Open Effects: Programmer-guided Effects for Open World Concurrent Programs

Yuheng Long and Mehdi Bagherzadeh and Hridesh Rajan

TR #13-04
Initial Submission: October 15, 2013

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

Open Effects: Programmer-guided Effects for Open World Concurrent Programs*

Yuheng Long and Mehdi Bagherzadeh and Hridesh Rajan

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

Abstract. The open world assumption makes the design of a type-and-effect system challenging, especially in concurrent object-oriented languages. The main problem is in the computation of the effects of a dynamically dispatched method invocation, because all possible dynamic types of its receiver are not known statically. Previous work proposes effect annotations that provide a static upper bound on the effects of a dynamically dispatched method, *conservative* enough to cover the effects of all methods which could possibly be executed upon its invocation. For two dynamically dispatched methods, a typical type-and-effect system may disallow concurrent execution of their invocations because their conservatively specified static effects conflict. However, such a conflict may not actually happen at runtime, depending on the dynamic types of their receivers. This work proposes *open effects*, a sound trust-but-verify type-and-effect system, to better enable concurrent execution of dynamically dispatched method invocations. If a programmer annotates the receiver of a certain method invocation as open, then the type system *trusts* the programmer and assigns an open effect to the method. The open effect is supposed, optimistically, not to conflict with other effects. Such optimistic assumptions are *verified* statically, if possible, or at runtime otherwise. Open effects is complementary to previously proposed static and dynamic effect analyses and combines them such that the accuracy of static analysis could help decrease the overhead of the dynamic analysis. Performance evaluations of an implementation of open effects, on various benchmarks, show that: open effects incurs negligible annotation and runtime overheads such that code with open effects does almost *as well as its manually tuned* concurrent version.

1 Introduction

A type-and-effect system [22, 33] helps programmers in analyzing locking disciplines [4], checked exceptions [6, 30], detecting race conditions [12], etc, by adding an encoding of computational effects into semantic objects of a language and a discipline for controlling these effects into its type system [45]. These effects describe how the state of a program will be modified by expressions in the language, e.g. a field expression may have a read or write effect to represent reading from or writing into memory [45].

The open world assumption [36], which says class hierarchies are *extensible*, makes the design of a type-and-effect system challenging, especially in concurrent object-oriented languages that support dynamic dispatch. The main problem is in the computation of the effects of a dynamically dispatched method invocation, because all possible

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

Library

```
1  class Pair {
2   int fst, snd;
3   @open Op f;
4   Pair init() { fst = 1; snd = 2; this }
5   Op setOp(Op f) { this.f = f }
6   int apply() {
7    fork{
8     fst = f.op(fst),
9     snd = f.op(snd)
10   }
11  }
12 }

14 class Op {
15  int res;
16  Op init() { res = 0; this }
17  int op(int o) {
18   0
19  }
20 }
```

Prog1

```
21 class Prefix extends Op {
22  int op(int o) {// Effects: writes res
23   res += o
24  }
25 }
26 Pair pr = new Pair().init();
27 Op pf = new Prefix().init();
28 pr.setOp(pf);
29 pr.apply()
```

Prog2

```
31 class Hash extends Op {
32  int op(int o) {// Effects: None
33   int key = o;
34   // Hash computation
35   key = ..
36  }
37 }
38 Pair pr = new Pair().init();
39 Op ha = new Hash().init();
40 pr.setOp(ha);
41 pr.apply()
```

Fig. 1. Library class `Pair` with open field `f` and clients *Prog1* and *Prog2* extending `Op`.

dynamic types of its receiver may not be known statically. To address this problem, previous work [10, 24] proposes static effect annotations that provide an upper bound on the effects of a dynamically dispatched method. This upper bound should be conservative enough to cover the effect of all methods which could be possibly executed as the result of invocation of the dynamically dispatched method.

To illustrate, consider computation of the effects of the dynamically dispatched method `op` in Figure 1. The code contains the library classes `Pair` and `Op` which represent a pair of integers and operations carried out on pair elements, respectively. It also contains client programs *Prog1* and *Prog2* that extend the library class `Op` and override its method `op`, in the `Prefix` and `Hash` classes. `Prefix` computes a prefix sum in its effectful overriding of `op` with the effect of writing into the field `res`, shown as $\mathbf{wr}(\mathit{res})$, whereas `Hash` computes a hash in its pure method `op` with no memory effects, i.e. $\emptyset$. To specify the effects of the method `op`, in a typical type-and-effect system, its effects should be broad enough to conservatively cover the effects of its overriding methods in all of its subtypes, i.e. `Prefix` and `Hash`. This results in the effect $\mathbf{wr}(\mathit{res})$ for the dynamically dispatched method `op` which is the *union* of the effects of its overriding methods, i.e. union of $\mathbf{wr}(\mathit{res})$ and $\emptyset$, in all of its subtypes.

For two dynamically dispatched methods, a typical type-and-effect system may disallow the concurrent execution of their invocations, because their broadly specified static effects may conflict. However, such a conflict may not actually happen at runtime, depending on the dynamic types of their receivers. To illustrate, consider the two invocations of the method `op`, on lines 7–10 of Figure 1, in the fork expression of the `apply` method. This, somewhat nontraditional, fork expression **`fork`**$\{e_1, e_2\}$ [9] executes the expressions e_1 and e_2 concurrently if their effects do not conflict, and sequentially otherwise. For a memory location, writing into the location conflicts with other reads and writes of the same location. A typical type-and-effect system would *serialize* the execution of these invocations of the method `op`, because their static effects $\mathbf{wr}(res)$ conflict with each other. Such serialization of these method invocations makes sense when their receiver `f` is of dynamic type `Prefix`, however, these invocations could run *concurrently*, for example when they have empty effects at runtime when their receiver `f` has the dynamic type of `Hash`. A typical type-and-effect system fails to expose such safe concurrency opportunities.

This work, in the spirit of hybrid type checking [27], proposes *open effects*, a sound trust-but-verify type-and-effect system, which uses programmer's knowledge to better expose and enable safe concurrent execution in the presence of dynamically dispatched method invocations. If a programmer annotates the receiver of certain method invocations as @*open*, the type system trusts the programmer and assigns an open effect to the method invocation, which supposedly does not conflict with other effects. Such optimistic assumptions are verified statically, if enough static information is available, or at runtime otherwise. Our effect system has two kinds of effects: open and concrete effects. An open effect represents the effects of a dynamically dispatched method invocation where the dynamic type of its receiver is not known statically, but the receiver is qualified with an open annotation @*open*, i.e. the receiver is an *open reference*. A concrete effect represents standard memory effects, which are reads and writes of memory [45]. An open effect is *concretized* at runtime when the dynamic type of its receiver is known. Open effects's type-and-effect system is complementary to previously proposed static and dynamic effect analyses [18, 37, 38], and combines them in such a way that the accuracy of the static analysis could decrease the overhead of the dynamic analysis. That is, in open effects, similar to typical type-and-effect systems, we statically compute the effects of each expression, however, unlike these systems which use conservative effect specifications for a dynamically dispatched method with the unknown dynamic type of its receiver, we use placeholder open effects to produce effect equations with unknown terms.

Our type-and-effect system has two parts:

- ***Static part***, that *(i)* computes the effects of the methods, one method at a time and independent of the dynamic types of the receivers of dynamically dispatched method invocations; and *(ii)* possibly verifies the optimistic assumption that open effects do not conflict with other effects, i.e. disjointness assumption.
- ***Dynamic part***, that *(iii)* concretizes the statically computed open effects and updates them by tracking open references and their values, and *(iv)* verifies, using runtime checks, the disjointness assumptions that could not be verified statically.

Basically, in open effects the responsibility of deciding the disjointness of effects is divided between the static and dynamic parts.

To illustrate, imagine that a programmer marked the field `f` of class `Pair` as open using @*open* annotation, on line 3 of Figure 1. Doing so, the programmer is hinting the type-and-effect system that there may be parallelism opportunities when invoking dynamically dispatched methods on the receiver `f`. The static part of our system trusts the programmer and assigns the open effects $\mathbf{open}(f\ op\ \gamma_1)$ and $\mathbf{open}(f\ op\ \gamma_2)$ to the method invocations `f.op(fst)`, on line 8, and `f.op(snd)`, on line 9, respectively. The open effect $\mathbf{open}(f\ op\ \gamma_1)$ is the effect of the invocation of the dynamically dispatched method `op` on the open receiver `f` with the statically unknown effect of γ_1. The open effect $\mathbf{open}(f\ op\ \gamma_2)$ is similar. These open effects are assumed to not conflict with each other and other effects.

The static part continues by computing the effect of the expression `fst = f.op(fst)` to be writing and reading fields `fst` and `f` plus the effect of the invocation of the method `op` on the open receiver `f`, i.e. $\sigma_1 = \{\mathbf{wr}(fst), \mathbf{rd}(f),\ \mathbf{open}(f\ op\ \gamma_1)\}$[1]. Similarly, the effects of `snd = f.op(snd)` is $\sigma_2 = \{\mathbf{wr}(snd), \mathbf{rd}(f),\ \mathbf{open}(f\ op\ \gamma_2)\}$. These two expressions, of the fork expression on lines 7–10 could be executed concurrently if their effects σ_1 and σ_2 do not conflict, which in turn boils down to the verification of their open effects not conflicting, since $\mathbf{wr}(fst)$ and $\mathbf{wr}(snd)$ do not conflict. This could be verified statically if there is enough static information about the unknown effects of the method `op` or otherwise dynamically. Different modular static analyses could be integrated into the open effects' type-and-effect system to boost its static analysis. In this work, we illustrate the integration of a modular alias analysis [18]. Our implementation integrates other static analyses such as purity analysis [37] and array effect analysis [38].

Open effects better enables exposure of safe concurrency opportunities, in the presence of dynamically dispatched method calls, as follows: for two subexpressions of a fork expression with their statically computed effects, which may contain open effects, there are three answers for the question of *do their effects conflict statically?*: yes (conflict), no (disjoint), and unknown (may or may not conflict). Using open effects and depending on the disjointness of the effects of its subexpressions, a fork expression is *soundly and statically* translated to:

(1) *Yes (conflict):* an unconditional sequential execution of its subexpressions.
(2) *No (disjoint):* an unconditional parallel execution of its subexpressions.
(3) *Unknown (may or may not conflict):* a conditional in which the unknown open effects of subexpressions are concretized and tested for conflicts. If the concretized effects conflict then run sequentially, i.e. *(1)*, else in parallel, i.e. *(2)*.

In Figure 1, there is not enough static information to decide if σ_1 and σ_2, especially their open effects, conflict and thus the case *(3)* above applies and the fork expression, lines 7–10, translates to a conditional. This brings into the picture the dynamic part of our type-and-effect system that decides the disjointness of effects that could not be decided statically.

The dynamic part concretizes, or fills in, the unknown effects of open effects when the open references are known at runtime. For example, upon the execution of the expression `pr.setOp(pf)`, on line 28 of *Prog1*, the open reference `f`, of static type `Op`, in the open effects of $\mathbf{open}(f\ op\ \gamma_1)$ and $\mathbf{open}(f\ op\ \gamma_2)$ is set to object `pf`, of the dynamic type of `Prefix`. This causes these two open effects to be concretized to $\mathbf{wr}(res)$, because the method `op` of type `Prefix` has the effect of $\mathbf{wr}(res)$. With such concretization, the effects σ_1 and σ_2 conflict at runtime and thus the fork expression, and the invocations of the dynamically dispatched method `op`, lines 7–10, is sequentialized. Unlike *Prog1*, in *Prog2*, assigning the object `ha` of dynamic type `Hash` to the open reference `f`,

[1] Write effect of a field, e.g. $\mathbf{wr}(fst)$ covers its read effect, e.g. $\mathbf{rd}(fst)$.

via `pr.setOp(ha)` on line 40, causes the open effects **open**$(f\ op\ \gamma_1)$ and **open**$(f\ op\ \gamma_2)$ to be concretized to empty set $\emptyset$, because the method `op` of type `Hash` is pure. This in turn allows the fork expression to be translated to concurrent execution of the invocations of method `op`, as their effects σ_1 and σ_2 do not conflict. *Depending on the dynamic type of the open field* `f`*, the fork expression could run sequentially or in parallel.*

1.1 Contributions

In summary, the main contributions of this work are the following:

- Open effects: a trust-but-verify hybrid type-and-effect system to expose safe concurrency in open world concurrent programs with dynamic dispatch;
- Static semantics of open effects, in §2, and its dynamic semantics, in §3;
- Proof of soundness for open effects, in §3;
- *OpenEffectJ*, an OpenJDK prototype implementation of open effects; and
- Speedup and overhead evaluations of *OpenEffectJ*, in §4.

Finally, §5 compares open effects with other previous work on reasoning about effects of programs in three categories of static, dynamic and hybrid techniques; and §6 concludes the paper after discussing some avenues for future work.

2 Open Effects: A Hybrid Type-and-Effect System

$$
\begin{array}{ll}
\mathit{prog} ::= \overline{\mathit{decl}}\ e & \\
\mathit{decl} ::= \texttt{class}\ c\ \texttt{extends}\ d\ \{\ \overline{\mathit{field}}\ \ \overline{\mathit{meth}}\ \} & \\
\mathit{field} ::= [\texttt{@open}]\ t\ f\,\texttt{;} & \\
\mathit{meth} ::= t\ m\ (\ \overline{\mathit{arg}}\ )\ \{\ e\ \} & \\
t ::= c \mid \texttt{int} \mid \texttt{bool} & \\
\mathit{arg} ::= t\ \mathit{var},\ \ \textbf{where}\ \ \mathit{var} \neq \texttt{this} & \\
e ::= x \mid \texttt{null} \mid \mathit{arg} = e\,\texttt{;}\,e & \text{“Var, Null, Definition”} \\
\quad\ \ \mid x\,\texttt{.}\,m\,(\bar{x})\ \mid \texttt{new}\ c\,(\,) & \text{“Call, New”} \\
\quad\ \ \mid x \circ x \mid n \mid \mathit{loc} & \text{“Binary, Number, Location”} \\
\quad\ \ \mid \texttt{if}\ \mathrm{x}\ \texttt{then}\ \mathrm{e}\ \texttt{else}\ \mathrm{e} & \text{“Conditional”} \\
\quad\ \ \mid \texttt{this}\,.f \mid \texttt{this}\,.f = x & \text{“Get, Set Field”} \\
\quad\ \ \mid e\ \#\ e & \text{“Disjointness Check”}
\end{array}
\quad \textbf{where} \quad
\begin{array}{rl}
c \in \mathscr{C}, & \text{set of class names} \\
d \in \mathscr{C} \cup \{\texttt{Object}\} & \\
f \in \mathscr{F}, & \text{set of field names} \\
m \in \mathscr{M}, & \text{set of method names} \\
n \in \mathscr{N}, & \text{set of natural numbers} \\
x, \mathit{var} \in \mathscr{V} \cup \{\texttt{this}\}, & \text{set of variable names} \\
\circ \in \{+,-,*,/\}, & \text{set of binary operations} \\
\mathit{loc} \in \mathscr{L}, & \text{set of locations}
\end{array}
$$

Fig. 2. Syntax for *OpenEffectJ*.

To encode open effects as a type-and-effect system, we use *OpenEffectJ*, a core expression language, shown in Figure 2, which is based on Classic Java [21]. The A-normal form syntax of *OpenEffectJ* is standard except for open annotations *@open* and the disjoint check expression $e_1\#e_2$[2]. This expression checks if the effects of the expressions e_1 and e_2 are disjoint and evaluates to true if they are, and false otherwise[3].

[2] Expression $e_1\#e_2$ decides the disjointness of effects of e_1 and e_2 without evaluating them.

[3] False is sugar for 0 and true is any non-zero integer.

Figure 2 only shows fields annotated with *@open*, i.e. open fields, however, we also support open local variables and open parameters [32]. Using the programmer's knowledge in annotating only certain receiver fields as open is to have as less overhead as possible compared to other alternatives. One alternative is to annotate types as open, however, it causes every reference of that type and its subtypes to be treated as open references which in turn could cause considerable concretization and verification overhead, especially when all references of a type have to pay the price for one reference being open. The same applies to another alternative in which every field of every object is considered open. For simplicity, we assume unique field names, up to alpha renaming, and no method overloading. In Figure 2 the notations $\overline{term}$ and $[term]$ denote a finite possibly empty sequence and an optional *term*, respectively.

2.1 Static Semantics

Our type-and-effect system has static and dynamic parts. The static part encoded in the typing rules, *(i)* computes the effects of the methods, one method at a time and independent of dynamic dispatch; and *(ii)* verifies optimistic disjointness assumptions of open effects, if enough static information is available.

Type-and-Effect Attributes Figure 3 shows *OpenEffectJ*'s type-and-effect attributes. The type of a program and its declarations are given as OK, whereas $(\bar{t} \rightarrow t, \sigma)$ **in** c, specifies the type of a method defined in class c with parameter types $\bar{t}$, return type t and a latent effect σ [45], which is the effects of the body of the method [46]. Finally, the attribute (t, σ) specifies an expression of type t with the effects σ.

$\theta ::= \text{OK}$	"program/decl types"	$\sigma,\gamma ::= \emptyset \mid \top \mid \sigma \cup \sigma$	"effects"
$\mid (\bar{t} \rightarrow t, \sigma)$ **in** c	"method types"	$\mid \mathbf{rd}(f)$	"read effect"
$\mid (t, \sigma)$	"expression types"	$\mid \mathbf{wr}(f)$	"write effect"
$\Pi ::= \{var_i \mapsto t_i\}_{i \in \mathbb{N}}$	"type environments"	$\mid \mathbf{open}(f\ m\ \gamma)$	"open effect"
$A ::= \{var_i \mapsto e_i\}_{i \in \mathbb{N}}$	"aliasing environments"		

Fig. 3. Type-and-effect attributes for *OpenEffectJ*, based on [24, 45].

Figure 3 allows two kinds of effects: concrete and open effects. Concrete effects are standard read and write memory effects[4] $\mathbf{rd}(f)$ and $\mathbf{wr}(f)$,[5] which read and write a field `f`, along with the empty effect $\emptyset$ and the top effect $\top$. The top effect allows read and write effects of any field [24]. An open effect $\mathbf{open}(f\ m\ \gamma)$ represents the effects of a dynamically dispatched method `m` invoked on an open receiver `f`. The placeholder γ represents the unknown effect of the body of the method `m`. We slightly misuse the set notation for presentation purposes.

[4] Previous work [24], uses regions as an abstraction to avoid exposure of implementation details in specifications. In our type-and-effect system there is no explicit specification, and thus exposure of the implementation details is not a concern.

[5] For simplicity, our formalism is not object sensitive, but our compiler implementation is field and object sensitive both [19].

Type checking rules are stated using an implicit fixed class table CT which contains a list of program declarations [21]. Each method in the class table CT has its statically computed effects as part of its signature. The typing rules use a type environment Π, which maps a variable name *var* to its type t. The typing judgement $\Pi \vdash e \rightsquigarrow e' : (t, \sigma)$ says that the expression e is translated to the expression e' and has the type t and the effects σ. The semantic preserving translation does not change the type or the effects of expressions and in spirit is similar to elaboration in languages such as ML [34]. Subtyping is denoted using the relation $<:$ which is the standard reflexive-transitive closure of the declared subclass relationships [21].

Type-and-Effect Rules This section presents select typing rules which form the novel basis of our effect computation using open effects in the presence of dynamically dispatched methods on open fields. Other omitted rules can be found in our report [32].

(T-DISJOINT)

$$\frac{\begin{array}{c}\Pi \vdash e_1 \rightsquigarrow e_1' : (t_1, \sigma_1) \qquad \Pi \vdash e_2 \rightsquigarrow e_2' : (t_2, \sigma_2) \\ \downarrow_O(\sigma_I) = \downarrow_O(\sigma_2) = \emptyset \qquad \sigma_c^1 = \downarrow_C(\sigma_I) \qquad \sigma_c^2 = \downarrow_C(\sigma_2) \qquad \sigma_c^1 \mathbin{\#\!\!\!\!\bigcirc} \sigma_c^2\end{array}}{\Pi \vdash e_1 \# e_2 \rightsquigarrow true : (bool, \emptyset)}$$

(T-CONFLICT)

$$\frac{\begin{array}{c}\Pi \vdash e_1 \rightsquigarrow e_1' : (t_1, \sigma_1) \\ \Pi \vdash e_2 \rightsquigarrow e_2' : (t_2, \sigma_2) \qquad \sigma_c^1 = \downarrow_C(\sigma_I) \qquad \sigma_c^2 = \downarrow_C(\sigma_2) \qquad !(\sigma_c^1 \mathbin{\#\!\!\!\!\bigcirc} \sigma_c^2)\end{array}}{\Pi \vdash e_1 \# e_2 \rightsquigarrow false : (bool, \emptyset)}$$

(T-UNKNOWN)

$$\frac{\begin{array}{c}\Pi \vdash e_1 \rightsquigarrow e_1' : (t_1, \sigma_1) \qquad \Pi \vdash e_2 \rightsquigarrow e_2' : (t_2, \sigma_2) \\ \sigma_c^1 = \downarrow_C(\sigma_I) \qquad \sigma_c^2 = \downarrow_C(\sigma_2) \qquad \sigma_c^1 \mathbin{\#\!\!\!\!\bigcirc} \sigma_c^2 \qquad \downarrow_O(\sigma_I) \neq \emptyset \vee \downarrow_O(\sigma_2) \neq \emptyset\end{array}}{\Pi \vdash e_1 \# e_2 \rightsquigarrow e_1' \# e_2' : (bool, \emptyset)}$$

Auxiliary Functions:

$$\downarrow_C(\sigma) = \{\varepsilon | \varepsilon \in \sigma \wedge \varepsilon \in \{\mathbf{rd}(f), \mathbf{wr}(f), \top, \emptyset\}\} \qquad \downarrow_O(\sigma) = \{\varepsilon | \varepsilon \in \sigma \wedge \varepsilon = \mathbf{open}(f\ m\ \gamma)\}$$

(READ-READ)
$$\mathbf{rd}(f) \mathbin{\#\!\!\!\!\bigcirc} \mathbf{rd}(f')$$

(READ-WRITE)
$$\frac{f \neq f'}{\mathbf{rd}(f) \mathbin{\#\!\!\!\!\bigcirc} \mathbf{wr}(f')}$$

(WRITE-WRITE)
$$\frac{f \neq f'}{\mathbf{wr}(f) \mathbin{\#\!\!\!\!\bigcirc} \mathbf{wr}(f')}$$

(EMPTY)
$$\frac{\forall \varepsilon \in \sigma}{\varepsilon \mathbin{\#\!\!\!\!\bigcirc} \emptyset}$$

(TOP)
$$\frac{\forall \varepsilon \in \sigma}{!(\varepsilon \mathbin{\#\!\!\!\!\bigcirc} \top)}$$

Fig. 4. Deciding disjointness of the effects of expressions e_1 and e_2.

Disjointness Two expressions e_1 and e_2 could safely run in parallel, if their effects do not conflict, i.e. they are disjoint. Figure 4 shows the typing rules for the disjoint expression $e_1 \# e_2$. This expression statically checks if the effects of the expressions e_1 and e_2 conflict and depending on the answer translates into true, false or unknown. Using open effects, the disjointness can be decided statically, provided enough static information is available, as in the rules (T-DISJOINT) and (T-CONFLICT), or otherwise it is deferred to runtime, as in (T-UNKNOWN). The availability of such static information, is dependent on the static analyses integrated into the type system. First, we assume no extra static analysis to focus on the basic ideas behind open effects. Later we discuss adding a modular alias analysis, as an example, which could be helpful in making some of the disjointness decisions in (T-UNKNOWN) statically, as shown later in this section.

In (T-DISJOINT), if there exists no open effects in the effects of the expressions, i.e. $\downarrow_O(\sigma_1) = \downarrow_O(\sigma_2) = \emptyset$, then their effects are disjoint only if their concrete effects are disjoint, i.e. $\sigma_c^1 \circledast \sigma_c^2$. If the concrete effects of e_1 and e_2 are disjoint then $e_1\#e_2$ statically translates to $true$. Since $e_1\#e_2$ does not execute any of the expressions e_1 or e_2, its effect is empty. Similar to (T-DISJOINT), the rule (T-CONFLICT) statically decides the disjointness of the effects of e_1 and e_2 and translates the expression $e_1\#e_1$ to $false$, if their concrete effects conflict, i.e. $!(\sigma_c^1 \circledast \sigma_c^2)$. In this rule there is no need to check the relation between open effects in σ_1 and σ_2 and such a check could be skipped. More importantly, the *concretization of these open effects, at runtime, could be skipped* which in turn results in less runtime checks and better performance. The auxiliary functions $\downarrow_O(\sigma)$ and $\downarrow_C(\sigma)$, return the set of open and concrete effects of the effect set σ, respectively. The function $\circledast$ simply checks for the disjointness of effects, in which a write and read of a field f, i.e. $\mathbf{wr}(f)$ and $\mathbf{rd}(f)$, conflict and other effects are disjoint. The top effect $\top$ conflicts with everything. ε is an effect element in the effect set σ.

Decision about disjointness in $e_1\#e_2$ is deferred to runtime if it cannot be made statically using (T-DISJOINT) and (T-CONFLICT). The rule (T-UNKNOWN) defers such a decision by translating $e_1\#e_2$ to $e_1'\#e_2'$. In (T-UNKNOWN), existence of open effects in either σ_1 or σ_2, i.e. $\downarrow_O(\sigma_1) \neq \emptyset \vee \downarrow_O(\sigma_2) \neq \emptyset$, prevents static decision making about disjointness, as their concretizations may cause conflicts. More static analyses, such as alias analysis, may help (T-UNKNOWN), to make some disjointness decisions statically, as discussed later in this section.

Without open effects, $e_1\#e_2$ would translate to false, if either e_1 or e_2 causes an invocation of a dynamically dispatched method whose receiver is not an open reference and the method does not have user-specified effect specifications. This is because a dynamically dispatched method with a non-open receiver of an unknown dynamic type and no effect specifications, has the top effect that conflicts with any other effect [9].

Fork: An Example Use Case of Disjointness An example use case of the disjoint expression $e_1\#e_2$ is to combine static and runtime decision making about parallel or sequential execution of two expressions e_1 and e_2 in a fork expression, e.g. as in Figure 1, lines 7–10. The fork expression $\mathbf{fork}\{e_1, e_2\}$ executes e_1 and e_2 concurrently if their effects do not conflict, and sequentially otherwise. The rules (T-FORK-SEQUENTIAL) and (T-FORK-PARALLEL) statically translate the fork expression to sequential or parallel executions of e_1 and e_2, respectively. The rule (T-FORK-UNKNOWN) defers such a decision to runtime, because of the lack of the static information.

The rule (T-FORK-SEQUENTIAL) statically translates the fork expression to the sequential composition $e_1'; e_2'$ in which e_1' and e_2' run sequentially. The expressions e_1' and e_2' are translations of e_1 and e_2, respectively. This translation is sound because the effects of expressions e_1 and e_2 do conflict, i.e. $\Pi \vdash e_1\#e_2 \rightsquigarrow false : (bool, \sigma)$. Similarly,

(T-FORK-SEQUENTIAL)

$$\frac{\Pi \vdash e_1 \rightsquigarrow e_1' : (t_1, \sigma_1) \qquad \Pi \vdash e_2 \rightsquigarrow e_2' : (t_2, \sigma_2) \qquad \Pi \vdash e_1 \# e_2 \rightsquigarrow \mathit{false} : (\mathit{bool}, \emptyset)}{\Pi \vdash \mathtt{\textbf{fork}}\{e_1, e_2\} \rightsquigarrow e_1'; e_2' : (t_2, \sigma_1 \cup \sigma_2)}$$

(T-FORK-PARALLEL)

$$\frac{\Pi \vdash e_1 \rightsquigarrow e_1' : (t_1, \sigma_1) \qquad \Pi \vdash e_2 \rightsquigarrow e_2' : (t_2, \sigma_2) \qquad \Pi \vdash e_1 \# e_2 \rightsquigarrow \mathit{true} : (\mathit{bool}, \emptyset)}{\Pi \vdash \mathtt{\textbf{fork}}\{e_1, e_2\} \rightsquigarrow e_1' || e_2' : (t_2, \sigma_1 \cup \sigma_2)}$$

(T-FORK-UNKNOWN)

$$\frac{\begin{array}{c}\Pi \vdash e_1 \rightsquigarrow e_1' : (t_1, \sigma_1) \qquad \Pi \vdash e_2 \rightsquigarrow e_2' : (t_2, \sigma_2) \qquad \Pi \vdash e_1 \# e_2 \rightsquigarrow e_1' \# e_2' : (\mathit{bool}, \sigma) \\ \sigma_c^1 = \downarrow_C (\sigma_1) \qquad \sigma_c^2 = \downarrow_C (\sigma_2) \qquad \sigma_o^1 = \downarrow_O (\sigma_1) \qquad \sigma_o^2 = \downarrow_O (\sigma_2) \\ \mathit{cond} = (\mathit{concretize}(\sigma_o^1) \textcircled{\#} \mathit{concretize}(\sigma_o^2)) \wedge (\mathit{concretize}(\sigma_o^1) \textcircled{\#} \sigma_c^2) \wedge (\mathit{concretize}(\sigma_o^2) \textcircled{\#} \sigma_c^1)\end{array}}{\Pi \vdash \mathtt{\textbf{fork}}\{e_1, e_2\} \rightsquigarrow \mathtt{\textbf{if}}(\mathit{cond})\ \mathtt{\textbf{then}}\ e_1' || e_2'\ \mathtt{\textbf{else}}\ e_1'; e_2' : (t, \sigma)}$$

Fig. 5. Translation of **`fork`**$\{e_1, e_2\}$, to concurrent or sequential execution of e_1 and e_2.

the rule (T-FORK-PARALLEL) translates the fork expression into the parallel composition $e_1' || e_2'$, since the effects of e_1 and e_2 are disjoint, i.e. $\Pi \vdash e_1 \# e_2 \rightsquigarrow \mathit{true} : (\mathit{bool}, \sigma)$[6].

If (T-FORK-SEQUENTIAL) and (T-FORK-PARALLEL) cannot decide about sequential or parallel execution of the expressions in the fork, the decision is deferred to run time using the rule (T-FORK-UNKNOWN). This rule translates a fork expression to an if expression **`if`**(cond) **`then`** $e_1' || e_2'$ **`else`** $e_1'; e_2'$, which in its condition *cond* checks for disjointness of open effects σ_1^o and σ_2^o of the expressions in the fork, and their concrete effects σ_1^c and σ_2^c. The unknown open effects should be concretized, using *concretize*, before being checked for disjointness. Concretization of open effects is discussed in §3. More static analyses, such as alias or purity analysis, may help (T-FORK-UNKNOWN) to make some disjointness decisions statically, as discussed later in this section.

Without open effects, the fork expression will be translated to the sequential composition $e_1; e_2$, if either the expression e_1 or e_2 of the fork expression contains an invocation of a dynamically dispatched method on a non-open receiver, mainly because $e_1 \# e_2$ translates to false.

Alias Analysis: An Example Static Analysis As discussed, our hybrid type-and-effect system divides the responsibility of checking for the disjointness of the effects into two phases: static type checking, and runtime concretization and tracking of open effects. Various modular static analyses could be integrated into the type-and-effects system to increase the precision of static decision making about disjointness of the effects. In this section, we show the integration of a modular definite alias analysis [18] into open effects and illustrate its use. For such integration, we add an aliasing environment A to the typing judgement which maps a variable to its aliases, in Figure 3. The typing judgement $\Pi, A \vdash e \rightsquigarrow e' : (t, \sigma, A')$ takes into account the aliasing environment and its changes. For readability, variables, which do not cause any effect or changes in

[6] Following previous work [9, 35], **`fork`**$\{e_1, e_2\}$ and $e_1 || e_2$ are used to illustrate a use case of the disjoitness expression $e_1 \# e_2$ and are not part of the core syntax, in Figure 2.

the aliasing, use a shorter typing judgement $\mathit{\Pi},A \vdash e : t$. The two rules that are most concerned with aliasing are (T-DEFINE) and (T-SET). Other omitted rules can be found in our accompanying technical report [32].

The rule (T-SET) assigns a variable x to a field f and thus makes them aliases, causing the aliasing relation $x = \texttt{this}.f$ to be added to the aliasing environment A after discarding older aliasing relations for f in A via the kill operation $A \backslash f$. The function *typeOf* returns the type t' of field f declared in class d. In (T-DEFINE), a variable x is assigned the expression e'_1 in the scope of e'_2, causing aliasing which should be taken into account when evaluating e_2. The notation $A;x = e'_1$ stands for extension of the aliasing environment A with the aliasing relation $x = e'_1$.

(T-SET)

$$\frac{\mathit{typeOf}(f) = (d,t') \qquad \mathit{\Pi},A \vdash \texttt{this} : c \qquad \mathit{\Pi},A \vdash x : t \qquad c <: d \qquad t <: t'}{\mathit{\Pi},A \vdash \texttt{this}.f = x \rightsquigarrow \texttt{this}.f = x : (t, \mathbf{wr}(f), A \backslash f \cup \{x = \texttt{this}.f\})}$$

(T-DEFINE)

$$\frac{\mathit{\Pi},A \vdash e_1 \rightsquigarrow e'_1 : (t_1, \sigma_1, A_1) \qquad \mathit{\Pi};x : t, A_1; x = e'_1 \vdash e_2 \rightsquigarrow e'_2 : (t_2, \sigma_2, A_2) \qquad t_1 <: t}{\mathit{\Pi},A \vdash t\ x = e_1; e_2 \rightsquigarrow t\ x = e'_1; e'_2 : (t_2, \sigma_1 \cup \sigma_2, A_2)}$$

Observational Purity. One use case of the aliasing is in detecting observational purity [37] in the rule (T-CALL-PURE) which says in an invocation of $x_0.m(\overline{x})$ if both the receiver x_0 and the parameters $\overline{x}$ are newly created objects, then the effects of the invocation of m is empty. The auxiliary function *findMeth* looks up the class table and returns the declaration of the method in a type or any of its supertypes. A complete list of the auxiliary functions and their definitions can be found in our technical report [32].

(T-CALL-PURE)

$$\frac{\begin{array}{c} A \vdash x_0 = \textbf{new}\ c_0() \qquad \forall x_i \in \overline{x}.\ A \vdash x_i = \textbf{new}\ c_i() \\ \mathit{findMeth}(c_0, m) = (c', t, m\ (\overline{t\ var})\ \{e\}, \sigma) \qquad \forall x_i \in \overline{x}.\ (\mathit{\Pi},A \vdash x_i : t'_i) \wedge (t'_i <: t_i) \end{array}}{\mathit{\Pi},A \vdash x_0.m(\overline{x}) \rightsquigarrow x_0.m(\overline{x}) : (t, \emptyset, A)}$$

Tracking Open References. Another use case of the aliasing is in statically tracking the open references, in the rule (T-CALL-OPEN). This rule assigns an open effect $\mathbf{open}(f\ m\ \gamma)$, to the method invocation $x_0.m(\overline{x})$ in which the receiver x_0 is an alias of the open field `f`, i.e. $A \vdash x_0 = \texttt{this}.f$.

(T-CALL-OPEN)

$$\frac{\begin{array}{c} A \vdash x_0 = \texttt{this}.f \qquad \mathit{typeOf}(f) = (c, @open\, c_0) \\ \mathit{findMeth}(c_0, m) = (c', t, m\ (\overline{t\ var})\ \{e\}, \sigma) \qquad \forall x_i \in \overline{x}.\ (\mathit{\Pi},A \vdash x_i : t'_i) \wedge (t'_i <: t_i) \end{array}}{\mathit{\Pi},A \vdash x_0.m(\overline{x}) \rightsquigarrow x_0.m(\overline{x}) : (t, \mathbf{open}(f\ m\ \gamma), A)}$$

To illustrate use of aliasing in deciding the disjointness of open effects, consider Figure 6 which shows a simplified example adapted from JavaGrande's RayTracer [43]. In this figure, the class `RayTracer` is responsible for rendering a display, and is extended by `RayTracer2D` and `RayTracer3D` to render two and three dimensional displays. Both of these subtypes override the method `run` in their supertype. Using alias and purity analyses, especially the rules (T-SET) and (T-DEFINE), one would conclude that the

```
1 @open RayTracer rt1;
2 rt1 = new RayTracer2D(new Display());
3  fork{
4   { RayTracer rt3D = new RayTracer3D(new Display()); rt3D.run() },
5   { rt1.run() }
6  }
```

Fig. 6. Concurrent execution of fork, because of observational purity [37].

two expression of the fork, lines 4 and 5 can run concurrently, since the expression on line 4 has empty effects because of their observational purity, rule (T-CALL-PURE).

Besides the alias analysis [18], other static analyses could be integrated into the type system to provide more static information. In our prototype implementation of open effects, besides the alias analysis, a few other static analyses, such as purity analysis [37] and array effect analysis [38], are integrated into the type system. These analyses are omitted here to focus on the basic ideas behind the open effects.

2.2 Dynamic Dispatch and Open World Assumption

There are two rules (T-CALL-OPEN) and (T-CALL) in *OpenEffectJ*'s type system for dynamically dispatched method invocations. The differences between these rules highlight the contrast between handling of dynamic dispatch in open effects and static analyses that do not rely on user-specified effect specifications [10, 24]. On one hand, the rule (T-CALL-OPEN), discussed previously, uses an open effect to represent the unknown effects of a dynamically dispatched method invocation on an open receiver, and thus allowing it to run concurrently with other expressions if their effects do not conflict, or sequentially otherwise. On the other hand, (T-CALL) assigns the top effect $\top$ as the effects of the invocation of a dynamically dispatched method on a non-open receiver, especially if there are no effect specifications or constraints, such as containment, for the method [9], and thus sequentializes its execution with any other expression, because the top effect conflicts with all other effects. The rules (T-CALL-OPEN) and (T-CALL) clearly showcase how open effects could be useful in exposing safe concurrency opportunities.

(T-CALL)

$$\frac{\begin{array}{c}\Pi, A \vdash x_0 : c_0 \qquad A \vdash x_0 \neq \mathbf{new}\ c() \\ A \vdash x_0 \neq \mathtt{this}.f \vee (A \vdash x_0 = \mathtt{this}.f \wedge \mathit{typeOf}(f) \neq (d, @\mathit{open}\ c'')) \\ \mathit{findMeth}(c_0, m) = (c', t, m\,(\overline{t\ var})\,\{e\}, \sigma) \qquad \forall x_i \in \bar{x}.\ (\Pi, A \vdash x_i : t_i') \wedge (t_i' <: t_i)\end{array}}{\Pi, A \vdash x_0.m(\bar{x}) \rightsquigarrow x_0.m(\bar{x}) : (t, \top, A)}$$

Method Declaration. A typical type-and-effect system under the open world assumption requires containment between the effects of a class and its subclasses [24], i.e. the effects of an overriding method in a subclass are required to be contained in the effects of the superclass method it overrides. However, in open effects, the overriding and overridden methods are allowed to have effects that are *not restricted* by the containment requirement. This is represented in the rule (T-METHOD) by not requiring any relation between the effects σ of the method m and any other method with effects σ' it may override. The auxiliary function *override* only checks for compatibility of the argument and return types of the overridden and overriding methods and allows their effects to be independent. The function *isType* checks for type validity. Not requiring effect containment in method declarations, improves usage flexibility, especially for libraries and frameworks in which it is common to define empty abstract methods in a class that are overridden by various clients to implement application-specific functionalities.

(T-METHOD)

$$\frac{\begin{array}{c} \mathit{override}(m, c, (\bar{t} \rightarrow t)) \qquad \forall t_i \in \bar{t}.\ \mathit{isType}(t_i) \\ \mathit{isType}(t) \qquad (\overline{var : t}, \mathtt{this} : c), \emptyset \vdash e \rightsquigarrow e' : (t', \sigma, A) \qquad t' <: t \end{array}}{\vdash t\ m(\overline{t\ var})\{e\} \rightsquigarrow t\ m(\overline{t\ var})\{e'\} : (\bar{t} \rightarrow t, \sigma, A)\ \mathbf{in}\ c}$$

Field Set. There are two rules (T-SET) and (T-SET-OPEN) for setting a field. The rule (T-SET), shown previously, sets a non-open field which results in a write effect and updating the aliasing information of the field. The rule (T-SET-OPEN) is similar to (T-SET) except that it generates a top effect $\top$, instead of a write effect. This is because setting an open field `f` results in concretizations of all open effects $\mathbf{open}(f\ m\ \gamma)$ with the open field `f` and any other open effect $\mathbf{open}(g\ m'\ \gamma')$ such that g transitively points to the object containing `f`. Concretization of open effects is discussed in more detail in §3.

(T-SET-OPEN)

$$\frac{\mathit{typeOf}(f) = (d, @\mathit{open}\ t') \qquad \Pi, A \vdash \mathtt{this} : c \qquad \Pi, A \vdash x : t \qquad c <: d \qquad t <: t'}{\Pi, A \vdash \mathtt{this}.f = x \rightsquigarrow \mathtt{this}.f = x : (t, \top, A \backslash f \cup \{x = \mathtt{this}.f\})}$$

3 A Dynamic Semantics with Open Effects

The dynamic part of open effects which is encoded in *OpenEffectJ*'s dynamic semantics, *(i)* concretizes the statically computed open effects using the typing rules, and updates the open effects by tracking their open references and changes in their values; and *(ii)* verifies, using runtime checks, the disjointness assumptions that could not be verified statically.

3.1 Dynamic Semantics Objects

The dynamic semantics of open effects transitions from one configuration to another. A configuration $\Sigma = \langle e, \mu \rangle$, shown in Figure 7, consists of an expression e and a global store μ. The store maps a location *loc* to an object record of the form $o = [c.F.E]$, containing the concrete type c of the object *loc*, a field map F which maps field names of c to their values, and a new dynamic effect map E which *maps the method names of c to their runtime effects*. The effect map is necessary in tracking and updating of runtime effects of dynamically dispatched methods, for concretization of open effects and efficient verification of their disjointness, as the values of open references change during the program execution. Performance efficiency of these mechanisms is shown in §4. Dynamic semantics rules are presented using a one-step call-by-value reduction relation and a set of evaluation contexts $\mathbb{E}$ [21] which specify the evaluation order. Omitted semantics rules and auxiliary functions can be found in our report [32].

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

$$\begin{array}{lllllll}
\Sigma & ::= & \langle e, \mu \rangle & \text{“Configurations”} & F & ::= & \{f_i \mapsto v_i\}_{i \in \mathbb{N}} \quad \text{“Field Maps”} \\
\mu & ::= & \{loc_i \mapsto o_i\}_{i \in \mathbb{N}} & \text{“Stores”} & v & ::= & \mathtt{null} \mid loc \mid n \quad \text{“Values”} \\
o & ::= & [c\,.\,F\,.\,E] & \text{“Object Records”} & E & ::= & \{m_i \mapsto \sigma_i\}_{i \in \mathbb{N}} \quad \text{“Effect Maps”}
\end{array}$$

Evaluation contexts: $\mathbb{E} \;::=\; - \;\mid\; t \;\; var \;=\; \mathbb{E}\mathtt{;}\, e$

Fig. 7. Domains and evaluation contexts.

3.2 Tracking and Updating of Open References

There are several rules in *OpenEffectJ*'s dynamic semantics, including the rule for object creation and setting a field, that are key in tracking of the open references and updating the concretization of open effects, which are dependent on these references.

(NEW)

$$\frac{\begin{array}{c} loc \notin dom(\mu) \\ F = \{f \mapsto default(f) \mid f \in fields(c)\} \qquad \mu' = \mu \oplus \{loc \mapsto [c.F.E]\} \\ E = \{m \mapsto \sigma \mid m \in methods(c)\textbf{.}\;\; findMeth(c,m) = (c', t, m(\overline{t\ var}), \sigma)\} \end{array}}{\langle \mathbb{E}[\mathtt{new}\ c()], \mu \rangle \hookrightarrow \langle \mathbb{E}[loc], \mu' \rangle}$$

The rule (NEW), in addition to initializing a new object in the memory, by assigning a fresh location *loc* to it, generates and initializes the effect map E for the newly created object. The effect map E maps the methods m of class c to their statically computed effects σ which have been computed using the typing rules as discussed in §2. The auxiliary function *findMeth* returns the definition of a method m of the class c in the class table CT. The function *default* returns the default value for each variable of a type. The operator $\oplus$ is an overriding operator such that if $\mu' = \mu \oplus \{loc \mapsto o\}$, then $\mu'(loc') = o$ if $loc' = loc$, otherwise $\mu'(loc') = \mu(loc')$. To illustrate, the object record for the newly created object `pr` of type `Pair` in Figure 1 is of the form $[Pair.\{fst \mapsto 0, snd \mapsto 0, f \mapsto \mathtt{null}\}.\{setOp \mapsto \{\top\}, apply \mapsto \{\mathbf{wr}(fst), \mathbf{wr}(snd), \mathbf{rd}(f), \mathbf{open}(f\ op\ \gamma)\}\}]$.

(SET)

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

Upon setting the field f of the object *loc* updates the concretization of all open effects that are directly or transitively dependent on the open field f, until a fixpoint is reached. This is done using the auxiliary function *update*, in Figure 8, that first concretizes the open effects in the effect map E of the object *loc*. If the effect map E changes to E', i.e. $E \neq E'$, then it updates the concretization of all other transitively dependent open effects. The function *reverse* backward traverses the object graph, starting from *loc*, and finds all the open fields dependent on f, directly or transitively. An open field g is dependent on the open field f, if g, directly or transitively, points to the object *loc* containing the field f. In practice, reverse pointers can be used to optimize this [8], as in our compiler's implementation.

Concretization of Open Effects. There are are two variations of concretization, shown in Figure 8: *(i)* concretization of an open effect when its open field is set, as in (SET), by *concretize(1)* and *(ii)* concretization of an open effect in use cases such as translation of the fork in the rule (T-FORK-UNKNOWN) in §2.1, by *concretize(2)*. The function *concretize(1)*, fills in the placeholder γ in open effects of the form $\textbf{open}(f\ m\ \gamma)$, upon setting the field f of the object loc. Recall that $\textbf{open}(f\ m\ \gamma)$ represents the effect of the invocation of the method m on the open field f. If f is set to **`null`**, then invocation of m on f will not have any effects, thus replacing γ in $\textbf{open}(f\ m\ \gamma)$ by the concretized effect $\emptyset$, i.e. $\textbf{open}(f\ m\ \emptyset)$. If f is set to an object loc', then the invocation of m on f will be the invocation of m on the object loc', thus replacing γ in $\textbf{open}(f\ m\ \gamma)$ with the union of the concrete effects of the method m in the object loc', i.e. $\downarrow_C (E(m))$, and its concretized open effects, i.e. σ''. Note that, the open effect $\textbf{open}(f\ m\ \gamma)$ is concretized whenever its open field f is set. The function *concretize(2)* basically returns the effects concretized by the first variation, rather than directly concretizing them. This is because concretization of an open effect happens only when its open field is set. For a non-concretized open effect $\textbf{open}(f\ m\ \gamma)$ with the open field f in the object loc, its effect map E is searched till a concretized effect $\textbf{open}(f\ m\ \sigma)$ is found and σ is returned as the result. The current store μ in the configuration and variable **`this`** are implicitly passed to *concretize(2)* in which **`this`** is passed as the value for loc.

$$update(\mu, loc, f, v) = \begin{cases} \mu \quad \text{if } E = E', \\ \mu_n \text{ if } E \neq E', \\ \qquad \mu_i = update(\mu_{i-1}, loc_i, f_i, loc) \\ \qquad \{\langle loc_i, f_i\rangle\} = reverse(\mu, loc), 1 \leq i \leq n \\ \qquad \mu_0 = \mu \oplus \{loc \mapsto [c\,.\,F\,.\,E']\} \end{cases} \qquad \begin{array}{r} \textbf{where } E' = updateEff(\mu, f, v, E), \\ \mu(loc) = [c.F.E] \end{array}$$

$$reverse(\mu, loc) = \{\langle loc', f\rangle \mid F(f) = loc \wedge loc' \in dom(\mu) \wedge \mu(loc') = [c.F.E]\}$$

$$updateEff(\mu, f, v, E) = \{\ m \mapsto \{concretize_{(1)}(\mu, f, v, \varepsilon')\} \mid (m \mapsto \sigma) \in E\ \wedge\ \varepsilon' \in \sigma\}$$

$$concretize_{(1)}(\mu, f, v, \varepsilon) = \begin{cases} \textbf{open}(f\ m\ \emptyset) & \text{if } \varepsilon = \textbf{open}(f\ m\ \gamma),\ v = \texttt{null} \\ \textbf{open}(f\ m\ \sigma') & \text{if } \varepsilon = \textbf{open}(f\ m\ \gamma),\ v = loc', \\ & \quad [c.F.E] = \mu(loc'), \\ & \quad \sigma' = \downarrow_C (E(m)) \cup \\ & \quad \{\varepsilon' \in \sigma'' \mid \varepsilon'' \in \downarrow_O (E(m)) \wedge \varepsilon'' = \textbf{open}(f'\ m'\ \sigma'')\} \\ \varepsilon & \text{otherwise} \end{cases}$$

$$concretize_{(2)}(\mu, loc, \varepsilon) = \sigma \qquad \begin{array}{l} \textbf{where } \varepsilon = \textbf{open}(f\ m\ \gamma),\ [c.F.E] = \mu(loc), \\ \qquad \exists m' \in dom(E)\textbf{.}\ \textbf{open}(f\ m\ \sigma) \in E(m') \end{array}$$

$$concretize_{(2)}(\sigma) = (\bigcup concretize_{(2)}(\mu, \texttt{this}, \varepsilon)) \cup \downarrow_C (\sigma) \qquad \textbf{where } \varepsilon \in \downarrow_O (\sigma)$$

Fig. 8. Auxiliary functions *update* and *concretize*

To illustrate, consider concretization of the open effect $\textbf{open}(f\ op\ \gamma)$ of the method `apply` when its open field `f` is set, by the expression `pr.setOp(pf)`, on line 28 of Figure 1. In *concretize*$_{(2)}$, the parameter v will be equal to `pf` and thus the placeholder γ in the open effect, will be replaced by $\textbf{wr}(res)$, which is the effect of of method `op` of `pf`. Concretization of the open effect $\textbf{open}(f\ op\ \gamma)$ to $\textbf{open}(f\ op\ \textbf{wr}(res))$ in the effect of `apply` causes the *update* to be invoked which updates all open effects which are dependent on `pr`, using *reverse*. In Figure 1, there is no object pointing to `pr` and thus the fixpoint is reached and concretization stops.

3.3 Soundness of Open Effects

The type-and-effect encoding of open effects is proven sound using theorems that say: *(i)* statically computed effects are a sound approximation of concretized effects, Theorem 1; and *(ii)* concretized effects soundly approximate runtime effects, Theorem 2.

Theorem 1. [Concretized effects refine static effects] *Given an expression e with the statically computed effects* σ_s*, which could contain open effects, and its dynamic concretization* σ_c*, i.e.* $\sigma_c = concretize(\sigma_s)$*, if* $\Pi, A \vdash e \rightsquigarrow e' : (t, \sigma_s, A')$ *holds statically and* $(\mu, A) \vdash e' : (\sigma_c, A')$ *holds dynamically for the runtime configuration* $\langle e', \mu \rangle$*, then:* $\sigma_c \subseteq \sigma_s$*.*

Theorem 2. [Dynamic effects refine concretized effects] *For two configurations* $\Sigma = \langle e, \mu \rangle$ *and* $\Sigma' = \langle e', \mu' \rangle$*, if* Σ *transitions to* Σ' *producing runtime effect* η*, i.e.* $\Sigma \overset{\eta}{\hookrightarrow} \Sigma'$*, if concretized effects of e is* σ_c*, i.e.* $(\mu, A) \vdash e : (\sigma_c, A')$*, then there is a concretized effect* σ'_c *such that:*

(a) $(\mu', A_1) \vdash e' : (\sigma'_c, A'_1)$ *and* $\sigma'_c \subseteq \sigma_c$*;*
(b) $\eta \in \sigma_c$

Proof Sketch: Theorem 1 is proved by structural induction on derivations of $\Pi, A \vdash e \rightsquigarrow e' : (t, \sigma_s, A')$ and $(\mu, A) \vdash e' : (\sigma_c, A')$ whereas proof of Theorem 2 is by cases on transition steps for the transition relation $\Sigma \overset{\eta}{\hookrightarrow} \Sigma'$ [32].

4 Evaluation: Speedup and Overhead of Open Effects

We hypothesize that open effects is performance efficient while exposing safe concurrency opportunities in frameworks and libraries, that could be *extended with possibly concurrency-unsafe code by clients*. To test our hypothesis we implemented open effects on top of OpenJDK[7] and parallelized a representative set of frameworks and libraries using open effects and the following widely used concurrency techniques: *(i)* Deuce [1], software transactional memory (STM); *(ii)* Multiverse [2], STM; *(iii)* RoadRunner (RR) [19], runtime race detector[8]; and *(iv)* Manually tuned concurrency; and compared their speedups and overheads. Results of our experiments show that: open effects almost does as well as manually tuned concurrency, with the negligible overhead of only 0.27% to at most 4.06% and less overhead compared to other techniques.

[7] *OpenEffectJ*'s compiler and evaluations are available at `http://paninij.org/open/`.

[8] The race detection tool set $-tool = TL : RS : LS$ was used.

4.1 Setup

We use the following frameworks, benchmarks and libraries in our experiments: a map-reduce framework (MapReduce), adapted from JSR [3], a pipeline framework (Pipeline) [10], Monte Carlo benchmark (MonteCarlo) [43], JDK's merge sort (MergeSort) and array list (ArrayList) libraries [3], depth first search graph traversal (DFS), a numerical integration application (Integrate) [3] and a sequence alignment application (Alignment) [5]. *In terms of annotation overhead, except MapReduce, with 2 @open annotations, open effects versions of other applications needed only 1 annotation.*

Client Code. In the MapReduce, the map phase computes the sum of the magnitudes formula $Math.sqrt(2*Math.pow(o,2))$ for each element o in a set of 100 million integers and the reduce step is simply addition of the results. Pipeline models Radix Sort in which the first stage generates a stream of 8 arrays of 1 million integers each and subsequent stages sort the arrays on different radixes. MergeSort sorts a list of 10 million randomly generated integers. For ArrayList, we apply the hash (Hash), prefix sum (Prefix) computations, illustrated in Figure 1, and a heavier computation (Heavy), which computes the same formula as in the MapReduce, on an array with 20 million elements. DFS solves an N-queens problem, with n equal to 11. Integration 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. Alignment uses a constant function returning -1 if two characters do not match for aligning two words of sizes 100 and 1 million.

Hardware. All our experiments were run on a system with a total of 4 cores (Intel Core2 chips 2.40GHz) running Fedora GNU/Linux. For each experiment, an average of the results over 30 runs was taken and the default JVM parameters were used.

4.2 Performance Evaluation

Application	**Serial**	**Manual**	**RoadRunner** [19]		**Deuce** [1]		**Multiverse** [2]		**OpenEffectJ**			**Pattern**
	time(s)	**time**	**time**	**overhead**	**time**	**overhead**	**time**	**overhead**	**time**	**overhead**	**speedup**	
Hash	0.13	0.12	0.85	585.5%	44.11	35400.0%	8.57	6801.0%	0.12	0.3%	1.07	Forall
Heavy	1.31	0.39	1.12	203.2%	34.87	8952.3%	12.43	3128.0%	0.39	0.09%	3.39	Forall
Prefix	0.12	×	×	×	×	×	×	×	0.12	1.62%	0.98	Forall
Alignment	2.44	1.86	21.50	1054.7%	14.91	700.9%	8.34	348.0%	1.93	4.07%	1.26	Forall
MonteCarlo	3.87	1.22	2.04	67.3%	10.52	762.7%	1.33	8.9%	1.25	2.68%	3.10	Forall
Pipeline	2.25	2.11	3.48	64.7%	↑	↑	2.21	4.5%	2.12	0.62%	1.06	Pipeline
MergeSort	2.71	1.32	3.39	156.1%	9.61	626.3%	16.00	1109.6%	1.34	1.67%	2.02	Recursive
DFS	18.83	9.20	17.88	94.3%	9.79	6.5%	12.23	32.9%	9.23	0.28%	2.04	Recursive
MapReduce	7.03	1.94	3.81	96.8%	5.25	170.6%	10.76	454.5%	1.91	-1.46%	3.68	Recursive
Integrate	2.13	0.59	1.53	158.9%	1.46	146.2%	2.42	309.7%	0.61	2.36%	3.50	Recursive

Fig. 9. Performance Experiments. × indicates result discrepancies because of sequential inconsistencies and ↑ shows running out of memory after a considerably long time.

Definition 1. *(Runtime Overhead and Speedup) For a program p, with its sequential, open effects and manually tuned parallel versions, which respectively take T1, T2, and T3 seconds for their execution, the speed up is T1 / T2 and overhead is (T2 - T3) / T3.*

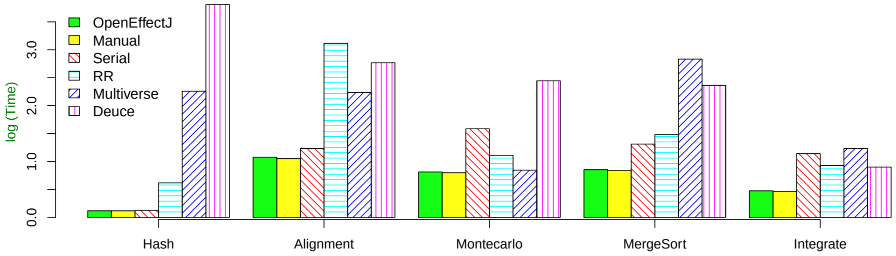


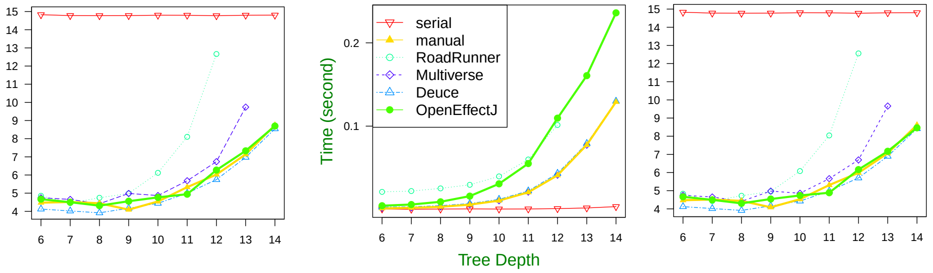
Fig. 10. Open effects are almost as good as manually tuned concurrency.

Figure 9 and Figure 10 show the performance results of running our experiments in terms of speedup and runtime overhead, as defined in Definition 1. Our results show that the open effect (*OpenEffectJ*) versions of the evaluation applications are almost as fast as the manually tuned concurrent versions and incur very small overhead, ranging from 0.27% to 4.06% at most, which is significantly less compared to other effect analysis techniques used in our experiments. In Figure 9, Multiverse or Deuce versions of the applications, run slower especially for ArrayList, because Multiverse creates a separate runnable object for each transaction which causes a slow down in applications with large number of transactions; Deuce, besides the transaction creation overhead, stores array access effects in a fine-grained manner, which could cause slow down for large arrays. *OpenEffectJ*, instead, uses an indexed array effect [9, 38] and a purity analysis, to decide about disjointness of effects. It is conceivable that using similar techniques, performance for Multiverse or Deuce's versions could be improved.

4.3 Concretization of Open Effects for Nested Objects

One may argue that concretization of open effects at runtime may not be efficient enough especially for deeply-nested data structures such as trees, mainly because setting an open field could cause updating various other open effects dependent on that field, as discussed in §3.2. However, this may not be always the case, especially when the benefits of the open effects outweigh its overheads. To investigate, we implemented a Fibonacci algorithm, adapted from OpenJDK's fork/join framework [3], using both open effects and techniques in Figure 9. To compute the n-th Fibonacci number $Fib(n)$, the algorithm uses a binary tree in which right and left subtrees represents $Fib(n-1)$ and $Fib(n-2)$ respectively, and the root adds them together to compute $Fib(n) = Fib(n-1) + Fib(n-2)$. This algorithm has two phases of *(i)* construction of the tree and *(ii)* computation of the Fibonacci numbers for the nodes. We ran experiments on trees of depths from 6 to 14, which can have up to 2^{14}-1 nodes, and the same number of open effects' concretizations, to compute $Fib(45)$.

Figure 11 shows the time each technique takes to compute the Fibonacci numbers, for the construction and computation phases as well as their total. It shows that for nested objects, open effects does almost as well as the manually tuned concurrent version and better than other concurrent approaches, which is consistent with the results in Figure 9. This is despite the fact that the construction of the Fibonacci trees may take more time compared to other techniques, because of the concretization of open effects. However, benefits of open effects in the computation phase outweigh this overhead.



Total Construction Computation

Fig. 11. Building Fibonacci tree and computation of n-th Fibonacci number.

5 Related Work

In this section, we compare open effects with related works on reasoning about effects of programs, in three categories of hybrid, static and dynamic techniques.

Hybrid. Open effects is closest in spirit to the ideas of gradual typing [41] and hybrid type checking [27] that blend advantages of static and dynamic type checking. Similarly, open effects blends the advantages of static and dynamic effect systems. Similar to Open Type Switch (Mach7) [44], which allows users to choose between type hierarchy openness and efficiency, open effects lets programmer choose the openness of the effects of dynamically dispatched method invocations. Synchronization via scheduling (SVS) [8] computes effects of concurrent tasks as their reachable object graph, for programs written in a simple C-like language, with no dynamic dispatch. However, open effects support a full OO language with the support for overriding and dynamic dispatch, which makes accurate effect computation more challenging [20], and use smaller effect sets compared to reachability graphs for effect computations. TWEJava [26] lets the programmer specify the effect of each task and has a scheduler that coordinates these concurrent tasks. However, open effects require only open annotations, compared to task specifications. In concurrent revisions [13], programmers annotate shared objects tasks could conflict on and provide their merge functions, and each task keeps a local copy of these objects to avoid data races, using copy-on-write. In contrast, open effects check for effect conflicts before the execution, either statically or at runtime.

Static. Boyapati *et al.* [12] propose an ownership type system for deadlock and data race detection, Gordon *et al.* [23] use uniqueness and reference immutability to provide safe parallelism, Deterministic Parallel Java (DPJ) [10] provides determinism for parallel programs using effect parameters and effect constraints, such as effect containment. There are also other works on effect systems [24] for sequential programs such as data groups [29], ownership type systems [14] and heap representation techniques [14]. However, open effects is a hybrid technique that combines static and dynamic type-and-effect to better handle invocation of dynamically dispatched methods, without restrictions of effect containment between a type and its subtypes.

Dynamic. FastTrack [17], Goldilocks [16], Pacer [11], and the work of Smaragdakis *et al.* [42] are data race detection techniques which monitor memory accesses. Transactional memory techniques [7, 15, 25, 31, 39, 40, 47] optimistically execute tasks concurrently, while also monitoring memory accesses, and rollback whenever effects of the tasks conflict. These techniques monitor memory footprints of a program, for all of its references. However, open effects only monitors open references and decide before execution of tasks, either statically or dynamically, if they need to be executed sequentially, because of conflicting effects, and thus do not roll back. In Galois [28], user provided commutativity specifications for methods are checked dynamically at runtime and the execution is rolled back if they are violated. However, open effects does not need commutativity specifications and does not roll back the execution.

6 Conclusion and Future Work

We proposed open effects, a trust-but-verify hybrid type-and-effect system to safely expose concurrency opportunities in invocations of dynamically dispatched methods in concurrent programs with open world assumption. Open effects trusts the programmer's knowledge in the form of open annotations which represent optimistic assumptions about disjointness of effects, and verifies them statically or dynamically. Our performance evaluations show that our prototype implementation of open effects is quite efficient in exposing concurrency, by combining static and dynamic analyses, and performs as well as manually tuned concurrency, with negligible overhead of only *0.27 – 4.06%*. Since open effects is *complementary* to previous static and dynamic analyses, we believe this overhead could be decreased even more by integrating more sophisticated static and dynamic analyses, which is one venue for future work. Another direction for future work, is to explore a logical extreme, in which all references are implicitly open and a static analysis systematically eliminates ones causing unacceptable overheads.

References

1. https://sites.google.com/site/deucestm/
2. http://multiverse.codehaus.org/
3. JSR-166y for Java 7. http://gee.oswego.edu/dl/concurrency-interest/
4. Abadi, M., Flanagan, C., Freund, S.: Types for safe locking: Static race detection for Java. TOPLAS '06 28
5. Aluru, S., Futamura, N., C, K.M.: Parallel biological sequence comparison using prefix computations. Journal of Parallel and Distributed Computing (2003)
6. Benton, N., Buchlovsky, P.: Semantics of an effect analysis for exceptions. In: TLDI '07
7. Berger, E.D., Yang, T., Liu, T., Novark, G.: Grace: safe multithreaded programming for C/C++. In: OOPSLA '09
8. 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
9. Bocchino, R., Adve, V., Dig, D., Adve, S., Heumann, S., Komuravelli, R., Overbey, J., Simmons, P., Sung, H., Vakilian, M.: A type and effect system for Deterministic Parallel Java. In: OOPSLA '09
10. Bocchino, R.L., Adve, V.S.: Types, regions, and effects for safe programming with object-oriented parallel frameworks. In: ECOOP '11
11. Bond, M., Coons, K., McKinley, K.: Pacer: proportional detection of data races. In: PLDI'11
12. Boyapati, C., Lee, R., Rinard, M.: Ownership types for safe programming: preventing data races and deadlocks. In: OOPSLA '02

13. Burckhardt, S., Baldassin, A., Leijen, D.: Concurrent programming with revisions and isolation types. In: OOPSLA '10
14. Cameron, N., Drossopoulou, S., Noble, J., Smith, M.: Multiple ownership. In: OOPSLA '07
15. Ding, C., Shen, X., Kelsey, K., Tice, C., Huang, R., Zhang, C.: Software behavior oriented parallelization. In: PLDI '07
16. Elmas, T., Qadeer, S., Tasiran, S.: Goldilocks: a race and transaction-aware Java runtime. In: PLDI '07
17. Flanagan, C., Freund, S.: FastTrack: efficient and precise dynamic race detection. In: PLDI '09
18. Flanagan, C., Freund, S.: Redcard: Redundant check elimination for dynamic race detectors. In: ECOOP' 13
19. Flanagan, C., Freund, S.: The roadrunner dynamic analysis framework for concurrent programs. In: PASTE '10
20. Flanagan, C., Freund, S.: Type-based race detection for Java. In: PLDI '00
21. Flatt, M., Krishnamurthi, S., Felleisen, M.: A Programmer's Reduction Semantics for Classes and Mixins. In: Formal Syntax and Semantics of Java. Springer (1999)
22. Gifford, D., Lucassen, J.: Integrating functional and imperative programming. In: LFP '86
23. Gordon, C., Parkinson, M., Parsons, J., Bromfield, A., Duffy, J.: Uniqueness and reference immutability for safe parallelism. In: OOPSLA '12
24. Greenhouse, A., Boyland, J.: An object-oriented effects system. In: ECOOP '99
25. Herlihy, M., Moss, J.E.B.: Transactional memory: architectural support for lock-free data structures. In: ISCA '93
26. Heumann, S., Adve, V., Wang, S.: The tasks with effects model for safe concurrency. In: PPoPP '13
27. Knowles, K., Flanagan, C.: Hybrid type checking. TOPLAS '10, 32
28. Kulkarni, M., Pingali, K., Walter, B., Ramanarayanan, G., Bala, K., Chew, L.P.: Optimistic parallelism requires abstractions. In: PLDI '07
29. Leino, K.R.M.: Data groups: specifying the modification of extended state. In: OOPSLA '98
30. Leroy, X., Pessaux, F.: Type-based analysis of uncaught exceptions. TOPLAS '00, 22(2)
31. Lesani, M., Palsberg, J.: Communicating memory transactions. In: POPL '11
32. Long, Y., Bagherzadeh, M., Rajan, H.: Open Effects: Programmer-guided Effects for Open World Concurrent Programs. Tech. rep., Iowa State Univ. (2013)
33. Lucassen, J.M., Gifford, D.K.: Polymorphic effect systems. In: POPL '88
34. Milner, R.: A theory of type polymorphism in programming (17), 348–375 (1978)
35. Neamtiu, I., Hicks, M., Foster, J.S., Pratikakis, P.: Contextual effects for version-consistent dynamic software updating and safe concurrent programming. In: POPL '08
36. Reiter, R.: On closed world data bases. Springer (1978)
37. Ru, S., Rinard, M.: Purity and side effect analysis for Java programs. In: VMCAI '05
38. Rugina, R., Rinard, M.: Automatic parallelization of divide and conquer algorithms. In: PPoPP '99
39. Shavit, N., Touitou, D.: Software transactional memory. In: PODC '95
40. Shpeisman, T., Menon, V., Adl-Tabatabai, A.R., Balensiefer, S., Grossman, D., Hudson, R., Moore, K., Saha, B.: Enforcing isolation and ordering in STM. In: PLDI '07
41. Siek, J., Taha, W.: Gradual typing for objects. In: ECOOP '07
42. Smaragdakis, Y., Evans, J.M., Sadowski, C., Jaeheon, Y., Flanagan, C.: Sound predictive race detection in polynomial time. In: POPL '12
43. Smith, L., Bull, J., Obdrizalek, J.: A parallel Java Grande benchmark suite. In: SC '01
44. Solodkyy, Y., Dos Reis, G., Stroustrup, B.: Open and efficient type switch for C++. In: OOPSLA '12
45. Talpin, J.P., Jouvelot, P.: Polymorphic type, region and effect inference. JFP '92, 2(3)
46. Talpin, J.P., Jouvelot, P.: The type and effect discipline. Inf. Comput. '94, 111
47. Welc, A., Jagannathan, S., Hosking, A.: Safe Futures for Java. In: OOPSLA '05

A Open Parameters and Local Variables

So far, open effects have been discussed only in terms of open fields. However, open effects are not limited to open fields and can support open parameters and open local variables as well, especially when object fields flow into parameters or local variables, as illustrated in Figure 12. In this figure, the method `Apply`, on the left, has an open parameter `g`, used as the receiver for the invocation of the method `op`. This causes the effects of the method `Apply` to be **open**(g op γ). Later this open effect is concretized to **open**(f op γ), lines 5–7 where `this.f` is passed to `Apply` as the parameter `g`. For the open variable `var`, line 2 on the right, the open effect **open**(var op γ) is generated for each method invocation on lines 4 and 5 which is concretized to **open**(par op γ) later because the local variable `var` is set to the value `par`, on line 2.

```
1 private final int Apply (@open Op g, int x) {
2    g.op(x)
3 }
4 int apply() {
5  // f and g are aliases.
6  fst = Apply(this.f, fst);
7  snd = Apply(this.f, snd)
8 }
```

```
1 int apply(Op par) {
2  @open Op var = par;
3  fork {
4   var.op(1),
5   var.op(2)
6  }
7 }
```

Fig. 12. Open parameter `g`, on the left, and open variable `var`, on the right.

B Static Semantics: Omitted Details

Figure 13 shows the rest of the *OpenEffectJ*'s typing rules omitted from §2.1. The rule (T-PROGRAM) says that a program type checks if all its declarations type check. The rule (T-CLASS) says that a class declaration type checks if all, all the newly declared fields are not fields of its super class, checked by the auxiliary function *validF*, its super class d is defined in the class table CT, checked by the auxiliary function *isClass*; and finally, all its declared methods type check.

Figure 14 shows the auxiliary functions used in the typing rules. The auxiliary function *override*, used in (T-METHOD) requires that an overriding and overriden method in a subtype and its supertype have compatible types for their parameters and return values. As discussed previously, because of the flexibility of open effects, this function puts no restriction on the effects of the overriding and overriden methods. The operation $\sqcap$ computes the intersection of two aliasing environments, i.e. $A_1 \sqcap A_2$ returns a map containing the aliasing information that exists in both A_1 and A_2.

C Dynamic Semantics: Omitted Details

Figure 15 shows the dynamic semantics rules that were omitted from §3 along with their auxiliary functions in Figure 16. The evaluation relation $\Sigma \overset{\eta}{\hookrightarrow} \Sigma'$ says that during

(T-PROGRAM)
$$\frac{\forall decl \in \overline{decl}.\ \vdash decl \rightsquigarrow decl' : \text{OK} \qquad \vdash e \rightsquigarrow e' : (t,\sigma,A)}{\vdash \overline{decl}\ e \rightsquigarrow \overline{decl'}\ e' : (t,\sigma,A)}$$

(T-CLASS)
$$\frac{\begin{array}{c} \forall [@open]\ t\ f \in \overline{field}.\ validF(f,d) \\ isClass(d) \qquad \forall meth \in \overline{meth}.\ \vdash meth \rightsquigarrow meth' : (t',\sigma,A)\ \mathbf{in}\ c \end{array}}{\vdash \mathtt{\mathbf{class}}\ c\ \mathtt{\mathbf{extends}}\ d\ \{\ \overline{field}\ \ \overline{meth}\ \} \rightsquigarrow \mathtt{\mathbf{class}}\ c\ \mathtt{\mathbf{extends}}\ d\ \{\ \overline{field}\ \ \overline{meth'}\ \} : \text{OK}}$$

(T-GET)
$$\frac{\begin{array}{c} \Pi,A \vdash \mathtt{\mathbf{this}} : c \\ typeOf(f) = (d,[@open]\ t) \qquad c <: d \end{array}}{\Pi,A \vdash \mathtt{\mathbf{this}}.f \rightsquigarrow \mathtt{\mathbf{this}}.f : (t,\mathbf{rd}(f),A)}$$

(T-BINARY)
$$\frac{\begin{array}{c} \Pi,A \vdash x_1 : int \\ \Pi,A \vdash x_2 : int \end{array}}{\Pi,A \vdash x_1 \circ x_2 : int}$$

(T-VAR)
$$\frac{(x:t) \in \Pi}{\Pi,A \vdash x : t}$$

(T-NULL)
$$\frac{isClass(t)}{\Pi,A \vdash \mathtt{\mathbf{null}} : t}$$

(T-NUM)
$$\Pi,A \vdash n : int$$

(T-NEW)
$$\frac{isClass(c)}{\Pi,A \vdash \mathbf{new}\ c() : c}$$

(T-LOC)
$$\frac{(loc:t) \in \Pi}{\Pi,A \vdash loc : t}$$

(T-CONDITION)
$$\frac{\Pi,A \vdash x : bool \qquad \Pi,A \vdash e_1 \rightsquigarrow e_1' : (t,\sigma,A_1) \qquad \Pi,A \vdash e_2 \rightsquigarrow e_2' : (t,\sigma',A_2)}{\Pi,A \vdash \mathtt{\mathbf{if}}\ x\ \mathtt{\mathbf{then}}\ e_1\ \mathtt{\mathbf{else}}\ e_2 \rightsquigarrow \mathtt{\mathbf{if}}\ x\ \mathtt{\mathbf{then}}\ e_1'\ \mathtt{\mathbf{else}}\ e_2' : (t,\sigma \cup \sigma',A_1 \sqcap A_2)}$$

Fig. 13. *OpenEffectJ*'s omitted type-and-effect rules

the evaluation, a configuration Σ transitions to another configuration Σ' producing the runtime memory read and write effects of η. The field and object sensitive runtime effect $\mathbf{read}(loc,f)$ repesents reading the field f of an object *loc* whereas $\mathbf{write}(loc,f)$ shows writing into the field. The transition relation $\Sigma \hookrightarrow \Sigma'$ represents a transition with no memory effects.

C.1 Proof of Effect Refinement

To prove the type-and-effect system of *OpenEffectJ* sound, we should prove that the dynamic runtime effects of a program refine its static effects, that are computed by the typing rules. We prove this using two theorems which say that:

(i) concretized effects are a sound approximation of statically computed effects, Theorem 1; and
(ii) concretized effects soundly approximate runtime effects, Theorem 2.

Preliminary Definitions We first present some definitions used in the proofs of Theorem 1 and Theorem 2.

Definition 2. *[Dynamic Trace] A dynamic trace $\overline{\eta}$ for an execution of a program is the sequence of dynamic effects η happening during its execution, where η can be a read effect $\mathbf{read}(loc,f)$ for field f of the object loc, or a write effect $\mathbf{write}(loc,f)$.*

$$\frac{\begin{array}{c} CT(c) = \texttt{class}\ c\ \texttt{extends}\ d\ \{\ \overline{field}\ \overline{meth}\ \} \\ \nexists meth \in \overline{meth}\textbf{.}\ meth = t\ \sigma\ m(\overline{v\ var})\{e\} \qquad override(m,d,\bar{t} \to t) \end{array}}{override(m,c,\bar{t} \to t)}$$

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

$$\frac{\begin{array}{c} CT(c) = \texttt{class}\ c\ \texttt{extends}\ d\ \{\ \overline{field}\ \overline{meth}\ \} \\ \exists meth \in \overline{meth}\ \textbf{.}\ \ meth = (t,\sigma,m(\overline{t\ var})\{e\}) \end{array}}{findMeth(c,m) = (c,t,m(\overline{t\ var})\{e\},\sigma)}$$

$$\frac{\begin{array}{c} CT(c) = \texttt{class}\ c\ \texttt{extends}\ d\ \{\ \overline{field}\ \overline{meth}\ \} \\ \nexists meth \in \overline{meth}\ \textbf{.}\ \ meth = (t,\sigma,m(\overline{t\ var})\{e\}) \qquad findMeth(d,m) = l \end{array}}{findMeth(c,m) = l}$$

$$\frac{\begin{array}{c} CT(c) = \texttt{class}\ c\ \texttt{extends}\ d\ \{\ \overline{field}\ \overline{meth}\ \} \\ \nexists field \in \overline{field}\ \textbf{.}\ \ field = [@open]\ t\ f; \qquad validF(f,d) \end{array}}{validF(f,c)} \qquad validF(f,Object)$$

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

$$\frac{\texttt{class}\ c\ \texttt{extends}\ d\{\ \overline{field}\ \overline{meth}\ \} \in CT \qquad \exists [@open]\ t\ f \in \overline{field}}{typeOf(f) = (c,[@open]\ t)}$$

$$A_1 \sqcap A_2 = \{x = e \mid (A_1 \vdash x = e) \wedge (A_2 \vdash x = e)\}$$

Fig. 14. Rest of *OpenEffectJ*'s auxiliary functions.

Definition 3. *[Static effect inclusion] A static effect ε is included in an effect set σ, which may contain open effects, written as $\varepsilon \in \sigma$, if:*

- *either $\varepsilon \in \downarrow_C(\sigma)$;*
- *or $\exists$ **open**$(f\ m\ \sigma') \in \downarrow_O(\sigma)\ \wedge\ \varepsilon \in \sigma'$.*

Definition 4. *[Dynamic effect refines static effect] A dynamic runtime effect η refines a static effect ε, written as $\eta \propto \varepsilon$, if:*

- *either $\eta =$ **read**$(loc,f) \wedge \varepsilon \in \{$**rd**$(f),$**wr**$(f)\}$;*
- *or $\eta =$ **write**$(loc,f) \wedge \varepsilon =$ **wr**(f).*

In this definition, a write effect covers a read effect.

Definition 5. *[Static effect refinement] An effect set σ' refines another effect set σ if $\sigma' \subseteq \sigma$.*

Evaluation relation: $\overset{\eta}{\hookrightarrow} : \Sigma \overset{\eta}{\dashrightarrow} \Sigma$

(SET)

$$\frac{\begin{array}{c}[c.F.E] = \mu(loc)\\ \mu_0 = \mu \oplus (loc \mapsto [c.(F \oplus (f \mapsto v)).E])\\ \mu' = update(\mu_0, loc, f, v)\end{array}}{\langle \mathbb{E}[loc.f = v], \mu\rangle \xhookrightarrow{\mathbf{write}(loc,f)} \langle \mathbb{E}[v], \mu'\rangle}$$

(GET)

$$\frac{\mu(loc) = [c.F.E] \qquad v = F(f)}{\langle \mathbb{E}[loc.f], \mu\rangle \xhookrightarrow{\mathbf{read}(loc,f)} \langle \mathbb{E}[v], \mu\rangle}$$

(CALL)

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

(BINARY)

$$\frac{v = v_1 \circ v_2}{\langle \mathbb{E}[v_1 \circ v_2], \mu\rangle \hookrightarrow \langle \mathbb{E}[v], \mu\rangle}$$

(DEFINE)

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

(CONDITION-TRUE)

$$\langle \mathbb{E}[\mathtt{if}\ true\ \mathtt{then}\ e\ \mathtt{else}\ e'], \mu\rangle \hookrightarrow \langle \mathbb{E}[e], \mu\rangle$$

(CONDITION-FALSE)

$$\langle \mathbb{E}[\mathtt{if}\ false\ \mathtt{then}\ e\ \mathtt{else}\ e'], \mu\rangle \hookrightarrow \langle \mathbb{E}[e'], \mu\rangle$$

Fig. 15. *OpenEffectJ*'s dynamic semantics rules.

$$\frac{CT(c) = \mathtt{class}\ c\ \mathtt{extends}\ d\ \{\ \overline{field}\ \overline{meth}\ \}}{methods(c) = methods(d) \cup \{m \mid (t, \sigma, m(\overline{t\ var})\{e\} \in \overline{meth}\}}$$

$$\frac{CT(c) = \mathtt{class}\ c\ \mathtt{extends}\ d\ \{\ \overline{field}\ \overline{meth}\ \}}{fields(c) = fields(d) \cup \{f \mid ([@open]t\ f) \in \overline{field}\}}$$

$$\frac{typeOf(f) = (d, [@open]\ int)}{default(f) = 0} \qquad \frac{typeOf(f) = (d, [@open]\ bool)}{default(f) = false} \qquad \frac{typeOf(f) = (d, [@open]\ c)}{default(f) = \mathtt{null}}$$

Fig. 16. Rest of *OpenEffectJ*'s auxiliary functions its dynamic semantics.

Definition 6. *[Effect equivalent stores] Two stores μ and μ' are effect equivalent, written as $\mu \cong \mu'$, if:*

- *$dom(\mu) \subseteq dom(\mu')$; and*
- *$\forall loc \in \mu, \mu(loc) = [c.F.E] \Rightarrow \mu'(loc) = [c.F'.E]$, for some F'.*

Definition 7. *[Well-formed object] An object record $o = [c.F.E]$ is a well-formed in μ, written as $\mu \vdash o$, if for all open effects $\mathbf{open}(f\ m\ \sigma_0) \in \sigma \in rng(E)$:*

- *either $(F(f) = loc) \wedge (\mu(loc) = [c'.F'.E']) \wedge (E'(m) \subseteq \sigma_0)$;*

- *or* $(F(f) = \texttt{null}) \wedge (\sigma_0 = \emptyset)$*;*
- *or* $(\mathit{typeOf}(f) = (c, \mathit{int}))$*;*
- *or* $(\mathit{typeOf}(f) = (c, \mathit{bool}))$*.*

Definition 8. *[Well-formed location] A location loc is well-formed in the store* μ*, written* $\mu \vdash loc$*, if:*

- *either* $\mu(loc) = [c.F.E]$, $\forall m \in dom(E)$. $\mathit{findMeth}(c,m) = (c', t, m(\overline{t\ var})\{e\}, \sigma') \wedge (\mu, \emptyset) \vdash [loc/\texttt{this}]e : (\sigma, A)$*, then* $\sigma \subseteq E(m)$*;*
- *or* $\mu(loc) = \texttt{null}$*.*

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

Effect Concretization. Figure 17 shows the rules for computation of concretized effects for *OpenEffectJ*'s expressions. In this figure, the effect judgement $(\mu, A) \vdash e : (\sigma, A')$ says that the expression e in a runtime configuration $\langle \mu, e \rangle$ with store μ and the aliasing environment A, has the concretized effect σ. The rule (E-CALL-OPEN) uses the *concretize* auxiliary function in Figure 8 for concretization of effects of a dynamically dispatched method invocation $x_0.m(\overline{x})$. The rules (E-GET) and (E-SET) assign a concretized effect $\mathbf{read}(loc,f)$ and $\mathbf{write}(loc,f)$ to the field read and write expressions. For other expressions, e.g. (T-DEFINE), their concretized effects is the union of the concretized effects of their subexpressions.

Theorem 1: Concretized effects refine static effects. *Given an expression e with statically computed effects* σ_s*, which could contain open effects, and its dynamic concretization* σ_c*, i.e.* $\sigma_c = concretize(\sigma_s)$*, if* $\Pi, A \vdash e \rightsquigarrow e' : (t, \sigma_s, A')$ *holds statically and* $(\mu, A) \vdash e' : (\sigma_c, A')$ *dynamically for the runtime configuration* $\langle e', \mu \rangle$*, then* $\sigma_c \subseteq \sigma_s$*.*

Proof: The proof is by a straightforward structural induction on the derivation of $\Pi, A \vdash e \rightsquigarrow e' : (t, \sigma, A')$ and $(\mu, A) \vdash e' : (\sigma', A')$.

1. For the base cases (GET), (SET), (SET-OPEN), (VAR), (NULL), (GET), (NUM), (NEW), (LOC), (BINARY), (CALL-PURE), (CALL), with no subexpressions, it is obvious that the effects are the same in the typing rules, §2 and Figure 13, and the effect judgment rules, Figure 17.

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.

2. (IF).

$$\frac{\begin{array}{c}\Pi, A \vdash x \rightsquigarrow x : (bool, \emptyset, A) \\ \Pi, A \vdash e_0 \rightsquigarrow e_0' : (t, \sigma_0, A_0) \qquad \Pi, A \vdash e_1 \rightsquigarrow e_1' : (t, \sigma_1, A_1)\end{array}}{\Pi, A \vdash \texttt{if}\ x\ \texttt{then}\ e_0\ \texttt{else}\ e_1 \rightsquigarrow \texttt{if}\ x\ \texttt{then}\ e_0'\ \texttt{else}\ e_1' : (t, \sigma_0 \cup \sigma_1, A_0 \sqcap A_1)}$$

$$\frac{(\mu, A) \vdash x : (\emptyset, A) \qquad (\mu, A) \vdash e_0' : (\sigma_0', A_0) \qquad (\mu, A) \vdash e_1' : (\sigma_1', A_1)}{(\mu, A) \vdash \texttt{if}\ x\ \texttt{then}\ e_0'\ \texttt{else}\ e_1' : (\sigma_0' \cup \sigma_1', A_0 \sqcap A_1)}$$

By IH, $\sigma_0 \subseteq \sigma_0'$ and $\sigma_1 \subseteq \sigma_1'$. Therefore $(\sigma' = \sigma_0' \cup \sigma_1') \subseteq (\sigma_0 \cup \sigma_1 = \sigma)$.

(E-CALL-OPEN)

$$\frac{A \vdash x = loc.f \qquad \mathbf{open}(f\ m\ \sigma) = concretize(\mu, loc, \mathbf{open}(f\ m\ \gamma)) \qquad typeOf(f) = (d, @open\ c_0)}{(\mu,A) \vdash x.m(\overline{x}) : (\sigma, A)}$$

(E-CALL-LOC)

$$\frac{\mu(loc) = [c.F.E] \qquad E(m) = \sigma}{(\mu,A) \vdash loc.m(\overline{x}) : (\sigma, A)}$$

(E-CALL)

$$(\mu,A) \vdash x.m(\overline{x}) : (\top, \emptyset)$$

(E-GET)

$$(\mu,A) \vdash x.f : (\mathbf{rd}(f), A)$$

(E-GET-LOC)

$$(\mu,A) \vdash loc.f : (\mathbf{rd}(f), A)$$

(E-SET-OPEN)

$$\frac{typeOf(f) = (c, @open\ c_0)}{(\mu,A) \vdash x.f = x' : (\top, \emptyset)}$$

(E-SET-OPEN-LOC)

$$\frac{typeOf(f) = (c, @open\ c_0)}{(\mu,A) \vdash loc.f = x : (\top, \emptyset)}$$

(E-SET)

$$\frac{typeOf(f) = (c,t)}{(\mu,A) \vdash x.f = x' : (\mathbf{wr}(f), A \backslash f \cup \{x' = x.f\})}$$

(E-SET-LOC)

$$\frac{typeOf(f) = (c,t)}{(\mu,A) \vdash loc.f = x : (\mathbf{wr}(f), A \backslash f \cup \{x = loc.f\})}$$

(E-NEW)

$$(\mu,A) \vdash \texttt{new}\ c() : (\emptyset, A)$$

(E-VAR)

$$(\mu,A) \vdash var : (\emptyset, A)$$

(E-NULL)

$$(\mu,A) \vdash \texttt{null} : (\emptyset, A)$$

(E-LOC)

$$(\mu,A) \vdash loc : (\emptyset, A)$$

(E-DEFINE)

$$\frac{(\mu,A) \vdash e_1 : (\sigma_1, A1) \qquad (\mu, A1; x = e_1) \vdash e_2 : (\sigma_2, A_2)}{(\mu,A) \vdash t\ var = e_1; e_2 : (\sigma_1 \cup \sigma_2, A_2)}$$

(E-BINARY)

$$(\mu,A) \vdash x_1 \circ x_2 : (\emptyset, A)$$

(E-BINARY-LOC)

$$(\mu,A) \vdash v_1 \circ v_2 : (\emptyset, A)$$

(E-NUMBER)

$$(\mu,A) \vdash n : (\emptyset, A)$$

(E-CONDITION)

$$\frac{(\mu,A) \vdash e_0 : (\sigma_0, A_0) \qquad (\mu,A) \vdash e_1 : (\sigma_1, A_1)}{(\mu,A) \vdash \texttt{if}\ x\ \texttt{then}\ e_0\ \texttt{else}\ e_1 : (\sigma_0 \cup \sigma_1, A_0 \sqcap A_1)}$$

Fig. 17. *OpenEffectJ*'s effect concretization rules.

3. (DEFINE).

$$\frac{\begin{array}{c} \Pi, A \vdash e_1 \rightsquigarrow e_1' : (t_1, \sigma_1, A_1) \\ \Pi; x : t, A_1; x = e_1' \vdash e_2 \rightsquigarrow e_2' : (t_2, \sigma_2, A_2) \\ t_1 <: t \end{array}}{\Pi, A \vdash t\ x = e_1; e_2 \rightsquigarrow t\ x = e_1'; e_2' : (t_2, \sigma_1 \cup \sigma_2, A_2)} \qquad \frac{\begin{array}{c} (\mu,A) \vdash e_1' : (\sigma_1', A1) \\ (\mu, A1; x = e_1') \vdash e_2' : (\sigma_2', A2) \end{array}}{(\mu,A) \vdash t\ var = e_1'; e_2' : (\sigma_1' \cup \sigma_2', A2)}$$

By IH, $\sigma_1' \subseteq \sigma_1$ and $\sigma_2' \subseteq \sigma_2$. Therefore $(\sigma' = \sigma_1' \cup \sigma_2') \subseteq (\sigma_1 \cup \sigma_2 = \sigma)$. ∎

Theorem 2: Dynamic effects refine concretized effects. [9] *For two configurations $\Sigma = \langle e, \mu \rangle$ and $\Sigma' = \langle e', \mu' \rangle$, if Σ transitions to Σ' producing runtime effect η, i.e. $\Sigma \overset{\eta}{\hookrightarrow} \Sigma'$, if the store μ is well-formed, i.e. $\mu \vdash \diamond$, and concretized effects of e is σ_c, i.e. $(\mu, A) \vdash e : (\sigma_c, A')$, then there is a concretized effect σ_c' such that:*

(a) $(\mu', A_1) \vdash e' : (\sigma_c', A_1')$ and $\sigma_c' \subseteq \sigma_c$;

[9] The theorem stated in §3 is the simplified version of the theorem presented here.

(b) $\eta \propto \sigma_c$

We first state few lemmas which are used in the proof of the theorem.

Lemma 1. [Store preservation] *Let the initial configuration of a program with a main expression e be* $\Sigma_\star = \langle e, \bullet \rangle$. *If* $\langle e, \bullet \rangle \overset{\overline{\eta}}{\hookrightarrow}{}^{*} \langle e', \mu' \rangle$, *then* $\mu' \vdash \diamond$.

Proof: The proof is by cases on the reduction step. In each case we show that $\mu \vdash \diamond$ implies that $\mu' \vdash \diamond$.

1. The cases (Condition-True), (Condition-False), (Define), (Call), (Binary) and (Get), are trivial, because they do not change the store, i.e., $\mu' = \mu$.

For all the remaining cases, to see $\mu' \vdash \mathit{loc}$, consider the definition of *initE*. It returns the effects computed by the static type-and-effect system, while the effect judgment is more accurate (Figure 17), i.e., by observation if $((\overline{var:t}, \texttt{this} : c), \emptyset) \vdash e : (u, \sigma, A)$ and $(\mu, \emptyset) \vdash [\mathit{loc}/\texttt{this}]e : (\sigma', A)$, then $\sigma' \subseteq \sigma$, therefore $\mu' \vdash \mathit{loc}$. Therefore, it suffices to show all the objects o are well-formed, i.e., $\mu' \vdash o$.

2. (New). Here $e = \mathbb{E}[\mathit{new}\ c()]$, $e' = \mathbb{E}[\mathit{loc}]$, where $\mathit{loc} \notin \mathit{dom}(\mu)$, $\mu' = \mu \oplus \{\mathit{loc} \mapsto [c.\{f \mapsto \mathit{default}(f) \mid f \in \mathit{fields}(c)\}.\{m \mapsto \sigma \in \mathit{initE}(c)\}]\}$. The only change to the store μ is the new object o created: $[c.\{f \mapsto \mathit{default}(f) \mid f \in \mathit{fields}(c)\}.\{m \mapsto \sigma \in \mathit{initE}(c)\}]$. All the fields are initiated to the default values, i.e., $\{f \mapsto \mathit{default}(f) \mid f \in \mathit{fields}(c)\}$. By the definition of *initE* (§3.2), all the open effects are initiated to `null`. Therefore, $\mu' \vdash o$.
3. (Set). Here $e = \mathbb{E}[\mathit{loc}.f = v]$, $e' = \mathbb{E}[v]$, $\mu' = \mu \oplus (\mathit{loc} \mapsto o)$, and $o = [u.F \oplus (f \mapsto v).E]$, where $\mu(\mathit{loc}) = [u.F.E]$ and $\mathit{typeOf}(f) = (c,t)$ for some c and t. The field f is not an open field, and by the function *update*, it does not update any effect, and $\mu' \vdash o$.
4. (Set Open), Here $e = \mathbb{E}[\mathit{loc}.f = v]$, $e' = \mathbb{E}[v]$, where $\mu_0 = \mu \oplus (\mathit{loc} \mapsto [c.(F \oplus (f \mapsto v)).E])$, and $\mu' = \mathit{update}(\mu_0, \mathit{loc}, f, v)$. The proof is by observation/construction of the *update* 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).■

Lemma 2. [Stationary effect] *Let e be an expression, and* μ *and* μ' *two effect equivalent stores, i.e.* $\mu \cong \mu'$, *then the expression e has the same effects in the two stores* μ *and* μ'. *In other words if* $(\mu, A) \vdash e : (\sigma, A')$, *then* $(\mu', A) \vdash e : (\sigma, A')$.

Proof: The proof is by induction on the structure of the expression e.

1. Cases of (New), (Null), (Loc), (Number), (Binary) and (VAR) are trivial, since in these cases, $\sigma' = \sigma = \emptyset$.

For the remaining steps, the induction hypothesis (IH) says that the claim of the lemma holds for all sub-derivations of the derivation being considered.

2. The cases for (Condition), (Define), (Get) and (Set) follow directly from IH.

3. (IF).

$$\frac{(\mu,A) \vdash x : (\emptyset,A) \qquad (\mu,A) \vdash e_0 : (\sigma_0,A_0) \qquad (\mu,A) \vdash e_1 : (\sigma_1,A_1)}{(\mu,A) \vdash \mathtt{if}\ x\ \mathtt{then}\ e_0\ \mathtt{else}\ e_1 : (\sigma_0 \cup \sigma_1, A_0 \sqcap A_1)}$$

$$\frac{(\mu',A) \vdash x : (\emptyset,A) \qquad (\mu',A) \vdash e_0 : (\sigma'_0,A_0) \qquad (\mu',A) \vdash e_1 : (\sigma'_1,A_1)}{(\mu',A) \vdash \mathtt{if}\ x\ \mathtt{then}\ e_0\ \mathtt{else}\ e_1 : (\sigma'_0 \cup \sigma'_1, A_0 \sqcap A_1)}$$

By IH, $\sigma'_0 = \sigma_0$ and $\sigma'_1 = \sigma_1$. Therefore $(\sigma' = \sigma'_0 \cup \sigma'_1) = (\sigma = \sigma_0 \cup \sigma_1)$.

4. (DEFINE)

$$\frac{(\mu,A) \vdash e_1 : (\sigma_1,A1) \qquad (\mu,A1;x=e_1) \vdash e_2 : (\sigma_2,A2)}{(\mu,A) \vdash t\ var = e_1;e_2 : (\sigma_1 \cup \sigma_2, A2)}$$

$$\frac{(\mu',A) \vdash e_1 : (\sigma'_1,A1) \qquad (\mu',A1;x=e_1) \vdash e_2 : (\sigma'_2,A2)}{(\mu',A) \vdash t\ var = e_1;e_2 : (\sigma'_1 \cup \sigma'_2, A2)}$$

By IH, $\sigma'_1 = \sigma_1$ and $\sigma'_2 = \sigma_2$. Therefore $(\sigma' = \sigma'_1 \cup \sigma'_2) = (\sigma = \sigma_1 \cup \sigma_2)$.

5. (GET)

$$(\mu,A) \vdash loc.f : (\mathbf{rd}(f),A) \qquad\qquad (\mu',A) \vdash loc.f : (\mathbf{rd}(f),A)$$

Therefore $\sigma' = \sigma = \mathbf{rd}(f)$.

6. (SET)

$$\frac{(\mu,A) \vdash x : (\emptyset,A) \qquad typeOf(f) = (c,t)}{(\mu,A) \vdash loc.f = x : (\mathbf{wr}(f), A \backslash f \cup \{x = loc.f\})}$$

$$\frac{(\mu',A) \vdash x : (\emptyset,A) \qquad typeOf(f) = (c,t)}{(\mu',A) \vdash loc.f = x : (\mathbf{wr}(f), A \backslash f \cup \{x = loc.f\})}$$

Thus $\sigma' = \sigma = \mathbf{wr}(f)$.

7. (SET-OPEN)

$$\frac{typeOf(f) = (c, @open\ t)}{(\mu,A) \vdash e_0.f = e_1 : (\top,\emptyset)} \qquad\qquad \frac{typeOf(f) = (c, @open\ t)}{(\mu',A) \vdash e_0.f = e_1 : (\top,\emptyset)}$$

Thus $\sigma' = \sigma = \top$.

8. (CALL-OPEN)

$$\frac{\begin{array}{c} A \vdash x = loc.f \\ \mathbf{open}(f\ m\ \sigma'_0) = concretize(\mu, loc, \mathbf{open}(f\ m\ \gamma)) \qquad typeOf(f) = (d, @open\ c_0) \end{array}}{(\mu,A) \vdash x.m(\overline{x}) : (\sigma'_0,A)}$$

$$\frac{\begin{array}{c} A \vdash x = loc.f \\ \mathbf{open}(f\ m\ \sigma'_1) = concretize(\mu', loc, \mathbf{open}(f\ m\ \gamma)) \qquad typeOf(f) = (d, @open\ c_0) \end{array}}{(\mu',A) \vdash x.m(\overline{x}) : (\sigma'_1,A)}$$

By IH, $(\sigma' = \sigma'_0) = (\sigma = \sigma'_1)$.

9. (CALL-LOC)

$$\frac{\mu(loc) = [c.F.E] \qquad E(m) = \sigma_0}{(\mu,A) \vdash loc.m(\overline{x}) : (\sigma_0,A)} \qquad \frac{\mu'(loc) = [c.F.E] \qquad E(m) = \sigma_0'}{(\mu',A) \vdash loc.m(\overline{x}) : (\sigma_0',A)}$$

Since $\mu \cong \mu'$, the effect maps E are the same and $\sigma_0 = \sigma_0'$. Thus $(\sigma' = \sigma_0') = (\sigma = \sigma_0)$.

10. (CALL)

$$(\mu,A) \vdash x.m(\overline{x}) : (\top,\emptyset) \qquad (\mu',A) \vdash x.m(\overline{x}) : (\top,\emptyset)$$

Thus $\sigma' = \sigma = \top$. ∎

Lemma 3. [Replacement with subeffect]

If $\mu \vdash \diamond$, $\Sigma \overset{\eta}{\hookrightarrow} \Sigma'$, $\Sigma = \langle \mathbb{E}[e],\mu \rangle$, $\Sigma' = \langle \mathbb{E}[e'],\mu' \rangle$, $(\mu,A) \vdash \mathbb{E}[e] : (\sigma,A')$, $(\mu,A) \vdash e : (\sigma_0,A_0')$, $(\mu,A) \vdash e' : (\sigma_1,A_0')$, $\mu \cong \mu'$, and $\sigma_1 \subseteq \sigma_0$, then $(\mu,A) \vdash \mathbb{E}[e'] : (\sigma',A') \wedge \sigma' \subseteq \sigma$.

For two expression e and e', in the configurations $\Sigma = \langle \mathbb{E}[e],\mu \rangle$ and $\Sigma' = \langle \mathbb{E}[e'],\mu' \rangle$ such that $\Sigma \overset{\eta}{\hookrightarrow} \Sigma'$, if the store μ is well-formed, i.e. $\mu \vdash \diamond$, and the expression e has the effects σ_0, i.e. $(\mu,A) \vdash e : (\sigma_0,A_0')$ and $\mathbb{E}[e]$ has the concrete effects σ, i.e. $(\mu,A) \vdash \mathbb{E}[e] : (\sigma,A')$, and $\mu \cong \mu'$, and $\sigma_1 \subseteq \sigma_0$, then $(\mu,A) \vdash \mathbb{E}[e'] : (\sigma',A') \wedge \sigma' \subseteq \sigma$.

Lemma 3 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 $\mathbb{E}[e]$.

Proof: The proof is by induction on the size of the evaluation context $\mathbb{E}$. The size of the $\mathbb{E}$ is the number of recursive applications of the syntactic rules necessary to create $\mathbb{E}$.

1. For the base case $\mathbb{E} = -$, the size of $\mathbb{E}$ is zero , and $(\sigma' = \sigma_1) \subseteq (\sigma = \sigma_0)$.

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 the size one. The induction hypothesis (IH) says that the lemma holds for all evaluation contexts, which their sizes are 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 $(\mu,A) \vdash \mathbb{E}_2[e] : (\sigma,A')$ implies that $(\mu',A) \vdash \mathbb{E}_2[e'] : (\sigma',A')$, for some $\sigma' \subseteq \sigma$, and thus the claim holds by the IH.

2. For (E-DEFINE), (E-GET) and (E-SET) the proof follows directly from the IH.
3. (E-SET-OPEN) holds because in this case $\sigma = \top$. ∎

Lemma 4. [Substitution effect] *If $(\mu,A) \vdash e : (\sigma,A')$, then there is some σ', such that $(\mu,A) \vdash \overline{[v/var]}e : (\sigma',A')$, for all values v in $\overline{v}$ and free variables var in $\overline{var}$, and $\sigma' \subseteq \sigma$.*

Proof: The proof is by structural induction on the derivation of $(\mu,A) \vdash e : (\sigma,A')$ and by cases, based on the last step in that derivation.

1. Proof for (E-NEW), (E-NULL), (E-LOC) is trivial, since e has no variables, $\sigma' = \sigma = \emptyset$.
2. For (E-VAR) case, $(\mu,A) \vdash v : (\emptyset,A)$ and $(\mu,A) \vdash var = (\emptyset,A)$.

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.

3. For (E-CONDITION), (E-BINARY), (E-GET) and (E-DEFINE) the proof follows directly from the IH.
4. The case for (E-SET-OPEN) and (E-CALL) hold because in these cases $\top \in \sigma$. The effect of e is $\top$ and every effect refines $\top$.
5. (E-CALL-OPEN)

$$\frac{\begin{array}{c} A \vdash x = loc.f \\ \mathbf{open}(f\ m\ \sigma) = concretize(\mu, loc, \mathbf{open}(f\ m\ \gamma)) \qquad typeOf(f) = (d, @open\ c_0) \end{array}}{(\mu,A) \vdash x.m(\overline{x}) : (\sigma,A)}$$

Let $e'_i = [\overline{v/var}]x_i$ for $i \in \{1..n\}$, $[\overline{v/var}]e = x.m(\overline{e'})$. They result in the same effect by (E-CALL-OPEN).

6. (E-CALL-LOC)

$$\frac{\mu(loc) = [c.F.E] \qquad E(m) = \sigma}{(\mu,A) \vdash loc.m(\overline{x}) : (\sigma,A)}$$

Let $e'_i = [\overline{v/var}]x_i$ for $i \in \{1..n\}$, then $[\overline{v/var}]e = loc.m(\overline{v})$. Clearly $(\mu,A) \vdash [\overline{v/var}]e : (\sigma,A)$.

7. (E-SET)

$$\frac{typeOf(f) = (c,t)}{(\mu,A) \vdash loc.f = x : (\mathbf{wr}(f), A \backslash f \cup \{x = loc.f\})}$$

Now $[\overline{v}/\overline{var}]e = (loc.f = [v/x]x)$. $(\mu,A) \vdash [v/x]x : (\emptyset,A)$. By the definition of *typeOf*, the result of $typeOf(f)$ remains unchanged, i.e. $typeOf(f) = (c,t)$. ∎

Lemma 5. [Subexpression effect containment] *If* $(\mu,A) \vdash e : (\sigma,A_0)$ *and* $(\mu,A) \vdash \mathbb{E}[e] : (\sigma',A'_0)$*, 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.

C.2 Proof of Theorem 2

Using Lemma 2 and Lemma 3. To prove Theorem 2, in each reduction case, let $e = \mathbb{E}[e_0]$, $e' = \mathbb{E}[e_1]$, $(\mu,A) \vdash e_0 : (\sigma_0,A')$ and $(\mu',A) \vdash e_1 : (\sigma_1,A')$. Given that (a) $\mu \cong \mu'$, by Lemma 3 and Lemma 2, 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*); the cases, in the second category, access the store; and the other cases are listed right below. Here the rule leaves no dynamic trace, and (c) holds.

- (NEW) Here $e = \mathbb{E}[\mathit{new}\ c()]$, $e' = \mathbb{E}[\mathit{loc}]$, where $\mathit{loc} \notin \mathit{dom}(\mu)$, $\mu' = \mu \oplus \{\mathit{loc} \mapsto [c.\{f \mapsto \mathit{default}(f) \mid f \in \mathit{fields}(c)\}.\{m \mapsto \sigma \in \mathit{initE}(c)\}]\}$. Because this rule does not change any object, $\mu \cong \mu'$. Also $(\mu,A) \vdash \mathtt{new}\, c() : (\emptyset,A)$ and $(\mu,A) \vdash \mathit{loc} : (\emptyset,A)$, and (b) holds.
- (BINARY) Here $e = \mathbb{E}[v_1 \circ v_2]$, $e' = \mathbb{E}[v]$, where $v = v_1 \circ v_2$, $\mu' = \mu$. It is trivial to see that (b) holds.

Using Lemma 4. We now present the case for method call and local declaration.

- (CALL) Here $e{=}\mathbb{E}[\mathit{loc}.m(\bar{v})]$, $(u', t_m, m(\overline{t\ \mathit{var}})\{e_2\}, \sigma_m){=}\mathit{findMeth}(u,m)$, $e'{=}\mathbb{E}[e_1]$, $e_1{=}[\mathit{loc}/\mathtt{this}, \overline{v/\mathit{var}}]e_2$, $\mu(\mathit{loc}){=}[u.F.E]$. Let $(\mu,A){\vdash}\mathit{loc}.m(\bar{v}){:}(\sigma_0,A)$, i.e., $E(m){=}\sigma_0$. Let $e_3{=}[\mathit{loc}/\mathtt{this}]e_2$, $(\mu,A){\vdash}e_3{:}(\sigma_3,A_3)$ and $(\mu,A){\vdash}e_1{:}(\sigma_1,A_1)$. By Lemma 4, $\sigma_1 \subseteq \sigma_3$. By $\mu \vdash \diamond$, Definition 8 and Definition 9, $\sigma_3 \subseteq \sigma_0$, thus $\sigma_1 \subseteq \sigma_0$.
- (DEFINE) Here $e = \mathbb{E}[t\ \ \mathit{var} = v; e_1]$, and $e' = \mathbb{E}[e'_1]$, where $e'_1 = [v/\mathit{var}]e_1$. Let $(\mu,A) \vdash e_1 : (\sigma_0,A_0)$, by (E-DEFINE), $(\mu,A) \vdash t\ \ \mathit{var} = v; e_1 : (\sigma_0,A_0)$. $(\mu,A) \vdash [v/\mathit{var}]e_1 : (\sigma_1,A_0)$, for some $\sigma_1 \subseteq \sigma_0$, by Lemma 4.

Using Lemma 5. We prove cases for field accesses:

- (GET) Here $e = \mathbb{E}[\mathit{loc}.f]$, $e' = \mathbb{E}[v]$, where $\mu(\mathit{loc}) = [u.F.E]$, $F(f) = v$, $\mu' = \mu$ and $\mu \cong \mu'$. Because $(\mu,A) \vdash \mathit{loc}.f : (\mathbf{rd}(f),A)$, and $(\mu',A) \vdash v : (\emptyset,A)$, (b) holds. Finally, $\eta = (\mathbf{read}(\mathit{loc},f))$, and $\eta \propto \mathbf{rd}(f) \subseteq \sigma$, by Lemma 5.
- (SET) Here $e = \mathbb{E}[\mathit{loc}.f = v]$, $e' = \mathbb{E}[v]$, $\mu' {=} \mu{\oplus}(\mathit{loc}{\mapsto}o)$, and $o = [u.F{\oplus}(f{\mapsto}v).E]$, where $\mu(\mathit{loc}){=}[u.F.E]$ and $\mathit{typeOf}(f) = (c,t)$ for some t and c. The field is not an open field, and by the function *update*, it does not update any effect, and $\mu \cong \mu'$. To see $(\mu,A) \vdash \mathbb{E}[v] : (\sigma',A')$ and $\sigma' \subseteq \sigma$, we have $(\mu,A) \vdash \mathit{loc}.f = v : (\mathbf{wr}(f),A)$, and $(\mu,A){\vdash}v{:}(\emptyset,A)$, thus $\sigma' \subseteq \sigma$. Finally, $\eta = (\mathbf{write}(\mathit{loc},f))$, and $\eta \propto \mathbf{wr}(f) \subseteq \sigma$, by Lemma 5.
- (SET OPEN) Here $e{=}\mathbb{E}[\mathit{loc}.f = v]$, $e' {=}\mathbb{E}[v]$, where $\mu_0 = \mu \oplus (\mathit{loc} \mapsto [c.(F \oplus (f \mapsto v)).E])$, and $\mu' = \mathit{update}(\mu_0, \mathit{loc}, f, v)$. The effect of e is $\top$ and every effect refines $\top$. ∎