Abstract Interpretation from Büchi Automata

Martin Hofmann
LMU Munich, Germany
martin.hofmann@ifi.lmu.de

Wei Chen
University of Edinburgh, UK
wchen2@inf.ed.ac.uk

Abstract

We describe the construction of an abstract lattice from a given Buchi automata. The abstract lattice is finite and has the following key properties. (i) There is a Galois connection between it and the (infinite) lattice of languages of finite and infinite words over a given alphabet. (ii) The abstraction is faithful with respect to acceptance by the automaton. (iii) Least fixpoints and ω-iterations (but not in general greatest fixpoints) can be computed on the level of the abstract lattice.

This allows one to develop an abstract interpretation capable of checking whether finite and infinite traces of a (recursive) program are accepted by a policy automaton. It is also possible to cast this analysis in form of a type and effect system with the effects being elements of the abstract lattice.

While the resulting decidability and complexity results are known (regular model checking for pushdown systems) the abstract lattice provides a new point of view and enables smooth integration with data types, objects, higher-order functions which are best handled with abstract interpretation or type systems.

We demonstrate this by generalising our type-and-effect systems to object-oriented programs and higher-order functions.

Categories and Subject Descriptors Theory of computation [*Semantics and reasoning*]: Program reasoning

Keywords Type Systems, Type-and-Effect Systems, Temporal Properties, Liveness

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 [8, 10, 11, 13, 25, 27]. One of them, originating from type inference in functional programming languages, is the type and effect discipline [24]. 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 [26], dependency analysis [1], resource allocation analysis [33], and amortised analysis [19, 20], etc. In particular, a type and effect system was developed by Grabowski et al. [16] 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 [3] can become relevant. For instance, a guideline could be that calls to appropriate logging functions must be made again and again or that event 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 [5, 8, 13, 25], 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 remark that in essence our language is the same as the *pushdown systems* that have been studied in detail by a number of authors [5, 29, 35].

Once a satisfactory type system for such a simple language has been found, it can be combined with known techniques [4, 28] 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 [4].

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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

CSL-LICS 2014, July 14–18, 2014, Vienna, Austria.
Copyright © 2014 ACM 978-1-4503-2886-9…$15.00.
http://dx.doi.org/10.1145/2603088.2603127

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 [6] and have subsequently been used by a number of authors [12, 17, 30], they have never been used in the context of type systems and abstract interpretation.

1.1 Related work

As already mentioned, our language of parameterless procedures is equivalent to pushdown systems for which model-checking of temporal properties has been extensively studied [7, 15, 29, 35]. Pushdown systems, on the other hand, are special cases of higher-order recursion systems introduced by Knapik et al. [22] and extensively studied by Ong and his collaborators, e.g. [2, 23].

The latter work [23] also casts model checking into the form of a type system. 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. Moreover, Büchi automata are a well-established means for formulating specifications.

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. Even though the types of [23] are also based on possible transitions of a word through the automaton there are important differences, most notably the closure operation we use and the precise analysis of ω-iteration using the Ramsey theorem. These two together allow us to precompute a finite abstract lattice that faithfully represents the concrete lattice of $\leq \omega$-languages. The analysis of recursive functions, whether mutual or not, thus reduces to a standard fixpoint iteration in an abstract lattice known from abstract interpretation.

Another recent work on the use of types for properties of infinite traces is [21] 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.

Finally, we mention the work by Skalka and his coauthors which furnish type-based translations of higher-order [31] and object-oriented [32] to pushdown systems. The so obtained pushdown systems can then be fed into off-the-shelf model-checking algorithms.

The difference to our approach is that types do not contain the full information about the possible traces but only a finite abstraction which is just fine enough to decide compliance with a given policy. In this way, one may expect more succinct types, more efficient inference, and better interaction with the user. Another difference is that our approach is entirely based on type systems and abstract interpretation and as such does not rely on external model-checking software. This might make it easier to integrate our approach with certification. One can also argue that our approach is more in line with classical type and effect systems where types do not contain programs either but rather succinct abstractions akin to our effects.

2. Preliminaries

Let Σ be a finite alphabet. We write Σ^* for the set of finite words over Σ, we write $\Sigma^{\leq\omega}$ for the set of finite and infinite words, and Σ^ω for the set of infinite words. Finite words can be concatenated with finite or infinite words as usual and this extends to languages.

We assume that programs are instrumented with special commands issuing *events* from Σ. In this way, we can associate with each execution of a(n) (instrumented) program P a *trace* which is a word from $\Sigma^{\leq\omega}$. Terminating executions have finite traces (in Σ^*) whereas nonterminating executions may have finite or infinite traces.

We also assume a language $L_0 \subseteq \Sigma^{\leq\omega}$ of finite and infinite traces modelling the allowed traces. This language L_0 will be represented by a finite state automaton $\mathfrak{A}$—the policy automaton. Writing $L(P)$ for the language of possible (finite and infinite) traces and $L_0 = L(\mathfrak{A})$ for the policy language we thus are interested in determining whether or not $L(P) \subseteq L(\mathfrak{A})$. While of course such questions are in general undecidable it becomes tractable if branches in the control flow of P are overapproximated by non-determinism or at least only depend on an appropriate finite abstraction of the data relevant in the branching conditions.

We remark that the expressive power of Büchi automata strictly subsumes the linear time mu-calculus and equals that of monadic second-order logic.

We now define a finite lattice $\mathfrak{M}$ and functions

$$\gamma : \mathfrak{M} \to \mathcal{P}(\Sigma^{\leq\omega})$$
$$\alpha : \mathcal{P}(\Sigma^{\leq\omega}) \to \mathfrak{M}$$

forming a Galois connection (Theorem 2(2)).

The abstraction is faithful with respect to containment in the language of the policy automaton in the sense that $\gamma(\alpha(L(\mathfrak{A}))) = L(\mathfrak{A})$ and hence $L \subseteq L(\mathfrak{A})$ iff $\alpha(L) \subseteq \alpha(L(\mathfrak{A}))$ (Theorem 2(5)).

On the other hand, various language-theoretic operations can be represented faithfully on the level of the abstraction, in particular, union, concatenation, least fixpoints and ω-iteration. These will allow us to design a type system that computes the abstraction $\alpha(L(P))$ of a program's behaviour.

For the most part, existence of these abstract operations follows from the Galois connection; in particular, if $\alpha \circ F = F_\alpha \circ \alpha$ then $\alpha(\text{lfp}(F)) = \text{lfp}(F_\alpha)$, where lfp denotes the least fixpoint, F_α is the corresponding abstraction function of F and $\circ$ denotes functional composition. A nontrivial property of our particular abstraction that does not follow from general lattice-theoretic considerations is the fact that ω-iteration can be faithfully represented on the level of the abstractions: There is an operation $(-)^{(\omega)} : \mathfrak{M}_* \to \mathfrak{M}$ (with $\mathfrak{M}_*$ denoting the sublattice containing abstractions of languages of finite words) satisfying $\alpha(L^\omega) = \alpha(L)^{(\omega)}$ (Theorem 4).

This then allows us to represent all the language-theoretic operations arising in the construction of $L(P)$ on the level of the abstraction. Therefore, we can describe the behaviour of program parts directly with the abstractions of their behaviours—the abstractions become effects in a type-and-effect system or can be used for abstract interpretation directly.

3. Extended Büchi automata

We define a mild generalization of Büchi automata which is capable of describing languages of both finite and infinite words.

Definition 1. *An extended Büchi Automaton is a quintuple $\mathfrak{A} = (Q, \Sigma, \delta, q_0, F)$ 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 admitting an infinite run (in the usual sense) 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.*

Note that if an extended Büchi automaton accepts an infinite word then it will also accept infinitely many prefixes of that word. This means that these automata cannot accept all unions of regular and ω-regular languages.

Our philosophy is that an actually terminating trace could, if so desired, marked by issuing a special end event, say \$. The finite traces not ending in \$ and thus stemming from an idling computation should be understood as infinite traces which from some point on consist exclusively of invisible stuttering events, say $\checkmark$. Assuming further that each state has an implicit self-loop labelled $\checkmark$ our acceptance condition for those words is subsumed by the usual Büchi condition. The following theorem summarises this.

Theorem 1. *Let $L_1 \subseteq \Sigma^*$ be a regular language over Σ and $L_2 \subseteq (\Sigma \cup \{\checkmark\})^\omega$ be an ω-regular language. Suppose furthermore that whenever $w \in L_1$ and w' arises from w by inserting $\checkmark$ symbols in arbitrary positions, then $w' \in L_2$ (Of course, L_2 may contain words other than those enforced by this clause). There is an extended Büchi automaton that accepts the language $L_1\$ \cup L_2^{\checkmark}$ where $L_2^{\checkmark} \subseteq \Sigma^{\leq\omega}$ comprises all finite and infinite words that can be obtained from words in L_2 by deleting all $\checkmark$-symbols.*

Proof. Start with a Büchi automaton for L_2 and replace all $\checkmark$-edges with ϵ-edges. This automaton viewed as an extended Büchi automaton then accepts $L_2^{\checkmark}$. Form a disjoint union with an NFA for $L_1\$$ which has the additional property that no final state has an outgoing edge so that no spurious infininite words get accepted accidentally. □

3.1 Equivalence classes

Following Büchi's original work we consider equivalence relations on finite words induced by an extended Büchi automaton.

We write $q \overset{w}{\rightsquigarrow} q'$ to mean that the state q' is reachable from state q by using the finite word w. Furthermore, $q \overset{w}{\rightsquigarrow}_F q'$ denotes 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 $w \sim u$ as:

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

We then extend $\sim$ to an equivalence relation on Σ^* by adding $\epsilon \sim \epsilon$. No nonempty word is equivalent to ϵ. We write $[w]$ for the equivalence class of $w \in \Sigma^*$. Note that the quotient set $\Sigma^*/{\sim}$ contains the $\sim$-equivalence classes consisting of nonempty words and the special class $[\epsilon] = \{\epsilon\}$.

We can effectively represent a class $[w]$ by the two relations on states it induces, i.e., by $\{(q, q') \mid q \overset{w}{\rightsquigarrow} q'\}$ and $\{(q, q') \mid q \overset{w}{\rightsquigarrow}_F q'\}$. Note that these relations are independent of the choice of the representative w. This also implies an upper bound on the number of classes and a fortiori that there are finitely many.

Lemma 1. *If n is the number of states of the policy automaton inducing $\sim$, then $\Sigma^*/{\sim}$ has at most $2^{2n^2} + 1$ elements.*

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 $\Sigma^*/\sim$ a monoid.

Notice, however, that monoid multiplication is not in general the same as concatenation of languages. We have $[u][v] = [uv]$ by definition and if $x \in [u], y \in [v]$ then $xy \in [uv]$, but if $xy \in [uv]$ it need not be the case that $x \in [u]$ and $y \in [v]$. On the other hand, the language-theoretic concatenation of $[u]$ and $[v]$ is contained in a (unique) class and this class is $[u][v]$. The operatorless juxtaposition for monoid multiplication is therefore to be used with some care, but on the other hand is vindicated by general mathematical practice.

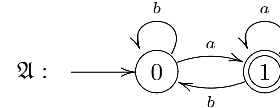
The following Lemma is a straightforward consequence of standard results about Büchi automata [34].

Lemma 2. *Fix an extended Büchi automaton $\mathfrak{A}$.*

(a) The elements of $\Sigma^*/{\sim}$ are regular languages;
(b) for all C in $\Sigma^*/{\sim}$, $C \cap L(\mathfrak{A}) \neq \emptyset$ implies $C \subseteq L(\mathfrak{A})$;
(c) for all C and D in $\Sigma^*/{\sim}$, $CD^\omega \cap L(\mathfrak{A}) \neq \emptyset$ implies $CD^\omega \subseteq L(\mathfrak{A})$;
(d) for all $w \in \Sigma^{\leq\omega}$ there exist classes $C, D \in \Sigma^*/{\sim}$ 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). It is actually a special case of Lemma 5 below so that we do not need to recall the proof here.

Example 1. *For a concrete example, consider the following automaton:*



There are four equivalence classes: $[\epsilon]$ and $[a] = (a+b)^*a$ and $[b] = b^+$ and $[ab] = (a+b)^*b - [b]$. We have $[a][a] = [b][a] = [ab][a] = [a]$ and $[b][b] = [b]$ and $[a][b] = [a][ab] = [b][ab] = [ab][b] = [ab][ab] = [ab]$. Now $(ab)^\omega \in [ab][ab]^\omega \cap [a][a]^\omega$, but $[ab][ab]^\omega \neq [a][a]^\omega$ because $a^\omega \in [a][a]^\omega \setminus [ab][ab]^\omega$.

4. Büchi Abstraction

Lemma 2 shows that given an extended Büchi automaton $\mathfrak{A}$, without affecting property checking, we can use sets of classes in $\Sigma^*/{\sim}$ to represent languages over Σ^* and sets of pairs of classes (C, D) such that $CD = C$ and $DD = D$ to represent languages over $\Sigma^{\leq\omega}$.

However, it is necessary to close such sets up under overlapping "patches"

Definition 2. *A pair (C, D) such that $CD = C$ and $DD = D$ is called a* patch. *One defines the extent of a patch (C, D) as $\gamma(C, D) = CD^\omega$, the set of all finite and infinite words that admit a decomposition $w = w_0w_1w_2\dots$ such that $w_0 \in C$ and $w_i \in D$ for $i > 0$. The extent of a set of patches $\mathcal{V}$ is defined by $\gamma(\mathcal{V}) = \bigcup_{(C,D)\in\mathcal{V}} \gamma(C, D)$. Two patches (U, V) and (C, D) meet if $\gamma(U, V) \cap \gamma(C, D) \neq \emptyset$. A set of patches $\mathcal{V}$ is closed if $\gamma(C, D) \cap \gamma(\mathcal{V}) \neq \emptyset$ implies $(C, D) \in \mathcal{V}$. We define the closure $\overline{\mathcal{V}}$*

of $\mathcal{V}$ as the least closed superset of $\mathcal{V}$. For typographical reasons, we also write $\overline{}\mathcal{V}$ for the closure.

Notice that since there are only finitely many patches we can effectively compute the closure by successively adding patches.

We call a patch (U, V) *finitary* if $V = [\epsilon]$ thus $\gamma(U, V) \subseteq \Sigma^*$.

Lemma 3. *The extent of a patch contains a finite word iff it is finitary. Finitary patches only contain finite words and they do not meet unless they are equal. Sets of finitary patches are always closed.*

Proof. No class other than $[\epsilon]$ contains the empty word, thus a patch (U, V) with $V \neq [\epsilon]$ contains infinite words only. The rest is obvious. □

4.1 The abstract lattice

Our abstract lattice now consists of the closed sets of patches ordered by inclusion. We thus define

$$\mathfrak{M} = \{\mathcal{V} \mid \mathcal{V} \text{ closed}\}$$

We also denote $\mathfrak{M}_*$ the sublattice consisting of sets of finitary patches.

We define the abstraction function $\alpha : \mathcal{P}(\Sigma^{\leq\omega}) \to \mathfrak{M}$ by

$$\alpha(L) = \overline{}\{(U, V) \mid \gamma(U, V) \cap L \neq \emptyset\}$$

In other words, to form the abstraction $\alpha(L)$ of a language L we take the smallest closed set of patches whose extents cover L.

Theorem 2. *1. The set $\mathfrak{M}$ ordered by inclusion is a complete lattice; the sublattice $\mathfrak{M}_*$ is isomorphic to the powerset lattice $\mathcal{P}(\Sigma^*/\sim)$.*

2. *The abstraction function together with the concretisation function γ form a Galois connection between $\mathfrak{M}$ and $\mathcal{P}(\Sigma^{\leq\omega})$:*

$$L \subseteq \gamma(\mathcal{V}) \iff \alpha(L) \subseteq \mathcal{V}$$

3. *$\alpha(\gamma(\mathcal{V})) = \mathcal{V}$ holds for all $\mathcal{V}$ so we have in fact a Galois* insertion
4. *The abstraction function preserves unions, least and greatest elements, but not in general intersections.*
5. *If $\mathfrak{A}$ is the underlying policy automaton then $L(\mathfrak{A}) = \gamma(\alpha(L(\mathfrak{A})))$, hence*

$$L \subseteq L(\mathfrak{A}) \iff \alpha(L) \subseteq \alpha(L(\mathfrak{A}))$$

Proof. Closed sets are closed under union and intersection. So the lattice-theoretic operations are inherited from the powerset lattice. The second part is direct from Lemma 3.

For the Galois connection assume $L \subseteq \gamma(\mathcal{V})$. To show $\alpha(L) \subseteq \mathcal{V}$ it is enough to show $(U, V) \in \mathcal{V}$ whenever (U, V) meets L since $\mathcal{V}$ is closed. So suppose $w \in \gamma(U, V)$ and $w \in L$ for some word w. By assumption, $w \in \gamma(\mathcal{V})$, so w is contained in some patch $(C, D) \in \mathcal{V}$. Thus, since $\mathcal{V}$ is closed, it must contain (U, V) as well. The converse follows directly from the obvious fact $L \subseteq \gamma(\alpha(L))$.

The fact that we have a Galois insertion, as well as preservation of least elements and unions is obvious from the definitions. Preservation of the greatest element follows from Lemma 2(d).

The last part follows from Lemma 2(c). □

Furthermore, the abstraction function preserves unions, least and greatest elements.

For a monotone operator F on a complete lattice $\mathcal{L}$ we denote lfp its least fixpoint. As is well-known this fixpoint is given by the formula $\text{lfp}(F) = \bigcap\{X \mid F(X) \subseteq X\}$ where $\bigcap$ and $\subseteq$ refer to the lattice-theoretic operations.

The following is folklore and is an easy exercise in general lattice-theoretic reasoning.

Lemma 4. *Let $\mathcal{L}, \mathcal{M}$ be complete lattices, $\alpha : \mathcal{L} \to \mathcal{M}$ and $\gamma : \mathcal{M} \to \mathcal{L}$ form a Galois insertion, i.e., both α, γ are monotone and $\alpha(L) \subseteq M \iff L \subseteq \gamma(M)$ and $\alpha(\gamma(M)) = M$, then α preserves least fixpoints in the sense that if $F : \mathcal{L} \to \mathcal{L}$ and $F_a : \mathcal{M} \to \mathcal{M}$ are monotone and such that $\alpha \circ F = F_a \circ \alpha$ then $\alpha(\mathit{lfp}(F)) = \mathit{lfp}(F_a)$.*

4.2 Concatenation and iteration

We will now define some operators on the abstract domain that track language-theoretic operations on $\mathcal{P}(\Sigma^{\leq\omega})$.

Definition 3. *For $\mathcal{U} \in \mathfrak{M}_*$ and $\mathcal{V} \in \mathfrak{M}$ we define their abstract concatenation by*

$$\mathcal{U} \cdot \mathcal{V} \quad = \quad \overline{}(\{(AC, D) \mid (A, [\epsilon]) \in \mathcal{U} \wedge (C, D) \in \mathcal{V}\})$$

Theorem 3. *1. If $\mathcal{U} \in \mathfrak{M}_*$ and $\mathcal{V} \in \mathfrak{M}$ then $\mathcal{U} \cdot \mathcal{V} \in \mathfrak{M}$.*

2. *$\alpha(L_1 L_2) = \alpha(L_1) \cdot \alpha(L_2)$.*
3. *If both $\mathcal{U}$ and $\mathcal{V}$ are finitary so is $\mathcal{U} \cdot \mathcal{V}$. The abstract concatenation operation is monotone.*

Proof. If $(AC, D)) \in \mathcal{U} \cdot \mathcal{V}$ and $(C, D) \in \mathcal{V}$ then $ACD = AC$ since $CD = C$.

For the second part it is easy to see from the definition and using the Galois connection that $\alpha(L_1 L_2) \subseteq \alpha(L_1) \cdot \alpha(L_2)$.

For the converse, since $\alpha(L_1 L_2)$ is closed, it suffices to prove

$$\{(AC, D) \mid (A, [\epsilon]) \in \mathcal{U} \wedge (C, D) \in \mathcal{V}\} \subseteq \alpha(L_1 L_2)$$

So, assume $(AC, D) \in \alpha(L_1) \cdot \alpha(L_2)$ where $(A, [\epsilon]) \in \alpha(L_1)$ and $(C, D) \in \alpha(L_2)$. It follows that $A \cap L_1 \neq \emptyset$. If (C, D) meets L_2 then (AC, D) meets $L_1 L_2$ and we are done. Otherwise, we can inductively assume that (C, D) meets $(C', D') \in \alpha(L_2)$ and $(AC', D') \in \alpha(L_1 L_2)$ has already been shown. It then follows that $(AC, D) \in \alpha(L_1 L_2)$ since (AC, D) meets (AC', D').

The third part of the lemma is direct. □

We remark that since finite iteration can be defined as a least fixpoint of operations that can be tracked on the level of the abstraction it is easy to define an operator $(-)^{(*)}$ on $\mathfrak{M}_*$ so that $\alpha(L^*) = \alpha(L)^{(*)}$ holds for $L \subseteq \Sigma^*$. Namely, we simply take $\mathcal{U}^{(*)} = \text{lfp}(\lambda\mathcal{X}.\alpha(\{\epsilon\}) \cup \mathcal{U} \cdot \mathcal{X})$.

The situation is different with the infinite iteration L^ω of a language $L \subseteq \Sigma^*$. It can be expressed as a greatest fixpoint of an operator that is representable on the level of the abstraction. To wit, L^ω equals the greatest fixpoint of $\lambda X.(L - \{\epsilon\}) \cdot X \cup G$ where $G = L^*$ if $\epsilon \in L$ and $G = \emptyset$, otherwise.

However, the abstraction does not in general preserve greatest fixpoints. For a concrete counterexample, consider the automaton from Example 1 and let $L = \{a\}$, we have $\alpha(L) = \{([a], [\epsilon])\}$ and

$$\alpha(L)^\omega = \{([a], [a]), ([ab], [ab])\}$$

Yet, the greatest fixpoint of $(\lambda\mathcal{X}.\alpha(L) \cdot \mathcal{X})$ also contains $([ab], [b])$.

Fortunately, it is possible to track infinite iteration on the level of the abstraction but this is a nontrivial result that again requires the use of the Ramsey theorem.

Definition 4. *For $\mathcal{U} \in \mathfrak{M}_*$ we define the abstract ω-iteration by*

$$\mathcal{U}^{(\omega)} = \alpha(\gamma(\mathcal{U})^\omega)$$

It is clear that if it is at all possible to track ω-iteration then this definition works but whether it is is not yet clear. We also remark that despite its seemingly nonconstructive definition $\mathcal{U}^{(\omega)}$ can easily be computed using, e.g. Büchi nonemptiness by noticing that $CD^\omega \cap \mathcal{U}^{(\omega)}$ is ω regular.

The required use of Ramsey's theorem is encapsulated in the following combinatorial lemma.

Lemma 5. *Let $(L_i)_{i\in I}$ be a family of classes (from $\Sigma^*/\sim$) and put $P = \prod_{i\in I} L_i \subseteq \Sigma^{\leq\omega}$, i.e., P comprises finite or infinite words of the form $w_1w_2w_3\dots$ 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 $U = L_1 \dots L_{n-1}$ and $V = [\epsilon]$. Otherwise, use Ramsey's theorem as in the proof of Lemma 2 to obtain a sequence of indices $i_1 < i_2 < i_3 < i_4 < \dots$ and classes U, V where $V \neq [\epsilon]$ and $VV = V$, $UV = U$ such that $w_1w_2\dots w_{i_1} \in U$, $w_{i_1+1}\dots w_{i_2} \in V$ and $w_{i_2+1}\dots w_{i_3} \in V$ and so on. It follows that $U = L_1L_2\dots L_{i_1}$ and $V = L_{i_k+1}\dots L_{i_{k+1}}$ for $k \geq 1$ and thus $P \subseteq UV^\omega$ as required. □

We are now in a position to assert the desired correctness of abstract ω-iteration which constitutes our main technical result.

Theorem 4. *For any $L \subseteq \Sigma^*$ one has*

$$\alpha(L^\omega) = \alpha(L)^{(\omega)}$$

The operator $(-)^{(\omega)}$ is monotone.

Proof. The direction $\subseteq$ is direct from the definition; for the converse assume $(U, V) \in \alpha(L)^{(\omega)} = \alpha(\gamma(\alpha(L))^\omega)$. Since $\alpha(L^\omega)$ 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 $w = w_1w_2\dots$ where $w_i \in \gamma(\alpha(L))$. Define $L_i := [w_i]$ and apply Lemma 5 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 $w'_1w'_2\dots$ is then contained in $L^\omega \cap U'V'^\omega$, so $(U', V') \in \alpha(L^\omega)$ and, finally, $(U, V) \in \alpha(L^\omega)$ since $\alpha(L^\omega)$ is closed and $UV^\omega \cap U'V'^\omega \neq \emptyset$. □

5. Applications

We begin by the definition of a type system for a simple language with recursive procedures and nondeterministic branching. It would be possible to include state and data-dependent branching, but since the type system we design is oblivious to those we refrain from doing so.

5.1 Recursive procedures

The syntax of expressions is: $e ::= o(a) \mid f \mid e_1\,;\,e_2 \mid e_1\,?\,e_2$ 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.

A formal semantics is given in Appendix A. Here, we merely assume that two judgments $e \downarrow w$ for $w \in \Sigma^*$ and $e \uparrow w$ for $w \in \Sigma^{\leq\omega}$ have been defined with the following intended meaning. If $e \downarrow w$ holds then there exists a terminating execution of e producing the finite trace w. If $e \uparrow w$ then there exists a non-terminating execution of e producing the finite or infinite trace w. All possible executions of e are captured by these two judgments. The judgements can be defined in a variety of ways, e.g. using inductive/coinductive definitions, by translation to pushdown systems [29], or using finite approximations. For a detailed elaboration of the third option we refer to our report [18].

5.2 Büchi types

A pair $(\mathcal{U}, \mathcal{V})$ where $\mathcal{U} \in \mathfrak{M}_*$ and $\mathcal{V} \in \mathfrak{M}$ is called a Büchi effect. A Büchi effect approximates the traces of an expression e if whenever $e \downarrow w$ then $w \in \gamma(\mathcal{U})$ and if $e \uparrow w$ then $w \in \gamma(\mathcal{V})$.

Intuitively, our typing judgement will derive judgements of the form $\vdash e\&(\mathcal{U}, \mathcal{V})$ with the idea that if such judgement is derivable then $(\mathcal{U}, \mathcal{V})$ approximates e. Given this intuition, typing rules like PRIM, IF are self-explanatory if we ignore for now the $(\mathfrak{X})$ decorations. Rule SEQ takes the fact into account that in a sequential composition $e_1; e_2$ even a finite trace of e_1 dominates if e_1 does not terminate.

In order to overcome the obstacle with recursive procedures discussed in the introduction we need to parameterise Büchi effects with abstractions of traces (finite or infinite) of non-terminating procedures. To this end, we introduce formal expressions of the form

$$\bigcup_{X\in\mathfrak{X}} (\mathcal{A}_X \cdot X) \,\cup\, \mathcal{B}$$

where $\mathfrak{X}$ is a finite set of variables. We use notation like $\mathcal{V}(\mathfrak{X})$ for such expressions.

If η maps variables from $\mathfrak{X}$ to elements from $\mathfrak{M}$ and $\mathcal{V}(\mathfrak{X})$ is an expression over $\mathfrak{X}$ then $\mathcal{V}(\eta) \in \mathfrak{M}$ denotes the value obtained by replacing a variable X with $\eta(X)$ and evaluating.

We extend concatenation and union to these formal expressions in the obvious way so that in particular $(\mathcal{U} \cdot \mathcal{V})(\eta) = \mathcal{U} \cdot (\mathcal{V}(\eta))$ and $(\mathcal{V}_1 \cup \mathcal{V}_2)(\eta) = \mathcal{V}_1(\eta) \cup \mathcal{V}_2(\eta)$. By extension, we also call a pair $(\mathcal{U}, \mathcal{V}(\mathfrak{X}))$ with $\mathcal{U} \in \mathfrak{M}_*$ a Büchi effect.

An environment Δ binds procedures f to pairs $(\mathcal{U}, X)$ with $\mathcal{U} \in \mathfrak{M}_*$ and X a variable. A typing judgement takes the form

$$\Delta \vdash e \,\&\, (\mathcal{U}, \mathcal{V}(\mathfrak{X}))$$

where Δ is an environment, e is an expression whose (non-primitive) procedures are all bound in Δ and $(\mathcal{U}, \mathcal{V}(\mathfrak{X}))$ is a Büchi effect and the variables $\mathfrak{X}$ are also bound in Δ.

PRIM

$$\frac{}{\Delta \vdash o(a) \,\&\, (\alpha(\{a\}), \emptyset)}$$

SEQ

$$\frac{\Delta \vdash e_1 \,\&\, (\mathcal{U}_1, \mathcal{V}_1(\mathfrak{X})) \qquad \Delta \vdash e_2 \,\&\, (\mathcal{U}_2, \mathcal{V}_2(\mathfrak{X}))}{\Delta \vdash e_1\,;\,e_2 \,\&\, (\mathcal{U}_1 \cdot \mathcal{U}_2, \mathcal{V}_1(\mathfrak{X}) \cup \mathcal{U}_1 \cdot \mathcal{V}_2(\mathfrak{X}))}$$

IF

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

CALL-A

$$\frac{}{\Delta, f \,\&\, (\mathcal{U}, X) \vdash f \,\&\, (\mathcal{U}, X)}$$

CALL-B

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

Figure 1. The Büchi Type and Effect System

To illustrate the remaining rules dealing with procedure, let us consider the example $m = o(b)\ ?\ o(a); m$. We derive the judgement

$$m\&(\alpha(a^*b), X) \vdash o(b)?o(a); m\&(\alpha(b)\cup\alpha(a)\cdot\alpha(a^*b), \alpha(a)\cdot X)$$

where, of course, a^*b was "cleverly chosen" and will in practice be found by automatic inference. Using the fact that $\alpha(b) \cup \alpha(a) \cdot (\alpha(a^*b)) = \alpha(a^*b)$ and rule CALL-B we can now conclude the expected judgement

$$\vdash m() \ \& \ (\alpha(a^*b), \alpha(a)^{(\omega)})$$

where we may note that $\alpha(a)^{(\omega)} = \alpha(a^\omega)$ by Theorems 2, 3, 4, and the remark about finite iteration after the proof of Theorem 3.

In general, we can offer the following intuition for rule CALL-B. The premise says that an infinite trace of e_f will either be captured by $\mathcal{V}(\mathfrak{X} - X)$ or else begin with a finite prefix $\mathcal{A}$ followed by a call to f. Thus, an infinite trace of f will either go through the $\mathcal{A}$ loop a finite number of times (possible not at all) and then evolve according to $\mathcal{V}(\mathfrak{X} - X)$ or keep doing $\mathcal{A}$ forever.

Next, we formulate the soundness property of our type system.

Definition 5. *An environment Δ is justified if for all $f \ \& \ (\mathcal{U}, X)$ in Δ one has $\Delta \vdash e_f \ \& \ (\mathcal{U}, \mathcal{A} \cdot X \cup \mathcal{V}(\mathfrak{X} - X))$ for some $\mathcal{A} \in \mathfrak{M}_*$, and expression $\mathcal{V}(\mathfrak{X} - X)$. An assignment function η* satisfies *an environment Δ if whenever $f \ \& \ (\mathcal{U}, X)$ in Δ and $f \uparrow w$ then $w \in \eta(X)$. We write $\eta \models \Delta$ to mean that the environment Δ is justified and that the assignment function η satisfies Δ.*

We can now state our soundness theorem as follows.

Theorem 5 (Soundness)**.** *Given an environment Δ and an assignment function η such that $\eta \models \Delta$ then whenever $\Delta \vdash e \ \& \ (U, V(\mathfrak{X}))$ for an expression e, then we have: $e \downarrow w$ implies $w \in U$ and $e \uparrow w$ implies $w \in V(\eta)$.*

The proof is by induction on derivations; since we have not formally defined traces there is no point giving detail here; instead we refer to [18].

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

Theorem 6 (Completeness)**.** *The judgements $\vdash f : (\alpha(\{w \mid f \downarrow w\}), \alpha(\{w \mid f \uparrow w\}))$ are derivable for each f.*

Now, in order to certify that all traces of a given program with main entry point f are accepted by the policy automaton it suffices to derive $\vdash f \ \&(\mathcal{U}, \mathcal{V})$ and to check that $\mathcal{U} \cup \mathcal{V} \subseteq \alpha(L(\mathfrak{A}))$.

5.3 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 $\mathfrak{M}$ 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 $\mathfrak{M}$.

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. Lemma 1 gives a single exponential bound on the number of classes. The resulting (single-)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.

5.4 Extended Example

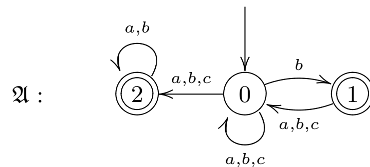
Consider the following C-like program:

```
0   #define TIMEOUT 65536
1   while (true) {
2     int i,s; i = s = 0;
3     while (i++ < TIMEOUT && s == 0) {
4       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:

$$f \ = \ g \,;\, o(b) \,;\, f \qquad\qquad g \ = \ (o(a) \,;\, g) \ ? \ o(c)$$

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:



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

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

$$[bca] = (a+b+c)^+c \ \cup$$
$$(a+b+c)^*c(a+b+c)^*a - [cca]\,.$$

Further, we have the following patches:

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

Then, the abstraction $\alpha(L(\mathfrak{A}))$ comprises all patches *except*

$$\{([\epsilon],[\epsilon]),([cca],[cca]),([bca],[cca])\}$$

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

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

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 $\mathcal{U}_f \cup \mathcal{V}_f \subseteq \alpha(\mathfrak{A})$ the program satisfies the desired fairness property as expected.

The following two subsections provide directions for future work; the results announced there have not been completely elaborated yet. We include them to demonstrate the potential of our method for possible extensions.

5.5 Region-Based Büchi Type and Effect System

We sketch an integration of the Büchi type and effect system with the region type system for Java given by Beringer et al [4]. 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:

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

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

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

Given a Büchi automaton $\mathfrak{A}$, let $\mathcal{M}_*$ and $\mathcal{M}_{\le\omega}$ be the Büchi abstractions for Σ^* and $\Sigma^{\le\omega}$ respectively, as defined in Section 4. We define effects as: $Eff = \mathcal{M}_* \times \mathcal{M}_{\le\omega}$. 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

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

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, we arrive at the region-based Büchi type and effect system as follows:

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

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

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

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

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

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

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

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

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

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

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. Definition

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

is used in the rule T-SUB. Notice that in Section 4, 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 rule T-CALL. 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. Notice that for different methods in different classes these placeholders range over $\mathfrak{X}$ and are indexed by their regions.

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

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

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 $\eta : \mathfrak{X} \rightarrow \mathcal{M}_{\leq\omega}$ 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.

5.6 Higher-order functions

We extend our language with parameters and higher-order functions so that terms are now given by the grammar:

$$e ::= x \mid x \,?\, y \mid o(a) \mid x \,;\, y \mid \textit{let}\ x = e_1\ \textit{in}\ e_2 \mid \textit{rec}\ f\ x.e$$

where x ranges over variables. We still do not model basic types and thus use nondeterministic choice in place of variables. The primitive event-issuing commands are as before. As in the previous section, we assume let-normal-form.

The construct *rec* $f\ x.e$ stands for a recursive function with argument x, body e, and recursive call f. For example, *rec* $f\ x.o(a) \,?\, x \,;\, f$ denotes a function which runs its argument x an undetermined number of times followed by an a event or else runs x ad infinitum. Of course, as before, we understand the non-determinism as arising from the abstraction of concrete data.

We now use a type-and-effect system with types given by the following grammar:

$$\tau ::= \textit{unit} \mid \bigwedge_{i \in I} \tau_1 \stackrel{\epsilon_i}{\rightarrow} \tau_2$$

where I is a finite index set and the ϵ_i are Büchi effects (with variables as before). A typing context Γ binds variables to types; the typing judgement takes the form

$$\Gamma \vdash e : \tau \,\&\, \epsilon$$

and expresses that given the bindings in Γ the evaluation of expression has effect ϵ and—if it terminates—produces a result of type τ. The typing rules are standard except for the following recursion rule where the premise is supposed to hold for each $i \in I$ and the variable $\mathcal{X}$ should not appear in any of Γ and the τ_i, τ_i'.

$$\text{T-REC}\ \frac{\Gamma, x{:}\tau_i, f{:} \bigwedge_{i \in I} \tau_i \stackrel{(\mathcal{U}_i, \mathcal{A}_i \cdot \mathcal{X} \cup \mathcal{B}_i)}{\longrightarrow} \tau_i' \vdash e : \tau_i' \,\&\, (\mathcal{U}_i, \mathcal{A}_i \cdot \mathcal{X} \cup \mathcal{B}_i)}{\Gamma \vdash \textit{rec}\ x\ f.e : \bigwedge_{i \in I} \tau_i \stackrel{(\mathcal{U}_i, \mathcal{A}_i^{(*)} \mathcal{B}_i \cup \mathcal{A}_i^{(\omega)})}{\longrightarrow} \tau_i' \,\&\, ([\epsilon], [\epsilon])}$$

We will give details including a formal statement and proof of type soundness and applications in a future paper.

6. Conclusions

We have shown how to obtain a finite abstraction of the lattice of languages of finite and infinite words over a fixed alphabet. The abstraction is parametrized by a fixed Büchi automaton formalising a desired policy. We have shown that the abstraction is fine enough to retain all relevant information for deciding whether or not the traces of a given program would be accepted by the policy automaton. We have also shown how the language-theoretic operations of union, concatenation, least-fixed point, Kleene star, and, finally ω-iteration can be adequately represented on the level of the abstraction.

As an application of these results, 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.

We have shown how the type system allows to express nontrivial fairness properties arising from the abstraction of loops that are known to terminate. We have sketched extensions of our simple type system to class-based object-oriented languages and also, albeit more briefly, with higher-order functions.

A. Trace Semantics

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 2. We write

$$\frac{}{o(a) \Downarrow a} \qquad \frac{}{o(a) \Uparrow a} \qquad \frac{}{e \Uparrow \epsilon} \qquad \frac{e_f \Downarrow w}{f \Downarrow w} \qquad \frac{e_f \Uparrow w}{f \Uparrow w}$$

$$\frac{e_1 \Downarrow w \qquad e_2 \Downarrow u}{e_1 \,;\, e_2 \Downarrow w \cdot u} \qquad \frac{e_1 \Downarrow w \qquad e_2 \Uparrow u}{e_1 \,;\, e_2 \Uparrow w \cdot u} \qquad \frac{e_1 \Uparrow w}{e_1 \,;\, e_2 \Uparrow w}$$

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

Figure 2. The Observed Trace Semantics

$e \Downarrow w$ to mean that the finite trace 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 6 (Trace Semantics)**.** *For all expressions* e *and extended traces* w *in* $(\Sigma \uplus \{\checkmark\})^{\leq\omega}$*,*

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

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 [9] 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.

Acknowledgments

This research is supported by the DFG-funded project "Verification of polymorphic noninterference for mobile code" (PolyNI).

References

[1] M. Abadi, A. Banerjee, N. Heintze, and J. G. Riecke. A core calculus of dependency. In A. W. Appel and A. Aiken, editors, *POPL*, pages 147–160. ACM, 1999. ISBN 1-58113-095-3.

[2] K. Aehlig, J. G. de Miranda, and C.-H. L. Ong. The monadic second order theory of trees given by arbitrary level-two recursion schemes is decidable. In P. Urzyczyn, editor, *TLCA*, volume 3461 of *Lecture Notes in Computer Science*, pages 39–54. Springer, 2005. ISBN 3-540-25593-1.

[3] B. Alpen and F. B. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2(3):117–126, 1987.

[4] L. Beringer, R. Grabowski, and M. Hofmann. Verifying pointer and string analyses with region type systems. *To appear in Computer Languages, Systems and Structures*, 2013.

[5] A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model-checking. In A. W. Mazurkiewicz and J. Winkowski, editors, *CONCUR*, volume 1243 of *Lecture Notes in Computer Science*, pages 135–150. Springer, 1997. ISBN 3-540-63141-0.

[6] J. R. Büchi. On a decision method in restricted second order arithmetic. In *Proc. Congress on Logic, Method, and Philosophy of Science*, pages 1–12, Stanford, CA, USA, 1962. Stanford University Press.

[7] O. Burkart and B. Steffen. Model checking the full modal mu-calculus for infinite sequential processes. *Theor. Comput. Sci.*, 221(1-2):251–270, 1999.

[8] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In Emerson and Sistla [14], pages 154–169. ISBN 3-540-67770-4.

[9] P. Cousot and R. Cousot. Inductive definitions, semantics and abstract interpretation. In R. Sethi, editor, *POPL*, pages 83–94. ACM Press, 1992. ISBN 0-89791-453-8.

[10] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In R. M. Graham, M. A. Harrison, and R. Sethi, editors, *POPL*, pages 238–252. ACM, 1977.

[11] P. Cousot and R. Cousot. Abstract interpretation frameworks. *J. Log. Comput.*, 2(4):511–547, 1992.

[12] C. Dax, M. Hofmann, and M. Lange. A proof system for the linear time μ-calculus. In S. Arun-Kumar and N. Garg, editors, *FSTTCS*, volume 4337 of *Lecture Notes in Computer Science*, pages 273–284. Springer, 2006. ISBN 3-540-49994-6.

[13] E. A. Emerson and E. M. Clarke. Characterizing correctness properties of parallel programs using fixpoints. In J. W. de Bakker and J. van Leeuwen, editors, *ICALP*, volume 85 of *Lecture Notes in Computer Science*, pages 169–181. Springer, 1980. ISBN 3-540-10003-2.

[14] E. A. Emerson and A. P. Sistla, editors. *Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings*, volume 1855 of *Lecture Notes in Computer Science*, 2000. Springer. ISBN 3-540-67770-4.

[15] J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon. Efficient algorithms for model checking pushdown systems. In Emerson and Sistla [14], pages 232–247. ISBN 3-540-67770-4.

[16] R. Grabowski, M. Hofmann, and K. Li. Type-based enforcement of secure programming guidelines - code injection prevention at sap. In G. Barthe, A. Datta, and S. Etalle, editors, *Formal Aspects in Security and Trust*, volume 7140 of *Lecture Notes in Computer Science*, pages 182–197. Springer, 2011. ISBN 978-3-642-29419-8.

[17] M. Heizmann, N. D. Jones, and A. Podelski. Size-change termination and transition invariants. In R. Cousot and M. Martel, editors, *SAS*, volume 6337 of *Lecture Notes in Computer Science*, pages 22–50. Springer, 2010. ISBN 978-3-642-15768-4.

[18] M. Hofmann and W. Chen. Büchi Types for Infinite Traces and Liveness. 2014. Technical report uploaded to arxiv.org.

[19] M. Hofmann and S. Jost. Type-based amortised heap-space analysis. In P. Sestoft, editor, *ESOP*, volume 3924 of *Lecture Notes in Computer Science*, pages 22–37. Springer, 2006. ISBN 3-540-33095-X.

[20] M. Hofmann and D. Rodriguez. Automatic type inference for amortised heap-space analysis. In M. Felleisen and P. Gardner, editors, *ESOP*, volume 7792 of *Lecture Notes in Computer Science*, pages 593–613. Springer, 2013. ISBN 978-3-642-37035-9.

[21] A. Jeffrey. Ltl types frp: linear-time temporal logic propositions as types, proofs as functional reactive programs. In K. Claessen and N. Swamy, editors, *PLPV*, pages 49–60. ACM, 2012. ISBN 978-1-4503-1125-0.

[22] T. Knapik, D. Niwinski, and P. Urzyczyn. Higher-order pushdown trees are easy. In M. Nielsen and U. Engberg, editors, *FoSSaCS*, volume 2303 of *Lecture Notes in Computer Science*, pages 205–222. Springer, 2002. ISBN 3-540-43366-X.

[23] N. Kobayashi and C.-H. L. Ong. A type system equivalent to the modal mu-calculus model checking of higher-order recursion schemes. In *LICS*, pages 179–188. IEEE Computer Society, 2009. ISBN 978-0-7695-3746-7.

[24] J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In J. Ferrante and P. Mager, editors, *POPL*, pages 47–57. ACM Press, 1988. ISBN 0-89791-252-7.

[25] K. L. McMillan. *Symbolic model checking*. Kluwer, 1993. ISBN 978-0-7923-9380-1.

[26] C. Mossin. Higher-order value flow graphs. In H. Glaser, P. H. Hartel, and H. Kuchen, editors, *PLILP*, volume 1292 of *Lecture Notes in Computer Science*, pages 159–173. Springer, 1997. ISBN 3-540-63398-7.

[27] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of program analysis (2. corr. print)*. Springer, 2005. ISBN 978-3-540-65410-0.

[28] B. C. Pierce. *Advanced Topics in Types and Programming Languages*. The MIT Press, 2004. ISBN 0262162288.

[29] S. Schwoon. *Model-Checking Pushdown Systems*. PhD thesis, Technische Universität München, 2002.

[30] A. P. Sistla, M. Y. Vardi, and P. Wolper. The complementation problem for büchi automata with appplications to temporal logic. *Theor. Comput. Sci.*, 49:217–237, 1987.

[31] C. Skalka. Types and trace effects for object orientation. *Higher-Order and Symbolic Computation*, 21(3):239–282, 2008.

[32] C. Skalka, S. F. Smith, and D. V. Horn. Types and trace effects of higher order programs. *J. Funct. Program.*, 18(2):179–249, 2008.

[33] P. Thiemann. Formalizing resource allocation in a compiler. In X. Leroy and A. Ohori, editors, *Types in Compilation*, volume 1473 of *Lecture Notes in Computer Science*, pages 178–193. Springer, 1998. ISBN 3-540-64925-5.

[34] W. Thomas. Languages, automata and logic. In A. Salomaa and G. Rozenberg, editors, *Handbook of Formal Languages*, volume 3, Beyond Words. Springer, Berlin, 1997.

[35] I. Walukiewicz. Pushdown processes: Games and model checking. In R. Alur and T. A. Henzinger, editors, *CAV*, volume 1102 of *Lecture Notes in Computer Science*, pages 62–74. Springer, 1996. ISBN 3-540-61474-5.