Büchi Types for Infinite Traces and Liveness

Martin Hofmann[1] and Wei Chen[2]

[1] LMU Munich martin.hofmann@ifi.lmu.de
[2] School of Informatics, University of Edinburgh

Abstract. We develop a new type and effect system based on Büchi automata to capture finite and infinite traces produced by programs in a small language which allows non-deterministic choices and infinite recursions. There are two key technical contributions: (a) an abstraction based on equivalence relations defined by the policy Büchi automata, the Büchi abstraction; (b) a novel type and effect system to correctly capture infinite traces. We show how the Büchi abstraction fits into the abstract interpretation framework and show soundness and completeness.

1 Introduction

A great range of techniques and tools have been developed and studied for the prediction of program behaviours without actually running the program [15,9,26,28,10,11]. One of them, originating from type inference in functional programming languages, is the type and effect discipline [25]. As a refinement of type systems in programming languages, types are annotated with information characterizing dynamic behaviours of programs—*effects*. As a result, a well-typed program satisfies some properties regarding its side-effects as well. This type-based technique has been used for all kinds of static analysis of programs, e.g. flow analysis [27], dependency analysis [1], resource allocation analysis [32], and amortised analysis [21,20], etc. In particular, a type and effect system was developed by Grabowski et al. [18] to verify that a particular programming guideline for secure web-programming has been adhered to. Generalizing from this, one could model a programming guideline as a property of traces that a program might have, where traces are sequences of events that are issued by a certain instrumentation of the program with special event-issuing operations. This instrumentation would be part of the formalised guideline. A finite state machine would then be used to specify the set of acceptable traces. Most policies involve safety properties which can be assessed by examining finite portions of traces. In some cases, however, properties pertaining to liveness and fairness [4] can become relevant. For instance, a guideline could be that calls to appropriate logging functions must be made again and again or that event (sic!) handlers should not become stuck, e.g. in the Java Swing framework.

This motivated us to investigate the possibility of using type systems in this situation as well. Our aim is not to offer new algorithms for deciding certain temporal properties or indeed to compete with the existing methods which are numerous [15,9,26,6], but to extend the reach of type systems. Our solution goes,

however, beyond a simple reformulation of an existing algorithm; the abstract domain based Büchi automata may well be useful in its on right and is an original contribution of this work.

For the sake of simplicity, we introduce and study a small language consisting of recursive first-order procedures and non-deterministic choices. The language explicitly allows infinite recursions. In this language, except for primitive procedures which have events as arguments, other procedures have no inputs. We define *trace semantics* which formalize finite and infinite traces generated by programs in this language. We also remark that in essence our language is the same as the *pushdown systems* that have been studied in detail by a number of authors [34,6,30]. Similar trace semantics were also studied by the Cousots [12] as a specific case of abstract interpretation.

Once a satisfactory type system for such a simple language has been found, it can be combined with known techniques [5,29] to scale to a type system for a large fragment of Java or similar languages. Alternatively, one can use our simple language as a target of a preliminary abstraction step.

Then, we develop the Büchi type and effect system to capture correctly finite and infinite traces. Since branching is non-deterministic in our language, we can even establish a completeness result. Completeness, of course, will be lost, once we re-introduce data-dependent branching.

As a demonstration, we extend the Büchi type and effect system for this small language to a Büchi type and effect system for Featherweight Java with field update [5].

The main technical contribution of this paper is the design of an abstract domain in the sense of abstract interpretation [10,11] based on Büchi automata or rather a mild extension of those allowing infinite as well as finite words. The proofs of soundness and completeness of the type system are based on clear-cut lattice-theoretic properties of this abstraction.

As in the finitary case, this *Büchi abstraction* is based on equivalence relations on finite words generated by the policy Büchi automaton. Abstracted effects are no longer sets of such equivalence classes, but rather sets of pairs of the form (U, V) with U, V classes and representing the infinitary language UV^ω. While such pairs appear in Büchi's original complementation construction for Büchi automata [7] and have subsequently been used by a number of authors [31,14,19], they have never been used in the context of type systems and abstract interpretation.

1.1 Related work

As already mentioned, our language is equivalent to pushdown systems for which model-checking of temporal properties has been extensively studied [34,30,16,8]. Pushdown systems, on the other hand, are special cases of higher-order recursion systems introduced by Knapik et al. [23] and extensively studied by Ong and his collaborators, e.g. [3,24].

The latter work [24] also casts model checking into the form of a type system and is thus quite closely related to our result. More precisely, from an alternating parity automaton a type system for higher-order recursion schemes is derived

such that a scheme is typable iff its evaluation tree would be accepted by the automaton. In this way, in particular all mu-calculus definable properties of the evaluation tree become expressible. Regarding trace languages as opposed to tree properties alternating parity automata are equivalent to Büchi automata since both capture the $(\omega-)$regular languages. Thus for the trace language of interest here the system from loc.cit. is equal in expressive power to our type system.

The difference is that Kobayashi and Ong's system has a much more semantic flavour not unlike the intersection type systems used to characterise strong normalisation. More concretely, the well-formedness condition for recursions in that system requires the solution of a parity game whose size is proportional to the size of the program (number of function symbols to be precise) which is known to be equivalent to model checking trees against mu-calculus formulas.

Our type system, on the other hand, deviates from the standard type systems used in programming and program analysis only very slightly; instead of the usual recursion rule (which is clearly unsound in the context of liveness) we use a rule involving a type variable. No further semantic conditions need to be checked once of course the given Büchi automaton has been analysed and preprocessed.

We can also mention that our method and approach are rather different. While *loc.cit.* uses games and automata we rely on the recently re-popularised [17,2] Ramseyian approach to the study of ω-regular language and automata.

Another recent work on the use of types for properties of infinite traces is [22] which embeds formulas of Linear Temporal Logic into types in the context of functional reactive programming. This work, however, relies on an encoding of linear temporal logic in first-order logic with integers, e.g., one models "the event x occurs infinitely often" as a formula like $\forall i \exists j.x_j$ where x_j refers to that the event x issues at the time j. Dependent types are being used to turn this into a type system, but questions of inference and decidability are not considered.

The discussed works [24,22] are—to our knowledge—the only attempts at extending the range of typing beyond safety properties.

1.2 Outline

In the next section we define a simple first-order language with parameterless recursive procedures and non-deterministic branching (meant, of course, to abstract ordinary conditionals). An alphabet of events Σ is assumed and for each event $a \in \Sigma$ a primitive procedure $o(a)$ is available that outputs a and has no effect on control flow or state. Programs are toplevel mutually recursive definitions of parameterless procedures comparable to the C-language. Given a program, any expression then admits a set of finite and infinite words over Σ—the traces of terminating and nonterminating computations of the program. We distinguish finite traces stemming from terminating execution from finite traces stemming from nonterminating but "unproductive" executions. Thus, for every expression e (relative to a well-formed program) and trace $w \in \Sigma^{\leq\omega} = \Sigma^* \cup \Sigma^\omega$ we define a judgement $e \Downarrow w$ meaning that e admits a terminating execution with trace w (necessarily $w \in \Sigma^*$ then) and another judgement $e \Uparrow w$ meaning that e admits a nonterminating computation with trace w. In this case both $w \in \Sigma^*$

and $w \in \Sigma^\omega$ are possible. Formally, this is defined by introducing ✓ events that are repeatedly issued so that any nonterminating computation will have an infinite trace *with* ✓ events. The official trace semantics ($e \downarrow w$ and $e \uparrow w$) is then defined by discarding these ✓ events. We then discuss alternative ways for defining the trace semantics and emphasize that it is merely meant to formalize the intuitively clear notion of event trace occurring during a computation.

In Section 3 we then define a type-and-effect system whose effects are pairs (U, V) with $U \subseteq \Sigma^*$ and $V \subseteq \Sigma^{\leq\omega}$. Semantically, an expression has effect (U, V) when $e \downarrow w$ implies $w \in U$ and $e \uparrow w$ implies $w \in V$. The typing rules are given in Figure 2. We notice here that for the U-part (terminating computation) the typing rules are as usual; one "guesses" a type for a recursively defined procedure and justifies it for its body. The rule for the nonterminating "V-part" is different. One assumes a type (and effect) variable for the recursive calls, typechecks the body and then takes the greatest fixpoint of the resulting type-and-effect equation.

With an ordinary recursive typing rule it would be possible to infer an effect like $(\emptyset, (a^*b)^\omega)$ ("infinitely often b") for the program $m() = o(a); m()$ which is unsound. We then establish soundness (Theorem 1) and completeness (Theorem 2) for this type-and-effect system. In particular, this shows that the proposed handling of recursive definitions does indeed work.

The type system is at this level, however, of limited use since the effects are infinitary objects. Therefore, in Section 4 we introduce an abstraction of this type-and-effect system where effects are taken from a fixed finite set. This finite set is calculated from an a priori given Büchi automaton and effects still denote pairs of finite and possibly infinite languages, but no longer is any such pair denotable.

Our main result Theorem 5 then asserts that if the set of all traces of an expression is accepted by the given Büchi automaton then this is provable in the abstracted type-and-effect system. So, no precision is lost in this sense. Of course, we also have an accompanying soundness theorem (Theorem 4) for the abstract type-and-effect system.

These results can be modularly deduced from soundness and completeness for the infinitary type-and-effect system (Thms 1 and 2) with the help of lattice-theoretic properties of the abstraction that are established in Section 4.2. In particular, we have a Galois connection between the lattice of all languages and the lattice of language denotations in the abstract type system and all operations needed in the typing rules, in particular least and greatest fixpoints can be correctly rendered on the level of the abstractions.

The crucial building block is the ability to compute abstractions of greatest fixpoints needed for recursive definitions entirely on the level of the abstracted types. This requires the combination of a combinatorial lemma (Lemma 8) with known covering properties (Lemma 3) of the abstractions which follow from Ramsey's theorem. Section 4.1 and Section 4.2 contain these lattice-theoretic results. We consider the discovery of this abstract lattice obtained from a Büchi automaton an important result of independent interest.

Section 4.2 then contains the actual definition of the abstracted type-and-effect system and its soundness and completeness theorems which, as already mentioned, then are direct consequences of earlier results. Section 4.3 discusses automatic type inference and its complexity.

An Appendix contains several worked out examples that did not fit into the main text and omitted proofs. We also sketch there, as a demonstration, a combination of an existing region-based type and effect system for Featherweight Java with field update [5] with Büchi types.

2 Trace Semantics

The syntax of expressions is given by $e ::= o(a) \mid f \mid e_1 \,;\, e_2 \mid e_1 \,?\, e_2$ where $o(a)$ is the only primitive procedure which generates an event a taken from a fixed alphabet Σ of events and f ranges over procedures defined by expressions. Parentheses are used to eliminate ambiguity. We assume that the operator ; is right-associative and has higher priority than the operator ?. As an example, we can define procedures f and g as: $f = o(b) \,?\, o(a) \,;\, g$ and $g = f \,;\, g \,;\, (o(b) \,?\, o(a))$. Formally, thus a *program* consists of a finite set of procedure identifiers $\mathcal{F}$ and for each $f \in \mathcal{F}$ an expression e_f defining f where calls to procedures from $\mathcal{F}$ are allowed and in particular, f may occur recursively in e_f.

From now on, we fix such a program $\mathcal{P} = (\mathcal{F}, (e_f)_{f \in \mathcal{F}})$ and call an expression e *well-formed* if it uses calls to procedures from $\mathcal{F}$ only.

Since the operator ? is non-deterministic and non-primitive procedures have no arguments, stacks and heaps are not needed at this level of abstraction.

Let $\Sigma^{\leq\omega}$ be the set of all finite and infinite sequences generated from the set Σ of primitive events. We call an element w in $\Sigma^{\leq\omega}$ a *trace*. Given traces w and u, we define the concatenation $w \cdot u$ as: wu if $w \in \Sigma^*$ and w if $w \in \Sigma^\omega$ where Σ^* and Σ^ω are respectively sets of all finite and infinite sequences over Σ. So, $\Sigma^{\leq\omega} = \Sigma^* \cup \Sigma^\omega$ and $\Sigma^* = \Sigma^+ \cup \{\epsilon\}$. As usual, we may write wu instead of $w \cdot u$. We are concerned with finite prefixes of the trace generated by a given expression. We call them *observed traces*. Notice that all observed traces are in Σ^*. Let e_f be the definition (a well-formed expression) of f. The observed trace semantics is given in Figure 1. We write $e \Downarrow w$ to mean that the finite trace

$$\frac{}{o(a) \Downarrow a} \quad \frac{}{o(a) \Uparrow a} \quad \frac{}{e \Uparrow \epsilon} \quad \frac{e_f \Downarrow w}{f \Downarrow w} \quad \frac{e_f \Uparrow w}{f \Uparrow w} \quad \frac{e_1 \Downarrow w}{e_1 \,;\, e_2 \Downarrow w \cdot u} \; e_2 \Downarrow u \quad \frac{e_1 \Downarrow w}{e_1 \,;\, e_2 \Uparrow w \cdot u} \; e_2 \Uparrow u$$

$$\frac{e_1 \Uparrow w}{e_1 \,;\, e_2 \Uparrow w} \quad \frac{e_1 \Downarrow w}{e_1 \,?\, e_2 \Downarrow w} \quad \frac{e_2 \Downarrow w}{e_1 \,?\, e_2 \Downarrow w} \quad \frac{e_1 \Uparrow w}{e_1 \,?\, e_2 \Uparrow w} \quad \frac{e_2 \Uparrow w}{e_1 \,?\, e_2 \Uparrow w}$$

Fig. 1. The Observed Trace Semantics

generated by e is w. In particular, e terminates. We write $e \Uparrow w$ to mean that w is a finite prefix of the trace generated by e. Let the notation $u \preccurlyeq w$ denote that u is a finite prefix of w. We have: if $e \Downarrow w$ or $e \Uparrow w$, then for all $u \preccurlyeq w$, $e \Uparrow u$.

We now turn to define infinite traces of non-terminating programs. Unfortunately, the observed trace semantics does not contain enough information for this. Let us consider the following definitions: $f = o(a)$, $g = (o(a)\ ;\ h)\ ?\ o(a)$, and $h = h$. Notice that the observed traces of f and g are exactly the same. However, the procedure g has a path leading to an unproductive infinite recursion h while f is non-recursive. In order to fix this problem, let us introduce the extended set $\Sigma \uplus \{\checkmark\}$ of events and use $(\Sigma \uplus \{\checkmark\})^{\leq\omega}$ for the set of all *extended traces*. The *observed extended trace semantics* is same as the observed trace semantics except for the rule for function application in which a $\checkmark$-event is automatically generated. That is, $\frac{e_f \Uparrow w}{f\,\Uparrow \checkmark \cdot w}$. The specific symbol $\checkmark$ is added to the beginning of trace w of e_f. By doing this, unproductive infinite recursions can be distinguished from productive cases by observed extended traces $\checkmark^*$.

For all observed extended traces w, let $\theta(w)$ denote the trace obtained from w by removing all $\checkmark$s. Based on the observed extended trace semantics, we define trace semantics as follows.

Definition 1 (Trace Semantics). *For all expressions e and extended traces w in $(\Sigma \uplus \{\checkmark\})^{\leq\omega}$,*

$$\begin{aligned} e \downarrow w &\ \equiv\ \exists w' \in (\Sigma \uplus \{\checkmark\})^*.\, e \Downarrow w' \wedge w = \theta(w'); \\ e \uparrow w &\ \equiv\ \exists w' \in (\Sigma \uplus \{\checkmark\})^{\omega}.\, (\forall u \preccurlyeq w'\,.\, e \Uparrow u) \ \wedge\ w = \theta(w'). \end{aligned}$$

We say w is a trace of e if $e \downarrow w$ or $e \uparrow w$.

Notice that if $e \downarrow w$, then w is in Σ^* and all executions of e terminate. If $e \uparrow w$, then w is in $\Sigma^{\leq\omega}$ and all executions of e do not terminate. In our definition of trace semantics, the symbol $\checkmark$ is introduced to distinguish finite traces generated by terminating programs and non-terminating programs. When the trace semantics is well-defined, we remove all $\checkmark$s.

We remark that this way of defining the semantics is one of several possibilities; alternatives would consist of using a small step operational semantics or a coinductive definition. For instance, Cousot et al [12] define a generalization of structured operational semantics (G^∞SOS), is used to describe the finite and infinite executions of programs. At the end of the day we need to define the two judgements $e \downarrow w$ meaning that e terminates with trace w so, necessarily $w \in \Sigma^*$ and $e \uparrow w$ meaning that e does not terminate (runs forever) and its trace is w. In this case, w may either be an infinite word ($w \in \Sigma^\omega$) or a finite word ($w \in \Sigma^*$) in which case e's evaluation gets stuck in an infinite loop but e does not output events during this loop.

An important fine point is that at our level of abstraction programs have a finite store which means that by König's lemma "arbitrarily long" and "infinitely long" coincide. In a language allowing the nondeterministic selection of integers we could write a program that admits traces (outputting as) of any finite length but not having an infinite trace. Then, our trace semantics would erroneously ascribe the trace a^ω to such a program. But, fortunately, in our situation this does not occur. As a result, for some language extensions, one may need to consider more complicated formal definitions of trace semantics. This would,

however, have no influence on the type system we define and only very little influence on correctness proofs.

3 Type and Effect System

In this section, we develop a type and effect system that captures the set of finite and infinite traces of a program. We also prove that this system is sound and complete. This system uses arbitrary languages for effect annotations and as such is not yet suitable for practical use let alone automatic inference. Later, in Section 4 we define a finitary abstraction of this system which still allows one to check soundly and completely whether the traces of a given program are accepted by a fixed Büchi automaton.

Definition 2 (Effect). *Let U be a subset of Σ^* and V be a subset of $\Sigma^{\leq\omega}$. An effect of a given expression e is a pair (U, V) satisfying: (a) if $e \downarrow w$, then w is in U; (b) if $e \uparrow w$, then w is in V. We use the notation e & (U, V) to denote that (U, V) is an effect of e.*

Let $\mathfrak{X}$ be a set of variables. Let $V(\mathfrak{X})$ range over expressions of the form: $\bigcup_{X\in\mathfrak{X}}(A_X \cdot X) \cup B$ with $A_X \subseteq \Sigma^*$ and $B \subseteq \Sigma^{\leq\omega}$. We abbreviate $\mathfrak{X} \setminus \{X\}$ by $\mathfrak{X} - X$ and thus use the notation $V(\mathfrak{X} - X)$ to denote expressions of the form: $\bigcup_{Y\in\mathfrak{X}-\{X\}}(A_Y \cdot Y) \cup B$. We use the symbol X itself for the expression where $B = \emptyset$, $A_X = \{\epsilon\}$, and $A_Y = \emptyset$ for all Y in $\mathfrak{X} - X$. We define the following operations on these expressions: $A \cdot V(\mathfrak{X}) = \bigcup_{X\in\mathfrak{X}}((A \cdot A_X) \cdot X) \cup (A \cdot B)$ and $V(\mathfrak{X}) \cup V'(\mathfrak{X}) = \bigcup_{X\in\mathfrak{X}}((A_X \cup A'_X) \cdot X) \cup (B \cup B')$ where A is a subset of Σ^*. Given an assignment function $\eta : \mathfrak{X} \to \mathcal{P}(\Sigma^{\leq\omega})$ that assigns a set of traces to each variable X in $\mathfrak{X}$, we obtain for each expression $V(\mathfrak{X})$ a language $V(\eta) \subseteq \Sigma^{\leq\omega}$ by substituting $\eta(X)$ for each variable X.

We define A^ω as the set of all words of the form $w = w_0w_1w_2\cdots w_i\cdots$ where $w_i \in A$. Note that $A^\omega \subseteq \Sigma^{\leq\omega}$. Let Δ be an *environment* that is a set of expressions of the form: f & (U, X) with f a non-primitive procedure, U a subset of Σ^*, and X a variable in $\mathfrak{X}$ such that if f & (U, X) and g & (V, Y) both occur in Δ then $X \neq Y$. With the above definitions, we define the *type-and-effect system* in Figure 2. An environment Δ is justified if for all f & (U, X) in

$$\frac{}{\Delta \vdash o(a)\ \&\ (\{a\}, \emptyset)} \qquad \frac{\Delta \vdash e_1\ \&\ (U_1, V_1(\mathfrak{X})) \qquad \Delta \vdash e_2\ \&\ (U_2, V_2(\mathfrak{X}))}{\Delta \vdash e_1\,;\, e_2\ \&\ (U_1 \cdot U_2, V_1(\mathfrak{X}) \cup U_1 \cdot V_2(\mathfrak{X}))}$$

$$\frac{\Delta \vdash e_1\ \&\ (U_1, V_1(\mathfrak{X})) \qquad \Delta \vdash e_2\ \&\ (U_2, V_2(\mathfrak{X}))}{\Delta \vdash e_1\,?\,e_2\ \&\ (U_1 \cup U_2, V_1(\mathfrak{X}) \cup V_2(\mathfrak{X}))} \qquad \frac{}{\Delta, f\ \&\ (U, X) \vdash f\ \&\ (U, X)}$$

$$\frac{\Delta, f\ \&\ (U, X) \vdash e_f\ \&\ (U, A \cdot X \cup V(\mathfrak{X} - X))}{\Delta \vdash f\ \&\ (U, A^* \cdot V(\mathfrak{X} - X) \cup A^\omega)}$$

Fig. 2. The Type and Effect System

Δ one has $\Delta \vdash e_f$ & $(U, A \cdot X \cup V(\mathfrak{X} - X))$ for some A, $V(\mathfrak{X} - X)$. A justified environment can be extended as follows:

Lemma 1. *Given a justified environment Δ such that $\Delta, f \;\&\; (U, X) \vdash e_f \;\&\; (U, A \cdot X \cup V(\mathfrak{X} - X))$, then the extended environment $\Delta, f \;\&\; (U, X)$ is also justified*

An assignment function η *satifies* an environment Δ if whenever $f \;\&\; (U, X)$ in Δ and $f \uparrow w$ then $w \in \eta(X)$. Let us use the notation $\eta \models \Delta$ to denote that the environment Δ is justified and that the assignment function η satisfies Δ.

Lemma 2. *Given an environment Δ and an assignment function η satisfying that $\eta \models \Delta$, let η' be an extension $\eta[X \mapsto V]$ of η such that $f \uparrow w$ implies $w \in V$ for all traces w. If we have the derivation: $\Delta, f \;\&\; (U, X) \vdash e_f \;\&\; (U, A \cdot X \cup V(\mathfrak{X} - X))$, then $\eta' \models \Delta, f \;\&\; (U, X)$.*

Theorem 1 (Soundness). *Given an environment Δ and an assignment function η satisfying that $\eta \models \Delta$, for all derivations: $\Delta \vdash e \;\&\; (U, V(\mathfrak{X}))$ of an expression e, we have: $e \downarrow w$ implies $w \in U$ and $e \uparrow w$ implies $w \in V(\eta)$.*

Proof. The only interesting case is that for the last rule in Figure 2 which relies on Lemma 2. For more details, see Appendix B.

Corollary 1. *For all derivations $\vdash e \;\&\; (U, V)$ of an expression e, we have: $e \downarrow w$ implies $w \in U$ and $e \uparrow w$ implies $w \in V$.*

Fix for each non-primitive procedure $f \in \mathcal{F}$ a unique variable X_f. If $\boldsymbol{A} = (A_f)_f$ is a family of languages with $A_f \subseteq \Sigma^*$ define the corresponding environment $\Delta(\boldsymbol{A})$ as to contain the bindings $f \;\&\; (A_f, X_f)$. For each function body e_f we can now derive using the rules except the last one a unique typing $\Delta(\boldsymbol{A}) \vdash e_f \;\&\; \ldots$. The passage from $\boldsymbol{A}$ to $\boldsymbol{C} = (C_f)_f$ defines a monotone operator Φ on the lattice $\mathcal{P}(\Sigma)^{\mathcal{F}}$. If $\boldsymbol{B}$ is the least fixpoint of this operator then $\Delta(\boldsymbol{B})$ is justified and we get the judgements $\Delta(\boldsymbol{B}) \vdash e_f \;\&\; (B_f, V(\mathfrak{X}))$. Successive application of the last rule then gives judgements $\vdash f \;\&\; (U_f, V_f)$ and a direct induction shows that in fact $U_f = \{w \mid f \downarrow w\}$ and $V_f = \{w \mid f \uparrow w\}$. We have thus shown:

Theorem 2 (Completeness). *The judgements $\vdash f : (\{w \mid f \downarrow w\}, \{w \mid f \uparrow w\})$ are derivable for each f.*

We have kept the proof of this theorem in the running text since the monotone operator Φ is still needed later.

4 Büchi Type and Effect System

Based on equivalence relations on finite words defined by the policy Büchi automata, we introduce an abstraction of languages of finite and infinite words: the Büchi abstraction. We place this abstraction into the framework of abstract interpretation and show that crucial operations, namely concatenation, least fixpoint, and infinite iteration ($(-)^\omega$) can be computed on the level of the abstraction. We also show that the abstraction does not lose any information as far as acceptance by the fixed policy automaton is concerned. This then allows us to replace the infinitary effects in the previous type system by their finite abstraction and thus to obtain a type-and-effect system which is decidable with

low complexity (in the program size) and yet complete. The soundness and completeness of this system follow directly from lattice-theoretic properties of this Büchi abstraction (Lemma 3 and Theorem 3).

4.1 Extended Büchi Automata

Given an expression e, our goal is to verify that the set $T(e)$ of all traces generated by e satisfies some property. We use a mild extension of the standard Büchi Automata which we call *extended Büchi automata*:

Definition 3. *An extended Büchi Automaton is a quadruple $\mathfrak{A} = (Q, \Sigma, \delta, q_0, F)$ where Q is a finite set of states, Σ is an alphabet; hereafter always required to be equal to the fixed alphabet of events; $\delta : Q \times \Sigma \to \mathcal{P}(Q)$ the transition function, the initial state $q_0 \in Q$, and the set $F \subseteq Q$ of final states. The language $L(\mathfrak{A})$ of $\mathfrak{A}$ is defined as: the set of all finite words by which a final state can be reached from the initial state and all infinite words for which there is a path which starts from the initial state and goes through final states infinitely often. Thus, $L(\mathfrak{A})$ is the union of $\mathfrak{A}$'s language when understood as a traditional NFA and its language when understood as a traditional Büchi automaton.*

Following Büchi's original works we use equivalence relations defined by extended Büchi automata themselves to obtain finite representations of U and V. We write $q \overset{w}{\rightsquigarrow} q'$ to mean that the state q' is reachable from the state q by using the finite word w. Let $q \overset{w}{\rightsquigarrow}_F q'$ denote that by using the finite word w, the state q' can be reached from the state q in such a way that a final state is visited on the way. In particular, $q \overset{w}{\rightsquigarrow} q'$ with $q \in F$ or $q' \in F$ implies $q \overset{w}{\rightsquigarrow}_F q'$. Formally, we have $q \overset{w}{\rightsquigarrow}_F q'$ iff there exists $q'' \in F$ and u, v such that $w = uv$ and $q \overset{u}{\rightsquigarrow} q''$ and $q'' \overset{v}{\rightsquigarrow} q'$.

For nonempty words $w, u \in \Sigma^+$ we define

$$w \sim u \;\equiv\; \forall p, q \in Q \;.\; p \overset{w}{\rightsquigarrow} q \Leftrightarrow p \overset{u}{\rightsquigarrow} q \;\wedge\; p \overset{w}{\rightsquigarrow}_F q \Leftrightarrow p \overset{u}{\rightsquigarrow}_F q \;.$$

We write $[w]$ for the equivalence class of $w \in \Sigma^+$ and additionally let $[\epsilon]$ stand for $\{\epsilon\}$. We write $\mathcal{Q}$ for $\Sigma^+/\!\sim \uplus \{[\epsilon]\}$. Thus $\mathcal{Q}$ comprises the $\sim$-equivalence classes and a special class for the empty word.

We notice that if $w \sim u$ and $w' \sim u'$ then $ww' \sim uu'$. As a result concatenation is well-defined on equivalence classes, thus $\Sigma^+/\!\sim$ becomes a semigroup and $\mathcal{Q}$ a monoid. The following Lemma is a straightforward consequence of standard results about Büchi automata [33].

Lemma 3. *Fix an extended Büchi automaton $\mathfrak{A} = (Q, \Sigma, \delta, q_0, F)$,*

(a) $\mathcal{Q}$ is finite and its elements are regular languages;
(b) for all classes C in $\mathcal{Q}$, $C \cap L(\mathfrak{A}) \neq \emptyset$ implies $C \subseteq L(\mathfrak{A})$;
(c) for all classes C and D in $\mathcal{Q}$, $CD^\omega \cap L(\mathfrak{A}) \neq \emptyset$ implies $CD^\omega \subseteq L(\mathfrak{A})$;
(d) for every word $w \in \Sigma^{\leq\omega}$ there exist classes $C, D \in \mathcal{Q}$ so that $w \in CD^\omega$ and $CD = C$ and $DD = D$.

The sets CD^ω (with $CD = C$ and $DD = D$) thus behave almost like classes themselves, but an important difference is that they may nontrivially overlap. If $CD^\omega \cap UV^\omega \neq \emptyset$ then in general one cannot conclude $CD^\omega = UV^\omega$. We also remark that Ramsey's theorem is used in the proof of (d).

4.2 Büchi Abstraction

Lemma 3 shows that given an extended Büchi automaton $\mathfrak{A}$, without affecting property checking, we can use sets of classes in $\mathcal{Q}$ to represent languages over Σ^* and sets of pairs of classes (C, D) such that $CD = C$ and $DD = D$ to represent languages over $\Sigma^{\leq\omega}$. Let us write $\mathcal{C} := \{(C, D) \mid C, D \in \mathcal{Q} \wedge CD = C, DD = D\}$. Let us define the pre-abstraction function $\mathfrak{f} : \mathcal{P}(\Sigma^{\leq\omega}) \to \mathcal{P}(\mathcal{C})$ and the pre-concretization function $\mathfrak{g} : \mathcal{P}(\mathcal{C}) \to \mathcal{P}(\Sigma^{\leq\omega})$ as: $\mathfrak{f}(V) = \{(C, D) \in \mathcal{C} \mid CD^\omega \cap V \neq \emptyset\}$ and $\mathfrak{g}(\mathcal{V}) = \bigcup_{(C,D)\in\mathcal{V}} CD^\omega$ respectively. A set $\mathcal{V} \subseteq \mathcal{C}$ is *closed*, if $\mathfrak{f}(\mathfrak{g}(\mathcal{V})) = \mathcal{V}$. Explicitly, $\mathcal{V}$ is closed if whenever $UV^\omega \cap CD^\omega \neq \emptyset$ for some $(C, D) \in \mathcal{V}$ and $(U, V) \in \mathcal{C}$ then already $(U, V) \in \mathcal{V}$.

Clearly, for every set $\mathcal{V} \subseteq \mathcal{C}$ there is a least closed superset and it is given by applying the *closure* function: $\mathfrak{c} : \mathcal{P}(\mathcal{C}) \to \mathcal{P}(\mathcal{C})$ which is defined as: $\mathfrak{c}(\mathcal{V}) = \bigcup_{n=1}^{\infty}(\mathfrak{f} \circ \mathfrak{g})^n(\mathcal{V})$. We write $\mathcal{M}_{\leq\omega} = \{\mathfrak{c}(\mathfrak{f}(V)) \mid V \subseteq \Sigma^{\leq\omega}\}$ for the set such closed subsets. The elements of $\mathcal{M}_{\leq\omega}$ will serve as abstractions of languages over $\Sigma^{\leq\omega}$.

We also define explicitly $\mathcal{M}_* = \mathcal{P}(\mathcal{Q})$ to represent languages over Σ^* but note that via the embedding $C \mapsto (C, \{\epsilon\})$ we could identify $\mathcal{M}_*$ with a subset of $\mathcal{M}_{\leq\omega}$.

Lemma 4. *Both $\mathcal{M}_*$ and $\mathcal{M}_{\leq\omega}$ are complete lattices with respect to inclusion.*

From now on we call $\mathcal{P}(\Sigma^*)$ and $\mathcal{P}(\Sigma^{\leq\omega})$ the *concrete domains* and $\mathcal{M}_*$ and $\mathcal{M}_{\leq\omega}$ the *abstract domains*. We introduce the following *abstraction functions*: $\alpha_* : \mathcal{P}(\Sigma^*) \to \mathcal{M}_*$ and $\alpha_{\leq\omega} : \mathcal{P}(\Sigma^{\leq\omega}) \to \mathcal{M}_{\leq\omega}$, which are respectively defined as: $\alpha_*(U) = \{C \in \mathcal{Q} \mid C \cap U \neq \emptyset\}$ and $\alpha_{\leq\omega}(V) = \mathfrak{c}(\mathfrak{f}(V))$. We also introduce the following *concretization functions*: $\gamma_* : \mathcal{M}_* \to \mathcal{P}(\Sigma^*)$ and $\gamma_{\leq\omega} : \mathcal{M}_{\leq\omega} \to \mathcal{P}(\Sigma^{\leq\omega})$, which are respectively defined as: $\gamma_*(\mathcal{U}) = \bigcup_{C\in\mathcal{U}} C$ and $\gamma_{\leq\omega}(\mathcal{V}) = \mathfrak{g}(\mathcal{V})$.

Lemma 5. *The abstraction and concretization functions are monotone and form Galois connections, that is: $\alpha_*(U) \subseteq \mathcal{U}$ iff $U \subseteq \gamma_*(\mathcal{U})$, and $\alpha_{\leq\omega}(V) \subseteq \mathcal{V}$ iff $V \subseteq \gamma_{\leq\omega}(\mathcal{V})$. Moreover, $\alpha_*(\gamma_*(\mathcal{U})) = \mathcal{U}$ and $\alpha_{\leq\omega}(\gamma_{\leq\omega}(\mathcal{V})) = \mathcal{V}$ so we have in fact a Galois* injection. *Furthermore, both abstraction and concretization functions preserve unions, least and greatest elements. The concretization functions also preserve intersections.*

The next lemma shows that the abstraction is sufficiently fine for our purposes. It is a direct consequence of the Galois connection and the fact that $L(\mathfrak{A})$ itself is closed which in turn is direct from Lemma 3 (b) and (c).

Lemma 6. *Let $p \in \{*, \leq\omega\}$. If $L \subseteq \Sigma^p$ then $\gamma_p(\alpha_p(L)) \subseteq L(\mathfrak{A})$ iff $L \subseteq L(\mathfrak{A})$.*

We now turn to define some new operators for the abstract domains. We have a concatenation operation on $\mathcal{M}_*$ given pointwise, i.e. for $\mathcal{U}, \mathcal{U}' \in \mathcal{M}_*$, we define $\mathcal{U} \cdot \mathcal{U}' = \{UU' \mid U \in \mathcal{U}, U' \in \mathcal{U}'\}$. We also define a concatenation $\cdot : \mathcal{M}_* \times \mathcal{M}_{\leq\omega} \to \mathcal{M}_{\leq\omega}$ as follows: $\mathcal{U} \cdot \mathcal{V} = \{(AC, D) \mid A \in \mathcal{U} \wedge (C, D) \in \mathcal{V}\}$. Note that $ACD = AC$.

Lemma 7. *If $\mathcal{U} \in \mathcal{M}_*$ and $\mathcal{V} \in \mathcal{M}_{\leq\omega}$ then $\mathcal{U} \cdot \mathcal{V} \in \mathcal{M}_{\leq\omega}$.*

Theorem 3. *The preservation properties listed in Table 1 are valid.*

Most of these properties are direct and folklore or have been asserted earlier. In the clause about least fixpoints denoted by *lfp* the operators Φ and F are supposed to be monotone operators on $\mathcal{P}(\Sigma^*)^n$ and $\mathcal{M}_*^n$ for some n. The clause is then validated by straightforward application of lattice-theoretic principles. Only preservation of $(-)^\omega$ is a nontrivial and original result; it requires the following lemma.

$\alpha_* : \mathcal{P}(\Sigma^*) \to \mathcal{M}_*$	$\alpha_{\leq\omega} : \mathcal{P}(\Sigma^{\leq\omega}) \to \mathcal{M}_{\leq\omega}$
$\alpha_*(\top) = \top$	$\alpha_{\leq\omega}(\top) = \top$
$\alpha_*(\bot) = \bot$	$\alpha_{\leq\omega}(\bot) = \bot$
$\alpha_*(U \cup U') = \alpha_*(U) \cup \alpha_*(U')$	$\alpha_{\leq\omega}(V \cup V') = \alpha_{\leq\omega}(V) \cup \alpha_{\leq\omega}(V')$
$\alpha_*(U \cdot U') = \alpha_*(U) \cdot \alpha_*(U')$	$\alpha_{\leq\omega}(U \cdot V) = \alpha_*(U) \cdot \alpha_{\leq\omega}(V)$
$\alpha_* \circ \Phi = F \circ \alpha_* \Longrightarrow \alpha_*(lfp(\Phi)) = lfp(F)$	$\alpha_{\leq\omega}(U^\omega) = \alpha_*(U)^\omega$

Table 1. Properties of the Büchi Abstraction

Lemma 8. *Let $(L_i)_{i\in I}$ be a family of classes (from $\mathcal{Q}$) and put $P = \prod_{i\in I} L_i \subseteq \Sigma^{\leq\omega}$, i.e., P comprises finite or infinite words of the form $w_1w_2w_3\ldots$ where $w_i \in L_i$ for $i \geq 1$. There exist classes $U, V \in \mathcal{Q}$ where $UV = U, VV = V$ such that $P \subseteq UV^\omega$.*

Proof. Let $w \in P$ and write $w = w_1w_2w_3\cdots w_i\cdots$ where $w_i \in L_i$. If w is a finite word then there exists n such that $w_i = \epsilon$ (and $L_i = [\epsilon]$) for $i \geq n$ and we can choose $U = L_1 \ldots L_{n-1}$ and $V = [\epsilon]$. Otherwise, use Ramsey's theorem as in the proof of Lemma 3 to obtain a sequence of indices $i_1 < i_2 < i_3 < i_4 < \ldots$ and classes U, V where $V \neq [\epsilon]$ and $VV = V$, $UV = U$ such that $w_1w_2 \ldots w_{i_1} \in U$, $w_{i_1+1} \ldots w_{i_2} \in V$ and $w_{i_2+1} \ldots w_{i_3} \in V$ and so on. It follows that $U = L_1L_2 \ldots L_{i_1}$ and $V = L_{i_k+1} \ldots L_{i_{k+1}}$ for $k \geq 1$ and thus $P \subseteq UV^\omega$ as required.

Proof (of Theorem 3). It only remains to prove preservation of $(-)^\omega$. So, fix $L \subseteq \Sigma^*$. We want to show that $\alpha_{\leq\omega}(L^\omega) = \alpha_*(L)^\omega$. Note that $\alpha_*(L)^\omega = \alpha_{\leq\omega}(\gamma_*(\alpha_*(L))^\omega)$. The direction "$\subseteq$" is obvious from monotonicity; towards proving "$\supseteq$" assume $(U, V) \in \alpha_{\leq\omega}(\gamma_*(\alpha_*(L))^\omega)$. Since $\alpha_{\leq\omega}(L^\omega)$ is closed, we may without loss of generality assume that $UV^\omega \cap \gamma_*(\alpha_*(L))^\omega \neq \emptyset$. Pick $w \in UV^\omega \cap \gamma_*(\alpha_*(L))^\omega$ and decompose $w = w_1w_2\ldots$ where $w_i \in \gamma_*(\alpha_*(L))$. Define $L_i := [w_i]$ and apply Lemma 8 to obtain U', V' with $\prod_i L_i \subseteq U'V'^\omega$. Note that, since $w \in P$, we have $UV^\omega \cap U'V'^\omega \neq \emptyset$.

Now, since $w_i \in \gamma_*(\alpha_*(L))$, by the definition of α_*, we must have that $L_i \cap L \neq \emptyset$. Choose $w_i' \in L_i \cap L$. The word $w_1'w_2'\ldots$ is then contained in $L^\omega \cap U'V'^\omega$, so $(U', V') \in \alpha_{\leq\omega}(L^\omega)$ and, finally, $(U, V) \in \alpha_{\leq\omega}(L^\omega)$ since $\alpha_{\leq\omega}(L^\omega)$ is closed and $UV^\omega \cap U'V'^\omega \neq \emptyset$.

Definition 4 (Büchi Effect). *Let $\mathcal{U}$ be an element in $\mathcal{M}_*$ and $\mathcal{V}$ be an element in $\mathcal{M}_{\leq\omega}$. The pair $(\mathcal{U},\mathcal{V})$ is a Büchi effect of a given expression e if it satisfies: (a) if $e\downarrow w$, then $w\in\gamma_*(\mathcal{U})$; (b) if $e\uparrow w$, then $w\in\gamma_{\leq\omega}(\mathcal{V})$.*

Let $\mathcal{V}(\mathfrak{X})$ range over expressions of the form: $\bigcup_{X\in\mathfrak{X}}(\mathcal{A}_X\cdot X)\cup\mathcal{B}$ with $\mathcal{A}_X\in\mathcal{M}_*$ and $\mathcal{B}\in\mathcal{M}_{\leq\omega}$. We define the notation $\mathcal{V}(\mathfrak{X}-X)$ and operations on these expressions in the same way as we have done for expressions in the type and effect system in section 3. The definitions of justifiedness and satisfaction of environments and assignments are adapted to the Büchi type system mutatis mutandis. That is, Δ is justified if for all f & $(\mathcal{U},X)$ in Δ one has $\Delta\vdash e_f$ & $(\mathcal{U},\mathcal{A}\cdot X\cup\mathcal{V}(\mathfrak{X}-X))$. It is satisfied by η if f & $(\mathcal{U},X)$ in Δ and $f\uparrow w$ implies $w\in\gamma_{\leq\omega}(\eta(X))$.

With the above definitions, we introduce the Büchi type and effect system in Figure 3.

$$\frac{}{\Delta\vdash_{\mathfrak{A}} o(a)\ \&\ (\alpha_*(\{a\}),\emptyset)}\qquad \frac{\Delta\vdash_{\mathfrak{A}} e_1\ \&\ (\mathcal{U}_1,\mathcal{V}_1(\mathfrak{X}))\qquad \Delta\vdash_{\mathfrak{A}} e_2\ \&\ (\mathcal{U}_2,\mathcal{V}_2(\mathfrak{X}))}{\Delta\vdash_{\mathfrak{A}} e_1\,;\,e_2\ \&\ (\mathcal{U}_1\cdot\mathcal{U}_2,\mathcal{V}_1(\mathfrak{X})\cup\mathcal{U}_1\cdot\mathcal{V}_2(\mathfrak{X}))}$$

$$\frac{\Delta\vdash_{\mathfrak{A}} e_1\ \&\ (\mathcal{U}_1,\mathcal{V}_1(\mathfrak{X}))\qquad \Delta\vdash_{\mathfrak{A}} e_2\ \&\ (\mathcal{U}_2,\mathcal{V}_2(\mathfrak{X}))}{\Delta\vdash_{\mathfrak{A}} e_1\,?\,e_2\ \&\ (\mathcal{U}_1\cup\mathcal{U}_2,\mathcal{V}_1(\mathfrak{X})\cup\mathcal{V}_2(\mathfrak{X}))}\qquad \frac{}{\Delta,f\ \&\ (\mathcal{U},X)\vdash_{\mathfrak{A}} f\ \&\ (\mathcal{U},X)}$$

$$\frac{\Delta,f\ \&\ (\mathcal{U},X)\vdash_{\mathfrak{A}} e_f\ \&\ (\mathcal{U},\mathcal{A}\cdot X\cup\mathcal{V}(\mathfrak{X}-X))}{\Delta\vdash_{\mathfrak{A}} f\ \&\ (\mathcal{U},\mathcal{A}^*\cdot\mathcal{V}(\mathfrak{X}-X)\cup\mathcal{A}^\omega)}$$

Fig. 3. The Büchi Type and Effect System

By using properties of the Büchi abstraction in Table 1, from Theorems 1 and 2, we have that this system is sound and complete.

Theorem 4 (Soundness). *Given an environment Δ and an assignment function η satisfying that $\eta\models_{\mathfrak{A}}\Delta$, for all derivations: $\Delta\vdash_{\mathfrak{A}} e$ & $(\mathcal{U},\mathcal{V}(\mathfrak{X}))$ of an expression e, we have: $e\downarrow w$ implies $w\in\gamma_*(\mathcal{U})$ and $e\uparrow w$ implies $w\in\gamma_{\leq\omega}(\mathcal{V}(\mathfrak{X})_\eta)$.*

Proof. It follows from the Galois connections in Lemma 5. In particular, $U\subseteq\gamma_*(\alpha_*(U))$ and $V\subseteq\gamma_{\leq\omega}(\alpha_{\leq\omega}(V))$.

Theorem 5 (Completeness). *Given a non-primitive procedure f, let $T(f)$ be the set of all traces generated by f. There is a derivation $\vdash_{\mathfrak{A}} f$ & $(\mathcal{U}_f,\mathcal{V}_f)$ such that $T(f)\subseteq L(\mathfrak{A})$ if and only if $\gamma_*(\mathcal{U}_f)\cup\gamma_{\leq\omega}(\mathcal{V}_f)\subseteq L(\mathfrak{A})$.*

Proof. Recall the monotone operator Φ from the proof of Theorem 2. Since Φ is built up from concatenation and union there is an abstract operator F such that $\alpha_*\circ\Phi=F\circ\alpha_*$. Thus $\alpha_*(\mathit{lfp}(\Phi))=\mathit{lfp}(F)$ and therefore, the judgements $\Delta(\alpha(\boldsymbol{B}))\vdash_{\mathfrak{A}} e_f$ & $(\alpha_*(B_f),\alpha_*(V(\mathfrak{X})))$ (again in keeping with the notation of that proof) are derivable in the Büchi type system. Using the preservation of $(-)^\omega$ repeatedly, we then obtain the judgements $\vdash_{\mathfrak{A}} f$ & $(\alpha_*(U_f),\alpha_{\leq\omega}(V_f))$ where $U_f=\{w\mid f\downarrow w\}$ and $V_f=\{w\mid f\uparrow w\}$. Letting $\mathcal{U}_f=\alpha_*(U_f)$ and $\mathcal{V}_f=\alpha_{\leq\omega}(V_f)$ the claim then follows using Lemma 6.

4.3 Type inference and complexity

Given that the abstract lattices and thus the set of types is finite, type inference is a standard application of well-known techniques. We therefore just sketch it here to give an idea of the complexity.

From a given program we can construct in linear time a skeleton typing derivation for the finitary effect annotations. The skeleton typing derivation contains variables in place of actual effect annotations; the number of these variables is linear in the program size. The side conditions of the typing rules then become constraints on these variables and any solution will yield a valid typing derivation. In quadratic time (assuming that $\mathcal{M}_{\leq\omega}$ has constant size) we can then compute the least solution of these constraints using the usual iteration algorithms known from abstract interpretation. Once we have in this way obtained the finitary effect annotations we can then (in linear time) derive the infinitary ones using the $(-)^\omega$ and infinitary concatenation operators on $\mathcal{M}_{\leq\omega}$.

Once the type of an expression has been found one can then check (in constant time) whether the language denoted by it is accepted by the policy automaton.

If we are interested in complexity as a function of the size of the policy automaton the situation is of course different. The important parameter here is the size of the abstract lattices since the number of iterations as well as the runtime of the algorithms for computing the abstractions of concatenation, union, infinite iteration are linear in this parameter. If n is the number of states of the policy automaton then the number of classes can be bounded by 2^{2n^2} since each class is characterised by two sets of pairs of states. The resulting exponential in n runtime of our algorithms is no surprise since the PSPACE-complete problem of universality of Büchi automata is easily reduced to type checking. We believe that by clever space management our algorithms can be implemented in polynomial space but we have not verified this.

On a positive note we remark that for a small policy automaton the set of classes is manageable as we see in the examples below. We also note that once the classes have been computed and the abstract functions tabulated one can then analyse many programs of arbitrary size.

5 Conclusions

We have developed a type-and-effect system for capturing possibly infinite traces of recursively defined first-order procedures. The type-and-effect system is sound and complete with respect to inclusion of traces in a given Büchi ("policy") automaton. The effect annotations are from a finite set that can be effectively computed from the Büchi automaton. Type inference using constraint solving is thus possible. We emphasize that the resulting ability to decide satisfaction of temporal properties of traces is not claimed as a new result here; since it has long been known in the context of model checking. The novelty lies in the presentation as a type and effect system that follows the standard pattern of such systems. As we explain below, this opens the way for smooth integration with existing type-theoretic technology.

The proofs of soundness and completeness are organised in a modular fashion and decomposed into a type-theoretic part expressed in the form of an infinitary system (Section 3) and a lattice-theoretic part (Section 4.2). Concretely, this Section defines an abstract domain from any given Büchi automaton and derives crucial properties of this abstraction. We consider a contribution of independent interest. The finite part of this abstraction, i.e., $\mathcal{M}_*$, is akin to the abstract domain proposed by Cousot et al [13] which also has finite abstraction values and its abstraction function preserves the least fixed point as well. The infinite part $\mathcal{M}_{\leq\omega}$ is a new abstract domain with its abstraction function preserving not only least fixed points but also the new operator $(-)^\omega$. We remark here that the abstraction function does not preserve greatest fixed points so that the introduction of the $(-)^\omega$ operator on the abstract domain is a necessary device. This extension makes the Büchi type and effect system powerful enough to capture and reason about infinitary properties like liveness and fairness.

We have sketched a combination of our simple type system with a generic type and effect system for class-based object-oriented languages. Other possible extensions are in the direction of effectful functional programming. The standard notation for type-and-effect systems as described e.g. in Henglein and Niss' survey [29] could be used for our effect system mutatis mutandis leading in particular to function types like $w \xrightarrow{\epsilon} w'$ where w, w' are types and ϵ is a Büchi effect (element of our abstract lattice) describing the latent effect of a function. Assuming that we only allow first-order recursive definitions the design of such a type system would be completely standard. For higher-order recursion some extra technical work would be needed to lift the last rule from Fig. 2 and its corresponding abstraction to this case.

It is this option of integration with expressive type systems that makes our abstraction so attractive and superior (in this context!) to classical methods based on model checking.

References

1. Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. A core calculus of dependency. *POPL*, pages 147–160. ACM, 1999.
2. Parosh Aziz Abdulla, et al. Advanced ramsey-based büchi automata inclusion testing. *CONCUR*, LNCS 6901, pp 187–202. Springer, 2011.
3. Klaus Aehlig, Jolie G. de Miranda, and C.-H. Luke Ong. The monadic second order theory of trees given by arbitrary level-two recursion schemes is decidable. *TLCA*, LNCS 3461 pp 39–54. Springer, 2005.
4. Bowen Alpen and Fred B. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2(3):117–126, 1987.
5. Lennart Beringer, Robert Grabowski, and Martin Hofmann. Verifying pointer and string analyses with region type systems. COMLAN, 2013.
6. Ahmed Bouajjani, et al. Reachability analysis of pushdown automata: Application to model-checking. *CONCUR*, LNCS 1243, pp 135–150. Springer, 1997.
7. J. R. Büchi. On a decision method in restricted second order arithmetic. In *Logic, Method, and Philosophy of Science*, Stanford University Press, 1962.
8. Olaf Burkart and Bernhard Steffen. Model checking the full modal mu-calculus for infinite sequential processes. *TCS*, 221(1-2):251–270, 1999.

9. Edmund M. Clarke, et al. Counterexample-guided abstraction refinement. *CAV*, LNCS 1855, pages 154–169. Springer, 2000.
10. Patrick Cousot and Radhia Cousot. Abstract interpretation. *POPL*, pages 238–252. ACM, 1977.
11. Patrick Cousot and Radhia Cousot. Abstract interpretation frameworks. *J. Log. Comput.*, 2(4):511–547, 1992.
12. Patrick Cousot and Radhia Cousot. Inductive definitions, semantics and abstract interpretation. In Ravi Sethi, editor, *POPL*, pages 83–94. ACM Press, 1992.
13. Patrick Cousot and Radhia Cousot. Formal language, grammar and set-constraint-based program analysis by abstract interpretation. In *FPCA*, pages 170–181, 1995.
14. Christian Dax, Martin Hofmann, and Martin Lange. A proof system for the linear time μ-calculus. *FSTTCS*, LNCS 4337, pages 273–284. Springer, 2006.
15. E. Allen Emerson and Edmund M. Clarke. Characterizing correctness properties of parallel programs using fixpoints. *ICALP*, LNCS 85, pp 169–181. Springer, 1980.
16. Javier Esparza, et al Efficient algorithms for model checking pushdown systems. *CAV*, LNCS 1855, pp 232–247. Springer, 2000.
17. Seth Fogarty and Moshe Y. Vardi. Büchi complementation and size-change termination. *Logical Methods in Computer Science*, 8(1), 2012.
18. Robert Grabowski et al. Type-based enforcement of secure programming guidelines - code injection prevention at sap. *FAST*, LNCS 7140, pp 182–197. Springer, 2011.
19. M. Heizmann, N. Jones, and A. Podelski. Size-change termination and transition invariants. *SAS*, LNCS 6337, pp 22–50. Springer, 2010.
20. Martin Hofmann and Steffen Jost. Type-based amortised heap-space analysis. *ESOP*, LNCS 3924, pp 22–37. Springer, 2006.
21. Martin Hofmann and Dulma Rodriguez. Automatic type inference for amortised heap-space analysis. *ESOP*, LNCS 7792, pp 593–613. Springer, 2013.
22. Alan Jeffrey. Ltl types frp: linear-time temporal logic propositions as types, proofs as functional reactive programs. *PLPV*, pages 49–60. ACM, 2012.
23. Teodor Knapik, Damian Niwinski, and Pawel Urzyczyn. Higher-order pushdown trees are easy. *FoSSaCS*, LNCS 2303 pp 205–222. Springer, 2002.
24. Naoki Kobayashi and C.-H. Luke Ong. A type system equivalent to the modal mu-calculus model checking of HORS. In *LICS*, pages 179–188. IEEE, 2009.
25. John M. Lucassen and David K. Gifford. Polymorphic effect systems. In Jeanne Ferrante and P. Mager, editors, *POPL*, pages 47–57. ACM Press, 1988.
26. Kenneth L. McMillan. *Symbolic model checking*. Kluwer, 1993.
27. Christian Mossin. Higher-order value flow graphs. *PLILP*, LNCS 1292, pp 159–173. Springer, 1997.
28. Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of program analysis (2. corr. print)*. Springer, 2005.
29. Benjamin C. Pierce. *Advanced Topics in Types and Programming Languages*. The MIT Press, 2004.
30. Stefan Schwoon. *Model-Checking Pushdown Systems*. PhD thesis, Technische Universität München, 2002.
31. A. Prasad Sistla, et al. The complementation problem for büchi automata with appplications to temporal logic. *Theor. Comput. Sci.*, 49:217–237, 1987.
32. Peter Thiemann. Formalizing resource allocation in a compiler. *Types in Compilation*, LNCS 1473, pp. 178–193. Springer, 1998.
33. W. Thomas. Languages, automata and logic. In A. Salomaa and G. Rozenberg, editors, *Handbook of Formal Languages*, vol 3. Springer, 1997.
34. Igor Walukiewicz. Pushdown processes: Games and model checking. *CAV*, LNCS 1102, pp 62–74. Springer, 1996.

A Examples

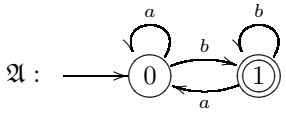
Example 1. Consider the following definition:

$$f \;=\; o(b)\;;\; o(a)\;;\; f$$

Suppose that we want to verify the property:

> *Every finite trace generated by f ends with b. Every infinite trace generated by f contains infinite many bs.*

We can use the following extended Büchi automaton $\mathfrak{A}$ to formalize this property.



We have:

$$L(\mathfrak{A}) = (a^*b^+)^+ \cup (a^*b)^\omega \;.$$

By the definition of $\sim$, we have that $\mathcal{Q}$ consists of four equivalence classes: the set of empty word ($[\epsilon] = \{\epsilon\}$), the set of non-empty words consists of a ($[a] = a^+$), the set of words ending with a and containing at least one b ($[ba] = (a+b)^*a - a^+ = (a+b)^*b(a+b)^*a$), the set of words ending with b ($[b] = (a+b)^*b$). Further, the set $\mathcal{C} = \{(C, D) \mid C, D \in \mathcal{Q} \wedge CD = C, DD = D\}$ is as follows:

$$\begin{aligned}&\{([\epsilon],[\epsilon]),([a],[\epsilon]),([a],[a]),([b],[\epsilon]),\\ &\;([b],[b]),([ba],[\epsilon]),([ba],[a]),([ba],[ba])\}\end{aligned}$$

and abstractions of $L(\mathfrak{A})$ are:

$$\begin{aligned}\mathcal{U}_L &= \{[b]\} \in \mathcal{M}_* \\ \mathcal{V}_L &= \{([b],[\epsilon]),([b],[b]),([ba],[ba])\} \in \mathcal{M}_{\leq\omega} \;.\end{aligned}$$

By using the Büchi type and effect system defined in Figure 3, we have that $(\mathcal{U}, \mathcal{V}) = (\emptyset, (\{[ba]\})^\omega)$ is a Büchi effect of f. Further,

$$\mathcal{V} = \alpha_{\leq\omega}((\gamma_*(\{[ba]\}))^\omega) = \{([b],[b]),([ba],[ba])\} \;.$$

So, $T(e_f) \subseteq \gamma_{\leq\omega}(\mathcal{V}) \subseteq \gamma_{\leq\omega}(\mathcal{V}_L)$. That is, all traces generated by f satisfy the target property. This coincides with our observation that f does not generate any finite traces and the only infinite trace generated by f is $(ba)^\omega$ which contains infinite many bs.

Example 2. Consider the following C-like program:

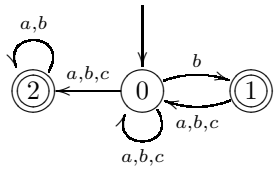
```
0   #define TIMEOUT 65536
1   while (true) {
2     i = 0;
3     while (i++ < TIMEOUT && s != 0) {
4       unsigned int s = auth(); /* o(a); */
5     }  /* o(c); */
6     work(); /* o(b); */
7   }
```

We would like to verify that line 6 is executed infinitely often under the fairness assumption that the while loop 3 always terminates. To this end, we can annotate the above program by uncommenting the event-issuing commands and abstract the so annotated program as the definition:

$$\begin{aligned} f &= g\,;\, o(b)\,;\, f \\ g &= (o(a)\,;\, g)\ ?\ o(c) \end{aligned}$$

We are then interested in the property "infinitely many b" assuming that "infinitely often c" (fairness) or equivalently: "infinitely many b or finitely many c." This property can be readily expressed as the following Büchi automaton

$\mathfrak{A}$:



By the definition of $\sim$, we have that the set $\mathcal{Q}$ consists of the following equivalence classes:

$$\begin{aligned} &[\epsilon] = \{\epsilon\} \qquad [a] = \{a\} \qquad [b] = \{b\} \qquad [c] = \{c\} \\ &[aa] = a^+a \qquad\qquad [ba] = (a+b)^+a - [aa] \\ &[ab] = a^+b \qquad\qquad [bb] = (a+b)^+b - [ab] \\ &[cb] = (a+c)^*c(a+c)^*b \\ &[bcb] = (a+b+c)^*c(a+b+c)^*b - [cb] \\ &[cca] = (a+c)^+c \ \cup\ (a+c)^*c(a+c)^*a \\ &[bca] = (a+b+c)^+c \ \cup \\ &\qquad\qquad (a+b+c)^*c(a+b+c)^*a - [cca]\ . \end{aligned}$$

Further, the set $\mathcal{C}$ consists of the following pairs:

$$\begin{array}{llll}
([\epsilon],[\epsilon]) & ([a],[\epsilon])\ \ o & ([b],[\epsilon]) & ([c],[\epsilon]) \\
([aa],[\epsilon]) & ([ba],[\epsilon]) & ([ab],[\epsilon]) & ([bb],[\epsilon]) \\
([cb],[\epsilon]) & ([bcb],[\epsilon]) & ([cca],[\epsilon]) & ([bca],[\epsilon]) \\
([aa],[aa]) & ([ba],[aa]) & ([ba],[ba]) & ([bb],[bb]) \\
([bcb],[bb]) & ([bcb],[bcb]) & ([cca],[aa]) & ([cca],[cca]) \\
([bca],[aa]) & ([bca],[ba]) & ([bca],[cca]) & ([bca],[bca])
\end{array}$$

Then, the abstractions of $L(\mathfrak{A})$ are as follows:

$$\begin{aligned}
&\mathcal{U}_L = \mathcal{Q} - \{[\epsilon]\} \in \mathcal{M}_* \\
&\mathcal{V}_L = \mathcal{C} - \{([\epsilon],[\epsilon]),([cca],[cca]),([bca],[cca])\} \in \mathcal{M}_{\leq\omega}\ .
\end{aligned}$$

By using the Büchi type and effect system, we get the effect of g as the pair:

$$(\mathcal{U}_g, \mathcal{V}_g) = (\{[c],[cca]\}, \{([aa],[aa])\})\ .$$

Then, the effect of f is the pair $(\mathcal{U}_f, \mathcal{V}_f)$ given as follows:

$$(\emptyset, \{([aa],[aa]),([bca],[aa]),([bcb],[bcb]),([bca],[bca])\})\ .$$

Since $\mathcal{U}_f \subseteq \mathcal{U}_L$ and $\mathcal{V}_f \subseteq \mathcal{V}_L$, we have that the program satisfies the property.

B Proofs

Proof. (of Theorem 1) We proceed by induction on the structure of the type and effect system. The only interesting case is that for the last rule in Figure 2. Let us show that if

$$f \uparrow w \quad \text{and} \quad \Delta \vdash f\ \&\ (U, A^* \cdot V(\mathfrak{X} - X) \cup A^\omega)\ ,$$

then w is in $A^* \cdot V(\eta) \cup A^\omega$. We introduce the assignment functions:

$$\eta_n = \begin{cases} \eta[X \mapsto \Sigma^{\leq\omega}] & \text{if}\ \ n = 0\ ; \\ \eta[X \mapsto A \cdot X_{\eta_{n-1}} \cup\ V(\eta)] & \text{if}\ \ n \geq 1\ . \end{cases}$$

From Lemma 2, $\eta_0 \models \Delta, f\ \&\ (U, X)$. Assume that $\eta_n \models \Delta, f\ \&\ (U, X)$. By inductive hypothesis, we have that for all $e_f \uparrow w$, w is in

$$A \cdot X_{\eta_n}\ \cup\ V(\mathfrak{X} - X)_{\eta_n} = A \cdot X_{\eta_n}\ \cup\ V(\mathfrak{X} - X)_{\eta} = X_{\eta_{n+1}}\ .$$

Further, by Lemma 2, $\eta_{n+1} \models \Delta, f\ \&\ (U, X)$. By mathematical induction, we have that $\eta_n \models \Delta, f\ \&\ (U, X)$ for all natural numbers n. Define η_ω as

$$\eta[X \mapsto \bigcap_{n=0}^{\infty} X_{\eta_n}]\ .$$

We get $\eta_\omega \models \Delta, f\ \&\ (U, X)$. Notice that $\eta_\omega(X)$ is equal to $A^* \cdot V(\eta)\ \cup\ A^\omega$. By the definition of $\models$, w is in $A^* \cdot V(\eta) \cup A^\omega$.

Proof. (of Lemma 3) a) and b) are obvious from the definition. Property c) coincides with b) when $D = \{\epsilon\}$. Otherwise, let $w, w' \in CD^\omega$. Decompose $w = w_1 w_2 w_3 \ldots$ and $w' = w'_1 w'_2 w'_3 \ldots$ so that $w_1, w'_1 \in C$ and $w_i, w'_i \in D$ for $i > 1$. We have $w_i \sim w'_i$ for all i so any accepting run for w yields an accepting run for w' by the definition of $\sim$. Property d) is again trivial when $w \in \Sigma^*$ (choose $C = [w]$ and $D = \{\epsilon\}$) and otherwise appears already in Büchi's work. For the record, write $w = a_1 a_2 a_3 \ldots$ with $a_i \in \Sigma$ and "colour" the set $\{i, j\}$ with $i < j$ with the $\sim$-class of $a_i a_{i+1} \ldots a_{j-1}$. By Ramsey's theorem there exists an infinite set of indices $i_1 < i_2 < i_3 \ldots$ and a class D so that $[a_{i_k} \ldots a_{i_{k+1}-1}] = D$ for all $k > 0$. The claim follows with $C := [a_1 \ldots a_{i_2-1}]$.

Proof. (Proof of Lemma 4) This is trivial for $\mathcal{M}_*$ which is a powerset lattice. As for $\mathcal{M}_{\leq\omega}$, one must show that unions and intersections of closed sets are again closed. So, let $(\mathcal{V}_i)_{i\in I}$ be a family of closed sets. To argue that the union of this family is closed, suppose that $UV^\omega \cap CD^\omega \neq \emptyset$ for some (C, D) contained in that union. Then, $(C, D) \in \mathcal{V}_i$ for some i, so $(U, V) \in \mathcal{V}_i$. Since $\mathcal{V}_i$ is closed, (U, V) is contained in the union. As for the intersection, suppose that $UV^\omega \cap CD^\omega \neq \emptyset$ for some (C, D) contained in the intersection. Then, $(C, D) \in \mathcal{V}_i$ for all i, so $(U, V) \in \mathcal{V}_i$ for all i. Since each $\mathcal{V}_i$ is closed, (U, V) is contained in the intersection, too.

Proof. (Proof of Lemma 7) We need to show that $\mathcal{U} \cdot \mathcal{V}$ is closed so suppose that $UV^\omega \cap ACD^\omega \neq \emptyset$ where $A \in \mathcal{U}$ and $(C, D) \in \mathcal{V}$. We may assume that $V \neq [\epsilon]$ for otherwise the claim is trivial. Decomposing the witnessing word, we get finite words x, y where $xy \in U$ and $x \in A$. Note that $UV = U$. Thus, $U = AY$ with Y the class of y. As a result, $YV^\omega \cap CD^\omega \neq \emptyset$, so $(YV, V) \in \mathcal{V}$ by closedness and finally, $(U, V) \in \mathcal{U} \cdot \mathcal{V}$ since $U = AYV$.

C Region-Based Büchi Type and Effect System

In order to make the Büchi type and effect system given in Section 4.2 more functional and effective in programming practices, by integrating with region types [25,29,5], we extend it to a region-based Büchi type and effect system for Featherweight Java with field update. Based on this, the future goal is to develop and implement a powerful type system for Java-like languages in which (ω)-regular properties of traces can be properly characterized for verification purposes.

C.1 Syntax

The syntax of an expression e in Featherweight Java with field update is given as follows:

$$x \in Var \quad f \in Fld \quad m \in Mtd \quad C, D \in Cls$$

$$\begin{aligned} e ::= {} & \mathbf{o}(a) \mid null \mid x \mid \mathbf{new}\ C \mid x.f \mid x.f := y \mid \\ & x.m(\overline{y}) \mid \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \mid \mathbf{if}\ x = y\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 \end{aligned}$$

For the sake of simplicity, we omit primitive types and casting and assume that every expression is in let normal form. In the definition of the **if-then-else** expression, the expression $x = y$ denotes an unusual judgement between objects which is independent on booleans. The notation $\overline{y}$ denotes a sequence of variables.

Additionally, the expression **o** is used to produce appealing events. It is a global primitive procedure which is not part of Featherweight Java with filed update. It is added as annotations to programs for the purposes of property characterization.

Let $\preceq \in \mathcal{P}(Cls \times Cls)$ be the subclass relation between classes. Let $fields \in Cls \to \mathcal{P}(Fld)$ and $methods \in Cls \to \mathcal{P}(Mtd)$ be mappings from a class to its fields and methods respectively. We use $mtable \in Cls \times Mtd \rightharpoonup \overline{Var} \times Expr$ to denote the method table which assigns to each method its definition, i.e., its formal parameters (a sequence of variables) and its body (an expression). With these definitions, a program P is given as follows:

$$P = (\preceq, fields, methods, mtable)$$

Usual well-formedness conditions on methods are assumed to ensure the inheritance relation between same methods from different classes.

C.2 Operational Semantics

The operational semantics of an expression e are given in form of:

$$\boxed{(s,h) \vdash e \Downarrow v, h' \;\&\; w \qquad (s,h) \vdash e \Uparrow \;\&\; w}$$

We use $e \Downarrow v$ to denote that the evaluation of e terminates and the value is v. The notation $e \Uparrow$ means that the evaluation of e doesn't terminate. A value $v \in Val$ of an expression is a location $l \in Loc$ or $null$. A state (s, h) is consisted of a stack $s \in Var \rightharpoonup Val$ which is a partial function assigning to each variable a value and a heap $h \in Loc \rightharpoonup Cls \times (Fld \rightharpoonup Val)$ which is a partial function assigning to each location a pair of a class and values assigned to fields of this class. In addition, as side effects, an infinite or a finite trace w which is generated from the set Σ of events by ordinary concatenations is attached to each evaluation. The whole operational semantics is given as follows:

$$\textsc{Prim-a}\ \frac{}{(s,h) \vdash \mathbf{o}(a) \Downarrow null, h \;\&\; a}$$

$$\textsc{Null}\ \frac{}{(s,h) \vdash null \Downarrow null, h \;\&\; \epsilon} \qquad \textsc{Var}\ \frac{}{(s,h) \vdash x \Downarrow s(x), h \;\&\; \epsilon}$$

$$\textsc{New}\ \frac{l \notin dom(h) \qquad F = [f \mapsto null]_{f \in fields(C)}}{(s,h) \vdash \mathbf{new}\ C \Downarrow l, h[l \mapsto (C,F)] \;\&\; \epsilon}$$

$$\textsc{Get}\ \frac{s(x) = l \qquad h(l) = (C,F)}{(s,h) \vdash x.f \Downarrow F(f), h \;\&\; \epsilon}$$

$$\textsc{Set}\ \frac{\begin{array}{c} s(x) = l \qquad h(l) = (C,F) \\ h' = h[l \mapsto (C, F[f \mapsto s(y)])] \end{array}}{(s,h) \vdash x.f := y \Downarrow s(y), h' \;\&\; \epsilon}$$

$$\textsc{Call-a}\ \frac{\begin{array}{c} s(x) = l \qquad h(l) = (C, F) \\ mtable(C, m) = (\overline{x}, e) \qquad |\overline{x}| = |\overline{y}| = n \\ s' = [this \mapsto l] \cup [x_i \mapsto s(y_i)]_{i \in \{1,2,\ldots,n\}} \\ (s', h) \vdash e \Downarrow v, h' \ \&\ w \end{array}}{(s, h) \vdash x.m(\overline{y}) \Downarrow v, h' \ \&\ w}$$

$$\textsc{Let-a}\ \frac{\begin{array}{c} (s, h) \vdash e_1 \Downarrow v_1, h_1 \ \&\ w_1 \\ (s[x \mapsto v_1], h_1) \vdash e_2 \Downarrow v_2, h_2 \ \&\ w_2 \end{array}}{(s, h) \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2 \Downarrow v_2, h_2 \ \&\ w_1 \cdot w_2}$$

$$\textsc{If-a}\ \frac{s(x) = s(y) \qquad (s, h) \vdash e_1 \Downarrow v, h' \ \&\ w}{(s, h) \vdash \textbf{if}\ x = y\ \textbf{then}\ e_1\ \textbf{else}\ e_2 \Downarrow v, h' \ \&\ w}$$

$$\textsc{If-b}\ \frac{s(x) \neq s(y) \qquad (s, h) \vdash e_2 \Downarrow v, h' \ \&\ w}{(s, h) \vdash \textbf{if}\ x = y\ \textbf{then}\ e_1\ \textbf{else}\ e_2 \Downarrow v, h' \ \&\ w}$$

$$\textsc{Epsilon}\ \frac{}{(s, h) \vdash e \Uparrow\ \ \&\ \epsilon} \qquad \textsc{Prim-b}\ \frac{}{(s, h) \vdash \mathbf{o}(a) \Uparrow\ \ \&\ a}$$

$$\textsc{Call-b}\ \frac{\begin{array}{c} s(x) = l \qquad h(l) = (C, F) \\ mtable(C, m) = (\overline{x}, e) \qquad |\overline{x}| = |\overline{y}| = n \\ s' = [this \mapsto l] \cup [x_i \mapsto s(y_i)]_{i \in \{1,2,\ldots,n\}} \\ (s', h) \vdash e \Uparrow\ \ \&\ w \end{array}}{(s, h) \vdash x.m(\overline{y}) \Uparrow\ \ \&\ w}$$

$$\textsc{Let-b}\ \frac{\begin{array}{c} (s, h) \vdash e_1 \Downarrow v_1, h_1 \ \&\ w_1 \\ (s[x \mapsto v_1], h_1) \vdash e_2 \Uparrow\ \ \&\ w_2 \end{array}}{(s, h) \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2 \Uparrow\ \ \&\ w_1 \cdot w_2}$$

$$\textsc{Let-c}\ \frac{(s, h) \vdash e_1 \Uparrow\ \ \&\ w}{(s, h) \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2 \Uparrow\ \ \&\ w}$$

$$\textsc{If-c}\ \frac{s(x) = s(y) \qquad (s, h) \vdash e_1 \Uparrow\ \ \&\ w}{(s, h) \vdash \textbf{if}\ x = y\ \textbf{then}\ e_1\ \textbf{else}\ e_2 \Uparrow\ \ \&\ w}$$

$$\textsc{If-d}\ \frac{s(x) \neq s(y) \qquad (s, h) \vdash e_2 \Uparrow\ \ \&\ w}{(s, h) \vdash \textbf{if}\ x = y\ \textbf{then}\ e_1\ \textbf{else}\ e_2 \Uparrow\ \ \&\ w}$$

Equipped with the rule Epsilon, all finite prefixes of an infinite trace produced by a non-terminate evaluation are captured. As for other rules, they are defined in the usual way.

C.3 Region-Based Büchi Type and Effect System

We now sketch an integration of the Büchi type and effect system with the region type system for Java given by Beringer et al [5]. Let us first explain why this integration is interesting and useful by an example. Considering the following fragment of Java-like code:

```
class C {
  void f (String arg);
}
```

It could be refined using two different regions `r` and `r'` with Büchi effects `(U,V)` and `(U',V')` respectively as follows:

```
class C@r {
  void f (String@X arg) & (U,V);
}
class C@r' {
  void f (String@X' arg) & (U',V');
}
```

Then, an object `o` typed `C@r` expects a `String@X` as argument to `f` and `o.f()` will exhibit a `(U,V)` effect. An object `o1` typed `C@r'` expects a `String@X'` as argument to `f` and `o1.f()` will exhibit a `(U',V')` effect. In this particular case, regions denote locations at which effects are produced.

Generally, a region $r \in Reg$ is a static abstraction of concrete locations which can be considered as a set of concrete locations. A class type C can then be equipped with a set R of regions, yielding a refined type C_R that places the constraint that its members belong to one of the regions in R. We summarize these definitions and introduce new varaiabls as follows:

$$R, S \in \mathcal{P}(Reg) \quad C_R, \tau, \sigma \in (Cls \times \mathcal{P}(Reg)) \uplus \{unit\} = Typ$$

Here, the *unit* type is introduced for typing the expression $\mathbf{o}(a)$. It is easy to define the subtype relation between region-based types: $C_R <: C'_{R'}$ if and only if $C \preceq C' \wedge R \subseteq R'$. and to extend this definition to sequences of types as follows:

$$\overline{\sigma} <: \overline{\sigma'} \quad \Leftrightarrow \quad |\overline{\sigma}| = |\overline{\sigma'}| \quad \wedge \quad \forall i \in |\overline{\sigma}| \,.\, \sigma_i <: \sigma'_i$$

With respect to the subtype relation, the following field typings:

$$A^{set}, A^{get} \in Cls \times Reg \times Fld \to Typ$$

assign to each field in each region-annotated class respectively a set-type which is a contravariant type for data written to the field and a get-type which is a covariant type for data read from the field. The following well-formedness conditions on A^{set} and A^{get} are imposed:

$$A^{set}(C, r, f) <: A^{get}(C, r, f)$$

and if $D \preceq C$, then

$$A^{set}(C, r, f) <: A^{set}(D, r, f) \quad \wedge \quad A^{get}(D, r, f) <: A^{get}(C, r, f)$$

Given a Büchi automaton $\mathfrak{A}$, let $\mathcal{M}_*$ and $\mathcal{M}_{\leq\omega}$ be the Büchi abstractions for Σ^* and $\Sigma^{\leq\omega}$ respectively, as defined in Section 4.2. We define effects as: $Eff = \mathcal{M}_* \times \mathcal{M}_{\leq\omega}$. Then, the following typing rule:

$$M \in Cls \times Reg \times Fld \to \overline{Typ} \times Typ \times Eff$$

assigns to each method in each region-annotated class a functional type with an effect. The subtype relation

$$\overline{\sigma} \xrightarrow{(\mathcal{U},\mathcal{V})} \tau \quad <: \quad \overline{\sigma}' \xrightarrow{(\mathcal{U}',\mathcal{V}')} \tau'$$

between effect annotated functional types is defined as:

$$\overline{\sigma}' <: \overline{\sigma} \ \wedge \ \tau <: \tau' \ \wedge \ \mathcal{U} \subseteq \mathcal{U}' \ \wedge \ \mathcal{V} \subseteq \mathcal{V}'$$

That is, the input type is contravariant and the output type is covariant. The well-formedness condition on M is:

$$D \preceq C \quad \Rightarrow \quad M(D,r,m) <: M(C,r,m)$$

for all classes, regions, and methods.

With the above definitions, combinined with the type and effect system given in Section 4.2, we arrive at the region-based Büchi type and effect system as follows:

$$\text{T-Sub}\ \frac{\Gamma \vdash_{\mathfrak{A}} e : \tau \ \&\ (\mathcal{U}, \mathcal{V}(\mathfrak{X})) \qquad \tau <: \tau' \qquad \mathcal{U} \subseteq \mathcal{U}' \qquad \mathcal{V}(\mathfrak{X}) \sqsubseteq \mathcal{V}'(\mathfrak{X})}{\Gamma \vdash_{\mathfrak{A}} e : \tau' \ \&\ (\mathcal{U}', \mathcal{V}'(\mathfrak{X}))}$$

$$\text{T-Prim}\ \frac{}{\Gamma \vdash_{\mathfrak{A}} \mathbf{o}(a) : \mathit{unit} \ \&\ (\alpha_*(\{a\}), \emptyset)}$$

$$\text{T-Null}\ \frac{}{\Gamma \vdash_{\mathfrak{A}} \mathit{null} : C_\emptyset \ \&\ (\alpha_*(\{\epsilon\}), \emptyset)}$$

$$\text{T-Var}\ \frac{}{\Gamma, x : \tau \vdash_{\mathfrak{A}} x : \tau \ \&\ (\alpha_*(\{\epsilon\}), \emptyset)}$$

$$\text{T-New}\ \frac{}{\Gamma \vdash_{\mathfrak{A}} \mathbf{new}\ C : C_{\{r\}} \ \&\ (\alpha_*(\{\epsilon\}), \emptyset)}$$

$$\text{T-Get}\ \frac{\forall r \in R \,.\, A^{get}(C,r,f) <: \tau}{\Gamma, x : C_R \vdash_{\mathfrak{A}} x.f : \tau \ \&\ (\alpha_*(\{\epsilon\}), \emptyset)}$$

$$\text{T-Set}\ \frac{\forall r \in R \,.\, \tau <: A^{set}(C,r,f)}{\Gamma, x : C_R, y : \tau \vdash_{\mathfrak{A}} x.f := y : \tau \ \&\ (\alpha_*(\{\epsilon\}), \emptyset)}$$

$$\text{T-Call}\ \frac{\overline{\sigma}_r \xrightarrow{(\mathcal{U}_r,\mathcal{V}_r)} \tau_r = M(C,r,m) \qquad \forall r \in R \,.\, \overline{\sigma}_r \xrightarrow{(\mathcal{U}_r,\mathcal{V}_r)} \tau_r \quad <: \quad \overline{\sigma} \xrightarrow{(\mathcal{U},\mathcal{V})} \tau}{\Gamma, x : C_R, \overline{y} : \overline{\sigma} \vdash_{\mathfrak{A}} x.m(\overline{y}) : \tau \ \&\ (\mathcal{U}, \cup_{r \in R}\{X_r\})}$$

$$\text{T-Let}\ \frac{\Gamma \vdash_{\mathfrak{A}} e_1 : \tau_1 \ \&\ (\mathcal{U}_1, \mathcal{V}_1(\mathfrak{X})) \qquad \Gamma, x : \tau_1 \vdash_{\mathfrak{A}} e_2 : \tau_2 \ \&\ (\mathcal{U}_2, \mathcal{V}_2(\mathfrak{X}))}{\Gamma \vdash_{\mathfrak{A}} \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \tau_2 \ \&\ (\mathcal{U}_1 \cdot \mathcal{U}_2, \mathcal{V}_1(\mathfrak{X}) \cup \mathcal{U}_1 \cdot \mathcal{V}_2(\mathfrak{X}))}$$

$$\text{T-If}\ \frac{\Gamma, x : C_{R \cap S}, y : D_{R \cap S} \vdash_{\mathfrak{A}} e_1 : \tau \ \&\ (\mathcal{U}_1, \mathcal{V}_1(\mathfrak{X})) \qquad \Gamma, x : C_R, y : D_S \vdash_{\mathfrak{A}} e_2 : \tau \ \&\ (\mathcal{U}_2, \mathcal{V}_2(\mathfrak{X}))}{\Gamma, x : C_R, y : D_S \vdash_{\mathfrak{A}} \mathbf{if}\ x = y\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 : \tau \ \&\ (\mathcal{U}_1 \cup \mathcal{U}_2, \mathcal{V}_1(\mathfrak{X}) \cup \mathcal{V}_2(\mathfrak{X}))}$$

The typing judgement for expressions e is

$$\boxed{\Gamma \vdash_{\mathfrak{A}} e : \tau \ \&\ (\mathcal{U}, \mathcal{V}(\mathfrak{X}))}$$

with Γ the type environment, $\mathfrak{A}$ the policy Büchi automaton, τ the type of e, $\mathcal{U}$ the set of finite traces, and $\mathcal{V}(\mathfrak{X})$ the expression for infinite traces. Among the typing rules, is the following rule:

$$\text{T-Sub}\ \frac{\begin{array}{c}\Gamma \vdash_{\mathfrak{A}} e : \tau \ \&\ (\mathcal{U}, \mathcal{V}(\mathfrak{X})) \\ \tau <: \tau' \qquad \mathcal{U} \subseteq \mathcal{U}' \qquad \mathcal{V}(\mathfrak{X}) \sqsubseteq \mathcal{V}'(\mathfrak{X})\end{array}}{\Gamma \vdash_{\mathfrak{A}} e : \tau' \ \&\ (\mathcal{U}', \mathcal{V}'(\mathfrak{X}))}$$

where

$$\mathcal{V}(\mathfrak{X}) \sqsubseteq \mathcal{V}'(\mathfrak{X}) \quad \Leftrightarrow \quad \forall \eta \in \mathfrak{X} \to \mathcal{M}_{\leq \omega} \,.\, \mathcal{V}(\eta) \subseteq \mathcal{V}'(\eta)$$

Notice that in Section 4.2, there are two typing rules for function calls. That is, one rule is used to directly get the effect if the effect has been assumed in the environment and another rule is used to derive the effect with an effect assumption added into the environment. However, in order to integrate with region-based type systems, in our region-based Büchi type and effect system, we only use the following rule:

$$\text{T-Call}\ \frac{\begin{array}{c}\overline{\sigma}_r \xrightarrow{(\mathcal{U}_r, \mathcal{V}_r)} \tau_r = M(C, r, m) \\ \forall r \in R \,.\, \overline{\sigma}_r \xrightarrow{(\mathcal{U}_r, \mathcal{V}_r)} \tau_r \quad <: \quad \overline{\sigma} \xrightarrow{(\mathcal{U}, \mathcal{V})} \tau\end{array}}{\Gamma, x : C_R, \overline{y} : \overline{\sigma} \vdash_{\mathfrak{A}} x.m(\overline{y}) : \tau \ \&\ (\mathcal{U}, \cup_{r \in R}\{X_r\})}$$

That is, the set $\mathcal{U}$ of finite traces is taken from the declarations of finite traces in M. As for the set of infinite traces, we only put a set of placeholders X_r.

Further, a program P is well-typed if and only if for all classes C, regions r, and methods m such that

$$mtable(C, r, m) = (\overline{x}, e) \ \wedge \ M(C, r, m) = \overline{\sigma}_r \xrightarrow{(\mathcal{U}_r, \mathcal{V}_r)} \tau_r$$

the following typing:

$$this : C_{\{r\}}, \overline{x} : \overline{\sigma} \vdash e : \tau \ \&\ (\mathcal{U}, \mathcal{A}_r \cdot X_r \cup \mathcal{V}(\mathfrak{X} - \{X_r\}))$$

is derivable and there is an assignment $\eta : \mathfrak{X} \to \mathcal{M}_{\leq \omega}$ satisfying:

$$\eta(X_r) = \mathcal{A}_r^* \cdot \mathcal{V}(\eta) \cup \mathcal{A}_r^\omega \ \wedge \ \mathcal{V}_r \supseteq \eta(X_r)$$

That is, a program is well-typed if and only if the constraints produced by the region-based Büchi type and effect system are satisfiable with respect to region-based type and effect declarations for all classes, all regions and all methods defined in this program.