Decidable Fragments of Matching Logic

Abstract

Matching logic has been put forward as a "lingua franca" for verifying and reasoning about programs and programming languages. We propose three increasingly powerful decidable fragments of matching logic. The largest of these fragments, called the guarded fragment, allows both fixedpoints and a restricted form of quantification. It is intended to extend the automated prover for uniform reasoning across logics in a previously developed framework, by providing a robust basis for unfolding fixedpoints and simplification.

ACM Reference Format:

. 2022. Decidable Fragments of Matching Logic. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Motivation

The recent proliferation of programming languages has brought into focus the need for a carefully streamlined language infrastructure. Computation, and therefore programming languages, have begun to pervade every domain of our being, from our day-to-day lives in our pockets and our homes, to financial instruments and quantum computing. Each of these languages are tailored to their specific domain, and yet, rightfully, users demand sophisticated tooling for optimized compilation, debugging, model checking and program verification. Implementing such tooling on a language-by-language basis is not just redundant, but also cost-prohibitive. Creating tooling that provides a depth in terms of quality and complexity, while still covering this breadth of diversity of languages and their domains requires that we systematically streamline these redundancies.

We envision an *ideal language framework*, shown in Figure 1, where a language designer must only provide a formal description of the syntax and a formal semantics for their language, from which language tooling may be automatically generated "for free" by the framework.

The $\mathbb{K}$ framework (https://kframework.org) pursues this vision. $\mathbb{K}$ provides an intuitive meta-language with which

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

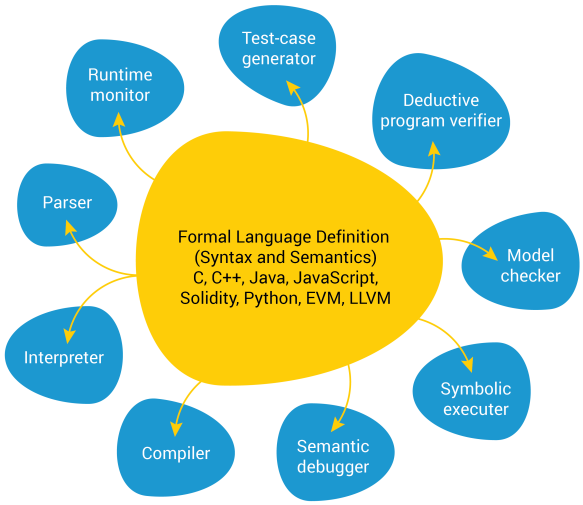

Figure 1. An ideal language framework: all language tools are generated from a formal language semantics.

language designers may define the formal semantics of their programming language as a transition system. The framework uses this to generate parsers, interpreters, deductive verifiers [11, 28], program equivalence checkers [21], among others. This approach is no longer of purely academic interest—diverse and complex programming languages have been specified in $\mathbb{K}$ including [18], Java [4], JavaScript [27], EVM [19] and x86 assembly [12]. The commercial success of verification tools built using this approach (e.g. https://runtimeverifcation.com) show that these tools are practical, valuable and in-demand.

In order to provide sound and powerful formal verification tools $\mathbb{K}$ needs a firm logical foundation. Without such a logical foundation the very meaning of what it means for a program to be "verified" or "correct" are in question.

Matching logic [7, 10, 29] provides this foundation. As shown in Figure 2, every $\mathbb{K}$ semantic definition of a language L yields a corresponding matching logic theory Γ_L, and every language task, such as executing a program or verifying a property, conducted by $\mathbb{K}$ is characterized by checking the validity of a matching logic entailment, $\Gamma_L \vdash \varphi_{task}$, where φ_{task} is the formal specification of the task in matching logic. These language tasks range from running a program in an interpreter (i.e. checking if there is a terminating execution trace for a program), to proving reachability claims. If these tools emit proof certificates, they may be checked with the matching logic proof checker [9].

Matching logic provides this foundation by creating a unifying logic, or *lingua franca*, for formal verification. Constructs for building terms, first-order quantification,

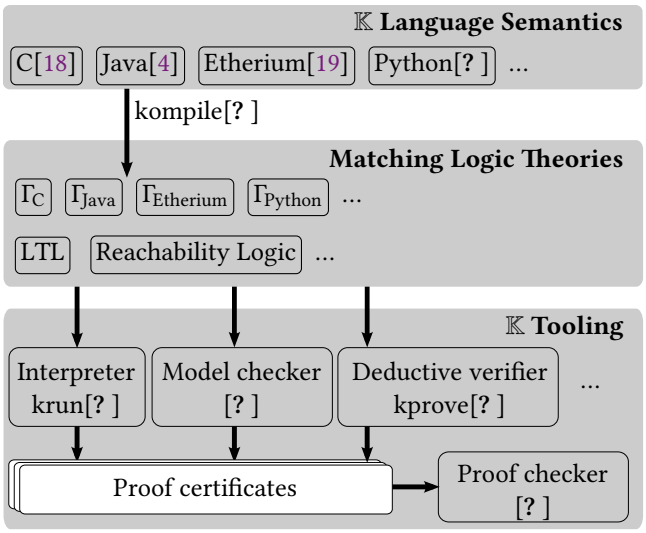

Figure 2. Matching Logic as the foundation for 𝕂

and fixedpoints, allow capturing the many formalisms important to verification, including linear temporal logic (LTL) [**?**], computation tree logic (CTL) [**?**], separation logic (SL) [**?**], and reachability logic [7, 29], as well as the language-semantics-as-a-theory produced by 𝕂. Together, these may be used to define the various language tasks described above. It does all this while maintaining minimal *representational distance*—because it preserves and respects the original syntactic and semantic structures, such as program ASTs, continuations, heaps and stacks, language semantics may be captured in a compact and modular way. In fact, the embeddings of many logics are syntactically identical to the original logics. This is in contrast to approaches that translate these to, say, first-order logic, that introduce quantifiers and other clutter.

𝕂's tools are best-effort checking for the validity of these entailments. Currently, this is done through ad-hoc reasoning developed on an as-needed basis and translation to SMT-LIB2 [2] for dispatch to the Z3 solver [14]. This leads to quite a few deficiencies—limited support for induction, users need to spell out many lemmas and simplifications, caching and optimization are at the mercy of what Z3's incremental interface will accept.

Our grand vision is to develop a matching logic solver, systematically and methodically, that unifies reasoning across embedded logics, and alleviates these problems. In [8], the authors took the first steps in this effort—they build the foundations for a unified proof framework that allows fixedpoint reasoning across logics. The authors developed a set of high-level (relative to matching logic's own proof rules) syntax-driven proof system, amenable to automation. This approach was effective—the framework was evaluated against four logical systems—first-order logic with fixedpoints, separation logic, reachability logic, and linear temporal logic (LTL).

However, as admitted in [8] itself, it would be unreasonable to hope that at such a nascent stage this framework would be able to compete with state-of-the-art domain-specific provers and decision procedures. We, however, believe that such a goal is possible and within reach in the near-term, but will likely take several years of sustained effort. Such an effort is worthwhile, because if successful could be transformative to the fields of automated deduction and program verification.

In this work, we take aim at some shortcomings of this framework. First, the core of the framework was based on an ad-hoc unfolding of fixedpoints. The boundaries of where this procedure is complete, and when it would not terminate are not known. To work around this, an arbitary bound on the number of unfoldings for fixedpoints. Further, the procedure unnecessarily tried all possible interleavings of unfoldings leading to poor performance when there were more than a few fixedpoints in the claim. Second, the framework peformed poorly with arbitary claims, getting "stuck" when not in a specific normalized form. Often the simplification procedures weren't powerful enough to re-normalize between applications of these strategies. TODO: make sure we answer precisely how these are solved later in the paper.

We propose that this heuristic be replaced with a decision procedure for a fragment of matching logic, analogously to how DPLL [13, 24], a decision procedure for propositional fragment of first-order logic, forms the core of many first-order SMT solvers [3, 5, 14]. Solvers for first-order logic are typically constructed around DPLL, an algorithm for checking the satisfiability of propositional logic formulae. A first-order formula is transformed into a propositional "skeleton" by replacing atoms with propositional variables. The DPLL algorithm then produces solutions to this skeleton—truth assignments to each of the introduced propositional variables. If any of these solutions are consistent with the atoms from the original formula, then the entire first-order formula is satisfiable.

Unfortunately, DPLL cannot directly be used in the same way as the core for matching logic automated proving because matching logic formulae (called "patterns") cannot be reduced to a propositional skeleton. This is because matching logic patterns are interpreted as the set of elements they match, unlike propositional variables which are two-valued (either true or false). In fact, translation to first-order logic or other logics is not desired in general, because the additional complexity, such as quantifiers, thwarts the nice properties such as compactness and modularity gained by using matching logic. In addition, since fixedpoint reasoning is an important aspect of program verification, the core motivation for matching logic, and it would be ideal if inductive reasoning is built into this core.

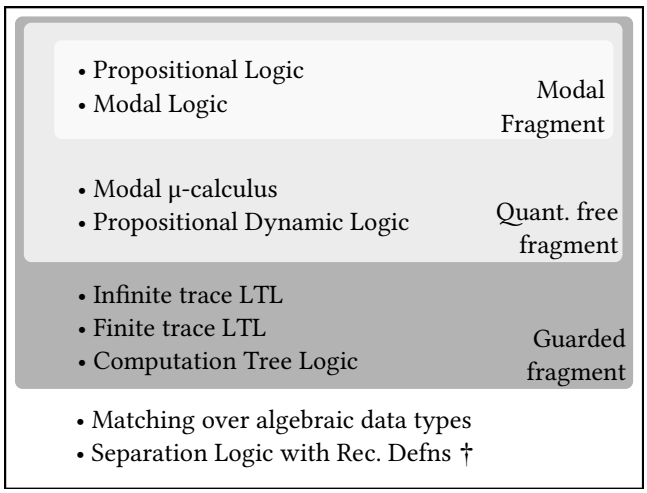

Figure 3. Proposed decidable fragments of matching logic, and some decidable embedded logics they subsume.

In this paper we propose three increasingly powerful decidable fragments of matching logic. Despite being redundant, presenting all three fragments allows us to incrementally introduce the techniques used instead of forcing the reader to deal with the complexity all at once.

The first and simplest fragment is called the *modal fragment* and allows neither fixedpoints nor quantification. To those familiar with modal logic, it may be thought of as a polyadic multimodal form of this logic. We prove that this fragment is decidable by showing that it has the small-model property—the size models for satisfiable are bound by a computable function of the size of the pattern.

The second fragment, the *quantifier-free fragment*, extends this to allow least- and greatest-fixedpoints, the modal μ-calculus extension of modal logic. This is proved decidable through an extension to the approach first show in [25].

The final fragment, called the *guarded fragment*, allows both fixedpoints and a restricted form of quantification. Again, this is proved decidable through an extension to the corresponding decision procedure for guarded fixedpoint logic. We introduce a resolution-like rule that plays the role of backtracking in DPLL. It also removes the need to iterate over all possible interpretations (as required by the algorithm presented by Grädel). and gives us a tangible object, called a refutation, in case the pattern is unsatisfiable. We may later work on a way of converting this to a formal proof. We propose using the presented decision procedure for this fragment as a new core for the proof framework in [8].

2 Matching Logic Preliminaries

Matching logic was first proposed in [29] as a unifying logic for specifying and reasoning about programming languages. An important feature of matching logic is that it makes no distinction between terms and formula. This flexibility makes many important concepts easily definable in matching logic, and allows for awkwardness free encodings of various abstractions and logics possible. For example, LTL formulae have identical syntax to their embedding in matching logic, and unification may be characterized by conjuncting two pattern built from constructors.

Matching logic formulae are called *patterns* and have a "pattern matching" semantics, in the sense that each pattern represents the set of elements that "match" it. For example, $\mathsf{cons}(42, x)$ matches lists whose first element is 42, while $\mathsf{prime} \wedge \mathsf{even}$ matches the natural 2, assuming correct axiomatizations for cons, prime, and even.

2.1 Matching Logic Syntax

For a set EVar of *element variables*, denoted $x, y, z, \ldots$, and a set SVar of *set variables*, denoted $X, Y, Z, \ldots$, we define the syntax of matching logic below.

Definition 2.1 (Matching logic signatures)**.** A matching logic *signature*, Σ is a set of symbols with an associated arity. Symbols with an arity of zero are called *constants*.

Definition 2.2 (Patterns)**.** Given a signature Σ, a countable set of element variables EVar and of set variables SVar, a matching logic *pattern* is built recursively using the following grammar:

$$\varphi := \underbrace{\sigma(\varphi_1, \ldots, \varphi_n)}_{\text{structure}} \mid \underbrace{\varphi_1 \wedge \varphi_2 \mid \neg\varphi}_{\text{logic}} \mid \underbrace{x \mid \exists x.\, \varphi}_{\text{quantification}} \mid \underbrace{X \mid \mu X.\, \varphi}_{\text{fixedpoint}}$$

where $x \in \mathsf{EVar}$, $X \in \mathsf{SVar}$ and $\sigma \in \Sigma$ has arity n, and X occurs only positively in $\mu X.\, \varphi$. That is, X may only occur under an even number of negations in φ.

We assume the standard notions for free variables, α-equivalence, and capture-free substitution $\varphi[\psi/x]$ and allow the usual syntactic sugar:

$$\top \equiv \exists x.\, x \qquad \bot \equiv \neg\top$$
$$\varphi_1 \vee \varphi_2 \equiv \neg(\neg\varphi_1 \wedge \neg\varphi_2) \qquad \varphi_1 \longrightarrow \varphi_2 \equiv \neg\varphi_1 \vee \varphi_2$$
$$\forall x.\, \varphi \equiv \neg\exists x.\, \varphi \qquad \nu X.\, \varphi \equiv \neg\mu X.\, \neg\varphi[\neg X/X]$$

$\sigma(\varphi_1, \ldots, \varphi_n)$ are called applications. Nullary applications are called constants, are denoted by using σ instead of $\sigma()$.

2.2 Semantics of Matching Logic

Unlike in FOL, matching logic patterns are interpreted as a set of elements in a model rather than a single element. Intuitively, the interpretation is the set of elements that match a pattern. For example, the constant even might have as interpretation the set of all even naturals, while greaterThan(3) may be interpreted as all integers greater than 3. Function symbols may be considered a special case of this, where

when applied to an argument the interpretation is a singleton set. Logical constructs are thought of as set operations over matched elements – for example, $\varphi \wedge \psi$ is interpreted as the intersection of elements matched by φ and ψ, while $\neg\varphi$ matches all elements *not* matched by φ. An existential $\exists x.\, \varphi(x)$ is interpreted as the union of all patterns matching $\varphi(x)$ for all valuations of x. $\mu X.\, \varphi(X)$ matches the *least* set X such that X and $\varphi(X)$ match the same elements. An important point to note here is that element variables have as evaluation exactly a single element, whereas set variables may be interpreted as any subset of the carrier set.

Definition 2.3 (Σ-models)**.** Given a signature Σ, a Σ-model is a tuple $(\mathbb{M}, \{\sigma_M\}_{\sigma\in\Sigma})$ where $\mathbb{M}$ is a set of elements called the carrier set, and $\sigma_M : M^n \to \mathcal{P}(M)$ is the interpretation of the symbol σ with arity n into the powerset of M.

We use M to denote both the model M, and it's carrier set, $\mathbb{M}$. We also tacitly use σ_M to denote the pointwise extension, $\sigma_M : \mathcal{P}(M)^n \to \mathcal{P}(M)$, defined as $\sigma_M(A_1, \ldots, A_n) \mapsto \bigcup_{a_i\in A_i} \sigma_M(a_1, \ldots, a_n)$ for all sets $A_i \subseteq M$.

Definition 2.4 (Semantics of matching logic)**.** Let $\rho : \mathsf{EVar} \cup \mathsf{SVar} \to \mathcal{P}(M)$ be a function such that $\rho(x)$ is a singleton set when $x \in \mathsf{EVar}$, called an evaluation. Then, the evaluation of a pattern φ, written $|\varphi|_{M,\rho}$ is defined inductively by:

$$
\begin{aligned}
|\sigma(\varphi_1, \ldots, \varphi_n)|_\rho &= \sigma_M(|\varphi_1|, \ldots, |\varphi_n|) \text{ for } \sigma \text{ of arity } n \\
|\varphi_1 \wedge \varphi_2|_\rho &= |\varphi_1|_\rho \cap |\varphi_2|_\rho \\
|\neg\varphi|_\rho &= M \setminus |\varphi|_\rho \\
|x|_\rho &= \rho(x) \text{ for } x \in \mathsf{EVar} \\
|\exists x.\, \varphi|_\rho &= \bigcup_{a\in M} |\varphi|_{\rho[a/x]} \\
|X|_\rho &= \rho(X) \text{ for } X \in \mathsf{SVar} \\
|\mu X.\, \varphi|_\rho &= \mathsf{LFP}(\mathcal{F})
\end{aligned}
$$

$$
\text{where} \qquad \mathcal{F}(A) = |\varphi|_{\rho[A/X]} \text{ for } A \subseteq M,
$$

$$
\text{and} \qquad \mathsf{LFP}(f) \mapsto \bigcap \{A \in \mathcal{P}M \mid f(A) \subseteq A\}
$$

takes a monotonic function to its least fixedpoint [7].

As seen, σ is interpreted as a relation. Its interpretation σ_M is not a function in the standard FOL sense. We say that σ_M is *functional*, if:

$$
\|\sigma_M(a_1, \ldots, a_n)\| = 1 \quad \text{for all } a_1 \in M_{s_1}, \ldots, a_n \in M_{s_n} \qquad \text{(functional-symbol)}
$$

2.3 Satisfiability and Validity

In this subsection, formally define satisfiability and validity in matching logic[1]. Because of the powerset interpretation of patterns, the notions of satisfiability and validity differ subtly from those in FOL. The interpretations of FOL sentences are two-valued—they must be true or false. This means that the notions of satisfiability and validity in a model coincide. However, in Matching logic patterns evaluate to a subset of the carrier set. We say a pattern is satisfiable in a model when its evaluation is non-empty, and that it is valid when its evaluation is the entire carrier set. In particular, even for closed patterns both a pattern and its negation may be satisfiable. For example, the model $\mathbb{N}$ with the usual interpretations, satisfies both even and $\neg$even (i.e. the set of odd naturals) but neither are valid.

Definition 2.5 (Satisfiability in a model)**.** We say a Σ-model M *satisfies* a Σ-pattern iff there is some evaluation ρ and an element m such that $m \in |\varphi|_{M,\rho}$. A Σ-pattern φ is *satisfiable* iff there is a model M that satisfies φ.

Definition 2.6 (Validity in a model)**.** We say a Σ-pattern is *valid* in a Σ-model M iff for all evaluations ρ, $|\varphi|_{M,\rho} = M$.

Analogously to FOL, we may define theories in matching logic. Essentially, a theory is a set of patterns, called axioms, that are valid in a model. A pattern is satisfiable modulo a theory if it is satisfiable in some model where all axioms are valid.

Definition 2.7 (Satisfiability modulo theories)**.** Let Γ be a set of Σ-patterns called *axioms*. We say φ is satisfiable modulo theory Γ if there is a model M such that each γ in Γ is valid and M satisfies φ.

Definition 2.8 (Validity modulo theories)**.** Let Γ be a set of Σ-patterns called *axioms*. We say φ is satisfiable modulo theory Γ if for all models M such that each γ in Γ is valid we have φ is valid in M.

Remark 2.9 (A note about variants of matching logic)**.** In its original formulation, matching logic had a many-sorted flavor where each symbol and pattern had a fixed sort. While it is convenient to define models that are also many-sorted, in [6] the authors point out that the many-sorted setting actually becomes an obstacle when it comes to more complex sort structures. Therefore, they proposed a much simpler, unsorted variant of matching logic called applicative matching logic (AML), where the many-sorted infrastructure is dropped and sorts are instead defined axiomatically. This also treated multi-arity applications, as syntactic sugar for nested applications. In this work, to maximize the expressivity of the fragment defined here while still avoiding the complexity of multiple sorts, we use a version of matching logic that sits between the two, allowing multi-arity applications, but without sorts. When we need to be explicit about this distinction, we will refer to this as *polyadic matching logic*. For the rest of this document unless explicitly mentioned, we will use pattern, model, etc, to refer to those concepts in polyadic matching logic although the same terms may be used in other variants of matching logic.

[1] Note that our definitions differ from [29] where only validity in a model is defined (but referred to as satisfiability). We avoid using the $\models$ notation to avoid confusion between the two.

2.4 Fragments and Meta-Properties

In general, matching logic's power and expressivity entails that the logic as a whole does not have some desirable properties. For example, because it subsumes first-order logic, the satisfiability problem must be undecidable. Further, because we can precisely pin down the standard model of the natural numbers using the fixedpoint operator, by Gödel's incompleteness theorem, it must also be incomplete.

When studying such properties in the context of matching logic, we must thus restrict ourselves to subsets of matching logic. In this section, we shall formally define what we mean by a "fragment" of matching logic, and define some properties we care about.

Definition 2.10 (Fragments of matching logic)**.** A *fragment of matching logic* is a pair $(\mathcal{P}, \mathcal{T})$ where $\mathcal{P}$ is a set of patterns and $\mathcal{T}$ is a set of theories. We say a pattern P is in a fragment if $P \in \mathcal{P}$, and a theory Γ is in a fragment if $\Gamma \in \mathcal{T}$

Fragments may be defined with any number of criteria, including the restrictions on the use of quantifiers and fixedpoints, number and arity of symbols, the number of axioms, quantifier alternation and so on.

We will now define the properties of fragments of matching logic that we will study in this document.

Definition 2.11 (Decidable fragment)**.** A fragment of matching logic, $(\mathcal{P}, \mathcal{T})$, is *decidable* if there is an algorithm for deterimining the satisfiability of any pattern $P \in \mathcal{P}$ in any theory $\Gamma \in \mathcal{T}$ in the fragment.

Notice that if $\mathcal{P}$ is closed under negation, then the validity problem for a decidable fragment is also decidable.

For proving the decidability of some fragments in this paper, we rely on a more specific property called the small-model property. This property says that every Γ-satisfiable pattern in a fragment has a model bound by a computable function on the size of the pattern. Formally:

Definition 2.12 (Small-model property)**.** A fragment of matching logic, $(\mathcal{P}, \mathcal{T})$, has the small-model property iff for every pattern $P \in \mathcal{P}$ in every theory $\Gamma \in \mathcal{T}$ if P is Γ-satisfiable then, there is some model $M \models \varphi$ whose size is bound by a computable function f on the size of φ. That is, $\|M\| \leq f(\|\varphi\|)$.

The small-model property implies that a fragment is decidable since one may simply enumerate all models of size up to $f(\|\varphi\|)$ and evaluations and check satisfiablility in each of them. The small-model property is a stonger version of another interesting property, called the finite-model property:

Definition 2.13 (Finite-model property)**.** A fragment of matching logic, $(\mathcal{P}, \mathcal{T})$, has the finite-model property iff for every pattern $P \in \mathcal{P}$ in every theory $\Gamma \in \mathcal{T}$ if P is Γ-satisfiable then, there is some model $M \models \varphi$ with finite size.

The finite-model property and decidablity are independent in the sense that a fragment may have the finite model property and yet be undecidable, or be decidable despite being infinite.

In the next sections, we will define some fragments and prove some properties about them.

We summarize the meta properties of these fragments in Table 4.

3 The Modal Fragment

The modal fragment of matching logic only allows quantifier- and fixedpoint-free patterns and the empty theory. This fragment may be regarded as a polyadic multiarity variant of modal logic.

Definition 3.1 (The modal fragment)**.** The *modal fragment* of matching logic has:

$\mathcal{P}$ = { patterns built from structure and logic }

$\mathcal{T} = \{\emptyset\}$

Our goal is to show the *small model property* of the modal fragment of matching logic, which states that if ψ is *satisfiable*, then there exists a (finite) model M with size bounded by a computable function $f(\|\psi\|)$ on the size of ψ such that $|\varphi|_M \neq \emptyset$ (or equivalently, there is an element $a \in M$ such that $a \vDash \psi$ in M).

The SMP implies decidability, because one can exhaustively search for a model for ψ up to the size bound $f(\|\psi\|)$.

Remark 3.2. In the interest of space, all proofs for this section are in Appendix **??**

Definition 3.3 (Closure)**.** Given ψ, let its *closure* $C(\psi)$ be the smallest set that contains all its sub-patterns and their negations.

The size of $C(\psi)$, written $\|C(\psi)\|$, is defined in the usual way and smaller than twice the size of ψ.

Definition 3.4 (Γ-indistinguishable)**.** Given Γ and M, we say that two elements $a, b \in M$ are Γ-*indistinguishable*, written $a \cong_\Gamma b$ or simply $a \cong b$ when Γ is understood, if $a \vDash \psi$ iff $b \vDash \psi$ for all $\psi \in \Gamma$.

Lemma 3.5. *$\cong_\Gamma$ is an equivalence relation on M.*

Proof. By directly applying the definition. □

Definition 3.6. We use $[a]_\Gamma = \{b \in M \mid a \cong_\Gamma b\}$ to denote the equivalence class of $a \in M$. We use $[A]_\Gamma = \{[a]_\Gamma \mid a \in A\}$ to denote the set of equivalence classes of all elements in $A \subseteq M$.

In the following, we drop Γ when it is understood.

Definition 3.7. Given ψ and M, we consider $C(\psi)$-indistinguishability $\cong_{C(\psi)}$ or simply $\cong$. Let $[a]$ be the equivalence class of $a \in M$. Let the *filtered model* $[M]$ contain the following:

Property \ Fragment	Empty theories			Finite theories			Recursively enumerable theories		
	Modal	Quant. free	Guarded	Modal	Quant. free	Guarded	Modal	Quant. free	Guarded
Small-model	✓[Sec.3]	✓[Sec.4]	✗	?	?	✗	✗	✗	✗
Finite-model	✓	✓	✗[Ex.??]	?	?	✗	✗[Ex. ??]	✗	✗
Decidability	✓	✓	✓	✓[Sec.5]	†[Sec.5]	✓[Sec.5]	✗[30]	✗	✗

Table 4. *The status quo:* Fragments of matching logic and their meta-prorties.
† This result has only been proved when there are no free set variables in axioms.

- the carrier set $[M]$;
- the interpretation $\sigma_{[M]}([a_1], \ldots, [a_n]) = [\sigma_M([a_1], \ldots, [a_n])]$ for symbol σ whose arity is n.

For any pattern φ, we write $\varphi_{[M]}$ to denote the interpretation of φ under any (irrelevant) valuation in $[M]$.

Remark 3.8. Note that $\sigma_{[M]}$ may not be functional, in the sense of (functional-symbol). It can even evaluate to a set of more than one equivalent classes, even when σ_M is functional.

We emphasize that $[M]$ is indeed an matching logic model, for which all it requires is a powerset interpretation of the symbols, as done in Definition 3.7. In fact, it is the nature of matching logic that makes this elegantly possible, because it allows symbols to be interpreted in the powerset. The above construction would not work for FOL, for example, if σ were a function symbol.

We point out that $\sigma_{[M]}([a_1], \ldots, [a_n])$ is not necessarily equal to $[\sigma_M(a_1, \ldots, a_n)]$. In general, $\varphi_{[M]}$ is not necessarily equal to $[\varphi_M]$ for arbitrary φ. Later, in Lemma 3.12, we show that $\varphi_{[M]} = [\varphi_M]$ holds for a selection of patterns, namely those which are in $C(\psi)$.

Lemma 3.9. *$[M]$ has at most $\sqrt{2^{\|C(\psi)\|}}$ elements.*

Note that $\|C(\psi)\| \leq 2\|\psi\|$ under a reasonable choice of the size function. Therefore,the space size of the filtered model $[M]$ is bounded by $\sqrt{2^{2\|\psi\|}} = 2^{\|\psi\|}$.

Lemma 3.10. *For any $\varphi \in C(\psi)$, the following propositions are equivalent:*

1. $\varphi_{[M]} = [\varphi_M]$;
2. $[a] \vDash \varphi_{[M]}$ *iff* $a \vDash \varphi_M$, *for all* $a \in M$.

Remark 3.11. The condition $\varphi \in C(\psi)$ is necessary in Lemma 3.10. It is used in proving that (1) implies (2), for the "only if" direction.

The following is the key lemma that links the semantics between the original and the filtered models.

Lemma 3.12. *For every $\varphi \in C(\psi)$, we have $\varphi_{[M]} = [\varphi_M]$.*

Theorem 3.13. *The modal fragment has the small model property. Formally, every satisfiable pattern ψ, without variables, $\exists$, or μ, has a model with at most $2^{\|\psi\|}$ elements.*

Proof. Let M and a satisfy $a \vDash \psi$. Applying Lemma 3.12 on ψ and a, we have that $[a] \vDash \varphi$, in the filter model $[M]$, which has at most $2^{\|\psi\|}$ elements, by Lemma 3.9. □

Theorem 3.14. *The modal fragment is decidable. Formally, given a pattern ψ without element variables, $\exists$, or μ, determining whether ψ is satisfiable is decidable.*

Proof. By Theorem 3.13, ψ is satisfiable iff there exist a model of size at most $2^{\|\psi\|}$. Given any finite size s, there are only finitely many models of size s, if we consider only the interpretations of the finitely many symbols that do occur in ψ. Therefore, we can use a brute force procedure to enumerate all possible models (with interpretations of symbols occur in ψ) up to size $2^{\|\psi\|}$ and to check the satisfiability. □

We can extend Theorems 3.13 and 3.14 to patterns that also have *set variables*, by replacing them by constant symbols. Note that all set variables are free variables, because there is no binder μ in the fragment.

Theorem 3.15. *Every satisfiable pattern φ, without element variables, $\exists$, or μ, has a model M with at most $2^{\|\varphi\|}$ elements, i.e., there exists ρ such that $|\varphi|_{M,\rho} \neq \emptyset$.*

Proof. Let $X_1, \ldots, X_k$ be all the set variables in ψ. We define k new constant symbols $c_1, \ldots, c_k$ and let $\psi' \cong \psi[c_1/X_1] \cdots [c_k/X_k]$. Note that $\|C(\psi')\| = \|C(\psi)\|$. By Theorem 3.13, ψ' has a model M with at most $2^{\|\psi\|}$ elements, such that there exists $a \in M$ satisfying $a \vDash \psi'$. We define a valuation ρ such that $\rho(X_i) = (c_i)_{M'}$, for every $1 \leq i \leq k$. By matching logic semantics, $|\psi'|_\rho = |\psi|_\rho$, so $a \in |\psi|_\rho$. Therefore, $|\psi|_\rho \neq \emptyset$. □

Theorem 3.16. *Given a pattern ψ, without element variables, $\exists$, or μ, determining whether ψ is satisfiable is decidable.*

Proof. The same as Theorem 3.14. □

4 The Quantifier-Free Fragment

The quantifier free fragment is less restrictive, allowing fixedpoints in patterns as well:

Definition 4.1 (Quantifier-Free Patterns)**.** The *quantifier-free patterns* of matching logic has:

$$\mathcal{P} = \left\{ \text{patterns built from } \boxed{\text{structure}}, \boxed{\text{logic}} \text{ and } \boxed{\text{fixedpoint}} \right\}$$
$$\mathcal{T} = \{\emptyset\}$$

This fragment also exhibits the small-model property as proved in [@sec:decidable-qf-fragment].

In this section, we prove that quantifier-free fragment is decidable and has the small model property. We do this by reducing the satisfiability problem to a solving a "parity game" (a decidable problem). Given a pattern, the parity game is built from a "tableau". The tableau is a possibly infinite tree constructed using a set of syntax driven rules. Although the tree itself is infinite, its labels range over a finite set of labels that repeat along infinite paths in a "regular" way and so has a finite representation.

In both this section and the next, we define our procedure over "positive-form" patterns – patterns where negations are pushed down as far as they can go using De Morgan's and related equivalences.

Definition 4.2 (Positive-Form Patterns)**.** Positive form patterns are defined using the syntax:

$$\begin{array}{llllll} \varphi := & \sigma(\varphi_1, \ldots, \varphi_n) & \mid \bar{\sigma}(\varphi_1, \ldots, \varphi_n) & & & \\ \mid & \varphi_1 \wedge \varphi_2 & \mid \varphi_1 \vee \varphi_2 & & & \\ \mid & x & \mid \neg x & \mid \exists x.\, \varphi & \mid & \forall x.\, \varphi \\ \mid & X & & \mid \mu X.\, \varphi & \mid & \nu X.\, \varphi \end{array}$$

where $\bar{\sigma}(\varphi_1, \ldots, \varphi_n) \equiv \neg\sigma(\neg\varphi_1, \ldots, \neg\varphi_n)$.

When σ is a nullary symbol we use σ and $\bar{\sigma}$ as shorthand for $\sigma()$ and $\bar{\sigma}()$.

We allow negation of element variables, but not set variables. This ensures that set variables may only occur positively in their binding fixedpoint. While positive form patterns allow existentials and universals, in the fragment under consideration in this section we do not allow them.

By definition, we have:

Lemma 4.3. *Every pattern is equivalent to some positive-form pattern.*

From now on, we only consider positive-form patterns and simply call them patterns.

4.1 Defintion lists and dependency orders

Definition 4.4 (Definition Lists)**.** For a quantifier-free pattern φ, a *definition list* (denoted $\mathcal{D}^\varphi$ or just $\mathcal{D}$ when φ is understood) is a mapping from each occurring set variable to the pattern by which it is bound. Since we assume well-named patterns, each set variable is bound by a unique fixedpoint pattern and such a mapping is well-defined. We use $\mathcal{D}^\varphi(X)$ to denote the fixedpoint sub-pattern corresponding to the set variable X.

Definition 4.5 (Fixedpoint Markers)**.** For a variable $X \in \mathrm{dom}(\mathcal{D}^\varphi)$, $\mathcal{D}^\varphi_X$ (or just $\mathcal{D}_X$ when φ is understood) is a *fixedpoint marker*. We call a marker a μ-marker if $\mathcal{D}^\varphi(X)$ is a μ pattern and a ν-marker otherwise. We extend the syntax of patterns to allow fixedpoint markers. These markers may be used whereever set variables may be used – in particular, they may not appear negated in positive-form patterns. We define the evaluation of fixedpoint makers as the evaluation of their corresponding fixedpoint pattern:

$$\left|\mathcal{D}^\varphi_X\right|^{\mathcal{D}}_{M,\rho} = |\mathcal{D}^\varphi(X)|_{M,\rho}$$

Since the evaluation of a pattern now also depends on its dependency list, we make the dependency list used explicit by adding it as a superscript as in $|\varphi|^{\mathcal{D}}_{M,\rho}$.

Definition 4.6 (Depends On)**.** For two fixedpoint markers $\mathcal{D}^\varphi_X$ and $\mathcal{D}^\varphi_Y$, we say $\mathcal{D}^\varphi_X$ *depends on* $\mathcal{D}^\varphi_Y$ if $\mathcal{D}^\varphi(Y)$ is a subpattern of $\mathcal{D}^\varphi(X)$.

The transitive closure of this relation is a pre-order – i.e. it is reflexive and transitive. It is also antisymmetric – for a pair of distinct markers $\mathcal{D}^\varphi_X$ and $\mathcal{D}^\varphi_Y$ we may have either $\mathcal{D}^\varphi_X$ transitively depends on $\mathcal{D}^\varphi_Y$ or $\mathcal{D}^\varphi_Y$ transitively depends on $\mathcal{D}^\varphi_X$ but not both. So, it is also a partial order.

Definition 4.7 (Dependency Orders)**.** For a pattern φ, a *dependency order*, is an (arbitary) extension of the above partial order to a total (linear) order.

Note that the partial order may extend to several total orders. For defining our parity game, it does not matter which one we choose so long as it is a total order. So, through abuse of notation, we use $\prec^\varphi$ to denote some arbitary dependency order.

Example 4.8. For the pattern, $\nu X.\, (s(X) \wedge \mu Y.\, z \vee s(Y))$, we have

$$\mathcal{D}^\varphi = \begin{cases} X & \mapsto \nu X.\, (s(X) \wedge \mu Y.\, z \vee s(Y)) \\ Y & \mapsto \mu Y.\, z \vee s(Y) \end{cases}$$

and a dependency order: $X \prec^\varphi Y$.

Example 4.9. For the pattern, $\nu X.\, s(X) \wedge \bar{p} \wedge \mu Y.\, s(Y) \wedge p$, we have

$$\mathcal{D}^\varphi = \begin{cases} X & \mapsto \nu X.\, s(X) \wedge \bar{p} \\ Y & \mapsto \mu Y.\, s(Y) \wedge p \end{cases}$$

and dependency order: $X \prec^\varphi Y$. However, this isn't the only dependency order – we also have $Y \prec^\varphi X$.

4.2 Tableau Construction

Definition 4.10. Let Γ be a set of patterns. We define Γ_σ (resp. $\Gamma_{\bar{\sigma}}$) to be the set of σ-patterns (resp. $\bar{\sigma}$-patterns) in Γ. Formally,

$$\Gamma_\sigma = \{\sigma(\varphi_1, \ldots, \varphi_n) \mid \sigma(\varphi_1, \ldots, \varphi_n) \in \Gamma\}$$
$$\Gamma_{\bar{\sigma}} = \{\bar{\sigma}(\varphi_1, \ldots, \varphi_n) \mid \bar{\sigma}(\varphi_1, \ldots, \varphi_n) \in \Gamma\}$$

$$\text{(CONFLICT)}\quad \frac{\sigma, \bar{\sigma}, \Gamma}{\mathsf{unsat}}$$

$$\text{(AND)}\quad \frac{\varphi_1 \wedge \varphi_2, \Gamma}{\varphi_1, \varphi_2, \Gamma} \qquad \text{(OR)}\quad \frac{\varphi_1 \vee \varphi_2, \Gamma}{\varphi_1, \Gamma \quad \varphi_2, \Gamma}$$

$$\text{(UNFOLD)}\quad \frac{U, \Gamma}{\varphi[U/X], \Gamma} \quad \text{where } (\mathcal{D}_U = \kappa X.\, \varphi) \text{ and } \kappa \in \{\mu, \nu\}$$

$$\text{(MU)}\quad \frac{\mu X.\, \varphi, \Gamma}{U, \Gamma} \quad \text{where } (\mathcal{D}_U = \mu X.\, \varphi) \qquad \text{(NU)}\quad \frac{\nu X.\, \varphi, \Gamma}{U, \Gamma} \quad \text{where } (\mathcal{D}_U = \nu X.\, \varphi)$$

$$\text{(CHOOSE-APP)}\quad \frac{\Gamma}{\{\sigma(\varphi_1, \ldots, \varphi_n) \rightsquigarrow \Gamma_{\bar{\sigma}} \mid \sigma(\varphi_1, \ldots, \varphi_n) \in \Gamma\}}$$

where Γ contains only σ-patterns, and $\bar{\sigma}$-patterns. (In other words, only if all other rules cannot be applied.)

$$\text{(CHOOSE-WIT)}\quad \frac{\sigma(\varphi_1, \ldots, \varphi_n) \rightsquigarrow \Gamma}{\{\sigma(\varphi_1, \ldots, \varphi_n) \rightsquigarrow \mathit{wit} \rightsquigarrow \Gamma \mid \mathit{wit} \in \mathit{Wit}(\Gamma, \sigma)\}}$$

$$\text{(APP)}\quad \frac{\sigma(\varphi_1, \ldots, \varphi_n) \rightsquigarrow \mathit{wit} \rightsquigarrow \Gamma}{\{\varphi_i, \Gamma_i^{\mathit{wit}} \mid i \in \{1, \ldots, n\}\}}$$

Figure 5. Tableau rules for the quantifier-free fragment.

Definition 4.11. Given Γ and any non-constant symbol σ of arity $n \geq 1$, we define a *witness function* $\mathit{wit}\colon \Gamma_{\bar{\sigma}} \to \{1, \ldots, n\}$ as a function that maps every pattern of the form $\bar{\sigma}(\psi_1, \ldots, \psi_n) \in \Gamma$ with a number between 1 and n, called the *witness*. Let $\mathit{Wit}(\Gamma, \sigma) = [\Gamma_{\bar{\sigma}} \to \{1, \ldots, n\}]$ denote the set of all witness functions with respect to Γ and σ. Let $\Gamma_i^{\mathit{wit}} = \{\psi_i \mid \bar{\sigma}(\psi_1, \ldots, \psi_n) \in \Gamma \text{ and } \mathit{wit}(\bar{\sigma}(\psi_1, \ldots, \psi_n)) = i\}$.

Remark 4.12. When $\Gamma_{\bar{\sigma}} = \emptyset$, we assume there is a unique witness function denoted $\mathit{wit}_\emptyset$, whose domain is empty. This is mainly for technical convenience.

We use $\|A\|$ to denote the cardinality of any set A.

Remark 4.13. Suppose σ has arity n and $\|\Gamma_{\bar{\sigma}}\| = m$ is finite. Then $\|\mathit{Wit}(\Gamma, \sigma)\| = n^m$. In particular, if σ is a unary symbol, i.e., $n = 1$, then $\|\mathit{Wit}(\Gamma, \sigma)\| = 1$.

Definition 4.14. A *tableau sequent* is one of the following:

1. a finite nonempty pattern set Γ;
2. $\Gamma \rightsquigarrow \sigma(\varphi_1, \ldots, \varphi_n)$, where $\sigma(\varphi_1, \ldots, \varphi_n) \in \Gamma$;
3. $\Gamma \rightsquigarrow \mathit{wit} \rightsquigarrow \sigma(\varphi_1, \ldots, \varphi_n)$, where $\sigma(\varphi_1, \ldots, \varphi_n) \in \Gamma$ and $\mathit{wit} \in \mathit{Wit}(\Gamma, \sigma)$.
4. unsat

Definition 4.15 (Tableaux)**.** Fix a definition list $\mathcal{D}$ for ψ. A *tableau* for ψ is a possibly infinite labeled tree (T, L). We denote its nodes as $\mathsf{Nodes}(T)$ and the root node is $\mathit{root}(T)$. The labeling function $L\colon \mathsf{Nodes} \to \mathsf{Sequents}$ associates every node of T with a sequent, such that the following conditions are satisfied:

1. $L(\mathit{root}(T)) = \{\psi\}$;
2. For every $s \in \mathsf{Nodes}(T)$, if one of the tableau rules in $\mathcal{S}$ in [@fig:qf-tableau] can be applied (with respect to the definition list $\mathcal{D}^\psi$), and the resulting sequents are $\mathit{seq}_1, \ldots, \mathit{seq}_k$, then s has exactly k child nodes $s_1, \ldots, s_k$, and $L(s_1) = \mathit{seq}_1$, ..., $L(s_k) = \mathit{seq}_k$.

In (3), we categorize the nodes by the corresponding tableau rules that are applied. For example, if the two child nodes of s is obtained by applying (OR), then we call n an (OR) node.

Remark 4.16. For any tableau (T, L) and an (CHOOSE-APP) node $s \in \mathsf{Nodes}(T)$, its child nodes (if there are any) must be (CHOOSE-WIT) nodes, whose child nodes must be (APP) nodes. (CHOOSE-WIT) and (APP) nodes must have at least one child nodes, i.e., they are not leaf nodes.

Remark 4.17. For any tableau (T, L), the leaves of T are either labeled with inconsistent sequents, or they are (CHOOSE-APP) nodes whose labels contain no σ-patterns for any σ. For any non-leaf node, unless it is labeled with (OR) or (APP_i) for $i \in \{1, 2, 3\}$, it has exactly one child node.

4.3 Parity games

Now that we know how to construct a tableau for a quantifier-free pattern, we can derive a parity game from it.

Definition 4.18 (Parity Games)**.** A *parity game* is a tuple $(\mathrm{Pos}_0, \mathrm{Pos}_1, E, \Omega)$ where $\mathrm{Pos} = \mathrm{Pos}_0 \sqcup \mathrm{Pos}_1$ is a possibly infinite set of positions; Each Pos_i is called the *winning set* for player i. $E : \mathrm{Pos} \times \mathrm{Pos}$ is a transition relation; and $\Omega : \mathrm{Pos} \to \mathbb{N}$ defines the *parity winning condition* mapping each position to a natural number below some finite bound.

The game is played between two players, player 0 and player 1. When the game is in a position $p \in \mathrm{Pos}_i$ then it is player i's turn – i.e. player i may choose the next node to transition to form the current node, along the transition relation E.

Definition 4.19 (Plays)**.** Each game results in a (possible infinite) sequence of positions, called plays: $p_0, p_1, p_2, \ldots$. A play is finite if a player is unable to make a move. In that case that player loses. For an infinite play, we look at the sequence of parities of the vertices in the play – i.e. $\Omega(p_0), \Omega(p_1), \ldots$. Player 0 wins iff the least parity that occurs infinitely often is even, otherwise player 1 wins.

Definition 4.20. For a $(\mathrm{Pos}_0, \mathrm{Pos}_1, E, \Omega)$, a *strategy* from a position q for a player i is a subtree $(P \subseteq \mathrm{Pos}, S \subseteq E)$ such that:

1. $q \in P$
2. If a node $p \in P \cap \mathrm{Pos}_i$ then there is *exactly* one edge outward edge (p, p') in S and $p' \in P$.
3. If a node $p \notin \mathrm{Pos}_i$ and $p \in P$ then all outward edges from p in E are in S_i and $p \in W_i$.

A strategy is *winning* for a player i if Player i wins on every trace.

TODO: (cite: Infinite games on finitely coloured graphs with applicatiosn to automata on infinite trees) TODO: (cite: Tree automata, mu-calculus and determinacy)

TODO: define memoryless strategy

In the case of the particular parity game we define for quantifier-free patterns, one may think of player 0 as trying to search for a model and player 1 as trying to show that there cannot be a model.

Definition 4.21. Let $\mathcal{T} = (T, L)$ be a tableau for a quantifier-free pattern ψ. Below, we define a parity game $\mathcal{G}^{\mathcal{T}} = (\mathrm{Pos}_0^{\mathcal{T}}, \mathrm{Pos}_1^{\mathcal{T}}, E^{\mathcal{T}}, \Omega^{\mathcal{T}})$. The positions are the set of pairs from Pattern $\times$ ($\{\mathsf{sat}\} \cup$ Sequent). The positions and edge relation are inductively defined as below:

- there is a position $p = (\psi, \mathit{root}(T))$ corresponding to the root sequent.
- If $p = (\varphi, s)$ is a position, and s' is a child of s in T
  - if s is not (APP) node, then
    - there is a positon $p' = (\varphi, s')$ with $(p, p') \in E$ if the rule does not reduce φ;
    - for each φ' in the set of reductions of φ in s' there is a positon $p' = (\varphi', s')$ with $(p, p') \in E$.
  - if s is an (APP) node with $s = \sigma(\varphi_1, \ldots, \varphi_n) \rightsquigarrow \mathit{wit} \rightsquigarrow \Gamma$, then
    - if $\varphi = \sigma(\varphi_1, \ldots, \varphi_n)$ then there is a position $p' = (\varphi_i, s')$ with $(p, p') \in E$
    - if $\varphi = \bar{\sigma}(\chi_1, \ldots, \chi_n)$ and $\mathit{wit}(\varphi) = i$ then there is a position $p' = (\varphi_i, s')$ with $(p, p') \in E$
  - if a position has no children as defined by the above rules then there is a position $p' = (\varphi, \mathsf{sat})$ with $(p, p') \in E$. (this is the case when a $\bar{\sigma}$–pattern is dropped by all children of a (CHOOSE-APP) node or when (APP) is applied to a nullary σ pattern)

These positions are partitioned into Pos_0 and Pos_1 as follows:

- A position $p = (\varphi, s)$ is in Pos_0 if either:
  - s is an (or)- or (CHOOSE-WIT)-node φ and φ was transformed by the rule
  - $s = \mathsf{unsat}$
- A position $p = (\varphi, s)$ is in Pos_1 if either:
  - s is an (and)-, (CHOOSE-APP)-, or (APP)-node φ and φ was transformed by the rule
  - $s = \mathsf{sat}$
- All other positions offer no choice—they have exactly one outward edge. We arbitarily assign this to Pos_1.

We define the parity condition Ω:

- $\Omega((\nu X.\, \varphi, s)) = 2i$ if $\mathcal{D}_X$ is ith in the dependency order
- $\Omega((\mu X.\, \varphi, s)) = 2i + 1$ if $\mathcal{D}_X$ is ith in the dependency order
- $\Omega((\varphi, s)) = 2N + 2$ if where N is the size of $\mathcal{D}$

By this definition, all leaf positions are either sat or unsat.

Definition 4.22 (Pre-models & Refutations)**.** We call a winning strategy for player 0 a pre-model, and a winning strategy for player 1 a refutation.

Any play consistent with a pre-model must terminate with sat if finite. An infinite play must have an even number as lowest parity—i.e. the priority corresponding to some ν fixedpoint marker. Any play consistent with a winning strategies for player 1 terminate with unsat if finite. An infinite play must have an odd number as lowest parity—i.e. the priority corresponding to some μ fixedpoint marker.

We show that for any quantifier-free sentence ψ, it is satisfied in a model (i.e., its interpretation is nonempty) iff a pre-model exists for ψ. In the interest of space we only give an informal overview of the proof here, and refer the reader to the [Appendix @sec:qf-proofs] for the proofs in their entirety.

Our first objective is to prove that if there is a satisfying model, then every tableau for ψ contains a pre-model as a sub-tree.

Proposition 4.23. *If a quantifier-free sentence ψ is satisfied in M on a, then any tableau for ψ contains a pre-model for ψ as a sub-tree.*

We do this by constructing a strategy for player 0 while maintaining the invariant that the patterns labeling each position reachable using the strategy are satisfied by some element in the model. We show that this strategy is winning for player 0, i.e. a pre-model. This is done by showing the every move taken must either reduces the size of the patterns in the sequent or a measure called the $\mathsf{Measure}^{\mu}$, corresponding roughly to the minimum number of unfoldings

of μ patterns needed to satisfy a pattern in the model, unless the move is unfolding a ν-pattern.

Next, we show that if we have a pre-model for some quantifier-free pattern, then that pattern must be satisfiable.

Proposition 4.24. *If there exists a pre-model for a positive-from quantifier-free sentence ψ then ψ is satisfiable in the corresponding canonical model.*

In this case we construct a model, called the cannonical model, from the pre-model. We then assume that the model does not satisfy the pattern, by way of contradiction, and show that there must be a μ-play in the strategy if this model does not satisfy the pattern.

Theorem 4.25. *For any positive-form quantifier-free sentence ψ, there exists a pre-model for ψ iff ψ is satisfiable.*

Proof. By Propositions 4.23 and 4.24. □

Theorem 4.26. *For any quantifier-free pattern ψ, determining whether ψ is satisfiable is decidable.*

Proof. By Theorem 4.25, we can look for a pre-model for ψ. We will construct a tree automaton $\mathcal{A}$ on infinite trees that accepts exactly the pre-models for ψ. Then by the Emerson-Jutla theorem, it is decidable to determine whether the language accepted by $\mathcal{A}$ is empty.

$\mathcal{A}$ is constructed in two steps. Firstly, we define an *outer automaton* $\mathcal{A}_o$ that accepts the quasi-models, which are essentially the *regular trees* generated by the set of tableau rules $\mathcal{S}_{\text{mod}}$. Secondly, we define an *inner automaton* $\mathcal{A}_i$ that is a Büchi automaton on infinite words that accepts the μ-traces. Then, we combine the two automatons using the Safra deterministic construction and obtain a tree automaton that accepts precisely the pre-models for ψ. □

TODO: talk about refutations

5 The Guarded Fragment

In this section, we present the guarded fragment of matching logic. This fragment is inspired by the guarded fragment of fixedpoint logic[17], with extensions to allow for the differences between matching logic and fixedpoint logic.

Inspired by the robust decidablity properties of modal logic, guarded logics were created as a means of "taming" a logic, i.e. of restricting a logic so that it becomes decidable. This is done through syntactic restrictions on quantification. In [31] we saw that the reason for this "robust" decidability was that its models have the "tree-model property", and that this property leads to automata based decision procedures. With this insight, fragments that preserved decidability under such extensions were identified. The *guarded fragment* of first-order logic defined in [1] allows quantification over an arbitary number of variables so long as it is in the form that, informally, relates each newly introduced variable to those previously introduced. This has since been generalized in two directions. First, to allow more general guards. In *loosely guarded* first-order logic presented in [20], guards are allowed to be conjunctions of atoms, rather than just single atoms. *Packed logic* extends this further, allowing even existentials to occur in guards. In the *clique guarded* fragment of first-order logic [16], quantification is semantically restricted to cliques within the Gaifman graph of models. Second, to allow fixedpoints: guarded fixedpoint logic, loosely guarded fixedpoint logic [17], and clique-guarded fixedpoint logic [16]. extend the corresponding guarded logics to allow fixedpoints constructs. An interesting property of guarded fixedpoint logics, is that despite being decidable, they admit "infinity axioms" – axioms that are satisfiable only in infinite models.

The algorithm we present here is an extension to the one presented in [17] modified for matching logic with an important extension, to enable resolution, that we found vital to a practical implementation. We also try to empasize its relation with the tableau defined in Section 4.

For a nonempty tuple $\bar{x}$, We treat $\exists\bar{x}.\,\varphi$ and $\forall\bar{x}.\,\varphi$ as shorthand for nested quantifier patterns.

Definition 5.1 (Guarded pattern)**.** A *guarded pattern* is a closed (i.e. without any free element or set variables) positive-form pattern such that:

1. Every existential sub-pattern is of the form $\exists\bar{x}.\alpha \wedge \varphi$ and every universal sub-pattern is of the form $\forall\bar{x}.\alpha \longrightarrow \varphi$ where:
   a) α is a conjunction where each conjunct is either an element variable, or an application pattern where each argument is an element variable,
   b) every variable $x \in \bar{x}$ appears in some conjunct,
   c) for each pair of variables $x \in \bar{x}$ and $y \in \mathsf{free}(\varphi)$ there is some application $\sigma_i(\bar{z}_i)$ in α where $x, y \in \bar{z}_i$.

   We call α a guard.
2. If $\sigma(\bar{\varphi})$ is an application, then each φ_i is a conjunction of the form $\xi \wedge \bigwedge_{y\in\bar{y}} y$ where $\bar{y}$ is a (possibly empty) set of element variables and ξ is a pattern, and every element variable in $\mathsf{free}(\sigma(\varphi_i))$ is in $\bar{y}$ for some φ_i.
3. Each fixedpoint sub-pattern $\mu X.\,\varphi$ and $\nu X.\,\varphi$ has no free element variables.

5.1 Tableau Construction

While in the previous section, it was sufficient to use simple sets of patterns as sequents, we now need something more complex. Previously, each sequent in the quantifier-free tableau corresponds to a single element in the model, in the more complex guarded patterns existentials introduce additional elements we must keep track of. We now build the nessesary constructs to represent those sequents.

Definition 5.2 (Assertion)**.** An *assertion* is either:

(CONFLICT) $\dfrac{\langle \{\alpha\} \cup \Gamma; \{\mathsf{not}(\alpha)\} \cup \mathcal{B}; \mathcal{U}\rangle}{\mathsf{unsat}}$ when α is a basic assertion.

(OK) $\dfrac{\langle \{\alpha\} \cup \Gamma; \{\alpha\} \cup \mathcal{B}; \mathcal{U}\rangle}{\langle \Gamma; \mathcal{B}; \mathcal{U}\rangle}$ when α is a basic assertion.

(CONFLICT-EL) $\dfrac{\langle \mathsf{matches}(x, y), \Gamma; \mathcal{B}; \mathcal{U}\rangle}{\mathsf{unsat}}$ when $x \neq y$.

(OK-EL) $\dfrac{\langle \mathsf{matches}(x, x), \Gamma; \mathcal{B}; \mathcal{U}\rangle}{\langle \Gamma; \mathcal{B}; \mathcal{U}\rangle}$

Rules for (AND), (OR), (DEF), (MU), and (NU) identical to those in Figure 5 but lifted to the new form of sequents.

(FORALL) $\dfrac{\langle \mathsf{matches}(z, \forall \bar{x}.\, \varphi), \Gamma; \mathcal{B}; \mathcal{U}\rangle}{\langle \mathsf{inst} \cup \Gamma; \mathcal{B}; \mathsf{matches}(z, \forall \bar{x}.\, \varphi), \mathcal{U}\rangle}$
where $\mathsf{inst} := \{\mathsf{matches}(z, \varphi[\bar{y}/\bar{x}]) \mid \bar{y} \subseteq \mathcal{E}\}$

(APP) $\dfrac{\langle \{\mathsf{matches}(z, \bar{\sigma}(\bar{\varphi}))\} \cup \Gamma; \mathcal{B}; \mathcal{U}\rangle}{\langle \mathsf{inst} \cup \Gamma; \mathcal{B}; \{\mathsf{matches}(z, \bar{\sigma}(\bar{\varphi}))\} \cup \mathcal{U}; \rangle}$
where $\mathsf{matches}(z, \bar{\sigma}(\bar{\varphi}))$ is not a basic assertion.
and $\mathsf{inst} := \{\mathsf{matches}(z, \mathsf{not}(\sigma(\bar{y}))) \vee \bigvee_i \mathsf{matches}(y_i, \varphi_i) \mid \bar{y} \subseteq \mathcal{E}\}$

(CHOOSE-EX) $\dfrac{\langle \Gamma; \mathcal{B}; \mathcal{U}\rangle}{\{\alpha \rightsquigarrow \langle \Gamma; \mathcal{B}; \mathcal{U}\rangle \mid \text{for each } \alpha \in \Gamma\}}$ when all assertions in α are either existentials or applications. i.e. when no other of the above rules apply

(APP) $\dfrac{\mathsf{matches}(z, \sigma(\bar{\varphi})) \rightsquigarrow \langle \Gamma; \mathcal{B}; \mathcal{U}\rangle}{\left\{ \left\langle \begin{array}{l} \mathsf{matches}(z, \sigma(\bar{k})) \\ \wedge \bigwedge_i \mathsf{matches}(k_i, \varphi_i), \Gamma' \cup \mathcal{U}'; \mathcal{B}'; \emptyset \\ \text{for each } \bar{k} \subseteq \{z\} \cup \mathsf{free}(\bar{\varphi}) \cup (K \setminus \mathcal{E}) \end{array} \right\rangle \right\}}$
where
$\mathcal{E}' = \bar{k} \cup \{z\} \cup \mathsf{free}(\bar{\varphi})$
$\mathcal{B}' = \mathcal{B}|_{\mathcal{E}'}$, $\Gamma' = \Gamma|_{\mathcal{E}'}$, and $\mathcal{U}' = \mathcal{U}|_{\mathcal{E}'}$

(EXISTS) $\dfrac{\mathsf{matches}(z, \exists \bar{x}.\, \varphi) \rightsquigarrow \langle \Gamma; \mathcal{B}; \mathcal{U}\rangle}{\{\langle \alpha, \Gamma' \cup \mathcal{U}'; \mathcal{B}'; \emptyset\rangle\}}$
for each $\alpha \in \{\mathsf{matches}(z, \varphi[\bar{k}/\bar{x}]) \mid \bar{k} \subseteq \{z\} \cup \mathsf{free}(\bar{\varphi}) \cup (K \setminus \mathcal{E})\}$
where
$\mathcal{E}' = \mathsf{free}(\alpha)$
$\mathcal{B}' = \mathcal{B}|_{\mathcal{E}'}$, $\Gamma' = \Gamma|_{\mathcal{E}'}$, and $\mathcal{U}' = \mathcal{U}|_{\mathcal{E}'}$

(RESOLVE) $\dfrac{\langle \Gamma; \mathcal{B}; \mathcal{U}\rangle}{\langle \Gamma; \mathsf{matches}(x_0, \sigma(x_1, \ldots, x_n)) \cup \mathcal{B}; \mathcal{U}\rangle \qquad \langle \Gamma; \mathsf{matches}(x_0, \neg\sigma(x_1, \ldots, x_n)) \cup \mathcal{B}; \mathcal{U}\rangle}$
when neither $\mathsf{matches}(x, \sigma(x_1, \ldots, x_n))$ nor $\mathsf{matches}(x, \mathsf{not}(\sigma(x_1, \ldots, x_n)))$ are in $\mathcal{B}$
and $\bar{x} \subseteq \mathcal{E}$ and $\bar{x} \cap \mathsf{fresh} \neq \emptyset$.
May be applied only directly on the root node, or immediately after the application (APP), (EXISTS) or (RESOLVE) rules.

Figure 6. Tableau rules for the guarded fragment.

1. a pair of an element variable and a pattern, denoted $\mathsf{matches}(x, \varphi)$,
2. a conjunction of assertions: $\alpha_1 \wedge a_2$
3. a disjunction of assertions: $\alpha_1 \vee \alpha_2$

Informally, assertions allow us to capture that a element in the model matches a pattern.

From here on, we treat $\mathsf{matches}(x, \varphi_1 \vee \varphi_2)$ as equivalent to $\mathsf{matches}(x, \varphi_1) \vee \mathsf{matches}(x, \varphi_2)$. and $\mathsf{matches}(x, \varphi_1 \wedge \varphi_2)$ as equivalent to $\mathsf{matches}(x, \varphi_1) \wedge \mathsf{matches}(x, \varphi_2)$.

Definition 5.3 (Basic assertions)**.** *Basic assertions* are assertions of the form:

1. $\mathsf{matches}(x_0, \sigma(x_1, \ldots, x_n))$,
2. $\mathsf{matches}(x_0, \bar{\sigma}(\neg x_1, \ldots, \neg x_n))$.

where each x_i is an element variable.

Basic assertions, capture the relational interpretation of each symbol and directly specify (portions) of the model.

Definition 5.4 (Restriction)**.** The *free variables* of an assertion $\mathsf{matches}(x, \varphi)$ are $\{x\} \cup \mathsf{free}(\varphi)$. For conjunctions and disjunctions of assertion, it is the union of the free variables in each sub-pattern. For a set of assertions A and a set of element variables E, the restriction of A to E, denoted $A|_E$, is the subset of assertion in A whose free variables are a subset of E.

The use of element variables in the (APP) allow us to drop the concept of *wit*.

Definition 5.5 (Sequent)**.** A *sequent* is:

1. a tuple, $\langle \Gamma; \mathcal{B}; \mathcal{U}\rangle$, where Γ is a set of assertions, $\mathcal{B}$ is a set of basic assertions, $\mathcal{U}$ is a set of assertions whose pattern is of the form $\bar{\sigma}(\ldots)$ or $\forall \bar{x}. \ldots$.
2. $\alpha \rightsquigarrow \langle \Gamma; \mathcal{B}; \mathcal{U}\rangle$ where α is an assertion and $\Gamma, \mathcal{B}, \mathcal{U}$ are as above.
3. unsat

For a sequent v of the first two forms, we use $\Gamma(v), \mathcal{B}(v)$ and $\mathcal{U}(v)$ to denote the corresponding component of the sequent.

Informally, Γ represents the set of assertions whose combined satisfiability we are checking. $\mathcal{B}$ and $\mathcal{U}$ are sets of consistent assertions that we are using to build our model. Each free element variable in these assertions corresponds (roughly) to a distinct element in the model we are building (if one exists).

Definition 5.6 (Tableaux)**.** Fix a definition list $\mathcal{D}$ for ψ. A *tableau* for ψ is a possibly infinite labeled tree (T, L). We

denote its nodes as $\mathsf{Nodes}(T)$ and the root node is $root(T)$. The labeling function $L\colon \mathsf{Nodes} \to \mathsf{Sequents}$ associates every node of T with a sequent, such that the following conditions are satisfied:

1. $L(root(T)) = \{\langle \mathsf{matches}(x, \psi)\rangle\}$ where x is a fresh element variable;
2. For every $s \in \mathsf{Nodes}(T)$, if one of the tableau rules in $\mathcal{S}$ in Figure 6 can be applied (with respect to the definition list $\mathcal{D}^{\psi}$), and the resulting sequents are $seq_1, \ldots, seq_k$, then s has exactly k child nodes $s_1, \ldots, s_k$, and $L(s_1) = seq_1$, ..., $L(s_k) = seq_k$.

Proposition 5.7. *For any sequent built using these rules, we cannot have both* $\mathsf{matches}(x, \varphi)$ *and* $\mathsf{matches}(x, \mathsf{not}(\varphi))$ *in* $\mathcal{B}$

Proof. The root node starts with $\mathcal{B}$ empty, and therefore this invariant is maintained. Basic assertions are added to $\mathcal{B}$ only through the (RESOLVE) rule which maintains the invariant. □

Proposition 5.8. *There is a tableau for any guarded pattern*

Proof. For any assertion that is not basic, there is some rule that applies. For a basic assertion if either the assertion itself or its negation in in $\mathcal{B}$, then the (CONFLICT) or (OK) rule applies. Otherwise, there was some application of the (EXISTS), (APP) rule after which all variables in the assertion are in $\mathcal{E}$. We may build a tableau where the (RESOLVE) rule is applied for this basic assertion after this (EXISTS)/(APP) rule. □

5.2 Parity Game

As before, using this tableau we now build a parity game. In this game, player 0 may be thought of as trying to prove the satisfiablity of the pattern, and player 1 as trying to prove it unsatisfiable.

Each position in the game is a pair (a, v) where a is an assertion and v is either a sequent or sat. If $v = \mathsf{unsat}$, then a is of the form $\mathsf{matches}(x, \bot)$. If $\Gamma = \emptyset$, then a is of the form $\mathsf{matches}(x, \top)$. Otherwise, $a \in \Gamma$.

There is an edge from a position (a_0, v_0) to (a_1, v_2) if:

a) $v_1 = \mathsf{unsat}$ is a child constructed through (conflict) and (conflict-el), $a_0 = a_1$. (same as above)

b)
1. v_1 is constructed from v_0 using the (and), (or), (def), (mu), (nu), ($\overline{\text{app}}$) or (forall) rules and a_0 was modified by this rule, and a_1 is one of the newly created assertions.
2. v_0's child is created using (ok) or (ok-el) and $a_0 = a_1$ and $v_1 = \mathsf{sat}$.
3. v_1 is constructed from v_0 using (choose-ex) and $a_0 = a_1 = \alpha$ is the matched existential.
4. v_1 is constructed from v_0 using (app) or (exists) and $a_0 = \alpha$ and a_1 is the an instantiation from inst.

c) v_1 is a child constructed through any rule besides (conflict) and (conflict-el), $a_0 = a_1$ is in $\Gamma(v_0) \cup \mathcal{U}(v_0)$ and $\Gamma(v_1) \cup \mathcal{U}(v_1)$. and $v_1 = v_0$

A position $p = (a, v)$ is in Pos_0 by rules with a green rule. That is, if:

1. v's children were built using (or), (app), or (exists) rules and a was the assertion matched on by that rule; or
2. v's children were built using (resolve); or
3. $v = \mathsf{unsat}$

A position $p = (a, v)$ is in Pos_1 by rules with a blue rule. That is if:

1. v's children were built using (and), ($\overline{\text{app}}$), (forall), or (choose-ex) rules and a was the assertion matched on by that rule;
2. $v = \mathsf{sat}$

All other positions do not offer a choice, and so are arbitrarily assigned to Pos_1.

The parity condition Ω is defined as follows:

- $\Omega((e \in D_X, v)) = 2 \times i$ if D_i is a μ-marker that is ith in the dependency order.
- $\Omega((e \in D_X, v)) = 2 \times i + 1$ if D_i is a ν-marker that is ith in the dependency order.
- $\Omega((e \in \exists \bar{x}.\, \varphi, v)) = 2 \times N + 1$ where N is the number of fixedpoint markers in $\mathcal{D}$.
- $\Omega(a, v) = 2 \times N + 2$ otherwise.

Similar to Theorems~4.26 we may prove that pre-models correspond to models and that guarded patterns are decidable.

Theorem 5.9. *If a guarded pattern has a tableau with a pre-model, then it is satisfiable.*

Theorem 5.10. *If a guarded pattern has a tableau with a refutation, then it is unsatisfiable and its negation is valid.*

5.3 Working modulo theories

The tableau presented in Figure 6 may be easily extended to handle satisfiability modulo theorems for finite theories with guarded axioms.

First, we extend assertions to allow quantifying over its variable—i.e. we allow assertions of the form $\forall x.\, \mathsf{matches}(x, \varphi)$. Next, we add a new tableau rule:

$$\text{(AXIOM)} \quad \frac{\langle \forall x.\, \mathsf{matches}(x, \varphi), \Gamma; \mathcal{B}; \mathcal{U}\rangle}{\langle \mathsf{inst} \cup \Gamma; \mathcal{B}; \mathcal{U}\rangle}$$

for each axiom τ in the theory and $x \in \mathsf{free}(\Gamma \cup \mathcal{B} \cup \mathcal{U})$.

5.4 Complexity

6 Examples

Due to space constraints, we show examples of theories captured by the above fragments, as well as example tableaux in the Appendix **??**.

7 Future Work

As mentioned in Section 1, our goal and primary motivation is to use the algorithm defined for guarded patterns as the basis of an automated prover. Towards that goal, there are two important milestones we need to reach.

First, we must implement our algorithm. This may prove tricky because we have not found any previous implementations for the corresponding procedure of guarded fixedpoint logic. Indeed, it was difficulties implementing the procedure in [17] that led us to develop the (RESOLVE) rule. Without it, we had needed to search through all possible combinations of consistent basic assertions. Our plan is to produce a parity game, and then use a pre-existing solver for parity games [15] to search for pre-models or refutations. TODO: should we remove this completely?

Second, we must be able to combine this procedure with our decision procedures and semi-algorithms. This will allow us to take advantage of this decision procedure even when the pattern we are dealing with lies outside the fragment. In the world of first-order SMT solvers, this is typically done using the Nelson-Oppen combination procedure [23**?**]. One approach may be to extend the (CONFLICT-*) and (OK-*) rules to treat the SMT solver as an oracle for checking the consistency of basic assertions. This approach will likely work when the algorithm decides that the pattern is unsatisfiable. However, if the algorithm instead produces a model, we aren't guaranteed that the pattern is satisfied by that model. This is because the proof of Theorem **??** depends on the properties of guarded patterns, whereas that of Theorem **??** does not. In particular, the model produced may have inconsistencies between assertions over elements that do not share nodes in the tableau. In the case where we are able to identify the elements that produce the conflict, we may be able to synthesize guarded lemmas that allow us to rule out these models. TODO: relatively complete wrt oracle TODO: mention integration with K newline

Another important direction is to extend our decidable fragment as much as possible. TODO: take away: decidability comes from the restriction to quantification One easy candidate to to drop the requirement that fixedpoints may not have free element variables. This will involve extending fixedpoint markers to take arguments, and changes to the (DEF), (MU), and (NU) rules to work with these extended fixedpoint markers.

A second low hanging fruit stems from the observation that the proof of Theorem **??** depends on the properties of guarded patterns, whereas that of Theorem **??** does not. This implies that we can likely drop the restriction to existentials patterns when checking satisfiablilty, and the restriction to universal patterns when checking validity. Further, axioms in theories are not negated when reducing validity to satisfiability, so the restriction do existentials can be dropped for axioms when checking both the validity and satisfiability modulo theories.

A third avenue draw inspiration from other decideable logics. For example, the decidability of the packed fragment of first-order logic [22] seems to imply that we can allow nested existentials or applications in guards. That of guarded separation logic [26] implies that we could handle associative-commutive symbols, perhaps by drawing inspiration from unification algorithms. The relationship between the finite variant property and the guarded fragment is another exciting direction.

As we draw inspiration from these decision procedures we increase the complexity of our decision procedure, and that complexity increases the risk of incorrectness. To mitigate this risk we aim to produce proofs of validity from refutations. These proofs can then be checked with a small trust-base, for example by the proof checker developed in [9].

For the most part the tableau rules in Figure 6 (when viewed through the lens of a refutation) correspond directly to matching logic proof rules. The (EXISTS) may be thought of as a combination of the (∃-GENERALIZATION) followed by case analysis between the introduced variables. One tricky aspect when converting the refutations to proofs is dealing with infinite plays. By definition of a refutation, these contain a μ-marker that is unfolded infinitely often. We must somehow turn this into a finite proof. Fortunately these plays must be ω-regular, and we should be able to represent them in a finite proof through the combination of *implication contexts* introduced in [8], and the (KNASTER-TARSKI) proof rule. TODO: no need to mention knaster-tarksi rule TODO: talk about the the proof checker TODO: ... so that we don't need to trust the prover.

8 Conclusion

We presented three decision procedures for fragments of matching logic. The guarded fragment is a powerful fragment supporting both fixedpoints and some quantification and allows us to capture many theories important to program verification. We outlined next steps to be taken in order to use this procedure as the core of an automated prover.

9 To do

- Examples:
  - $(\mu X.\, z \vee s(X)) \wedge \nu X.\, \neg z \wedge \bar{s}(X)$
  - this is an interesting example because it shows the need for alpha renaming in the implementation.
- Extension to theories.
- Examples of patterns inside and outside this fragment
  - Inside: LTL, PDL, Sorts and Subsorts
  - Outside: Guarded Separation Logic, ...
- What happens when we're outside the guarded fragment? Refutations are still correct, though models produced may not make sense.

References

[1] Hajnal Andréka, István Németi, and Johan van Benthem. Modal languages and bounded fragments of predicate logic. *Journal of philosophical logic*, 27(3):217–274, 1998.

[2] Clark Barrett, Aaron Stump, Cesare Tinelli, et al. The smt-lib standard: Version 2.0. In *Proceedings of the 8th international workshop on satisfiability modulo theories (Edinburgh, England)*, volume 13, page 14, 2010.

[3] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV'11)*, pages 171–177. Springer, 2011.

[4] Denis Bogdanas and Grigore Roşu. K-java: A complete semantics of java. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 445–456, 2015.

[5] Roberto Bruttomesso, Alessandro Cimatti, Anders Franzén, Alberto Griggio, and Roberto Sebastiani. The mathsat 4 smt solver. In Aarti Gupta and Sharad Malik, editors, *Computer Aided Verification*, pages 299–303, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. ISBN 978-3-540-70545-1.

[6] Xiaohong Chen and Grigore Roşu. Applicative matching logic. Technical Report http://hdl.handle.net/2142/104616, University of Illinois at Urbana-Champaign, 2019.

[7] Xiaohong Chen and Grigore Roşu. Matching μ-logic. In *2019 34th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–13. IEEE, 2019.

[8] Xiaohong Chen, Minh-Thai Trinh, Nishant Rodrigues, Lucas Peña, and Grigore Roşu. Towards a unified proof framework for automated fixpoint reasoning using matching logic. In *PACMPL Issue OOPSLA 2020*, pages 1–29. ACM/IEEE, Nov 2020.

[9] Xiaohong Chen, Zhengyao Lin, Minh-Thai Trinh, and Grigore Roşu. Towards a trustworthy semantics-based language framework via proof generation. In *Proceedings of the 33rd International Conference on Computer-Aided Verification*. ACM, July 2021.

[10] Xiaohong Chen, Dorel Lucanu, and Grigore Roşu. Matching logic explained. *Journal of Logical and Algebraic Methods in Programming*, 120:100638, 2021. ISSN 2352-2208. doi: https://doi.org/10.1016/j.jlamp.2021.100638. URL https://www.sciencedirect.com/science/article/pii/S2352220821000018.

[11] Andrei Ştefănescu, Daejun Park, Shijiao Yuwen, Yilong Li, and Grigore Roşu. Semantics-based program verifiers for all languages. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'16)*, pages 74–91, Amsterdam, Netherlands, 2016. ACM. ISBN 978-1-4503-4444-9.

[12] Sandeep Dasgupta, Daejun Park, Theodoros Kasampalis, Vikram S Adve, and Grigore Roşu. A complete formal semantics of x86-64 user-level instruction set architecture. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1133–1148, 2019.

[13] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM (JACM)*, 7(3):201–215, 1960.

[14] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[15] Oliver Friedmann and Martin Lange. The pgsolver collection of parity game solvers. *University of Munich*, pages 4–6, 2009.

[16] Erich Grädel. Guarded fixed point logics and the monadic theory of countable trees. *Theoretical Computer Science*, 288(1):129–152, 2002.

[17] Erich Gradel and Igor Walukiewicz. Guarded fixed point logic. In *Proceedings. 14th Symposium on Logic in Computer Science (Cat. No. PR00158)*, pages 45–54. IEEE, 1999.

[18] Chris Hathhorn, Chucky Ellison, and Grigore Roşu. Defining the undefinedness of C. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 336–345, 2015.

[19] Everett Hildenbrandt, Manasvi Saxena, Nishant Rodrigues, Xiaoran Zhu, Philip Daian, Dwight Guth, Brandon Moore, Daejun Park, Yi Zhang, Andrei Stefanescu, et al. KEVM: A complete formal semantics of the ethereum virtual machine. In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, pages 204–217. IEEE, 2018.

[20] Ian Hodkinson. Loosely guarded fragment of first-order logic has the finite model property. *Studia Logica*, 70(2):205–240, 2002.

[21] Theodoros Kasampalis, Daejun Park, Zhengyao Lin, Vikram S Adve, and Grigore Roşu. Language-parametric compiler validation with application to llvm. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1004–1019, Virtual, 2021. ACM New York, NY, USA.

[22] Maarten Marx. Tolerance logic. *Journal of Logic, Language and Information*, 10(3):353–374, 2001.

[23] Greg Nelson and Derek C Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1(2):245–257, 1979.

[24] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving sat and sat modulo theories: From an abstract davis–putnam–logemann–loveland procedure to dpll(t). *J. ACM*, 53(6):937–977, November 2006. ISSN 0004-5411. doi: 10.1145/1217856.1217859. URL https://doi.org/10.1145/1217856.1217859.

[25] Damian Niwiński and Igor Walukiewicz. Games for the μ-calculus. *Theoretical Computer Science*, 163(1):99–116, 1996. ISSN 0304-3975. doi: https://doi.org/10.1016/0304-3975(95)00136-0. URL http://www.sciencedirect.com/science/article/pii/0304397595001360.

[26] Jens Pagel, Christoph Matheja, and Florian Zuleger. A decision procedure for guarded separation logic: Complete entailment checking for separation logic with inductive definitions. *arXiv preprint arXiv:2002.01202*, 2020.

[27] Daejun Park, Andrei Stefănescu, and Grigore Roşu. Kjs: A complete formal semantics of javascript. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 346–356, 2015.

[28] Grigore Roşu. K—A semantic framework for programming languages and formal analysis tools. In *Dependable Software Systems Engineering*. IOS Press, 2017.

[29] Grigore Roşu. Matching Logic. *Logical Methods in Computer Science*, Volume 13, Issue 4, December 2017. doi: 10.23638/LMCS-13(4:28)2017. URL https://lmcs.episciences.org/4153.

[30] Alasdair Urquhart. Decidability and the finite model property. *Journal of Philosophical Logic*, 10(3):367–370, 1981. ISSN 00223611,

15730433. URL http://www.jstor.org/stable/30226231.

[31] Moshe Y Vardi. Why is modal logic so robustly decidable? Technical report, 1997.