Open Effects
Yuheng Long and Hridesh Rajan
TR #12-02
Initial Submission: October 17, 2012

Abstract: This is the Technical Report version of the 2013 ESOP submission by the same title. It includes the ESOP 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
# Table of Contents

- **Open Effects**
  - Yuheng Long and Hridesh Rajan
  - 1 Introduction ......................................................... 1
    - 1.1 Contributions to the State-of-the-Art: Open Effects .............. 3
  - 2 An Object-oriented Calculus with Open Effects .......................... 6
  - 3 A Type-and-Effect System with Open Effects ............................ 6
    - 3.1 Notations and Conventions .................................................. 7
    - 3.2 Type-and-Effect Rules for Method Declaration ......................... 7
    - 3.3 Open Effects of Polymorphic Method calls ........................... 8
    - 3.4 Type-and-effects for Field Related Expressions ...................... 9
  - 4 A Dynamic Semantics with Open Effects ................................... 9
    - 4.1 Notations and Conventions .................................................. 9
    - 4.2 Dynamic Effect Management in OO Expressions ......................... 10
    - 4.3 Soundness: Type and Effect Preservation ............................ 12
  - 5 Adding Open Effects to the OpenJDK Java Compiler ......................... 13
  - 6 Using Open Effects for Safe Concurrency ................................ 14
    - 6.1 Checking (non)interference of concurrent tasks using open effects . 14
    - 6.2 Parallelizing representative libraries ................................ 15
    - 6.3 Performance Evaluation .................................................. 15
    - 6.4 Discussion: Scope and Applicability of Open Effects .............. 17
  - 7 Comparative Analysis with Related Work ................................ 17
    - 7.1 Comparison with ideas related to open effects based concurrency .. 18
      - Overview of related ideas .................................................. 18
      - Criteria and results ...................................................... 18
  - 8 Conclusion and Future Work ............................................... 19
  - References ................................................................. 19
  - 9 Type-and-effect System: Omitted Details .................................. 21
    - 9.1 Type-and-Effect Rules for Declarations .............................. 21
    - 9.2 Type-and-Effect Rules for Expressions .............................. 21
  - 10 Dynamic Semantics: Omitted Details ..................................... 22
  - 11 Proof of Key Properties .................................................. 23
    - 11.1 Preliminary Definitions .................................................. 23
    - 11.2 Effect Preservation .......................................................... 25
      - Replacement with Subeffect ................................................. 26
      - Substituting Variables with Values ...................................... 29
      - Fields Access ............................................................. 30
    - 11.3 Type Soundness ............................................................. 31
Open Effects

Yuheng Long and Hridesh Rajan

Iowa State University, Ames, Iowa, USA
{csgzlong,hridesh}@iastate.edu

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

1 Introduction

A type-and-effect system \cite{20,29} is a valuable tool for programmers, e.g. it can help analyze locking disciplines \cite{2}, detect race conditions \cite{18}, analyze checked exceptions \cite{27,4}, analyze dynamic updating mechanisms \cite{31}, etc. In essence, a type-and-effect system adds an encoding of computational effects into a language’s set of semantic objects and a discipline for controlling these effects into the language’s type system \cite{40,39}. Typically, these computational effects describe how the state of the program will be modified by expressions in the language. For example, a field read expression may have a read effect to represent reading from a memory region or a field write expression may have a write effect to represent writing to a memory region \cite{40}.

Object-oriented features such as dynamic dispatch presents a challenge for such type-and-effect system — these features require conservative handling for soundness\cite[pp.222]{18}. To illustrate, consider a library with classes ArrayList and Command in Figure 1. Two applications, PROGRAM and PROGRAM’ (among others) use this library: each provides a separate and distinct subclass of the Command class. PROGRAM computes a prefix sum, whereas PROGRAM’ computes a hash. Computing hash is implemented as an independent operation for each array element, whereas prefix sum is not.
Now imagine that either the programmer or an analysis is using a type-and-effect system to help with parallelization of the method `applyall`, specifically the for loop on lines 5-6. The type-and-effect system would be used to compute the effects of each loop iteration. If the effect of each iteration of `run` on line 6 does not interfere with any other iteration, then this for loop may be safely parallelized. However, the exact runtime type of the parameter `c` is unknown statically, so the effects of `run` are unknown also.

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

```
9 class Command {
10     void run(int index, int o) { }
11 }
12}
```

```
13 class Prefix extends Command {
14     int sum = 0;
15     int[] eles;
16     void run(int i, int o){
17         sum += o; // conflicts on sum
18         eles[i] = sum; // commutative
19     }
20 }
```

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

**Fig. 1.** A library class `ArrayList` and two separate applications that make use of it.

To overcome this hurdle, a type-and-effect system could compute the sets of effects, e.g. `{reads field f}`, produced by all overriding implementations of the `run` method and take the set of potential effects of `run` to be the union of these sets. However, this solution would not work for libraries and frameworks in modern object-oriented languages, where many such overriding implementations, such as those in `PROGRAM` and `PROGRAM'`, may not be available during the compilation of the library class `ArrayList`.

An alternative solution is to ask programmers to annotate the `run` method’s implementation in the class `Command` and use these effect annotations as an upper bound on the potential effects produced by all overriding implementations of that method. However, computing such upper bound can be difficult for the programmer primarily due to the variety of, and often unanticipated, usage of library classes such as `ArrayList`. Even if the programmer is able to anticipate all such usage and compute an upper bound, such a bound may turn out to be overly conservative. For example, based on the computational effects of the method `run` in `PROGRAM`, one may conclude that the parallelization of the `applyall` method would be unsafe. Whereas, in reality there may be several subclasses of `Command`, such as class `Hash` in `PROGRAM'`, whose computational effects will permit safe parallelization of the `applyall` method.

Computing effects at runtime can help \[81617137\]. However, analyses based entirely on dynamic effect computation are reported to be expensive [17, Table 1] and often do not provide preventative detection and defect avoidance [26, pp. 6:2].
1.1 Contributions to the State-of-the-Art: Open Effects

A promising idea, in the spirit of hybrid type checking [26], is for the programmer to optimistically assert that method calls, with certain special references as receivers, will produce safe effects, and the compiler to trust the programmer statically, but also emit code to verify programmer’s assertion at runtime. We present a new optimistic effect system that takes this idea and blends static effect computation with dynamic effect verification, producing a system that has many of the advantages of both the static and dynamic effect systems, but suffers from none of their 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 statically, but it is filled in at runtime. Concrete effects include reads and writes to memory regions [40]. Like static type-and-effect system, we compute effects for each expression. However, unlike static approaches that make conservative approximations when the exact dynamic type is unavailable, we use placeholder open effects.

To illustrate, imagine that the programmer optimistically marked the argument c’s type in the class ArrayList as open as on line 4 in Figure 2. Our system 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 the 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.

```java
class ArrayList {
    int[] elements;
    int size;
    void applyall(@open Command c) {
        for (int i : size)
            c.run(i, elements[i]); // open effect
    }
}
```

Fig. 2. Modified ArrayList with @open on line 4. Applications remain the same.

The dynamic part of our analysis fills in, or concretizes, open effects when references marked with @open annotations, such as c, are assigned. We illustrate below.

```java
// PROGRAM: snippets from the main method.
ArrayList al = new ArrayList();
// add elements to the list al
Command p = new Prefix();
al.applyall(p);
```

Assume that the class ArrayList is already compiled. 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 5 in the listing above 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.
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 a Prefix instance 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 17 in Figure 1).

**Open effects for concurrency.** Like dynamic approaches, a parallelization technique based on open effects may treat each parallelization opportunity as optimistically parallel, if concrete effects of parallel tasks do not interfere. An open effect is treated as a blank effect statically, so an iteration of the for loop in method applyall would be treated (trustingly) as parallel since it is independent of other iterations. However, unlike dynamic approaches that detect conflicts after-the-fact, we require verifying that open effects are noninterfering prior to forking off parallel tasks. This could be done by inserting a runtime check before the parallel section as we show in Section 5. 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.

So, a parallelization technique based on open effects would run the for loop on line 5 sequentially in PROGRAM since it has a loop carried dependence.

On yet another day, the developer of PROGRAM' imports the ArrayList class. PROGRAM' also instantiates ArrayList, but passes an instance of the 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 could run in parallel in PROGRAM'.

```
// PROGRAM*: snippets from main method.
Command p = new Prefix();
> al.applyall(p); // Sequential

Command h = new Hash();
> al.applyall(h); // Parallel
```

Consider another program PROGRAM" that computes both prefix sum and hash of the same list. This program would reap the best of both worlds, prefix sum would be computed safely in serial, whereas has computation would happen in parallel.

Thus, our optimistic type-and-effect system can help expose safe concurrency in these scenarios that are typically challenging for a purely static type-and-effect system analyzing libraries and programs separately. Our type-and-effect system also eases the programmers’ task because it does not ask them for effect annotations.

Compared to a purely dynamic effect system that monitors memory accesses by concurrent tasks, which is then be used to detect conflicts between tasks; our type-and-effect system monitors references marked as @open and updates open effects when these references change. This could then be used to check conflicts before running parallel tasks. So programmers have greater control over which references are monitored.

**Open-effects and open-world assumption.** Languages like Java and C# incorporate the open world assumption in several of their design decisions, e.g. separate compilation, dynamic class loading. Open effects integrate well with this assumption in language design, e.g., the static effect computation for the library, PROGRAM, and PROGRAM' in our example, can proceed independently. Since statically computed effects
are composed at runtime, open effects also works well with dynamic class loading, with
the proviso that all classes provide statically computed effects.

Our open-effects based type-and-effect system also has the following benefits.

– It is modular and so it enables analysis of libraries and frameworks, which is impor-
tant 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, effect computation for ArrayList relies only on the
code for ArrayList and the interface of the Command classes, but not neces-
ecessarily on its implementation. This would be essential for analyzing ArrayList
without requiring PROGRAM or PROGRAM’ to also be present. This benefit is
critical for libraries, which are analyzed and compiled once, but reused often.

– For our use cases, it had a small annotation overhead, e.g. one annotation was
needed in the ArrayList. A majority of this benefit arises from the treatment of
dynamic dispatch, which does not require annotating supertype methods to give
upper bounds on the effect of all subtypes, e.g. the run method in type Command
(Figure 2, line 10). Also, 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 system, but would have some runtime
overhead. Our evaluation shows that these overheads are negligible. For exam-
ple, our effect system was able to distinguish between effects of the method call
run in PROGRAM (with Prefix class) and PROGRAM’ (with Hash class), de-
signed by two different programmers at two different times. This could allow 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 would not suffer significantly
also. Since conflicts are detected preemptively, so no rollback mechanism would be
required. Rather an operation would be attempted in parallel if and only if the open
effect assertions hold, which is useful for preventative detection and avoidance.

These benefits make open effects an interesting point in the design space between
fully static and fully dynamic effect systems. Since the annotation @open is explicit,
programmers can control the optimism in effects and the dynamic overhead.

In summary, main contributions of this work are:

– a language design with open effects;
– a type-and-effect semantics with open effects in Section 5, where the novelty lies
  in the integration of the open effects with standard effects;
– a dynamic semantics with open effect concretization in Section 4;
– a proof of soundness of an open-effects based type-and-effect system, which is
  challenging compared to the static effect systems, because the effect of a method
call could change at runtime, due to the open effects;
– a prototype Java compiler based on the OpenJDK in Section 5 that uses efficient
  effect storage and retrieval strategies to yield a low-overhead hybrid effect-system;
– an application of open effects for (non) interference analysis of concurrent tasks;
– an evaluation that uses several canonical programs in Section 6 and demonstrates
  that open effects has benefits, acceptable overhead, and low annotation cost; and
– a comparative analysis with related ideas in Section 7.
2 An Object-oriented Calculus with Open Effects

This section introduces OpenEffectJ, a minimal expression language based on Classic Java [19]. The grammar is shown in Figure 3. The grammar includes an interim expression for semantics: loc that represents locations. The notation over-bar denotes a finite ordered sequence (\( \overline{a} \) stands for \( a_1 \ldots a_n \)). The notation [a] means that a is optional.

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

Fig. 3. The Grammar for OpenEffectJ.

A programmer writing code for reusable classes, e.g. the class ArrayList, typically knows that a reference such as \( c \) in that class may point to concrete objects of different types at runtime, and if a method is called on such reference, it may result in different effects. They can annotate these references as \text{@open}, e.g. line 4 in Figure 2.

To simplify presentation, only fields can be annotated \text{@open}, but not other references, e.g., variables and parameters. In Section 4, we discuss how we could handle other references. In the following, we will refer to these annotated fields as open fields.

3 A Type-and-Effect System with Open Effects

We now describe a hybrid (static/dynamic) type-and-effect system for OpenEffectJ. The main novelty of this type-and-effect system is the notion of open effects. An open effect is a special placeholder effect. The static part of this system computes effects of every method, and these computed effects may contain open effects. The statically computed effects of a method contain an open effect, if the method’s body contains a method call expression with a field as receiver object whose type is annotated with \text{@open}. For an analysis of an effect-based property this signifies that an optimistic assumption should be made that this method call’s computational effects will satisfy that property, and that assumption should be verified by the dynamic part of the type-and-effect system.

The dynamic part of our type-and-effect system has two roles. First, it leverages the statically computed effect information to maintain up-to-date dynamically computed effects of a method. The dynamically computed effects of a method may contain open effects. These open effects may change as more information about their receiver objects becomes available. For example, a previously unknown field may become known as a result of a field set. Second role is to verify optimistic assumptions made statically about open effects using the more precise, dynamically computed effect information.
3.1 Notations and Conventions

The type-and-effect system uses domains defined in Figure 4 that are based on previous work on effects [40, 39, 22]. The type of a program and declarations are given as OK. A method’s type specifies the argument and result types, and the latent effects $\sigma$. An expression’s type attribute is given as $(t, \sigma)$, the type $t$ of an expression and its effects $\sigma$. We use the term effects to refer to a set of read/write effects, open effects, and bottom effect. The read and write effects contain the name of the field that is being read and written. In the dynamic semantics, the field name and the identity of object that contains this effect suffices to identify which object’s field is modified. Previous work on object-oriented effect systems, e.g., Greenhouse and Boyland [22], uses regions as an abstraction to avoid exposing implementation details in specifications. Since in our type-and-effect system there is no explicit specification, that concern doesn’t arise. In the formal core, we do not track objects but our compiler in Section 5 has it.

An open effect contains the name of the open field, the method $m$ being invoked, and a placeholder for concrete effects $\sigma$. The placeholder $\sigma$ is used by the dynamic part of our system. At runtime, the dynamic part concretizes the open effects by filling in $\sigma$ with actual effects. These concretization points happen whenever $f$ is set.

$$\theta ::= \text{OK} \quad \text{“program/decl types”} \quad \sigma ::= \emptyset \cup \sigma \cup \{ \bot \} \quad \text{“program effects”}$$

$$\{ (\rightarrow t, \sigma) \} \quad \text{“method types”} \quad \{ \{\text{read } f\} \} \quad \text{“read effect”}$$

$$\{ (t, \sigma) \} \quad \text{“expression types”} \quad \{ \{\text{write } f\} \} \quad \text{“write effect”}$$

$$\Pi ::= \{ \text{var} \mapsto t \} \in \mathbb{N} \quad \text{“type environments”} \quad \{ \{\text{open } f m \sigma\} \} \quad \text{“open effect”}$$

Fig. 4. Domains of types and effects in our type-and-effect system.

![Fig. 4. Domains of types and effects in our type-and-effect system.](image)

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

The rules for top-level declarations, object creation, variable reference and declaration and null reference are standard (Section 9.1 contains these rules).

3.2 Type-and-Effect Rules for Method Declaration

Main novelty is in the rules for method declaration and method calls. In our type-and-effect system an overriding method is allowed to have different effects compared to the overridden method. This improves flexibility in the usage, especially for libraries and frameworks, where it is a common practice to define empty abstract methods in a superclass that are overridden by the client to implement application-specific functionality.

The $(T\text{-METHOD})$ 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 $\sigma$. This rule uses a function $\text{override}$. The function $\text{findMeth}$ (used by $\text{override}$) looks up the method $m$, starting from the class $c$, looking in superclasses if necessary.
3.3 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. For effects we distinguish based on the kind of the receiver object. We first discuss the pessimistic case, in which the receiver of the call is not an open field.

\[(T\text{-}\textbf{METH})\]
\[
\begin{aligned}
\forall i \in \{1..n\} \ s.t. \ \text{meth}_i &= (t, \sigma, m(\bar{\text{var}})) \\
\text{findMeth}(c, m) &= (c, t, m(\bar{\text{var}})) (\sigma)
\end{aligned}
\]

Here, 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, that is similar to saying that the method writes everything \[22,33\]. As Section 5 discusses, if the receiver’s exact type is known, this effect can be made more precise.

The optimistic case \((T\text{-}\textbf{CALL})\) applies when a method \(m\) is called with an open field \(f\) as its receiver object (this can be extended to local variables aliases of \(f\) \[21\]).

\[(T\text{-}\textbf{CALL})\]
\[
\begin{aligned}
e_0 &\neq \text{this}.f \lor (e_0 = \text{this}.f \land \text{typeOf}(f) \neq (c, \emptyset)) \\
\text{findMeth}(c_0, m) &= (c_1, t, m(\bar{\text{var}})) (\sigma_0)
\end{aligned}
\]

In this case, statically we assume that this method call will have no effect (represented by \(\emptyset\) in \(\text{open} \ f \ m \ \emptyset\)). To illustrate the implication of this assumption, let \(\Phi\) be an effect-based property, \(e\) an expression and \(\Pi \vdash e : (t, \sigma)\). Let \(\sigma' \subseteq \sigma\) such that all read and write effects in \(\sigma\) are in \(\sigma'\) and no open effects in \(\sigma\) are in \(\sigma'\). If \(\sigma'\) entails \(\Phi\) then \(\sigma\) is also assumed to entail \(\Phi\), provided that dynamic part of our type-and-effect system verifies that concretized open effects entail \(\Phi\).

This difference is the main benefit of our approach. Unlike purely static effect systems, we defer some effect computation to runtime. Unlike purely dynamic effect systems, we defer only programmer-selected effect computations to runtime.
3.4 Type-and-effects for Field Related Expressions

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

\[
\begin{align*}
\text{(T-GET)} & \quad \Pi \vdash \textbf{this}: c \\
\text{typeOfF}(f) & = (d, \texttt{@open} t) \\
\Pi \vdash \textbf{this}.f : (t, \{\texttt{read } f\})
\end{align*}
\]

\[
\begin{align*}
\text{(T-SET)} & \quad \Pi \vdash \textbf{this}: c \\
\text{typeOfF}(f) & = (d, t') \\
\Pi \vdash e : (t, \sigma) \\
\Pi \vdash \textbf{this}.f = e : (t, \{\texttt{write } f\})
\end{align*}
\]

\[
\begin{align*}
\text{(T-SET-OPEN)} & \quad \Pi \vdash \textbf{this}: c \\
\text{typeOfF}(f) & = (d, \texttt{@open} t') \\
\Pi \vdash e : (t, \sigma') \\
\Pi \vdash \textbf{this}.f = e : (t, \{\bot\})
\end{align*}
\]

As we show in the next section, an open field set expression represents a program location where concrete effects in some open effects may change. Our type-and-effect gives this expression a bottom effect to maintain soundness; however, in cases where a field assignment does not change the concrete effects, the rule (T-SET) can be applied.

4 A Dynamic Semantics with Open Effects

We now give a small-step operational semantics for OpenEffectJ. To the best of our knowledge, this is the first integration of a hybrid effect system with an OO semantics.

4.1 Notations and Conventions

The small steps taken in the semantics are defined as transitions from one configuration (Σ in Figure 5) to another. Some rules use an implicit attribute, the class table CT.

Evaluation relation: \( \vdash : \Sigma \rightarrow \Sigma \)

Evaluation contexts:

\[
\begin{align*}
\Sigma & ::= (e, \mu) & \text{“Program Configurations”} \\
\mu & ::= \{\text{loc} \mapsto o_i\}_{i \in \mathbb{N}} & \text{“Store”} \\
o & ::= [c.F.E] & \text{“Object Records”} \\
F & ::= \{f_i \mapsto v_i\}_{i \in \mathbb{N}} & \text{“Field Maps”} \\
v & ::= \text{null} | \text{loc} & \text{“Values”} \\
E & ::= \{m_i \mapsto \sigma_i\}_{i \in \mathbb{N}} & \text{“Effect Maps”} \\
\end{align*}
\]

\[
\begin{align*}
\text{E} & ::= - | E.m(\tau) \\
v.m(\tau, E, \tau) & | v.f=e \\
\text{E.f} & | \text{E.f} \\
t \text{var}=E; e &
\end{align*}
\]

Fig. 5. Domains used in the dynamic semantics of OpenEffectJ.

A configuration consists of an expression e 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.
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 \( \Sigma_v = ⟨e, ◦⟩ \). The operator \( \oplus \) is an overriding operator for finite functions, i.e. if \( µ′ = µ \oplus \{loc → o\} \), then \( µ′(loc') = o \) if \( loc' = loc \), otherwise \( µ′(loc') = µ(loc') \).

### 4.2 Dynamic Effect Management in OO Expressions

The rules for OO expressions are shown below (Section [10] contains omitted auxiliary functions). The novelty is that some of the rules manipulate the effect map \( E \).

\[
\text{(NEW)}\quad \begin{align*}
\text{loc} & \notin \text{dom}(µ) \quad F = \{f → \text{null} \mid f \in \text{fields}(c)\} \\
E & = \{m → σ \mid m → σ \in \text{methE}(c)\} \\
µ' & = \{loc → [c.F.E] \} \oplus µ
\end{align*}
\]

\[
\langle E[\text{new} c()], µ \rangle \mapsto \langle E[loc], µ' \rangle
\]

The \text{(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 \( E \) that contains each method \( m \) found in previous step and its statically computed effects \( σ \). The static type-and-effect rules in Section [3] are used to compute the effects \( σ \), which is then stored in \( CT \).

\[
\text{methE}(c) = E \oplus \bigcup_{i=0}^{n} \{m_i → σ_i\} \quad \text{where } \text{CT}(c) = \text{class } c \text{ extends } d \{\text{field meth}\}
\]

\[
\text{and methE}(d) = E \quad \text{and } (\forall i \in \{1..n\} :: \text{findMeth}(c, m_i) = (c, i, m_i(F \text{var}), σ_i))
\]

\[
\text{(GET)}\quad \mu(loc) = [c.F.E] \\
μ = update(µ, loc, F(f), v)
\]

\[
\langle E[loc.f], µ \rangle \mapsto \langle E[v], µ' \rangle
\]

\[
\text{(SET)}\quad \begin{align*}
[c.F.E] & = \mu(loc) \\
µ_0 & = µ \oplus \{loc → [c.(F \oplus (f → v)).E]\}
\end{align*}
\]

\[
\langle E[loc.f = v], µ \rangle \mapsto \langle E[v], µ' \rangle
\]

The semantics of field get is standard, whereas that of field set is new. If a field is declared open, assigning a value to it may change the effect of those methods that access it. The function \( \text{update} \) shown below models this.

\[
\text{update}(µ, loc, f, v) = µ \quad \text{where } µ(loc) = [c.F.E] \quad \text{and } E = \text{updateEff}(µ, f, v, E)
\]

\[
\text{update}(µ, loc, f, v) = µ'' \quad \text{where } µ(loc) = [c.F.E] \quad \text{and } E' \neq E
\]

\[
\text{and } E' = \text{updateEff}(µ, f, v, E) \quad \text{and reverse}(µ, loc) = κ
\]

\[
\text{and } µ' = \{loc → [c.F.E']\} \oplus µ \quad \text{and } \text{fixPoint}(µ', loc, κ) = µ''
\]

The inputs to \( \text{update} \) are the current store \( µ \), the object reference \( loc \), the field \( f \), and the R-value \( v \). The result is a modified store. This function first updates the effect (by
calling \textit{updateEff} of the object pointed to by \textit{loc} (by the effect of an object \textit{o}, we mean the effects of all the methods of \textit{o}). If the effects of \textit{o} remain unchanged, the algorithm stops. Otherwise, the effect of an object \textit{o’}, which has some open field pointing to \textit{o}, should also be changed. Effects are further propagated using the function \textit{fixPoint} until a fixed point is reached.

\begin{equation}
\text{reverse}(\mu, \text{loc}) = \bigcup_{i=1}^{n} S_i \quad \text{where} \quad \forall i \in \{1..n\} \text{ s.t. } \text{loc}_i \in \text{dom}(\mu) \implies S_i = \{(\text{loc}_i, f) \mid F(f) = \text{loc} \land \mu(\text{loc}_i) = \text{e\text{.F.E}}\}
\end{equation}

\begin{equation}
\text{fixPoint}(\mu, \text{loc}, \kappa) = \mu_n \quad \text{where} \quad \kappa = \{(\text{loc}_i, f_i) \mid 1 \leq i \leq n\}
\end{equation}

\begin{equation}
\text{and } \text{update}(\mu, \text{loc}_1, f_1, \text{loc}) = \mu_1 \quad \text{and } \forall i \in \{2..n\} \implies \text{update}(\mu, \text{loc}_{i-1}, f_i, \text{loc}) = \mu_i
\end{equation}

The function \textit{reverse}, searches the input store \(\mu\) for objects \(\text{loc}_i\) and field \(f_i\) pair that is pointing to the current object \textit{loc}. In practice, reverse pointers could be used to optimize this update \cite{6}.

\begin{equation}
\text{updateEff}(\mu, f, v, \langle m, \sigma \rangle) = \langle m, \sigma’ \rangle \quad \text{where} \quad \forall i \in \{1..n\} \sigma_i = \{\varepsilon_i|1 \leq k \leq p\}
\end{equation}

\begin{equation}
\text{and } \sigma’_i = \{\varepsilon_i|1 \leq k \leq p\} \quad \text{and } \forall j \in \{1..p\} \implies \varepsilon_j \in \sigma_i : \text{concretize}(\mu, f, v, \varepsilon_j) = \varepsilon_j’
\end{equation}

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

\begin{equation}
\text{concretize}(\mu, f, v, \varepsilon) = \text{match } \varepsilon \text{ with }
\begin{cases}
\text{@open } f’ m \sigma \rightarrow f’ \text{ with } \\
| f \rightarrow \text{match } v \text{ with } \\
| \text{null } \rightarrow \text{open } f m \emptyset \\
| \text{loc } \rightarrow \text{open } f m \sigma’ \text{ where } [c.F.E] = \mu(\text{loc}), \text{ and } \sigma’ = \cup_{i=1}^{n} \sigma_i \\
| \text{and } E(m) = \{\varepsilon_i|1 \leq i \leq n\} \text{ and } \forall i \in \{1..n\} \implies \text{cp}(\varepsilon_i) = \sigma_i
\end{cases}
\end{equation}

\begin{equation}
\text{cp}(\varepsilon) = \text{match } \varepsilon \text{ with } 
\begin{cases}
| \text{open } f m \sigma \rightarrow \sigma \\
| _{-} \rightarrow \varepsilon
\end{cases}
\end{equation}

The function \textit{concretize} changes the concrete effects in the placeholder inside an open effect. Note that when the open field \textit{f} is set (in the \(\text{SET}\) rule), only the open effects that have \textit{f} as receiver are concretized, i.e., \text{@open } \textit{f} \text{ m } \sigma.

\begin{equation}
\text{cp}(\varepsilon) = \text{match } \varepsilon \text{ with } 
\begin{cases}
| \text{open } f m \sigma \rightarrow \sigma \\
| _{-} \rightarrow \varepsilon
\end{cases}
\end{equation}

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

The (\text{CALL}) rule is standard. It acquires the method signature via the function \textit{findMeth} (Section\cite{9.1}) that uses dynamic dispatch\cite{12}.

\begin{equation}
\text{(CALL)} \quad (e’, t, m(T\text{var})\langle e \rangle, \sigma) = \text{findMeth}(c, m) \quad [c.F.E] = \mu(\text{loc}) \quad e’ = [\text{loc}/\textbf{this}.v/\text{var}]e
\end{equation}

\begin{equation}
\langle e\prime \rangle, \mu \quad \rightarrow \quad \langle e\prime \rangle, \mu
\end{equation}
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. These effects can then be used at runtime for checking effect based properties. We show an example of such property in Section 6.

As discussed in Section 2, other open references can be allowed, e.g., the type system can be extended to generate an open effect @open var m @open for a method call var.m(...). In semantics, the concretization of this open effect would happen when an open variable var is set, i.e., when variables are bound to a location in the (CALL) rule.

4.3 Soundness: Type and Effect Preservation

We have proven two key formal properties: effect preservation and type preservation. The proof of type preservation uses the standard subject reduction argument [19]. It is contained in Section 11, which also contains detailed proof for effect preservation.

The effect preservation property is that the dynamic effect, i.e., heap accesses, of each expression $e$ refines the open effects of $e$ computed right before it is evaluated. Proving effect preservation is non-trivial compared to static effect approaches [33, 13], in which the exact effect of a task is known statically. Main technical challenge is to prove that even though open effects of an expression may change due to concretization, dynamic effects continue to refine open effects.

A dynamic effect $\eta$ of an expression $e$ can be a read effect (rd, loc, f) or a write effect (wt, loc, f). A dynamic effect $\eta$ refines a static effect $\sigma$, written $\eta \prec \sigma$, if either $\eta = (rd, loc, f) \wedge (read, f) \in \sigma$; or $\eta = (wt, loc, f) \wedge (write, f) \in \sigma$. The dynamic effect of an expression $e$ is a dynamic trace $\chi = \eta_1$, a sequence of dynamic effects.

To record dynamic effects for proofs, we use an instrumented semantics $\text{dyn}$. If $\Sigma$ reduces to $\Sigma'$ in the original semantics, then $\text{dyn}(\Sigma, \chi)$ reduces to $\text{dyn}(\Sigma', \chi')$. This reduction continues until it evaluates to a value, i.e., $\text{dyn}(v, \mu_v ; \chi_v)$; $\mu_v$ is the final store and $\chi_v$ contains all the heap accesses. Here, if $\Sigma$ is $([E[\text{loc}. f], \mu]$ and $([E[\text{loc}. f = v], \mu])$ then $\chi' = \chi + (rd, loc, f)$ and $\chi + (wt, loc, f)$ respectively. Otherwise, $\chi' = \chi$. It is trivial to see that this instrumented semantics retains the formal properties of the original dynamic semantics [33, 13].

The semantics stores the effects $\sigma$ in object records for methods. Therefore, two invariants (Definition 3), for these method effects $\sigma$, are necessary to maintain the effect preservation property. These invariants include: the placeholder effect $\sigma_0$, of an open effect @open f m $\sigma_0$, should be supereffect $\supseteq$ of the effect $E'(m)$ for the method $m$ of the object $f$ is pointing to (Definition 3), i.e., $E'(m) \subseteq \sigma_0$, and the effect $E(m)$ of a method $m$ stored in the object record should be supereffect of the effect $\sigma$ of the body $e$ of $m$ (Definition 3). For example, in Section 4 after the open parameter c is bound to the instance p, the open effect, of the for loop of the ArrayList instance a, is supereffect of the effect of the method run of p; $\text{run} = \text{null}$.

Definition 1. [Well-formed object] An object record $o = [c.F.E]$ is a well-formed object in $\mu$, written $\mu \vdash o$, if for all open effect @open f m $\sigma_0 \in \sigma \in \text{rng}(E)$, either $(F(f) = \text{loc}) \land (\mu(\text{loc}) = [c.F'.E']) \land (E'(m) \subseteq \sigma_0)$; or $(F(f) = \text{null}) \land (\sigma_0 = \emptyset)$. 
Definition 2. [Well-formed location] A location $loc$ is well-formed in $\mu$, written $\mu \vdash loc$, if either $\mu(\{loc\} = \{c.F.E\}, \forall m \in \text{dom}(E) \text{ s.t.} \text{findMeth}(c,m) = \langle e', t, m(f,\var{var}) \{ e \}, \sigma' \rangle \land \\
\mu[\{loc/\text{this}\} e \sigma]$, then $\sigma \subseteq E(m)$; or $\mu(\{loc\} = \text{null}$.

Definition 3. [Well-formed store] A store $\mu$ is well-formed, written $\mu \vdash \sigma$, if $\forall o \in \text{rng}($$\mu$$) \text{ s.t.} \mu \vdash o$ and $\forall loc \in \text{dom}(\mu)$ s.t. $\mu \vdash loc$.

In the following, we use the relation $\mu \vdash e : \sigma$. Given an expression $e$ and a store $\mu$, it computes the potential dynamic effects of $e$. Please see Section [11] for details.

Theorem 1. [Effect preservation] Given two program configurations $\Sigma = \langle e, \mu \rangle$ and $\Sigma' = \langle e', \mu' \rangle$, if $\Sigma \rightarrow \Sigma'$, the store is well-formed $\mu \vdash \sigma$, then there is some effect $\sigma'$ and dynamic trace $\chi$ such that

(a) potential dynamic effects of resulting expression $e'$ are subeffect of static effects, if $\mu' \vdash e' : \sigma'$ then $\sigma' \subseteq \sigma$; and

(b) new dynamic effect in trace refines static effects, $\text{dyn}(\Sigma, \chi) = \langle \Sigma', \chi + \eta \rangle \Rightarrow (\eta \propto \sigma)$.

Proof Sketch: The essence of Theorem[1] is that during program execution, the subsequent expression $e'$ has a subeffect $\sigma' \subseteq \sigma$ of the previous expression $e$, with the effect judgment $\mu \vdash e : \sigma$. We prove that the dynamic effect $\eta$ in each step refines the static $\sigma$ of the original expression $e$, $\eta \propto \sigma$. Thus with (a), $\eta$ refines the effect $\sigma_0$ of the expression $e_0$ right before it is evaluated, with the heap $\mu_0$. Unlike the static approaches, which compute $\sigma_0$ at compile-time, $\text{OpenEffectJ}$ computes $\sigma_0$ before evaluating $e_0$.

5 Adding Open Effects to the OpenJDK Java Compiler

To show the feasibility of supporting open effects in an industrial-strength compiler, we have extended the OpenJDK Java compiler to add support for open effects. Apart from modifications to support the $\text{open}$ annotation, parsing remains unchanged. Type checking, the Attribute and Flow phases in the compiler, are modified to implement new constraints specified in Section [3]. This phase is also extended with an effect analysis. This phase attributes each AST node with static effects for each method, which is then used by the tree rewriting phase to generate code for runtime effect manipulation.

Stronger Effect Analysis. The effect analysis is augmented with two modular analyses to improve precision. These include an intra-procedural definite alias analysis [21] and a purity analysis [34]. The alias analysis tracks the aliasing information for local variables and parameters. This is useful for finding more accurate type information for receiver objects of method call expressions, and thus giving more accurate effects than the $\bot$ in the (T-CALL-OPEN) rule in Section [3]. E.g., inferring the concrete effect if the exact type is known. The purity analysis detects objects allocated within the scope of a method, which reveals more pure methods and removes redundant effects.

Object and Field Sensitive Effect Storage. 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. The concrete effects are object sensitive, which tracks the object $o$, whose field $f$ is being accessed. For instance, in Section [3], the rules (T-GET) and (T-SET) will produce an effect $\{\text{read } f\}$.
and \{\textbf{write} f\}, respectively. When an object \(o\) is created, i.e., the (NEW) rule in Section 4.2, the implicit placeholder \texttt{this} will be replaced with the location \(\text{loc}\) of \(o\) being created. The read/write effects will be stored as \{\texttt{read} loc f\} and \{\texttt{write} loc f\}, making the dynamically-computed effects more precise.

**Effect Maintenance.** 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 \texttt{update}. In the implementation, we maintain a reverse pointer from \(o\) to \(o'\) for efficiency. 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 \(f\) of an object \(o\) is assigned a value, concrete effects of the methods, of \(o\), may change. We generate a method \texttt{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. Otherwise, it calls the \texttt{cascade} method of all its reverse pointers.

### 6 Using Open Effects for Safe Concurrency

We hypothesize that open effects are useful for exposing safe and optimistic concurrency in libraries and frameworks, which could be extended with possibly concurrency-unsafe code by clients, e.g. \texttt{ArrayList}. To test this hypothesis, we have extended the infrastructure discussed in the previous section to add a concurrency library. We then use this library to parallelize several applications. This section reports on these results.

#### 6.1 Checking (non)interference of concurrent tasks using open effects

Our concurrency library provides one method \texttt{fork} that take two arguments: \(t\) of type \texttt{Task} and \(\text{input}\) an array of parameterized type \(U\). The type \texttt{Task} is an interface provided by our library that provides one method \texttt{run} that takes a single argument of type \(U\). When called, the method \texttt{fork} first retrieves the dynamic effects of the \texttt{run} method from the object \(t\) using the compiler-generated methods made available by the \texttt{OpenEffectJ} compiler.

The method \texttt{fork} then tests to see if multiple invocations of the \texttt{run} method with \(t\) as the receiver object will have mutually conflicting effects. Read effects do not interfere; read/write and write/write pairs conflict if they access the same field \(f\) of the same object \(\text{loc}\) (object and field sensitivity Section 5); open effect \(\otimes\text{open} f m\sigma\) conflicts with another effect \(\sigma'\) if any effect \(\sigma''\) in \(\sigma\) conflicts with \(\sigma'\); bottom effect \(\bot\) conflicts with any effect.

If multiple calls will not have conflicting effects then the library executes \(n\) parallel copies of the \texttt{run} method, where \(n\) is the size of the array \text{input}. Otherwise, the library executes a sequential loop that calls the \texttt{run} method with each element of the array \text{input} as argument.
6.2 Parallelizing representative libraries

To assess the usefulness of open effects, we have studied several representative libraries.

Map-Reduce. In this framework [1], the first step is map, i.e. partitioning the problem and distributing it to worker, and the second step is reduce, i.e. combining results from workers. For extensibility and reuse, this framework is designed to use abstract implementation of classes Mapper and Reducer, which are extended by clients to implement application-specific functionality. The class Mapper provides one method `map` that takes one argument, the input to be processed and the class Reducer provides one method `reduce` that takes two arguments, the results to be combined. Since we may not know the effects of the overriding implementations of the `map` and the `reduce` methods, in the implementation of the MapReduce algorithm, we annotate these types with `@open`. This allows safe parallelization, in recursions on the subarrays, when mapper and reducer point to instances of concurrency safe subclasses of Mapper and Reducer, respectively. In total, we added two `@open` annotations.

MergeSort. The MergeSort library is from the package `java.util` in OpenJDK. It uses a divide-and-conquer technique with an insertion sort as a base case for small inputs. To sort the elements in an array, MergeSort uses an instance of the class Comparator to compare two elements in the array. The clients extend the sorting by implementing application-specific Comparators. The comparators may not be pure, e.g., in OpenJDK itself, the class `RuleBasedCollator` (RBC) in package `java.text` is-a Comparator, but has side effects. So the parallelization of MergeSort, on the recursive calls, may have heap conflicts if an instance of RBC is used as a Comparator and would result in incorrect output [1]. We declare the parameter `c` of type Comparator in the MergeSort method as `@open`. Nothing else changes!

DFS. Depth-first search is a representative search algorithm, typically formulated as a graph traversal [25]. It recursively traverses the nodes in a graph and returns all the nodes that satisfies certain objective. This library uses the abstract implementation of the class Goal and let the clients extend it to implement application-specific search objectives. We annotate the field `goal` of type Goal with `@open`. With more precise effect, the algorithm can be parallelized by executing the recursive DFS concurrently.

Numerical Integration. This application (NI) uses Guassian Quadrature for numerical integration [1]. NI computes the area from lower bound to the center point of interval, and from the center point to the upper bound. If the sum of the above areas differs from the value from lower to upper by more than the predefined error tolerance, it recurses on each half. NI uses the abstract implementation of the class Function, extended by clients to implement application-specific function to be integrated. We annotate the field `f` of type Function with `@open`. NI can be safely parallelized on recursion if `f` points to a concurrency safe Function.

6.3 Performance Evaluation

We have conducted an initial evaluation of OpenEffectJ’s prototype compiler using the library classes described in Section 1 and Section 6. These are array list with hash

1 The original code in `RuleBasedCollator` is thread safe though.
computation (Hash), array list with slightly heavier computation (Heavy), merge sort algorithm (MergeSort), depth-first search algorithm (Search), a map-reduce application (MapReduce) and a numerical integration (NI). 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 30 runs was taken.

For each element \( o \) in the list, the Heavy variant of array list computes the formula

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

For both Hash and Heavy variant, array list contained 20 Million elements. Merge Sort sorts a list of 10 Million randomly generated integers. The search algorithm searches for solutions to an n-queens problem, where \( n \) is 11. In the map-reduce algorithm, map step computes the formula

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

for the element \( o \) and reduce step is simply addition. It is applied to 100 Million integers. NI uses a recursive Gaussian quadrature of \( (2 \cdot i - 1) \cdot x^{2^i - 1} \), summing over odd values of \( i \) from 1 to 12 and integrating from \(-5\) to 6.

The performance results of running these applications are shown in the table above. In this table, the columns marked \( \text{Serial} \) shows the time taken by the single-threaded version of the application. The columns marked \( \text{Manual} \) 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 \( \text{OpenEffectJ} \) shows the time taken by (and speedup of) a version of our compiler, which does effect comparison prior to forking off each task. The columns marked \( \text{OpenEffectJ}' \) 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).

<table>
<thead>
<tr>
<th>Program</th>
<th>Serial time (s)</th>
<th>Manual (No effects/Unsafe) time (s)</th>
<th>speedup</th>
<th>OpenEffectJ speedup</th>
<th>overhead</th>
<th>OpenEffectJ' (Improved) time (s)</th>
<th>speedup</th>
<th>overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>ArrayList(Hash)</td>
<td>0.13</td>
<td>0.11</td>
<td>1.19</td>
<td>0.11</td>
<td>1.17</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>ArrayList(Heavy)</td>
<td>1.30</td>
<td>0.55</td>
<td>2.35</td>
<td>0.56</td>
<td>2.33</td>
<td>0.71%</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>MergeSort</td>
<td>2.54</td>
<td>1.15</td>
<td>2.20</td>
<td>1.31</td>
<td>1.94</td>
<td>13.52%</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Depth First Search</td>
<td>27.63</td>
<td>13.80</td>
<td>2.00</td>
<td>19.89</td>
<td>1.39</td>
<td>44.17%</td>
<td>13.67</td>
<td>2.02</td>
</tr>
<tr>
<td>MapReduce</td>
<td>6.58</td>
<td>2.43</td>
<td>2.71</td>
<td>3.03</td>
<td>2.17</td>
<td>24.69%</td>
<td>2.44</td>
<td>2.70</td>
</tr>
<tr>
<td>Integrate</td>
<td>2.13</td>
<td>1.79</td>
<td>1.18</td>
<td>2.49</td>
<td>0.85</td>
<td>38.79%</td>
<td>1.91</td>
<td>1.11</td>
</tr>
</tbody>
</table>

The manual (unsafe) version showed decent to good speedup for all of these applications. The \( \text{OpenEffectJ} \) and \( \text{OpenEffectJ}' \) version also showed comparable speedup. For the array list, results for both \( \text{OpenEffectJ} \) and \( \text{OpenEffectJ}' \) are the same. The overhead for this example was also small. For the MergeSort, Search, MapReduce and Integrate examples, however, the overhead for \( \text{OpenEffectJ} \) increased significantly. This was because each of these problems use recursive, parallel algorithm. This was precisely where \( \text{OpenEffectJ}' \) 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). More attention to \( \text{OpenEffectJ}' \)'s compiler will help discover simple and more clever optimizations that will decrease overheads further.
6.4 Discussion: Scope and Applicability of Open Effects

We now compare open effects with static and dynamic effect systems from the viewpoint of concurrency. The best scenario for a static effect system is when using static type-and-effects, 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 effect system wins hands down with no runtime overhead. However, if a static system makes use of many conservative approximations, because accurate type information is not available, an optimistic approach would be a more desirable model. The best scenario for a dynamic effect system is when the parallel section has alternate paths, e.g., $p_1$ and $p_2$, some of which, say $p_1$, have data races, but these are not the hot paths in program. The others, say $p_2$, have no side effect and are frequently executed. This is because 1) $p_1$ will indeed be executed, and so all the sound models which make decisions before the parallel section must indicate that it is not safe; and 2) $p_2$ is more frequently used.

There are at least two scenarios when open effects 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 information is available 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. A technique based on open effects can indicate that the task $c$ should be run after the tasks $a$ and $b$ are done and tasks $a$ and $b$ can be run concurrently, without requiring rollbacks.

Summary. We applied OpenEffectJ to 5 representative examples, 3 of which are library classes from OpenJDK. For each case, OpenEffectJ gracefully assists the programmer in parallelization of reusable libraries and/or frameworks. Here the libraries or frameworks could be extended by the clients. Thus, OpenEffectJ optimistically provides safe concurrency opportunities. In each case, at most two annotations were needed to safely parallelize the library class under consideration. Finally, in each case OpenEffectJ did not require the entire client code for effect analysis.

7 Comparative Analysis with Related Work

The notion of open effects is closest in spirit to the ideas of gradual typing [36] and hybrid type checking [26] that blend the advantages of static and dynamic type checking. Similarly, open effects blends the advantages of static and dynamic effect systems. We now compare open effects with closely related ideas.

There exists a rich body of work on type, regions and effect-based approaches for reasoning about object-oriented programs [22,33,7,9,12]. These approaches are static, whereas open effects uses a hybrid approach. A recent work in this category is that on deterministic parallel Java (DPJ). DPJ [7] 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, open effects require 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, open effects suggest running those tasks serially.
There is also a large body of work on dynamic approaches for reasoning about object-oriented programs [17,37,16,8,24]. In essence, these approaches monitor memory footprints of programs to compute dynamic effects that can then be used for checking effect-based properties. In contrast, open effects requires monitoring references annotated as `open` and updating statically computed effects.

### 7.1 Comparison with ideas related to open effects based concurrency

**Overview of related ideas** Like open effects, synchronization via scheduling (SVS) [6] computes effects between potentially concurrent tasks right before forking them off. SVS supports a C like language. It uses reachable objects graph (OG) of tasks as their effects [32]. Compared to SVS, open effects supports a full OO language with support for overriding and dynamic dispatch, which makes accurate effect computation much more challenging [18]. Also, using effects sets instead of reachable OG may be more precise for OO features, e.g., in every library in Section 6, 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 open effects suggest parallelism.

Transactional memory [38,35,23,28] optimistically executes tasks concurrently, but monitors memory accesses. It rollbacks side-effects when conflicts happen. There are TM-like approaches [15,5,41] that provide sequential consistency (DTM) by enforcing a deterministic commit order, instead of rolling back nondeterministically on conflict. In contrast, an open-effect-based approach does not need state buffering.

In concurrent revisions [10,11] 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 contrast, open effects is useful where all overriding subclasses may not be known. For example, when a library class `c` is developed, the accurate effect may be unknown, because `c` can be extended with concurrency-unsafe code by clients.

#### Criteria and results
The comparison criteria and the results are summarized below:

<table>
<thead>
<tr>
<th>Work</th>
<th>Shared memory</th>
<th>Object-oriented</th>
<th>Effect annotation</th>
<th>Deployment time</th>
<th>Type Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>Open effects</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Hybrid</td>
<td>Partial dynamic types</td>
</tr>
<tr>
<td>DPJ [33]</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Static</td>
<td>Static types</td>
</tr>
<tr>
<td>SVS [6]</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Hybrid</td>
<td>Partial dynamic types</td>
</tr>
<tr>
<td>TM [38,35,23,28]</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Dynamic</td>
<td>Full dynamic types</td>
</tr>
<tr>
<td>DTM etc. [15]</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Dynamic</td>
<td>Full dynamic types</td>
</tr>
<tr>
<td>FastTrack [17], CP [37]</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Dynamic</td>
<td>Full dynamic types</td>
</tr>
<tr>
<td>Goldilocks [16]</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Hybrid</td>
<td>Full dynamic types</td>
</tr>
<tr>
<td>Galois [50]</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Dynamic</td>
<td>Full dynamic types</td>
</tr>
<tr>
<td>Revision [10,11]</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Dynamic</td>
<td>Full dynamic types</td>
</tr>
<tr>
<td>Actor [3]</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Static</td>
<td>Static types</td>
</tr>
</tbody>
</table>

The criteria shared memory, object-oriented and effect annotation are self-explanatory. All of the approaches except actor-based approaches are for shared-memory model, all of the approaches except SVS are for object-oriented languages, and all of the approaches except DPJ do not require effect annotations.

The last two columns, deployment time and type information show when the systems are activated and how optimistic they are for concurrency. A static approach does...
reasoning at compile time, has least runtime information and is least optimistic. A hybrid approach, like open effects, uses static information to facilitate the runtime analysis and is more optimistic than a static one. A dynamic approach reasons about correctness completely at runtime and is the most optimistic. Goldilocks is considered hybrid because it could apply static analysis to reduce runtime overhead, however, that analysis requires closed world assumption.

8 Conclusion and Future Work

We presented an optimistic type-and-effect system for modern object-oriented languages with an open world assumption. New to our type-and-effect system is the notion of open effects, which is a placeholder effect. It is produced by method calls when the dynamic type of the receiver object is unknown. For an effect-based property, an open effect is assumed to be satisfying that property statically but verified to be truly so when the dynamic type of the receiver is known. Open effects have several benefits. It enables modular analysis of partial programs and libraries. It has a negligible annotation overhead. It enables more precise treatment of dynamic dispatch in hybrid analyses compared to static effect systems with similar annotation requirements, but incurs some runtime overhead. The treatment of dynamic dispatch may be less precise than a dynamic analysis based on effects, but has less overhead. We have formalized a type-and-effect system that includes open effects and proven that it is sound. We have also extended the OpenJDK Java compiler with support for open effects. To investigate the utility of open effects, we applied it to analyze (non)interference of concurrent tasks, where it shows only about 0.37-6.76% overhead and good speedup. In future, it would be sensible to explore a logical extreme, where every reference is implicitly open and a static analysis is used to systematically eliminate @open references.

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

References

1. JSR-166y for Java 7. [http://gee.oswego.edu/dl/concurrency-interest/]
8. Bond, M.D., Coons, K.E., McKinley, K.S.: Pacer: proportional detection of data races. In: PLDI ’11
13. Clarke, D., Drossopoulou, S.: Ownership, encapsulation and the disjointness of type and effect. In: OOPSLA ’02
27. Leroy, X., Pessaux, F.: Type-based analysis of uncaught exceptions. TOPLAS ’00, 22(2)
31. Neamtiu, I., Hicks, M., Foster, J.S., Pratikakis, P.: Contextual effects for version-consistent dynamic software updating and safe concurrent programming. In: POPL ’08
33. R. Bocchino et al.: A type and effect system for deterministic parallel Java. In: OOPSLA ’09
34. Salcianu, R.D., Rinard, M.C.: Purity and side effect analysis for Java programs. In: VMCAI ’05
38. T. Shpeisman et al.: Enforcing isolation and ordering in STM. In: PLDI ’07
40. Talpin, J.P., Jouvelot, P.: Polymorphic type, region and effect inference. JFP ’92, 2(3)
9 Type-and-effect System: Omitted Details

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

9.1 Type-and-Effect Rules for Declarations

The rules for top-level declarations are fairly standard. Below, the (T-PROGRAM) rule

\[
\begin{align*}
\forall decl_i \in decl \vdash decl_i : OK & \quad \vdash e : (t, \sigma) \\
\vdash decl e : (t, \sigma)
\end{align*}
\]

The (T-CLASS) rule says that a class declaration type checks if all the following

\[
\begin{align*}
\forall field_k \in field : validF(field_k, d) & \quad isClass(d) \quad \forall meth_j \in meth \vdash meth_j : (t_j, \sigma_j) \text{ in } c \\
\text{class } c \text{ extends } d \{ field \; meth \} : OK
\end{align*}
\]

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

\[
CT(c) = \text{class } c \text{ extends } d \{ field_1 \ldots field_n \; meth \} \\
\forall i \in \{1 \ldots n\} \text{ s.t. field}_i = [\texttt{open}] t f; \quad validF(f, d) \\
\quad validF(f, c) \\
\quad validF(f, Object) \\
\text{class } c \text{ extends } d \{ field \; meth \} \in CT \quad isClass(t) \lor (t = \texttt{void}) \\
isType(t)
\]

9.2 Type-and-Effect Rules for Expressions

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

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

\[
(T-DEFINE) & \quad isClass(c) \quad \Pi \vdash e_1 : (t_1, \sigma) \quad \Pi, var : c \vdash e_2 : (t_2, \sigma') \quad t_1 <: c \\
\Pi \vdash c \; var = e_1; e_2 : (t_2, \sigma \cup \sigma')
\]
The (T-NEW) rule ensures that the class \( c \) being instantiated was declared. This expression has empty effect. The (T-VAR) rule checks that \( var \) is in the environment. The (T-NULL) rule says that the null expression could be of any valid type. The declaration expression (T-DEFINE) rule ensures that the initial expression should be a subtype of the type of the new variable. Also, the subsequent expression \( e_2 \) types check if the type of the variable is placed in the environment.

The auxiliary function \( \text{typeOfF} \) (used in the rules in Section \( \text{3} \)), 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 \( f \).

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

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

## 10 Dynamic Semantics: Omitted Details

This section presents auxiliary functions that were omitted in Section \( \text{4} \) for brevity.

The \( \text{fields} \) function, used in the (NEW) rule, returns all the fields declared in the class and its super classes (it uses the \( \text{fieldOf} \) function defined in Section \( \text{9.2} \)).

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

We now prove the key properties of OpenEffectJ: Effect and Type Preservation. Some of the definitions, descriptions and proof sketches are also in Section 4.3. We write all these for the sake of clarity.

We have proven the soundness of OpenEffectJ’s type system (Section 11.3 contains proof that use the standard subject reduction argument [19]). The Effect preservation property is that the dynamic effect (heap accesses) of each expression refines the static effect before it is evaluated. We prove this in Section 11.2. Proving effect soundness is non-trivial compared to static effect approaches [33,13], in which the exact effect of an expression is known statically. A technical challenge for proving the soundness of OpenEffectJ is that the effects of an expression may change due to the open effect, i.e., the effect concretization.

11.1 Preliminary Definitions

We now give some preliminary definitions used in the proofs for OpenEffectJ’s properties. A standard approach to show effect soundness for a type-and-effect system is to prove that the static effect of an expression $e$ computed before the evaluation of $e$ bounds the heap accesses of $e$. To record the heap accessed for $e$, we define dynamic trace $\chi$, which contains a sequence of dynamic effects (heap accesses) by the tasks.

**Definition 4. [Dynamic Trace]** A dynamic trace ($\chi$) consists of a sequence of dynamic effects ($\eta$), where $\eta$ can be a read effect ($rd$, loc, f) or write effect ($wt$, loc, f).

The function $\text{dyn}$, defined in Figure 6, records the dynamic memory footprint for the evaluation of an expression $e$. With it, we can prove that the dynamic effects of each expression refines the static effect $\sigma$ computed when it is evaluated.

**Fig. 6.** Dynamic Effect function $\text{dyn}$.

In the following, $\text{dyn}(\Sigma, \chi) = \text{dyn}(\Sigma', \chi')$ and $\Sigma \rightarrow \Sigma'$.

<table>
<thead>
<tr>
<th>$\Sigma$</th>
<th>Side Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>$[e[\text{loc}].f, \mu]$</td>
<td>$\chi = \chi + (rd, \text{loc}, f)$</td>
</tr>
<tr>
<td>$[e[\text{loc}].f = v, \mu]$</td>
<td>$\chi = \chi + (wt, \text{loc}, f)$</td>
</tr>
<tr>
<td>Other cases</td>
<td>$\chi = \chi$</td>
</tr>
</tbody>
</table>

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 5. [Static effect inclusion]** An effect $\varepsilon$ is included in an effect set $\sigma = \{\varepsilon_i | 1 \leq i \leq n\}$, written $\varepsilon \in \sigma$, if either: $\exists \varepsilon_i$ s.t. $\varepsilon = \varepsilon_i$; or $\exists \varepsilon_i$ s.t. $(\varepsilon_i = \emptyset \text{open } f \text{ m } \sigma') \wedge (\sigma' = \{\varepsilon'_j | 1 \leq j \leq n'\}) \wedge (\varepsilon = \varepsilon'_j)$. 
This definition says that an effect $\varepsilon$ is included in an effect set $\sigma$ if it is one of the elements in $\sigma$; or there is an open effect $\emptyset$ in $\sigma$ and $\varepsilon$ is an element of $\sigma$.

**Definition 6.** [Dynamic effect refines static effect] A dynamic effect $\eta$ refines a static effect $\sigma$, written $\eta \equiv \sigma$, if either $\eta = (\text{rd}, \text{loc}, f) \land (\text{read } f) \in \sigma$; or $\eta = (\text{wt}, \text{loc}, f) \land (\text{write } f) \in \sigma$.

In Section 11.2, we will show that during the evaluation, the effect $\sigma$ of an expression $e$, is refined by the effect $\sigma'$ of its subsequent expression $e'$, i.e., if $\langle e, \mu \rangle \rightarrow \langle e', \mu' \rangle$, $\mu \vdash e : \sigma$ and $\mu \vdash e' : \sigma'$, then $\sigma' \subseteq \sigma$. This guarantees that the static effect, computed before an expression is evaluated, is a sound approximation of the effects of all subsequent expressions. Here we define how an effect $\sigma'$ refines another effect $\sigma$.

**Definition 7.** [Static effect refinement] An effect set $\sigma'$ refines another effect set $\sigma$ if $\sigma' \subseteq \sigma$.

During the evaluation of an expression, the store keeps changing, and we want to ensure that the same expression has the same static effect. To do so, we define effect equivalent stores (Definition 8) and prove that these stores give the same effects for a same expression.

**Definition 8.** [Effect equivalent stores] Two stores $\mu$ and $\mu'$ are effect equivalent, written $\mu \equiv \mu'$, if both conditions hold: $\text{dom}(\mu) \subseteq \text{dom}(\mu')$; and $\forall \text{loc} \text{ if } \mu(\text{loc}) = [c.\text{F}.\text{E}]$, then $\mu'(\text{loc}) = [c.\text{F}'.\text{E}]$, for some $\text{F}'$.

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 [33, 13]. The novelty is that effects for method calls are new in this work and OpenEffectJ needs to maintain proper effects for methods (Definition 9 and Definition 10). To prove that a method call on an open field has static effects that are refined by their subsequent expression, we introduce well-formed object.

**Definition 9.** [Well-formed object] An object record $o = [c.\text{F}.\text{E}]$ is a well-formed object in $\mu$, written $\mu \vdash o$, if for all open effect $\emptyset \text{ open } f m \sigma_0 \in \sigma \in \text{rng}(E)$, either

\[ F(f) = \text{loc} \land (\mu(\text{loc}) = [c'.\text{F}'.\text{E}']) \land (E'(m) \subseteq \sigma_0); \text{ or } (F(f) = \text{null}) \land (\sigma_0 = \emptyset). \]

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

To prove that a method call on a location $\text{loc}$ have static effects that are refined by their subsequent expression, we introduce well-formed location (Definition 10).

**Definition 10.** [Well-formed location] A location $\text{loc}$ is well-formed in $\mu$, written $\mu \vdash \text{loc}$, if either $\mu(\text{loc}) = [c.\text{F}.\text{E}]$, $\forall \text{m} \in \text{dom}(E)$ s.t. $\text{findMeth}(c.m) = (c', t, m(\overline{\text{var}})\{\text{e}\}, \sigma') \land \mu \vdash \text{loc/this}\ e : \sigma$, then $\sigma \subseteq E(m)$; 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 (\( \supseteq \)) of the effect given by the effect judgment of the body \( e \) of the method \( m \).

Finally, to prove the effect preservation theorem (Theorem 1), we need to prove an invariant of any OpenEffectJ program, i.e., the store is well-formed (Definition 11). 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 11.** [Well-formed store] A store \( \mu \) is well-formed, written \( \mu \vdash o \), if \( \forall o \in \text{rng}(\mu) \) s.t. \( \mu \vdash o \) and \( \forall loc \in \text{dom}(\mu) \) s.t. \( \mu \vdash loc \).

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

**Effect Judgment.** In the following, we use the relation \( \mu \vdash e : \sigma \). Given an expression \( e \) and a store \( \mu \), it computes the potential dynamic effects of \( e \).

\[
\begin{align*}
\text{(E-CALL-OPEN)} & \quad \mu \vdash e : \sigma \quad \mu(\text{loc}) = [c.F.E] \\
\text{(E-CALL-LOC)} & \quad \mu(\text{loc}) = [c.F.E] \\
& \quad E(m) = \sigma_0 \\
& \quad (\forall i \in \{1..n\}) :: \mu \vdash e_i : \sigma_i \\
& \quad \mu \vdash \text{loc}.m(\overline{\text{i}}) : \bigcup_{i=0}^{n} \sigma_i \\
& \quad \exists i \text{ s.t. } (\exists e = @\text{open } f \forall m \sigma \text{ s.t. } e \in \sigma_i) \quad \text{typeOfF}(f) = (d, @\text{open } c_0) \\
& \quad (\forall i \in \{1..n\}) :: \mu \vdash e_i : \sigma_i \\
& \quad \mu \vdash e_0.m(\overline{\text{i}}) : \bot \\
\text{(E-CALL)} & \quad \mu \vdash e : \sigma \\
\text{(E-GET)} & \quad \mu \vdash \text{loc}.f : \{\text{read } f\} \\
\text{(E-SET)} & \quad \text{typeOfF}(f) = (c, @\text{open } c_0) \\
& \quad \mu \vdash \text{loc}.f = e : \{\bot\} \\
& \quad \mu \vdash e : \sigma \\
& \quad \text{typeOfF}(f) = (c.t) \\
& \quad \mu \vdash \text{loc}.f = e : \sigma \cup \{\text{write } f\} \\
\text{(E-NEW)} & \quad \mu \vdash \text{new } c() : \emptyset \\
\text{(E-VAR)} & \quad \mu \vdash \text{var} : \emptyset \\
\text{(E-NULL)} & \quad \mu \vdash \text{null} : \emptyset \\
\text{(E-LOC)} & \quad \mu \vdash \text{loc} : \emptyset \\
\text{(E-DEFINE)} & \quad \mu \vdash e_1 : \sigma_1 \\
& \quad \mu \vdash e_2 : \sigma_2 \\
& \quad \mu \vdash c \text{ var } = e_1; e_2 : \sigma_1 \cup \sigma_2
\end{align*}
\]

The dynamic rules (above) are similar to the static rules in Section 3 except for method calls on open fields (E-CALL-OPEN). The open effect now has a concrete part \( \sigma \), instead of \( \emptyset \), because we know the concrete object that an open field is pointing to. Note that field access can only via the object pointed to by this and we have substitution based semantics, i.e., the this variable will be substituted by the receiver object immediately before the execution of the object. Therefore, the field accesses rules in the effect judgment have the forms \( \text{loc}.f = e \) and \( \text{loc}.f \) for field set and field get respectively.

### 11.2 Effect Preservation

In this section, we prove OpenEffectJ’s effect preservation property. This involves three invariants during the evaluation of an expression. First, the effect \( \sigma \) produced by an
expression \( e \) is refined by the effect \( \sigma' \), by its subsequence expression \( e' \) (\( \sigma' \subseteq \sigma \)); and second, the dynamic effect \( \eta \), produced by the reduction, if any, refines \( \sigma \) (\( \eta \preceq \sigma \)); finally, the store remains effect equivalent (\( \mu \simeq \mu' \)).

**Theorem 2.** [Effect preservation] Let the program configuration \( \Sigma = (e, \mu) \). If it transits to another configuration \( \Sigma \mapsto (e', \mu') \), the store is well-formed \( \mu \vdash \circ \), and \( \mu \vdash e : \sigma \), then there is some \( \sigma' \subseteq \sigma \) s.t.

\( (a) \quad \mu' \vdash e' : \sigma' \), and \( \sigma' \subseteq \sigma \);
\( (b) \quad (\text{dyn}(\Sigma, \chi) = (\Sigma', \chi + \eta)) \Rightarrow (\eta \preceq \sigma) \).

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

**Replacement with Subeffect** The following lemma says that two effect equivalent stores give the same effect \( \sigma \) to the same expression \( e \). Because we will prove that during the execution of an expression, except for the field set open rule, all the stores are effect equivalent \( \mu \simeq \mu' \). Therefore, it suffices to prove that the effect of the subsequent expression \( e' \) refines the effect of the expression \( e \), given by effect equivalent stores.

**Lemma 1.** [Stationary effect] Let \( e \) be an expression, \( \mu \) and \( \mu' \) two stores s.t. \( \mu \simeq \mu' \). If \( \mu \vdash e : \sigma \), then \( \mu' \vdash e : \sigma \).

**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 \( \sigma \). In each case we show that \( \mu \vdash e : \sigma \) implies that \( \mu' \vdash e' : \sigma' \), and thus the claim holds by the induction hypothesis (IH). The base cases include \((\text{NEW}), (\text{NULL}), (\text{LOC}), \text{and} (\text{VAR})\). These cases are obvious: \( \sigma' = \sigma = \emptyset \). The remaining cases cover the induction step. The IH is that the claim of the lemma holds for all sub-derivations of the derivation being considered.

The cases for \((\text{DEFINE})\) and \((\text{SET})\) follow directly from IH.

**DEFINE.** Here \( e = c \ text{var} = e_1 ; e_2 \). The last step is:

\[
\frac{\mu \vdash e_1 : \sigma_1 \quad \mu \vdash e_2 : \sigma_2}{\mu \vdash c \text{var} = e_1 ; e_2 : \sigma_1 \cup \sigma_2} \quad \frac{\mu' \vdash e_1 : \sigma_1' \quad \mu' \vdash e_2 : \sigma_2'}{\mu' \vdash c \text{var} = e_1 ; e_2 : \sigma_1' \cup \sigma_2'}
\]

By the IH, \( \sigma_1' = \sigma_1 \) and \( \sigma_2' = \sigma_2 \). Therefore \( \sigma' = \sigma_1' \cup \sigma_2' = \sigma_1 \cup \sigma_2 = \sigma \). The claim holds.

**GET.** Here \( e = \text{loc}.f \). The last derivation step is:

\[
\frac{\mu \vdash \text{loc}.f : \{\text{read} \ f\}}{\mu \vdash \text{loc}.f : \{\text{read} \ f\}}
\]

Clearly, it holds.

**SET.** Here \( e = \text{loc}.f = e' \). The last derivation step is:

\[
\frac{\mu \vdash e' : \sigma_0 \quad \text{typeOf}(f) = (c, t)}{\mu \vdash \text{loc}.f = e' : \sigma_0 \cup \{\text{write} \ f\}} \quad \frac{\mu' \vdash e' : \sigma_0' \quad \text{typeOf}(f) = (c, t)}{\mu' \vdash \text{loc}.f = e_1 : \sigma_0' \cup \{\text{write} \ f\}}
\]

By IH, \( \sigma_0' \subseteq \sigma_0 \). Thus \( \sigma' = \sigma_0' \cup \{\text{write} \ f\} \subseteq \sigma_0 \cup \{\text{write} \ f\} = \sigma \). The claim holds.
The claim holds clearly.

\textbf{(Call-Open).} Here \( e = \text{loc.f.m}(e_1, \ldots, e_n) \). The last type derivation step has the following form:

\[
\begin{align*}
\mu \vdash \text{loc.f.m(\pi)} & : \{ \text{open f m } \sigma' \} \cup \bigcup_{i=0}^{n} \sigma_i' \\
\mu' \vdash \text{loc.f.m(\pi)} & : \{ \text{open f m } \sigma' \} \cup \bigcup_{i=0}^{n} \sigma_i' \\
\end{align*}
\]

Clearly, \( \sigma_0 = \sigma_0' = \{ \text{read } f \} \). Since \( \mu \cong \mu' \), the effect maps \( E \) are the same. By the IH, \( \forall i \in \{1..n\} :: \sigma_i' = \sigma_i \). Thus \( \sigma' = \{ \text{open f m } \sigma' \} \cup \bigcup_{i=0}^{n} \sigma_i' = \{ \text{open f m } \sigma' \} \cup \bigcup_{i=0}^{n} \sigma_i = \sigma \), and the claim holds.

\textbf{(Call-Loc).} Here \( e = \text{loc.m}(e_1, \ldots, e_n) \). The last type derivation step has the following form:

\[
\begin{align*}
\mu \vdash \text{loc.m(\pi)} & : \bot \\
\mu' \vdash \text{loc.m(\pi)} & : \bot \\
\end{align*}
\]

Since \( \mu \cong \mu' \), the effect maps \( E \) are the same and \( \sigma_0 = \sigma_0' \). By the induction hypothesis, \( \forall i \in \{1..n\} :: \sigma_i' = \sigma_i \). Thus \( \sigma' = \bigcup_{i=0}^{n} \sigma_i' = \bigcup_{i=0}^{n} \sigma_i = \sigma \), and the claim holds.

\textbf{(Call).} Here \( e = e_0.m(\bar{e}) \). The last type derivation step has the following form:

\[
\begin{align*}
\mu \vdash e_0.m(\bar{e}) & : \bot \\
\mu' \vdash e_0.m(\bar{e}) & : \bot \\
\end{align*}
\]

Obviously, the claim holds.

Thus, for all possible derivations of \( \mu \vdash e : \sigma \) and \( \mu' \vdash e : \sigma' \), we see that \( \sigma' = \sigma \). \( \blacksquare \)

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

\textbf{Lemma 2.} [Replacement with subeffect] If \( \mu \vdash_0, \Sigma \mapsto \Sigma', \Sigma = (\mathbb{E}[e], \mu), \Sigma' = (\mathbb{E}[e'], \mu') \), \( \mu \vdash \mathbb{E}[e] : \sigma \), \( \mu \vdash e : \sigma_0 \), \( \mu \vdash e' : \sigma_1 \), \( \mu \cong \mu' \), and \( \sigma_1 \subseteq \sigma_0 \), then \( \mu \vdash \mathbb{E}[e'] : \sigma' \land \sigma' \subseteq \sigma \).

\textbf{Proof:} Proof is by induction on the size of the evaluation context \( \mathbb{E} \). Size of the \( \mathbb{E} \) refers to the number of recursive applications of the syntactic rules necessary to create \( \mathbb{E} \). In
Thus, by \((E-S)\), 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 \(\mu \vdash E_2[e] : \sigma\) implies that \(\mu' \vdash E_2[e'] : \sigma'\), for some \(\sigma' \subseteq \sigma\), and thus the claim holds by the IH.

The cases for \((E-\text{GET})\) and \((E-\text{DEFINE})\) follow directly from the IH.

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

**Case** \(e = c\ var = -; e_2\). The last step is \((E-\text{DEFINE})\):

\[
\frac{\mu \vdash e : \sigma_0 \quad \mu \vdash e_2 : \sigma_2}{\mu \vdash E_2[e] : \sigma_0 \cup \sigma_2}
\]

Thus, by \((E-\text{DEFINE})\), \(\mu \vdash E_2[e'] : \sigma_1 \cup \sigma_2\);

**Case** \(e = -; f\). The last step is \((E-\text{GET})\):

\[
\mu \vdash E_2[e] : \{\text{read } f\}
\]

Thus, by \((E-\text{GET})\), \(\mu \vdash E_2[e'] : \{\text{read } f\};\)

**Case** \(f = e_2\). The last step for \(E_2[e]\) should be \((E-\text{SET})\):

\[
\frac{\text{typeOff}(f) = (c,t) \quad \mu \vdash e_2 : \sigma_2}{\mu \vdash E_2[e] : \sigma_2 \cup \{\text{write } f\}}
\]

By the definition of field lookup, \(\text{typeOff}(f)\) remains unchanged, i.e. \(\text{typeOff}(f) = (c,t)\). Thus, by \((E-\text{SET})\), \(\mu \vdash E_2[e'] : \sigma_2 \cup \{\text{write } f\};\)

**Case** \(loc.f = -;\). The last step for \(E_2[e]\) should be \((E-\text{SET})\):

\[
\frac{\text{typeOff}(f) = (c,t) \quad \mu \vdash e : \sigma_0}{\mu \vdash E_2[e] : (u, \sigma_0 \cup \{\text{write } f\})}
\]

So \(\mu \vdash E_2[e'] : \sigma_1 \cup \{\text{write } f\};\)

**Case** \(-m(e_1, \ldots, e_n)\). The last step for \(E_2[e]\) should be \((E-\text{CALL-OPEN})\): \(e = loc.f\)

\[
\begin{align*}
e &= loc.f & \mu(loc) &= [c.F.E] \\
E &= \{m_i \mapsto \sigma_i \mid 1 \leq i \leq n\} & \exists i \text{ s.t. } (\exists \mathcal{E} = \emptyset \text{open } f m \sigma'' \in \sigma_i) \\
\text{typeOff}(f) &= (c, \emptyset \text{open } c_0) & \mu \vdash loc.f : \sigma_0' \quad (\forall i \in \{1..n\} \implies \mu \vdash e_i : \sigma_i') \\
\Pi &\vdash E_2[e] : \{\text{open } f m \sigma'' \} \cup \bigcup_{i=0}^{n} \sigma_i'
\end{align*}
\]

By \((\text{GET})\), \(e' = loc;\):

\[
\begin{align*}
e' &= loc & \mu(loc') &= [c'.F'.E'] \\
E(m) &= \sigma_0'' & (\forall i \in \{1..n\} \implies \mu \vdash e_i : \sigma_i'') \\
\Pi &\vdash E_2[e'] : \bigcup_{i=0}^{n} \sigma_i''
\end{align*}
\]
Because \( \mu \vdash \circ \), by Definition 11 and Definition 9, we have \( \sigma''_0 \subseteq \sigma'' \), \( \forall i \in \{1..n\} e_i \) does not change, thus \( \sigma_i' = \sigma''_i \). Therefore, the claim holds.

**Case** \( \text{loc}.m(v_1,\ldots,v_{p-1},-,e_{p+1},\ldots,e_n) \). Here \( p \in \{1..n\} \). The last step for \( E_2[e] \) must be (E-CALL-LOC):

\[
\begin{align*}
\mu(\text{loc}) &= \{c.F.E\} \\
E(m) &= \sigma'' \\
\mu(e) &= \sigma_0 \\
(\forall i \in \{(p+1)..n\} :: \mu \vdash e_i : \sigma''_i) \\
\mu \vdash E_2[e] : \sigma_0 \cup \sigma'' \cup \bigcup_{i=(p+1)}^{n} \sigma_i 
\end{align*}
\]

By (E-CALL-LOC), \( \mu \vdash E_2[e'] : \sigma_1 \cup \sigma'' \cup \bigcup_{i=(p+1)}^{n} \sigma_i \).

**Using the lemmas.** We will prove, in each case, that \( \mu \equiv \mu' \), except for (Set-Open).

To prove Theorem 11.2 in each reduction case, let \( e = E[e_0], e' = E[e_1], \mu = e_0 : \sigma_0 \) and \( \mu' \vdash e_1 : \sigma_1 \). Given that \( \mu \equiv \mu' \), by Lemma 2 and Lemma 1 to prove (a), it suffices to prove \( \sigma_1 \subseteq \sigma_0 \). We divide the cases into 3 categories: in the first category, some variables (var) will be replaced by actual values (v), in Section 11.2 the cases, in the second category, access the store, in Section 11.2 and the other cases are listed right below. Here the rule leaves no dynamic trace, and (b) holds.

**New Object.** Here \( e = E[new(c)\mid f \in \text{fields}(e)] \), \( e' = E[\text{loc}] \), where \( \text{loc} \notin \text{dom}(\mu) \), \( \mu' = \{\text{loc} \mapsto [c.f \mapsto \text{null} \mid f \in \text{fields}(e)]\} \oplus \mu \). Because this rule does not change any object, \( \mu \equiv \mu' \). Also \( \mu \vdash \text{new}(c) : \emptyset \) and \( \mu \vdash \text{loc} : \emptyset \), and (a) holds.

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

**Lemma 3.** [Substitution effect] If \( \mu \vdash e : \sigma \), then there is some \( \sigma' \), such that \( \mu \vdash [v_1/var_1,\ldots,v_n/var_n]e : \sigma' \), for all values \( v_i \) and free variables \( \text{var}_i \), and \( \sigma' \subseteq \sigma \).

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

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

The case for (E-DEFINE) follows directly from the IH.

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

**E-DEFINE.** Here \( e = c \var = e_1; e_2 \). The last effect derivation step is:

\[
\begin{align*}
\mu \vdash e_1 : \rho_1 & \quad \mu \vdash e_2 : \rho_2 \\
\mu \vdash c \var = e_1; e_2 : \rho_1 \cup \rho_2
\end{align*}
\]
Let \( e' = \boxed{\text{var}} e \) for \( i \in \{1, 2\} \), \( \boxed{\text{var}} e = c \), \( \boxed{\text{var}} e' = e'_1 \). We show that \( \mu \vdash \boxed{\text{var}} e : \sigma'_1 \cup \sigma'_2 \), where \( \sigma'_1 \subseteq \sigma_1 \) and \( \sigma'_2 \subseteq \sigma_2 \). The above is truth by IH.

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

\[
\mu(\text{loc}) = [\text{c.F.E}] \\
E = \{m_i \mapsto \sigma_i \mid 1 \leq i \leq n\} \\
\exists i \text{ s.t. } (\exists e = \boxed{\text{open}} f m \sigma' \text{ s.t. } e \in \sigma_i) \\
typeOfF(f) = (d, \boxed{\text{open}} e_0) \\
\mu \vdash \text{loc } f : \sigma_0 \\
(\forall i \in \{1..n\} :: \mu \vdash e_i : \sigma_i)
\]

\[
\mu \vdash \text{loc } f.m(e_1, \ldots, e_n) : \boxed{\text{open}} f m \sigma'' \cup \bigcup_{i=0}^n \sigma_i
\]

Let \( e'_1 = \boxed{\text{var}} e \) for \( i \in \{1..n\} \), \( \boxed{\text{var}} e = \text{loc } f.m(\varphi) \). We show that \( \mu \vdash \boxed{\text{var}} e : \boxed{\text{open}} f m \sigma'' \cup \bigcup_{i=0}^n \sigma'_i \), where \( \forall i \in \{0..n\} \sigma'_i \subseteq \sigma_i \). Because \( \text{loc } f \) has no free variable, \( \sigma'_0 = \sigma_0 \) and \( \boxed{\text{open}} f m \sigma'' \) are unchanged. Also by IH \( \forall i \in \{1..n\} :: \mu \vdash e'_i : \sigma'_i \) and \( \sigma'_i \subseteq \sigma_i \). Thus the claim holds.

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

\[
\mu(\text{loc}) = [\text{c.F.E}] \\
E(m) = \sigma_0 \\
(\forall i \in \{1..n\} :: \mu \vdash e_i : \sigma_i)
\]

\[
\mu \vdash \text{loc } m(e_1, \ldots, e_n) : \bigcup_{i=0}^n \sigma_i
\]

Let \( e'_i = \boxed{\text{var}} e \) for \( i \in \{1..n\} \), then \( \boxed{\text{var}} e = \text{loc } m(\varphi) \). We show that \( \mu \vdash \boxed{\text{var}} e : \bigcup_{i=0}^n \sigma'_i \), where \( \forall i \in \{0..n\} \sigma'_i \subseteq \sigma_i \). Clearly, \( \sigma'_0 = \sigma_0 \). By IH \( \forall i \in \{1..n\} :: \mu \vdash e'_i : \sigma'_i \) and \( \sigma'_i \subseteq \sigma_i \).

(E-SET). Here \( e = \text{loc } f = e_0 \). The last derivation step is:

\[
\mu(\text{loc}) = [\text{c.F.E}] \\
E(f) = (c, t) \\
\mu \vdash \text{loc } f = e_0 : \sigma_0 \subseteq \{\boxed{\text{write }} f\}
\]

Now \( \boxed{\text{var}} e = (\text{loc } f = \boxed{\text{var}} e_0) \). By IH, \( \mu \vdash \boxed{\text{var}} e_0 : \sigma'_0 \subseteq \sigma_0 \). By the definition of typeOfF, the result of typeOfF(f) remains unchanged, i.e. \( \text{typeOfF}(f) = \text{typeOf}(f) \). Therefore \( \Pi \vdash \boxed{\text{var}} e : \sigma'_0 \cup \{\text{write } f\} \), and it holds.

Thus, for all possible derivations of \( \mu \vdash e : \sigma \) we see that \( \mu \vdash \boxed{\text{var}} e : \sigma' \) for some \( \sigma' \subseteq \sigma \).

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

Method Call:. Here \( e' = \boxed{\text{var}} \boxed{\text{var}} e_1, e'_1 = \boxed{\text{var}} \boxed{\text{var}} e_2, \mu(\text{loc}) = [\text{c.F.E}] \). Let \( \mu \vdash \text{loc } m(\varphi) : \sigma_0, \mu(\text{loc}) = \bigcup_{i=0}^n \sigma_i \). By Lemma 3 \( \sigma_1 \subseteq \sigma_3 \). By \( \mu \vdash \boxed{\text{var}} e : \sigma' \), by \( \boxed{\text{var}} e \)

Local Declaration:. Here \( e = \boxed{\text{var}} e_1, e'_1 = \boxed{\text{var}} e_1, \mu(\text{loc}) = \bigcup_{i=0}^n \sigma_i \). By \( \mu \vdash \boxed{\text{var}} e : \sigma' \), by \( \boxed{\text{var}} e_1 \)

Fields Access. In this subsection, we first state a lemma for the effect relationship between an expression and its subexpression.
The following lemma says that the effect $\sigma$ of subexpression $e$ is a subset $\subseteq$ of the effect $\sigma'$ of its entire expression $E[e]$.

**Lemma 4.** [Subexpression effect containment] If $\mu \vdash e : \sigma$ and $\mu \vdash E[e] : \sigma'$, then $\sigma \subseteq \sigma'$.

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

**Using the lemma.** We now prove cases for field accesses.

**Field Get:** Here $e = E[\text{loc}.f]$, $e' = E[v]$, where $\mu(\text{loc}) = [u.F.E]$, $F(f) = v$, $\mu' = \mu$ and $\mu \equiv \mu'$. Because $\mu \vdash \text{loc}.f : \{\text{read} f\}$, and $\mu' \vdash v : \emptyset$, (a) holds. Finally, $\eta = (\text{rd}, \text{loc}, f)$, and $\eta \approx \{\text{read} f\} \subseteq \sigma$, by Lemma 4.

**Field Set:** Here $e = E[\text{loc}.f = v]$, $e' = E[v]$, $\mu' = \mu \oplus (\text{loc} \mapsto o)$, and $o = [u.F \oplus (f \mapsto v).E]$, where $\mu(\text{loc}) = [u.F.E]$ and $\text{typeOf}(f) = (c, t)$ for some $t$. The field is not an open field, and by the function update, it does not update any effect, and $\mu \equiv \mu'$. To see $\mu \vdash E[v] : \sigma' \subseteq \sigma$, we have $\mu \vdash \text{loc}.f = v : \{\text{write} f\}$, and $\mu' \vdash v : \emptyset$, thus $\sigma' \subseteq \sigma$. Finally, $\eta = (\text{wt}, \text{loc}, f)$, and $\eta \approx \{\text{write} f\} \subseteq \sigma$, by Lemma 4.

**Field Set Open:** Here $e = E[\text{loc}.f = v]$, $e' = E[v]$, where $\mu_0 = \mu \oplus (\text{loc} \mapsto [c.(F \oplus (f \mapsto v)).E])$, and $\mu' = \text{update}(\mu_0, \text{loc}, f, v)$. $\bot$ is the maximum, and effect $\sigma \subseteq (\bot)$. ![End of proof]

### 11.3 Type Soundness

In this section, we prove the standard type preservation property. Type rules omitted in Section 3 are in Figure 7. 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. Also, we need to ensure that during the evaluation, all the expressions have proper types (Definition 14). Finally, we state the standard lemmas [19] (Lemma 5, Lemma 6, Lemma 7, and Lemma 8).

**Definition 12.** [Environment-store consistency] A store $\mu$ is consistent with a type environment $\Pi$, written $\mu \approx \Pi$, if all of the following hold:

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

**Definition 13.** [Environment enlargement] Let $\Pi$ and $\Pi'$ be two type environments. We write $\Pi \prec \Pi'$, if dom($\Pi'$) $\subseteq$ dom($\Pi$) and $\forall a \in$ dom($\Pi$), if $\Pi(a) = t$, 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 OpenEffect program, we can use an ever increasing type environment to type check the expressions.
\[\Pi \vdash \text{loc}: \{\text{read } f\}\]

\[\Pi \vdash \text{loc}: \{\text{write } f\}\]

\[\Pi \vdash \text{loc}: \{\text{open } m\}: \{\text{open } f \text{ m } \emptyset \} \cup \bigcup_{i=0}^{n} \sigma_{i}\]

\(\Pi := \{t_i \mapsto t_i\}_{i \in \mathbb{N}}\)

where \(r \in (\mathcal{X} \cup \{\text{this}\} \cup \forall)\)

\(\Pi \vdash \text{loc}: \{\text{open } m \emptyset\} \; \text{“type environments”}\)

\[\Pi \vdash \text{loc}: \{\text{call } \text{meth}(c_0, m)\}: (t, \{\text{open } f \text{ m } \emptyset\} \cup \bigcup_{i=0}^{n} \sigma_{i})\]

\[\Pi \vdash \text{loc}: \{\emptyset\}\]

\[\Pi \vdash \text{loc}: \{\text{call } \text{meth}(c_0, m)\}\]

\[\Pi \vdash \text{findMeth}(c_0, m) = (c_1, t, m (\overline{\text{var}}) \{e_n+1\}, \sigma)\]

\[\Pi \vdash \text{findMeth}(c_0, m) = (c_1, t, m (\overline{\text{var}}) \{e_n+1\}, \sigma)\]

Fig. 7. Type and effect rules for \text{loc}.

**Definition 14.** [Well-typed configuration] A configuration \(\Sigma = (e, \mu)\) is well-typed in \(\Pi\), written \(\Pi \vdash \Sigma\), if \(\Pi \vdash e : (t, \sigma)\) and \(\mu \approx \Pi\).

**Lemma 5.** [Substitution] If \(\Pi, \overline{\text{var}}: t \vdash e : (t, \sigma)\) and \(\forall i \in \{1..n\}, \Pi \vdash v_i : (s_i, \sigma')\) where \(s_i < : t\), then \(\Pi' \vdash \overline{\text{var}}: e : (s, \sigma')\) for some \(s < : t\) and some \(\sigma'\).

**Proof:** To simplify the notations, we let \(\Pi' = \Pi, \overline{\text{var}}: t\). We prove it by structural induction on the derivation of \(\Pi \vdash e : (t, \sigma)\) and by cases, based on the last step in that derivation. The base cases include \((\text{T-NEW}), (\text{T-NUL}), (\text{T-LOC}), (\text{T-GET}), (\text{T-GET-LOC})\), and \((\text{T-VAR})\). The first five of these cases are obvious: \(e\) has no variables, \(s = t\). In the \((\text{T-VAR})\) case, \(e = \text{var}\), and there are two subcases. If \(\text{var} \notin \{\text{var}_1, \ldots, \text{var}_n\}\), then \(\Pi' (\text{var}) = \Pi (\text{var}) = t\) and the claim holds. Otherwise, suppose \(\text{var} = \text{var}_k\). Then \(\overline{\text{var}}: e = v_k\) and, by the assumptions of the lemma, \(\Pi \vdash \overline{\text{var}}: e : (s_k, \emptyset)\) and \(s_k \ll t_k = t\).

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

\[\text{isClass}(c) \quad \Pi' \vdash e'_1 : (t_1, \sigma_1) \quad \Pi', \text{var} : c \vdash e_2 : (t, \sigma_2) \quad t_1 \ll : c\]

\(\Pi' \vdash c \text{ var} = e'_1; e'_2 : (t, \rho_1 \cup \rho_2)\)

Let \(e''_1 = \overline{\text{var}} e'_1\) and \(e''_2 = \overline{\text{var}} e'_2\), then \(\overline{\text{var}}: e = c \text{ var} = e''_1; e''_2\). We show that \(\Pi \vdash \overline{\text{var}}: e : (t, \sigma')\). By \(\text{IH}\), \(\Pi' \vdash e'_1 = (t'_1, \sigma'_1)\), where \(t'_1 \ll : t_1 \ll : c\), and \(\Pi \vdash e'_2 = (t', \sigma'_2)\), where \(t' \ll : t\). Therefore, the claim holds.
(T-CALL-OPEN). Here \( e = \texttt{this.	extit{f}.m}(\overline{e}) \). The last type derivation step has the following form:

\[
\begin{align*}
\Pi' \vdash e'_{0} : (c_0, \sigma_0) \\
\text{typeOfF}(f) = (d, \texttt{open } c_0) \\
\text{findMeth}(c_0, m) = (c_1, t, m (u_1 \texttt{var} 1, \ldots, u_n \texttt{var} n) (e_{n+1}) , \sigma_2) \\
\forall i \in \{1..n\} :: \Pi' \vdash e'_{i} : (u'_i, \sigma_i) \land u'_i <: u_i \\
\Pi' \vdash e'_{0} . m(e'_1, \ldots, e'_n) : (t, \{\texttt{open } f m \emptyset\} \cup \bigcup_{i=0}^{n} \sigma_i)
\end{align*}
\]

Let \( e''_{0} = [\overline{\texttt{var}}]e'_{0} \) for \( i \in \{0..n\} \), then \( [\overline{\texttt{var}}]e = e''_{0} . m(\overline{e'}) \). We show that \( \Pi \vdash [\overline{\texttt{var}}]e : (t, \sigma') \) for some \( \sigma' \). By IH, \( \Pi \vdash e'_{0} : (c_2, \sigma''_0) \) where \( c_2 <: c_0 \). If \((c_1, t, m (\overline{\texttt{var}}) (e_{n+1}), \sigma_2) = \text{findMeth}(c_0, m) \) and \((c_2, t_2, m (\overline{\texttt{var}}) (e''_{n+1}), \sigma_3) = \text{findMeth}(c_2, m) \), by the definitions of \text{findMeth} and \text{override}, \( t_2 = t \). Also, by IH, \( \forall i \in \{1..n\} :: \Pi \vdash e''_{i} : (u'_i, \sigma_i) \land u'_i <: u_i \). Finally, \( \forall i \in \{1..n\} :: u''_{i} <: u_i \), by transitivity, the claim holds.

(T-CALL). Here \( e = e'_{0} . m(\overline{e'}) \). The last type derivation step has the following form:

\[
\begin{align*}
\Pi' \vdash e'_{0} : (u'_0, \sigma_0) \\
\forall i \in \{1..n\} :: \Pi' \vdash e'_{i} : (u'_i, \sigma_i) \land u'_i <: u_i \\
\Pi' \vdash e'_{0} . m(e'_1, \ldots, e'_n) : (t, \bot)
\end{align*}
\]

Let \( e''_{0} = [\overline{\texttt{var}}]e'_{0} \) for \( i \in \{0..n\} \), then \( [\overline{\texttt{var}}]e = e''_{0} . m(\overline{e'}) \). We show that \( \Pi \vdash [\overline{\texttt{var}}]e : (t, \sigma') \) for some \( \sigma' \). By IH, \( \Pi \vdash e'_{0} : (u'_0, \sigma_0) \), where \( u'_0 <: u_0 \). By the definitions of \text{findMeth} and \text{override}, if \((c_1, t, m (\overline{\texttt{var}}) (e'_{n+1}), \sigma''_0) = \text{findMeth}(u'_0, m) \) and \((c_2, t_2, m (\overline{\texttt{var}}) (e''_{n+1}), \sigma''_0) = \text{findMeth}(u''_0, m) \), then \( t_2 = t \). Also, by IH, \( \forall i \in \{1..n\} :: \Pi \vdash e''_{i} : (u''_i, \sigma_i) \land u''_i <: u'_i \). Finally, \( \forall i \in \{1..n\} :: u''_{i} <: u_i \), by transitivity, the claim holds.

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

\[
\begin{align*}
\Pi' \vdash \text{loc} : (u'_0, \sigma_0) \\
\forall i \in \{1..n\} :: \Pi' \vdash e'_{i} : (u'_i, \sigma_i) \land u'_i <: u_i \\
\Pi' \vdash \text{loc} . m(e'_1, \ldots, e'_n) : (t, \bot)
\end{align*}
\]

Let \( e''_{0} = [\overline{\texttt{var}}]e'_{0} \) for \( i \in \{1..n\} \), then \( [\overline{\texttt{var}}]e = e''_{0} . m(\overline{e'}) \). We show that \( \Pi \vdash [\overline{\texttt{var}}]e : (t, \sigma') \) for some \( \sigma' \). By IH, \( \forall i \in \{1..n\} :: \Pi \vdash e''_{i} : (u''_i, \sigma_i) \land u''_i <: u'_i \). Finally, \( \forall i \in \{1..n\} :: u''_{i} <: u_i \), by transitivity, the claim holds.

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

\[
\begin{align*}
\text{typeOfF}(f) = (d, \texttt{open } c_0) \\
\Pi' \vdash \text{loc.f} : (c_0, \sigma_0) \\
\text{findMeth}(c_0, m) = (c_1, t, m (u_1 \texttt{var} 1, \ldots, u_n \texttt{var} n) (e_{n+1}) , \sigma_2) \\
\forall i \in \{1..n\} :: \Pi' \vdash e'_{i} : (u'_i, \sigma_i) \land u'_i <: u_i \\
\Pi' \vdash \text{loc.f} . m(e'_1, \ldots, e'_n) : (t, \{\texttt{open } f m \emptyset\} \cup \bigcup_{i=0}^{n} \sigma_i)
\end{align*}
\]

Let \( e''_{0} = [\overline{\texttt{var}}]e'_{0} \) for \( i \in \{1..n\} \), then \( [\overline{\texttt{var}}]e = \texttt{loc.f.m}(\overline{e'}) \). We show that \( \Pi \vdash [\overline{\texttt{var}}]e : (t, \sigma') \) for some \( \sigma' \). By IH, \( \forall i \in \{1..n\} :: \Pi \vdash e''_{i} : (u''_i, \sigma_i) \land u''_i <: u'_i \). Finally, \( \forall i \in \{1..n\} :: u''_{i} <: u_i \), by transitivity and thus the claim holds.
(T-SET). Here $e = \text{this}.f = e'_1$. The last derivation step is:

$$\Pi' \vdash \text{this}: (c, \emptyset) \quad \text{typeOfF}(f) = (d, u) \quad c :< d \quad \Pi' \vdash e'_1 : (t, \sigma_1) \quad t :< u$$

$$\Pi' \vdash \text{this}.f = e'_1 : (t, \sigma_1 \cup \{\text{write} \ f\})$$

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

(T-SET-LOC). Here $e = \text{loc}.f = e'_1$. The last derivation step is:

$$\Pi' \vdash \text{loc} : (c, \emptyset) \quad \text{typeOfF}(f) = (d, u) \quad c :< d \quad \Pi' \vdash e'_1 : (t, \sigma_1) \quad t :< u$$

$$\Pi' \vdash \text{loc}.f = e'_1 : (t, \sigma_1 \cup \{\text{write} \ f\})$$

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

(T-SET-OPEN). Here $e = \text{this}.f = e'_1$. The last step is:

$$\Pi'(\text{this}) = c \quad \text{typeOfF}(f) = (d, \emptyset \text{open} u) \quad c :< d \quad \Pi' \vdash e'_1 : (t, \sigma_0) \quad t :< u$$

$$\Pi' \vdash \text{this}.f = e'_1 : (t, \{\bot\})$$

Now $[\overline{\text{var}}]e = (\text{this}.f = [\overline{\text{var}}]e'_1)$. By IH, $\Pi \vdash [\overline{\text{var}}]e'_1 : (u'_2, \sigma_1)$, where $u'_2 :< t$. By the definition of $\text{typeOfF}$, its result does not change. By transitivity $t' = u'_2 :< t$. There are two subcases: 1) if $\text{this} \notin \{\text{var}_1, ..., \text{var}_n\}$, then nothing changes. Otherwise, suppose $\text{this} = \text{var}_k$. Then $[\overline{\text{var}}]e = v_k, f = [\overline{\text{var}}]e'$ and, by (T-SET-OPEN-LOC), $\Pi \vdash [\overline{\text{var}}]e : (t', \{\perp\})$, $t' :< t$ and the claim holds.

(T-SET-OPEN-LOC). Here $e = \text{loc}.f = e'_1$. The last step is:

$$\Pi'(\text{loc}) = c \quad \text{typeOfF}(f) = (d, \emptyset \text{open} u) \quad c :< d \quad \Pi' \vdash e'_1 : (t, \sigma') \quad t :< u$$

$$\Pi' \vdash \text{loc}.f = e'_1 : (t, \{\perp\})$$

Now $[\overline{\text{var}}]e = \text{loc}.f = [\overline{\text{var}}]e'_1)$. By IH, $\Pi \vdash [\overline{\text{var}}]e'_1 : (u'_2, \sigma'_2)$, where $u'_2 :< t$. By the definition of $\text{typeOfF}$, its result does not change. By transitivity $t' = u'_2 :< t$. Therefore $\Pi \vdash [\overline{\text{var}}]e : (t', \{\perp\})$, $t' :< t$ and the claim holds.

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

**Lemma 6.** [Environment extension] If $\Pi \vdash e : (t, \sigma)$ and $a \notin \text{dom}(\Pi)$, then $(\Pi, a : t') \vdash e : (t, \sigma)$.

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

\[ \text{Lemma 7.} \] If \( \Sigma = (\mathbb{E}[e], \mu), \Sigma' = (\mathbb{E}[e'], \mu'), \Sigma \mapsto \Sigma', \Pi \vdash \mathbb{E}[e] : (t, \sigma), \Pi \vdash e : \langle t', \sigma' \rangle \text{ and } \Pi \vdash e' : \langle t', \sigma'_0 \rangle, \text{ then } \Pi \vdash \mathbb{E}[e'] : (t, \sigma_0) \text{ for some } \sigma_0. \]

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

The cases for \( (\text{loc} = -) \), \( (\text{open} = -) \), and \( (\text{var} = -) \) follow directly from the induction hypothesis.

**Case \(-m(\sigma)\)**. The last step for \( \mathbb{E}_2[e] \) could be

\[
\begin{align*}
&\text{Case} 1: \text{(T-CALL)}: \\
&\Pi \vdash e : (u, \sigma_0) \quad \forall i \in \{1, n\} :: \Pi \vdash e_i : (t'_i, \sigma'_i) \land t'_i < : t_i \\
&\Pi \vdash \mathbb{E}_2[e] : (t, \{\bot\})
\end{align*}
\]

Here \( \text{findMeth}(u', m) = (c_2, t, m (\text{var}) e'_{n+1}, \sigma'''_i) \), by the definitions of \( \text{override} \) and \( \text{findMeth} \), where \( c_2 < c_1 \), so (T-CALL) gives \( \Pi \vdash \mathbb{E}_2[e'] : (t, \sigma') \); or

\[
\begin{align*}
&\text{Case 2: \text{(T-CALL-OPEN)}}: \\
&e = \text{this} \cdot f \text{ typeOfF}(f) = (d, \text{open} t') \\
&\Pi \vdash e : (t', \sigma') \quad \forall i \in \{1, n\} :: \Pi \vdash e_i : (t'_i, \sigma'_i) \land t'_i < : t_i \\
&\Pi \vdash \mathbb{E}_2[e] : (t, \{\text{open} \ f \ m \ 0\} \cup \sigma' \cup \bigcup_{i=1}^{n} \sigma''_i)
\end{align*}
\]

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

\[
\text{Case 3: \text{(T-CALL-OPEN-LOC)}}: \\
&e = \text{loc}. f \text{ typeOfF}(f) = (d, \text{open} t') \\
&\Pi \vdash e : (t', \sigma') \quad \forall i \in \{1, n\} :: \Pi \vdash e_i : (t'_i, \sigma'_i) \land t'_i < : t_i \\
&\Pi \vdash \mathbb{E}_2[e] : (t, \{\text{open} \ f \ m \ 0\} \cup \sigma' \cup \bigcup_{i=1}^{n} \sigma''_i)
\]

\[2\text{ Formulation of the proof is similar Flatt’s work [19]}.\]
It must be the case that \( e' = \text{loc}' \). From the statement of the lemma, we have \( \Pi \vdash \text{loc}' : (t', \sigma_0') \). Also the results of \( \text{findMeth} \) does not change, therefore, by (T-CALL), the type of the expression is \( t \).

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

\[
\Pi \vdash \text{loc} : (u, \emptyset) \quad (c_1, t, m (t_1 \text{var}_1, \ldots, t_n \text{var}_n) (e_{p+1}), \sigma_0'') = \text{findMeth}(u, m) \\
(\forall j \in \{1, \ldots, p \}) \quad \Pi \vdash v_j : (t'_j, \emptyset) \land t'_j < : t_j \\
(\forall j \in \{p+1, \ldots, n \}) \quad \Pi \vdash e_j : (t'_j, \sigma'_j) \land t'_j < : t_j \\
\Pi \vdash e_p : (t', \sigma') \land t' < : t_p \\
\Pi \vdash \mathbb{E}_2[e] : (t, \{ \bot \})
\]

We have \( \Pi \vdash e' : (t', \sigma_0') \) and \( t' < : t_p \) and other parts of conditions do not change. The claim holds.

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

\[
\Pi \vdash e : (c, \sigma') \quad \text{typeOfF}(f) = (d, t) \quad c < : d \\
\Pi \vdash e_2 : (t_2, \sigma_2) \quad t_2 < : t \\
\Pi \vdash \mathbb{E}_2[e] : (t_2, \{ \bot \})
\]

By the definition of field lookup, \( \text{typeOfF}(f) \) does not change. Thus, by (T-SET), \( \Pi \vdash \mathbb{E}_2[e'] : (t_2, \sigma_0) \); or

<2> (T-SET-OPEN):

\[
\Pi \vdash e = \text{this} \quad \Pi(\text{this}) : t' \\
\text{typeOfF}(f) = (d, \emptyset \text{open } t) \quad t' < : d \\
\Pi \vdash e_2 : (t_2, \sigma_2) \quad t_2 < : t \\
\Pi \vdash \mathbb{E}_2[e] : (t_2, \{ \bot \})
\]

The only possibility is that \( e' = \text{loc} \), for some \( \text{loc} \). By the statement of this lemma \( \Pi \vdash e' : (t', \sigma_0') \), i.e., \( \Pi \vdash \text{loc} : (t', \sigma_0') \), thus by (T-SET-OPEN-LOC), the claim holds.

**Case \( \text{loc} = f \).** The last step for \( \mathbb{E}_2[e] \) could be

<1> (T-SET-LOC):

\[
\Pi(\text{loc}) : c \\
\Pi \vdash e_2 : (t_1, \sigma) \quad t_1 < : t \\
\Pi \vdash \mathbb{E}_2[e] : (t_1, \{ \bot \})
\]

By the definition of field lookup, \( \text{typeOfF}(f) \) does not change. Thus, by (T-SET-LOC), \( \Pi \vdash \mathbb{E}_2[e'] : (t_1, \sigma_0) \); or

<2> (T-SET-OPEN-LOC):

\[
\Pi(\text{loc}) : t' \\
\Pi \vdash e : (t_1, \sigma) \quad t_1 < : t \\
\Pi \vdash \mathbb{E}_2[e] : (t_1, \{ \bot \})
\]

By the definition of field lookup, \( \text{typeOfF}(f) \) does not change. Therefore, thus by (T-SET-OPEN-LOC), the claim holds.

**Case \( \neg f \).** The last step for \( \mathbb{E}_2[e] \) must be (T-GET):

\[
\Pi \vdash e : (c, \sigma_1) \quad \text{typeOfF}(f) = (d, t) \quad c < : d \\
\Pi \vdash \mathbb{E}_2[e] : (t, \sigma_1 \cup \{ \text{read } f \})
\]

The result of \( \text{typeOfF} \) does not change. Thus, by (T-GET), \( \Pi \vdash \mathbb{E}_2[e'] : (t, \sigma_0) \)
Lemma 8. [Replacement with subtyping] If $\Sigma \rightarrow \Sigma'$, $\Sigma = (E[e], \mu)$, $\Sigma' = (E[e'], \mu')$, $\Pi \vdash E[e] : (t, \sigma)$, $\Pi \vdash e : (u, \sigma_0)$, and $\Pi \vdash e' : (u', \sigma_1)$ and $u' < u$, then $\Pi \vdash E[e'] : (t', \sigma')$ where $t' < t$.

Proof: Proof is by induction on the size of the evaluation context $E$. Size of the $E$ refers to the number of recursive applications of the syntactic rules necessary to create $E$. In the base case, $E$ has size zero, $E = \perp$, and $t' = u' < u = t$. 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 $\Pi \vdash E_2[e'] : (s, \sigma)$ implies that $\Pi \vdash E_2[e_1] : (s', \sigma')$, for some $s' < s$, and the claim holds by IH. The cases for $(loc0, f = -)$, $(--; e2)$, and $(c \ var = --; e2)$ follow directly from IH.

Case $-m(\overline{f})$. The last step for $E_2[e_1]$ could be

<1> (T-CALL):

$$\Pi \vdash e : (t', \sigma') \quad (c_1, t, m \ (\overline{\text{var}} \ (e_{n+1}), \sigma_1) = \text{findMeth}(t', m)$$

$$\forall i \in \{1..n\} : \Pi \vdash e_i : (t'_i, \sigma'_i) \land t'_i < t_i$$

$$\Pi \vdash E_2[e] : (t, \{ \perp \})$$

We have $\text{findMeth}(t', m) = (c_2, t, m \ (\overline{\text{var}} \ (e_{n+1}), \sigma_1)$, by the definitions of override and $\text{findMeth}$, $c_2 < c_1$, so (T-CALL) gives $\Pi \vdash E_2[e'] : (t, \{ \perp \})$; or

<2> (T-CALL-OPEN):

$$e = \text{this} \cdot f$$

$$\Pi \vdash e : (u, \sigma_2) \quad \forall i \in \{1..n\} : \Pi \vdash e_i : (t'_i, \sigma'_i) \land t'_i < t_i$$

$$\Pi \vdash E_2[e] : (t, \{ \text{open} \ f \ m \ \emptyset \} \cup \sigma_2 \cup \bigcup_{i=1}^{n} \sigma'_i)$$

It must be the case that $e' = \text{loc} \cdot f$. From the statement of the lemma, we have $\Pi \vdash \text{loc} \cdot f : (u', \sigma_1)$, where $u' < u$. By the definitions of override and $\text{findMeth}$, $\text{findMeth}(t', m) = (c_2, t, m \ (\overline{\text{var}} \ (e_{n+1}), \sigma'_1)$. The result of typeOff does not change, so the type of the expression is $t$, by (T-CALL-OPEN-LOC);

<3> (T-CALL-OPEN-LOC):

$$e = \text{loc} \cdot f$$

$$\Pi \vdash e : (u, \sigma_0) \quad \forall i \in \{1..n\} : \Pi \vdash e_i : (t'_i, \sigma'_i) \land t'_i < t_i$$

$$\Pi \vdash E_2[e] : (t, \{ \text{open} \ f \ m \ \emptyset \} \cup \sigma_0 \cup \bigcup_{i=1}^{n} \sigma'_i)$$

It must be the case that $e' = \text{loc} \cdot f$. From the statement of the lemma, we have $\Pi \vdash \text{loc} \cdot f : (u', \sigma'_0)$, where $u' < u$. By the definitions of override and $\text{findMeth}$, $\text{findMeth}(u', m) = (c_2, t, m \ (\overline{\text{var}} \ (e_{n+1}), \sigma'_0)$. Therefore, by (T-CALL), the type of the expression is $t$.  

---

3 Formulation of the proof is similar to Clifton’s work [14] and Flatt’s work [19].
Case $v_0.m(v_1, \ldots, v_{p-1}, \neg e_{p+1}, \ldots, e_n)$. Here $p \in \{1..n\}$. The last step for $E_2[e]$ must be (T-CALL-LOC):

$$\Pi \vdash \text{loc} : (u_0, \emptyset) \quad \text{loc}, t, m \dfrac{}{(T \forall f) (e_{n+1}, \sigma'')} = \text{findMeth}(u_0, m)$$

\[
\begin{align*}
(\forall i \in \{1..(p-1)\} : \Pi \vdash v_i : (u'_i, \emptyset) & \quad (\forall i \in \{(p+1)..n\} : \Pi \vdash e_i : (u'_i, \sigma'') \\
\Pi \vdash e : (u, \sigma_0) & \quad (\forall i \in \{1..n\} \setminus \{p\} : u'_i : \vdash t_i) \quad u : <: t_p \\
\Pi \vdash E_2[e] : (t, \{\bot\}) & \quad \Pi \vdash E_2[e'] : (t, \emptyset)
\end{align*}
\]

Now $u' :<: u :<: t_p$, so by (T-CALL), $\Pi \vdash E_2[e'] : (t, \{\bot\})$.

Case $\neg f = e_2$. The last step for $E_2[e]$ could be

<1> (T-SET):

$$\Pi \vdash e : (u, \sigma_0) \quad \text{typeOfF}(f) = (d, t_0) \quad u : <: d \quad \Pi \vdash e_2 : (t, \sigma_2) \quad t : <: t_0$$

$$\Pi \vdash E_2[e] : (t, \sigma_0 \cup \sigma_2 \cup \{\text{write } f\})$$

Now $u' :<: u :<: d$. The result of $\text{typeOfF}(f)$ does not change. Thus, by (T-SET), $\Pi \vdash E_2[e'] : (t, \sigma')$; or

<2> (T-SET-OPEN):

$$e = \text{this} \quad \Pi(\text{this}) : u$$

$$\text{typeOfF}(f) = (d, @\text{open } t') \quad u : <: d \quad \Pi \vdash e_2 : (t, \sigma_2) \quad t : <: t'$$

$$\Pi \vdash E_2[e] : (t, \{\bot\})$$

The only possibility is that $e' = \text{loc}$, for some loc. By the statement of this lemma $\Pi \vdash e' : (u', \sigma_1)$, where $u' :<: u :<: d$. Thus by (T-SET-OPEN-LOC), the type is $t$.

Case loc.$f = \neg$. The last step for $E_2[e]$ could be

<1> (T-LOC):

$$\Pi(\text{loc}) : u \quad \text{typeOfF}(f) = (d, t_0) \quad u : <: d \quad \Pi \vdash e : (t, \sigma_0) \quad t : <: t_0$$

$$\Pi \vdash E_2[e] : (t, \sigma_0 \cup \{\text{write } f\})$$

Now $t' :<: t :<: t_0$. The result of $\text{typeOfF}(f)$ does not change. Thus, by (T-LOC), $\Pi \vdash E_2[e'] : (t', \sigma')$; or

<2> (T-SET-OPEN-LOC):

$$\Pi(\text{loc}) : u$$

$$\text{typeOfF}(f) = (d, @\text{open } t') \quad u : <: d \quad \Pi \vdash e : (t, \sigma_0) \quad t : <: t'$$

$$\Pi \vdash E_2[e] : (t, \{\bot\})$$

Now $t' :<: t :<: t_0$. The result of $\text{typeOfF}(f)$ does not change. Thus, by (T-SET-OPEN-LOC), $\Pi \vdash E_2[e'] : (t', \{\bot\})$.

Case $\neg f$. The last step for $E_2[e]$ must be (T-GET):

$$\Pi \vdash e : (c, \sigma_1) \quad \text{typeOfF}(f) = (d, t) \quad c : <: d$$

$$\Pi \vdash E_2[e] : (t, \sigma_1 \cup \{\text{read } f\})$$

The result of $\text{typeOfF}$ does not change. Thus, by (T-GET), $\Pi \vdash E_2[e'] : (t', \sigma_0)$.
Theorem 3. [Type preservation] If \( \Pi \vdash \Sigma \), where \( \Sigma = \langle e, \mu \rangle \), \( \Sigma \rightarrow \langle e', \mu' \rangle \), and \( \Pi \vdash e : (t, \sigma) \), then there is some \( \Pi' \), \( t' \) and \( \sigma' \) such that

(a) \( (\mu' \approx \Pi') \), i.e. \( \Pi' \vdash \Sigma' \); 
(b) \( \Pi \ll \Pi' \); and 
(c) \( \Pi' \vdash e' : (t', \sigma') \wedge (t' <:\cdot t) \).

Proof: The proof is by cases on the reduction step applied. We prove the first seven second hypothesis of \( \mu \). Now \( \approx \). We prove the first seven second hypothesis of \( \mu \). Now \( \approx \). We now show that \( \Pi' \approx \Pi' \). Because \( \mu \notin \mu' \). Now \( \approx \). Now \( \approx \). We now show that \( \Pi' \approx \Pi' \). Now \( \approx \). We now show that \( \Pi' \approx \Pi' \). Clearly \( \Pi' \approx \Pi' \) and \( \Pi \ll \Pi' \).

We now show that \( \Pi' \vdash \Pi' : (t', \sigma') \) for some \( t' <:\cdot t \) and some \( \sigma' \). We have \( \Pi' \vdash \Pi' : (s, \{ \text{read } f \}) \). 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 \( \Pi \approx \mu \), we have \( \Pi(\Sigma) = u. \). By the second hypothesis of (T-GET), typeOf(\( f \)) = (c, s). By the second hypothesis of (T-GET-OPEN-LOC), typeOf(\( f \)) = (c, open s). Also by \( \Pi \approx \mu \), if \( (a) \mu(v) = [u'.F'.E'] \), then \( \Pi(v) = \mu' \) and \( \mu' : \ll s \); otherwise \( (b) \mu(v) = nil \). In both cases, the type of \( v \) is subtype of \( s \), by Lemma 8 (Replacement with subtyping), \( \Pi \vdash E[v] : (t', \sigma') \).

Field Set. Here \( \mu = \Pi(\Sigma) = u. \). For part 1(a) \( \Pi(\Sigma) = u, \mu(\Sigma) = [u.F.E] \) and \( \Pi = \mu \). For part 1(b) \( dom(F \oplus (f \rightarrow v)) = dom(fields(u)), \) since \( loc.f = v \) is well-typed. For part 1(c) \( rng(F \oplus (f \rightarrow v)) \subseteq rng(F) \cup \{v\} \). Now since \( loc.f = v \) is well-typed, then \( v \in dom(\Pi) \) or \( v = \text{null} \). In the former case, by \( \Pi \approx \mu \), then \( v \in dom(\mu) \). \( \Pi = \mu \).

Method Call. Here \( e = \Pi(\Sigma) = u. \). For part 1(d) \( \Pi(f) = f \). Part 1(d) holds vacuously for \( f = \text{null} \). Otherwise, \( (F \oplus (f \rightarrow v))(f) = v \), and by (T-SET) or (T-SET-OPEN-LOC) and (T-LOC), \( \Pi(v) = \text{null} \). Parts 2 holds since \( dom(\mu') = dom(\mu) \).

To see \( \Pi \vdash \Pi : (t, \sigma) \), let \( \Pi \vdash \Pi : (t, \sigma) \). By (T-SET) or (T-SET-OPEN-LOC), \( \Pi \vdash \Pi : (t, \sigma) \). Clearly \( \Pi' \approx \mu' \), and \( \Pi \ll \Pi' \).
We now show that $\Pi \vdash e : (t, \sigma)$ implies that $loc.m(\overline{v})$ and all its subterms are well-typed in $\Pi$. By part 1(a) of $\Pi \approx \mu$, $\Pi \vdash loc:(u, \emptyset)$. By the definition of $findMeth$, $u < \overline{u}'$. Let $\Pi \vdash v_i : (u_i, \emptyset)$ $\forall i \in \{1..n\}$ and let $\Pi \vdash loc.m(\overline{v}) : (t_m, \sigma_m)$. This last judgment must be (T-CALL), with $(u', t_m, m(\overline{v} \overline{a}r) \{e_2\}, \sigma_m) = findMeth(u, m)$, where $\forall i \in \{1..n\} : u_i < t_i$. By the definition of the function $findMeth$, rules (T-METHOD) and override, $(\overline{a}r : T, this : u') \vdash e_2 : (u'_m, \sigma_1)$, and $u'_m < t_m$. By Lemma 6 (Environment extension) (and appropriate alpha conversion of free variables in $e_2$), $\Pi, \overline{a}r : T, this : u' \vdash e_2 : (u'_m, \sigma_1)$. By Lemma 5 (Substitution), $\Pi \vdash [loc/this, v/\overline{a}r] e_2 : (u'', \sigma_1)$, for some $u'' < u'_m < t_m$. Finally, Lemma 8 (Replacement with subtyping) gives $\Pi \vdash e'' : (t', \sigma')$ for some $t' < t$.

**Local Declaration.** In this case $e = E[t \overline{a}r = v; e_1]$, $e' = E[e'_1]$, where $e'_1 = [v/\overline{a}r]e_1$ and $\mu' = \mu$. Let $\Pi' = \Pi$. Obviously $\Pi' \approx \mu'$, and $\Pi' \ll \Pi'$. We show $\Pi \vdash E[e'_1] : (t', \sigma')$, for some $t' < : t$. $\Pi \vdash e : (t, \sigma)$ implies that $t \overline{a}r = v; e_1$ and all its subterms are well typed in $\Pi$, let $\Pi \vdash t \overline{a}r = v; e_1 : (s, \sigma_0)$. By (T-DEFINE), $\Pi, \overline{a}r : t \vdash e_1 : (s, \sigma_0)$. By Lemma 5 (Substitution), $\Pi \vdash [v/\overline{a}r]e_1 : (s', \sigma_1)$, for some $s' < : s$. Finally, Lemma 8 (Replacement with subtyping) gives $\Pi \vdash e' : (t', \sigma')$ for some $t' < : t$.}