12-20-2012

Open Effects: Optimistic Effects for Dynamic Dispatch

Yuheng Long
Iowa State University, csgzlong@iastate.edu

Hridesh Rajan
Iowa State University

Open Effects: Optimistic Effects for Dynamic Dispatch

Yuheng Long and Hridesh Rajan
TR #12-02
Initial Submission: December 20, 2012

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

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

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

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

Table of Contents

Open Effects: Optimistic Effects for Dynamic Dispatch*

Yuheng Long and Hridesh Rajan

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

Abstract. The open world assumption in modern object-oriented languages makes the design of a type-and-effect system challenging. The 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. We show that the open world assumption and the need for a type-and-effect system can be reconciled. We propose open effects, an optimistic effect 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 open effects; 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 noninterference of concurrent tasks.

1 Introduction

A type-and-effect system [19,29] is a valuable tool for programmers. It can help analyze locking disciplines [2], detect race conditions [17], analyze checked exceptions [27,4], analyze dynamic updating mechanisms [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 [44,43]. 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 [44].

Object-oriented features, such as dynamic dispatch, present a challenge for such type-and-effect system – these features require conservative handling for soundness [17, pp.222]. To illustrate, consider a library with classes `Pair` and `Op` in Figure 1. Two applications, PROGRAM and PROGRAM' (among others) use this library: each provides a separate and distinct subclass of the `Op` class. PROGRAM computes a prefix sum, whereas PROGRAM' computes a hash. Computing hash is implemented as an independent operation for each input, whereas prefix sum is not.

* This work was supported in part by the NSF under grants CCF-08-46059, and CCF-11-17937. Comments from Mehdi Bagherzadeh, Robert Dyer, Lorand Szakacs, Steve Kautz, Sarah Kabala, and ESOP 2013 anonymous reviewers were helpful in improving the quality of this draft. The notion of open effects is implemented in the OpenJDK Java compiler. This implementation and examples are available from the URL: http://www.cs.iastate.edu/∼csgzlong/OpenEffectJ/.

Now imagine that either a programmer or an analysis is using a type-and-effect system to help with parallelization of the method `apply`, executing its two statements, `fst = f.op(fst)` and `snd = f.op(snd)` on lines 7-8, concurrently. The type-and-effect system would be used to compute the effects of each statement. If the effects of these statements do not interfere with each other, then this parallelization is safe. However, the exact runtime type of the field `f`, on line 3 is unknown statically, so the effects of the method call `f.op(...)`, on line 7 and line 8 are also unknown.

```
1 class Pair {
2   int fst; int snd;
3   Op f;
4   Pair init() { fst = 1; snd = 2; this }
5   Op setOp(Op f) { this.f = f }
6   int apply() {
7     fst = f.op(fst); // write fst, read f, call f.op
8     snd = f.op(snd)  // write snd, read f, call f.op
9   }
10 }
```

```
11 class Op {
12   int op(int o) {
13     0
14   }
15 }
```

LIBRARY

```
16 class Prefix extends Op {
17   int sum;
18   Prefix init() { sum = 0; this }
19   int op(int o) {
20     sum += o // conflicts on sum
21   }
22 }
```

PROGRAM

```
23 class Hash extends Op {
24   int op(int o) { // pure
25     int key = o;
26     // Hash computation
27     key = ...
28   }
29 }
```

PROGRAM'

Fig. 1. Library classes Pair and Op and two separate applications that make use of it.

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

An alternative solution is to ask programmers to annotate the `op` method's implementation in class `Op` and use these effect annotations as an upper bound on the potential effects produced by all overriding implementations of that method. However, computing such an upper bound can be difficult for the programmer, primarily due to the variety of, and often unanticipated usage of library classes such as `Pair`. Even if the programmer is able to anticipate all such usages 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 `op` in PROGRAM, one may conclude that the parallelization of the `apply` method would be unsafe. Whereas, in reality there may be several subclasses of `Op`, such as class `Hash` in PROGRAM', whose computational effects will permit safe parallelization of the `apply` method.

Computing effects at runtime can help [8,15,16,40]. However, analyses based entirely on dynamic effect computation are reported to be expensive [16, 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. The compiler trusts the programmer statically, but also emits code to verify the 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 the limitations described above.

Our effect system has two kinds of effects: *open* and *concrete* effects. An open effect is produced by a method call, whose receiver's dynamic type is unknown, but whose 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 [44]. Like static type-and-effect systems, 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.

```
1  class Pair {
2    int fst, snd;
3    @open Op f;          // an open field
4    Pair init(){ fst = 1; snd = 2; this }
5    Op setOp(Op f) { this.f = f; }
6    int apply() {
7      fst = f.op(fst); // write fst, read f, open f.op
8      snd = f.op(snd)  // write snd, read f, open f.op
9    }
10 }
```

```
11 class Op {
12   int op(int o){
13     0
14   }
15 }
```

LIBRARY

Fig. 2. Modified `Pair` library with type `Op` annotated with `@open` on line 3 (left).

To illustrate, imagine that the programmer optimistically marked the field `f`'s type in class `Pair` as *open* as on line 3 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 `f.op(..)` on line 7 would be taken as an *open effect*, because the dynamic type of `f` is unknown. Thus, the effect of the statement on line 7 would be reading the field `f`, writing the field `fst` (write effect covers read) and an *open effect*. Similarly, the effect of the statement on line 8 would be reading the field `f`, writing the field `snd` and an *open effect*. Since open effects are assumed to be blank statically, the statements on line 7 and line 8 are, optimistically, assumed to be independent of each other. This optimistic assumption is verified later. We will show a utility of this optimistic assumption in §1.2.

The dynamic part of our analysis fills in, or *concretizes*, open effects when references marked with `@open`, such as `f`, are assigned. We illustrate using Figure 3.

Assume that the class `Pair` is already compiled. On a different day, the developer of PROGRAM imports the class `Pair`. At runtime, PROGRAM creates an instance `pr` of `Pair`, and sets the field `f` to an instance of the class `Prefix` on line 3 in Figure 3. This assignment to the field `f` concretizes the effect of the statements, on line 7 and line 8 in Figure 2, because their effects contain an *open* effect (method call effect on an

```
1 Pair pr = new Pair().init();
2 Prefix pf = new Prefix().init();
3 pr.setOp(pf);
4 pr.apply()
```

Fig. 3. The main expression of the client PROGRAM.

unknown reference `f`). Note that the receiver object `f` is now an alias of the instance `pf` of class `Prefix`. So, the concretized effect (the original effect union with the concrete effect of the method `op` of the class `Prefix`) of the statement on line 7 in Figure 2, when run within the context of the receiver object `pr`, is now reading the field `f`, writing to the field `fst` and writing to the field `sum` of a `Prefix` instance. Similarly, the concretized effect of the statement on line 8 in Figure 2, when run within the context of the receiver object `pr`, is now reading the field `f`, writing to the field `snd` and writing to the field `sum` of a `Prefix` instance. As a result, the effects of these statements conflict on the field `sum`, line 20 in Figure 1, in the control flow of the client PROGRAM that starts at the method call on line 4 in Figure 3.

```
1 Pair pr = new Pair().init();
2 Hash h = new Hash();
3 pr.setOp(h);
4 pr.apply()
```

Fig. 4. The main expression of the client PROGRAM'.

On yet another day, the developer of PROGRAM' imports the `Pair` class. PROGRAM' also instantiates `Pair`, but sets the field `f`, on line 2 to an instance of the class `Hash` in Figure 1, which concretizes the effects of method call `f.op(..)` on line 7 and line 8. These concrete effects are pure. The effects of statements are similarly enlarged. As a result, the statements are still independent.

1.2 Utilizing Open Effects: Example Usage for Concurrency

To illustrate the utility of open effects for concurrency, we use a hypothetical expression of the form **`fork`** `{` e_1 `;` e_2 `}` that runs e_1 and e_2 in parallel if assumptions related to open effects hold; otherwise e_1 and e_2 are run sequentially. To optimistically parallelize the `Pair` library using this expression, the programmer would rewrite the statements on lines 7-8 in Figure 2 as **`fork`** `{ fst = f.op(fst) ; snd = f.op(snd) }`.

An *open effect* is optimistically assumed to be blank statically. So, at compile-time, a parallelization technique based on open effects may treat each parallelization opportunity as optimistically parallel, if *concrete* effects of parallel tasks do not interfere, ignoring open effects altogether. So, in the **`fork`** expression of the previous paragraph, since the concrete effects of `fst = f.op(fst)` and `snd = f.op(snd)` do not interfere, these two expressions are treated as optimistically parallel.

At runtime, we require verifying open-effect related assumptions prior to running optimistically parallelized tasks. For our running example, this means checking immediately prior to running the fork expression, whether `f.op(fst)` and `f.op(snd)` interfere for the dynamic type of `f`.

Consider this dynamic check for the client PROGRAM (Figure 3). As discussed previously, in this program, the concretized effects of `f.op(fst)` and `f.op(snd)` interfere when the dynamic type of `f` is `Prefix`. Therefore, the fork expression runs serially within the control flow of the method call on line 4 in Figure 3.

Consider this dynamic check for the client PROGRAM' (Figure 4). As discussed previously, in this program, the concretized effects of `f.op(fst)` and `f.op(snd)` are independent when the dynamic type of `f` is `Hash`. Therefore, the fork expression runs in parallel within the control flow of the method call on line 4 in Figure 4.

Consider another program PROGRAM" that computes both the prefix sum and hash of the same pair in Figure 5. This program would reap the best of both worlds; the prefix sum would be computed safely in serial, whereas the hash would run in parallel.

```
1 Pair pr = new Pair().init();
2 pr.setOp(new Hash());
3 pr.apply();                  // parallel
```

```
4 Prefix pf = new Prefix().init();
5 pr.setOp(pf);
6 pr.apply()                   // sequential
```

Fig. 5. The main expression of the client PROGRAM".

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 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.

1.3 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 and 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 work well with dynamic class loading, with the proviso that all classes provide statically computed effects.

1.4 Summary of Benefits

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 important for software reuse and maintenance. Here "modular" means that the analysis can be done using only the code in question and the interface of the static types used in the code. For example, the effect computation for `Pair` relies only on the code for `Pair` and the interface of the `Op` classes, but not necessarily on its implementation. This would be essential for analyzing `Pair` 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 has a *small annotation overhead*, e.g. one annotation was needed in `Pair`. 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 `op` method in type `Op` (Figure 1, line 12). 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 has some runtime overhead. Our evaluation shows that this *overhead is negligible*. For example, our effect system was able to distinguish between effects of the method call `op` in PROGRAM (with `Prefix` class) and PROGRAM' (with `Hash` class), designed by two different programmers at two different times. This could allow the statements in the `apply` method in the `Pair` class to be optimistically parallelized. Main benefits of this parallelization are reaped by PROGRAM', where the implementation of `op` method is safe to parallelize. However, PROGRAM would not suffer significantly. Since conflicts are detected preemptively, 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.

1.5 Main Contributions and Outline

In summary, the main contributions of this work are:

- a language design with *open* effects;
- a type-and-effect semantics with open effects in §3, where the novelty lies in the integration of the *open* effects with standard effects;
- a dynamic semantics with open effect concretization in §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 §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 §6 and demonstrates that open effects has benefits, acceptable overhead, and low annotation cost; and
- a comparative analysis with related ideas in §7.

2 An Object-oriented Calculus with Open Effects

This section introduces *OpenEffectJ*, a minimal expression language based on Classic Java [18]. The grammar is shown in Figure 6. 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.

prog ::= $\overline{\text{decl}}$ e
decl ::= **class** c **extends** d { $\overline{\text{field}}$ $\overline{\text{meth}}$ }
field ::= [**@open**] t f;
meth ::= t m ($\overline{\text{arg}}$) { e }
t ::= c | **int**
arg ::= c var, where var $\neq$ **this**
e ::= var | **null** | arg = e; e "Var, null and def"
| **new** c () | e.m ($\overline{\text{e}}$) "New object and call"
| **this**.f | **this**.f = e | *loc* "Get, set and loc"
| e + e | n "Add and number"

where
$c \in \mathscr{C}$, a set of class names
$d \in \mathscr{C} \cup \{Object\}$, superclass names
$f \in \mathscr{F}$, a set of field names
$m \in \mathscr{M}$, a set of method names
$n, i \in \mathscr{N}$, a set of natural numbers
var $\in \mathscr{V} \cup \{\textbf{this}\}$, a set of variable names
$loc \in \mathscr{L}$, a set of locations

Fig. 6. The Grammar for *OpenEffectJ*.

A programmer writing code for reusable classes, e.g. the class `Pair`, typically knows that a reference such as `f` in that class may point to concrete objects of different types at runtime, and if a method is called on such a reference, it may result in different effects. These references can be annotated as `@open`, e.g., line 3 in Figure 2.

To simplify the presentation, only fields can be annotated `@open`. We call these annotated fields *open fields*. In §4, we discuss how we could handle other references. Sometimes we omit the implicit reference **this**, using `f` as a shorthand for **this**`.f`.

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 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 the receiver object whose type is annotated with `@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 the 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, e.g., a previously unknown field may become known as a result of a field set. The 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 7. 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* σ. An expression's type attribute is given as (t, σ), the type t of an expression and its effects σ. 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.[1] In the dynamic semantics, this name and the identity of the object suffices to identify which object's field is modified.[2]

An open effect contains the name of the open field f, the method, m, being invoked, and a placeholder for concrete effects, σ. The placeholder σ is used by our dynamic effect system to store concrete effects whenever f is set.

$\theta ::=$ OK	"program/decl types"	$\sigma ::= \emptyset \mid \sigma \cup \sigma \mid \{\bot\}$	"program effects"
$\mid (\bar{t} \to t, \sigma)$ in c	"method types"	$\mid \{\texttt{read f}\}$	"read effect"
$\mid (t, \sigma)$	"expression types"	$\mid \{\texttt{write f}\}$	"write effect"
$\Pi ::= \{var_i \mapsto t_i\}_{i \in \mathbb{N}}$	"type environments"	$\mid \{\texttt{open f m}\,\sigma\}$	"*open* effect"

Fig. 7. Domains of types and effects in our type-and-effect system based on [44,43,22].

The notation $t' <: t$ means t' is a subtype of t. It is the standard reflexive-transitive closure of the declared subclass relationship [18]. All class types and `int` are unrelated.

We state the type checking rules using a fixed class table (list of declarations CT [18]). The typing rules for expressions use a type environment Π, which is a finite partial mapping from variable names var to types t. Each method in the class table (CT) contains its effects, σ, computed by *OpenEffectJ*'s static type-and-effect system, in its signature.

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

3.2 Type-and-Effect Rules for Method Declaration

The main novelty, compared to typical static type-and-effect systems for OO languages, is in the rules for method declarations and calls. A static type-and-effect system typically requires that the effects of an overridden method in the subclass is contained in the effects of the method declared in the superclass. This allows the use of the effects of the superclass method as a sound approximation.

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 clients to implement application-specific functionalities.

[1] In the formal core, we do not track objects, but our compiler does (described in §5).

[2] 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 does not arise.

The (T-METHOD) rule says that a method m type checks in class $\mathtt{C}$, in which m is declared, if the body has type u and latent effect σ. The standard function *isType* checks if a type is valid. This rule uses a function *override*. The function *findMeth* (used by *override*) looks up the method m, starting from class c, looking in superclasses if necessary.

$$\text{(T-METHOD)}\quad \frac{\begin{array}{c} \mathit{override}(m,c,(\bar{t}\to t)) \\ \forall i\in\{1..n\},\ \mathit{isType}(t_i) \qquad \mathit{isType}(t) \\ (\overline{var:t},\mathtt{this}:c)\vdash e:(u,\sigma) \qquad u <: t \end{array}}{\vdash\ t\ m(\overline{t\ var})\{e\}:(\bar{t}\to t,\sigma)\ \text{in}\ c}$$

$$\frac{CT(c)=\mathtt{class}\ \ c\ \mathtt{extends}\ d\ \{\overline{field}\ \overline{meth}\} \quad \nexists i\in\{1..n\}\ \text{s.t.}\ meth_i=(t,\sigma,m(\overline{t\ var})\{e\}) \quad \mathit{override}(m,d,(\bar{t}\to t))}{\mathit{override}(m,c,(\bar{t}\to t))}$$

$$\frac{(d,t,m\ (\overline{t\ var})\ \{e\},\sigma)=\mathit{findMeth}(c,m)}{\mathit{override}(m,c,(\bar{t}\to t))} \qquad \mathit{override}(m,Object,(\bar{t}\to t))$$

$$\frac{CT(c)=\mathtt{class}\ \ c\ \mathtt{extends}\ d\ \{\overline{field}\ \overline{meth}\} \quad \exists i\in\{1\ldots n\}\ \text{s.t.}\ meth_i=(t,\sigma,m(\overline{t\ var})\{e\})}{\mathit{findMeth}(c,m)=(c,t,m(\overline{t\ var})\{e\},\sigma)}$$

$$\frac{CT(c)=\mathtt{class}\ \ c\ \mathtt{extends}\ d\ \{\overline{field}\ \overline{meth}\} \quad \nexists i\in\{1\ldots n\}\ \text{s.t.}\ meth_i=(t,\sigma,m(\overline{t\ var})\{e\}) \quad \mathit{findMeth}(d,m)=l}{\mathit{findMeth}(c,m)=l}$$

Fig. 8. Method declaration; overriding, which does not require effect containment (novel); and method lookup, which enables dynamic dispatch [18].

3.3 Open Effects of Polymorphic Method calls

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

$$\text{(T-CALL)}\quad \frac{\begin{array}{c} e_0\neq\mathtt{this}.f\vee(e_0=\mathtt{this}.f\wedge \mathit{typeOfF}(f)\neq(c,\texttt{@open}\ c_0)) \\ \mathit{findMeth}(c_0,m)=(c_1,t,m\ (\overline{t\ var})\ \{e_{n+1}\},\sigma) \\ \Pi\vdash e_0:(c_0,\sigma_0) \qquad (\forall\, i\in\{1..n\}\ ::\ \Pi\vdash e_i:(t_i',\sigma_i)\ \wedge\ t_i'<:t_i) \end{array}}{\Pi\vdash e_0.m(\bar{e}):(t,\{\bot\})}$$

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 *reads/writes everything* [22,34]. As §5 discusses, if the receiver's exact type is known, this effect can be made more precise.

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

$$\text{(T-CALL-OPEN)}\quad \frac{\begin{array}{c} \Pi\vdash\mathtt{this}.f:(c_0,\sigma_0) \qquad \mathit{typeOfF}(f)=(d,\texttt{@open}\ c_0) \\ \mathit{findMeth}(c_0,m)=(c_1,t,m\ (\overline{t\ var})\ \{e_{n+1}\},\sigma) \qquad (\forall\, i\in\{1..n\}::\Pi\vdash e_i:(t_i',\sigma_i)\ \wedge\ t_i'<:t_i) \end{array}}{\Pi\vdash\mathtt{this}.f.m(\bar{e}):(t,\{\texttt{open f m}\emptyset\}\cup\bigcup_{i=0}^{n}\sigma_i)}$$

In this case, statically we assume that this method call will have no effect (represented by $\emptyset$ in $open\ \mathtt{f}\ \mathtt{m}\ \emptyset$). To demonstrate this rule, consider the method call expression `f.op(...)`, on line 7 in Figure 2. Here, the receiver object's type is annotated as *open* on line 3. Therefore, by the (T-CALL-OPEN) rule, the type-and-effect system will generate the effect $\{open\ \mathtt{f}\ \mathtt{op}\ \emptyset\}$ for this method call expression.

To illustrate the implications of assuming empty effects, let Φ 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 σ are in σ' and no open effects in σ are in σ'. If σ' satisfies Φ, then σ is also assumed to satisfy Φ, provided that the dynamic part of our type-and-effect system verifies that concretized open effects satisfy Φ.

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 for the field read rule (T-GET) is standard and the effect is a read effect.

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

For field set, we distinguish the effects based on whether the field `f` is annotated as *open*, by the *typeOfF* function, see §3.3. If `f` is not an *open* field, the (T-SET) rule applies and gives a write effect. The typings for this rule is standard.

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

To illustrate these rules, consider the expression `fst = f.op(fst)`, on line 7 in Figure 2. The (T-GET) rule gives a read effect ($read\ \mathtt{f}$) for referencing the receiver `f` and a read effect for the parameter `fst` ($read\ \mathtt{fst}$). Since `fst` is not an *open* field, the (T-SET) rule applies and gives a write effect ($write\ \mathtt{fst}$) for assigning the field `fst`. So, the type-and-effect system gives the expression `fst = f.op(fst)` the effect $\{read\ \mathtt{f}, write\ \mathtt{fst}, open\ \mathtt{f}\ \mathtt{op}\ \emptyset\}$. Here, $write\ \mathtt{fst}$ covers $read\ \mathtt{fst}$.

If `f` is an *open* field, the alternative (T-SET-OPEN) rule applies:

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

As we show in §4.2, an open field set expression represents a program location where concrete effects in some open effects may change. Our type-and-effect system 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. We will revisit the implications of this rule in §4.3.

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 9) to another. Some rules use an implicit attribute, the class table CT.

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

Domains:

Σ	::=	$\langle e, \mu \rangle$	"Program Configurations"
μ	::=	$\{loc_i \mapsto o_i\}_{i \in \mathbb{N}}$	"Store"
o	::=	$[c\,.\,F\,.\,E]$	"Object Records"
F	::=	$\{f_i \mapsto v_i\}_{i \in \mathbb{N}}$	"Field Maps"
v	::=	$\texttt{null} \mid loc \mid n$	"Values"
E	::=	$\{m_i \mapsto \sigma_i\}_{i \in \mathbb{N}}$	"Effect Maps"

Evaluation contexts:

$$\begin{array}{rcl} \mathbb{E} & ::= & - \mid \mathbb{E}\,.\,m(\bar{e}) \\ & & \mid v\,.\,m(\bar{v}, \mathbb{E}, \bar{e}) \\ & & \mid \mathbb{E}\,.\,f{=}e \mid v\,.\,f{=}\mathbb{E} \\ & & \mid \mathbb{E}.f \mid t\ var{=}\mathbb{E}\texttt{;}\,e \\ & & \mid \mathbb{E}{+}e \mid v{+}\mathbb{E} \end{array}$$

Fig. 9. Domains used in the dynamic semantics of *OpenEffectJ*.

A configuration consists of an expression e and a global store μ. A store maps locations to object records. An object record $o = [c.F.E]$ contains the concrete type c of the object, a field map F, and a dynamic effect map E (which is new). An effect map E is a function from a method name to its runtime effects.

We present the semantics as a set of evaluation contexts $\mathbb{E}$ and a one-step reduction relation that acts on the position in the overall expression identified by the evaluation context [18]. This avoids the need for writing out standard recursive rules and clearly presents the order of evaluation. The language uses a call-by-value evaluation strategy. The initial configuration of a program with a main expression e is $\Sigma_\star = \langle e, \bullet \rangle$. The operator $\oplus$ is an overriding operator for finite functions, i.e. if $\mu' = \mu \oplus \{loc \mapsto o\}$, then $\mu'(loc') = o$ if $loc' = loc$, otherwise $\mu'(loc') = \mu(loc')$.

4.2 Dynamic Effect Management in OO Expressions

The rules for OO expressions are shown below (§10 contains omitted auxiliary functions). The novelty is that some of the rules manipulate the effect map E. This map contains more precise, dynamically computed effect, enabling verification of optimistic assumptions made statically about *open* effects.

(NEW)

$$\frac{loc \notin dom(\mu) \qquad F = \{f \mapsto default(f) \mid f \in fields(c)\} \qquad E = \{m \mapsto \sigma \mid m \mapsto \sigma \in methE(c)\} \qquad \mu' = \{loc \mapsto [c.F.E]\} \oplus \mu}{\langle \mathbb{E}[\texttt{new}\ c()], \mu \rangle \hookrightarrow \langle \mathbb{E}[loc], \mu' \rangle}$$

The (NEW) rule uses a function *methE* (below) to initialize the effect map of the new instance. This function searches the class table CT for all the methods declared in class

c and all its super classes. Its result is a map E that contains each method m found in the previous step and its statically computed effects σ. The static type-and-effect rules in §3 are used to compute the effects σ, which are then stored in CT. The function *default* returns 0, if the input is an `int` type; it returns `null` otherwise.

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

To illustrate the (NEW) rule, consider the example in Figure 5. In this example, three objects are created: a `Pair`, a `Hash`, and a `Prefix`. The object records for each of these objects are shown in Figure 10.

Expression	Object records [c.F.E] created in the store μ	Location created
`new Pair() (line 1)`	$[Pair.\{fst \mapsto 0, snd \mapsto 0, f \mapsto \texttt{null}\}.\{setOp \mapsto \{\bot\}, apply \mapsto \{write\ \texttt{fst}, write\ \texttt{snd}, read\ \texttt{f}, open\ \texttt{f}\ \texttt{op}\ \emptyset\}\}]$	loc
`new Hash() (line 2)`	$[Hash.\emptyset.\{op \mapsto \emptyset\}]$	loc_0
`new Prefix() (line 5)`	$[Prefix.\{sum \mapsto \texttt{null}\}.\{op \mapsto \{write\ \texttt{sum}\}\}]$	loc_1

Fig. 10. Examples of object records: created by expressions in Figure 5.

In the object record for the `Pair` instance in Figure 10, the field mapping $\{fst \mapsto 0, snd \mapsto 0, f \mapsto \texttt{null}\}$ is standard, whereas the effect mapping $\{setOp \mapsto \ldots, apply \mapsto \ldots\}$ is new. Here, the functions *fields* and *methE* return all the fields, $\{fst, snd, f\}$, and methods, $\{setOp, apply\}$, declared in class `Pair` and its super classes, respectively. In the effect mapping, the $\bot$ effect is computed statically for the method `setOp`, because the *open* field `f` is set in this method; whereas the statically computed effect of method `apply` is $\{write\ \texttt{fst}, write\ \texttt{snd}, read\ \texttt{f}, open\ \texttt{f}\ \texttt{op}\ \emptyset\}$.

(GET)
$$\frac{\mu(loc) = [c.F.E] \quad v = F(f)}{\langle \mathbb{E}[loc.f], \mu \rangle \hookrightarrow \langle \mathbb{E}[v], \mu \rangle}$$

(SET)
$$\frac{[c.F.E] = \mu(loc) \quad \mu_0 = \mu \oplus (loc \mapsto [c.(F \oplus (f \mapsto v)).E]) \quad \mu' = update(\mu_0, loc, f, v)}{\langle \mathbb{E}[loc.f = v], \mu \rangle \hookrightarrow \langle \mathbb{E}[v], \mu' \rangle}$$

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

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

The inputs to function *update* are the current store μ, the location loc, the field f, and the R-value v. The result is another store μ''. This function first calls function

updateEff to construct the new effect E' for the object o pointed to by *loc* (by the effect of an object o, we mean the effects of all the methods of o). If the effects of o remain unchanged, i.e., $E' = E$, *update* stops and returns the original store μ. Otherwise, the effect of an object o', which has some *open* field pointing to o, should also be changed. Effects are further propagated using the function *fixPoint* until a fixed point is reached.

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

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

Function *reverse* searches the input store μ for object loc_i and field f_i pairs that are pointing to the location *loc*. In practice, reverse pointers can be used to optimize this [6].

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

Each object contains a map of effects E. Function *updateEff* constructs a new map E' by invoking the function *concretize* on each entry of E.

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

$$\begin{array}{ll}
cp(\varepsilon) = \textbf{match } \varepsilon \textbf{ with} & | \ \texttt{open f m}\ \sigma \rightarrow \sigma \\
 & | \ _ \qquad\qquad\quad \rightarrow \varepsilon
\end{array}$$

Note that when an *open* field f is set, only the *open* effects, i.e. `open f m` σ, that have f as receiver are changed. Other effects remain unchanged. So if the input effect ε is an open effect whose field matches the input field `f`, function *concretize* returns a new open effect ε'. Function *cp* is used by function *concretize* to retrieve the concrete effects, i.e., the effects of the R-value v are copied to fill the placeholder effects of ε'.

To illustrate the *update* function, take the expression `pr.setOp(new Hash())`, on line 2 in Figure 5, where `pr` is a `Pair` instance. This expression sets the *open* field `f` to a `Hash` instance `h`. Function *updateEff* gets the new effects of both methods `setOp` and `apply` of the object `pr` being changed. Method `setOp` has no *open* effect and its effect remains unchanged. The original effect of method `apply` is {`write fst`, `write snd`, `read f`, `open f op` $\emptyset$}. For the function *concretize*, the input v is now an alias of `h`, the R-value. Here, only the *open* effect (`open f op` $\emptyset$) matches the field `f` being changed: the third line of *concretize*; other effects remain unchanged. Function *cp* copies the concrete effect $\emptyset$ of the pure method `op` of `h`, the R-value. Then *concretize*

puts the effect $\emptyset$ from *cp*, into the *open* effect: (`open f op` $\emptyset$), which remains unchanged. Because the effects of both methods `setOp` and `apply` are unchanged, the first *update* function applies, and *update* stops.

To show the second *update* function, consider the expression `pr.setOp(pf)`, on line 5. Here, `pf` is a `Prefix` instance. The *open* effect of method `apply` is `open f op` $\emptyset$. For *concretize*, *v* is an alias of `pf`. Function *cp* copies the effect {`write sum`} of method `op` of `pf`. Then, function *concretize* puts {`write sum`} into the *open* effect: (`open f op {write sum}`). Because the effect of method `apply` changes, i.e. $\emptyset \neq$ {`write sum`}, the second *update* function applies. It uses function *reverse* to obtain all the objects *o*, having *open* field(s) pointing to the current object *pr* being set; and updates the effects of *o* until there are no further changes. In the example, there is no object pointing to `pr`, so *update* stops.

The (CALL) rule is standard. It acquires the method signature via the function *findMeth* (§9.1 Figure 8) that uses dynamic dispatch[18].

(CALL)
$$\frac{(c',t,m(\overline{t\ var})\{e\},\sigma) = \mathit{findMeth}(c,m) \quad [c.F.E] = \mu(loc) \quad e' = [loc/\mathbf{this},\overline{v/var}]e}{\langle \mathbb{E}[loc.m(\bar{v})],\mu\rangle \hookrightarrow \langle \mathbb{E}[e'],\mu\rangle}$$

To summarize, in *OpenEffectJ*'s semantics, object creation is augmented to initialize the effect map; and field assignment to *open* fields updates these effect maps. These effects can then be used at runtime for checking effect based properties. We show an example of such a property in §6.

As discussed in §2, other *open* references can be allowed, e.g., the type system can be extended to generate an *open* effect `open var m` $\emptyset$ for a method call *var*.*m*(..). In the semantics, the concretization of this *open* effect would happen when an *open* variable *var* is set, e.g., when an *open* parameter is bound to a location in the (CALL) rule.

4.3 Open Effects Verification

In this section, We first describe open effect verification using concurrency as a usecase. Then we will illustrate the essence of the (T-SET-OPEN) rule (defined in §3.4).

To illustrate the utility of open effects for concurrency, we use the expression **`fork`** {e_1 `;` e_2} introduced in §1.2, that runs e_1 and e_2 in parallel if their effects do not interfere. The noninterference relation for effects is as follows, read effects do not interfere; read/write and write/write pairs conflict if they access the same field f of the same object *loc* (object and field sensitivity §5); *open* effect `open f m` σ conflicts with another effect σ' if σ conflicts with σ'; bottom effect $\perp$ conflicts with any effect.

To illustrate the case that the tasks do not conflict, consider the expression `pr.apply()` on line 3 in Figure 5. This expression is right after setting the *open* field `f` to a `Hash` instance `h`. In the store μ, the object record for `pr` is $[\mathit{Pair}.\{\mathit{fst} \mapsto 1, \mathit{snd} \mapsto 2, f \mapsto \mathit{loc}_0\}.\{\mathit{setOp} \mapsto \{\perp\}, \mathit{apply} \mapsto \{$`write fst, write snd, read f, open f op` $\emptyset\}\}]$ (see the examples for *update* in §4.2), where loc_0 maps to `h`. The **`fork`** checks the open effects `open f op` $\emptyset$ and the concrete effects of the two expressions in method `apply` (the table below). There is no conflict and the expressions can be run concurrently:

Expression in Figure 2	Effect of the expression (`f` points to a `Hash` instance).
`fst = f.op(fst) (line 7)`	{*write* `fst`, *read* `f`, *open* `f op` ∅}
`snd = f.op(snd) (line 8)`	{*write* `snd`, *read* `f`, *open* `f op` ∅}

To illustrate the case that the tasks do conflict, consider the expression `pr.apply()`, on line 6. This expression is right after setting `f` to a `Prefix` instance `pf`. Now, `pr` maps to the object: $[Pair.\{..,f \mapsto loc_1\}.\{..,apply \mapsto \{..,$ *open* `f op{` *write* `sum}`$\}\}]$ (§4.2), where loc_1 maps to `pf`. The **`fork`** checks the effects of the two expressions:

Expression in Figure 2	Effect of the expression (`f` points to a `Prefix` instance).
`fst = f.op(fst) (line 7)`	{*write* `fst`, *read* `f`, *open* `f op{` *write* `sum}`}
`snd = f.op(snd) (line 8)`	{*write* `snd`, *read* `f`, *open* `f op{` *write* `sum}`}

There is a write conflict on `sum`, thus the two expressions should be run sequentially.

We now illustrate the (T-SET-OPEN) rule in §3.4. Consider a sequence of 3 expressions $e_1;e_2;e_3$, where e_1 is `f = new Hash()`, e_2 is `f = new Prefix()` and e_3 is `f.op(2)`, and `f` is an open field. After e_1 is evaluated, `f` maps to a `Hash` instance `h`. Consider now we want to verify the effect of the 2 tail expressions $e_2;e_3$. The effect is $\{\bot\}$, by the (T-SET-OPEN) rule, because `f` is an *open* field. If, however, the normal (T-SET) rule is used, the effect would have been {*write* `f`, *open* `f op` ∅}. The open effect is ∅, because `f` maps to `h` before evaluating $e_2;e_3$. This means that if $e_2;e_3$ is evaluated, it will not writes to any field other than `f`. This is not a sound approximation. Becuase after e_2 is evaluated, the open effect become *open* `f op{` *write* `sum}`, which invalidates the verification before (this is analogous to the grandfather paradox). And `f` is set to a `Prefix` instance `pf`, so e_3 is a call on `pf` and it writes to the field `sum`. The effect $\{\bot\}$ for $e_2;e_3$, by the (T-SET-OPEN) rule, is a sound approximation.

4.4 Soundness: Type and Effect Preservation

We have proven two key formal properties: effect preservation and type preservation [34,12]. The proof of type preservation uses the standard subject reduction argument [18](§11 contains contains a detailed proof for effect preservation).

The effect preservation property is that the dynamic effects, 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 [34,12], in which the exact effect of a task is known statically. The 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 η can be a read effect (*rd loc* `f`) or a write effect (*wt loc* `f`). A dynamic effect η refines a static effect σ, written $\eta \propto \sigma$, if either $\eta = ($*rd loc* `f`$) \wedge$ ((*read* `f`$) \in \sigma \vee$ (*write* `f`$) \in \sigma$); or $\eta = ($*wt loc* `f`$) \wedge$ (*write* `f`$) \in \sigma$. The dynamic effect of an expression e is a dynamic trace $\chi = \overline{\eta}$, a sequence of dynamic effects.

To record dynamic effects for proofs, we use an instrumented semantics *dyn*. If Σ reduces to Σ' in the original semantics, then $dyn(\Sigma,\chi)$ reduces to $dyn(\Sigma',\chi')$. This reduction continues until it evaluates to a value, i.e. $dyn(\langle v,\mu_v\rangle,\chi_v)$; μ_v is the final store and χ_v contains all the heap accesses. Here, if Σ is $\langle \mathbb{E}[loc.f],\mu\rangle$ and $\langle \mathbb{E}[loc.f = v],\mu\rangle$ then χ' is $\chi +$ (*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 [34,12].

To illustrate dynamic traces, consider the expression `fst = f.op(fst)`, shorthanded for `this.fst = this.f.op(this.fst)`, in method `apply`, on line 7 in Figure 2. If `apply` is called on the receiver `pr`, on line 6 in Figure 5, then the *open* field `f` is an alias of the `Prefix` instance `pf` (line 5). In the store μ, the location *loc* maps to the `Pair` instance (Figure 10) and loc_1 maps to `pf`. When `apply` is called, the variable `this` is substituted with *loc*. The second column in Figure 11 illustrates the dynamic effect trace generated for evaluating the expressions in the first column.

Expression	Dynamic Effect	Static Effect	Semantics
loc.fst = *loc*.f.op(*loc*.fst)	*rd loc* f	{*read* f, *write* fst, *write* sum}	**(Get)**
loc.fst = loc_1.op(*loc*.fst)	*rd loc* fst	{*write* fst, *write* sum}	**(Get)**
loc.fst = loc_1.op(1)	-	{*write* fst, *write* sum}	**(Call)**
loc.fst = loc_1.sum += 1	*wt* loc_1 sum	{*write* fst, *write* sum}	**(Set)**
loc.fst = 1	*wt loc* fst	{*write* fst}	**(Set)**

Fig. 11. An example of dynamic effect. The first column shows the evaluation sequences of the expressions. The second column shows the dynamic effect η generated in each reduction step. The third column shows the static effect σ of the expression. The last column shows the semantics rule applied for the reduction.

The semantics stores the effects σ in object records for methods. Therefore, two invariants for σ are necessary to maintain the *effect preservation* property (Definition 3). The invariants include: the placeholder effect σ_0, of an *open* effects *open* f m σ_0, should be supereffect $\supseteq$ of the effect $E'(m)$ for the method m of the object f is pointing to (Definition 1), 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 σ of the body e of m (Definition 2).

Definition 1. *[Well-formed object] An object record* $o = [c.F.E]$ *is a well-formed object in* μ*, written* $\mu \vdash o$*, if for all* open *effect* open f m $\sigma_0 \in \sigma \in rng(E)$*, either* $(F(f) = loc) \wedge (\mu(loc) = [c'.F'.E']) \wedge (E'(m) \subseteq \sigma_0)$*; or* $(F(f) = \mathbf{null}) \wedge (\sigma_0 = \emptyset)$.

To illustrate a well-formed object, consider the `Pair` instance `pr` created on line 1 in Figure 5. An object o is created for `pr`: $Pair.\{\ldots, f \mapsto \mathbf{null}\}.\{\ldots, apply \mapsto \{\ldots,$ *open* f op $\emptyset\}\}$ (Figure 10). Because its only *open* field `f` maps to **`null`** and the *open* effect is empty, o is a well-formed. After setting `f` to a `Hash` instance `h`, on line 2, o becomes (§4.2): $Pair.\{\ldots, f \mapsto loc_0\}.\{\ldots, apply \mapsto \{\ldots,$ *open* f op $\emptyset\}\}$, where loc_0 maps to `h`. Because the effect of method `op` of `h` is $\emptyset$, o is well-formed. After setting `f` to a `Prefix` instance `pf`, on line 5, o becomes (§4.2): $Pair.\{\ldots, f \mapsto loc_1\}.\{\ldots, apply \mapsto \{\ldots,$ *open* f op {*write* sum}$\}\}$, where loc_1 maps to `pf`. Because the effect of method `op` of `pf` is {*write* sum}, o is well-formed.

In the static effect computation, the concrete effect in an *open* effect is empty $\emptyset$ (§3.3). During program execution, the *open* effects are concretized when *open* fields are set (§4.2). To reflect this, in the following, we use the effect judgment $\mu \vdash e : \sigma$.

Given an expression e and a store μ, it takes the concrete effect out of the *open* effect to represent the effect of a method call in runtime. This dynamic relation for effect is similar to the static rules in §3, except for method calls on *open* fields (E-CALL-OPEN). The *open* effect now has a concrete part σ, instead of $\emptyset$, because we know the concrete object that an *open* field is pointing to(§11 contains omitted rules).

$$\text{(E-CALL-LOC)}\quad \frac{\mu(loc)=[c.F.E] \quad E(m)=\sigma_0 \quad (\forall\, i\in\{1..n\} \;::\; \mu\vdash e_i:\sigma_i)}{\mu\vdash loc.m(\overline{e}):\bigcup_{i=0}^{n}\sigma_i}$$

$$\text{(E-CALL-OPEN)}\quad \frac{\mu\vdash loc.f:\sigma_0 \quad \mu(loc)=[c.F.E] \quad E=\{m_i\mapsto\sigma_i \| 1\le i\le q\} \quad \exists i \text{ s.t. } (\texttt{open f m}\,\sigma\in\sigma_i) \quad (\forall\, i\in\{1..n\} \;::\; \mu\vdash e_i:\sigma_i)}{\mu\vdash loc.f.m(\overline{e}):\sigma\cup\bigcup_{i=0}^{n}\sigma_i}$$

To illustrate the effect judgment $\mu\vdash e:\sigma$, consider the first expression in Figure 11. If the *open* field `f` is an alias of the `Prefix` instance `pf` (line 5) and `pr` points to an object $[Pair.\{\ldots, f\mapsto loc1\}.\{\ldots, apply\mapsto\{$`write fst`$, $`write snd`$, $`read f`$, $`Open f op {write sum}$\}]$, where $loc_1$ maps to `pf`. The effect judgment copies the concrete effect $\{$`write sum`$\}$, so the expression `fst = f.op(fst)` has effect $\{$`read f`$, $`write fst`$, $`write sum`$\}$ at runtime (third column in Figure 11), when `f` is an alias of `pf`.

Definition 2. *[Well-formed location] A location loc is well-formed in μ, written $\mu\vdash loc$, either $\mu(loc)=$* **`null`***; or if $\mu(loc)=[c.F.E]$, $\forall m\in dom(E)$ s.t.$findMeth(c,m)=(c',t,m(\overline{t\ var})\{e\},\sigma')\wedge\mu\vdash[loc/$***`this`***$]e:\sigma$, then $\sigma\subseteq E(m)$.*

To illustrate well-formed locations, consider the location loc_0, in Figure 10. In the store μ, $\mu(loc_0)=[Hash.\emptyset.E]$, where $E=\{op\mapsto\emptyset\}$. Function $findMeth(Hash,op)=(Hash,int,op(int\ o)\{..\},\emptyset)$ (Figure 8). Since $\emptyset\subseteq E(op)$, loc_0 is well-formed. For the location loc_1, $\mu(loc_1)=[Prefix.\{..\}.E']$, where $E'=\{op\mapsto\{$`write sum`$\}\}$. Here, $findMeth(Prefix,op)=(Prefix,int,op(int\ o)\{..\},\{$`write sum`$\})$, loc_1 is well-formed.

To show the effect preservation, we need to ensure that the store is well-formed throughout the program execution (Lemma 2). Lemma 2 says that after one or more reduction steps $\hookrightarrow^*$ from the initial configuration $\Sigma_\star$, the store μ' is well-formed.

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

Lemma 1. [Store preservation] *If $\Sigma_\star\hookrightarrow^*\langle e',\mu'\rangle$, where $\Sigma_\star=\langle e,\bullet\rangle$, then $\mu'\vdash\diamond$.*

Proof Sketch: The initial store, when the program starts, is empty $\bullet$ and $\bullet\vdash\diamond$. Therefore, it suffices to show that if the store μ before the transition is well-formed, then the store μ' after the transition is also well-formed (§11 contains the details). ∎

The effect preservation (Theorem 1) says that the dynamic effects of each expression e refine the open effects of e computed right before it is evaluated.

Theorem 1. [Effect preservation] *Given two program configurations $\Sigma=\langle e,\mu\rangle$ and $\Sigma'=\langle e',\mu'\rangle$, such that $\Sigma\hookrightarrow\Sigma'$, if $\mu\vdash\diamond$ and $\mu\vdash e:\sigma$, then there is some effect σ' and dynamic trace χ such that*

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

(b) the new dynamic effect η, in the trace χ, refines the static effects σ: $dyn(\Sigma,\chi)=(\Sigma',\chi+\eta)\Rightarrow(\eta\propto\sigma)$.

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 η in each step refines the static

effect σ of the original expression e, $\eta \propto \sigma$. Thus with (a), η refines the effect σ_0 of the expression e_0 right before it is evaluated, with the heap μ_0. Unlike the static approaches, which compute σ_0 at compile-time, *OpenEffectJ* computes σ_0 before evaluating e_0.

To illustrate conclusion (a) of Theorem 1, consider the effect judgment in the third column in Figure 11, the static effect of a subsequent expression is a subset of the current expression. To illustrate conclusion (b), consider the second and third column in Figure 11. The dynamic effect generated in each step, in the second column, refines the static effect in the third column, computed for the expression in the first column.

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 `@open` annotation, parsing remains unchanged. The Type checking, Attribute and Flow phases in the compiler are modified to implement new constraints specified in §3. The Attribute phase is also extended with an effect analysis. It attributes each AST node with static effects for each method, which are then used by the tree rewriting phase to generate code for runtime effect manipulation.

Stronger Effect Analysis. The effect analysis is augmented with a few modular analyses to improve precision. These include an intra-procedural definite alias analysis [21], a purity analysis [37], an array effect analysis [36] and an escape analysis [32]. 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 $\perp$ in the (T-CALL-OPEN) rule in §3.3, e.g., inlining 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. The array analysis gives a better precision on which segment of an array is accessed. This is useful, e.g., in scenarios where concurrent tasks write to different slots of the same array, i.e., in-place updates. The escape analysis detects objects `f` that never escape outside another object `o`. Following ownership systems [12], we say that object `f` is owned by `o`. This is useful if the effect verification phase can ensure that different tasks can only access different owners.

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. 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. E.g., in §3.4, the rules (T-GET) and (T-SET) will give an effect {`read f`} and {`write f`}, respectively. When an object o is created by the (NEW) rule in §4.2, the implicit placeholder ***this*** will be replaced with the location *loc* of o being created. The read/write effects will be stored as {`read` *loc* `f`} and {`write` *loc* `f`}, making the dynamic effects more precise.

Effect Maintenance. We noted in §4 that if the effects of an object o change, 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 *reverse*. 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 effects of all of its methods 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 `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 `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. 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 Noninterference of Concurrent Tasks Using Open Effects

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

The method `fork` then tests to see if multiple invocations of the `run` method, with `t` as the receiver object, will have mutually conflicting effects (see §4.3 for when effects conflict). If multiple calls will not have conflicting effects, then the library executes n parallel copies of the `run` method, where n is the size of the array `input`. Otherwise, the library executes a sequential loop that calls the `run` method with each element of the array `input` as argument.

6.2 Parallelizing Representative Libraries

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

ArrayList. The Class `ArrayList` is from OpenJDK. We added a method `apply` to the class `ArrayList`. Similar to the `Pair` example in Figure 1, this method applies an operation on each element of the `ArrayList`. We use the abstract implementation of the class `Op`, the operation to be applied on the elements. Also, we annotate the variable of type `Op` with `@open`. This allows safe parallelization on applying the operation on each element, when the operation points to instances of concurrency safe subclasses of `Op`. In total, we added one `@open` annotation.

Map-Reduce. In this framework [1], the first step is *map*, i.e partitioning the problem and distributing it to workers, and the second step is *reduce*, i.e. combining results from workers. For extensibility and reuse, this framework is designed to use abstract implementations 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 `Comparator`s. 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 read/write conflicts if an instance of RBC is used as a `Comparator` and would result in incorrect output (the original code in RBC is thread safe though). 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 satisfy a certain objective. This library uses the abstract implementation of the class `Goal` and lets the clients extend it to implement application-specific search objectives. We annotate the field `goal` of type `Goal` with `@open`. With more precise effects, 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 the lower bound to the center point of the 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 an 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`.

Pipeline. In this pipeline framework (Pipeline) [45,7], each pipeline stage applies a certain operation on the input data. These operations are referred to as filters. Pipeline uses the abstract implementation of the class `Filter`. Each data element goes through the filters sequentially. Different filters could be applied to different elements in parallel, resulting in pipeline parallelism. We annotate the field `filter` of type `Filter` with `@open`. Pipeline can be safely parallelized if the filters do not conflict with each other.

6.3 Performance Evaluation

Experimental Setup. We have conducted an initial evaluation of *OpenEffectJ*'s prototype compiler using the library classes described in §6.2. All experiments in §6.3 were run on a system with a total of 4 cores (Intel Core2 chips 2.40GHz) running Fedora GNU/Linux. For each experiments, an average of the results over 30 runs was taken.

Client Code. For ArrayList, we apply the hash computation (Hash) and a slightly heavier computation (Heavy). For each element o in the list, the Heavy variant computes the formula $Math.sqrt(2 * Math.pow(o,2))$. For both Hash and Heavy variants, `ArrayList` contained 20 million elements. MergeSort sorts a list of 10 million randomly generated integers. DFS searches for solutions to an n-queens problem, where n is 11. In the map-reduce algorithm, the map step computes the formula $Math.sqrt(2 * Math.pow(o,2))$ for each element o and the reduce step is simply addition. It is applied to 100 million integers. NI uses a recursive Gaussian quadrature of $(2 * i - 1) \cdot x^{2*i-1}$, summing over odd values of i from 1 to 12 and integrating from -5 to 6. Pipeline models Radix Sort. The first stage generates a stream of 8 integer arrays, having 1 million elements each. The subsequent stages sort the arrays on a different radix.

Program	Serial	Manual (No effects/Unsafe)		*OpenEffectJ*			Parallelism
↓	time (s)	time (s)	speedup	time (s)	speedup	overhead	↓
ArrayList(Hash)	**0.13**	**0.11**	**1.19**	**0.11**	**1.18**	**0.44%**	**Forall**
ArrayList(Heavy)	**1.30**	**0.55**	**2.39**	**0.53**	**2.45**	**-2.50%**	**Forall**
Pipeline	**2.08**	**1.65**	**1.26**	**1.70**	**1.23**	**2.29%**	**Pipeline**
MergeSort	**2.54**	**1.10**	**2.31**	**1.14**	**2.23**	**3.58%**	**Recursive**
Depth First Search	**27.82**	**13.77**	**2.02**	**12.68**	**2.19**	**-7.91%**	**Recursive**
MapReduce	**6.58**	**2.40**	**2.73**	**2.43**	**2.71**	**0.95%**	**Recursive**
Integrate	**2.12**	**1.83**	**1.16**	**2.00**	**1.06**	**9.10%**	**Recursive**

The performance results of running these applications are shown in the table above. The column marked *Serial* shows the time taken by the single-threaded version of the application. The column marked *Manual* shows the time taken by (and speedup of) the manually parallelized version, which does not manage effects (and thus could be unsafe). The column marked *OpenEffectJ* shows the time taken by (and speedup of) our prototype compiler, which does effect verification prior to forking off tasks. The column marked *Parallelism* shows the pattern of the parallelization applied.

The *OpenEffectJ* version, as well as the manual (unsafe) version showed decent to good speedup for all of these applications. The overhead for this example was also small. 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 *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 statically, we can 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 the 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.

Note that a dynamic effect system has to ensure that the conflicts of the concurrent tasks are commutative [35]. Otherwise, when conflicts, it has to rollback in a deterministic manner. E.g., in Figure 1, the `Prefix` is not commutative and the two tasks conflict. If the task with the statement `snd = f.op(snd)` commits first, while the other task rollbacks, the result is not the same as executing the tasks sequentially (§7.1).

There are at least two scenarios when open effects outperforms the other two. The 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*. Tasks *a* and *b* do not conflict, but both conflict with *c*. The static approach may serialize all of them. A technique based on open effects can indicate that 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 six representative examples, three 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 [39] and hybrid type checking [26] that blend the advantages of static and dynamic type checking. Similarly, open effects blend the advantages of static and dynamic effect systems. Open effects are related to Open Type Switch (Mach7) [41]. Mach7 lets users choose between type hierarchy openness and efficiency. Similarly, open effects let users choose the openness of the effects of the method calls.

There exists a rich body of work on type, regions and effect-based approaches for reasoning about object-oriented programs [22,34,7,9,11,20]. These approaches are static, whereas open effects uses a hybrid approach. A recent work in this category is deterministic parallel Java (DPJ). DPJ [7] uses effect parameters [29] and effect constraints to reason about the correctness of the client code. Effect constraints are 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 a 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 [16,40,15,8,24]. 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 the reachable objects graph (OG) of tasks as their effects [33]. 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 [17]. Also, using effects sets instead of reachable OG may be more precise for OO features, e.g., in every library in §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 [42,38,23,28] optimistically executes tasks concurrently, but monitors memory accesses. It rollbacks side-effects when conflicts happen. There are TM-like approaches [14,5,46] 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] programmers know that tasks conflict on shared objects and annotate these objects. Eash task has a local copy of the objects to avoid data races. In contrast, open effects is useful where all overriding subclasses may not be known. For example, when a library class c is developed, accurate effects 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:

Work	Shared memory	Object-oriented	Effect annotation	Deployment time	Type information
Open effects	**Yes**	**Yes**	**No**	**Hybrid**	**Partial dynamic types**
DPJ [34,7]	**Yes**	**Yes**	**Yes**	**Static**	**Static types**
SvS [6]	**Yes**	**No**	**No**	**Hybrid**	**Partial dynamic types**
TM [42,38,23,28]	**Yes**	**Yes**	**No**	**Dynamic**	**Full dynamic types**
DTM etc. [5,46,14]	**Yes**	**Yes**	**No**	**Dynamic**	**Full dynamic types**
FastTrack [16], CP [40]	**Yes**	**Yes**	**No**	**Dynamic**	**Full dynamic types**
Goldilocks [15]	**Yes**	**Yes**	**No**	**Hybrid**	**Full dynamic types**
Galois [30]	**Yes**	**Yes**	**No**	**Dynamic**	**Full dynamic types**
Revision [10]	**Yes**	**Yes**	**No**	**Dynamic**	**Full dynamic types**
Actor [3]	**No**	**Yes**	**No**	**Static**	**Static types**

The criteria labeled shared memory, object-oriented and effect annotation are self-explanatory. All of the approaches except actor-based approaches are for shared memory models, 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 the 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 a 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 are placeholder effects, 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 satisfy that property statically but verified to be true when the dynamic type of the receiver is known. Open effects have several benefits. They enables modular analysis of partial programs and libraries. They have 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 0.44-9.1% overhead and good speedup. In the 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.

References

1. JSR-166y for Java 7. `http://gee.oswego.edu/dl/concurrency-interest/`
2. Abadi, M., Flanagan, C., Freund, S.N.: Types for safe locking: Static race detection for Java. TOPLAS '06 28
3. Agha, G., Hewitt, C.: Concurrent programming using actors: Exploiting large-scale parallelism. In: FSTTCS '85
4. Benton, N., Buchlovsky, P.: Semantics of an effect analysis for exceptions. In: TLDI '07
5. Berger, E.D., Yang, T., Liu, T., Novark, G.: Grace: safe multithreaded programming for C/C++. In: OOPSLA '09
6. Best, M.J., Mottishaw, S., Mustard, C., Roth, M., Fedorova, A., Brownsword, A.: Synchronization via scheduling: techniques for efficiently managing shared state. In: PLDI '11
7. Bocchino, R.L., Adve, V.S.: Types, regions, and effects for safe programming with object-oriented parallel frameworks. In: ECOOP '11
8. Bond, M.D., Coons, K.E., McKinley, K.S.: Pacer: proportional detection of data races. In: PLDI '11
9. Boyapati, C., Lee, R., Rinard, M.: Ownership types for safe programming: preventing data races and deadlocks. In: OOPSLA '02
10. Burckhardt, S., Baldassin, A., Leijen, D.: Concurrent programming with revisions and isolation types. In: OOPSLA '10
11. Cameron, N.R., Drossopoulou, S., Noble, J., Smith, M.J.: Multiple ownership. In: OOPSLA '07
12. Clarke, D., Drossopoulou, S.: Ownership, encapsulation and the disjointness of type and effect. In: OOPSLA '02
13. Clifton, C., Leavens, G.T.: MiniMAO_1: Investigating the semantics of proceed. SCP 63(3), 321–374 (2006)
14. Ding, C., Shen, X., Kelsey, K., Tice, C., Huang, R., Zhang, C.: Software behavior oriented parallelization. In: PLDI '07

15. Elmas, T., Qadeer, S., Tasiran, S.: Goldilocks: a race and transaction-aware Java runtime. In: PLDI '07
16. Flanagan, C., Freund, S.N.: FastTrack: efficient and precise dynamic race detection. In: PLDI '09
17. Flanagan, C., Freund, S.N.: Type-based race detection for Java. In: PLDI '00
18. Flatt, M., Krishnamurthi, S., Felleisen, M.: A Programmer's Reduction Semantics for Classes and Mixins. In: Formal Syntax and Semantics of Java. Springer (1999)
19. Gifford, D.K., Lucassen, J.M.: Integrating functional and imperative programming. In: LFP '86
20. Gordon, C.S., Parkinson, M.J., Parsons, J., Bromfield, A., Duffy, J.: Uniqueness and reference immutability for safe parallelism. In: OOPSLA '12
21. Goyal, D.: An improved intra-procedural may-alias analysis algorithm. Tech. rep., New York, NY, USA (1999)
22. Greenhouse, A., Boyland, J.: An object-oriented effects system. In: ECOOP '99
23. Herlihy, M., Moss, J.E.B.: Transactional memory: architectural support for lock-free data structures. In: ISCA '93
24. Kassios, I.T.: Dynamic frames: Support for framing, dependencies and sharing without restrictions. In: FM '06
25. Kleinberg, J., Tardos, E.: Algorithm Design. Addison Wesley (2005)
26. Knowles, K., Flanagan, C.: Hybrid type checking. TOPLAS '10, 32
27. Leroy, X., Pessaux, F.: Type-based analysis of uncaught exceptions. TOPLAS '00, 22(2)
28. Lesani, M., Palsberg, J.: Communicating memory transactions. In: POPL '11
29. Lucassen, J.M., Gifford, D.K.: Polymorphic effect systems. In: POPL '88
30. M. Kulkarni *et al.*: Optimistic parallelism requires abstractions. In: PLDI '07
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
32. Park, Y.G., Goldberg, B.: Escape analysis on lists. In: PLDI '92
33. Pluquet, F., Langerman, S., Wuyts, R.: Executing code in the past: efficient in-memory object graph versioning. In: OOPSLA '09
34. R. Bocchino *et al.*: A type and effect system for deterministic parallel Java. In: OOPSLA '09
35. Rinard, M.C., Diniz, P.C.: Commutativity analysis: a new analysis framework for parallelizing compilers. In: PLDI '96
36. Rugina, R., Rinard, M.: Automatic parallelization of divide and conquer algorithms. In: PPoPP '99
37. Salcianu, R.D., Rinard, M.C.: Purity and side effect analysis for Java programs. In: VMCAI '05
38. Shavit, N., Touitou, D.: Software transactional memory. In: PODC '95
39. Siek, J., Taha, W.: Gradual typing for objects. In: ECOOP '07
40. Smaragdakis, Y., Evans, J.M., Sadowski, C., Jaeheon, Y., Flanagan, C.: Sound predictive race detection in polynomial time. In: POPL '12
41. Solodkyy, Y., Dos Reis, G., Stroustrup, B.: Open and efficient type switch for c++. In: OOPSLA '12
42. T. Shpeisman *et al.*: Enforcing isolation and ordering in STM. In: PLDI '07
43. Talpin, J.P.: Theoretical and Practical Aspects of Type and Effect Inference. Ph.D. thesis, Ecole des Mines de Paris and University Paris VI (1993)
44. Talpin, J.P., Jouvelot, P.: Polymorphic type, region and effect inference. JFP '92, 2(3)
45. Thies, W., Karczmarek, M., Amarasinghe, S.P.: Streamit: A language for streaming applications. In: CC '02
46. Welc, A., Jagannathan, S., Hosking, A.: Safe Futures for Java. In: OOPSLA '05

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 says that the entire program type checks if all the declarations type check and the expression e has any type t and any effect σ.

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

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

$$\frac{\text{(T-CLASS)}\quad \forall[\texttt{@open}]\ t_i\ f_i \in \overline{field} : validF(f_i,d) \qquad isClass(d) \qquad \forall meth_j \in \overline{meth} \vdash\ meth_j : (t_j,\sigma_j) \text{ in } c}{\vdash\ \texttt{class}\ c\ \texttt{extends}\ d\ \{\overline{field}\ \ \overline{meth}\} : \text{OK}}$$

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

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

$$\frac{\texttt{class}\ c\ \texttt{extends}\ d\ \{\overline{field}\ \overline{meth}\} \in CT}{isClass(c)} \qquad \frac{isClass(t) \vee (t = int)}{isType(t)}$$

9.2 Type-and-Effect Rules for Expressions

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

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

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

$$\text{(T-ADD)}\ \frac{\Pi \vdash e_1 : (int, \sigma) \qquad \Pi \vdash e_2 : (int, \sigma')}{\Pi \vdash e_1 + e_2 : (int, \sigma \cup \sigma')} \qquad \text{(T-NUM)}\ \Pi \vdash n : (int, \emptyset)$$

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 *typeOfF* (used in the rules in §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.

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

$$\begin{array}{l} fieldOf(\texttt{@open c f}) = (f, \texttt{@open c}) \\ fieldOf(c\ f) = (f,c) \end{array}$$

10 Dynamic Semantics: Omitted Details

This section presents auxiliary functions that were omitted in §4 for brevity.

The *fields* function, used in the (NEW) rule, returns all the fields declared in the class and its super classes (it uses the *fieldOf* function defined in §9.2).

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

The function *fixPoint*, used in the (SET) rule is shown below. It calls the *update* function in §4.2 until the store μ, or more specifically the effects in the store, does not

change. The *update* is called on all the loc_i and field f pairs that are pointing to the *loc*, whose effects have been changed. The effects of loc_i are changed by calling the *update* function.

$$\begin{array}{l} \mathit{fixPoint}(\mu, loc, \kappa) = \mu_n \quad \textbf{where } \kappa = \{\langle loc_i, f_i\rangle \,|\, 1 \leq i \leq n\} \\ \qquad \textbf{and } \mathit{update}(\mu, loc_1, f_1, loc) = \mu_1 \\ \qquad \textbf{and } \forall\, i \in \{2..n\} \;::\; \mathit{update}(\mu, loc_{i-1}, f_{i-1}, loc) = \mu_i \end{array}$$

Below shows the standard rule for variable declaration and integer addition.

$$\text{(DEFINE)}\quad \langle \mathbb{E}[t\ var = v; e], \mu\rangle \hookrightarrow \langle \mathbb{E}[[v/var]e], \mu\rangle$$

$$\text{(ADD)}\quad \frac{v = v_1 + v_2}{\langle \mathbb{E}[v_1 + v_2], \mu\rangle \hookrightarrow \langle \mathbb{E}[v], \mu\rangle}$$

11 Proof of Key Properties

We now prove the key properties of *OpenEffectJ*: Effect and Type Preservation. Some of the definitions, descriptions and proof sketchs are also in §4.4. We write all these for the sake of clarity.

We have proven the soundness of *OpenEffectJ*'s type system (§11.3 contains proof that use the standard subject reduction argument [18]). 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 §11.2. Proving effect soundness is non-trivial compared to static effect approaches [34,12], 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 χ, which contains a sequence of dynamic effects (heap accesses) by the tasks.

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

The function *dyn*, defined in Figure 12, records the dynamic memory footprint for the evaluatioin of an expression *e*. With it, we can prove that the dynamic effects of each expression refines the static effect σ computed when it is evaluated.

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.

In the following, $dyn(\Sigma,\chi) = dyn(\Sigma',\chi')$ and $\Sigma \hookrightarrow \Sigma'$.	
Σ	**Side Conditions**
$\langle \mathbb{E}[loc.f], \mu \rangle$	$\chi' = \chi + (\texttt{rd}\ loc\ \texttt{f})$
$\langle \mathbb{E}[loc.f = v], \mu \rangle$	$\chi' = \chi + (\texttt{wt}\ loc\ \texttt{f})$
Other cases	$\chi' = \chi$

Fig. 12. Dynamic Effect function *dyn*.

Definition 5. *[Static effect inclusion] An effect ε 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. $(\varepsilon_i = \texttt{open}\ f\ m\ \sigma') \wedge (\sigma' = \{\varepsilon'_j | 1 \leq j \leq n'\}) \wedge (\varepsilon = \varepsilon'_j)$.*

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

Definition 6. *[Dynamic effect refines static effect] A dynamic effect η refines a static effect σ, written $\eta \propto \sigma$, if either $\eta = (\texttt{rd}\ loc\ f) \wedge ((\texttt{read}\ f) \in \sigma \vee (\texttt{write}\ f) \in \sigma)$; or $\eta = (\texttt{wt}\ loc\ f) \wedge (\texttt{write}\ f) \in \sigma$.*

In §11.2, we will show that during the evaluation, the effect σ of an expression e, is refined by the effect σ' of its subsequent expression e', i.e., if $\langle e, \mu \rangle \hookrightarrow \langle e', \mu' \rangle$, $\mu \vdash e : \sigma$ 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 σ' refines another effect σ.

Definition 7. *[Static effect refinement] An effect set σ' refines another effect set σ 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 μ and μ' are effect equivalent, written $\mu \cong \mu'$, if both conditions hold: $dom(\mu) \subseteq dom(\mu')$; and $\forall loc$ if $\mu(loc) = [c.F.E]$, then $\mu'(loc) = [c.F'.E]$, for some F'.*

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

Except for the method call expression, proving that an expression has static effects that are refined by their subsequent expression is standard [34,12]. 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 *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 $o = [c.F.E]$ is a well-formed object in μ, written $\mu \vdash o$, if for all* open *effect* $\texttt{open}\ f\ m\ \sigma_0 \in \sigma \in rng(E)$, *either* $(F(f) = loc) \wedge (\mu(loc) = [c'.F'.E']) \wedge (E'(m) \subseteq \sigma_0)$; *or* $(F(f) = \texttt{null}) \wedge (\sigma_0 = \emptyset)$.

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

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

$$\text{(E-CALL-LOC)}\ \frac{\mu(loc) = [c.F.E] \quad E(m) = \sigma_0 \quad (\forall\, i \in \{1..n\}\ ::\ \mu \vdash e_i : \sigma_i)}{\mu \vdash loc.m(\overline{e}) : \bigcup_{i=0}^{n} \sigma_i}$$

$$\text{(E-CALL-OPEN)}\ \frac{\mu \vdash loc.f : \sigma_0 \quad \mu(loc) = [c.F.E] \quad E = \{m_i \mapsto \sigma_i \| 1 \le i \le n\} \quad \exists i \text{ s.t. } (\exists \varepsilon = \texttt{open f m}\,\sigma \text{ s.t. } \varepsilon \in \sigma_i) \quad \mathit{typeOfF}(f) = (d, \texttt{@open}\ c_0) \quad (\forall\, i \in \{1..n\}\ ::\ \mu \vdash e_i : \sigma_i)}{\mu \vdash loc.f.m(\overline{e}) : \sigma \cup \bigcup_{i=0}^{n} \sigma_i}$$

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

$$\text{(E-GET)}\ \frac{\mu \vdash e : \sigma}{\mu \vdash e.f : \sigma \cup \{\texttt{read f}\}} \qquad \text{(E-SET-OPEN)}\ \frac{\mathit{typeOfF}(f) = (c, \texttt{@open}\ c_0)}{\mu \vdash e.f = e' : \{\bot\}} \qquad \text{(E-SET)}\ \frac{\mu \vdash e' : \sigma' \quad \mu \vdash e : \sigma \quad \mathit{typeOfF}(f) = (c,t)}{\mu \vdash e.f = e' : \sigma \cup \sigma' \cup \{\texttt{write f}\}}$$

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

$$\text{(E-DEFINE)}\ \frac{\mu \vdash e_1 : \sigma_1 \quad \mu \vdash e_2 : \sigma_2}{\mu \vdash c\ var = e_1; e_2 : \sigma_1 \cup \sigma_2} \qquad \text{(E-ADD)}\ \frac{\mu \vdash e_1 : \sigma_1 \quad \mu \vdash e_2 : \sigma_2}{\mu \vdash e_1 + e_2 : \sigma_1 \cup \sigma_2} \qquad \text{(E-NUMBER)}\ \mu \vdash n : \emptyset$$

Fig. 13. Effect judgment for expressions.

The dynamic rules, in Figure 13 are similar to the static rules in §3, except for method calls on *open* fields (E-CALL-OPEN). The *open* effect now has a concrete part σ, instead of $\emptyset$, because we know the concrete object that an *open* field is pointing to.

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

Definition 10. *[Well-formed location] A location loc is well-formed in μ, written $\mu \vdash loc$, if either $\mu(loc) = [c.F.E]$, $\forall m \in dom(E)$ s.t. $findMeth(c,m) = (c', t, m(\overline{t\ var})\{e\}, \sigma') \wedge \mu \vdash [loc/\textbf{this}]e : \sigma$, then $\sigma \subseteq E(m)$; or $\mu(loc) = \textbf{null}$.*

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

Finally, to prove the effect preservation theorem (Theorem 2), 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 μ is well-formed, written $\mu \vdash \diamond$, if $\forall o \in rng(\mu)$ s.t. $\mu \vdash o$ and $\forall loc \in dom(\mu)$ s.t. $\mu \vdash loc$.*

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

To show the effect preservation Theorem 1, we need to ensure that the store is well-formed throughout the program execution (Lemma 3). The initial store, when the program starts, is empty $\bullet$ and $\bullet \vdash \diamond$. Therefore, it suffices to show that if the store μ before the transition is well-formed, then the store μ' after the transition is also well-formed.

Lemma 2. [Stores preservation] *If $\Sigma_\star \hookrightarrow^* \langle e', \mu' \rangle$, where $\Sigma_\star = \langle e, \bullet \rangle$, then $\mu' \vdash \diamond$.*

The proof is by cases on the reduction step applied. In each case we show that $\mu \vdash \diamond$ implies that $\mu' \vdash \diamond$. The cases (DEFINE), (CALL), (ADD) and (GET) are obvious, because they do not change the store, i.e., $\mu' = \mu$.

For all the remaining cases, to see $\mu' \vdash loc$, consider the definition of *methE*. It returns the effects computed by the static type-and-effect system, while the effect judgment is more accurate (Figure 13). I.e., by observation if $(\overline{var : t}, \texttt{this} : c) \vdash e : (u, \sigma)$ and $\mu \vdash [loc/\texttt{this}]e : \sigma'$, then $\sigma' \subseteq \sigma$, therefore $\mu' \vdash loc$. Therefore, it suffices to show all the objects o are well-formed, i.e., $\mu' \vdash o$.

New. Here $e = \mathbb{E}[new\ c()]$, $e' = \mathbb{E}[loc]$, where $loc \notin dom(\mu)$, $\mu' = \{loc \mapsto [c.\{f \mapsto \texttt{null} \mid f \in fields(c)\}.\{m \mapsto \sigma \in methE(c)\}]\} \oplus \mu$. The only change to the store *mu* is the new object o created: $[c.\{f \mapsto \texttt{null} \mid f \in fields(c)\}.\{m \mapsto \sigma \in methE(c)\}]$. All the fields are initiated to `null`, i.e., $\{f \mapsto \texttt{null} \mid f \in fields(c)\}$. By the definition of *methE* (§4.2), all the *open* effects are initiated to `null`. Therefore, $\mu' \vdash o$.

Set. Here $e = \mathbb{E}[loc.f = v]$, $e' = \mathbb{E}[v]$, $\mu' = \mu \oplus (loc \mapsto o)$, and $o = [u.F \oplus (f \mapsto v).E]$, where $\mu(loc) = [u.F.E]$ and $typeOfF(f) = (c,t)$ for some t. The field is not an *open* field, and by the function *update*, it does not update any effect, and $\mu' \vdash o$.

Field Set Open. Here $e = \mathbb{E}[loc.f = v]$, $e' = \mathbb{E}[v]$, where $\mu_0 = \mu \oplus (loc \mapsto [c.(F \oplus (f \mapsto v)).E])$, and $\mu' = update(\mu_0, loc, f, v)$. The proof is by observation/construction of the *update* auxiliary function. Each time it updates an object, it copied the corresponding effects of updated object and put it in the *open* effect (see the *concretize* function).

Thus, for all possible reduction steps, we see if $\mu \vdash \diamond$, then $\mu' \vdash \diamond$. ∎

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 σ produced by an expression e is refined by the effect σ', by its subsequence expression e' ($\sigma' \subseteq \sigma$); and second, the dynamic effect η, produced by the reduction, if any, refines σ ($\eta \propto \sigma$); finally, the store remains effect equivalent ($\mu \cong \mu'$).

Theorem 2. [Effect preservation] *Let the program configuration $\Sigma = \langle e, \mu \rangle$. If it transits to another configuration $\Sigma \hookrightarrow \langle e', \mu' \rangle$, the store is well-formed $\mu \vdash \diamond$, and $\mu \vdash e : \sigma$, then there is some σ', χ s.t.*

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

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 σ 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 $\mu \cong \mu'$. Therefore, it suffices to prove that the effect of the subsequent expression e' refines the effect of the expression e, given by effect equivalent stores.

Lemma 3. [Stationary effect] *Let e be an expression, μ and μ' two stores s.t. $\mu \cong \mu'$. 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 σ. 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 (NEW), (NULL), (LOC), (NUMBER) and (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 (ADDITION), (DEFINE), (GET) and (SET) follow directly from IH.

(ADDITION). Here $e = e_1 + e_2$. The last step is:

$$\frac{\mu \vdash e_1 : \sigma_1 \qquad \mu \vdash e_2 : \sigma_2}{\mu \vdash e_1 + e_2 : \sigma_1 \cup \sigma_2} \qquad \frac{\mu' \vdash e_1 : \sigma_1' \qquad \mu' \vdash e_2 : \sigma_2'}{\mu' \vdash e_1 + e_2 : \sigma_1' \cup \sigma_2'}$$

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.

(DEFINE). Here $e = c\ var = e_1; e_2$. The last step is:

$$\frac{\mu \vdash e_1 : \sigma_1 \qquad \mu \vdash e_2 : \sigma_2}{\mu \vdash c\ var = e_1; e_2 : \sigma_1 \cup \sigma_2} \qquad \frac{\mu' \vdash e_1 : \sigma_1' \qquad \mu' \vdash e_2 : \sigma_2'}{\mu' \vdash c\ var = e_1; e_2 : \sigma_1' \cup \sigma_2'}$$

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 $e = e'.f$. The last derivation step is:

$$\frac{\mu \vdash e' : \sigma_0}{\mu \vdash e'.f : \sigma_0 \cup \{\texttt{read f}\}} \qquad \frac{\mu' \vdash e' : \sigma_0'}{\mu' \vdash e'.f : \sigma_0' \cup \{\texttt{read f}\}}$$

By the IH, $\sigma_0' = \sigma_0$. Therefore $\sigma' = \sigma_0' \cup \{\texttt{read f}\} = \sigma_0 \cup \{\texttt{read f}\} = \sigma$, and the claim holds.

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

$$\frac{\mu \vdash e_1 : \sigma_1 \qquad \mu \vdash e_0 : \sigma_0 \qquad \mathit{typeOfF}(f) = (c,t)}{\mu \vdash e_0.f = e_1 : \sigma_0 \cup \sigma_1 \cup \{\texttt{write f}\}} \qquad \frac{\mu' \vdash e_1 : \sigma_1' \qquad \mu' \vdash e_0 : \sigma_0' \qquad \mathit{typeOfF}(f) = (c,t)}{\mu' \vdash e_0.f = e_1 : \sigma_0' \cup \sigma_1' \cup \{\texttt{write f}\}}$$

By the induction hypothesis, $\sigma_0' \subseteq \sigma_0$ and $\sigma_1' \subseteq \sigma_1$. Thus $\sigma' = \sigma_0' \cup \sigma_1' \cup \{\texttt{write f}\} \subseteq \sigma_0 \cup \sigma_1 \cup \{\texttt{write f}\} = \sigma$, and the claim holds.

(SET-OPEN). Here the last derivation step is:

$$\frac{\mathit{typeOfF}(f) = (c, \texttt{@open t})}{\mu \vdash e_0.f = e_1 : \{\bot\}} \qquad \frac{\mathit{typeOfF}(f) = (c, \texttt{@open t})}{\mu' \vdash e_0.f = e_1 : \{\bot\}}$$

The claim holds clearly.

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

$$\frac{\begin{array}{c}\mu(\mathit{loc}) = [c.F.E] \qquad \mu \vdash \mathit{loc}.f : \sigma_0 \\ E = \{m_i \mapsto \sigma_i \| 1 \le i \le n\} \\ \exists i \text{ s.t. } (\exists \varepsilon = \texttt{open f m}\,\sigma' \in \sigma_i) \\ \mathit{typeOfF}(f) = (c, \texttt{@open}\ c_0) \\ (\forall\, i \in \{1..n\} \ ::\ \mu \vdash e_i : \sigma_i)\end{array}}{\mu \vdash \mathit{loc}.f.m(\overline{e}) : \{\texttt{open f m}\,\sigma'\} \cup \bigcup_{i=0}^{n} \sigma_i} \qquad \frac{\begin{array}{c}\mu'(\mathit{loc}) = [c.F'.E] \qquad \mu' \vdash \mathit{loc}.f : \sigma_0' \\ E = \{m_i \mapsto \sigma_i \| 1 \le i \le n\} \\ \exists i \text{ s.t. } (\exists \varepsilon = \texttt{open f m}\,\sigma' \in \sigma_i) \\ \mathit{typeOfF}(f) = (c, \texttt{@open}\ c_0) \\ (\forall\, i \in \{1..n\} \ ::\ \mu' \vdash e_i : \sigma_i')\end{array}}{\mu' \vdash \mathit{loc}.f.m(\overline{e}) : \{\texttt{open f m}\,\sigma'\} \cup \bigcup_{i=0}^{n} \sigma_i'}$$

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

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

$$\frac{\begin{array}{c}\mu(\mathit{loc}) = [c.F.E] \qquad E(m) = \sigma_0 \\ (\forall\, i \in \{1..n\} \ ::\ \mu \vdash e_i : \sigma_i)\end{array}}{\mu \vdash \mathit{loc}.m(\overline{e}) : \bigcup_{i=0}^{n} \sigma_i} \qquad \frac{\begin{array}{c}\mu'(\mathit{loc}) = [c.F'.E] \qquad E(m) = \sigma_0' \\ (\forall\, i \in \{1..n\} \ ::\ \mu' \vdash e_i : \sigma_i')\end{array}}{\mu' \vdash \mathit{loc}.m(\overline{e}) : \bigcup_{i=0}^{n} \sigma_i'}$$

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.

(CALL). Here $e = e_0.m(\overline{e})$. The last type derivation step has the following form:

$$\mu \vdash e_0.m(\overline{e}) : \bot \qquad\qquad \mu' \vdash e_0.m(\overline{e}) : \bot$$

Obviously, the claim holds.

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

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

Lemma 4. [Replacement with subeffect] *If* $\mu \vdash \diamond$, $\Sigma \hookrightarrow \Sigma'$, $\Sigma = \langle \mathbb{E}[e], \mu \rangle$, $\Sigma' = \langle \mathbb{E}[e'], \mu' \rangle$, $\mu \vdash \mathbb{E}[e] : \sigma$, $\mu \vdash e : \sigma_0$, $\mu \vdash e' : \sigma_1$, $\mu \cong \mu'$, *and* $\sigma_1 \subseteq \sigma_0$, *then* $\mu \vdash \mathbb{E}[e'] : \sigma' \wedge \sigma' \subseteq \sigma$.

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 $\sigma' = \sigma_1 \subseteq \sigma_0 = \sigma$. For the induction step we divide the evaluation context into two parts such that $\mathbb{E}[e_1] = \mathbb{E}_1[\mathbb{E}_2[e_2]]$, and $\mathbb{E}_2$ has size one. The induction hypothesis (IH) is that the lemma holds for all evaluation contexts, which is smaller than the one ($\mathbb{E}_1$) considered in the induction step. We prove it case by case on the rule used to generate $\mathbb{E}_2$. In each case we show that $\mu \vdash \mathbb{E}_2[e] : \sigma$ implies that $\mu' \vdash \mathbb{E}_2[e'] : \sigma'$, for some $\sigma' \subseteq \sigma$, and thus the claim holds by the IH.

The cases for (E-ADD), (E-GET) and (E-DEFINE) follow directly from the IH.

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

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

$$\frac{\mu \vdash e : \sigma_0 \qquad \mathit{typeOfF}(f) = (c,t) \qquad \mu \vdash e_2 : \sigma_2}{\mu \vdash \mathbb{E}_2[e] : \sigma_0 \cup \sigma_2 \cup \{\texttt{write f}\}}$$

By the definition of field lookup, $\mathit{typeOfF}(f)$ remains unchanged, i.e. $\mathit{typeOfF}(f) = (c,t)$. Thus, by (E-SET), $\mu' \vdash \mathbb{E}_2[e'] : \sigma_1 \cup \sigma_2 \cup \{\texttt{write f}\}$;

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

$$\frac{\mathit{typeOfF}(f) = (c,t) \qquad \mu \vdash e : \sigma_0}{\mu \vdash \mathbb{E}_2[e] : (u, \sigma_0 \cup \{\texttt{write f}\})}$$

So $\mu' \vdash \mathbb{E}_2[e'] : \sigma_1 \cup \{\texttt{write f}\}$;

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

$$\frac{\begin{array}{ccc} & e = loc.f & \\ \mu(loc) = [c.F.E] \qquad E = \{m_i \mapsto \sigma_i \| 1 \le i \le n\} & & \exists i \text{ s.t. } (\exists \varepsilon = \texttt{open f m } \sigma'' \in \sigma_i) \\ \mathit{typeOfF}(f) = (c, \texttt{@open } c_0) & \mu \vdash loc.f : \sigma_0' & (\forall\, i \in \{1..n\} \;::\; \mu \vdash e_i : \sigma_i') \end{array}}{\Pi \vdash \mathbb{E}_2[e] : \{\texttt{open f m } \sigma''\} \cup \bigcup_{i=0}^{n} \sigma_i'}$$

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

$$\frac{e' = loc' \qquad \mu(loc') = [c'.F'.E'] \qquad E(m) = \sigma_0'' \qquad (\forall\, i \in \{1..n\} \;::\; \mu \vdash e_i : \sigma_i'')}{\Pi \vdash \mathbb{E}_2[e'] : \bigcup_{i=0}^{n} \sigma_i''}$$

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

$$\frac{E(m)=\sigma'' \qquad \mu(e)=\sigma_0 \qquad \begin{array}{c}\mu(loc)=[c.F.E]\\ (\forall\, i \in \{(p+1)..n\} \;::\; \mu \vdash e_i : \sigma_i'')\end{array}}{\mu \vdash \mathbb{E}_2[e] : \sigma_0 \cup \sigma'' \cup \bigcup_{i=(p+1)}^{n} \sigma_i}$$

By (E-CALL-LOC), $\mu' \vdash \mathbb{E}_2[e'] : \sigma_1 \cup \sigma'' \cup \bigcup_{i=(p+1)}^{n} \sigma_i$.

Using the lemmas. To prove Theorem 11.2, in each reduction case, let $e = \mathbb{E}[e_0]$, $e' = \mathbb{E}[e_1]$, $\mu \vdash e_0 : \sigma_0$ and $\mu' \vdash e_1 : \sigma_1$. Given that (a) $\mu \cong \mu'$, by Lemma 4 and Lemma 3, to prove (b), 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 §11.2; the cases, in the second category, access the store, in §11.2; and the other cases are listed right below. Here the rule leaves no dynamic trace, and (c) holds.

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

Addition:. Here $e = \mathbb{E}[v_1 + v_2]$, $e' = \mathbb{E}[v]$, where $v = v_1 + v_2$, $\mu' = \mu$. It is trivial to see that (b) holds.

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

Lemma 5. [Substitution effect] *If* $\mu \vdash e : \sigma$*, then there is some* σ'*, such that* $\mu \vdash [v_1/var_1,\ldots,v_n/var_n]e : \sigma'$*, for all values* v_i *and free variables* var_i*, and* $\sigma' \subseteq \sigma$.

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

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

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

$$\frac{\begin{array}{c}\mu(loc) = [c.F.E] \\ E = \{m_i \mapsto \sigma_i \| 1 \le i \le n\} \qquad \exists i \text{ s.t. } (\exists \varepsilon = \texttt{open f m}\,\sigma'' \text{ s.t. } \varepsilon \in \sigma_i) \\ typeOfF(f) = (d, \texttt{@open}\ c_0) \qquad \mu \vdash loc.f : \sigma_0 \qquad (\forall\, i \in \{1..n\}\ ::\ \mu \vdash e_i : \sigma_i)\end{array}}{\mu \vdash loc.f.m(e_1,\ldots,e_n) : \{\texttt{open f m}\,\sigma''\} \cup \bigcup_{i=0}^{n} \sigma_i}$$

Let $e'_i = [\overline{v/var}]e_i$ for $i \in \{1..n\}$, $[\overline{v/var}]e = loc.f.m(\overline{e'})$. We show that $\mu \vdash [\overline{v/var}]e : \{\texttt{open f m}\,\sigma''\} \cup \bigcup_{i=0}^{n} \sigma'_i$, where $\forall i \in \{0..n\}\ \sigma'_i \subseteq \sigma_i$. Because $loc.f$ has no free variable, $\sigma'_0 = \sigma_0$ and $\{\texttt{open f m}\,\sigma''\}$ 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 $e = loc.m(\overline{e})$. The last step is:

$$\frac{\mu(loc) = [c.F.E] \qquad E(m) = \sigma_0 \qquad (\forall\, i \in \{1..n\}\ ::\ \mu \vdash e_i : \sigma_i)}{\mu \vdash loc.m(e_1,\ldots,e_n) : \bigcup_{i=0}^{n} \sigma_i}$$

Let $e'_i = [\overline{v/var}]e_i$ for $i \in \{1..n\}$, then $[\overline{v/var}]e = loc.m(\overline{e'})$. We show that $\mu \vdash [\overline{v/var}]e : \bigcup_{i=0}^{n} \sigma'_i$, 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 $e = e_0.f = e_1$. The last derivation step is:

$$\frac{\mu \vdash e_0 : \sigma_0 \qquad typeOfF(f) = (c,t) \qquad \mu \vdash e_1 : \sigma_1}{\mu \vdash e_0.f = e_1 : \sigma_0 \cup \sigma_1 \cup \{\texttt{write f}\}}$$

Now $[\overline{v/var}]e = ([\overline{v/var}]e_0.f = [\overline{v/var}]e_1)$. By IH, $\mu \vdash [\overline{v/var}]e_0 : \sigma'_0$, and $\mu \vdash [\overline{v/var}]e_1 : \sigma'_1$, where $\sigma'_0 \subseteq \sigma_0$ and $\sigma'_1 \subseteq \sigma_1$. By the definition of *typeOfF*, the result of $typeOfF(f)$ remains unchanged, i.e. $typeOfF(f) = (c,t)$. Therefore $\Pi \vdash [\overline{v/var}]e : \sigma'_0 \cup \sigma'_1 \cup \{\texttt{write f}\}$, and it holds.

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

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

Method Call:. Here $e = \mathbb{E}[loc.m(\overline{v})]$, $(u', t_m, m(\overline{t\ var})\{e_2\}, \sigma_m) = findMeth(u,m)$, $e' = \mathbb{E}[e_1]$, $e_1 = [loc/\texttt{this}, \overline{v/var}]e_2$, $\mu(loc) = [u.F.E]$. Let $\mu \vdash loc.m(\overline{v}) : \sigma_0$, i.e., $E(m) = \sigma_0$. Let $e_3 = [loc/\texttt{this}]e_2$, $\mu \vdash e_3 : \sigma_3$ and $\mu \vdash e_1 : \sigma_1$. By Lemma 5, $\sigma_1 \subseteq \sigma_3$. By $\mu \vdash \diamond$, Definition 10 and Definition 11, $\sigma_3 \subseteq \sigma_0$, thus $\sigma_1 \subseteq \sigma_0$.

Local Declaration:. Here $e = \mathbb{E}[t\ var = v; e_1]$, and $e' = \mathbb{E}[e'_1]$, where $e'_1 = [v/var]e_1$. Let $\mu \vdash e_1 : \sigma_0$, by (E-DEFINE), $\mu \vdash t\ var = v; e_1 : \sigma_0$. $\mu \vdash\ [v/var]e_1 : \sigma_1$, for some $\sigma_1 \subseteq \sigma_0$, by Lemma 5.

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 σ of subexpression e is a subset $\subseteq$ of the effect σ' of its entire expression $\mathbb{E}[e]$.

Lemma 6. [Subexpression effect containment] *If* $\mu \vdash e : \sigma$ *and* $\mu \vdash \mathbb{E}[e] : \sigma'$, *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 $e = \mathbb{E}[loc.f]$, $e' = \mathbb{E}[v]$, where $\mu(loc) = [u.F.E]$, $F(f) = v$, $\mu' = \mu$ and $\mu \cong \mu'$. Because $\mu \vdash loc.f : \{\texttt{read f}\}$, and $\mu' \vdash v : \emptyset$, (b) holds. Finally, $\eta = (\texttt{rd}\ loc\ \texttt{f})$, and $\eta \propto \{\texttt{read f}\} \subseteq \sigma$, by Lemma 6.

Field Set:. Here $e = \mathbb{E}[loc.f = v]$, $e' = \mathbb{E}[v]$, $\mu' = \mu \oplus (loc \mapsto o)$, and $o = [u.F \oplus (f \mapsto v).E]$, where $\mu(loc) = [u.F.E]$ and $typeOfF(f) = (c,t)$ for some t. The field is not an *open* field, and by the function *update*, it does not update any effect, and $\mu \cong \mu'$. To see $\mu \vdash \mathbb{E}[v] : \sigma' \subseteq \sigma$, we have $\mu \vdash loc.f = v : \{\texttt{write f}\}$, and $\mu \vdash v : \emptyset$, thus $\sigma' \subseteq \sigma$. Finally, $\eta = (\texttt{wt}\ loc\ \texttt{f})$, and $\eta \propto \{\texttt{write f}\} \subseteq \sigma$, by Lemma 6.

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

11.3 Type Soundness

In this section, we prove the standard type preservation property. Type rules omitted in §3 are in Figure 14. To prove the type preservation, we extend the type environment, which maps variables and locations to types.

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

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

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

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

$$\frac{\Pi \vdash loc.f : (c_0, \sigma_0) \qquad typeOfF(f) = (d, \texttt{@open}\ c_0) \qquad findMeth(c_0, m) = (c_1, t, m\ (\overline{t\ var})\ \{e_{n+1}\}, \sigma) \qquad (\forall\, i \in \{1..n\}\ ::\ \Pi \vdash e_i : (t_i', \sigma_i)\ \wedge\ t_i' <: t_i)}{\Pi \vdash loc.f.m(\overline{e}) : (t, \{\texttt{open f m}\,\emptyset\} \cup \bigcup_{i=0}^{n} \sigma_i)}\ \text{(T-CALL-OPEN-LOC)}$$

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

Fig. 14. Type and effect rules for loc.

Before proving the type preservation theorem, we define the consistency between a type environment and a store [18], which is standard. Also, we need to ensure that during the evaluation, all the expressions have proper types (Definition 14). Finally, we state the stardard lemmas [18,13] (Lemma 7, Lemma 8, Lemma 9 and Lemma 10).

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

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

Definition 13. *[Environment enlargement] Let* Π *and* Π' *be two type environments. We write* $\Pi \lessdot \Pi'$*, if* $dom(\Pi) \subseteq dom(\Pi')$ *and* $\forall a \in dom(\Pi)$*, if* $\Pi(a) = t$*, then* $\Pi'(a) = t$.

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

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

Lemma 7. [Substitution] *If* $\Pi, \overline{var : t} \vdash e : (t, \sigma)$ *and* $\forall i \in \{1..n\}$, $\Pi \vdash v_i : (s_i, \emptyset)$ *where* $s_i <: t_i$ *then* $\Pi \vdash [\overline{v/var}]e : (s, \sigma')$ *for some* $s <: t$ *and some* σ'.

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

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

The cases for (T-ADD) and (T-DEFINE) follow directly from the induction hypothesis.

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

$$\frac{\begin{array}{c} e'_0 = \texttt{this}.f \qquad typeOfF(f) = (d, \texttt{@open}\ c_0) \\ \Pi' \vdash e'_0 : (c_0, \sigma_0) \qquad findMeth(c_0, m) = (c_1, t, m\,(u_1\ var_1, \ldots, u_n\ var_n)\ \{\, e_{n+1} \,\}, \sigma_2) \\ (\forall\ i \in \{1..n\}\ ::\ \Pi' \vdash e'_i : (u'_i, \sigma_i)\ \wedge\ u'_i <: u_i) \end{array}}{\Pi' \vdash e'_0.m(e'_1, \ldots, e'_n) : (t, \{\texttt{open f m}\,\emptyset\} \cup \bigcup_{i=0}^{n} \sigma_i)}$$

Let $e_i'' = [\overline{v/var}]e_i'$ for $i \in \{0..n\}$, then $[\overline{v/var}]e = e_0''.m(\overline{e''})$. We show that $\Pi \vdash [\overline{v/var}]e : (t, \sigma')$. By IH, $\Pi \vdash e_0'' = (c_2, \sigma_0'')$, where $c_2 <: c_0$. If $(c_1, t, m\ (\overline{u\ var})\ \{e_{n+1}\}, \sigma_2) = \mathit{findMeth}(c_0, m)$ and $(c_3, t_2, m\ (\overline{u\ var})\ \{e'_{n+1}\}, \sigma_3) = \mathit{findMeth}(c_2, m)$, by the definitions of *findMeth* and *override*, $t_2 = t$. Also, by IH, $\forall i \in \{1..n\} :: \Pi \vdash e_i'' : (u_i'', \sigma_i)$ and $u_i'' <: u_i'$. Finally, $\forall i \in \{1..n\}\ ::\ u_i'' <: u_i$, by transitivity, the claim holds.

(T-Call). Here $e = e_0'.m(\overline{e'})$. The last type derivation step has the following form:

$$\frac{(c_1, t, m\ (u_1\ var_1, \ldots, u_n\ var_n)\ \{e_{n+1}\}, \sigma'') = \mathit{findMeth}(u_0', m) \quad \Pi' \vdash e_0' : (u_0', \sigma_0) \quad (\forall\, i \in \{1..n\}\ ::\ \Pi' \vdash e_i' : (u_i', \sigma_i)\ \wedge\ u_i' <: u_i)}{\Pi' \vdash e_0'.m(e_1', \ldots, e_n') : (t, \bot)}$$

Let $e_i'' = [\overline{v/var}]e_i'$ for $i \in \{0..n\}$, then $[\overline{v/var}]e = e_0''.m(\overline{e''})$. We show $\Pi \vdash [\overline{v/var}]e : (t, \sigma')$, for some σ'. By IH, $\Pi \vdash e_0'' : (u_0'', \sigma_0')$, where $u_0'' <: u_0'$. By the definitions of *findMeth* and *override*, if $\mathit{findMeth}(u_0', m) = (c_1, t, m(\overline{u\ var})(e'_{n+1}), \sigma'')$, and $\mathit{findMeth}(u_0'', m) = (c_2, t_2, m(\overline{t\ var})(e''_{n+1}), \sigma''')$, then $t_2 = t$. Also by IH $\forall i \in \{1..n\} :: \Pi \vdash e_i'' : (u_i'', \sigma_i')$ and $u_i'' <: u_i'$. Finally, $\forall i \in \{1..n\} :: u_i'' <: u_i$, by transitivity the claim holds.

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

$$\frac{\mathit{typeOfF}(f) = (d, \texttt{@open}\ c_0) \quad \Pi' \vdash loc.f : (c_0, \sigma_0) \quad \mathit{findMeth}(c_0, m) = (c_1, t, m\ (u_1\ var_1, \ldots, u_n\ var_n)\ \{e_{n+1}\}, \sigma'') \quad (\forall\, i \in \{1..n\}\ ::\ \Pi' \vdash e_i' : (u_i', \sigma_i)\ \wedge\ u_i' <: u_i)}{\Pi' \vdash loc.f.m(e_1', \ldots, e_n') : (t, \{\texttt{open f m}\emptyset\} \cup \bigcup_{i=0}^{n} \sigma_i)}$$

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

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

$$\frac{\Pi' \vdash \mathbf{this} : (c, \sigma') \quad \mathit{typeOfF}(f) = (d, [\texttt{@open}]\ t) \quad c <: d}{\Pi' \vdash \mathbf{this}.f : (t, \sigma_0 \cup \{\texttt{read f}\})}$$

Now $[\overline{v/var}]e = [\overline{v/var}]\mathbf{this}.f$. By IH, $\Pi \vdash [\overline{v/var}]\mathbf{this} : (u', \sigma_1)$, where $u' <: u$. By the definition of *typeOfF*, $\mathit{typeOfF}(f)$ does not change. Therefore $\Pi \vdash [\overline{v/var}]e : (t, \sigma_1 \cup \{\texttt{read f}\})$ and the claim holds.

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

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

$$\frac{\Pi' \vdash \mathbf{this} : (c, \sigma_1) \quad \mathit{typeOfF}(f) = (d, u) \quad c <: d \quad \Pi' \vdash e' : (t, \sigma_2) \quad t <: u}{\Pi' \vdash \mathbf{this}.f = e' : (t, \sigma_1 \cup \sigma_2 \cup \{\texttt{write f}\})}$$

Now $[\overline{v/var}]e = ([\overline{v/var}]\mathbf{this}.f = [\overline{v/var}]e')$. By IH, $\Pi \vdash [\overline{v/var}]\mathbf{this} : (u_1'', \sigma_1')$, where $u_1'' <: u_1'$; $\Pi \vdash [\overline{v/var}]e' : (u_2', \sigma_2')$, where $u_2' <: t$. By the definition of *typeOfF*, its result

does not change. By transitivity $t' = u'_2 <: t <: u$. Therefore $\Pi \vdash [\overline{v}/\overline{var}]e : (t', \sigma'_1 \cup \sigma'_2 \cup \{\texttt{write f}\})$, $t' <: t$. The claim holds.

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

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

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

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

Lemma 8. [Environment extension] *If* $\Pi \vdash e : (t, \sigma)$ *and* $a \notin dom(\Pi)$*, 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), (T-NUMBER), and (T-VAR). In (T-NEW) and (T-NULL), the type environment does not appear in the hypotheses of the judgment, so the claim holds. For the (T-VAR) case, $e = var$ and $\Pi(var) = t$. But $a \notin dom(\Pi)$, so $var \neq a$. Therefore $(\Pi, a : t')(var) = t$ and the claim holds for this case. The (T-LOC) case is similar. The remaining rules cover the induction step. By the induction hypothesis, changing the type environment to $\Pi, a : t'$ does not change the types and effects assigned by any hypotheses of each rule. Therefore, the types and effects assigned by each rule are also unchanged and the claim holds. ∎

Lemma 9. [Replacement] *If* $\Sigma = \langle \mathbb{E}[e], \mu \rangle$, $\Sigma' = \langle \mathbb{E}[e'], \mu' \rangle$, $\Sigma \hookrightarrow \Sigma'$, $\Pi \vdash \mathbb{E}[e] : (t, \sigma)$, $\Pi \vdash e : (t', \sigma')$ *and* $\Pi \vdash e' : (t', \sigma'_0)$*, then* $\Pi \vdash \mathbb{E}[e'] : (t, \sigma_0)$*, for some* σ_0.

Proof: Proof is by induction on the size of the evaluation context $\mathbb{E}$ [3]. 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 σ', and thus the claim holds by IH.

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

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

$$\frac{\begin{array}{c}(c_1, t, m\ (\overline{t\ var})\ \{\, e_{n+1} \,\}, \sigma'''_0) = findMeth(u, m) \\ \Pi \vdash e : (u, \sigma_0) \qquad (\forall\, i \in \{1..n\}\ ::\ \Pi \vdash e_i : (t'_i, \sigma''_i)\ \wedge\ t'_i <: t_i)\end{array}}{\Pi \vdash \mathbb{E}_2[e] : (t, \{\bot\})}$$

[3] Formulation of the proof is similar to Flatt's work [18]

Here $\mathit{findMeth}(u',m) = (c_2, t, m\ (\overline{t\ var})\ \{e'_{n+1}\}, \sigma'''_1)$, by the definitions of *override* and *findMeth*, where $c_2 <: c_1$, so (T-CALL) gives $\Pi \vdash \mathbb{E}_2[e'] : (t, \sigma')$; or
<2> (T-CALL-OPEN):

$$\frac{\begin{array}{c} e = \mathbf{this}.f \qquad \mathit{typeOfF}(f) = (d, \texttt{@open t'}) \\ (c_1, t, m\ (t_1\ var_1, \ldots, t_n\ var_n)\ \{e_{n+1}\}, \sigma'''_0) = \mathit{findMeth}(t', m) \\ \Pi \vdash e : (t', \sigma') \qquad (\forall\, i \in \{1..n\}\ ::\ \Pi \vdash e_i : (t'_i, \sigma''_i)\ \wedge\ t'_i <: t_i) \end{array}}{\Pi \vdash \mathbb{E}_2[e] : (t, \{\texttt{open f m}\ \emptyset\} \cup \sigma' \cup \bigcup_{i=1}^{n} \sigma''_i)}$$

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

$$\frac{\begin{array}{c} e = \mathit{loc}.f \qquad \mathit{typeOfF}(f) = (d, \texttt{@open t'}) \\ (c_1, t, m\ (t_1\ var_1, \ldots, t_n\ var_n)\ \{e_{n+1}\}, \sigma'''_0) = \mathit{findMeth}(t', m) \\ \Pi \vdash e : (t', \sigma') \qquad (\forall\, i \in \{1..n\}\ ::\ \Pi \vdash e_i : (t'_i, \sigma''_i)\ \wedge\ t'_i <: t_i) \end{array}}{\Pi \vdash \mathbb{E}_2[e] : (t, \{\texttt{open f m}\ \emptyset\} \cup \sigma' \cup \bigcup_{i=1}^{n} \sigma''_i)}$$

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

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

$$\frac{\begin{array}{c} \Pi \vdash v : (u, \emptyset) \qquad (c_1, t, m\ (t_1\ var_1, \ldots, t_n\ var_n)\ \{e_{n+1}\}, \sigma'''_0) = \mathit{findMeth}(u, m) \\ (\forall\, i \in \{1..(p-1)\}\ ::\ \Pi \vdash v_i : (t'_i, \emptyset)\ \wedge\ t'_i <: t_i) \\ (\forall\, j \in \{(p+1)..n\}\ ::\ \Pi \vdash e_j : (t'_j, \sigma''_j)\ \wedge\ t'_j <: t_j) \qquad \Pi \vdash e_p : (t', \sigma')\ \wedge\ t' <: t_p) \end{array}}{\Pi \vdash \mathbb{E}_2[e] : (t, \{\bot\})}$$

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

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

$$\frac{\Pi \vdash e : (c, \sigma') \qquad \mathit{typeOfF}(f) = (d, t) \qquad c <: d \qquad \Pi \vdash e_2 : (t_2, \sigma_2) \qquad t_2 <: t}{\Pi \vdash \mathbb{E}_2[e] : (t_2, \sigma' \cup \sigma_2 \cup \{\texttt{write f}\})}$$

By the definition of field lookup, $\mathit{typeOfF}(f)$ does not change. Thus, by (T-SET), $\Pi \vdash \mathbb{E}_2[e'] : (t_2, \sigma_0)$; or
<2> (T-SET-OPEN):

$$\frac{\begin{array}{c} e = \mathbf{this} \qquad \Pi(\mathbf{this}) : t' \\ \mathit{typeOfF}(f) = (d, \texttt{@open t}) \qquad t' <: d \qquad \Pi \vdash e_2 : (t_2, \sigma_2) \qquad t_2 <: t \end{array}}{\Pi \vdash \mathbb{E}_2[e] : (t_2, \{\bot\})}$$

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

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

$$\frac{\Pi \vdash e : (c, \sigma_1) \qquad typeOfF(f) = (d, [\texttt{@open}]\ t) \qquad c <: d}{\Pi \vdash \mathbb{E}_2[e] : (t, \sigma_1 \cup \{\texttt{read f}\})}$$

The result of *typeOfF* does not change. Thus, by (T-GET), $\Pi \vdash \mathbb{E}_2[e'] : (t, \sigma_0)$.∎

Lemma 10. [Replacement with subtyping] *If* $\Sigma \hookrightarrow \Sigma'$, $\Sigma = \langle \mathbb{E}[e], \mu \rangle$, $\Sigma' = \langle \mathbb{E}[e'], \mu' \rangle$, $\Pi \vdash \mathbb{E}[e] : (t, \sigma)$, $\Pi \vdash e : (u, \sigma_0)$, *and* $\Pi \vdash e' : (u', \sigma_1)$ *and* $u' <: u$, *then* $\Pi \vdash \mathbb{E}[e'] : (t', \sigma')$ *where* $t' <: t$.

Proof: Proof is by induction on the size of the evaluation context $\mathbb{E}$ [4]. Size of the $\mathbb{E}$ refers to the number of recursive applications of the syntactic rules necessary to create $\mathbb{E}$. In the base case, $\mathbb{E}$ has size zero, $\mathbb{E} = -$, and $t' = u' <: u = t$. For the induction step we divide the evaluation context into two parts such that $\mathbb{E}[e_1] = \mathbb{E}_1[\mathbb{E}_2[e_2]]$, and $\mathbb{E}_2$ has size one. The induction hypothesis (IH) is that the lemma holds for all evaluation contexts, which is smaller than the one ($\mathbb{E}_1$) considered in the induction step. We prove it case by case on the rule used to generate $\mathbb{E}_2$. In each case we show that $\Pi \vdash \mathbb{E}_2[e] : (s, \sigma)$ implies that $\Pi \vdash \mathbb{E}_2[e'] : (s', \sigma')$, for some $s' <: s$, and the claim holds by IH. The cases for $(loc_0.f = -)$, $(- + e_2)$, $(n + -)$,and $(c\ var = -; e_2)$ follow directly from IH.

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

$$\frac{\begin{array}{c}\Pi \vdash e : (t', \sigma') \qquad (c_1, t, m\ (\overline{t\ var})\ \{e_{n+1}\}, \sigma_1) = findMeth(t', m) \\ (\forall\ i \in \{1..n\}\ ::\ \Pi \vdash e_i : (t_i', \sigma_i'')\ \wedge\ t_i' <: t_i)\end{array}}{\Pi \vdash \mathbb{E}_2[e] : (t, \{\bot\})}$$

We have $findMeth(t', m) = (c_2, t, m\ (\overline{t\ var})\ \{e'_{n+1}\}, \sigma_1)$, by the definitions of *override* and *findMeth*, $c_2 <: c_1$, so (T-CALL) gives $\Pi \vdash \mathbb{E}_2[e'] : (t, \{\bot\})$; or
<2> (T-CALL-OPEN):

$$\frac{\begin{array}{c}e = \texttt{this}.f \\ typeOfF(f) = (d, \texttt{@open u}) \qquad (c_1, t, m\ (\overline{t\ var})\ \{e_{n+1}\}, \sigma_1) = findMeth(u, m) \\ \Pi \vdash e : (u, \sigma_2) \qquad (\forall\ i \in \{1..n\}\ ::\ \Pi \vdash e_i : (t_i', \sigma_i'')\ \wedge\ t_i' <: t_i)\end{array}}{\Pi \vdash \mathbb{E}_2[e] : (t, \{\texttt{open f m}\emptyset\} \cup \sigma_2 \cup \bigcup_{i=1}^{n} \sigma_i'')}$$

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

[4] Formulation of the proof is similar to Clifton's work [13] and Flatt's work [18]

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

$$\frac{\begin{array}{c} e = loc.f \\ typeOfF(f) = (d, \texttt{@open t'}) \qquad (c_1, t, m\ (\overline{t\ var})\ \{e_{n+1}\}, \sigma_0') = findMeth(t', m) \\ \Pi \vdash e : (u, \sigma_0) \qquad (\forall\ i \in \{1..n\}\ ::\ \Pi \vdash e_i : (t_i', \sigma_i'')\ \wedge\ t_i' <: t_i) \end{array}}{\Pi \vdash \mathbb{E}_2[e] : (t, \{\texttt{open f m}\ \emptyset\} \cup \sigma_0 \cup \bigcup_{i=1}^{n} \sigma_i'')}$$

It must be the case that $e' = loc'$. From the statement of the lemma, we have $\Pi \vdash loc' : (u', \sigma_0')$, where $u' <: u$. By the definitions of *override* and *findMeth*, $findMeth(u', m) = (c_2, t, m\ (\overline{t\ var})\ \{e_{n+1}'\}, \sigma_1')$. Therefore, by (T-CALL), the type of the expression is t.

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

$$\frac{\begin{array}{c} \Pi \vdash v_0 : (u_0, \emptyset) \qquad (c, t, m\ (\overline{t\ var})\ \{e_{n+1}\}, \sigma''') = findMeth(u_0, m) \\ (\forall\ i \in \{1..(p-1)\}\ ::\ \Pi \vdash v_i : (u_i', \emptyset) \qquad (\forall\ i \in \{(p+1)..n\}\ ::\ \Pi \vdash e_i : (u_i', \sigma_i'') \\ \Pi \vdash e : (u, \sigma_0) \qquad (\forall\ i \in \{1..n\} \setminus \{p\}\ ::\ u_i' <: t_i) \qquad u <: t_p \end{array}}{\Pi \vdash \mathbb{E}_2[e] : (t, \{\bot\})}$$

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

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

$$\frac{\Pi \vdash e : (u, \sigma_0) \qquad typeOfF(f) = (d, t_0) \qquad u <: d \qquad \Pi \vdash e_2 : (t, \sigma_2) \qquad t <: t_0}{\Pi \vdash \mathbb{E}_2[e] : (t, \sigma_0 \cup \sigma_2 \cup \{\texttt{write f}\})}$$

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

$$\frac{\begin{array}{c} e = \textbf{this} \qquad \Pi(\textbf{this}) : u \\ typeOfF(f) = (d, \texttt{@open t'}) \qquad u <: d \qquad \Pi \vdash e_2 : (t, \sigma_2) \qquad t <: t' \end{array}}{\Pi \vdash \mathbb{E}_2[e] : (t, \{\bot\})}$$

The only possibility is that $e' = loc$, 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 $-.f$**.** The last step for $\mathbb{E}_2[e]$ should be (T-GET):

$$\frac{\Pi \vdash e : (c, \sigma_1) \qquad typeOfF(f) = (d, [\texttt{@open}]\ t) \qquad c <: d}{\Pi \vdash \mathbb{E}_2[e] : (t, \sigma_1 \cup \{\texttt{read f}\})}$$

The result of *typeOfF* does not change. Thus, by (T-GET), $\Pi \vdash \mathbb{E}_2[e'] : (t', \sigma_0)$∎

Theorem 3. [Type preservation] *If* $\Pi \vdash \Sigma$*, where* $\Sigma = \langle e, \mu \rangle$*,* $\Sigma \rightarrowtail \langle e', \mu' \rangle$*, and* $\Pi \vdash e : (t, \sigma)$*, then there is some* Π'*,* t' *and* σ' *such that*

(a) $(\mu' \approx \Pi')$*, i.e.* $\Pi' \vdash \Sigma'$*;*

(b) $\Pi \lessdot \Pi'$*; and*
(c) $\Pi' \vdash e' : (t', \sigma') \wedge (t' <: t)$.

Proof: The proof is by cases on the reduction step applied. We prove the first seven cases where the reduction takes only one task local step. Then, we prove the case for yielding controls to other tasks.

New Object. Here $e = \mathbb{E}[new\ c()]$ and $e' = \mathbb{E}[loc]$, where $loc \notin dom(\mu)$, and $\mu' = \{loc \mapsto [c.\{f \mapsto \texttt{null} \mid f \in fields(c)\}.\{(m \mapsto \sigma) \in methE(c)\}]\} \oplus \mu$. Let $\Pi' = \Pi, loc : c$, then $\Pi \lessdot \Pi'$. We now show that $\Pi' \approx \mu'$. Because $loc \notin dom(\mu)$, $(\Pi \approx \mu) \Rightarrow (loc \notin dom(\Pi))$ by Definition 12. Thus part 1 of the definition for $\Pi' \approx \mu'$ holds for all $loc' \neq loc$. Now $\mu'(loc) = [c.F.E]$, $\Pi'(loc) = c$, $dom(F) = dom(fields(c))$, $rng(F) = \{\texttt{null}\} \subseteq dom(\mu) \cup \{\texttt{null}\}$, and 1(d) holds vacuously. So part 1 of $\Pi' \approx \mu'$ holds. Part 2 holds because $\Pi \approx \mu$, $loc \in dom(\Pi')$, $loc \in dom(\mu')$.
By Lemma 8 (Environment extension) and $loc \notin dom(\Pi)$, we have $\Pi' \vdash \mathbb{E}[\texttt{new}\ c()] : (t, \sigma)$. Now $\Pi' \vdash \texttt{new}\ c() : (c, \emptyset)$ and $\Pi' \vdash loc : (c, \emptyset)$, so by Lemma 9, $\Pi' \vdash \mathbb{E}[loc] : (t, \sigma')$.

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

We now show that $\Pi \vdash \mathbb{E}[v] : (t', \sigma')$ for some $t' <: t$ and some σ'. We have $\Pi \vdash loc.f : (s, \{\texttt{read f}\})$. The last step in this derivation must be (T-GET). By the first hypothesis of (T-GET) and by (T-LOC), and by $\Pi \approx \mu$, we have $\Pi(loc) = u$. By the second hypothesis of (T-GET), $typeOfF(f) = (c, s)$. By the second hypothesis of (T-GET-OPEN-LOC), $typeOfF(f) = (c, \texttt{@open s})$. Also by $\Pi \approx \mu$, if (a) $\mu(v) = [u'.F'.E']$, then $\Pi(v) = u'$ and $u' <: s$; otherwise (b) $\mu(v) = \texttt{null}$. In both cases, the type of v is subtype of s, by Lemma 10 (Replacement with subtyping), $\Pi \vdash E[v] : (t', \sigma')$.

Field Set. Here $e = \mathbb{E}[loc.f = v]$, $e' = \mathbb{E}[v]$, $\mu_0 = \mu \oplus (loc \mapsto [u.F \oplus (f \mapsto v)])$, $\mu' = update(\mu_0, loc, f, v)$, and $\mu(loc) = [u.F.E]$. Let $\Pi' = \Pi$, thus $\Pi \lessdot \Pi'$. We now show that $\Pi \approx \mu'$. Observe that the *update* function changes the effect mapping E in each of the object record, but not the fields F, which have no impact on $\Pi \approx \mu'$, by Definition 12. Here $\mu'(loc) = [c.F \oplus (f \mapsto v).E']$, for some E'. For part 1(a) $\Pi(loc) = u$, since $\mu(loc) = [u.F.E]$ and $\Pi \approx \mu$. For part 1(b) $dom(F \oplus (f \mapsto v) = dom(fields(u))$, since $loc.f = v$ is well-typed. For part 1(c), $rng(F \oplus (f \mapsto v)) \subseteq rng(F) \cup \{v\}$. Now since $loc.f = v$ is well-typed, then $v \in dom(\Pi)$ or $v = \texttt{null}$. In the former case, by $\Pi \approx \mu$, then $v \in dom(\mu)$. $v \in dom(\mu)$ implies $v \in dom(\mu')$. In either case $rng(F) \cup \{v\} \subseteq dom(\mu') \cup \{\texttt{null}\}$. Part 1(d) holds for all $f \in dom(F)$, $f' = f$. Part 1(d) holds vacuously for f if $v = \texttt{null}$. Otherwise, $(F \oplus (f \mapsto v))(f) = v$, and by (T-SET) or (T-SET-OPEN-LOC) and (T-LOC), $\Pi(v) <: s'$, where $fields(u) = (c, s')$ and $u <: c$. Parts 2 holds since $dom(\mu') = dom(\mu)$.

To see $\Pi \vdash e' : (t, \sigma)$, let $\Pi \vdash loc.f = v : (s, \sigma_0)$. By (T-SET) or (T-SET-OPEN-LOC), $\Pi \vdash v : (s, \emptyset)$ and Lemma 9 (Replacement), $\Pi \vdash \mathbb{E}[v] : (t, \sigma_1)$.

Method Call. Here $e = \mathbb{E}[loc.m(\overline{v})]$, $e' = \mathbb{E}[e_1]$, $\mu(loc) = [u\ .\ F\ .\ E]$, $(findMeth(u, m) = (u', t, m(\overline{t\ var})\{e_2\}, \sigma_0)$, $\mu' = \mu$ and $e_1 = [loc/\texttt{this}, \overline{v/var}]e_2)$. Let $\Pi' = \Pi$. Clearly $\Pi' \approx \mu'$, and $\Pi \lessdot \Pi'$.

We now show that $\Pi \vdash e' : (t', \sigma')$ for some $t' <: t$ and some σ'. $\Pi \vdash e : (t, \sigma)$ implies that $loc.m(\overline{v})$ and all its subterms are well-typed in Π. By part 1(a) of $\Pi \approx \mu$, $\Pi \vdash loc : (u, \emptyset)$. By the definition of *findMeth*, $u <: u'$. Let $\Pi \vdash v_i : (u_i, \emptyset)\ \forall i \in \{1..n\}$ and let $\Pi \vdash loc.m(\overline{v}) : (t_m, \sigma_m)$. This last judgment must be (T-CALL), with $(u', t_m, m(\overline{t\ var})\{e_2\}, \sigma_m) = findMeth(u, m)$, where $\forall i \in \{1..n\} :: u_i <: t_i$. By the definition of the function *findMeth*,

rules (T-METHOD) and *override*, $(\overline{var:t}, \mathtt{this}:u') \vdash e_2 : (u'_m, \sigma_1)$, and $u'_m <: t_m$. By Lemma 8 (Environment extension) (and appropriate alpha conversion of free variables in e_2), $\Pi, \overline{var:t}, \mathtt{this}:u' \vdash e_2 : (u'_m, \sigma_1)$. By Lemma 7 (Substitution), $\Pi \vdash [loc/\mathtt{this}, \overline{v/var}]e_2 : (u'', \sigma_1)$, for some $u'' <: u'_m <: t_m$. Finally, Lemma 10 (Replacement with subtyping) gives $\Pi \vdash e' : (t', \sigma')$ for some $t' <: t$.

Local Declaration. In this case $e = \mathbb{E}[t\ var = v; e_1]$, $e' = \mathbb{E}[e'_1]$, where $e'_1 = [v/var]e_1$ and $\mu' = \mu$. Let $\Pi' = \Pi$. Obviously $\Pi' \approx \mu'$, and $\Pi \lessdot \Pi'$. We show $\Pi \vdash \mathbb{E}[e'_1] : (t', \sigma')$, for some $t' <: t$. $\Pi \vdash e : (t, \sigma)$ implies that $t\ var = v; e_1$ and all its subterms are well typed in Π, let $\Pi \vdash t\ var = v; e_1 : (s, \sigma_0)$. By (T-DEFINE), $\Pi, var : t \vdash e_1 : (s, \sigma_0)$. By Lemma 7 (Substitution), $\Pi \vdash [v/var]e_1 : (s', \sigma_1)$, for some $s' <: s$. Finally, Lemma 10 (Replacement with subtyping) gives $\Pi \vdash e' : (t', \sigma')$ for some $t' <: t$.

Addition. Here $e = \mathbb{E}[n_1 + n_2]$ and $e' = \mathbb{E}[n]$, where $n = n_1 + n_2$, and $\mu' = \mu$. Let $\Pi' = \Pi$. Clearly $\Pi' \approx \mu'$, and $\Pi \lessdot \Pi'$.
We now show that $\Pi \vdash \mathbb{E}[n] : (t, \sigma')$ for some σ'. We have $\Pi' \vdash n_1 + n_2 : (int, \emptyset)$ and $\Pi' \vdash n : (int, \emptyset)$, so by Lemma 9, $\Pi' \vdash \mathbb{E}[loc] : (t, \sigma')$.∎