \rho \), if either \( \eta = (\text{rd.loc, f, id}) \land (\text{read f}) \in \rho \); or \( \eta = (\text{wt.loc, f, id}) \land (\text{write f}) \in \rho \).

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

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

During the evaluation of concurrent tasks, the store keeps changing, and we want to ensure that the same expression has the same static effect in the presence of task interleaving (Theorem C.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.6. [Effect equivalent stores] Two stores \( \mu \) and \( \mu' \) are effect equivalent, written \( \mu \equiv \mu' \), if both conditions hold: dom(\( \mu \)) ⊆ dom(\( \mu' \)); and \( \forall \text{loc} \in \text{dom}(\mu) = \{\text{c.F,E}\} \), then \( \mu'(\text{loc}) = \{\text{c.F,E}\} \), for some \( F \).

This definition says that two stores are effect equivalent if they have the same effects for all common locations. Except for the method call expression, proving that an expression has static effects that are refined by their subsequent expression is standard [13, 34]. The novelty is that effects for method calls are new in this work and OpenEffectJ needs to maintain proper effects for methods (Definition C.7, and Definition C.8). To prove that a method call on open field has static effects that are refined by their subsequent expression, we introduce well-formed object.

DEFINITION C.7. [Well-formed object] An object record o = \{\text{c.F,E}\} is a well-formed object in \( \mu \), written \( \mu \vdash o \), if for all open effect \( \text{open} f m \rho \in \mu \in \text{ring}(E) \), either \( (F(f) = \text{loc} \land (\mu(\text{loc}) = \{\text{c.F,E}\}) \land (E(m) \subseteq \rho) \); or \( (F(f) = \text{null}) \land (\rho(0) = \{\text{c.F,E}\}) \).

This definition says that an object record is well-formed, if all of its open effect \( \text{open} f m \rho \) is supereffect (2) of the expression that the method m of the object the field f is pointing to.

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

DEFINITION C.8. [Well-formed location] A location loc is well-formed in \( \mu \), written \( \mu \vdash \text{loc} \), if either \( \mu(\text{loc}) = \{\text{c.F,E}\} \), \( \forall \text{m} \in \text{dom}(E) \) s.t. \( \text{findMeth}(c.m) = (\langle e', m, t \rangle \lor \langle e, \mu \rangle \land \mu \vdash \text{loc}/\langle e, t \rangle) \land (\mu(\text{loc}) = \{\text{c.F,E}\}) \land \rho \), then \( \rho \subseteq \mu(\text{loc}) \); or \( \mu(\text{loc}) = \text{null} \).

A location loc is well-formed in a store \( \mu \), if the effect, of each method m of the object loc is pointing to, is supereffect (2) of the effect given by the effect judgment of the body e of the method m.

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

DEFINITION C.9. [Well-formed store] A store \( \mu \) is well-formed, written \( \mu \vdash o \), if \( \forall o \in \text{ring}(\mu) \) s.t. \( \mu \vdash o \) and \( \forall \text{loc} \in \text{dom}(\mu) \) s.t. \( \mu \vdash \text{loc} \).
The definition says that the store is well-formed, if all the locations and object records are well-formed.

C.2 Effect Preservation

In this section, we prove OpenEffect1’s effect preservation property. This involves three invariants during the evaluation of the concurrent tasks. First, the effect η produced by an expression e is refined by the effect ρ, by its subsequence expression e′ (e′ ⊆ ρ); and second, the dynamic effect η, produced by the reduction, if any, refines ρ (η ⊆ ρ); finally, the store remains effect equivalent (µ ≡ µ′).

THEOREM C.10. [Effect preservation] Let the program configuration Σ = (⟨e, (id, I)⟩ + Ψ, µ). If it transits to another configuration Σ ′ = (⟨e′, (id, I′)⟩ + Ψ′, µ′), the store is well-formed µ ⊩ o, µ ⊩ e : ρ, and (⊥) ⊄ ρ, then there is some ρ′, χ s.t.

(a) µ ≡ µ';
(b) µ ⊩ e' : ρ', and ρ′ ⊆ ρ;
(c) (dyn(Σ, χ, Y) = (Σ′, χ + η, Y′)) ⇒ (η ⊆ ρ).

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

C.2.1 Replacement with Subeffect

The following lemma says that two effect equivalent stores give the same effect ρ to the same expression e. Because we will prove that during the concurrent execution of the tasks, all the stores are effect equivalent µ ≡ µ′. Therefore, it suffices to prove that the effect of the subsequence expression e′ refines the effect of the expression e, given by effect equivalent stores.

LEMMA C.11. [Stationary effect] Let be an expression, µ and µ′ two stores s.t. µ ≡ µ′. If µ ⊩ e : ρ and (⊥) ⊄ ρ, then µ ⊩ e : ρ.

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

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:

µ ⊩ e0 : ρ0 µ ⊩ e1 : ρ1
µ ⊩ FORK (e0,e1) : ρ0 U ρ1

By the IH, ρ0 ⊆ ρ0 and ρ1 ⊆ ρ1. Therefore ρ′ = ρ0 U ρ1 = ρ0 U ρ1 = ρ, and the claim holds.

(CALL-OPEN) Here e = loc.m(e1, . . . , en). The last type derivation step has the following form:

µ ⊩ loc.m[Σ] : (open f m ρ′) U ∪ρ0

Clearly, ρ0 = ρ0 = {read f}. Since µ ≡ µ′, the effect maps E are the same. By the IH, ∀ i ∈ [1..n] :: ρ′i = ρ. Thus ρ′ = (open f m ρ′) U ∪ρ0 = (open f m ρ′) U ∪ρ0 = ρ, and the claim holds.

(CALL-LOC) Here e = loc.m[e1, . . . , en]. The last type derivation step has the following form:

µ ⊩ loc.m = [c.f.E] E(m) = ρ0
∀ i ∈ [1..n] :: µ ⊩ ei : ρ0

µ ⊩ loc.m[Σ] : ∪ρ0
µ ⊩ loc.m[Σ] : ∪ρ0

Since µ ≡ µ′, the effect maps E are the same and ρ0 = ρ0. By the induction hypothesis, ∀ i ∈ {1..n} :: ρ′i = ρi. Thus ρ′ = (open f m ρ′) U ∪ρ0 = ρ, and the claim holds.

Thus, for all possible derivations of µ ⊩ e : ρ and µ ⊩ e : ρ′, we see that ρ′ = ρ.

The following lemma says that given two effect equivalent stores, and the same evaluation context, if the effect of the subsequence expression 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 subsequence subexpression e′ refines the original subexpression e.

LEMMA C.12. [Replacement with subeffect] If µ ⊩ o, Σ ′ = (⟨E[e], (id, I)⟩ + Ψ, µ), Σ = (⟨E[e′], (id, I′)⟩ + Ψ′, µ′), then µ ⊩ E[e] : ρ, µ′ ⊩ E[e′] : ρ′, µ ⊩ e : ρ, and µ′ ⊩ e′ : ρ′, then µ′ ⊩ E[e′] : ρ′.

Proof: Proof is by induction on the size of the evaluation context E. Size of Σ 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 E′ ⊆ ρ ⊆ ρ = ∅. For the induction step we divide the evaluation context into two parts such that E[e] = E[1[E[2][e]]], 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 (Σ) 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 µ ⊩ E[2][e] : ρ implies that µ′ ⊩ E[2][e′] : ρ′, for some ρ′ ⊆ ρ, 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 ⊥ ⊆ ρ. The maximum, any effect ρ′ ⊆ ⊥.

Case −.f = e2 The last step for E[2][e] should be (E-SET):

µ ⊩ e : ρ0 typeOF(f) = (c.i) µ ⊩ e2 : ρ2
µ ⊩ E[2][e] : ρ0 U ρ2 U {write f}

By the definition of field lookup, typeOF(f) remains unchanged, i.e. typeOF(f) = (c.i). Thus, by (E-SET), µ′ ⊩ E[2][e′] : ρ1 U ρ2 U {write f};

Case v0.f = − The last step for E[2][e] should be (E-SET):

µ ⊩ E[2][e] : (c.i) µ ⊩ e : ρ0
µ ⊩ E[2][e] : {u0, p0} U {write f}

So µ′ ⊩ E[2][e′] : ρ1 U {write f};

Case −.m(e1, . . . , en) The last step in the effect derivation for E[2][e] should be (E-CALL-OPEN): e = loc.f

e = loc.f µ ⊩ loc.f = [c.f.E] E = {m1 → p0}[i ≤ i ≤ n]
∀ i ∈ [1..n] :: µ ⊩ ei : ρ0

Π ⊩ E[2][e] : (open f m ρ′) U ∪ρ0

By (GET), e′ = loc′:

e′ = loc′ µ ⊩ loc′ = [c.f′.E′] E(m) = ρ0
∀ i ∈ [1..n] :: µ ⊩ ei : ρ′i

Π ⊩ E[2][e′] : ρ′i U ∪ρ0

Because µ ⊩ o, by Definition C.9 and Definition C.7, we have ρ0′ ⊆ ρ′, ∀ i ∈ [1..n] . Thus, does not change, thus ρ′i = ρ′i. Therefore, the claim holds.
Case \[\text{loc.}\, m(v_1,\ldots,v_p-1,-,e_{p+1},\ldots,e_n)\] \(p \in \{1..n\}\). The last step for \(E_2[e]\) must be (E-CALL-LOC):

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

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

Using the lemmas To prove Theorem 2.2 in each reduction case, let \(e = E[v_0]\), \(e' = E[v_1]\), \(\mu = \rho_0 \rho_1\) and \(\mu' = e_1 : \rho_1\). Given that \(\mu \cong \mu'\), by Lemma C.12 and Lemma C.11, it suffices to prove \(\rho_1 \cong \rho_0\). We divide the cases into 3 categories: in the first category, some variables (var) will be replaced by actual values (\(v\), in Section C.2.2), the cases in the second category, access the store, in Section C.2.2, and the other cases are listed right below.

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

New Object: Here \(e = E[\text{new}\, c(e)]\), \(e' = E[\text{yield}\, loc]\), where \(loc \notin \text{dom}(\mu)\), \(\mu' = \rho\) and \(\mu' = e_1 : \rho_1\). Given that \(\mu \cong \mu'\), also new \(c\) \(\equiv \emptyset\) and \(\mu \vdash \text{yield}\, \text{loc} : \emptyset\) (and (b) holds).

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

Fork Sequential: Here \(e = E[\text{fork}\, (e_0; e_1)]\), \(e' = E[\text{yield\, e_1}]\) and \(e'_1 = e_0'; e_1'\). Let \(\mu = e_0 : \rho_0; \rho_1\) and \(\mu' = e_1 : \rho_1\). By (E-FORK), \(\mu \vdash e_0 : \rho_0; \rho_1\). We have \(\mu \vdash e_0'; e_1'\). (b) holds. Therefore, (b) holds.

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

C.2.2 Substituting Variables with Values

Here all the rules leave no dynamic trace, and (c) holds. Neither do they change the store, i.e., \(\mu = \mu'\) and \(\mu \cong \mu'\), (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 C.13. [Substitution effect] If \(\mu \vdash e : \rho\), then there is some \(\rho'\), such that \(\mu \vdash [v_1/var_1,\ldots,v_n/var_n]e : \rho'\), for all values \(v_i\) and free variables \(var_i\), and \(\rho' \subseteq \rho\).

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

The case for \(\text{(E-SET-OPEN)}\) and \(\text{(E-CALL)}\) hold because in these cases \(\bot \in \rho\). \(\bot\) is the maximum, any effect \(\rho' \subseteq \bot\).
Using the lemma We now prove cases for field accesses.

Field Get: Here \( e = \text{E}[\text{loc}, f] \), \( e' = \text{yield} v \), where \( \mu(\text{loc}) = [u.F, E] \), \( F(f) = v \), \( \mu' = \mu \) and \( \mu \cong \mu' \). Because \( \mu \vdash \text{loc} : \{\text{read} f\} \), and \( \mu' \vdash \text{yield} v : \emptyset \), (b) holds. Finally, \( \eta = [\text{rd}, \text{loc}, f, i.d.] \) and \( \eta \ovar para \{\text{read} f\} \leq \rho \), by Lemma C.14.

Field Set: Here e = E[loc, f] = v, e' = yield v, \( \mu' = \mu \oplus (\text{loc} \to \alpha) \), and \( \alpha = [u.F \iff (f \to v).E] \), where \( \mu(\text{loc}) = [u.F,E] \) and typeOff(f) = (c.i) for some t. The field is an open field, and by the function update, it does not update any effect, and \( \mu \cong \mu' \). To see \( \mu \vdash [\text{E}[\text{loc}, f] : v'] \leq \rho \), we have \( \mu \vdash \text{loc} : v' : \{\text{write} f\} \), and \( \mu \vdash v : \emptyset \), thus \( \rho' \leq \rho \). Finally, \( \eta = (\text{wt}, \text{loc}, f, i.d.) \), and \( \eta \ovar para \{\text{write} f\} \leq \rho \), by Lemma C.14.

Field Set Open: Here e = E[loc, f] = v, e' = yield v, \( \mu_0 = \mu \oplus (\text{loc} \to \alpha) \), and \( \alpha = [u.F \iff (f \to v).E] \), and \( \mu' = \text{update}(\mu_0, \text{loc}, f, v) \). Impossible, since \( (\text{loc}) \notin \rho \). □

C.3 Deterministic Semantics

The goal of this section is to show that the tasks generated by a fork expression do not interfere (Theorem C.25) and therefore OpenEffectI 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 id and all its descendants (Definition C.18), refine the static effect computed when id is forked off.

Definition C.15. [Multiple reduction steps] We write \( \Sigma \rhd_0 \Sigma' \) if \( \Sigma \rhd_\Sigma' \), and \( \Sigma \rhd_\Sigma' \Sigma'' \) if \( (\Sigma \rhd_\Sigma' \Sigma_0) \land (\Sigma_0 \rhd_\Sigma'' \Sigma') \). We write \( \Sigma \rhd_\Sigma \Sigma' \) if \( \exists n \geq 0 \text{ s.t. } \Sigma \rhd_\Sigma^n \Sigma' \).

Definition C.16. [Multi step effect] The multi step effect function dynE is: dynE(Σ, χ, Y, n) = dynE(Σ', χ', Y', n − 1), if dyn(Σ, χ, Y) = (Σ', χ', Y', 0) = dyn(Σ, χ, Y).

Here, the function dyn produces the dynamic effect and the relationship Y for an one step transition from a configuration to another. The function 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 \( \Sigma \rhd_\psi \psi' \), if \( \Sigma \rhd_\Sigma \Sigma' \), \( \Sigma = (\langle E \land \text{fork}(e, \psi) \rangle, \{\mu \land \psi\}, \mu) \) and \( \Sigma' = (\langle e_0, \{\mu \land \psi\}, \mu \rangle + \{\psi'\} \mu = \{\psi'\}) \).

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 (\( \varpi \varpara \eta \)) 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 \( \varphi = \{\eta_1 \varpara \eta_n\} \) do not conflict with another set \( \varphi' = \{\eta'_1 \varpara \eta'_m\} \) if \( \forall i \in \{1, \ldots, n\}, j \in \{1, \ldots, p\} \text{ s.t. } \eta_i \varpara \eta'_j \).

Definition C.18. [Descendant] A task id' is a descendant of a task id, with Y, written as \( \text{id}' \leq \text{id}, \text{if id}' \in \text{desc(id, Y)} \). Here, \( \text{desc}(id, Y) = Y \cup \bigcup_{i \in [0, \ldots, d]} Y(id) = I = \{\text{id}_0, \ldots, \text{id}_d\} \), and \( \forall \text{id}_d \in I, \text{desc}(id_d, Y) = I_d \).

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'.
Let \( \Sigma = (\mathcal{E}([e], (id, f')) + \psi, \mu_\Sigma) \). By Lemma C.20, \( \mu_j \vdash o \) and \( \mu_2 \vdash o \): 1. if \( id' = id_{j+1} \), by Theorem C.10 (a) holds; 2. \( id' \neq id_{j+1} \), then \( e_{j+1} = yield e' \) for some \( e' \). By the (YIELD) rule, \( \mu_{j+1} = \mu_j \). By IH, if \( \exists k \in \mathcal{E}([e], (id, f')) + \psi, \mu_\Sigma) \) and \( \mu' : e \to f \), then \( \mu' = \mu \). For all steps \( n < N \), we see that \( \eta \vdash \rho_n \) and for each local reduction \( \Sigma \Rightarrow \Sigma_0, \rho_n \vdash \rho_0 \). By Lemma C.22 the effect of a child task refines the corresponding fork expression of its parent task. Thus, \( dynESet(\Sigma', \chi, \Sigma) \Rightarrow \Sigma' \) and \( dynESet(\Sigma, \chi, \Sigma) \Rightarrow \Sigma' \).

**Theorem C.25.** [Deterministic semantic] Let \( \mu \vdash o, \Sigma \Rightarrow \Sigma', \Sigma \Rightarrow \psi' \) and \( \Sigma = (\mathcal{E}(\psi, (e, e'), \tau) + \psi, \mu_\Sigma) \). For the two tasks \( \langle e, (id, f', t) \rangle \in \psi \) and \( \langle e', (id, f, t') \rangle \in \psi' \), \( \mu \vdash e \to f \) and \( \mu \vdash e' \to f' \), if \( \rho_1 \vdash p_2 \) then \( \forall n \in \mathcal{E}([e], (id, f'), \tau) \to \psi' \). Let \( \mu \vdash e \to f \), then \( \mu \vdash \mu' \). Observe that all the conditions of Lemma C.24 are still correct in this theorem and the claim holds.

**C 4 Type Soundness**

In this section, we prove the standard type preservation property. Type rules omitted in Section C.3 are in Figure 11. To prove the type preservation, we extend the type environment, which maps variables and locations to types.

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

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. \( \Pi \ll P \)). 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 [14, 19] (Lemma C.30, Lemma C.31).

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

1. \( \forall v \vdash \mu(v) = [\ell, E] \), (a) \( \Pi(\ell) = \top \) and \( \Pi(\top) = \top \)
   (b) \( dom(F) = dom(fields(t)) \) and \( \Pi(dom(F)) \subset \{\top\} \)
   (c) \( \Pi(dom(F)) \subset \{\top\} \)
   (d) \( \forall f \in dom(F) \) s.t. \( F(f) = \ell' \), \( \mu(\ell') = [\ell', \ell'.E'] \) and \( typeOF(F) = [\ell, \top] \Rightarrow \ell' < u \)

2. \( \exists v \in dom(P) \Rightarrow \exists v \in dom(\mu) \)

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

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 ever increasing type environment to type check the expressions.

**Definition C.28.** [Well-typed queues] A queue \( \psi \) is well-typed in \( \Pi \), written \( \Pi \vdash \psi \), if \( \forall v, \tau \in \psi, \Pi \vdash e : (\ell, \rho) \) for some \( t \) and \( \rho \).

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 \( \mu \ll \Pi \).
LEMMA C.30. [Substitution] If \( \Pi, \overline{v}[\overline{w}] \vdash e : (t, p) \) and \( \forall i \in \{1...n\} \), \( \Pi \vdash v_i : (s_i, 0) \) where \( s_i <: t_i \) then \( \Pi \vdash [\overline{v}/\overline{w}] e : (s, p') \) for some \( s <: t \) and some \( p' \).

Proof: To simplify the notations, let \( \Pi' = \Pi, \overline{v}[\overline{w}] \). We prove it by structural induction on the derivation of \( \Pi \vdash e : (t, p) \) 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 \( \var{ var_1,..., var_k } \), then \( \Pi(\text{var}) = \Pi(\text{var}) = t \) and the claim holds. Otherwise, suppose \( var = var_k \). Then \( [\overline{v}/\overline{w}] e = v_k \) and, by the assumptions of the lemma, \( \Pi \vdash [\overline{v}/\overline{w}] e : (s_k, 0) \) and \( s_k <: t_k = t \).

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

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

\begin{itemize}
  \item \textbf{(T-CALL-OPEN)} Here \( e = \textit{this}.f.m(\overline{\sigma}) \). The last type derivation step has the following form:

\[
\begin{align*}
  \Pi' &\vdash e'' : (t, \{\textit{open} f m \Theta\} \cup \bigcup \rho_i) \\
  \Pi'' &\vdash e'' : (\{t, \textit{open} f m \Theta\}) \cup \bigcup \rho_i \\
  \Pi'' &\vdash e'' : (\{t, \textit{open} f m \Theta\}) \cup \bigcup \rho_i \\
  \Pi'' &\vdash e'' : (\{t, \textit{open} f m \Theta\}) \cup \bigcup \rho_i \\
  \Pi'' &\vdash e'' : (\{t, \textit{open} f m \Theta\}) \cup \bigcup \rho_i.
\end{align*}
\end{itemize}

\begin{itemize}
  \item \textbf{(T-SET)} Here \( e = e_1.f = e_2 \). The last type derivation step is:

\[
\begin{align*}
  \Pi' &\vdash e'_2 : (c, p_1) \\
  \Pi'' &\vdash e'_2 : (c, p_1) \\
  \Pi'' &\vdash e'_2 : (c, p_1) \\
  \Pi'' &\vdash e'_2 : (c, p_1) \\
  \Pi'' &\vdash e'_2 : (c, p_1).
\end{align*}
\end{itemize}

\begin{itemize}
  \item \textbf{(T-SEQ)} Here \( e = e_1.f = e_2 \). The last type derivation step has the following form:

\[
\begin{align*}
  \Pi' &\vdash e'' : (t, \{\textit{open} f m \Theta\} \cup \bigcup \rho_i) \\
  \Pi'' &\vdash e'' : (\{t, \textit{open} f m \Theta\}) \cup \bigcup \rho_i \\
  \Pi'' &\vdash e'' : (\{t, \textit{open} f m \Theta\}) \cup \bigcup \rho_i \\
  \Pi'' &\vdash e'' : (\{t, \textit{open} f m \Theta\}) \cup \bigcup \rho_i \\
  \Pi'' &\vdash e'' : (\{t, \textit{open} f m \Theta\}) \cup \bigcup \rho_i.
\end{align*}
\end{itemize}

\begin{itemize}
  \item \textbf{(T-CALL-OPEN-LOC)} Here \( e = \text{loc}.f \). The last type derivation step has the following form:

\[
\begin{align*}
  \Pi' &\vdash e'' : (\{t, \textit{open} f m \Theta\}) \cup \bigcup \rho_i \\
  \Pi'' &\vdash e'' : (\{t, \textit{open} f m \Theta\}) \cup \bigcup \rho_i \\
  \Pi'' &\vdash e'' : (\{t, \textit{open} f m \Theta\}) \cup \bigcup \rho_i \\
  \Pi'' &\vdash e'' : (\{t, \textit{open} f m \Theta\}) \cup \bigcup \rho_i \\
  \Pi'' &\vdash e'' : (\{t, \textit{open} f m \Theta\}) \cup \bigcup \rho_i.
\end{align*}
\end{itemize}

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

\[
\begin{align*}
  \Pi' &\vdash e'' : (t, \{t, \textit{open} f m \Theta\}) \cup \bigcup \rho_i \\
  \Pi'' &\vdash e'' : (t, \{t, \textit{open} f m \Theta\}) \cup \bigcup \rho_i \\
  \Pi'' &\vdash e'' : (t, \{t, \textit{open} f m \Theta\}) \cup \bigcup \rho_i \\
  \Pi'' &\vdash e'' : (t, \{t, \textit{open} f m \Theta\}) \cup \bigcup \rho_i \\
  \Pi'' &\vdash e'' : (t, \{t, \textit{open} f m \Theta\}) \cup \bigcup \rho_i.
\end{align*}
\end{itemize}

\begin{itemize}
  \item \textbf{(T-SEQ-OPEN)} Here \( e = e_1.f = e_2 \). The last step is:

\[
\begin{align*}
  \Pi' &\vdash e'' : (t, \{t, \textit{open} f m \Theta\}) \cup \bigcup \rho_i \\
  \Pi'' &\vdash e'' : (t, \{t, \textit{open} f m \Theta\}) \cup \bigcup \rho_i \\
  \Pi'' &\vdash e'' : (t, \{t, \textit{open} f m \Theta\}) \cup \bigcup \rho_i \\
  \Pi'' &\vdash e'' : (t, \{t, \textit{open} f m \Theta\}) \cup \bigcup \rho_i \\
  \Pi'' &\vdash e'' : (t, \{t, \textit{open} f m \Theta\}) \cup \bigcup \rho_i.
\end{align*}
\end{itemize}
and (T-NULL), the type environment does not appear in the hypotheses of the judgment, so the claim holds. For the (T-VAR) case, \(e = \text{var} \) and \(\Pi(\text{var}) = t\). But \(a \notin \text{dom}(\Pi)\), so \(\varnothing \neq a\). Therefore (\(\Pi, a : t') (\text{var}) = t\) 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 \(\Sigma = ((E[e], (id, l)) + \psi, \mu)\), \(\Sigma' = ((E[e'], (id, l')) + \psi', \mu')\) \(\Sigma \Rightarrow \Sigma', \Pi \vdash E[e] : (t, \rho)\), \(\Pi \vdash e : (t', \rho')\) and \(\Pi' \vdash e' : (t', \rho''_0)\), then \(\Pi' \vdash E[e'] : (t, \rho_0)\), for some \(\rho_0\).

**Proof:** Proof by induction on the size of the evaluation context \(E[e]\). 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, \(E_0 = \text{--}\), and \(t' = u' \prec u = t\). For the induction step we divide the evaluation context into two parts such that \(E[e_1] = E_1[E_2[e_2]]\), 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 \((\text{loc}, f = \text{--}), (-; e_2)\) follow directly from the induction hypothesis.

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

\(\text{T-CALL-OPEN}():\)

\[
\begin{align*}
(c_1, t, m \vdash (\text{var}) & : e_{n+1}, \rho''_0) = \text{findMeth}(u, m) \\
\Pi & \vdash e : (u, \rho_0) \\
\forall i \in \{1..n\} : \Pi' & \vdash e : (t'_i, \rho''_i) & \wedge & t'_i < : t_i \\
\Pi' & \vdash E_2[e] : (t, \{x_i\})
\end{align*}
\]

Here \(\text{findMeth}(u, m) = (c_2, t, m \vdash (\text{var}) : e_{n+1}, \rho''_0)\), by the definitions of \(\text{override}\) and \(\text{findMeth}\), where \(c_2 < c_1\), so (T-CALL) gives \(\Pi' \vdash E_2[e'] : (t', \rho')\); or

\(\text{T-CALL-OPEN}():\)

\[
\begin{align*}
& e = \text{this}, f & \text{typeOf}(f) = (d, \text{open} t) \\
& (c_1, t, m \vdash (\text{var}) & : e_{n+1}, \rho''_0) = \text{findMeth}(t', m) \\
\Pi' & \vdash e : (t', \rho') \\
\forall i \in \{1..n\} : \Pi' & \vdash e : (t'_i, \rho''_i) & \wedge & t'_i < : t_i \\
\Pi' & \vdash E_2[e] : (t, \{\text{open } f m \theta\} \cup & \rho' \cup & \bigcup_{i=1}^{n} \rho''_i)
\end{align*}
\]

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

**Case \(\varnothing.m(\overline{x})\)** The last step in the type derivation for \(E_2[e]\) must be (T-CALL) :

\[
\begin{align*}
\Pi & \vdash v : (u, \theta) \\
(\forall i \in \{1..(p-1)\} : \Pi & \vdash v : (t'_i, \theta) & \wedge & t'_i < : t_i) \\
\forall j \in \{p..n\} : \Pi & \vdash e : (t'_j, \rho''_j) & \wedge & t'_j < : t_j \\
\Pi' & \vdash e : (t', \rho') & \wedge & t' < : t
\end{align*}
\]

\(\Pi' \vdash E_2[e'] : (t, \{x_i\})\)

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

**Case \(-f = e_2\)** The last step for \(E_2[e]\) must be

\(\text{T-SET}():\)

\[
\begin{align*}
\Pi & \vdash e : (c, \rho_1) & \text{typeOf}(f) = (d, t) \\
\Pi & \vdash e : (t_2, \rho_2) & \vdash t_2 < : t \\
\Pi' & \vdash E_2[e'] : (t_2, \{\text{write } f\})
\end{align*}
\]

By the definition of field lookup, \(\text{typeOf}(f)\) does not change. Thus, by (T-SET), \(\Pi' \vdash E_2[e'] : (t_2, \rho_0)\); or

\(\text{T-SET-OPEN}():\)

\[
\begin{align*}
e & \equiv \text{this} & \Pi(\text{this}) : t' & \text{typeOf}(f) = (d, \text{open} t) \\
\Pi' & \vdash e : (t_2, \rho_2) & \vdash t_2 < : t \\
\Pi' & \vdash E_2[e'] : (t_2, \{\text{read } f\})
\end{align*}
\]

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

**Case \(-f\)** The last step for \(E_2[e]\) could be

\(\text{T-GET}():\)

\[
\begin{align*}
\Pi & \vdash e : (c, \rho_1) & \text{typeOf}(f) = (d, \text{open} t) \\
\Pi & \vdash e \vdash (t_2, \rho_2) & \vdash t_2 < : t \\
\Pi' & \vdash E_2[e'] : (t_2, \{\text{read } f\})
\end{align*}
\]

The result of \(\text{typeOf}\) does not change. Thus, by (T-GET), \(\Pi' \vdash E_2[e'] : (t, \rho_0)\); or

\(\text{T-GET-OPEN}():\)

\[
\begin{align*}
e & \equiv \text{this} & \Pi(\text{this}) : t' & \text{typeOf}(f) = (d, \text{open} t) \\
\Pi & \vdash e \vdash (t, \rho_1) & \vdash t < : u \\
\Pi' & \vdash E_2[e'] : (t, \{\text{read } f\})
\end{align*}
\]

**Lemma C.33. [Replacement with subtyping]** If \(\Sigma \Rightarrow \Sigma', \Sigma = ((E[e], (id, l)) + \psi, \mu)\), \(\Sigma' = ((E[e'], (id, l')) + \psi', \mu')\), \(\Pi \vdash E[e] : (t, \rho)\), \(\Pi' \vdash e : (u, \rho_0)\), and \(\Pi' \vdash e : (u', \rho_1)\) and \(u < : u\), then \(\Pi' \vdash E'[e'] : (t', \rho'_{i})\) where \(t' < : t\).

**Proof:** Proof by induction on the size of the evaluation context \(E[e]\). 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, \(E_0 = \text{--}\), and \(t' = u' \prec u = t\). For the induction step we divide the evaluation context into two parts such that \(E[e_1] = E_1[E_2[e_2]]\), 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 \(s' < : s\) and the claim holds by IH. The cases for \((\text{loc}, f = \text{--}), (-; e_2)\) and \((c \varnothing \vdash e_2)\) follow directly from IH.

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

5 Formulation of the proof is similar to Clifton’s work [14] and Flatt’s work [19]
### Case \(-m(\overline{\pi})\)

The last step for \(E_2[e]\) could be

\[
\begin{align*}
\Pi \vdash e : (t', \rho') & \quad \text{typeOff}(f) = (d, \emptyset) \\
(c_1, t, m, (T\overline{\var}) (e_{n+1}), \rho_1) & = \text{findMeth}(t', m) \\
(\forall i \in [1..n] :: \Pi \vdash e : (t_i', \rho_i') \land t_i' < t_t) & \\
\Pi \vdash E_2[e] : (t, \{\})
\end{align*}
\]

We have \(\text{findMeth}(t', m) = (c_2, t, m, (T\overline{\var}) (e_{n+1}'), \rho_1)\), by the definitions of \(\text{override}\) and \(\text{findMeth}\), \(c_2 < c_1\), so (T-CALL) gives \(\Pi \vdash E_2[e] : (t, \{\})\); or

\[
\begin{align*}
\Pi \vdash E_2[e] : (t, \{\})
\end{align*}
\]

### Case \(-f\)

The last step for \(E_2[e]\) could be

\[
\begin{align*}
\Pi \vdash e : (c, \rho_1) & \quad \text{typeOff}(f) = (d, t) \quad c < : d \\
\Pi \vdash E_2[e] : (t, \{\text{read } f\})
\end{align*}
\]

The result of \(\text{typeOff}\) does not change. Thus, by (T-GET), \(\Pi \vdash E_2[e'] : (t', \rho_0)\); or

\[
\begin{align*}
\Pi(e) = c \quad \text{typeOff}(f) = (d, \text{open } t) \quad c < : d \\
\Pi \vdash E_2[e] : (t, \{\text{read } f\})
\end{align*}
\]

Here \(e'\) must be \(\text{loc}\), for some \(\text{loc}\). By the statement \(\Pi \vdash \text{loc} : (e', \emptyset)\), and \(c' < c\). The result of \(\text{typeOff}\) does not change. Thus, by (T-GET-OPEN-LOC), \(\Pi \vdash E_2[e'] : (t, \{\text{read } f\})\).

### THEOREM C.34

[Type preservation] If \(\Pi \vdash \Sigma\), where \(\Sigma = \langle(e, (id, I)) + \psi, \mu\rangle\), \(\Sigma \vdash (e', (id, I')) + \psi', \mu'\), and \(\Pi \vdash e : (t, \rho)\), then there is some \(\Pi'\), and \(\rho'\) such that

\[
\begin{align*}
(\mu' \approx \Pi') \land (\Pi' \approx \psi') & . \text{i.e. } \Pi' \approx \Sigma' \\
(\mu' \approx \Pi') \land (\Pi' \approx \psi') & ; \\
\Pi' \approx \Pi & ; \text{and} \\
\Pi' \vdash e' : (t', \rho') \land (t' < t).
\end{align*}
\]

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, so by Lemma [C.31] (Environment extension) and Definition [G.28], if \(\Pi \vdash \psi\), then \(\Pi' \vdash \psi'\).

### New Object

Here \(e = E[\text{new } c()]\) and \(e' = E[\text{yield } \var]\), where \(\text{loc } \notin \text{dom } \mu\), and \(\mu' = \langle\text{loc} - \langle c, f \to \text{null } \text{fields}(c), \langle(m \to \rho) \in \text{meth}(c)\rangle\rangle\rangle \approx \mu\). Let \(\Pi' = \Pi, \text{loc} = \mu\). We now show that \(\Pi' \approx \mu\). Because \(\text{loc } \notin \text{dom } \mu\), \(\Pi' \approx \mu\) holds vacuously. So part 1 of \(\Pi' \approx \mu\) holds. Part 2 holds because \(\Pi = \mu, \text{loc } \in \text{dom } \Pi\), \(\text{loc } \notin \text{dom } \mu\). By Lemma [C.32] (Environment extension) and \(\text{loc } \notin \text{dom } \Pi\), we have \(\Pi' \vdash E[\text{new } c()] : (t, \rho). \Pi' \vdash E[\text{yield } \var] : (c, \rho)\), so by Lemma [C.32], \(\Pi' \vdash E[\text{yield } \var] : (t, \rho)\).

### Field Get

In this case \(e = E[\text{loc } f()]\), \(e' = E[\text{yield } \var]\) where \(\mu(\text{loc }) = [u,F.E]\) and \(F(f) = v\), and \(\mu' = \mu\). Let \(\Pi = \Pi\). Clearly \(\Pi' \approx \mu'\), and \(\Pi' \approx \Pi\).

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

### Field Set

Here \(e = E[\text{loc } f = v], e' = E[\text{yield } \var], j_0 = \mu \ni (\text{loc} \ni [u,F.E] \ni (f \to \text{null } v)), \mu' = \text{update}(\mu_0, \text{loc}, f, v),\) and \(\mu(\text{loc }) = [u,F.E]\). Let \(\Pi = \Pi, \text{loc } \in \Pi'\). We now show that \(\Pi' \approx \mu'\). Observe that the update function changes the effect mapping \(E\) in each of the object record, but not the fields \(F\), which have no impact on \(\Pi \approx \mu'\), by Definition [C.26]. Here \(\mu'(\text{loc }) = [c, F \ni (f \to \text{v}').E'\rangle,\) for some \(E'\). For part (a) \(\Pi(\text{loc }) = u, \mu(\text{loc }) = [u,F.E]\) and \(\Pi = \mu\). For part (b) \(\text{dom}(F \ni (f \to \text{v}')) = \text{dom}(\text{fields}(u)),\) since \(\text{loc } f = v\) is
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 \(\Pi \vdash \mu\), then \(v \in \text{dom}(\mu)\). \(v \in \text{dom}(\mu)\) implies \(v \in \text{dom}(\mu')\). In either case \(\text{rng}(F) \cup \{v\} \subseteq \text{dom}(\mu') \cup \{\text{null}\}\). Part 1(d) holds for all \(f \in \text{dom}(F)\), \(f' = f\). Part 1(d) holds vacuously for \(f = \text{null}\). Otherwise, \((F \oplus (f \mapsto v))(f) = v\), and by \((\text{T-Set})\) or \((\text{T-Set-open-loc})\) and \((\text{T-Loc})\), \(\Pi(\psi) < x'\), where \(\text{fields}(\mu) = (c, x')\) and \(c < t\). Parts 2 holds since \(\text{dom}(\mu') = \text{dom}(\mu)\).

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

**Method Call** Here \(e = \text{E}[\text{loc}.m(\bar{v})], e' = \text{E}[\text{yield } e_1], \mu(\bar{v}) = \{u,F,E\}, (\text{findMeth}(u,m) = (u', i, m(\bar{v})), \rho_0\), \(\mu' = \mu\) and \(e_1 = \lfloor \text{this}, \lfloor \text{var} \rfloor e_2 \rfloor\). Let \(\Pi' = \Pi\). Clearly \(\Pi' \approx \mu'\), and \(\Pi' \vdash \Pi'\).

We now show that \(\Pi' \vdash e' : (t', \rho')\) for some \(t' < t\) and some \(\rho'\). \(\Pi \vdash e : (t, \rho)\) implies that \(\text{loc}.m(\bar{v})\) and all its subterms are well-typed in \(\Pi\). By part 1(a) of \(\Pi \approx \mu\), \(\Pi \vdash \text{loc}.m : (u, \emptyset)\). By the definition of \(\text{findMeth}\), \(u : < u'\). Let \(\Pi' \vdash v : (u, \emptyset)\) \(\forall i \in \{1..n\}\) and let \(\Pi' \vdash \text{loc}.m(\bar{v}) : (u_m, \rho_m)\). This last judgment must be \((\text{T-Call}),\) with \(\{u', u_m, m(\bar{v})\} = \text{findMeth}(u, m)\), \(\forall i \in \{1..n\}\) :: \(u_i < t_m\). By the definition of the function \(\text{findMeth}\), rules \((\text{T-Method})\) and \(\text{override}\), \(\{\text{var}:\text{this}:u'\} \vdash e_2 : (u_m, \rho_1)\), and \(u_m < t_m\). By Lemma \(\text{C.31} \text{(Environment extension)}\) and \(\text{(appropriate alpha conversion of free variables in } e_2)\), \(\Pi', \text{var}:\text{this}:u' \vdash u' : (u_m, \rho_1)\).

By Lemma \(\text{C.30} \text{(Substitution)}\), \(\Pi \vdash \text{loc}.m(\bar{v}) \vdash e_2 : (u', \rho_1)\), for some \(u' < u_m < t_m\). Finally, Lemma \(\text{C.33} \text{(Replacement with subtyping)}\) gives \(\Pi' \vdash e' : (t', \rho')\) for some \(t' < t\).

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

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

We show that \(\Pi' \vdash e' : (t', \rho')\) for some \(t' < t\) and some \(\rho'\). By \((\text{T-Fork})\), \(\Pi \vdash \text{fork}(e_2; e_3) : (\text{void}, \rho_0)\). Because the expression is well typed, \(e_2\) and \(e_3\) are well-typed. We know \(\Pi \vdash \text{null} : (\text{void}, \emptyset)\), so by \((\text{T-Seq})\), \(\Pi \vdash \text{yield } e_0 : (u, \rho_1)\), for any valid type \(u\) and some \(\rho_1\). Since the type of \(\text{null}\) is subtype of any type, Lemma \(\text{C.33} \text{(Replacement with subtyping)}\) 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, \(e = \text{E}[\text{yield } e_1], e' = \text{E}[e_1]\). There are two cases: (a) there is no reduction step happens for other tasks during this reduction; (b) there are reduction steps happen for other tasks during this reduction.

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