Mechanizing Matching Logic In Coq

Péter Bereczky
Eötvös Loránd University, Hungary

Xiaohong Chen
University of Illinois Urbana-Champaign, USA

Dániel Horpácsi
Eötvös Loránd University, Hungary

Lucas Peña
University of Illinois Urbana-Champaign, USA

Jan Tušil
Masaryk University, Brno, Czech Republic

Matching logic is a formalism for specifying, and reasoning about, mathematical structures, using patterns and pattern matching. Growing in popularity, it has been used to define many logical systems such as separation logic with recursive definitions and linear temporal logic. In addition, it serves as the logical foundation of the K semantic framework, which was used to build practical verifiers for a number of real-world languages. Despite being a fundamental formal system accommodating substantial theories, matching logic lacks a general-purpose, machine-checked formalization. Hence, we formalize matching logic using the Coq proof assistant. Specifically, we create a new representation of matching logic that uses a locally nameless encoding, and we formalize the syntax, semantics, and proof system of this representation in the Coq proof assistant. Crucially, we prove the soundness of the formalized proof system and provide a means to carry out interactive matching logic reasoning in Coq. We believe this work provides a previously unexplored avenue for reasoning about matching logic, its models, and the proof system.

1 Introduction

Matching logic [14, 38] is a unifying logical framework for defining the formal semantics of programming languages and specifying their language tools. Given a programming language L, its formal semantics is defined by a *matching logic theory* Γ^L, i.e., a set of axioms. Many language tools, such as parsers, interpreters, compilers, and even deductive program verifiers, are best-effort implementation of matching logic reasoning. The correctness of language tools is justified by matching logic proof objects, checkable by a 240-line proof checker [25].

The formal semantics of many real-world programming languages have been *completely defined* as matching logic theories. These languages include C [23], Java [7], JavaScript [36], Python [21], Rust [26, 41], Ethereum bytecode [24], x86-64 [18], and LLVM [27]. The $\mathbb{K}$ framework (https://kframework.org) is a best-effort implementation of matching logic. From the formal semantics of these real-world languages, $\mathbb{K}$ automatically generates implementations and formal analysis tools, some of which have been commercialized [22]. Ultimately, $\mathbb{K}$ can be used as a tool for formally defining languages, and as a tool for formally reasoning about properties of programming languages and programs.

$\mathbb{K}$ currently provides the most comprehensive support for *automated reasoning* for matching logic by means of various algorithms that are specifically targeted at the (automatic) generation of language tools such as interpreters and deductive verifiers, which, respectively, *are* specific forms of matching logic reasoning, and an integration with the state-of-the-art SMT solvers such as Z3 [19]. However, there is no "exit solution" in $\mathbb{K}$ when these automatic algorithms and external solvers fail, in which case *lemmas*, whose correctness is justified externally, often informally, need to be manually added to fill in the reasoning gaps.

Vlad Rusu (Ed.) Sixth Working Formal Methods Symposium (FROM 2022)
EPTCS 369, 2022, pp. 17–36, doi:10.4204/EPTCS.369.2

© P. Bereczky et al.
This work is licensed under the
Creative Commons Attribution License.

In this paper, we aim to bridge the aforementioned reasoning gaps by bringing *interactive reasoning* to matching logic. Specifically, we give, for the first time, a complete mechanization of matching logic in Coq [6].

Contributions

In this work, we investigate the formal definition of matching logic in an interactive theorem prover, with the aim of enabling computer-aided reasoning *about* and *within* matching logic:

- *Mechanize the soundness of the matching logic proof system*—while matching logic has been proven sound on paper, its soundness has not been formalized yet. This soundness proof is crucial in providing the highest level of assurance for mechanized reasoning.
- *Enable interactive, mechanically verified reasoning about matching logic theories*—the formalization needs to leverage the full power of the theorem prover to encode matching logic theories in a modular way and do reasoning at the highest abstraction level possible.

Following upon similar formalization efforts (such as the solutions to the POPLMark Challenge [5]), we have decided to encode the logic in a widely used and mature system, the Coq proof assistant [6]. Furthermore, we carry out the embedding in a locally nameless [10, 20, 31, 33] representation, which, in contrast to named approaches, is more amenable to computer-aided verification. The entire formalization is open, and available online [2]. The paper shows the following main contributions:

- A locally nameless definition of the matching logic syntax, semantics, and proof system;
- The design of the embedding of the locally nameless matching logic in the Coq proof assistant;
- The first *mechanized* proof of the soundness of the matching logic proof system;
- Example theories with proofs about their semantic and proof-theoretical properties in the mechanized matching logic;
- A preliminary implementation of a matching logic proof mode to simplify interactive reasoning.

The paper is structured as follows. Section 2 introduces matching logic, in a locally nameless representation. In Section 3, we outline the Coq formalization, including some technical challenges faced. Section 4 discusses examples of meta-level reasoning and the soundness proof in particular. Section 5 presents an example interactive proof in our formalization. Finally, Section 6 discusses related work, and Section 7 presents areas for future work and concludes.

2 Introduction to Matching Logic

In this section, we present the syntax, semantics, and proof system of matching logic. Unlike in previous work [12, 14, 16], here we introduce a *locally nameless* presentation of matching logic. This new presentation is more convenient to be formalized in proof assistants, which is discussed in detail in Section 3.

2.1 Matching Logic Locally Nameless Syntax

The syntax we present in this section is in literature known as that of a *locally nameless* one [10, 20, 31, 33]. A locally nameless syntax is a combination of the traditional named representation and the entirely nameless de Bruijn encoding, using named variables if they occur free and de Bruijn encoding if they are

bound. In particular, it distinguishes free and bound variables on the syntactic level, enabling capture-avoiding substitutions without variable renaming. This design decision eliminates the need for reasoning about α-equivalence.

Firstly, we define matching logic *signatures*. A signature provides us infinitely many named variables and a set of (user-defined) constant symbols.

Definition 1 (Signatures)**.** *A* matching logic signature *is a tuple* $(\mathsf{EVar}, \mathsf{SVar}, \Sigma)$ *where*

- EVar *is a set of* element variables*, denoted* $x, y, \ldots$*;*
- SVar *is a set of* set variables*, denoted* $X, Y, \ldots$*;*
- Σ *is a set of* (constant) symbols*, denoted* $\sigma, f, g, \ldots$

Both EVar *and* SVar *are countably infinite sets.* Σ *is a countable set, possibly empty. When the sets of variables are understood from the context, we feel free to use* Σ *to denote a matching logic signature.*

Given a signature Σ, the syntax of matching logic generates a set of well-formed formulas, called *patterns*. In the following, we first define pseudo-patterns.

Definition 2 (Locally Nameless Representation of Pseudo-Patterns)**.** *Given a signature* Σ*, the set of* pseudo-patterns *is inductively defined by the following grammar:*

$$\varphi ::= x \mid X \mid \underline{n} \mid \underline{\underline{N}} \mid \sigma \mid \varphi_1\, \varphi_2 \mid \bot \mid \varphi_1 \to \varphi_2 \mid \exists\,.\,\varphi \mid \mu\,.\,\varphi$$

In the above grammar, x *and* X *are element and set variables, respectively;* $\underline{n}$ *and* $\underline{\underline{N}}$ *are de Bruijn indices that represent the bound element and set variables, respectively, where* $n, N \in \mathbb{N}$*;* σ *is any symbol in the given signature* Σ*;* $\varphi_1\, \varphi_2$ *is called* application*, where* φ_1 *is applied to* φ_2*;* $\bot$ *and* $\varphi_1 \to \varphi_2$ *are propositional operations;* $\exists\,.\,\varphi$ *is the FOL (first-order logic)-style quantification; and* $\mu\,.\,\varphi$ *is the least fixpoint pattern. Both* $\exists$ *and* μ *use the nameless de Bruijn encoding for their bound variables.*

Definition 3 (Locally Nameless Representation of Patterns)**.** *We say that a pseudo-pattern* ψ *is* well-formed*, if (1) for any subpattern* $\mu\,.\,\varphi$*, the nameless variable bound by* μ *has no negative occurrences in* φ*; and (2) all of its nameless variables (i.e., de Bruijn indices) are correctly bound by the quantifiers* $\exists$ *and* μ*, that is, for any de Bruijn indices* n *and* N *that occur in* ψ*,*

- $\underline{n}$ *is in the scope of at least* $n+1$ $\exists$*-quantifiers;*
- $\underline{\underline{N}}$ *is in the scope of at least* $N+1$ μ*-quantifiers.*

A well-formed pseudo-pattern is called a pattern*. For example, the definition of transitive closure in Section 2.4 is a pattern, but* $\exists\,.\,\underline{0} \to \underline{1}$ *is not, since* $\underline{1}$ *is not bound by any quantifier.*

Opening quantified patterns. In a locally nameless representation, we use both named and unnamed variables. Named variables are for free variables, while unnamed variables are for bound variables. Therefore, when we have a pattern $\exists\,.\,\varphi$ (similarly for $\mu\,.\,\varphi$) and want to extract its body, we need to assign a fresh named variable to the unnamed variable that corresponds to the topmost quantifier $\exists$ (resp. μ). This operation is called *opening* a (quantified) pattern [10]. We use $\mathsf{open}_{\mathsf{ele}}(\varphi, x)$ and $\mathsf{open}_{\mathsf{set}}(\varphi, X)$ to denote the opening of the bodies of the quantified patterns $\exists\,.\,\varphi$ and $\mu\,.\,\varphi$, respectively, where x and X are the corresponding new named variables.

These opening operations are special instances of the general case of substitution, which is defined in the usual way[1].

[1] We denote substitution with $\phi[\psi/x]$, where x (a *bound* or *free*, *set* or *element* variable) is replaced by ψ in ϕ.

2.2 Matching Logic Models and Semantics

In this section, we formally define the models and semantics of matching logic. Intuitively speaking, matching logic has a *pattern matching semantics*. A matching logic pattern is interpreted as the *set* of elements that *match* it.

Firstly, we define the notion of matching logic models.

Definition 4 (Matching Logic Models)**.** *Let Σ be a signature. A Σ-model, or simply a* model*, is a tuple $(M, @_M, \{\sigma_M\}_{\sigma\in\Sigma})$ where:*

- *M is a nonempty carrier set;*
- *$@_M \colon M \times M \to \mathscr{P}(M)$ is a binary application function, where $\mathscr{P}(M)$ denotes the powerset of M;*
- *$\sigma_M \subseteq M$ is the interpretation of σ, for each $\sigma \in \Sigma$.*

We use the same letter M to denote the model defined as above.

Next, we define valuations of variables. Note that matching logic has both element and set variables. As expected, a valuation assigns element variables to elements and set variables to subsets of the underlying carrier set. Formally,

Definition 5 (Variable Valuations)**.** *Given a signature Σ and a Σ-model M, a variable valuation ρ is a mapping such that:*

- *$\rho(x) \in M$ for all $x \in \mathsf{EVar}$;*
- *$\rho(X) \subseteq M$ for all $X \in \mathsf{SVar}$.*

We are now ready to define the semantics of matching logic patterns.

Definition 6 (Matching Logic Semantics)**.** *Given a matching logic model M and a variable valuation ρ, we define the semantics of any (well-formed) pattern ψ, written $|\psi|_{M,\rho}$, inductively as follows:*

$$|x|_{M,\rho} = \{\rho(x)\}; \qquad |X|_{M,\rho} = \rho(X);$$

$$|\sigma|_{M,\rho} = \sigma_M; \qquad |\bot|_{M,\rho} = \emptyset;$$

$$|\varphi_1 \to \varphi_2|_{M,\rho} = M \setminus (|\varphi_1|_{M,\rho} \setminus |\varphi_2|_{M,\rho}); \qquad |\varphi_1\, \varphi_2|_{M,\rho} = \bigcup_{a_1 \in |\varphi_1|_{M,\rho}} \bigcup_{a_2 \in |\varphi_2|_{M,\rho}} a_1 \,@_M\, a_2;$$

$$|\exists\,.\,\varphi|_{M,\rho} = \bigcup_{a\in M} |\mathsf{open}_{\mathsf{ele}}(\varphi, x)|_{M,\rho[a/x]} \text{ for fresh } x \in \mathsf{EVar};$$

$$|\mu\,.\,\varphi|_{M,\rho} = \mathbf{lfp}\ \mathscr{F}^{\rho}_{\varphi,X}, \text{ where } \mathscr{F}^{\rho}_{\varphi,X}(A) = |\mathsf{open}_{\mathsf{set}}(\varphi, X)|_{M,\rho[A/X]} \text{ for fresh } X \in \mathsf{SVar}.$$

The above definition is well-defined ($\mathsf{open}_{\mathsf{ele}}$ and $\mathsf{open}_{\mathsf{set}}$ are defined at the end of Section 2.1). For examples, we refer to the formalization [2] and to [13, Section 4].

In the following, we define validity and the semantic entailment relation in matching logic.

Definition 7. *For M and φ, we write $M \models \varphi$ iff $|\varphi|_{M,\rho} = M$ for all valuations ρ. For a pattern set Γ, called a* theory*, we write $M \models \Gamma$ iff $M \models \varphi$ for all $\varphi \in \Gamma$. We write $\Gamma \models \varphi$ iff for any M, $M \models \Gamma$ implies $M \models \varphi$.*

2.3 Matching Logic Proof System

In this section, we present the proof system of matching logic. Matching logic has a Hilbert-style proof system with 19 simple proof rules, making it small and easy to implement. The proof system defines the *provability relation* written as $\Gamma \vdash \varphi$, where Γ is a set of patterns (often called a *theory* and the patterns are called *axioms*) and φ is a pattern that is said to be *provable* from the axioms in Γ.

We present the proof system of matching logic in Table 1. To understand it, we first need to define the notion of *contexts* and a particular type of contexts called *application contexts*.

Table 1: Matching Logic Proof System under Locally Nameless Representation
(C_1, C_2 are application contexts, $FV(\varphi)$ denotes the set of free variables in φ)

Proof Rule Names	***Proof Rules***	***Proof Rule Names***	***Proof Rules***
(Proposition 1)	$\varphi_1 \to (\varphi_2 \to \varphi_1)$	(Proposition 2)	$(\varphi_1 \to (\varphi_2 \to \varphi_3)) \to (\varphi_1 \to \varphi_2) \to (\varphi_1 \to \varphi_3)$
(Proposition 3)	$((\varphi \to \bot) \to \bot) \to \varphi$	(Modus Ponens)	$\dfrac{\varphi_1 \quad \varphi_1 \to \varphi_2}{\varphi_2}$
($\exists$-Quantifier)	$\mathsf{open}_{\mathsf{ele}}(\varphi, x) \to \exists.\,\varphi$ with $x \in \mathsf{EVar}$		
($\exists$-Generalization)	$\dfrac{\mathsf{open}_{\mathsf{ele}}(\varphi_1, x) \to \varphi_2}{(\exists.\,\varphi_1) \to \varphi_2}$ with $x \notin FV(\varphi_2)$		
(Propagation Left$_\bot$)	$\bot\,\varphi \to \bot$	(Propagation Right$_\bot$)	$\varphi\,\bot \to \bot$
(Propagation Left$_\vee$)	$(\varphi_1 \vee \varphi_2)\,\varphi_3 \to (\varphi_1\,\varphi_3) \vee (\varphi_2\,\varphi_3)$		
(Propagation Right$_\vee$)	$\varphi_1\,(\varphi_2 \vee \varphi_3) \to (\varphi_1\,\varphi_2) \vee (\varphi_1\,\varphi_3)$		
(Propagation Left$_\exists$)	$(\exists.\,\varphi_1)\,\varphi_2 \to \exists.\,\varphi_1\,\varphi_2$	(Propagation Right$_\exists$)	$\varphi_1\,(\exists.\,\varphi_2) \to \exists.\,\varphi_1\,\varphi_2$
(Framing Left)	$\dfrac{\varphi_1 \to \varphi_2}{\varphi_1\,\varphi_3 \to \varphi_2\,\varphi_3}$	(Framing Right)	$\dfrac{\varphi_2 \to \varphi_3}{\varphi_1\,\varphi_2 \to \varphi_1\,\varphi_3}$
(Substitution)	$\dfrac{\varphi}{\varphi[\psi/X]}$	(Pre-Fixpoint)	$\varphi[(\mu.\,\varphi)/\underline{0}] \to \mu.\,\varphi$
(Knaster-Tarski)	$\dfrac{\varphi_1[\varphi_2/\underline{0}] \to \varphi_2}{(\mu.\,\varphi_1) \to \varphi_2}$		
(Existence)	$\exists.\,\underline{0}$	(Singleton)	$\neg(C_1[x \wedge \varphi] \wedge C_2[x \wedge \neg\varphi])$

Definition 8. *A* context *C is simply a pattern with one unique placeholder denoted $\Box$. We write $C[\varphi]$ to mean the result of plugging φ in $\Box$ in the context C. We call C an* application context *if from the root of C to $\Box$ there are only applications; that is, C is (inductively) constructed as follows:*

- *C is $\Box$ itself, called the identity context; or*
- *$C \equiv C_1\,\varphi$, where C_1 is an application context; or*
- *$C \equiv \varphi\,C_2$, where C_2 is an application context.*

The proof rules in Table 1 can be divided into four categories:

- **FOL reasoning** containing the standard proof rules as in FOL;
- **Frame reasoning** consisting of six propagation rules and two framing rules, which allow us to propagate formal reasoning that is carried out within an application context throughout the context. Note that we proved these rules equivalent to the ones in [14], where application contexts are not splitted into applications to the left and right;
- **Fixpoint reasoning** containing the fixpoint rules as in modal μ-calculus;
- **Miscellaneous rules**.

Next, we state the soundness theorem of the proof system, which has been proved by induction over the structure of the proof $\Gamma \vdash \varphi$ in [14]. We elaborate on the mechanization of this proof in Section 4.

Theorem 1 (Soundness Theorem)**.** *$\Gamma \vdash \varphi$ implies $\Gamma \models \varphi$.*

2.4 Example Matching Logic Theories

First-order logic. It is easy to see that matching logic is at least as expressive as classical first-order logic, and Chen et al. [13] show that it is indeed more expressive than FOL. At the same time, they describe a direct and natural connection between FOL and matching logic.

A FOL term t is interpreted as an element in the underlying carrier set. From the matching logic point of view, t is a pattern that is matched by *exactly one element*. We use the terminology *functional patterns* to refer to patterns whose valuations are singleton sets. Intuitively, FOL terms *are* functional patterns in matching logic. FOL formulas are interpreted as two logical values: true and false. In matching logic, there is a simple analogy where we use the empty set $\emptyset$ to represent the logical false and the total carrier set to represent the logical true. A pattern whose interpretation is always $\emptyset$ or the full set is called a *predicate pattern*. Intuitively, FOL formulas *are* predicate patterns in matching logic.

Equality. Although matching logic has no built-in notion of equality, it can be easily defined using a construct called *definedness*. Formally, we define $\Sigma^{\mathsf{DEF}} = \{\lceil_\rceil\}$, $\Gamma^{\mathsf{DEF}} = \{\lceil x\rceil\}$ ($\lceil x\rceil$ denotes $\lceil_\rceil\, x$); this axiom ensures that whenever a pattern φ is matched by some model element, the pattern $\lceil\varphi\rceil$ is matched by all elements of the model, and vice versa. Equality is then defined as a notation $\varphi_1 = \varphi_2 \equiv \neg\lceil\neg(\varphi_1 \leftrightarrow \varphi_2)\rceil$, intuitively saying that there is no element that would match only one of the two formulas. It is easy to see that for any pattern φ, the pattern $\lceil\varphi\rceil$ is a predicate pattern, and that equality of two patterns is a predicate. With this, we can similarly define other notations, such as membership, subset, totality, etc. seen below. To make things easier, we use the notion of matching logic *specification* introduced in [13]. The signature Σ^{DEF} and theory Γ^{DEF} are then defined by Spec. 1. For more details on definedness, we refer to Section 4.2.

spec DEF
  Symbol $\lceil_\rceil$
  Notation

$$\lceil\varphi\rceil \equiv \lceil_\rceil\,\varphi \qquad \lfloor\varphi\rfloor \equiv \neg\lceil\neg\varphi\rceil$$
$$\varphi_1 = \varphi_2 \equiv \lfloor\varphi_1 \leftrightarrow \varphi_2\rfloor \qquad \varphi_1 \neq \varphi_2 \equiv \neg(\varphi_1 = \varphi_2)$$
$$x \in \varphi \equiv \lceil x \wedge \varphi\rceil \qquad \varphi_1 \subseteq \varphi_2 \equiv \lfloor\varphi_1 \to \varphi_2\rfloor$$
$$x \notin \varphi \equiv \neg(x \in \varphi) \qquad \varphi_1 \not\subseteq \varphi_2 \equiv \neg(\varphi_1 \subseteq \varphi_2)$$

  Axiom
    (DEFINEDNESS) $\lceil x\rceil$
endspec

Spec. 1: Definedness and related notions

Induction and transitive closure. Chen et al. show how matching logic, by using the application and least fixpoint operators, can not only axiomatize equality, but also product types ($\langle_,_\rangle$), and inductive types [13]. Hence, another notable example of matching logic's expressiveness is that it can specify the transitive closure of a binary relation R the following way (where $\in$ is defined by Spec. 1):

$$\mu X\,.\,R \vee \exists x\,.\,\exists y\,.\,\exists z\,.\,\langle x,z\rangle \wedge \langle x,y\rangle \in X \wedge \langle y,z\rangle \in X$$

Or rather, the same matching logic pattern expressed in the locally nameless representation:

$$\mu\,.\,R \vee \exists\,.\,\exists\,.\,\exists\,.\,\langle \underline{2},\underline{0}\rangle \wedge \langle \underline{2},\underline{1}\rangle \in \underline{\underline{0}} \wedge \langle \underline{1},\underline{0}\rangle \in \underline{\underline{0}}$$

3 Matching Logic Formalization in Coq

In this section, we describe how the locally nameless matching logic (as defined in Section 2) has been encoded in Coq. The formalization is distributed as a library including the definition of the logic as well as that of some standard theories, with the two dependencies being the Std++ library [32] and the Equations plugin [39]. Some of our proofs rely on classical and extensionality axioms (namely, functional extensionality, propositional extensionality, and the axiom of excluded middle), but these are known to be compatible with Coq's logic [9].

We implement matching logic in a *deep embedding* style; that is, formulas, models, and proofs are represented as data in Coq. Presumably, a shallow embedding could provide a more lightweight implementation, but deep embedding has couple of advantages for our use cases; mainly, it allows us to inspect matching logic proofs without reflection, reason about them directly (for instance, when checking the side conditions of the *deduction theorem* from [15]) and to extract them from Coq (see Section 7) [2] .

3.1 Syntax

We represent a matching logic signature $(\mathsf{EVar}, \mathsf{SVar}, \Sigma)$, defined in Definition 1, as an instance of the `Class Signature` that encapsulates the sets of variables and the set of symbols. The sets for variables are required to be countable and infinite, and in addition, it is also required that equality on variables and symbols is decidable. We also provide an off-the-shelf, string-based instance of the type class as a default option. Although this instance will suffice for most applications, to prove the completeness of the matching logic proof system, a more general carrier set will be needed[3]. The way we represent free variable names has some features in common with the concept of *atoms* used in *nominal* approaches [4] (e.g., any countably infinite set with decidable equality can be used for names), but in the locally nameless approach we do not rely on permutative renaming when implementing capture-avoidance.

We formalize (pseudo-)patterns (Definition 2) as the inductive definition `Inductive Pattern : Set`. Due to using a locally nameless embedding, we have separate constructors for free variables (named variables) and bound variables (de Bruijn indices). The main advantage this provides is in the equivalence of quantified patterns. Specifically, in a fully named representation, the patterns $\exists x.x$ and $\exists y.y$ are equivalent, yet not syntactically equal. To formally prove their equivalence, we would need a notion of α-equivalence of patterns that is constantly applied. However, in our locally nameless representation, both these can only be represented by the pattern $\exists.\underline{0}$, and thus are syntactically equal, so no additional notions of equality need to be supplied. This is implicitly used throughout our soundness proof and in proving properties of specifications.

Well-formedness. For practical reasons, the type `Pattern` corresponds to the definition of pseudo-patterns, while the restrictions on non-negativity and scoping defined for patterns (Definition 3) are implemented as a pair of `bool`-valued auxiliary functions:

```
well_formed_positive : Pattern -> bool
well_formed_closed   : Pattern -> bool
```

where the first function performs a check for the positivity constraint and the second one checks the scoping requirements. The predicate `well_formed_closed` is constructed from two parts:

```
well_formed_closed_ex_aux : Pattern -> nat -> bool
well_formed_closed_mu_aux : Pattern -> nat -> bool
```

[2] It is beyond the scope of this work to tell if a shallow embedding would be more suitable for object-level reasoning.

[3] We refer to the proof of the completeness of matching logic without μ and the extension lemma [15].

These predicates return `true`, when the parameter pattern contains only smaller unbound de Bruijn indices for element and set variables than the given number. A pattern is `well_formed` if it satisfies both `well_formed_closed` and `well_formed_positive`. Most of our functions operate on the type `Pattern` without the well-formedness constraint; we use the constraint mainly for theorems. This way we separate proofs from data.

Substitution and opening. In the locally nameless representation, there are separate substitution functions for bound and free variables (both for element and set variables). In our formalization, we followed the footsteps of Leroy [31], so the bound variable substitution decrements the indices of those (bound) variables that are greater than the substituted index. We define

```
Definition evar_open (k : db_index) (x : evar) (p : Pattern) : Pattern.
Definition svar_open (k : db_index) (X : svar) (p : Pattern) : Pattern.
```

which correspond to $\mathsf{open}_{\mathsf{ele}}(\varphi, x)$ and $\mathsf{open}_{\mathsf{set}}(\varphi, X)$ from Section 2.1, with a difference that this version of opening allows for substitution of any de Bruijn index, not only the one corresponding to the topmost quantifier ($\underline{0}$ or $\underline{\underline{0}}$).

Derived notations. Matching logic is intentionally minimal. As a consequence, any non-trivial theory is likely to heavily rely on notations that abbreviate common operations. Besides basic notations for boolean operations, universal quantification and greatest fixpoint, one also can define equality, subset and membership relations on top of the definedness symbol (as we did in Spec. 1), which is not part of matching logic, but is usually considered as a part of the "standard library" for the core logic.

Coq provides (at least) two ways to extend the syntax of the core logic with derived notations. The first option is to use Coq's `Notation` mechanism. For example, the following would define the notations for negation, disjunction, and conjunction:

```
Notation "! p" := (p ---> ⊥).
Notation "p or q" := (! p ---> q).
Notation "p and q" := ! (! p or ! q).
```

The problem with this is that the pattern

```
(x ---> ⊥) ---> ⊥
```

could be interpreted either as `! (! x)`, or as `x or ⊥`, which would be confusing to the user, having no control on which interpretation Coq chooses to display.

Therefore, we decided to opt for the other option, representing each derived notation as a Coq `Definition`, as in the following snippet:

```
Definition patt_not p := p ---> ⊥.
Definition patt_or p q := patt_not p ---> q.
Definition patt_and p q := patt_not (patt_or (patt_not p) (patt_not q))
```

We can define the notations on top of these definitions. This way, the user can fold/unfold derived notations as needed. However, this representation of notations poses another problem: many functions, especially the substitutions such as `bevar_subst`, preserve the structure of the given formula, but since they build a `Pattern` from the low-level primitives, the information about derived notations is lost whenever such function is called. We solve this by defining for each kind of syntactical construct (e.g., for unary operation, binary operation, element variable binder) a type class containing a rewriting lemma for `bevar_subst` such as this one:

```
Class Binary (binary : Pattern -> Pattern -> Pattern) :=
{
  binary_bevar_subst :
    forall ψ, well_formed_closed ψ ->
      forall n ϕ1 ϕ2,
        bevar_subst (binary ϕ1 ϕ2) ψ n
        = binary (bevar_subst ϕ1 ψ n) (bevar_subst ϕ2 ψ n) ;
(* ... *)
}.
```

The user of our library then can instantiate the class for their derived operations and use the tactic `simpl_bevar_subst` to simplify the expressions containing `bevar_subst` and `evar_open`.

Fresh variables. We say that a variable is fresh in a pattern φ if it does not occur among the free variables of φ. Sometimes (for example, in the semantics of existential and fixpoint patterns) it is necessary to find a variable that does not occur in the given pattern. We required the variable types (`evar` and `svar`) to be infinite, thus we can use the solution of the Coq Std++ library [32] for fresh variable generation. We then have a function `fresh_evar : Pattern -> evar` and the following lemma:

```
Lemma set_evar_fresh_is_fresh φ : fresh_evar φ ∉ free_evars φ.
```

3.2 Semantics

On the semantics side we have a `Record Model`. We do not require the carrier set of the model to have decidable equality. We represent the variable valuation function defined in Definition 5 as a record of two separate functions, one mapping element variables to domain elements and another mapping set variables to sets of domain elements. With this, we define the interpretation of patterns ($|_|_{M,\rho}$ in Definition 6) as expected, with two points worth noting:

1. The interpretation of patterns cannot be defined using structural recursion on the formula, because in the μ (and $\exists$) case, one calls `eval` on `svar_open 0 X p'` (and `evar_open 0 x p'`), which is not a structural subformula of `p`. Therefore, we do recursion over the size of the formula, implemented as an `Equation`:

   ```
   Equations eval (ρ : @Valuation M) (φ : Pattern) : propset (@Domain M) by wf (size φ) :=
       (* ... *)
       eval ρ (patt_mu φ') := let X := fresh_svar φ' in
         @LeastFixpointOf _ OS L (fun S =>
           let ρ' := (update_svar_val X S ρ) in
             eval ρ' (svar_open 0 X φ')).
   ```

2. We decided to give semantics to patterns that are not well-formed, including arbitrary μ patterns. This way, we do not have to supply the `eval` function with the well-formedness constraint, which makes it easier to use. We did that by (1) defining the function `LeastFixpointOf` to return the intersection of all prefixpoints; (2) mechanizing the relevant part of the Knaster-Tarski fixpoint theorem [40], and (3) proving that the function associated to a well-formed μ pattern is monotone.

3.3 Proof System

We formalize the proof system of matching logic as an inductive definition:

```
Inductive ML_proof_system theory : Pattern -> Set := (* ... *) .
```

The proof system is defined as expected; however, one may ask why the proof system lives in `Set` and not in `Prop`. The answer is that in our *deep embedding* we care about the internal structure of matching logic proofs, not only about provability. We do not want two matching logic proofs to be considered identical only because they prove the same formula, for at least two reasons. First, some theorems (e.g. the deduction theorem from [15]) can only be applied to a proof if that proof satisfies particular conditions regarding its internal structure. Second, when extracting OCaml or Haskell programs from this Coq formalization, we need the manipulation of the proof system terms to be preserved; that will allow us to extract Metamath proof objects in the future (see Section 7).

Another point worth mentioning is that the rules in the formalization of the proof system often require some well-formedness constraints. A consequence of this is that only well-formed patterns are provable:

```
Lemma proved_impl_wf Γ ϕ: Γ ⊢ ϕ -> well_formed ϕ.
```

We discuss the soundness of the proof system in Section 4. Crucially, we rely on the μ patterns to be interpreted as least fixpoints, as explained in Section 3.2.

4 Reasoning about Matching Logic

After encoding matching logic in Coq, we overview some results it allows for concerning meta-level reasoning. In particular, we highlight some challenges we faced when proving the mechanized matching logic proof system sound, and we demonstrate semantics-based reasoning about the theory of equality.

4.1 Soundness of The Proof System

The most crucial result of the mechanization of matching logic is the proof of the soundness of its proof system (Table 1). Even though this theorem has already been investigated in related publications, we have developed the first complete, machine-checked proof, which verifies the prior paper-based results.

We state our soundness theorem below:

```
Theorem Soundness : forall phi : Pattern, forall theory : Theory,
  well_formed phi -> theory ⊢ phi -> theory ⊨ phi.
```

The proof of soundness begins via induction on the hypothesis `theory ⊢ phi`, meaning we consider all cases from the proof system which may have produced this hypothesis. Many cases, such as the propositional proof rules, were straightforwardly discharged using set reasoning. Other cases, like Modus Ponens, were discharged via the application of the induction hypothesis.

The most involved cases were the proof rules involving quantification, specifically the $\exists$-Quantifier, Prefixpoint, and Knaster-Tarski rules. For these rules, the key steps were to establish complex substitution lemmas (separate lemmas for existential quantification and for μ-quantification). The proofs of these lemmas were very involved, and we note that the set substitution lemma was not proved in related work previously. For existential quantification, we adapt the "element substitution lemma" which appears in [15, Lemma 41].

For the soundness of the Pre-fixpoint and Knaster-Tarski rules with μ-quantification, we introduce a new similar lemma called *set substitution lemma*, which links syntactic substitution with semantic substitution, stating that the following two ways of plugging a pattern φ_2 into a pattern φ_1 are equivalent:

1. syntactically substitute φ_2 for a free set variable X in φ_1 and interpret the resulting pattern;
2. interpret φ_2 separately, then interpret φ_1 in a valuation where X is mapped to the value of φ_2.

4.2 Theory of Equality

The formalization also allows for reasoning about matching logic models. We implemented the theory of definedness and equality as presented in [13, 38], and Section 2.4. Then, we established some results about models that satisfy the definedness axiom, which provides support for common cases of semantic reasoning. This branch of the development showcases applications of our mechanization for reasoning about matching logic models.

Definedness and totality. Definedness has the important property that, applied to any formula φ which matches at least one model element, the result matches all elements of the model (represented by $\top$):

```
Lemma definedness_not_empty_iff : forall (M : @Model Σ),
    M ⊨^T theory ->
    forall (ϕ : Pattern) (ρe : @EVarVal Σ M) (ρs : @SVarVal Σ M),
      (@eval Σ M ρe ρs ϕ) <> ∅ <-> (@eval Σ M ρe ρs ⌈ ϕ ⌉ ) = ⊤.
```

This is why it is called *definedness*: $\lceil\phi\rceil$ evaluates to full set if and only if ϕ is *defined*; that is, matched by at least one element. Note that one needs the definedness axiom only for the "if" part; the "only if" part is guaranteed by the definition of the extension of application: anything applied to the empty set results in the empty set. The dual of definedness is called *totality*: a pattern ϕ is considered *total* iff it is matched by all elements of the model, and totality of a pattern ($\lfloor\phi\rfloor$) *holds* (is matched by all elements of the model) only in that case:

```
Lemma totality_not_full_iff : forall (M : @Model Σ),
    M ⊨^T theory ->
    forall (ϕ : Pattern) (ρe : @EVarVal Σ M) (ρs : @SVarVal Σ M),
      @eval Σ M ρe ρs ϕ <> ⊤ <-> @eval Σ M ρe ρs ⌊ ϕ ⌋ = ∅.
```

Equality. As we have seen in Section 2, equality is defined using totality. We proved that equality defined this way indeed has the intended property, i.e., equality of two patterns holds iff the two patterns are interpreted to equal sets.

```
Lemma equal_iff_interpr_same : forall (M : @Model Σ),
    M ⊨^T theory ->
    forall (ϕ1 ϕ2 : Pattern) (ρe : @EVarVal Σ M) (ρs : @SVarVal Σ M),
      @eval Σ M ρe ρs (ϕ1 =ml ϕ2) = ⊤ <->
      @eval Σ M ρe ρs ϕ1 = @eval Σ M ρe ρs ϕ2.
```

Our formalization also demonstrates that in matching logic, one cannot simply use equivalence ($\leftrightarrow$) instead of equality. As argued in [38], one could expect that the pattern $\exists.\, f\ x \leftrightarrow \underline{0}$ specifies that f behaves like a function; however, there exists a model in which that is not the case. In the model whose domain is `exampleDomain` and whose interpretation of application is defined as `example_app_interp` below:

```
Inductive exampleDomain : Set := one | two | f.
Definition example_app_interp (d1 d2 : exampleDomain) : Power exampleDomain :=
  match d1, d2 with
  | f, one => ⊤
  | f, two => ∅
  | _, _ => ∅
  end.
```

the pattern $\exists.\, f\ x \leftrightarrow \underline{0}$ holds (in every interpretation of x), even though the model does not implement f as a function. For more technical details, we refer to the formalization [2].

5 Reasoning in Matching Logic

In Section 3, we encoded matching logic and its proof system in Coq. With the minimal proof system, one can already reason about syntactic consequence by using Coq's `apply` tactic. However, only using the presented rules to reason about derived operations and complex theories is not really productive: it is easy to get lost when facing a complex proof obligation expressed in vanilla matching logic after unfolding the derived notations.

Derived rules. To support object-level reasoning, we proved several derived axioms and rules, essentially enriching the proof system with rules about derived constructs (commonly used operations that are not included in the syntax of matching logic) and common theories (such as definedness). These alone can shorten a typical matching logic proof script significantly. For example, think of destructing a disjunctive premise into two premises, which is naturally one step in an informal proof, but takes a couple of steps with the matching logic proof system. However, writing proofs with the derived rules is still cumbersome, because now we get lost in the details of combining and applying these theorems with the correct parameters. We tackle this problem with a dedicated Coq proof mode.

5.1 Matching Logic Proof Mode

To further simplify reasoning in the embedded logic, we conceptually separate the matching logic proof state from the Coq proof state, introduce a local proof context, and define a set of special Coq tactics that manipulate the dedicated proof state. We call these concepts together *the matching logic proof mode*[4]. The ultimate goal with the proof mode is to make matching logic proofs simple to read and write, especially for users familiar with Coq. The contents of this section are work-in-progress, but nicely demonstrate the potential that lies in carrying out interactive matching logic proofs in Coq.

Matching logic proof state. The concept of the proof state allows us to nicely mimic Coq-style reasoning in matching logic by rendering matching logic proof goals as a list of named hypotheses and a goal pattern. Behind the scenes, the goal on provability is turned into a record that stores the list of the premises along with the conclusion. The proof mode allows for moving left-hand sides of implication conclusions to the list of premises (the local context), which is essential in matching logic as the deduction theorem [15] can be applied to totality patterns only. The proof mode provides a better overview on the state of the proof in the interactive mode. In particular, it contains the following sections:

- A meta-level context (such as hypotheses on the well-formedness of patterns)
- A global matching logic context (a set of patterns known to be valid);
- A local matching logic context (a named list of patterns assumed to be valid);
- A matching logic goal (a single pattern, the conclusion).

We provide a notation for this proof state, which also resembles the proof state in Coq (see Figure 1a). In this example, $\phi_1, \ldots, \phi_n$ form the global matching logic context, while $\psi_1, \ldots, \psi_n$ form the local one, and χ is the goal. A matching logic proof state can automatically be converted to a syntactic provability statement as presented in Figure 1b which describes this conversion of the proof state in Figure 1a.

[4]We borrow the term *proof mode* and the approach from the authors of the Iris proof mode [30], and the Coq reference manual [1].

```
ϕ1 , ... , ϕn , ψ1 , ... , ψm , χ : Pattern
______________________________________(1/1)
{[ϕ1,...,ϕn]} ⊢
"H1" : ψ1
...
"Hn" : ψm
--------------------------------------
χ
```

(a) Notation

```
ϕ1 , ... , ϕn , ψ1 , ... , ψm , χ : Pattern
______________________________________(1/1)
{[ϕ1,...,ϕn]} ⊢ ψ1 ---> ... ---> ψm ---> χ
```

(b) Conversion

Figure 1: Matching logic proof state

The mapping between matching logic proof states and matching logic proofs of syntactic consequence is not injective: there can be multiple proof states that represent the same matching logic proof obligation when the the conclusion is an implication pattern.

Matching logic proof tactics. To create proof tactics, we first lift the derived proof rules to work with matching logic proof states. By lifting, we actually mean proving the derived rules for the new proof state. The created tactics can be divided into three main groups:

- Tactics that restructure the local context (e.g., `mlIntro`, `mlRevertLast`, `mlClear`).
- Tactics that apply lifted derived rules to the proof state (e.g., `mlApply`, `mlApplyMeta`, `mlDestructOr`).
- Miscellaneous tactics (e.g., `mlRewrite` which replaces parts of the matching logic goal, `mlTauto` which is a preliminary tautology solver).

We can use these tactics in a similar way as their Coq counterparts (e.g., `mlIntro` mimics the effect of `intro`), and create matching logic syntactic proofs conveniently. For the sake of brevity, we do not go into details about the implementation of the tactics, but in the background, they expand to applications of the proof system rules, therefore they construct valid matching logic proofs.

5.2 An Interactive Proof

In this section, we show an interactive proof outline (Figure 2) with the matching logic proof mode. The complete proof is available in the formalization [2] (moreover, there is also a short tutorial about the currently formalized tactics). We show an example proof state transformation from each tactic category, but first, we present two lemmas that are essential to construct the proof.

The first lemma is about the connection of conjunction and totality.

```
Lemma patt_total_and {Σ : Signature} {syntax : Syntax}:
  forall Γ φ ψ, theory ⊆ Γ -> well_formed φ -> well_formed ψ ->
  Γ ⊢ ⌊ φ and ψ ⌋ <---> ⌊ φ ⌋ and ⌊ ψ ⌋.
```

The second lemma is the congruence lemma, which states that one can replace equivalent subpatterns in *any* context results in equivalent patterns.

```
Lemma prf_equiv_congruence Γ p q C:
  PC_wf C -> Γ ⊢ (p <---> q) -> Γ ⊢ ((C [p]) <---> (C [q])).
```

```
1.  Lemma overlapping_variables_equal {Σ : Signature} {syntax : Syntax} :
2.    forall x y Γ, theory ⊆ Γ ->
3.    Γ ⊢ ⌈ (patt_free_evar y) and (patt_free_evar x) ⌉ --->
4.            patt_free_evar y  =ml  patt_free_evar x.
5.  Proof.
6.    intros x y Γ HΓ.
7.    remember (patt_free_evar x) as pX. assert (well_formed pX) by (rewrite HeqpX;auto).
8.    remember (patt_free_evar y) as pY. assert (well_formed pY) by (rewrite HeqpY;auto).
9.    toMLGoal. wf_auto2.
10.   unfold patt_equal, patt_iff.
11.   mlRewrite (@patt_total_and Σ syntax Γ
12.                                 (pY ---> pX)
13.                                 (pX ---> pY) HΓ
14.                                 ltac:(wf_auto2) ltac:(wf_auto2)) at 1.
15.   mlIntro "H0". mlIntro "H1". mlDestructOr "H1" as "H1'" "H1'".
16..   * mlApply "H1'". mlClear "H1'". mlIntro "H2".
17.     (* ... *)
18.   * (* ... *)
19. Defined.
```

Figure 2: Case Study for the Proof Mode

We implemented `mlRewrite` based on the congruence lemma. At line 11 (Figure 2) we can use this tactic with the first lemma, since it states the equivalence of two patterns. With this, we are able to propagate totality to the subpatterns of the conjunction for our concrete patterns.

```
______________________________________(1/1)
Γ ⊢
⌈ pY and pX ⌉ --->
  ⌊ (pY ---> pX) and (pX ---> pY) ⌋
```

```
______________________________________(1/1)
Γ ⊢
⌈ pY and pX ⌉ --->
  ⌊ (pY ---> pX) ⌋ and ⌊ (pX ---> pY) ⌋
```

Next, in line 15, we reshape the structure of the matching logic proof state by using `mlIntro` twice. This tactic moves the left-hand side of the implication in the goal to the local matching logic context (note that conjunction is a syntactic sugar).

```
______________________________________(1/1)
Γ ⊢
⌈ pY and pX ⌉ --->
   ⌊ (pY ---> pX) ⌋ and ⌊ (pX ---> pY) ⌋
```

```
______________________________________(1/1)
Γ ⊢
"H0" : ⌈ pY and pX ⌉,
"H1" : ! ⌊ pY ---> pX ⌋ or ! ⌊ pX ---> pY ⌋,
______________________________________
⊥
```

Finally, we also show the usage of `mlApply` in line 16. The conclusion of `"H1'"` matches the goal, thus we can apply it, and show its premise.

```
______________________________________(1/1)
Γ ⊢
"H0" : ⌈ pY and pX ⌉,
"H1'" : ! ⌊ pY ---> pX ⌋,
______________________________________
⊥
```

```
______________________________________(1/1)
Γ ⊢
"H0" : ⌈ pY and pX ⌉,
"H1'" : ⌊ pY ---> pX ⌋ ---> ⊥,
______________________________________
⌊ pY ---> pX ⌋
```

It can be observed that we used a number of standard Coq tactics, and explicit parameters during the proof (in Figure 2). It is ongoing work to continue refining the proof mode and adding new tactics on demand to formalize as many paper-based matching logic proofs in Coq as possible.

6 Related Work

6.1 Embedding Logical Languages in Coq

Ideally, different sorts of problems are specified in different logical languages which fit the problem domain best. For instance, separation logics excel at describing algorithms manipulating shared and mutable states, temporal logics provide abstractions for specifying systems properties qualified in terms of time, whereas matching logic gives a fairly generic basis for reasoning about programming language semantics and program behavior. It is highly desired to carry out proofs in these domain-specific logical systems interactively and mechanically verified, but these logics tend to significantly diverge from the logics of general-purpose proof assistants such as Coq, leading to an abstraction gap.

To use a proof assistant to formalize reasoning in a specific logic, one needs to encode the logic as a theory in the proof assistant and then carry out reasoning at the meta-level with considerable overhead. Related works have been investigating different ways of embedding with the aim of reducing the overhead and facilitate productive object-level reasoning in various logical languages. To name a few, (focused) linear logic [37, 43], linear temporal logic [17], different dialects of separation logic [3, 29, 34] and differential dynamic logic [8] have been addressed in the past. Note that some of these encodings are full-featured proof modes, which create a properly separated proof environment and tactic language for the object logic. A slightly different idea worth mentioning is encoding one theorem prover's logic in another to make the proofs portable, such as taking HOL proofs to Coq [28, 42].

The existing approaches show significant differences depending upon whether the formalization is aimed at proving the properties of the logic or at advocating reasoning in the logic. One particular consideration is to variable representation and the level of embedding. The majority of the cited formalizations apply a so-called shallow embedding, where they reuse core elements of the meta-logic; for instance, names are realized by using higher-order abstract syntax or the parametrized variant thereof, and exploit the binding and substitution mechanism built-in the proof assistant. With this, name binding, scopes, and substitution come for free, but the formalization is tied to the meta-logic's semantics in this aspect, which may not be suitable in all cases. In fact, one of the main design decisions in our work was to use deep embedding facilitated by notations and locally nameless variable representation.

6.2 Matching Logic Implementations

This paper is not the only attempt that tries to formalize matching logic using a formal system. In [11], the authors propose a matching logic formalization based on Metamath [35], a formal language used to encode abstract mathematical axioms and theorems. The syntax and proof system of matching logic are defined in Metamath in a few hundreds lines of code [25]. Matching logic (meta)theorems can be formally stated in Metamath, and their formal proofs can be encoded in Metamath as machine-checkable proof objects. While the Metamath implementation is simpler with a smaller trust base, the Coq formalization of matching logic is more versatile; in Coq one can express a larger variety of metatheorems such as the deduction theorem. In general, the Metamath formalization focuses only on proofs of explicit matching logic theories, while our formalization, despite a larger trust-base, focuses mostly on models, semantics, and metatheorems of matching logic. This dichotomy allows for potential integration with the Metamath formalization (see 7).

Another matching logic implementation is through the $\mathbb{K}$ framework. The $\mathbb{K}$ framework is a very robust engine for formalizing programming language syntax and semantics. A version of matching logic, called Kore, is used by $\mathbb{K}$ to represent processed formal semantics. This is how full large-scale

languages can be simply represented as matching logic theories. Admittedly, defining matching logic theories on the scale of programming languages directly in Coq is not currently feasible. However, our formalization brings more interactivity to matching logic reasoning, which is currently missing in $\mathbb{K}$.

7 Conclusion and Future Work

In this work, we defined a locally nameless representation of matching logic. We also presented the first formal definition of *any* version of matching logic using an interactive theorem prover, namely Coq. We mechanized the soundness theorem of matching logic, and presented some nontrivial matching logic theories and interactive proofs with a preliminary matching logic proof mode. We believe this paves the way for Coq and interactive theorem provers to be used more frequently with matching logic. We discuss some areas for future work below.

- *Complete Coq Proof Mode for Matching Logic.* Proving formulas using the matching logic Hilbert-style proof system is not always convenient, especially when compared to the way one can prove theorems in Coq. For this reason we are working on the presented proof mode for matching logic in Coq, that allows users to prove matching logic theorems using tactics that manipulate the goal and local context. We took the inspiration mainly from the Iris project, where the authors built a proof mode for a variant of separation logic [30].
- *Create Tactics for Type Class Instantiation.* While using the formalization with actual signatures or new derived notations, the user needs to instantiate certain simple type classes. We plan to create tactics to carry out this work automatically.
- *Exporting Metamath Proof Objects.* An interesting way of combining advantages of both our Coq formalization and the Metamath formalization in [11] would be the ability to convert matching logic proofs in Coq to matching logic proofs in Metamath. One challenge here is posed by the fact that Metamath uses the traditional named representation of matching logic patterns, which is different from the locally nameless representation used in our Coq development.
- *Importing $\mathbb{K}$ Definitions.* As mentioned in Section 6, the $\mathbb{K}$ framework is a matching logic (specifically Kore) implementation with the advantage of being able to naturally define real large-scale programming languages. As future work, we plan to formalize Kore as a matching logic theory inside Coq and write a translator from Kore files to Coq files using this theory, thus giving $\mathbb{K}$ framework a Coq-based backend. This would allow languages defined in $\mathbb{K}$ and properties of those languages proved in $\mathbb{K}$ to be *automatically* translated to Coq definitions and theorems.
- *Completeness.* For the fragment of matching logic without the μ operator, the proof system is complete. We would like to formalize the proof of completeness from [15]; however, we expect the proof to be non-constructive, which implies we would not be able to compute proof terms (and extract Metamath proofs) from proofs of semantic validity.

Acknowledgements. We warmly thank Runtime Verification Inc. for their generous funding support. Supported by the ÚNKP-21-4 New National Excellence Program of the Ministry for Innovation and Technology from the source of the National Research, Development and Innovation Fund.

References

[1] *Coq reference manual*. Available at https://coq.inria.fr/refman/. Accessed on 20th, July 2022.

[2] *Matching logic formalization*. Available at https://github.com/harp-project/AML-Formalization/releases/tag/v1.0.6. Accessed on 12th, September 2022.

[3] Andrew W. Appel & Sandrine Blazy (2007): *Separation logic for small-step Cminor*. In: *International Conference on Theorem Proving in Higher Order Logics*, Springer, pp. 5–21, doi:10.1007/978-3-540-74591-4_3.

[4] Brian Aydemir, Aaron Bohannon & Stephanie Weirich (2007): *Nominal reasoning techniques in Coq*. *Electron. Notes Theor. Comput. Sci.* 174(5), p. 69–77, doi:10.1016/j.entcs.2007.01.028.

[5] Brian E. Aydemir et al. (2005): *Mechanized metatheory for the masses: The PoplMark challenge*. In Joe Hurd & Tom Melham, editors: *Theorem Proving in Higher Order Logics*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 50–65, doi:10.1007/11541868_4.

[6] Yves Bertot & Pierre Casteran (2004): *Interactive theorem proving and program development: Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science An EATCS Series, Springer Berlin Heidelberg, Berlin, Heidelberg, doi:10.1007/978-3-662-07964-5.

[7] Denis Bogdanas & Grigore Roşu (2015): *K-Java: A complete semantics of Java*. In Sriram K. Rajamani & David Walker, editors: *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, ACM, pp. 445–456, doi:10.1145/2676726.2676982.

[8] Brandon Bohrer, Vincent Rahli, Ivana Vukotic, Marcus Völp & André Platzer (2017): *Formally verified differential dynamic logic*. In: *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs*, CPP 2017, Association for Computing Machinery, New York, NY, USA, p. 208–221, doi:10.1145/3018610.3018616.

[9] Mario Carneiro (2019): *The type theory of Lean*. Master's thesis. Available at https://github.com/digama0/lean-type-theory/releases/tag/v1.0. Accessed on 12th, September 2022.

[10] Arthur Charguéraud (2012): *The locally nameless representation*. *J. Autom. Reason.* 49(3), pp. 363–408, doi:10.1007/s10817-011-9225-2.

[11] Xiaohong Chen, Zhengyao Lin, Minh-Thai Trinh & Grigore Roşu (2021): *Towards a trustworthy semantics-based language framework via proof generation*. In Alexandra Silva & K. Rustan M. Leino, editors: *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part II, Lecture Notes in Computer Science* 12760, Springer, pp. 477–499, doi:10.1007/978-3-030-81688-9_23.

[12] Xiaohong Chen, Dorel Lucanu & Grigore Roşu (2020): *Initial algebra semantics in matching logic*. Technical Report. Available at http://hdl.handle.net/2142/107781. Accessed on 12th, September 2022.

[13] Xiaohong Chen, Dorel Lucanu & Grigore Roşu (2021): *Matching logic explained*. *Journal of Logical and Algebraic Methods in Programming*, p. 100638, doi:10.1016/j.jlamp.2021.100638.

[14] Xiaohong Chen & Grigore Roşu (2019): *Matching μ-Logic*. In: *34th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2019, Vancouver, BC, Canada, June 24-27, 2019*, IEEE, pp. 1–13, doi:10.1109/LICS.2019.8785675.

[15] Xiaohong Chen & Grigore Roşu (2019): *Matching mu-logic*. Technical Report, University of Illinois at Urbana-Champaign. Available at http://hdl.handle.net/2142/102281. Accessed on 12th, September 2022.

[16] Xiaohong Chen & Grigore Roşu (2020): *A general approach to define binders using matching logic*. *Proc. ACM Program. Lang.* 4(ICFP), pp. 88:1–88:32, doi:10.1145/3408970.

[17] Solange Coupet-Grimal (2003): *An axiomatization of linear temporal logic in the Calculus of Inductive Constructions*. *Journal of Logic and Computation* 13(6), pp. 801–813, doi:10.1093/logcom/13.6.801.

[18] Sandeep Dasgupta, Daejun Park, Theodoros Kasampalis, Vikram S. Adve & 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 (PLDI'19)*, ACM, pp. 1133–1148, doi:10.1145/3314221.3314601.

[19] Leonardo De Moura & Nikolaj Bjørner: *Z3: An efficient SMT solver*. In: *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08)*, Springer, pp. 337–340, doi:10.1007/978-3-540-78800-3_24.

[20] Andrew D. Gordon (1993): *A mechanisation of name-carrying syntax up to alpha-conversion*. In Jeffrey J. Joyce & Carl-Johan H. Seger, editors: *Higher Order Logic Theorem Proving and its Applications, 6th International Workshop, HUG '93, Vancouver, BC, Canada, August 11-13, 1993, Proceedings*, *Lecture Notes in Computer Science* 780, Springer, pp. 413–425, doi:10.1007/3-540-57826-9_152.

[21] Dwight Guth: *A formal semantics of Python 3.3*. Available at http://hdl.handle.net/2142/45275. Accessed on 12th, September 2022.

[22] Dwight Guth, Chris Hathhorn, Manasvi Saxena & Grigore Roşu: *RV-Match: Practical semantics-based program analysis*. In: *Proceedings of the 28th International Conference on Computer Aided Verification (CAV'16)*, 9779, Springer, pp. 447–453, doi:10.1007/978-3-319-41528-4_24.

[23] Chris Hathhorn, Chucky Ellison & Grigore Roşu: *Defining the undefinedness of C*. In: *Proceedings of the 36th annual ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'15)*, ACM, pp. 336–345, doi:10.1145/2813885.2737979.

[24] Everett Hildenbrandt, Manasvi Saxena, Xiaoran Zhu, Nishant Rodrigues, Philip Daian, Dwight Guth, Brandon Moore, Yi Zhang, Daejun Park, Andrei Ştefănescu & Grigore Roşu: *KEVM: A complete semantics of the Ethereum virtual machine*. In: *Proceedings of the 2018 IEEE Computer Security Foundations Symposium (CSF'18)*, IEEE, pp. 204–217, doi:10.1109/CSF.2018.00022.

[25] K Team: *Matching logic proof checker*. Available at https://github.com/kframework/matching-logic-proof-checker. Accessed on 29th, April 2022.

[26] Shuanglong Kan, David Sanan, Shang-Wei Lin & Yang Liu: *KRust: An executable formal semantics for Rust*, doi:10.48550/arXiv.1804.10806.

[27] Theodoros Kasampalis, Daejun Park, Zhengyao Lin, Vikram S. Adve & 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*, ASPLOS 2021, Association for Computing Machinery, New York, NY, USA, pp. 1004–1019, doi:10.1145/3445814.3446751.

[28] Chantal Keller & Benjamin Werner (2010): *Importing HOL Light into Coq*. In: *International Conference on Interactive Theorem Proving*, Springer, pp. 307–322, doi:10.1007/978-3-642-14052-5_22.

[29] Robbert Krebbers, Jacques-Henri Jourdan, Ralf Jung, Joseph Tassarotti, Jan-Oliver Kaiser, Amin Timany, Arthur Charguéraud & Derek Dreyer (2018): *MoSeL: A general, extensible modal framework for interactive proofs in separation logic*. *Proceedings of the ACM on Programming Languages* 2(ICFP), pp. 1–30, doi:10.1145/3236772.

[30] Robbert Krebbers, Amin Timany & Lars Birkedal (2017): *Interactive proofs in higher-order concurrent separation logic*. In Giuseppe Castagna & Andrew D. Gordon, editors: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, ACM, pp. 205–217, doi:10.1145/3009837.3009855.

[31] Xavier Leroy (2007): *A locally nameless solution to the POPLmark challenge*. Available at https://xavierleroy.org/POPLmark/locally-nameless/. Accessed on 12th, September 2022.

[32] Max Planck Institute for Software Systems: *Coq-std++: An extended "standard library" for Coq*. Available at https://gitlab.mpi-sws.org/iris/stdpp. Accessed on 12th, September 2022.

[33] Conor McBride & James McKinna (2004): *Functional pearl: I am not a number-I am a free variable*. In Henrik Nilsson, editor: *Proceedings of the ACM SIGPLAN Workshop on Haskell, Haskell 2004, Snowbird, UT, USA, September 22-22, 2004*, ACM, pp. 1–9, doi:10.1145/1017472.1017477.

[34] Andrew McCreight (2009): *Practical tactics for separation logic*. In: *International Conference on Theorem Proving in Higher Order Logics*, Springer, pp. 343–358, doi:10.1007/978-3-642-03359-9_24.

[35] Norman Megill & David A. Wheeler: *Metamath: A computer language for mathematical proofs*. Available at http://us.metamath.org. Accessed on 12th, September 2022.

[36] Daejun Park, Andrei Stefanescu & Grigore Roşu (2015): *KJS: a complete formal semantics of JavaScript*. In David Grove & Steve Blackburn, editors: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, ACM, pp. 346–356, doi:10.1145/2737924.2737991.

[37] James F Power & Caroline Webster (1999): *Working with linear logic in Coq*. Available at https://mural.maynoothuniversity.ie/6461/1/JP-Working-Linear-Logic.pdf. Accessed on 12th, September 2022.

[38] Grigore Roşu (2017): *Matching logic*. *Log. Methods Comput. Sci.* 13(4), doi:10.23638/LMCS-13(4:28)2017.

[39] Matthieu Sozeau & Cyprien Mangin (2019): *Equations reloaded: high-level dependently-typed functional programming and proving in Coq*. *Proc. ACM Program. Lang.* 3(ICFP), doi:10.1145/3341690.

[40] Alfred Tarski (1955): *A lattice-theoretical fixpoint theorem and its applications*. *Pacific J. Math.* 5(2), pp. 285–309, doi:10.2140/PJM.1955.5.285.

[41] Feng Wang, Fu Song, Min Zhang, Xiaoran Zhu & Jun Zhang: *KRust: A formal executable semantics of Rust*. In: *Proceedings of the 12th International Symposium on Theoretical Aspects of Software Engineering (TASE'18)*, IEEE, pp. 44–51, doi:10.1109/TASE.2018.00014.

[42] Freek Wiedijk (2007): *Encoding the HOL Light logic in Coq*. Available at https://www.cs.ru.nl/~freek/notes/holl2coq.pdf. Accessed on 12th, September 2022.

[43] Bruno Xavier, Carlos Olarte, Giselle Reis & Vivek Nigam (2018): *Mechanizing focused linear logic in Coq*. *Electronic Notes in Theoretical Computer Science* 338, pp. 219–236, doi:10.1016/j.entcs.2018.10.014.