Type-based Enforcement of Infinitary Trace Properties for Java

Serdar Erbatur
serdar.erbatur@utdallas.edu
University of Texas at Dallas
Richardson, USA

Ulrich Schöpp
schoepp@fortiss.org
fortiss GmbH
Munich, Germany

Chuangjie Xu
xu@fortiss.org
fortiss GmbH
Munich, Germany

ABSTRACT

A common approach to improve software quality is to use programming guidelines to avoid common kinds of errors. In this paper, we consider the problem of enforcing guidelines for Featherweight Java (FJ). We formalize guidelines as sets of finite or infinite execution traces and develop a region-based type and effect system for FJ that can enforce such guidelines. We build on the work by Erbatur, Hofmann and Zălinescu, who presented a type system for verifying the finite event traces of terminating FJ programs. We refine this type system, separating region typing from FJ typing, and use ideas of Hofmann and Chen to extend it to capture also *infinite* traces produced by *non-terminating* programs. Our type and effect system can express properties of both finite and infinite traces and can compute information about the possible infinite traces of FJ programs. Specifically, the set of infinite traces of a method is constructed as the greatest fixed point of the operator which calculates the possible traces of method bodies. Our type inference algorithm is realized by working with the *finitary abstraction* of the system based on Büchi automata.

1 INTRODUCTION

To improve the quality of software, it is desirable to support software development with automatic static analysis tools. While the full verification of large-scale software is out of scope for fully automated tools, it is nevertheless useful to use such tools to identify potential bugs. Current static analysis tools such as Facebook's infer [7] can identify a wide range of possible bugs, such as memory errors, issues with thread safety or resource management.

In addition to identifying general bugs, it is also useful to identify logical errors in the software being developed. Many software development processes enforce a set of programming guidelines that are designed to rule out particular kinds of errors. Secure programming guidelines, for example, may stipulate that all inputs from external sources are to be sanitized before further processing, that all password changes are to be recorded in a log, *etc.* While such guidelines typically capture fairly simple program properties, they help programmers to avoid making typical errors. However, guidelines need to be enforced to be useful.

In this paper, we consider the automatic static enforcement of programming guidelines for Java. At scale, the fully automatic verification of Java code is out of scope of current methods. We therefore concentrate on verifying properties of event traces of programs. Due to the relative simplicity of typical guidelines, this should already be useful for their enforcement.

Let us illustrate the idea using an example. Consider a server that accepts queries for various actions on private data from a user and executes them if the user is authorized to do so. At the end, it logs that sensitive data was accessed.

```
void serve() {
  while(hasQuery()) {
    String query = nextQuery();
    boolean authorized = verifyAuthorization();
        // emits authcheck;
    if (authorized) {
      readSensitiveData(); // emits access;
    }
  }
  LogAccess(); // emits log;
}
```

Imagine that functions of interest emit events when they are executed. We can then think of a guideline as a set of allowed event traces. In the example, we have indicated the emitted events in comments. An example event trace of `serve` would be *authcheck access authcheck log*. It is emitted in a run where a user first inputs an authorized request, then an unauthorized one, and then closes the connection. Guidelines, such as "each access to sensitive data occurs after authorization" or "each access to sensitive data is logged", would then be formalized as "*access* must be preceded by *authcheck*" and "each *access* is eventually followed by *log*" respectively.

The example is chosen to illustrate that we want to verify properties of traces not just for terminating runs, but also for non-terminating ones. The example has intended non-terminating behavior, where the `while`-loop is repeated indefinitely. The corresponding traces of such non-terminating runs are infinite repetitions of *authcheck* or *authcheck access*. This means we want to verify that the infinite words in $\{authcheck, authcheck\ access\}^{\omega}$ are allowed by the guideline. In addition to such safety properties ("nothing bad happens"), one would also like to verify liveness properties ("something good will happen eventually"). An example for a liveness property would be that any access must be logged, *i.e.* that *access* must be followed by *log*. The above example code would not have this property, since there are infinite traces where no access is logged.

There are a number of options of using automatic verification techniques to verify programming guidelines that have been formalized as sets of allowed traces. The most obvious option would be to use software model checkers. This approach should certainly be expressive enough to handle most guidelines, but it may be limited by scalability problems. One would like to balance expressiveness and scalability. An alternative is to investigate type systems and abstract interpretation. The idea is to develop type systems that are strong enough to express and automatically verify programming guidelines.

In this paper, we define a type system for Featherweight Java (FJ) [15] that allows the automatic analysis of traces of both terminating and non-terminating runs. In previous work, Erbatur et al. [5] have developed a similar type system, which was however limited to finite runs. For infinite runs, Hofmann and Chen [12, 13] have introduced an abstract interpretation approach for the type-based analysis of infinite traces. It is based on Büchi automata, shares their expressiveness and can therefore be used for both safety and liveness properties. However, it was defined for a simple procedural language. While [12, 13] contain a sketch of how to apply their methods in a type system like [5], this development remains at the level of a sketch that needs to be fully developed in further work[1].

Here we develop a type system for FJ on the basis of ideas from both approaches. In contrast to previous work like [5, 12], our type system is not a refinement of the FJ type system, but a separate flow type system in the spirit of Microsoft's TypeScript [19] and Facebook's Flow [6]. The resulting type system is simpler, more precise, and more flexible than a refinement of FJ's type system.

Our development is based on three main ideas: effect types, regions and abstraction.

Effect Types. Type systems can be extended to statically approximate sets of possible event traces by means of effect types. The idea is to extend the type with effect annotations that give information about the possible traces. This approach was followed in [5], where method types have the form `void serve()` $\&\, A$, with an *effect annotation* "$\&\, A$" that gives static information about the possible traces of terminating runs of `serve`. In essence, A represents a set of traces that includes all possible traces of terminating runs. In our example, A could be $\{authcheck, authcheck\ access\}^* \cdot \{log\}$, that is, the set of traces that are a finite repetition of *authcheck* or *authcheck access*, followed by *log*.

Regions. To obtain a useful type system for the analysis of traces in Java, the type system needs to perform some kind of flow analysis. Consider, for example, the Java interface `Runnable` with a single method `void run()`. If we try to annotate this method with an effect annotation for the set of its possible traces, then, without further knowledge, we need to include the traces of all classes that implement `Runnable`. The effect typing of a call like `x.run()` would not be very useful, since it would include the effects of all implementations of `run()`. What is needed is a way to narrow down the possible objects that `x` can point to. This is why [5] integrates both regions and effects in their type system. We also use regions, but diverge from [5] in that we capture them in a separate type system rather than integrating them in the Java type system. Our type system is simpler, more precise and can accommodate richer kinds of regions.

Abstraction. In an implementation of automatic type checking it is of course not possible to store infinite languages like A above directly. Some computable abstraction is needed, a natural choice being finite automata. This choice has been taken for finite words in [5] and for potentially infinite words in [12, 13]. In particular, Hofmann and Chen [12] define an abstraction of Büchi automata for the finite representation of infinite words. Our type system works with the same abstractions. However, the particular choice of abstraction is not important for it. We introduce a notion of Büchi abstraction and establish soundness of the type system for any instance of it. The automata-theoretic constructions of [5, 12] provide an example.

In summary, the contributions of this paper include:

- We introduce a new effect type system that is simpler than those of [5, 12] but allows more precise analysis. It can analyze both terminating and non-terminating programs.
- We give a simple characterization of the infinitary effect analysis as a single greatest fixed point, which allows us to prove its soundness simply by induction on the approximation of the greatest fixed point.
- We realize the type inference by working with the finitary abstraction of the type system following [12]. To have a generic formulation, we introduce a notion of Büchi abstraction that abstracts from the automata-theoretic constructions of [12].

Moreover, we have a prototype implementation of type inference based on Soot [22].

2 EVENTS AND TRACES

Let Σ be a finite alphabet. We use Σ^* to denote the set of finite words over Σ, and $\Sigma^{\leq\omega}$ the set of finite and infinite words. We write uv for the concatenation of the finite word u with the word v which can finite or infinite. We now extend concatenation to and recall some other operations on languages: Let $U, U' \subseteq \Sigma^*$ and $V \subseteq \Sigma^{\leq\omega}$. We write

- $U \cdot U' = \{uu' \mid u \in U, u' \in U'\} \subseteq \Sigma^*$,
- $U \cdot V = \{uv \mid u \in U, v \in V\} \subseteq \Sigma^{\leq\omega}$,
- $U^* = \{u_1 u_2 \dots u_n \mid u_i \in U \text{ for all } i \in \{1, \dots, n\}\} \subseteq \Sigma^*$,
- $U^+ = U^* \setminus \{\varepsilon\} \subseteq \Sigma^*$, where ε is the empty word, and
- $U^\omega = \{u_1 u_2 \dots u_i \dots \mid u_i \in U \text{ for all } i \in \mathbb{N}\} \subseteq \Sigma^{\leq\omega}$.

We often write a^* rather than $\{a\}^*$ and similarly a^+, a^ω.

For the sake of simplicity, we assume programs have special commands issuing *events* from Σ. Then a terminating execution of a program will generate a finite *trace*, that is, a word from Σ^*, whereas a non-terminating execution will generate a finite or infinite trace, that is, a word from $\Sigma^{\leq\omega}$. Our objective is to capture all possible traces of a given program via a type and effect system, so that we can verify whether they are allowed by a given *guideline*, that is, a language of acceptable traces.

3 FEATHERWEIGHT JAVA

We work with a variant of Featherweight Java (FJ) that extends the FJ calculus of [15] with filed updates. Following the formulation of [5], we add primitive if- and let-expressions for convenience and omit constructors for simplicity. Moreover, we add a primitive operation `emit`(a) that generates the event a from a finite alphabet Σ.

The syntax of the language uses four kinds of names:

$$\begin{array}{rlrl} \text{variables:} & x, y \in Var, & \text{classes:} & C, D \in Cls, \\ \text{fields:} & f \in Fld, & \text{methods:} & m \in Mtd. \end{array}$$

[1]Hofmann and Chen [12] do not yet formulate a soundness theorem and the sketched notion of well-typedness is not fully precise (the formulation is unclear w.r.t. the quantification of η). In addition, some choices in the sketched type system seem unnecessarily limiting: For example, any method that calls another method from the same class would be analyzed as having a non-terminating run even when there is no recursion at all.

Program expressions are defined as follows:

$$\begin{aligned}\mathit{Expr} \ni e ::=\ & x \mid \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 \mid \mathtt{if}\ x = y\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 \\ & \mid \mathtt{null} \mid \mathtt{new}^{\ell}\ C \mid (C)\, e \mid \mathtt{emit}(a) \\ & \mid x^C.m(\bar{y}) \mid x^C.f \mid x^C.f := y\end{aligned}$$

The expression $\mathtt{new}^{\ell}\ C$ is annotated with a label ℓ. We use labels only to distinguish different occurrences of new in a program; since our type system will track where objects were created. In a few expressions we have added type annotations and write x^C for a variable of type C. They will be needed when looking up in the class table (see rules GET, SET and CALL in Figure 1). We sometimes omit annotations when they are not needed.

We assume three distinguished formal elements: $\mathtt{this} \in \mathit{Var}$, $\mathtt{Object} \in \mathit{Cls}$ and $\mathtt{NullType} \in \mathit{Cls}$. The NullType class plays the role of the type of null from the Java language specification [10, §4]. It may not be used in programs, *i.e.* we require $C \neq \mathtt{NullType}$ in the expressions $\mathtt{new}^{\ell}\ C$ and $(C)\, e$. When x is not a free variable of e_2, we may write $e_1; e_2$ rather than $\mathtt{let}\ x = e_1\ \mathtt{in}\ e_2$.

An FJ program $(\prec, \mathit{fields}, \mathit{methods}, \mathit{mtable})$ consists of

- a subtyping relation $\prec\ \in \mathcal{P}^{\mathrm{fin}}(\mathit{Cls}\times\mathit{Cls})$ with $C \prec D$ meaning that C is an immediate subclass of D,
- a field list $\mathit{fields} : \mathit{Cls} \to \mathcal{P}^{\mathrm{fin}}(\mathit{Fld})$ mapping a class to its fields,
- a method list $\mathit{methods} : \mathit{Cls} \to \mathcal{P}^{\mathrm{fin}}(\mathit{Mtd})$ mapping a class to its methods, and
- a method table $\mathit{mtable} : \mathit{Cls} \times \mathit{Mtd} \rightharpoonup \mathit{Var}^* \times \mathit{Expr}$ mapping a method to its formal parameters and its body.

All components are required to be *well-formed* (see *e.g.* [5, Section 3] for details). Let $\preceq$ be the reflexive and transitive closure of $\prec$. We have $C \preceq \mathtt{Object}$ and $\mathtt{NullType} \preceq C$ for all $C \in \mathit{Cls}$.

Example 3.1. As a running example, we consider the following Java code, in which ℓ_1, ℓ_2 and ℓ_3 are arbitrary fresh labels.

```
class Node {
 Node next;
 Node last() {
  emit(a);
  if (next == null) {
   return this;
  } else {
   return next.last();
  }
 }
}
```

```
class Test {
 Node linear() {
  Node x = new^ℓ1 Node();
  Node y = new^ℓ2 Node();
  y.next = x;
  return y.last();
 }
 Node cyclic() {
  Node z = new^ℓ3 Node();
  z.next = z;
  return z.last();
 }
}
```

In Featherweight Java, the class Node is represented by

$$\mathit{fields}(\mathsf{Node}) = \{\mathsf{next}\}$$
$$\mathit{methods}(\mathsf{Node}) = \{\mathsf{last}\}$$
$$\mathit{mtable}(\mathsf{Node}, \mathsf{last}) = ((), e_{\mathtt{last}})$$

where the expression $e_{\mathtt{last}}$ is defined as follows:

$$\begin{aligned} e_{\mathtt{last}} := \ & \mathtt{let}\ \_ = \mathtt{emit}(a)\ \mathtt{in} \\ & \mathtt{let}\ x = \mathtt{this}^{\mathsf{Node}}.\mathtt{next}\ \mathtt{in} \\ & \mathtt{let}\ y = \mathtt{null}\ \mathtt{in} \\ & \mathtt{if}\ x = y\ \mathtt{then\ this} \\ & \mathtt{else\ let}\ z = \mathtt{this}^{\mathsf{Node}}.\mathtt{next}\ \mathtt{in}\ z^{\mathsf{Node}}.\mathtt{last}() \end{aligned}$$ □

In the standard FJ type system [15], types are simply classes. In the rest of this paper, we consider only well-typed FJ programs. The type information of an FJ program can be modeled by a class table (F_0, M_0), where $F_0 : \mathit{Cls} \times \mathit{Fld} \rightharpoonup \mathit{Cls}$ is a *field typing* that assigns to each class C and each field $f \in \mathit{fields}(C)$ the class of f, and $M_0 : \mathit{Cls} \times \mathit{Mtd} \rightharpoonup \mathit{Cls}^* \times \mathit{Cls}$ a *method typing* that specifies for each class C, method $m \in \mathit{methods}(C)$ and classes of the argument variables the class of the result value of m.

The operational semantics of FJ uses a notion of state (s, h) that consists of a *store* $s : \mathit{Var} \rightharpoonup \mathit{Val}$ mapping variables to values and a *heap* $h : \mathit{Loc} \rightharpoonup \mathit{Obj}$ mapping locations to objects. The only kinds of values are object locations and *null*. An object $(C, G, \ell) \in \mathit{Obj}$ contains a class identifier $C \in \mathit{Cls}$, a valuation $G : \mathit{Fld} \rightharpoonup \mathit{Val}$ of its fields and a label $\ell \in \mathit{Pos}$ indicating where it was created (*i.e.*, the object was created by a $\mathtt{new}^{\ell}\ C$ expression).

locations:	$l \in \mathit{Loc}$
values:	$v \in \mathit{Val} = \mathit{Loc} \uplus \{\mathit{null}\}$
stores:	$s \in \mathit{Var} \rightharpoonup \mathit{Val}$
heaps:	$h \in \mathit{Loc} \rightharpoonup \mathit{Obj}$
objects:	$(C, G, \ell) \in \mathit{Obj} = \mathit{Cls} \times (\mathit{Fld} \rightharpoonup \mathit{Val}) \times \mathit{Pos}$

The operational semantics of terminating evaluation is given as a big-step relation $(s, h) \vdash e \Downarrow v, h' \,\&\, w$. It expresses that, in state (s, h), the expression e evaluates to the value v with the heap updated to h', generating the event trace $w \in \Sigma^*$. The inference rules are given in Appendix A. The operational semantics of divergence $(s, h) \vdash e \Uparrow \,\&\, w$ is defined coinductively from the rules in Appendix A where double horizontal lines are used.

4 REGION TYPES AND EFFECTS

We introduce a type system for region and effect analysis for FJ. In contrast to previous work [5, 12], we do not refine FJ type system, but instead build an independent region typing system for FJ. Region type information is complementary to FJ type information and can be captured without repeating the FJ type system. This also makes sense practically: we would not want to modify the Java compiler but develop an additional analysis system.

For the sake of understandability, in this section we present the system with arbitrary languages over Σ as effect annotations. In Section 5 we shall make it algorithmic by moving to a finitary abstraction of languages.

4.1 Region Types

Our type system captures region information. A *region* represents provenance information about a value, such as where in the program an object was created or what its actual class is. The intention is that the set of values is subdivided in regions according to various criteria.

In this paper, we use the following definition of regions:

$$\mathit{Reg} \ni r, s ::= \mathtt{Null} \mid \mathtt{CreatedAt}(\ell) \mid \mathtt{Unknown}$$

These regions have the following meaning. The region Null contains only the value *null*. The region $\mathtt{CreatedAt}(\ell)$ contains all references to objects that were created by an expression of the form $\mathtt{new}^{\ell}\ C$. This region allows us to track where in the program an object originates. The region Unknown is for references to objects of unknown origin, *e.g.* from library code. A formal interpretation of regions is given in Section 4.5.

It is possible to use a richer definition of regions to capture other properties of interest, such as taintedness, or to give regions more structure. For example, one may allow unions or intersections of regions. We plan to do that in further work. Here we use a simple representative definition of regions in order to focus on the new type system itself and on infinitary traces.

Regions provide information about the actual classes of objects. We write $Cls(r)$ for the set of possible classes of an object in region r, which is defined as follows: $Cls(\mathtt{Null}) = \emptyset$, $Cls(\mathtt{CreatedAt}(\ell)) = \{C \mid \text{program contains } \mathtt{new}^{\ell}\ C\}$, and $Cls(\mathtt{Unknown}) = Cls$. Note that that $Cls(r)$ is *not* required to be closed under superclasses. Indeed, $Cls(\mathtt{CreatedAt}(\ell))$ will be just a singleton if the label ℓ was used only once.

Regions also provide information about the identity of objects. Objects in disjoint regions cannot be identical. The region Null and all regions of the form $\mathtt{CreatedAt}(\ell)$ are all pairwise disjoint. However, Unknown overlaps with all other regions. We write $disjoint(r, s)$ to indicate that two regions are disjoint.

4.2 Region and Effect Expressions

Before diving into the details of the type system, we explain the region and effect expressions that are used in the type system. The typing judgement will have the following form:

$$\Gamma \vdash e : T \ \text{ calls } \ S.$$

The context Γ simply maps program variables to regions. More interesting are T and S on the right-hand side. They both are expressions that contain region and effect information about the computation of e. In this section we explain the meaning of these expressions.

First, T is an expression of the form $r_1 \,\&\, U_1 \mid \cdots \mid r_n \,\&\, U_n$. It lists the possible options for the terminating computations of e. The option $r_i \,\&\, U_i$ means that the result value is in region r_i and the computation trace is in $U_i \subseteq \Sigma^*$. Thus, T expresses that whenever e evaluates to a value, there will be some i such that the result is in region r_i and the computation trace is in U_i.

For example, the following expression

$$\begin{array}{l}\mathtt{let}\ x = \mathtt{if}\ cond\ \mathtt{then}\ (\mathtt{emit}(a);\ \mathtt{new}^{\ell_1}\ C)\ \mathtt{else}\ (\mathtt{new}^{\ell_2}\ D)\\ \mathtt{in}\ \mathtt{emit}(b);\ x\end{array}$$

will have type $\mathtt{CreatedAt}(\ell_1) \,\&\, \{ab\} \mid \mathtt{CreatedAt}(\ell_2) \,\&\, \{b\}$. It either evaluates to a reference pointing to an object that was created by a new with label ℓ_1 or to one that was created by a new with label ℓ_2. In the first case, the trace of the computation is ab, and in the latter case it is just b. The expression may also be given the less precise type $\mathtt{Unknown} \,\&\, \{ab, b\}$, which just expresses that it produces some value and either has effect ab or b while doing so.

Second, S is a *call expression* of the form $U_1 \cdot \delta_1 \cup \cdots \cup U_k \cdot \delta_k$. It contains information about the calls that e may make. In the expression, each δ_i is a method signature, which is given by a class name, a method name and region information. We specify signatures precisely below, for a first explanation of S the details are not important. The expression S gives us information about which method calls e can make and what the traces leading up to such calls can be. It expresses that for any call that e makes, there exists an i such the call goes to a method with signature δ_i and that the event trace leading to the call is in U_i.

Consider, for example, the following FJ-expression e:

$$\mathtt{emit}(a);\ x.\mathtt{f}();\ \mathtt{emit}(b);\ x.\mathtt{g}()\ \ .$$

If we assume that f and g have no effects on their own, then the call expression S for e would be $\{a\} \cdot \delta_{\mathtt{f}} \cup \{ab\} \cdot \delta_{\mathtt{g}}$, where $\delta_{\mathtt{f}}$ and $\delta_{\mathtt{g}}$ are the signatures of f and g respectively.

The information about possible method calls will be useful to approximate the traces of non-terminating computations. A non-terminating computation in Featherweight Java must consist of an infinite series of method calls. To compute an approximation of all infinite traces of such sequences, it suffices to compute the finite effect traces from one method call to the next and to consider their concatenation in all possible infinite sequences of method calls. We do this by computing an expression like S for the body of each method (Section 4.3). Then we compute an approximation of all infinite traces from this information (Section 4.4).

To work with the expressions T and S, we need some notation. While we use different notation for both kinds of expressions to emphasize their different meaning, they are in fact instances of the same kind of formal expressions.

Definition 4.1 (Formal effect expression). Given a set K of keys, we define the set $\mathcal{P}(\Sigma^*)\langle K \rangle$ of *formal effect expressions* to be the set of finite partial functions from K to $\mathcal{P}(\Sigma^*)$. We write $\emptyset$ to denote the empty expression. For $T, T' \in \mathcal{P}(\Sigma^*)\langle K \rangle$ and $U \in \mathcal{P}(\Sigma^*)$, we define:

- $T \subseteq T'$ if and only if $T(x) \subseteq T(x')$ for all $x \in K$,
- $T \cup T' \in \mathcal{P}(\Sigma^*)\langle K \rangle$ by $(T \cup T')(x) = T(x) \cup T'(x)$,
- $U \cdot T \in \mathcal{P}(\Sigma^*)\langle K \rangle$ by $(U \cdot T)(x) = U \cdot T(x)$,

where $T(x)$ is treated as the empty set if $x \notin \mathrm{dom}(T)$.

We use the notation $r_1 \,\&\, U_1 \mid \cdots \mid r_n \,\&\, U_n$ to represent the elements of $\mathcal{P}(\Sigma^*)\langle Reg \rangle$. Such an expression denotes the function mapping r_i to U_i for $i = 1, \ldots, n$. For example, we have $(r_1 \,\&\, U_1 \mid r_2 \,\&\, U_2) \cup (r_1 \,\&\, U_1' \mid r_3 \,\&\, U_3) = (r_1 \,\&\, (U_1 \cup U_1') \mid r_2 \,\&\, U_2 \mid r_3 \,\&\, U_3)$ and $U \cdot (r_1 \,\&\, U_1 \mid r_2 \,\&\, U_2) = (r_1 \,\&\, (U \cdot U_1) \mid r_2 \,\&\, (U \cdot U_2))$. Likewise, call expressions $U_1 \cdot \delta_1 \cup \cdots \cup U_n \cdot \delta_n$ are elements of $\mathcal{P}(\Sigma^*)\langle Sig \rangle$, where Sig is a set of method signatures (to be defined below). When $U_i = \{\varepsilon\}$ we may omit it and simply write δ_i in the expressions.

4.3 Typing Rules

Having explained region and effect expressions, we can now define the type system in detail. As for Featherweight Java, we need a *class table* to record the region types and effect expressions of methods and fields. This is needed to formulate typing rules for method call and field access.

We call a tuple $(C, r, m, \bar{s})$ of a class C, a region r, a method $m \in methods(C)$ and a sequence $\bar{s}$ of regions for the arguments of m a *method signature*. We use $Sig \subseteq Cls \times Reg \times Mtd \times Reg^*$ to denote the set of method signatures.

Definition 4.2 (Class table). A *class table* (F, M) consists of

- a *field typing* $F : Cls \times Reg \times Fld \rightharpoonup \mathcal{P}(Reg)$ that assigns to each class C, region r and field $f \in fields(C)$ a set $F(C, r, f)$ of possible regions of the field f, and

- a *method typing* $M : Sig \to \mathcal{P}(\Sigma^*)\langle Reg\rangle \times \mathcal{P}(\Sigma^*)\langle Sig\rangle$ that assigns to each method signature a pair of an expression for regions and effects of terminating executions and a call expression.

A class table is required to be *well-formed* in the following sense:

- $\mathtt{Null} \in F(C, r, f) \subseteq F(C, \mathtt{Unknown}, f)$;
- $F(C, r, f) = F(D, r, f)$ whenever $C \preceq D$ and $f \in \mathit{fields}(D)$;
- $M(C, r, m, \bar{s}) \subseteq M(D, r, m, \bar{s})$ whenever $C \preceq D$ and $m \in \mathit{methods}(D)$, where the order $\subseteq$ on $\mathcal{P}(\Sigma^*)\langle Reg\rangle \times \mathcal{P}(\Sigma^*)\langle Sig\rangle$ is defined componentwise.

The well-formedness conditions reflect the subtyping properties of FJ. We additionally require all entries of F to contain the region `Null`, because the fields of newly created objects are all initialized to *null* (see the operational semantics in Appendix A). We require F to be invariant w.r.t. also the "largest" region `Unknown`. Note that the tables may contain impossible entries whose class and its subclasses are not in the region. They are not important in the typing inference. We can set them to the "bottom". A typical example is $M(C, \mathtt{Null}, m, \bar{s}) = (\emptyset, \emptyset)$.

We denote $F(C, r, f) = R$ and $M(C, r, m, \langle s_1, \ldots, s_n\rangle) = (T, S)$ as

$$\begin{array}{l}\mathtt{class}\ \ C@r\\ \quad f\ \ :\ \ R\\ \quad m(s_1, \ldots, s_n)\ \ :\ \ T\ \mathsf{calls}\ S\end{array}$$

with which we hope to improve readability.

The typing judgments take the form $\Gamma \vdash e : T\ \ \mathsf{calls}\ \ S$, where $\Gamma : Var \rightharpoonup Reg$ is a typing environment, $e \in Expr$ a term expression, $T \in \mathcal{P}(\Sigma^*)\langle Reg\rangle$ an expression for regions and terminating effects, and $S \in \mathcal{P}(\Sigma^*)\langle Sig\rangle$ an expression for call effects.

The typing rules are given in Figure 1. Rule WEAK says that any possible region r can be "weakened" to `Unknown`. It allows us to merge the sets of possible traces. For instance, if $e : r\,\&\,U \mid \mathtt{Unknown}\,\&\,U'$ then by WEAK we have $e : \mathtt{Unknown}\,\&\,(U \cup U')$. Rule PRIM says that $\mathtt{emit}(a)$ has the terminating trace a and no calls. Rules NULL, NEW and GET have similar effects. Rule ELSE says that if x and y are in disjoint regions, then the whole statement has the same type as the else-branch, because objects in disjoint regions cannot be identical. This illustrates how the usage of regions allows the analysis to achieve some path-sensitivity. In rule LET, e_1 may have multiple regions and effects, so we type e_2 with x in each of these regions r_i to get the effects of e_2 and then take the union of them. In rule CALL for method call, we simply look up the method typing M for the terminating effect and take the method signature as the call expression.

Example 4.3 (Expression Typing). Suppose that there are two classes $D \prec C$ with two methods f and g. Consider the following class table:

$$\begin{array}{lll}\mathtt{class}\ \ C@\mathtt{CreatedAt}(\ell_1) & \qquad & \mathtt{class}\ \ D@\mathtt{CreatedAt}(\ell_2)\\ \quad f()\ :\ \mathtt{Null}\,\&\,\{a\}\ \mathsf{calls}\ \ldots & & \quad f()\ :\ \mathtt{Null}\,\&\,\{b\}\ \mathsf{calls}\ \ldots\\ \quad g()\ :\ \mathtt{Null}\,\&\,\{aa\}\ \mathsf{calls}\ \ldots & & \quad g()\ :\ \mathtt{Null}\,\&\,\{bb\}\ \mathsf{calls}\ \ldots\end{array}$$

Following the typing rules, the FJ-expression

$$\mathtt{if}\ \mathit{cond}\ \mathtt{then}\ (\mathtt{new}^{\ell_1}\ C)\ \mathtt{else}\ (\mathtt{new}^{\ell_2}\ D)$$

has type $\mathtt{CreatedAt}(\ell_1)\,\&\,\{\varepsilon\} \mid \mathtt{CreatedAt}(\ell_2)\,\&\,\{\varepsilon\}$. Write e for the above expression. Now consider the following expressions:

$$\mathtt{let}\ x = e\ \mathtt{in}\ x.f() \quad \mathtt{let}\ x = e\ \mathtt{in}\ x.f(); x.f() \quad \mathtt{let}\ x = e\ \mathtt{in}\ x.g()$$

$$\textsc{sub}\ \frac{\Gamma \vdash e : T\ \ \mathsf{calls}\ S \qquad T \subseteq T' \qquad S \subseteq S'}{\Gamma \vdash e : T'\ \ \mathsf{calls}\ S'}$$

$$\textsc{weak}\ \frac{\Gamma \vdash e : (r\,\&\,U) \cup T\ \ \mathsf{calls}\ S}{\Gamma \vdash e : (\mathtt{Unknown}\,\&\,U) \cup T\ \ \mathsf{calls}\ S}$$

$$\textsc{prim}\ \frac{}{\Gamma \vdash \mathtt{emit}(a) : \mathtt{Null}\,\&\,\{a\}\ \ \mathsf{calls}\ \emptyset}$$

$$\textsc{var}\ \frac{}{\Gamma,\ x{:}\,r \vdash x : r\,\&\,\{\varepsilon\}\ \ \mathsf{calls}\ \emptyset}$$

$$\textsc{if}\ \frac{\Gamma,\ x{:}\,r,\ y{:}\,s \vdash e_1 : T_1\ \ \mathsf{calls}\ S_1 \qquad \Gamma,\ x{:}\,r,\ y{:}\,s \vdash e_2 : T_2\ \ \mathsf{calls}\ S_2}{\Gamma,\ x{:}\,r,\ y{:}\,s \vdash \mathtt{if}\ x = y\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 : T_1 \cup T_2\ \ \mathsf{calls}\ S_1 \cup S_2}$$

$$\textsc{else}\ \frac{\mathit{disjoint}(r, s) \qquad \Gamma,\ x{:}\,r,\ y{:}\,s \vdash e_2 : T_2\ \ \mathsf{calls}\ S_2}{\Gamma,\ x{:}\,r,\ y{:}\,s \vdash \mathtt{if}\ x = y\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 : T_2\ \ \mathsf{calls}\ S_2}$$

$$\textsc{let}\ \frac{\begin{array}{c}\Gamma \vdash e_1 : r_1\,\&\,U_1 \mid \ldots \mid r_n\,\&\,U_n\ \ \mathsf{calls}\ S\\ \Gamma,\ x{:}\,r_i \vdash e_2 : T_i\ \ \mathsf{calls}\ S_i\end{array}}{\Gamma \vdash \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 : \bigcup_{i=1}^{n} U_i \cdot T_i\ \ \mathsf{calls}\ S \cup \bigcup_{i=1}^{n} U_i \cdot S_i}$$

$$\textsc{null}\ \frac{}{\Gamma \vdash \mathtt{null} : \mathtt{Null}\,\&\,\{\varepsilon\}\ \ \mathsf{calls}\ \emptyset}$$

$$\textsc{new}\ \frac{}{\Gamma \vdash \mathtt{new}^{\ell}\ C : \mathtt{CreatedAt}(\ell)\,\&\,\{\varepsilon\}\ \ \mathsf{calls}\ \emptyset}$$

$$\textsc{cast}\ \frac{\Gamma \vdash e : T\ \ \mathsf{calls}\ S}{\Gamma \vdash (D)\,e : T\ \ \mathsf{calls}\ S}$$

$$\textsc{get}\ \frac{T = \bigcup\{s\,\&\,\{\varepsilon\} \mid s \in F(C, r, f)\}}{\Gamma,\ x{:}\,r \vdash x^{C}.f : T\ \ \mathsf{calls}\ \emptyset}$$

$$\textsc{set}\ \frac{s \in F(C, r, f)}{\Gamma,\ x{:}\,r,\ y{:}\,s \vdash x^{C}.f := y : s\,\&\,\{\varepsilon\}\ \ \mathsf{calls}\ \emptyset}$$

$$\textsc{call}\ \frac{(T, \_) = M(C, r, m, \bar{s})}{\Gamma,\ x{:}\,r,\ \bar{y}{:}\,\bar{s} \vdash x^{C}.m(\bar{y}) : T\ \ \mathsf{calls}\ (C, r, m, \bar{s})}$$

Figure 1: Region-based type and effect system

The first expression has type $\mathtt{Null}\,\&\,\{a, b\}$, the second and third one have type $\mathtt{Null}\,\&\,\{aa, bb\}$. One may have expected the traces of the second expression to be $\{a, b\} \cdot \{a, b\}$, that is, $\{aa, ab, ba, bb\}$, because x may be in two different regions. Indeed, this is the case in the system of [5], but not in ours. In the LET rule, the body $x.f(); x.f()$ is typed twice with the two regions of x, and the results $\{aa\}$ and $\{bb\}$ are joined in the end, resulting in $\{aa, bb\}$.

This seems to be a slight defect of [5]. Indeed, the function g could have been defined to have body $\mathtt{this}.f(); \mathtt{this}.f()$. In this case, the second expression would be the inlining of the third and one would perhaps expect inlining not to lose information. $\square$

Definition 4.4 (Well-typedness). An FJ program $(\prec, \mathit{fields}, \mathit{methods}, \mathit{mtable})$ is *well-typed* w.r.t. a class table (F, M) if, for each method signature $(C, r, m, \bar{s}) \in Sig$ with $C \in Cls(r)$ and $M(C, r, m, \bar{s}) = (T, S)$ and $\mathit{mtable}(C, m) = (\bar{x}, e)$, the judgment $\mathtt{this}{:}\,r,\ \bar{x}{:}\,\bar{s} \vdash e : T\ \mathsf{calls}\ S$ is derivable.

The program from Example 3.1 is well-typed w.r.t. the class table in Example 4.7. Another example of a well-typed program is given in Example 4.5.

One subtle aspect of well-typedness is the condition $C \in \mathit{Cls}(r)$. This condition allows us to use the region information to statically narrow down the possible targets of calls. This is an improvement over [5], where the region information was not used. With the condition, well-typedness imposed no requirement at all on the method table entry $M(C, r, m, \bar{s})$ when $C \notin \mathit{Cls}(r)$. Informally, this is because the method table entry belongs to a method that, in region r, can never be executed.

The following example illustrates how our type system makes use of region information to narrow down the possible targets of call expressions.

Example 4.5 (Regions and Subtyping). Consider a program with two simple classes (the return type is `Object` only because we do not have a `void` type):

```
class A {                      class B extends A {
  Object f() {                   Object f() {
    emit(a);                       emit(b);
    return null;                   return null;
  }                              }
}                              }
```

Let ℓ_1 and ℓ_2 be labels such that $\mathit{Cls}(\texttt{CreatedAt}(\ell_1)) = \{\texttt{A}\}$ and $\mathit{Cls}(\texttt{CreatedAt}(\ell_2)) = \{\texttt{B}\}$.

The program is well-typed w.r.t. the following class table:

```
class A@Unknown                   class B@Unknown
  f(): Null & {a, b} calls ∅        f(): Null & {b} calls ∅

class A@CreatedAt(ℓ1)             class B@CreatedAt(ℓ1)
  f(): Null & {a} calls ∅           f(): ∅ calls ∅

class A@CreatedAt(ℓ2)             class B@CreatedAt(ℓ2)
  f(): Null & {b} calls ∅           f(): Null & {b} calls ∅
```

The types in region `Unknown` should be unsurprising. The requirement that the typings be closed under superclasses requires us to include the effect b also in class A.

In region $\texttt{CreatedAt}(\ell_1)$, the entry for `f` in class `A` should be unsurprising. The entry for `f` in `B` can be explained as follows. Because of $\mathit{Cls}(\texttt{CreatedAt}(\ell_1)) = \{\texttt{A}\}$, we know that all objects in region $\texttt{CreatedAt}(\ell_1)$ have class `A`. This means that the code for `f` in `B` will never be executed and we may assume the method to have empty effect. The type system supports this reasoning because the definition of well-typedness for programs makes a requirement only for entries $(C, r, m, \bar{s})$ with $C \in \mathit{Cls}(r)$. Accordingly, we may take the empty effect for the method table entry for `f` in `B`.

In region $\texttt{CreatedAt}(\ell_2)$, it is the entry for `f` in `A` that needs explanation. It is reasonable because $\mathit{Cls}(\texttt{CreatedAt}(\ell_1)) = \{\texttt{B}\}$ tells us that the actual class of any object in region $\texttt{CreatedAt}(\ell_2)$ is `B`, so that any call of `f` would go to the code from class `B`. In the method table, we cannot take the empty effect for `f` in `A`, however, as the table needs to be closed under superclasses. We need to include at least the effects of `B.f` in those of `A.f`.

With the above class table, consider the following code fragment.

```
A x = new^ℓ2 B();
x.f();
```

The type system allows us to give `x` type $\texttt{CreatedAt}(\ell_2)$. In this case, the method call `x.f()` is analyzed to have effect $\{b\}$, which is exact. We could also give `x` the type `Unknown`, but this choice would lead to an over-approximation of the actual effect. We would get that the method call `x.f()` has effect $\{a, b\}$. However, we would not be allowed to give `x` the type $\texttt{CreatedAt}(\ell_1)$. □

4.4 Infinite Traces

We now turn to analyzing possibly infinite traces of non-terminating programs. These cannot be captured directly from an inductive typing derivation. Instead, we use the type system to justify the call expressions (see Definition 4.4 of well-typed programs) and subject them the following *coinductive* treatment which will lead to the correctness of the non-terminating effects (Theorem 4.8).

Definition 4.6 (Infinitary effect typing). Given a method typing M, we write S_δ to denote the second components of $M(\delta)$. We call an assignment $\eta : \mathit{Sig} \to \mathcal{P}(\Sigma^{\leq\omega})$ an *infinitary effect typing for M* if it is the *greatest solution* of the equation system $\{\delta = S_\delta \mid \delta \in \mathit{Sig}\}$.

In the equation system in this definition, we treat method signatures as variables ranging over $\mathcal{P}(\Sigma^{\leq\omega})$. We can then consider call expressions to define languages in $\mathcal{P}(\Sigma^{\leq\omega})$ too, by interpreting the operations in them as language operations.

We will prove the correctness of the non-terminating effects by induction on an approximation to the greatest solution η, but begin with an example for the definition.

Example 4.7 (Finite and Infinite Effects). A possible class table for the program from Example 3.1 is shown below. It uses the regions $r_1 = \texttt{CreatedAt}(\ell_1)$, $r_2 = \texttt{CreatedAt}(\ell_2)$ and $r_3 = \texttt{CreatedAt}(\ell_3)$. In the table we use the abbreviation $\delta_r = (\texttt{Node}, r, \texttt{last}, ())$.

```
class Node@Null
  next : {Null}
  last() : ∅ calls ∅

class Node@r1
  next : {Null}
  last() : r1 & {a} calls {a} · δNull

class Node@r2
  next : {Null, r1}
  last() : r1 & {aa} | r2 & {a} calls {a} · δr1

class Node@r3
  next : {Null, r3}
  last() : r3 & a+ calls {a} · δr3

class Test@Unknown
  linear() : r1 & {aa} | r2 & {a} calls δr2
  cyclic() : r3 & a+ calls δr3
```

The members of class `Node` are given different types depending on the region. In region r_1, the field `next` has region `Null`. This means that `next` can only be *null*. As a result, method `last` will always have a as its trace, thus its terminating effect is $\{a\}$. In region r_2, the field `next` has region $\{\texttt{Null}, r_1\}$. If method `last` returns `this` then it has region r_2 and effect $\{a\}$; otherwise, it returns `next.last()` and thus has region r_1 and effect $\{aa\}$, where the second a is emitted by `last` at r_1. This explains the annotation $r_1 \mathbin{\&} \{aa\} \mid r_2 \mathbin{\&} \{a\}$. In region r_3, we have a circularity, as `next` has also region r_3.

We have the following equation system from the method typing:

$$\begin{aligned} &\delta_{\texttt{Null}} = \emptyset & \delta_{r_3} &= \{a\} \cdot \delta_{r_3} \\ &\delta_{r_1} = \{a\} \cdot \delta_{\texttt{Null}} & (\texttt{Test}, \texttt{Unknown}, \texttt{linear}, ()) &= \delta_{r_2} \\ &\delta_{r_2} = \{a\} \cdot \delta_{r_1} & (\texttt{Test}, \texttt{Unknown}, \texttt{cyclic}, ()) &= \delta_{r_3} \end{aligned}$$

The greatest solution η maps $\delta_{\texttt{Null}}$, δ_{r_1}, δ_{r_2} and $(\texttt{Test}, \texttt{Unknown}, \texttt{linear}, ())$ to $\emptyset$, and δ_{r_3} and $(\texttt{Test}, \texttt{Unknown}, \texttt{cyclic}, ())$ to a^ω, and is suitable as an infinitary effect typing. □

4.5 Soundness of the Type System

To formulate the soundness of the region type system w.r.t. the operational semantics, we give a formal interpretation of regions as a relation $(v, h) \vdash r$, read as "the value v at heap h satisfies the property r", generated from the following rules.

$$\frac{}{(null, h) \vdash \texttt{Null}} \qquad \frac{}{(v, h) \vdash \texttt{Unknown}}$$

$$\frac{h(l) = (C, G, \ell) \qquad C \in Cls(\texttt{CreatedAt}(\ell))}{(l, h) \vdash \texttt{CreatedAt}(\ell)}$$

It is lifted to a relation between stores and typing environments and one between heaps and field typings as follows:

- $(s, h) \vdash \Gamma$ iff $(s(x), h) \vdash r$ for all $(x{:}\, r) \in \Gamma$.
- $h \vdash F$ iff $(G(f), h) \vdash F(C, r, f)$ for all $l \in \mathrm{dom}(h)$ with $h(l) = (C, G, \_)$ and for all r, f with $(C, r, f) \in \mathrm{dom}(F)$.

In words, $(s, h) \vdash \Gamma$ expresses that all the values in the store s satisfy the properties specified in the environment Γ, and $h \vdash F$ says that the fields of each object in the heap h satisfy the properties specified in the field typing F. We write $(s, h) \vdash \Gamma, F$ to denote the conjunction of $(s, h) \vdash \Gamma$ and $h \vdash F$.

Theorem 4.8 (Soundness). *Let P be a well-typed program w.r.t. a class table (F, M). Let η be an infinitary effect typing for M. For any Γ, e, T, S, s, h, v, h' and w with*

$$\Gamma \vdash e : T \ \texttt{ calls } \ S \quad \textit{and} \quad (s, h) \vdash \Gamma, F$$

(1) if $(s, h) \vdash e \Downarrow v, h' \,\&\, w$, then $(s, h') \vdash \Gamma, F$ and $(v, h') \vdash r$ and $w \in U$ for some $(r \,\&\, U) \in T$;

(2) if $(s, h) \vdash e \Uparrow \,\&\, w$, then $w \in S(\eta)$, where $S(\eta) \subseteq \Sigma^{\leq \omega}$ is obtained by substituting every occurrence of δ in S by $\eta(\delta)$.

Proof of Theorem 4.8(1). We carry out the proof by induction over the sum of the depth of the derivation of the typing judgment and the depth of the derivation of the operational semantics judgment. We refer the reader to a similar proof of [5, Theorem 5]. □

We focus on the soundness of the infinitary effect analysis. Recall that any expression $S \in \mathcal{P}(\Sigma^*)\langle Sig \rangle$ is of the form $\bigcup_{\delta \in \Delta} A_\delta \cdot \delta$ for some $\Delta \subseteq Sig$. If we treat the signatures as variables then each expression gives a $\mathcal{P}(\Sigma^{\leq\omega})$-valued function, *i.e.*, for any assignment $\eta : Sig \to \mathcal{P}(\Sigma^{\leq\omega})$, we obtain a language $S(\eta) \subseteq \Sigma^{\leq\omega}$ by substituting all δ in S by $\eta(\delta)$. It is obvious that if $S \subseteq S'$ then $S(\eta) \subseteq S'(\eta)$ for any η.

For the proof of Theorem 4.8(2), we use the following approximations of the greatest solution: Given a class table (F, M), let $\{S_\delta \mid \delta \in Sig\}$ be the call expressions encoded in M. We define for each $n \in \mathbb{N}$ an assignment function $\eta_n : Sig \to \mathcal{P}(\Sigma^{\leq\omega})$ by

$$\eta_0(\delta) = \Sigma^{\leq\omega} \qquad \eta_{n+1}(\delta) = S_\delta(\eta_n).$$

Let η be the greatest solution of $\{\delta = S_\delta \mid \delta \in Sig\}$. Then we have $\eta(\delta) = \bigcap_{i \in \mathbb{N}} \eta_i(\delta)$ for every $\delta \in Sig$. More generally, we have $\eta(\delta) = \bigcap_{i \in N} \eta_i(\delta)$ for any infinite subset $N \subseteq \mathbb{N}$.

We firstly show that the above approximations satisfy the soundness theorem in the following sense:

Lemma 4.9. *Given a well-typed program w.r.t. (F, M), let η_n be the approximations to the greatest solution as defined above. For any n, Γ, e, T, S, s, h and w, if*

$$\Gamma \vdash e : T \ \texttt{ calls } \ S \quad \textit{and} \quad (s, h) \vdash \Gamma, F \quad \textit{and} \quad (s, h) \vdash e \Uparrow \,\&\, w$$

then $w \in S(\eta_n)$.

Proof. By induction on n and on the depth of the typing derivation. When $n = 0$, it is trivial because η_0 assigns $\Sigma^{\leq\omega}$ to all variables. Assume that the statement holds for n. To show that it holds for $n+1$, we perform an induction on the depth of the typing derivation.

If the last used rule of the typing derivation is sub or weak, then the induction hypothesis gives the desired result. Otherwise, we perform a case distinction over the last rule used in the derivation of the operational semantics judgment. The cases of x, $\texttt{emit}(a)$, *null* and $\texttt{new}\ C$ are ruled out because their execution always terminates. Here we check the following cases as examples:

Suppose the last used typing rule is let, *i.e.*, the judgment is

$$\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \textstyle\bigcup_{i=1}^n U_i \cdot T_i \ \texttt{ calls } \ S \cup \bigcup_{i=1}^n U_i \cdot S_i.$$

By rule inversion, we have $\Gamma \vdash e_1 : r_1 \,\&\, U_1 \mid \ldots \mid r_n \,\&\, U_n \ \texttt{ calls } \ S$ and $\Gamma, x{:}\, r_i \vdash e_2 : T_i \ \texttt{ calls } \ S_i$ for all $i \in \{1, \ldots, n\}$. Assume $(s, h) \vdash \Gamma, F$ and $(s, h) \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \Uparrow \,\&\, w$. By rule inversion, we have two possibilities:

- If $(s, h) \vdash e_1 \Downarrow v_1, h_1 \,\&\, w_1$ and $(s[x \mapsto v_1], h_1) \vdash e_2 \Uparrow \,\&\, w_2$, then $w = w_1 w_2$. By Theorem 4.8(1), we have $(s, h_1) \vdash \Gamma, F$ and $(v_1, h_1) \vdash r_i$ and $w_1 \in U_i$ for some i. From the former we get $(s[x \mapsto v_1], h_1) \vdash (\Gamma, x{:}\, r_i), F$. Then by induction hypothesis we know $w_2 \in S_i(\eta_{n+1})$, and thus $w = w_1 w_2 \in U_i \cdot S_i(\eta_{n+1}) \subseteq (S \cup \bigcup_{i=1}^n U_i \cdot S_i)(\eta_{n+1})$.
- If $(s, h) \vdash e_1 \Uparrow \,\&\, w$, then by induction hypothesis we have $w \in S(\eta_{n+1}) \subseteq (S \cup \bigcup_{i=1}^n U_i \cdot S_i)(\eta_{n+1})$.

Suppose the last used typing rule is call, *i.e.*, the judgment is

$$\Gamma,\ x{:}\, r,\ \bar{y}{:}\, \bar{s} \vdash x^C.m(\bar{y}) : T \ \texttt{ calls } \ (C, r, m, \bar{s})$$

where $(T, \_) = M(C, r, m, \bar{s})$. Assume $(s, h) \vdash (\Gamma, x{:}\, r, \bar{y}{:}\, \bar{s}), F$ and $(s, h) \vdash x.m(\bar{y}) \Uparrow \,\&\, w$. The goal is to show $w \in \eta_{n+1}(C, r, m, \bar{s})$. Let $s(x) = l$ and $h(l) = (D, \_, \_)$ and $mtable(D, m) = (\bar{z}, e)$ and $s' = [\texttt{this} \mapsto l] \cup [z_i \mapsto s(y_i)]_{i \in \{1, \ldots, |\bar{z}|\}}$. We have $(s', h) \vdash e \Uparrow \,\&\, w$ by rule inversion. From the assumption $(s, h) \vdash (\Gamma, x{:}\, r, \bar{y}{:}\, \bar{s}), F$ we have $(l, h) \vdash r$ which implies $D \in Cls(r)$. Because the program is well-typed, we have $\Gamma' \vdash e : T' \ \texttt{ calls } \ S'$ where $\Gamma' = \texttt{this}{:}\, r, \bar{z}{:}\, \bar{s}$ and $(T', S') = M(D, r, m, \bar{s})$. From $(s, h) \vdash (\Gamma, x{:}\, r, \bar{y}{:}\, \bar{s}), F$ we have also $(s', h) \vdash \Gamma', F$. Then the induction hypothesis gives us $w \in S'(\eta_n)$. Because M is well-formed and $D \leq C$, we have $S' \subseteq S_{C,r,m,\bar{s}}$ and thus $w \in S_{C,r,m,\bar{s}}(\eta_n) = \eta_{n+1}(C, r, m, \bar{s})$ by the definition of the approximations.

We leave the remaining cases to the reader. □

Lemma 4.10. *For any call expression S and infinite set $N \subseteq \mathbb{N}$, we have $\bigcap_{i \in N} S(\eta_i) \subseteq S(\eta)$.*

Proof. By induction on the number of signatures in S. □

The second half of the soundness theorem follows from the above lemmas:

Proof of Theorem 4.8(2). If $\Gamma \vdash e : T$ calls S and $(s,h) \vdash \Gamma, F$ and $(s,h) \vdash e \Uparrow \& w$, then we have $w \in \bigcap_{n\in\mathbb{N}} S(\eta_n) \subseteq S(\eta)$ by the above lemmas. □

Therefore, if a program is well-typed, then the effects encoded in the method typing are correct in the following sense:

Corollary 4.11 (Soundness). *Suppose that a program is well-typed w.r.t. (F, M) and that η is an infinitary effect typing for M. Then, for any $(C, r, m, \bar{s}) \in Sig$ with $M(C, r, m, \bar{s}) = (T, S)$ and $\eta(C, r, m, \bar{s}) = V$ and for any $x{:}\,r$ and $\bar{y}{:}\,\bar{s}$, we have*

(1) if $x.m(\bar{y})$ evaluates to a value v and generates a trace w, then v is in region t and $w \in U$ for some $(t \,\&\, U) \in T$; and
(2) if $x.m(\bar{y})$ diverges and generates a trace w, then $w \in V$.

As a result, to verify whether a program adheres to a guideline, it suffices to check if the effects encoded in the typings M and η are allowed by the guideline.

5 BÜCHI EFFECT TYPE SYSTEM

The effect type system introduced in Section 4 is not yet suitable for practical use. Because effects are given as arbitrary languages, we have no direct way to compute greatest fixed points of equation systems over $\mathcal{P}(\Sigma^{\leq\omega})$. In this section, we introduce a system (Section 5.2) where effect annotations are taken from a *finite* structure to represent the languages over Σ (Section 5.1) using the methods from abstract interpretation [4]. In Section 5.3, we present a type inference algorithm that infers a class table for a given program, provided a standard FJ typing for the program is given.

5.1 Büchi Abstraction

Instead of $\mathcal{P}(\Sigma^*)$ and $\mathcal{P}(\Sigma^{\leq\omega})$, we work with a finite structure that abstracts the languages over Σ for effect annotations, over which we can compute fixed points.

Definition 5.1 (Büchi algebra). Let $\mathcal{M}_*, \mathcal{M}_{\leq\omega}$ be join-semilattices, that is, a partial order together with a join operation. We use the symbols $\sqsubseteq, \sqcup$ to refer to their generic structure of join-semilattices. We call $(\mathcal{M}_*, \mathcal{M}_{\leq\omega})$ a *Büchi algebra* if it is equipped with the following operations:

- a binary operation on $\mathcal{M}_*$, written as $\mathcal{A} \cdot \mathcal{B}$ for $\mathcal{A}, \mathcal{B} \in \mathcal{M}_*$,
- a mapping $\mathcal{M}_* \times \mathcal{M}_{\leq\omega} \to \mathcal{M}_{\leq\omega}$, also written as $\mathcal{A} \cdot \mathcal{V}$ for $\mathcal{A} \in \mathcal{M}_*$ and $\mathcal{V} \in \mathcal{M}_{\leq\omega}$, and
- an operation $(-)^\omega : \mathcal{M}_* \to \mathcal{M}_{\leq\omega}$,

such that all the operations are monotone and the binary operations are associative, *i.e.*, $\mathcal{A} \cdot (\mathcal{B} \cdot \mathcal{U}) = (\mathcal{A} \cdot \mathcal{B}) \cdot \mathcal{U}$ for any $\mathcal{A}, \mathcal{B} \in \mathcal{M}_*$ and $\mathcal{U} \in \mathcal{M}_* \cup \mathcal{M}_{\leq\omega}$.

For instance, $\mathcal{P}(\Sigma^*)$ and $\mathcal{P}(\Sigma^{\leq\omega})$ form a Büchi algebra. They are complete lattices. But note that meets are not required here because set intersections are not needed in the type system. We are looking for a *finite* Büchi algebra $(\mathcal{M}_*, \mathcal{M}_{\leq\omega})$ that abstracts them.

Recall that a *Galois insertion* between partial orders A, C consists of monotone maps $\alpha : C \to A$ and $\gamma : A \to C$ such that, for all $a \in A$ and $c \in C$,

$$\alpha(\gamma(a)) = a \qquad \gamma(\alpha(c)) \sqsupseteq c.$$

Intuitively, the function α furnishes for each concrete value in C its *abstraction*, whereas γ *concretizes* the abstract values in A. Moreover, the abstraction of c has a concretization lying above c.

Now we are ready to introduce the finite structure needed for presenting the type system and the type inference algorithm.

Definition 5.2 (Büchi abstraction). A *Büchi abstraction* is a *finite* Büchi algebra $(\mathcal{M}_*, \mathcal{M}_{\leq\omega})$ equipped with Galois insertions (α_*, γ_*) between $\mathcal{M}_*$ and $\mathcal{P}(\Sigma^*)$ and $(\alpha_{\leq\omega}, \gamma_{\leq\omega})$ between $\mathcal{M}_{\leq\omega}$ and $\mathcal{P}(\Sigma^{\leq\omega})$ such that the abstraction functions α_* and $\alpha_{\leq\omega}$ are compatible with the algebraic structure in the sense that

$$\alpha_*(A \cdot B) = \alpha_*(A) \cdot \alpha_*(B)$$
$$\alpha_{\leq\omega}(A \cdot V) = \alpha_*(A) \cdot \alpha_{\leq\omega}(V)$$
$$\alpha_{\leq\omega}(A^\omega) = \alpha_*(A)^\omega$$

for all $A, B \subseteq \Sigma^*$ and $V \subseteq \Sigma^{\leq\omega}$.

One may have expected more structures on Büchi abstraction due to the analogy to $(\mathcal{P}(\Sigma^*), \mathcal{P}(\Sigma^{\leq\omega}))$. Some structure can be constructed. For instance, the least elements of $\mathcal{M}_*$ and $\mathcal{M}_{\leq\omega}$ are exactly the abstraction of the empty set. We define the finite iteration $(-)^*$ on $\mathcal{M}_*$ as the least fixed point $\mathcal{A}^* = \mathsf{lfp}(\lambda X.\alpha_*(\{\varepsilon\}) \sqcup \mathcal{A} \cdot X)$, and then have $\alpha_*(A^*) = (\alpha_*(A))^*$. Moreover, we have the following properties of the Büchi abstraction.

Proposition 5.3. *Büchi abstractions have the following properties:*

(1) The abstraction functions α_* and $\alpha_{\leq\omega}$ preserve joins, i.e.,

$$\alpha_*(A \cup B) = \alpha_*(A) \sqcup \alpha_*(B)$$
$$\alpha_{\leq\omega}(U \cup V) = \alpha_{\leq\omega}(U) \sqcup \alpha_{\leq\omega}(V).$$

(2) The concretization functions γ_* and $\gamma_{\leq\omega}$ may not preserve joins or concatenations, but they satisfy

$$\gamma_*(\mathcal{A} \sqcup \mathcal{B}) \supseteq \gamma_*(\mathcal{A}) \cup \gamma_*(\mathcal{B})$$
$$\gamma_{\leq\omega}(\mathcal{U} \sqcup \mathcal{V}) \supseteq \gamma_{\leq\omega}(\mathcal{U}) \cup \gamma_{\leq\omega}(\mathcal{V})$$
$$\gamma_*(\mathcal{A} \cdot \mathcal{B}) \supseteq \gamma_*(\mathcal{A}) \cdot \gamma_*(\mathcal{B})$$
$$\gamma_{\leq\omega}(\mathcal{A} \cdot \mathcal{V}) \supseteq \gamma_*(\mathcal{A}) \cdot \gamma_{\leq\omega}(\mathcal{V}).$$

(3) The concatenation operations are distributive over joins, i.e.,

$$(\mathcal{A} \sqcup \mathcal{B}) \cdot \mathcal{V} = \mathcal{A} \cdot \mathcal{V} \sqcup \mathcal{B} \cdot \mathcal{V}$$
$$\mathcal{A} \cdot (\mathcal{U} \sqcup \mathcal{V}) = \mathcal{A} \cdot \mathcal{U} \sqcup \mathcal{A} \cdot \mathcal{V}.$$

(4) Lastly, we have $\alpha_{\leq\omega}(\gamma_*(\mathcal{A})^\omega) = \mathcal{A}^\omega = \mathcal{A} \cdot \mathcal{A}^\omega$.

Note that, differing from finite iteration, here we do not define (or require) ω-iteration to be the greatest fixed point. We have only $\mathcal{A}^\omega \sqsubseteq \mathsf{gfp}(\lambda X.\mathcal{A} \cdot X)$ in general, because $\mathcal{A}^\omega$ is a fixed point of $\lambda X.\mathcal{A} \cdot X$ by Proposition 5.3(4). In the following example, the ω-iteration is strictly smaller than the greatest fixed point.

Example 5.4 (Büchi Abstraction). Let the alphabet be $\Sigma = \{a\}$. We give a concrete example for a Büchi abstraction that distinguishes empty from non-empty words:

$$\mathcal{M}_* = \{\emptyset, \{\varepsilon\}, a^+, a^*\}$$
$$\mathcal{M}_{\leq\omega} = \{\emptyset, \{\varepsilon\}, a^+, a^*, a^\omega, \{\varepsilon\} \cup a^\omega, a^+ \cup a^\omega, a^* \cup a^\omega\}.$$

For both, $\sqcup$ and $\sqsubseteq$ are set union and inclusion. All other operations are given by the evident operations on languages. The abstraction function α_* is determined by $\alpha_*(\{\varepsilon\}) = \{\varepsilon\}$ and $\alpha_*(X) = a^+$ for any $X \subseteq a^+$. All other values are determined by preservation of least element and joins. Similarly, $\alpha_{\le\omega}$ is determined by $\alpha_{\le\omega}(\{\varepsilon\}) = \{\varepsilon\}$, $\alpha_{\le\omega}(X) = a^+$ for any $X \subseteq a^+$ and $\alpha_{\le\omega}(a^\omega) = a^\omega$. The concretion functions γ_* and $\gamma_{\le\omega}$ are both the identity.

We note that the greatest solution of the equation $X = a^+ \cdot X$ over $\mathcal{M}_{\le\omega}$ is $a^+ \cup a^\omega$ just by direct calculation. When calculated in $\mathcal{P}(\Sigma^{\le\omega})$, the greatest solution is a^ω, which is the ω-iteration of a^+ over $\mathcal{M}_{\le\omega}$. ◻

As will become clear, a Büchi abstraction is sufficient for presenting the type system and the type inference algorithm. Hofmann and Chen [12, 13] construct a Büchi abstraction from an *extended Büchi automaton* $\mathfrak{A}$ whose language is the union of the language of $\mathfrak{A}$ when understood as a traditional nondeterministic finite automaton and the language of $\mathfrak{A}$ when understood as a traditional Büchi automaton. A programming guideline is simply a set of finite and infinite traces, and thus can be formalized by an extended Büchi automaton. Moreover, the Büchi abstraction of Hofmann and Chen is faithful with respect to acceptance by the automaton, which makes it suitable for representing effects in our type system. However, the development of the system (Section 5.2) and the type inference algorithm (Section 5.3) are *independent* of the construction of the Büchi abstraction. Readers should be able to access them without probing into the construction of Hofmann and Chen.

Similarly to the development in the previous section, we work with formal expressions $\mathcal{M}_*\langle Reg\rangle$ and $\mathcal{M}_*\langle Sig\rangle$ in the type system. We define the order $\sqsubseteq$ and the operations of concatenation $(\cdot)$ and join $(\sqcup)$ on expressions in the same way as in Definition 4.1. To relate to the previous system, we extend the abstraction and concretization functions to expressions: Let K be either Reg or Sig. Given $\mathcal{T} \in \mathcal{M}_*\langle K\rangle$, we concretize it to $\gamma(\mathcal{T}) \in \mathcal{P}(\Sigma^*)\langle K\rangle$ by defining $\gamma(\mathcal{T})(x) = \gamma_*(\mathcal{T}(x))$. Similarly, we abstract $T \in \mathcal{P}(\Sigma^*)\langle K\rangle$ to $\alpha(T) \in \mathcal{M}_*\langle K\rangle$. By the definition and properties of Büchi abstraction, we have:

$$\begin{aligned} \alpha(\gamma(\mathcal{T})) &= \mathcal{T} & \gamma(\alpha(T)) &\supseteq T \\ \alpha(T \cup T') &= \alpha(T) \sqcup \alpha(T') & \gamma(\mathcal{T} \sqcup \mathcal{T}') &\supseteq \gamma(\mathcal{T}) \cup \gamma(\mathcal{T}') \\ \alpha(U \cdot T) &= \alpha_*(U) \cdot \alpha(T) & \gamma(\mathcal{U} \cdot \mathcal{T}) &\supseteq \gamma_*(\mathcal{U}) \cdot \gamma(\mathcal{T}). \end{aligned}$$

5.2 Büchi Effect Type System

In this section, we assume a programming guideline given by the language $\mathfrak{A}$ and a Büchi abstraction $(\mathcal{M}_*, \mathcal{M}_{\le\omega})$ which is faithful w.r.t. $\mathfrak{A}$, *i.e.*, $\gamma_{\le\omega}(\alpha_{\le\omega}(\mathfrak{A})) = \mathfrak{A}$.

We now present an algorithmic type system with effects being elements of $\mathcal{M}_*$ and $\mathcal{M}_{\le\omega}$. The development is similar to the one in the previous section. We remove the subtyping rule and replace sets of traces by their abstractions in the type system. Thanks to the finite abstraction, we can compute greatest solutions to capture the possible non-terminating traces of a program. However, the abstraction may not preserve greatest fixed points in general. To make the analysis more precise, we concretize the abstract call expressions when defining infinitary effect typing. And we provide a simple algorithm to compute the infinitary effects that abstract the greatest fixed point of the concretized expressions.

In a *Büchi class table* $(F, M_{\mathfrak{A}})$, the field typing $F : Cls \times Reg \times Fld \rightharpoonup \mathcal{P}(Reg)$ is the same as the one in the previous setting, while the method typing $M_{\mathfrak{A}} : Sig \to \mathcal{M}_*\langle Reg\rangle \times \mathcal{M}_*\langle Sig\rangle$ gives effect expressions with abstract sets of traces. We require $(F, M_{\mathfrak{A}})$ to be well-formed as in the previous setting.

A *Büchi typing judgment* takes the form $\Gamma \vdash_{\mathfrak{A}} e : \mathcal{T}$ calls $\mathcal{S}$ where $\mathcal{T} : \mathcal{M}_*\langle Reg\rangle$ lists the possible regions and terminating effects of e and $\mathcal{S} : \mathcal{M}_*\langle Sig\rangle$ specifies the information about the calls that e may make as in the previous system, except that effects are given by the Büchi abstraction. The typing rules are adapted correspondingly: Any singleton $\{a\}$ becomes $\alpha_*(\{a\})$ and the operations $\cup$ and $\cdot$ on languages are replaced by those from the Büchi algebra. The rules are listed in Figure 2. An FJ program is *well-typed* w.r.t. $(F, M_{\mathfrak{A}})$ if for each $(C, r, m, \bar{s}) \in Sig$ with $C \in Cls(r)$ and $M_{\mathfrak{A}}(C, r, m, \bar{s}) = (\mathcal{T}, \mathcal{S})$ and $mtable(C, m) = (\bar{x}, e)$, the judgment $\mathtt{this}{:}r,\ \bar{x}{:}\bar{s} \vdash_{\mathfrak{A}} e : \mathcal{T}$ calls $\mathcal{S}$ is derivable.

$$\textsc{sub}\ \frac{\Gamma \vdash_{\mathfrak{A}} e : \mathcal{T} \text{ calls } \mathcal{S} \quad \mathcal{T} \sqsubseteq \mathcal{T}' \quad \mathcal{S} \sqsubseteq \mathcal{S}'}{\Gamma \vdash_{\mathfrak{A}} e : \mathcal{T}' \text{ calls } \mathcal{S}'}$$

$$\textsc{weak}\ \frac{\Gamma \vdash_{\mathfrak{A}} e : (r \,\&\, \mathcal{U}) \sqcup \mathcal{T} \text{ calls } \mathcal{S}}{\Gamma \vdash_{\mathfrak{A}} e : (\mathsf{Unknown} \,\&\, \mathcal{U}) \sqcup \mathcal{T} \text{ calls } \mathcal{S}}$$

$$\textsc{prim}\ \frac{}{\Gamma \vdash_{\mathfrak{A}} \mathtt{emit}(a) : \mathtt{Null} \,\&\, \alpha_*(\{a\}) \text{ calls } \emptyset}$$

$$\textsc{var}\ \frac{}{\Gamma,\ x{:}\,r \vdash_{\mathfrak{A}} x : r \,\&\, \alpha_*(\{\varepsilon\}) \text{ calls } \emptyset}$$

$$\textsc{if}\ \frac{\Gamma,\ x{:}\,r,\ y{:}\,s \vdash_{\mathfrak{A}} e_1 : \mathcal{T}_1 \text{ calls } \mathcal{S}_1 \quad \Gamma,\ x{:}\,r,\ y{:}\,s \vdash_{\mathfrak{A}} e_2 : \mathcal{T}_2 \text{ calls } \mathcal{S}_2}{\Gamma,\ x{:}\,r,\ y{:}\,s \vdash_{\mathfrak{A}} \mathtt{if}\ x = y\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 : \mathcal{T}_1 \sqcup \mathcal{T}_2 \text{ calls } \mathcal{S}_1 \sqcup \mathcal{S}_2}$$

$$\textsc{else}\ \frac{disjoint(r, s) \qquad \Gamma,\ x{:}\,r,\ y{:}\,s \vdash_{\mathfrak{A}} e_2 : \mathcal{T}_2 \text{ calls } \mathcal{S}_2}{\Gamma,\ x{:}\,r,\ y{:}\,s \vdash_{\mathfrak{A}} \mathtt{if}\ x = y\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 : \mathcal{T}_2 \text{ calls } \mathcal{S}_2}$$

$$\textsc{let}\ \frac{\begin{array}{c}\Gamma \vdash_{\mathfrak{A}} e_1 : r_1 \,\&\, \mathcal{U}_1 \mid \ldots \mid r_n \,\&\, \mathcal{U}_n \text{ calls } \mathcal{S} \\ \Gamma,\ x{:}\,r_i \vdash_{\mathfrak{A}} e_2 : \mathcal{T}_i \text{ calls } \mathcal{S}_i\end{array}}{\Gamma \vdash_{\mathfrak{A}} \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 : \bigsqcup_{i=1}^n \mathcal{U}_i \cdot \mathcal{T}_i \text{ calls } \mathcal{S} \sqcup \bigsqcup_{i=1}^n \mathcal{U}_i \cdot \mathcal{S}_i}$$

$$\textsc{null}\ \frac{}{\Gamma \vdash_{\mathfrak{A}} \mathtt{null} : \mathtt{Null} \,\&\, \alpha_*(\{\varepsilon\}) \text{ calls } \emptyset}$$

$$\textsc{new}\ \frac{}{\Gamma \vdash_{\mathfrak{A}} \mathtt{new}^{\ell}\ C : \mathtt{CreatedAt}(\ell) \,\&\, \alpha_*(\{\varepsilon\}) \text{ calls } \emptyset}$$

$$\textsc{cast}\ \frac{\Gamma \vdash_{\mathfrak{A}} e : \mathcal{T} \text{ calls } \mathcal{S}}{\Gamma \vdash_{\mathfrak{A}} (D)\, e : \mathcal{T} \text{ calls } \mathcal{S}}$$

$$\textsc{get}\ \frac{\mathcal{T} = \bigsqcup\{s \,\&\, \alpha_*(\{\varepsilon\}) \mid s \in F(C, r, f)\}}{\Gamma,\ x{:}\,r \vdash_{\mathfrak{A}} x^C.f : \mathcal{T} \text{ calls } \emptyset}$$

$$\textsc{set}\ \frac{s \in F(C, r, f)}{\Gamma,\ x{:}\,r,\ y{:}\,s \vdash_{\mathfrak{A}} x^C.f := y : s \,\&\, \alpha_*(\{\varepsilon\}) \text{ calls } \emptyset}$$

$$\textsc{call}\ \frac{(\mathcal{T}, \_) = M_{\mathfrak{A}}(C, r, m, \bar{s})}{\Gamma,\ x{:}\,r,\ \bar{y}{:}\,\bar{s} \vdash_{\mathfrak{A}} x^C.m(\bar{y}) : \mathcal{T} \text{ calls } (C, r, m, \bar{s})}$$

Figure 2: Region-based Büchi effect type system

Given a Büchi method typing $M_{\mathfrak{A}}$, we obtain a method typing $M_{\mathfrak{A}}^{\gamma}$ with concretized effects by defining for $\delta \in \mathrm{dom}(M_{\mathfrak{A}})$ with $M_{\mathfrak{A}}(\delta) = (\mathcal{T}, \mathcal{S})$:

$$M_{\mathfrak{A}}^{\gamma}(\delta) = (\gamma(\mathcal{T}), \gamma(\mathcal{S})).$$

We relate the Büchi type system to the type system introduced in Section 4 as follows.

Lemma 5.5. *If $\Gamma \vdash_{\mathfrak{A}} e : \mathcal{T}$ calls $\mathcal{S}$ is derivable w.r.t. the Büchi class table $(F, M_{\mathfrak{A}})$, then so is $\Gamma \vdash e : \gamma(\mathcal{T})$ calls $\gamma(\mathcal{S})$ w.r.t. the corresponding class table $(F, M_{\mathfrak{A}}^{\gamma})$.*

The following is a direct consequence of Lemma 5.5.

Theorem 5.6. *If a program P is well-typed w.r.t. a Büchi class table $(F, M_{\mathfrak{A}})$, then it is well-typed w.r.t. the corresponding class table $(F, M_{\mathfrak{A}}^{\gamma})$.*

Moreover, we can compute the infinitary effect typing over the Büchi abstraction.

Theorem 5.7. *Given a Büchi method typing $M_{\mathfrak{A}}$, we can compute directly an assignment $\eta_{\mathfrak{A}} : Sig \to \mathcal{M}_{\leq\omega}$ that abstracts the infinitary effect typing $\eta_{\gamma} : Sig \to \mathcal{P}(\Sigma^{\leq\omega})$ for $M_{\mathfrak{A}}^{\gamma}$,* i.e., *for all $\delta \in Sig$ we have $\eta_{\mathfrak{A}}(\delta) = \alpha_{\leq\omega}(\eta_{\gamma}(\delta))$ where η_{γ} is the greatest solution of $\{\delta = \gamma(\mathcal{S}_{\delta}) \mid \delta \in Sig\}$ with $\mathcal{S}_{\delta}$ the call expression in $M_{\mathfrak{A}}(\delta)$. We call such an assignment $\eta_{\mathfrak{A}}$ a* Büchi infinitary effect typing.

Proof. Let $\mathcal{S}_{\delta}$ be the call expressions in $M_{\mathfrak{A}}(\delta)$. We solve the equation system $\{\delta = \mathcal{S}_{\delta} \mid \delta \in Sig\}$ as follows: Recall that each expression has the form $\bigsqcup_{\delta\in\Delta} \mathcal{A}_{\delta} \cdot \delta$ for some $\Delta \subseteq Sig$. Viewing as an $\mathcal{M}_{\leq\omega}$-valued function, each $\mathcal{S}_{\delta}$ can be rewritten to an equivalent form $\mathcal{A}_{\delta} \cdot \delta \sqcup \mathcal{F}_{\delta}$ where the function $\mathcal{F}_{\delta}$ does not contain δ, by the fact that the join operation is commutative. We repeat the following steps for each $\delta \in Sig$:

(1) Rewrite the right-hand side of the equation for δ to the form $\mathcal{A}_{\delta} \cdot \delta \sqcup \mathcal{F}_{\delta}$.
(2) Set $\delta = \mathcal{A}_{\delta}^{*} \cdot \mathcal{F}_{\delta} \sqcup \mathcal{A}_{\delta}^{\omega}$.
(3) Substitute δ by $\mathcal{A}_{\delta}^{*} \cdot \mathcal{F}_{\delta} \sqcup \mathcal{A}_{\delta}^{\omega}$ on the right-hand side of all the other equations.

One variable is eliminated on the right-hand side of the equations after each iteration. In the end we solve the system and obtain an assignment function $\eta_{\mathfrak{A}} : Sig \to \mathcal{M}_{\leq\omega}$.

It abstracts the greatest solution η_{γ} of $\{\delta = \gamma(\mathcal{S}_{\delta}) \mid \delta \in Sig\}$ where the equations are concretized. For instance, consider a singleton equation system $\{X = \mathcal{A} \cdot X \sqcup \mathcal{B}\}$. Its solution is

$$\eta_{\mathfrak{A}}(X) = \mathcal{A}^{*} \cdot \mathcal{B} \sqcup \mathcal{A}^{\omega}$$

according to the above algorithm. The greatest solution of its concretization $\{X = \gamma_{*}(\mathcal{A}) \cdot X \cup \gamma_{\leq\omega}(\mathcal{B})\}$ is $\eta_{\gamma}(X) = \gamma_{*}(\mathcal{A})^{*} \cdot \gamma_{\leq\omega}(\mathcal{B}) \cup \gamma_{*}(\mathcal{A})^{\omega}$. It is not hard to prove $\eta_{\mathfrak{A}}(X) = \alpha_{\leq\omega}(\eta_{\gamma}(X))$ using Proposition 5.3. One can easily generalize the proof to systems with multiple equations. We refer the reader to *e.g.* [9] for the characterization of simultaneous fixed points. □

Remark 5.8. One may expect a Büchi infinitary effect typing to be the greatest solution of $\{\delta = \mathcal{S}_{\delta} \mid \delta \in Sig\}$ without concretization. However, the Büchi abstraction may not preserve greatest fixed points, which may lead to a loss in precision. Consider the example of a single method `f` with body `emit`(a); `this.f()`. The equation system is $\{\delta_{\mathtt{f}} = \alpha_{*}(\{a\}) \cdot \delta_{\mathtt{f}}\}$. If we solve it in the Büchi abstraction from Example 5.4, then we get the solution $\eta(\delta_{\mathtt{f}}) = a^{+} \cup a^{\omega}$, as noted there. However, if we concretize this equation and solve over sets, then we get the solution $\eta_{\gamma}(\delta_{\mathtt{f}}) = a^{\omega}$. Therefore, we choose a better approximation of the infinite traces and formulate the Büchi infinitary effect typing as the abstraction of the greatest fixed point of the concretized call expressions.

The soundness of the Büchi type system follows from the soundness of the previous type system (Theorem 4.8) by the properties of abstraction and concretion.

Corollary 5.9 (Soundness). *Suppose that a program is well-typed w.r.t. $(F, M_{\mathfrak{A}})$ and that $\eta_{\mathfrak{A}}$ is a Büchi infinitary effect typing for $M_{\mathfrak{A}}$. Then, for any $(C, r, m, \bar{s}) \in Sig$ with $M_{\mathfrak{A}}(C, r, m, \bar{s}) = (\mathcal{T}, \mathcal{S})$ and $\eta_{\mathfrak{A}}(C, r, m, \bar{s}) = \mathcal{V}$, and for any $x : r$ and $\bar{y} : \bar{s}$, we have*

(1) if $x.m(\bar{y})$ evaluates to a value v and generates a trace w, then v is in region t and $w \in \gamma_{*}(\mathcal{U})$ for some $(t \,\&\, \mathcal{U}) \in \mathcal{T}$;
(2) if $x.m(\bar{y})$ diverges and generates a trace w, then $w \in \gamma_{\leq\omega}(\mathcal{V})$.

To certify that all the traces of a given method are allowed by the language $\mathfrak{A}$, we only need to check if all its effects stored in the typings $M_{\mathfrak{A}}$ and $\eta_{\mathfrak{A}}$ are below the abstraction of $\mathfrak{A}$.

Let us spell out directly what this theorem means if we use the Büchi abstraction constructed from an extended Büchi automaton using the construction of Hofmann and Chen [13]. In this case, $\mathfrak{A}$ is the language of the automaton. The Büchi abstraction of the automaton is such that the automaton accepts a language $V \subseteq \Sigma^{\leq\omega}$ if and only if $\alpha_{\leq\omega}(V)$ is below the abstraction of $\mathfrak{A}$. This follows from [13, Lemma 3(c)]. Since we have defined a Büchi infinitary effect typing $\eta_{\mathfrak{A}}$ as the abstraction of an infinitary effect typing η_{γ}, this means that checking that $\eta_{\mathfrak{A}}(\delta)$ is below the abstraction of $\mathfrak{A}$ amounts to checking that the extended Büchi automaton accepts all words in $\eta_{\gamma}(\delta)$.

Come back to the example of a server given in the Introduction. Let $\mathfrak{A}$ be the guideline consisting of traces where each event *access* is immediately preceded by *authcheck*. It can be represented by a Büchi automaton. Based on the automaton, we employ the construction of Hofmann and Chen [12] to get a Büchi abstraction. Checking whether the non-terminating effect of `serve` is below $\mathfrak{A}$ amounts to checking whether it is accepted by the Büchi automaton. In this example, this will be the case and `serve` is verified to adhere to the guideline.

5.3 Type Inference Algorithm

Given a program P that is well-typed w.r.t. a given standard FJ class table (F_0, M_0), we extend the inference algorithm of [5, Appendix F] to construct a Büchi class table $(F, M_{\mathfrak{A}})$ with respect to which P is well-typed.

We follow the typing rules to construct a procedure `typeff` that maps an environment $\Gamma : Var \rightharpoonup Reg$ and a term $e \in Expr$ to formal expressions $\mathcal{T} \in \mathcal{M}_{*}\langle Reg \rangle$ and $\mathcal{S} \in \mathcal{M}_{*}\langle Sig \rangle$, if e is typable in Γ, such that $\Gamma \vdash_{\mathfrak{A}} e : \mathcal{T}$ calls $\mathcal{S}$ is derivable. In particular, `typeff` may update the field typing F, which happens in the case of a field-set statement $x^{C}.f = y$. The SET rule requires that the region of y must be in those allowed for the field f by F. If this condition is not satisfied, then `typeff` updates F by joining the offending entry with the region of y. We then use `typeff` to construct a Büchi class table. Rule WEAK does not have to be applied, but it can be used to trade precision of analysis for efficiency of the inference algorithm. See the discussion in Section 8.

In brief, the class table is computed as a least fixed point using the `typeff` procedure. We start with a table $(F, M_{\mathfrak{A}})$ with the possibly "lowest" entries, *i.e.*, all the entries of F are $\{\texttt{Null}\}$ and those of $M_{\mathfrak{Y}}$ are the empty expression $\emptyset$. For each method signature, we "increase" the corresponding entry in $M_{\mathfrak{Y}}$ if it differs from the result given by calling `typeff` with the method body. Note that `typeff` may update F as discussed above. Then we ensure that the table is well-formed. We repeat this until the table cannot be updated further. Because there are only finitely many regions and abstract effects, we will reach the fixed point after some iterations. The resulting table $(F, M_{\mathfrak{Y}})$ encodes the terminating effects and the call expressions. More details of the type inference algorithm are available in Appendix B

Theorem 5.10. *Let P be a program that is well-typed w.r.t. some standard FJ class table (F_0, M_0). Then the above algorithm gives a Büchi class table $(F, M_{\mathfrak{Y}})$ with respect to which P is well-typed.*

Then we employ the algorithm in the proof of Theorem 5.7 to compute a typing $\eta_{\mathfrak{Y}}$ of nonterminating effects.

6 EXPERIMENTAL EVALUATION

We have a prototype implementation[2] of the type inference algorithm (Section 5.3) using the Soot framework [22]. In the implementation, we extend the approach explained in this paper to support exception handling. But other language features such as reflection and concurrency are not covered yet. Our tool takes a Java bytecode program and a programming guideline as inputs. The guideline is represented as an automaton. Using Hofmann and Chen's construction [12], a Büchi abstraction for representing effects is generated from the automaton. The guideline includes a configuration specifying the default effects of intrinsic methods. For instance, the `verifyAuthorization` method in the `serve` example would be specified to have an effect representing *authcheck*. From the standard Java typing provided by the Java bytecode, our tool infers the effect information of the program and then verifies if it is accepted by the guideline automaton. If the inferred effect is not acceptable, the tool tries to report a counterexample, *i.e.*, searching for an execution path of the program that violates the guideline. Such an execution path can help programers to identify the bug(s) of the program.

We tested and evaluated our implementation with the Securibench Micro benchmark[3] as well as a number of additional examples. The benchmark provides 122 test cases in 12 categories. Each benchmark program implements a small self-contained servlet. We ran our tool to verify if they adhere to the guideline that only untainted commands can be executed. In addition to the examples of the paper, we implemented 19 programs with typical nonterminating behaviors and 10 programs with exception handling for testing the infinitary and exceptional effect analysis. For these programs, we tested if our tool computes the correct effects. We ran the experiment on a machine with a 3,5 GHz Dual-Core Intel Core i7 processor. The analysis in each test takes around 200ms on average, and the slowest one takes almost a second.

The results of the experiment are given in Table 1, where the results of the 12 categories of Securibench Micro are listed first, followed by the results of our three groups of test cases. A cell n/m of the "Pass" column means that there are totally m test cases and n of them run as expected. When a test fails, a reference to the reason is given in the "Comments" column. Here are the reasons:

(1) We assume that if a tainted element is added into an array then the array is tainted (*i.e.*, all its elements are tainted), which can cause false positives.
(2) Our analysis is not fully path-sensitive.
(3) Our field updates are conservatively treated as weak updates. For example, if a field $x.f$ has a region type $\{T\}$ indicating that it is tainted, and later it is updated to a untainted value, then the type of $x.f$ becomes $\{T, U\}$ where U is the region for untainted values. This sometimes leads to false positives in taint analysis.
(4) Static initialization is not supported.
(5) Reflection is not supported.
(6) Concurrent features are not supported.

The "CE Report" column contains the results of searching counterexamples. A cell i/j of it expresses that there are j programs violating the guideline, and for i of them an execution path that leads to the violation is found.

Test Category	Pass	CE Report	Comments
aliasing	6/6	5/5	
arrays	9/10	9/9	(1)
basic	42/42	42/42	
collections	14/14	13/13	
data structures	6/6	4/5	
factories	3/3	3/3	
inter	12/14	12/12	(4)
pred	6/9	5/5	(2)
reflection	0/4	0/0	(5)
sanitizers	6/6	3/3	
session	3/3	3/3	
strong updates	3/5	1/1	(3), (6)
paper examples	8/8	1/1	
infinitary	19/19	0/0	
exceptions	10/10	0/0	
total	145/157	101/102	

Table 1: Results of the experiment

7 PREVIOUS AND RELATED WORK

We work along the line of typing Java programs with regions and effects [2, 5, 11]. Instead of refining the standard Featherweight Java type system as in the previous work, we introduce a flow type system where class information from the standard type system can be omitted. Our approach to region typing is in the spirit of Microsoft's TypeScript [19] and Facebook's Flow [6]. Moving to a flow type system has the benefit of simplifying the type system and making the choice of regions more flexible. Our system is similar to the session type and effect system for PCF studied in [21]. But ours can capture also effects of non-terminating programs.

The idea of a type system with Büchi effects for Java has been sketched in [12, 13]. But its meta-theoretic properties like soundness

[2]The implementation is available at https://github.com/cj-xu/GuideForceJava.
[3]The website of Securibench Micro: http://too4words.github.io/securibench-micro/

and the correctness of type inference were not investigated. Here we fully develop this idea into to a new flow type system, and establish the meta-theory. Furthermore, we make the type system independent from the automata-theoretic constructions of [12, 13] by capturing the necessary structure abstractly in terms of a Büchi abstraction. Büchi automata have been employed also in model checking and testing for infinite executions of programs [1, 14, 29]. Our approach is entirely based on type systems and abstract interpretation.

Regions serve as the basis of various techniques for *e.g.* memory management [3, 27], pointer analysis [2, 28], and race detection [23]. They are closely interrelated with effects. Thus the use of regions makes our type-based approach of analysis context-sensitive. In higher-order model checking [16–18, 26], intersection types are employed instead to achieve the context-sensitivity.

Our approach is closely related to the one of Nanjo et al. [20]. They work with a type system where effects are represented by formulas of some fixpoint logic. Specifically, their formulas may contain least and greatest fixpoints to specify the finite and infinite behavior of the program, which is highly similar to what our call expressions do (see Section 4.2). Their effects are value-dependent, because the formulas can contain program values. This makes their analysis path-sensitive. In our setting, effects can depend on regions which are a notion of abstracted value. In the computation of infinitary effects (Section 4.4), we use an equation system with method signatures as variables. Notice that method signatures come with a region for each argument. Therefore, we have a restricted form of dependent temporal effect. To capture more precise dependencies, it would be possible to refine the notion of region, perhaps even to refinement types [8].

Skalka *et al.* [24, 25] also work with effect type systems to verify trace-based safety properties. But they apply model-checking tools to trace effects inferred by the type systems for the verification. In their setting, trace effects are represented as label transition systems (LTSs). It seems that the equation systems that we use for approximating infinite traces can be considered as a form of LTS, if one orients the equations, that plays the same role as the LTSs in [24, 25]. However, in our case the variables in the equation system (*i.e.*, method signatures) depend on regions, which improves the precision of analysis. The effect type system of [24] is an refinement of the FJ type system. Ours is simpler because it is separate from the FJ type sytem.

8 CONCLUSION AND DISCUSSION

This paper presents a framework to verify if a given Featherweight Java program adheres to a given programming guideline. Let $\mathfrak{A}$ be a language representing the guideline. Suppose a Büchi abstraction that is faithful w.r.t. $\mathfrak{A}$ is given, *e.g.* the one of Hofmann and Chen [13]. From the standard Java types of the fields and methods, we construct a Büchi class table $(F, M_{\mathfrak{A}})$ with respect to which the program is well-typed. The table $M_{\mathfrak{A}}$ and its infinitary effect typing $\eta_{\mathfrak{A}}$ provide the terminating and non-terminating effects to each method signature. If the effects of the methods are below the abstraction of $\mathfrak{A}$, then any event trace generated by the program is allowed by $\mathfrak{A}$ and thus the program adheres to the guideline.

Comparing to the previous work [5, 12], our approach enhances the precision of analysis by capturing the effect for each region where a value may locate. The cost of this improvement is that the type inference is less efficient. For instance, to compute the effect of the expression $\texttt{let}\ x = e_1\ \texttt{in}\ e_2$, we need to infer e_2 as many times as the number of possible regions where the value of e_1 may locate, while in the systems of [5, 12] e_2 is inferred only once. However, our framework is actually very flexible. If we apply the WEAK rule in every step of the typing derivation, then we essentially work with an effect type system without regions, because in the end only the region Unknown remains in the result and all effects of a value are merged. If we extend regions with a join operation $\texttt{Union}(r_1, r_2)$ and a corresponding lattice structure, and change the WEAK rule to allow merging $r_1 \,\&\, U_1 \mid r_2 \,\&\, U_2$ to $\texttt{Union}(r_1, r_2) \mid (U_1 \cup U_2)$, then we can recover the systems of [5, 12]. In addition, we can reduce the size of the method table in an implementation by letting $M(C, \texttt{Union}(r_1, r_2), m, \bar{s}) = M(C, r_1, m, \bar{s}) \sqcup M(C, r_2, m, \bar{s})$ and storing only the latter two entries. Moreover, one may come up with certain strategy about when to apply the WEAK rule in the inference algorithm to balance efficiency and precision. It is our future work to investigate such directions.

REFERENCES

[1] Damián Adalid, Alberto Salmerón, María del Mar Gallardo, and Pedro Merino. 2014. Using SPIN for automated debugging of infinite executions of Java programs. *Journal of Systems and Software* 90 (2014), 61–75. https://doi.org/10.1016/j.jss.2013.10.056

[2] Lennart Beringer, Robert Grabowski, and Martin Hofmann. 2013. Verifying pointer and string analyses with region type systems. *Computer Languages, Systems & Structures* 39, 2 (2013), 49–65. https://doi.org/10.1016/j.cl.2013.01.001

[3] Sigmund Cherem and Radu Rugina. 2004. Region Analysis and Transformation for Java Programs. In *Proceedings of the 4th International Symposium on Memory Management (ISMM '04)*. Association for Computing Machinery, New York, NY, USA, 85–96. https://doi.org/10.1145/1029873.1029884

[4] Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Principles of Programming Languages (POPL 1977)*, Robert M. Graham, Michael A. Harrison, and Ravi Sethi (Eds.). 238–252. https://doi.org/10.1145/512950.512973

[5] Serdar Erbatur, Martin Hofmann, and Eugen Zălinescu. 2017. Enforcing Programming Guidelines with Region Types and Effects. In *Programming Languages and Systems (APLAS 2017) (Lecture Notes in Computer Science)*, Bor-Yuh Evan Chang (Ed.), Vol. 10695. Springer, Cham, 85–104. https://doi.org/10.1007/978-3-319-71237-6_5

[6] Facebook. [n.d.]. Flow - A static type checker for JavaScript. URL: https://flow.org.

[7] Facebook. [n.d.]. infer. https://fbinfer.com/

[8] Tim Freeman and Frank Pfenning. 1991. Refinement Types for ML. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation (PLDI '91)*. Association for Computing Machinery, New York, NY, USA, 268–277. https://doi.org/10.1145/113445.113468

[9] Carsten Fritz. 2002. Some Fixed Point Basics. In *Automata Logics, and Infinite Games*, Erich Grädel, Wolfgang Thomas, and Thomas Wilke (Eds.). Lecture Notes in Computer Science, Vol. 2500. Springer, Berlin, Heidelberg, 359–364. https://doi.org/10.1007/3-540-36387-4_20

[10] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. 2020. *The Java Language Specification* (Java SE 14 ed.). Oracle America, Inc. http://docs.oracle.com/javase/specs/jls/se14/jls14.pdf

[11] Robert Grabowski, Martin Hofmann, and Keqin Li. 2012. Type-Based Enforcement of Secure Programming Guidelines — Code Injection Prevention at SAP. In *Formal Aspects of Security and Trust (FAST 2011) (Lecture Notes in Computer Science)*, G. Barthe, A. Datta, and S. Etalle (Eds.), Vol. 7140. Springer, Berlin, Heidelberg, 182–197. https://doi.org/10.1007/978-3-642-29420-4_12

[12] Martin Hofmann and Wei Chen. 2014. Abstract Interpretation from Büchi Automata. In *Computer Science Logic and Logic in Computer Science (CSL-LICS 2014)*. Association for Computing Machinery, 51:1–51:10. https://doi.org/10.1145/2603088.2603127

[13] Martin Hofmann and Wei Chen. 2014. Büchi Types for Infinite Traces and Liveness. Technical report, arXiv:1401.5107 [cs.LO].

[14] Gerard J Holzmann. 2003. *The SPIN Model Checker: Primer and Reference Manual.* Addison-Wesley Professional, Boston, MA.

[15] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. 2001. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems* 23, 3 (2001), 396–450. https://doi.org/10.1145/503502.503505

[16] Naoki Kobayashi. 2013. Model Checking Higher-Order Programs. *Journal of the ACM* 60, 3 (2013), 20:1–20:62. https://doi.org/10.1145/2487241.2487246

[17] Naoki Kobayashi and Xin Li. 2015. Automata-Based Abstraction Refinement for μHORS Model Checking. In *2015 30th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS'15).* IEEE Computer Society, 713–724. https://doi.org/10.1109/LICS.2015.71

[18] Naoki Kobayashi and C.-H. Luke Ong. 2009. A Type System Equivalent to the Modal Mu-Calculus Model Checking of Higher-Order Recursion Schemes. In *2009 24th Annual IEEE Symposium on Logic In Computer Science (LICS'09).* IEEE Computer Society, 179–188. https://doi.org/10.1109/LICS.2009.29

[19] Microsoft. [n.d.]. TypeScript - Typed JavaScript at any scale. URL: https://www.typescriptlang.org/.

[20] Yoji Nanjo, Hiroshi Unno, Eric Koskinen, and Tachio Terauchi. 2018. A Fixpoint Logic and Dependent Effects for Temporal Property Verification. In *2018 33rd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS'18).* Association for Computing Machinery, 759–768. https://doi.org/10.1145/3209108.3209204

[21] Dominic Orchard and Nobuko Yoshida. 2016. Effects as Sessions, Sessions as Effects. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'16).* Association for Computing Machinery, 568–581. https://doi.org/10.1145/2837614.2837634

[22] McGill University Sable Group. [n.d.]. Soot - A framework for analyzing and transforming Java and Android applications. URL: https://soot-oss.github.io/soot/.

[23] Helmut Seidl and Vesal Vojdani. 2009. Region Analysis for Race Detection. In *Proceedings of the 16th International Symposium on Static Analysis (SAS '09).* Springer-Verlag, Berlin, Heidelberg, 171–187. https://doi.org/10.1007/978-3-642-03237-0_13

[24] Christian Skalka. 2008. Types and trace effects for object orientation. *Higher-Order and Symbolic Computation* 21 (2008), 239–282. https://doi.org/10.1007/s10990-008-9032-6

[25] Christian Skalka, Scott Smith, and David Van Horn. 2008. Types and Trace Effects of Higher Order Programs. *Journal of Functional Programming* 18, 2 (2008), 179–249. https://doi.org/10.1017/S0956796807006466

[26] Ryota Suzuki, Koichi Fujima, Naoki Kobayashi, and Takeshi Tsukada. 2017. Streett Automata Model Checking of Higher-Order Recursion Schemes. In *2nd International Conference on Formal Structures for Computation and Deduction (FSCD'2017) (Leibniz International Proceedings in Informatics (LIPIcs))*, Dale Miller (Ed.), Vol. 84. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 32:1–32:18. https://doi.org/10.4230/LIPIcs.FSCD.2017.32

[27] Mads Tofte and Jean-Pierre Talpin. 1997. Region-Based Memory Management. *Information and Computation* 132, 2 (1997), 109–176. https://doi.org/10.1006/inco.1996.2613

[28] Sen Ye, Yulei Sui, and Jingling Xue. 2014. Region-Based Selective Flow-Sensitive Pointer Analysis. In *Static Analysis (SAS 2014) (Lecture Notes in Computer Science)*, M. Müller-Olm and H. Seidl (Eds.), Vol. 8723. Springer, Cham, 319–336. https://doi.org/10.1007/978-3-319-10936-7_20

[29] Bolong Zeng and Li Tan. 2016. Test Reactive Systems with Büchi-Automaton-Based Temporal Requirements. In *Theoretical Information Reuse and Integration (Advances in Intelligent Systems and Computing)*, T. Bouabana-Tebibel and S. Rubin (Eds.), Vol. 446. Springer, Cham, 31–57. https://doi.org/10.1007/978-3-319-31311-5_2

A OPERATIONAL SEMANTICS RULES

We define an auxiliary function $\mathrm{classOf}_h(v)$ to determine the type of value v in heap h as follows:

$$\mathrm{classOf}_h(v) := \begin{cases} \texttt{NullType} & \text{if } v = \mathit{null} \\ C & \text{if } h(v) = (C, \_, \_) \in \mathit{Obj} \end{cases}$$

The operational semantics $(s, h) \vdash e \Downarrow v, h' \mathbin{\&} w$ of terminating evaluation is defined inductively from the following inference rules:

$$\frac{}{(s,h) \vdash \texttt{emit}(a) \Downarrow \mathit{null}, h \mathbin{\&} a}$$

$$\frac{}{(s,h) \vdash x \Downarrow s(x), h \mathbin{\&} \varepsilon} \qquad \frac{}{(s,h) \vdash \texttt{null} \Downarrow \mathit{null}, h \mathbin{\&} \varepsilon}$$

$$\frac{s(x) = s(y) \qquad (s,h) \vdash e_1 \Downarrow v, h' \mathbin{\&} w}{(s,h) \vdash \texttt{if } x = y \texttt{ then } e_1 \texttt{ else } e_2 \Downarrow v, h' \mathbin{\&} w}$$

$$\frac{s(x) \neq s(y) \qquad (s,h) \vdash e_2 \Downarrow v, h' \mathbin{\&} w}{(s,h) \vdash \texttt{if } x = y \texttt{ then } e_1 \texttt{ else } e_2 \Downarrow v, h' \mathbin{\&} w}$$

$$\frac{(s,h) \vdash e_1 \Downarrow v_1, h_1 \mathbin{\&} w_1 \qquad (s[x \mapsto v_1], h_1) \vdash e_2 \Downarrow v_2, h_2 \mathbin{\&} w_2}{(s,h) \vdash \texttt{let } x = e_1 \texttt{ in } e_2 \Downarrow v_2, h_2 \mathbin{\&} w_1 w_2}$$

$$\frac{l \notin \mathrm{dom}(h) \qquad G = [f \mapsto \mathit{null}]_{f \in \mathit{fields}(C)}}{(s,h) \vdash \texttt{new}^{\ell}\ C \Downarrow l, h[l \mapsto (C, G, \ell)] \mathbin{\&} \varepsilon}$$

$$\frac{(s,h) \vdash e \Downarrow v, h' \mathbin{\&} w \qquad \mathrm{classOf}_{h'}(v) \leq C}{(s,h) \vdash (C)\, e \Downarrow v, h' \mathbin{\&} w}$$

$$\frac{s(x) = l \qquad h(l) = (\_, G, \_)}{(s,h) \vdash x.f \Downarrow G(f), h \mathbin{\&} \varepsilon}$$

$$\frac{s(x) = l \qquad h(l) = (D, G, \ell) \qquad h' = h[l \mapsto (D, G[f \mapsto s(y)], \ell)]}{(s,h) \vdash x.f := y \Downarrow s(y), h' \mathbin{\&} \varepsilon}$$

$$\frac{\begin{array}{c} s(x) = l \qquad h(l) = (D, \_, \_) \qquad \mathit{mtable}(D, m) = (\bar{z}, e) \\ ([\texttt{this} \mapsto l] \cup [z_i \mapsto s(y_i)]_{i \in \{1, \ldots, |\bar{z}|\}}, h) \vdash e \Downarrow v, h' \mathbin{\&} w \end{array}}{(s,h) \vdash x.m(\bar{y}) \Downarrow v, h' \mathbin{\&} w}$$

The operational semantics of divergence $(s, h) \vdash e \Uparrow \mathbin{\&} w$ is defined coinductively from the following inference rules, where double horizontal lines are used:

$$\frac{s(x) = s(y) \qquad (s,h) \vdash e_1 \Uparrow \mathbin{\&} w}{(s,h) \vdash \texttt{if } x = y \texttt{ then } e_1 \texttt{ else } e_2 \Uparrow \mathbin{\&} w}$$

$$\frac{s(x) \neq s(y) \qquad (s,h) \vdash e_2 \Uparrow \mathbin{\&} w}{(s,h) \vdash \texttt{if } x = y \texttt{ then } e_1 \texttt{ else } e_2 \Uparrow \mathbin{\&} w}$$

$$\frac{(s,h) \vdash e_1 \Downarrow v_1, h_1 \mathbin{\&} w_1 \qquad (s[x \mapsto v_1], h_1) \vdash e_2 \Uparrow \mathbin{\&} w_2}{(s,h) \vdash \texttt{let } x = e_1 \texttt{ in } e_2 \Uparrow \mathbin{\&} w_1 w_2}$$

$$\frac{(s,h) \vdash e_1 \Uparrow \mathbin{\&} w}{(s,h) \vdash \texttt{let } x = e_1 \texttt{ in } e_2 \Uparrow \mathbin{\&} w}$$

$$\frac{(s,h) \vdash e \Uparrow \mathbin{\&} w}{(s,h) \vdash (C)\, e \Uparrow \mathbin{\&} w}$$

$$\frac{\begin{array}{c} s(x) = l \qquad h(l) = (D, \_, \_) \qquad \mathit{mtable}(D, m) = (\bar{z}, e) \\ ([\texttt{this} \mapsto l] \cup [z_i \mapsto s(y_i)]_{i \in \{1, \ldots, |\bar{z}|\}}, h) \vdash e \Uparrow \mathbin{\&} w \end{array}}{(s,h) \vdash x.m(\bar{y}) \Uparrow \mathbin{\&} w}$$

```
proc $\mathcal{A}(P)$
  $(F, M_{\mathfrak{Y}}) \leftarrow \texttt{init}(P)$
  do
    $(F', M') \leftarrow (F, M_{\mathfrak{Y}})$
    foreach $(C, r, m, \bar{s}) \in \mathrm{dom}(M_{\mathfrak{Y}})$ with $C \in Cls(r)$
                                                  and $(C, m) \in \mathrm{dom}(mtable)$
      $(\bar{x}, e) \leftarrow mtable(C, m)$
      $\Gamma \leftarrow [\texttt{this} \mapsto r] \cup [x_i \mapsto s_i]_{i \in \{1,\ldots,|\bar{x}|\}}$
      $(\mathcal{T}, \mathcal{S}) \leftarrow \texttt{typeff}(\Gamma, e)$
      $M_{\mathfrak{Y}}(C, r, m, \bar{s}) \leftarrow M_{\mathfrak{Y}}(C, r, m, \bar{s}) \sqcup (\mathcal{T}, \mathcal{S})$
      $(F, M_{\mathfrak{Y}}) \leftarrow \texttt{checkClassTable}(F, M_{\mathfrak{Y}})$
  until $(F', M') = (F, M_{\mathfrak{Y}})$
  return $(F, M_{\mathfrak{Y}})$

proc $\texttt{init}(P)$
  $Cls(\texttt{Unknown}) \leftarrow Cls$
  $Cls(\texttt{Null}) \leftarrow \texttt{NullType}$
  foreach $(\texttt{new}^{\ell}\ C) \in P$
    $Cls(\texttt{CreatedAt}(\ell)) \leftarrow Cls(\texttt{CreatedAt}(\ell)) \cup \{C\}$
  foreach $C \in Cls, r \in Reg$
    foreach $f \in fields(C)$
      $F(C, r, f) \leftarrow \{\texttt{Null}\}$
    foreach $m \in methods(C), \bar{s} \in Reg^{\mathrm{ar}(m)}$
      $M_{\mathfrak{Y}}(C, r, m, \bar{s}) \leftarrow (\emptyset, \emptyset)$
  return $(F, M_{\mathfrak{Y}})$

proc $\texttt{typeff}(\Gamma, e)$
  match $e$ with
  | $\texttt{emit}(a) \rightarrow$ return $((\texttt{Null} \,\&\, \alpha_*(\{a\})), \emptyset)$
  | $x \rightarrow$ return $((\Gamma(x) \,\&\, \alpha_*(\{\varepsilon\})), \emptyset)$
  | $\texttt{if}\ x = y\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \rightarrow$
    if $disjoint\,(\Gamma(x), \Gamma(y))$ then return $\texttt{typeff}(\Gamma, e_2)$
    else return $\texttt{typeff}(\Gamma, e_1) \sqcup \texttt{typeff}(\Gamma, e_2)$
  | $\texttt{let}\ x = e_1\ \texttt{in}\ e_2 \rightarrow$
    $((r_1 \,\&\, \mathcal{U}_1 \mid \ldots \mid r_n \,\&\, \mathcal{U}_n), \mathcal{S}) \leftarrow \texttt{typeff}(\Gamma, e_1)$
    foreach $i \in \{1, \ldots, n\}$
      $(\mathcal{T}_i, \mathcal{S}_i) \leftarrow \texttt{typeff}(\Gamma[x \mapsto r_i], e_2)$
    return $(\bigsqcup_{i=1}^{n} \mathcal{U}_i \cdot \mathcal{T}_i, \mathcal{S} \sqcup \bigsqcup_{i=1}^{n} \mathcal{U}_i \cdot \mathcal{S}_i)$
  | $\texttt{null} \rightarrow$ return $((\texttt{Null} \,\&\, \alpha_*(\{\varepsilon\})), \emptyset)$
  | $\texttt{new}^{\ell}\ C \rightarrow$ return $((\texttt{CreatedAt}(\ell) \,\&\, \alpha_*(\{\varepsilon\})), \emptyset)$
  | $(D)\ e \rightarrow$ return $\texttt{typeff}(\Gamma, e)$
  | $x^{C}.f \rightarrow$ return $(\bigsqcup\{s \,\&\, \alpha_*(\{\varepsilon\}) | s \in F(C, \Gamma(x), f)\}, \emptyset)$
  | $x^{C}.f := y \rightarrow$
    $F(C, \Gamma(x), f) \leftarrow F(C, \Gamma(x), f) \cup \{\Gamma(y)\}$
    return $((\Gamma(y) \,\&\, \alpha_*(\{\varepsilon\})), \emptyset)$
  | $x^{C}.m(\bar{y}) \rightarrow$
    $(\mathcal{T}, \_) \leftarrow M_{\mathfrak{Y}}(C, \Gamma(x), m, \Gamma(\bar{y}))$
    return $(\mathcal{T}, (C, \Gamma(x), m, \Gamma(\bar{y})))$
```

Figure 3: Pseudo-code of the type inference algorithm

B DETAILED TYPE INFERENCE ALGORITHM

We present the details of the type inference algorithm that was briefly discussed in Section 5.3. Our algorithm extends the one in [5, Appendix F] to capture the call expressions of programs for computing infinitary effects. In this section, we assume that a Büchi abstraction is given.

Our type inference algorithm, denoted by $\mathcal{A}$, takes as input an FJ program P which is well-typed w.r.t. a standard FJ class table (F_0, M_0), and returns a Büchi class table $(F, M_{\mathfrak{Y}})$ w.r.t. which P is well-typed. The pseudo-code of $\mathcal{A}$ and its auxiliary procedures `init` and `typeff` is given in Figure 3. In the definition of the `init` procedure, we write $\mathrm{ar}(m)$ to denote the number of arguments of the method $m \in methods(C)$.

Specifically, $\mathcal{A}$ starts by initializing $(F, M_{\mathfrak{Y}})$ as follows: All entries in F are set to be $\{\texttt{Null}\}$, and those in $M_{\mathfrak{Y}}$ are set to be the pair $(\emptyset, \emptyset)$ of empty expressions. In words, we start with the assumption that all fields are *null* and all methods have no effects. The initialization is carried out by the `init` procedure. It computes also the set $Cls(r)$ of related classes for each region r. After initializing $(F, M_{\mathfrak{Y}})$, the algorithm $\mathcal{A}$ infers the types and effects of each method body of the program, "weakens" the corresponding entry in $M_{\mathfrak{Y}}$ if it differs from the inference result, and repeats this until no further updates of $(F, M_{\mathfrak{Y}})$ are possible. The inference of types and effects is performed via the `typeff` procedure that will be explained below. Once the typings F and $M_{\mathfrak{Y}}$ are updated, the procedure `checkClassTable` ensures that they are well-formed in the sense of Definition 4.2. For instance, if $C \preceq D$ and $F(C, r, f) \neq F(D, r, f)$ for some region r and field f, then both entries are set to $F(C, r, f) \cup F(D, r, f)$ by `checkClassTable`.

The `typeff` procedure maps an environment $\Gamma : Var \rightharpoonup Reg$ and a term $e \in Expr$ to formal expressions $\mathcal{T} \in \mathcal{M}_*\langle Reg\rangle$ and $\mathcal{S} \in \mathcal{M}_*\langle Sig\rangle$. It can be viewed as a function representation of the typing rules listed in Figure 2. For example, when e is an if-expression $\texttt{if}\ x = y\ \texttt{then}\ e_1\ \texttt{else}\ e_2$, the procedure looks at the regions of x and y; if they are disjoint, then it returns $\texttt{typeff}(\Gamma, e_2)$; otherwise, it returns the joint of $\texttt{typeff}(\Gamma, e_1)$ and $\texttt{typeff}(\Gamma, e_2)$. This captures the idea of the typing rules IF and ELSE. Note that `typeff` implicitly takes the class table $(F, M_{\mathfrak{Y}})$ as input. For instance, when e is a call expression $x^{C}.m(\bar{y})$, `typeff` looks up its effects and types in the method typing $M_{\mathfrak{Y}}$. Moreover, `typeff` may update the field typing F when e is a field-update expression $x^{C}.f := y$. For example, the region $\Gamma(y)$ of y will be added into $F(C, \Gamma(x), f)$ if it is not in $F(C, \Gamma(x), f)$.