

TDK-dolgozat

Kurucz Ádám



EÖTVÖS LORÁND UNIVERSITY

Faculty of Informatics

Department of Programming Languages and Compilers

Implementation of the unification algorithm in matching logic

Dr. Dániel Horpácsi
Assistant Professor

Ádám Kurucz
Computer Science MSc

Budapest, 2024.05.13

Abstract

Matching logic is a logical framework for specifying and reasoning about programs using pattern matching semantics. A pattern is made up of a number of structural components and constraints. Structural components are syntactically matched, while constraints need to be satisfied. Having multiple structural patterns poses a problem, as it requires multiple matching operations. This is easily remedied by unification, for which an algorithm has already been defined and proven correct in matching logic; however, it is not included in the existing formalizations, including the one in Coq, that this thesis builds upon. We present a machine-checked formalization for the unification algorithm with the proof system of matching logic and prove its correctness based on related work.

Contents

1	Introduction	1
2	Background	3
2.1	Locally nameless representation	3
2.2	Matching Logic	4
2.3	Formalization	13
2.4	Unification	14
3	Unification in Applicative Matching Logic	18
3.1	Substitutions	18
3.2	Going from polyadic to applicative	19
3.3	Injectivity axiom	20
3.4	Unification problem	21
4	Proofs	25
5	Conclusion	31
5.1	Discussion	31
5.2	Future work	33

1 Introduction

Matching logic [1] is a unifying logical framework for specifying programming language semantics and reasoning about the behaviour of programs. It serves as the foundation of the $\mathbb{K}$ framework which is used to give a formal definition of programming languages and automatically generate various tools for handling the language. Several real world languages have already been fully defined as matching logic theories, such as C [2], Java [3], Python [4], Rust [5], x86_64 assembly [6], EVM [7], or LLVM [8].

While $\mathbb{K}$ may also be used to reason about the properties of these languages, and it provides comprehensive support for automation of these proofs, as well as integration with SMT solvers, it does not have a way to manually provide proofs in the event that automation fails. These additional proofs need to be done on paper and added to the system as trusted lemmas. However, since these are all essentially matching logic theories, they could be proven using a proof assistant and then imported into $\mathbb{K}$. The Coq formalization [9] aims to solve precisely this problem.

Matching logic formulas are called patterns, which consist of variables, symbols, various connectives known from first order logic, and fixed points. These patterns may be interpreted as a set of elements over a carrier set that match them. For example, a pattern like $0 \vee 1$ over the set of natural numbers would be matched by either 0 or 1. These patterns are called structural, and they resemble terms from first order logic. Some patterns may evaluate only to either the empty set or the full carrier set, in which case we call them predicate patterns and use them to limit what was matched by other structural patterns. Note that unlike in first order logic, terms and formulas are both patterns, and not distinguished from each other.

However, in case there are multiple structural components in a pattern, which may be introduced by connectives such as conjunction, finding the elements that match them all is not an easy task. Specifically, we lose the ability to use an SMT solver to solve the predicate constraints. For this reason, it would be beneficial to reduce any additional structural parts to predicates. This can be achieved with unification, by keeping only one of the structural patterns and setting the remaining patterns

equal to it. This is still difficult to solve, however, using the algorithm shown in [10], it is possible to further reduce that to an externally solvable predicate. This thesis adapts the results of that paper to the applicative variant of the logic and proves them with the logic's proof system (i.e. provides syntactic proofs) using the aforementioned Coq formalization.

Contributions. The main contributions of this thesis are:

- A unification algorithm and its semantic proof expressed in applicative matching logic, derived from the polyadic variant.
- A general definition of the unification problem, independent of the underlying container
- A machine checked implementation of the unification algorithm, proving its correctness

The rest of the thesis is organised as follows. Chapter 2 discusses the specifics of matching logic, its formalization, and unification. Chapter 3 describes the adaptation of the algorithm and the unification problem. Chapter 4 highlights some of the lemmas and the notable changes and challenges brought about by the syntactical proofs. Finally, Chapter 5 concludes the thesis and discusses future work.

2 Background

In this chapter we present the required background knowledge on matching logic, its locally nameless representation, the Coq formalization and its proof mode based on sequent calculus, some of the theorems that are already present and proven in the formalization, that our proofs will rely on, as well as provide a formal introduction to the unification algorithm and its related concepts.

2.1 Locally nameless representation

The syntax used in the formalization, and therefore in this thesis is a locally nameless one [11]. This is a combination of the traditional named representation commonly seen in first order logic, and the totally nameless one, that utilizes de Bruijn indices to denote every variable. De Bruijn indices are natural numbers that for bound variables show how many binders they are in scope of, and for free variables provides a unique index, which could be used to keep track of the formula's context. Take for example the following expression, in named, nameless, and locally nameless form:

$$P(x) \vee P(y) \wedge \exists x.Q(x, y)$$

$$P(\underline{0}) \vee P(\underline{1}) \wedge \exists.Q(\underline{0}, \underline{2})$$

$$P(x) \vee P(y) \wedge \exists.Q(\underline{0}, y)$$

Here we immediately see the issues with the named and nameless versions. In the first one, we see two uses of the variable x but they refer to two different things. The $\exists$ quantifier shadows the x from the outer scope, and the only way to use it is to rename the variable of the quantifier. Which showcases yet another issue: although $P(x) \vee P(y) \wedge \exists z.Q(z, y)$ is equivalent to the original expression, they are not syntactically equal. This is called α -equivalence. In the case of a more complicated formula, reasoning about α -equivalence can be tiresome, especially in a machine-checked formalism. Further, such a renaming is error-prone, as had we

chosen to rename the bound variable to y , that would have altered the meaning of the formula. Whenever we want to rename or introduce a bound variable, we also need to reason about the name being fresh in the formula.

All of these problems disappear in the totally nameless version, and so that makes it useful for defining mathematical models and formalizing and reasoning about them in a computer-aided environment. However, it is much harder to understand for humans. Notice how y is referred to by two different numbers. This is because the $\exists$ quantifier introduces a new $\underline{0}$ to refer to its bound variable, shifting every index up by one. This is useful because the index can also be interpreted as the number of quantifiers it is in scope of that we need to “jump over” to get to the one binding the variable. In this case it is zero, but in a formula like $\exists.\exists.\underline{1}$, the $\underline{1}$ is bound by the outer $\exists$, and we need to jump over the inner one to get to it. Any index that is larger than the number of quantifiers that binds it refers to a free variable. This can cause confusion however, as in the example above, we use $\underline{0}$ to refer to two different variables (both were x in the original), and for human minds it is easy to think they are indeed the same.

This issue is exactly what the locally nameless representation aims to solve. Remember, in the original formula, the problem was only with the bound variables, so it should be enough to represent only those with de Bruijn indices. This still makes it so that we do not need to reason about α -equivalence or worry about capturing a variable when renaming or substituting, but by making free and bound variables syntactically distinct, it is also easy for humans to understand and work with. The one major disadvantage of this method is that now it is possible to create ill-formed formulas, by using an index that is too high. This can be remedied using a type system that will reject such expressions or by introducing a side condition that requires all formulas to be well-formed.

2.2 Matching Logic

In this section we introduce the syntax, semantics, and proof system of matching logic [1, 12].

Definition 1 (Signature). A matching logic signature is the 3-tuple (EV, SV, Σ) , where

- EV is a countably infinite set of element variables, denoted with $x, y, \dots$,
- SV is a countably infinite set of set variables, denoted with $X, Y, \dots$,
- Σ is a countable set of constant symbols, denoted with $f, g, \dots$.

When EV and SV are understood from the context, only Σ is used to denote the whole signature.

Definition 2 (Pseudo-pattern). Given a (EV, SV, Σ) signature, we use the following grammar to define pseudo-patterns:

$$\varphi := x \mid X \mid \underline{n} \mid \underline{\underline{N}} \mid f \mid \varphi_1 \varphi_2 \mid \perp \mid \varphi_1 \rightarrow \varphi_2 \mid \exists.\varphi \mid \mu.\varphi$$

where

- $x \in EV$ is a free element variable,
- $X \in SV$ is a free set variable,
- $\underline{n}$ ($n \in \mathbb{N}$) is a de Bruijn index representing an element variable bound by $\exists$,
- $\underline{\underline{N}}$ ($N \in \mathbb{N}$) is a de Bruijn index representing a set variable bound by μ ,
- $f \in \Sigma$ is a constant symbol,
- $\varphi_1 \varphi_2$ is application, where φ_1 is applied to φ_2 ,
- $\perp$ is a special pattern called bottom,
- $\varphi_1 \rightarrow \varphi_2$ is the generalization of the propositional implication,
- $\exists.\varphi$ is the existential quantification, known from first order logic,
- $\mu.\varphi$ is the least fixed point.

Application is left associative and binds the tightest. Implication is right associative. The scope of binders extends as far to the right as possible. Also note that there are two kinds of de Bruijn indices, one for each quantifier, but the number only takes into account the correct one ($\exists$ for $\underline{n}$ and μ for $\underline{\underline{N}}$).

We use $FV(\varphi)$ to refer to the set of free variables of some pattern φ .

Derived operators

Using the above definition we can derive other patterns as syntactic sugar. These are, in order: negation, disjunction, conjunction, top, equivalence, and universal quantification.

$$\begin{array}{ll}
\neg\varphi \equiv \varphi \rightarrow \perp & \varphi_1 \vee \varphi_2 \equiv \neg\varphi_1 \rightarrow \varphi_2 \\
\varphi_1 \wedge \varphi_2 \equiv \neg(\neg\varphi_1 \vee \neg\varphi_2) & \top \equiv \neg\perp \\
\varphi_1 \leftrightarrow \varphi_2 \equiv (\varphi_1 \rightarrow \varphi_2) \wedge (\varphi_2 \rightarrow \varphi_1) & \forall.\varphi \equiv \neg\exists.\neg\varphi
\end{array}$$

In polyadic matching logic [12], application is written as $f(x_1, \dots, x_n)$. While this thesis is only concerned with applicative matching logic, the paper that serves as its basis [10] uses the polyadic version to describe unification. We will convert these applications to the applicative version, however, this form could also be thought of as a derived notation for $f\ x_1 \dots x_n$.

Well-formedness

First we define positive patterns, to ensure that our fixed points are positive.

Definition 3 (Positive pattern). A pattern is considered positive if, for every $\mu.\varphi$, the newly bound $\underline{0}$ does not appear negatively in φ , that is, it does not appear on the left side of an odd number of implications. Note that if φ contains additional μ binders, the de Bruijn index needs to be incremented, so as to ensure we are still checking for the negativity of the same variable.

Next we introduce closed patterns to ensure variables are correctly bound.

Definition 4 (Closed pattern). A pattern is considered closed with respect to a given binder if all bound variables are actually bound by an instance of that quantifier. This means that every de Bruijn index needs to be in scope of at least as many of the corresponding binders as the index itself.

Finally we combine the two to define (well-formed) patterns.

Definition 5 (Pattern). A pattern is said to be well-formed if it is positive and closed for both the $\exists$ and μ binders. We will refer to well-formed patterns as simply patterns.

The following is an example of a well-formed pattern:

$$\exists.x \rightarrow f \underline{0} \rightarrow \perp \vee \mu.\underline{0} \vee g X \underline{0}$$

This is an example of a pattern that violates both closedness and positivity:

$$\mu.\underline{0} \rightarrow \underline{1}$$

Given some o and s symbols in Σ , the following is a way to describe a set isomorphic to the natural numbers:

$$\mu.o \vee s \underline{0}$$

Next we define substitutions in the usual way.

Definition 6 (Substitution, opening). We denote substitutions as $\varphi[\psi/x]$ to say that we replace every occurrence of x in φ with ψ . x may be a free or bound element or set variable. If the variable $\underline{0}$ or $\underline{0}$ is substituted by a fresh variable, we can also call this substitution the opening of the $\exists$ or μ binder.

A special instance of substitutions is contexts.

Definition 7 (Pattern context, application context). A pattern context, denoted with C is a pattern with a distinguished variable $\square$, called a hole. When substituting φ into a hole, we simply write $C[\varphi]$ instead of the above mentioned $C[\varphi/\square]$. When a context contains only applications from the root of the context to the $\square$, i.e. it can be described by the following grammar:

$$C^\$:= \square \mid \varphi C^\$ \mid C^\$ \varphi$$

then it is called an application context, and will be denoted $C^\$$.

Contexts are useful for defining theorems that are only concerned with a smaller part of a pattern while the rest is irrelevant. A good example of this is the extensive use of application contexts in defining the proof system of matching logic. See Table 2.1 for more details.

Pattern contexts are also used in defining the congruence theorem which allows us to replace equivalent patterns in an arbitrary context:

Theorem 1 (Congruence). Let φ_1 and φ_2 be patterns, and C be a pattern context. If we know that $\varphi_1 \leftrightarrow \varphi_2$ then it is also true that $C[\varphi_1] \leftrightarrow C[\varphi_2]$.

Semantics of matching logic

Although one of the significant contributions of this thesis is providing syntactical counterparts to the semantic proofs presented in [10], it is still beneficial to discuss some of the underlying semantic meanings to the patterns to better understand why certain statements are expressed the way they are and how some of the proofs work.

Matching logic semantics is based on pattern matching. Patterns are interpreted as a set of elements that match them. But to understand exactly what these sets are, we must first introduce models.

Definition 8 (Model). A model is a tuple $(M, _ \cdot _, M_\sigma)$, where

- M is the nonempty carrier set, which contains all elements a pattern may evaluate to,
- $_ \cdot _$ is a binary function that serves as the interpretation of application,
- $M_\sigma \subseteq M$ is the interpretation of all symbols $\sigma \in \Sigma$.

As before, we may use M to refer to the entire model. Patterns can be defined and constructed independently of models, as models are only used when evaluating patterns. Special cases of patterns, such as predicate or term patterns must fulfill their extra condition (evaluating to the full, empty, or singleton sets) in every model.

If we choose our carrier set to be the natural numbers, and numerical symbols to mean their corresponding natural number, then we can use the pattern $0 \vee 1$ to denote the set $\{0, 1\}$. Note that the 0 and 1 in the pattern may refer to different numbers if we choose so.

Because patterns always evaluate to sets, and both the arguments and result of application are patterns, the expected type of $_ \cdot _$ is $\mathcal{P}(M) \rightarrow \mathcal{P}(M) \rightarrow \mathcal{P}(M)$. However, it is enough to define $_ \cdot _$ with the type $M \rightarrow M \rightarrow \mathcal{P}(M)$, i.e. with the parameters being only the members and not subsets of the carrier set, because this can be expanded pointwise to work with subsets as well in the following way:

$$A \cdot B = \bigcup_{\substack{a \in A \\ b \in B}} a \cdot b$$

This expanded function will also be called $_ \cdot _$ for simplicity.

Given a signature and a model, we can define interpretations for our patterns. Element variables are interpreted as singleton sets, while set variables may be interpreted as any set. The meaning of symbols is given by M_σ and in the case of application, the meanings of the subpatterns may be combined using the $_ \cdot _$ function. $\perp$ is matched by the empty set and the implication $\varphi_1 \rightarrow \varphi_2$ is matched by elements that match φ_2 if they match φ_1 , like in first order logic. $\exists.\varphi$ is matched by all instances of φ as $\underline{0}$ ranges over M . This means, for example that $\exists.\underline{0}$ is matched by M . Similarly, $\mu.\varphi$ is matched by the least fixed point of φ as $\underline{0}$ ranges over $\mathcal{P}(M)$. Note that $\exists$ only binds element variables, while μ only binds set variables. The semantics of the derived operations follow from these.

Proof system of matching logic

Matching logic has a Hilbert-style proof system with a handful of simple rules, shown in Table 2.1, which makes it small and easy to implement.

The proof rules in are divided into four categories:

- **FOL reasoning**, which contains basic proof rules and propositions from first order logic
- **Frame reasoning**, consisting of propagation and framing rules that allow propagating formal reasoning that is done within an application context, throughout the context
- **Fixpoint reasoning**, containing fixpoint rules as in modal μ -calculus
- **Technical rules**

FOL reasoning	
PROPOSITION 1	$\varphi_1 \rightarrow (\varphi_2 \rightarrow \varphi_1)$
PROPOSITION 2	$(\varphi_1 \rightarrow (\varphi_2 \rightarrow \varphi_3)) \rightarrow (\varphi_1 \rightarrow \varphi_2) \rightarrow (\varphi_1 \rightarrow \varphi_3)$
PROPOSITION 3	$((\varphi \rightarrow \perp) \rightarrow \perp) \rightarrow \varphi$
MODUS PONENS	$\frac{\varphi_1 \quad \varphi_1 \rightarrow \varphi_2}{\varphi_2}$
$\exists$ -QUANTIFIER	$\varphi[x/\underline{0}] \rightarrow \exists.\varphi$
$\exists$ -GENERALIZATION	$\frac{\varphi_1[x/\underline{0}] \rightarrow \varphi_2}{(\exists.\varphi_1) \rightarrow \varphi_2} \quad \text{if } x \text{ is fresh in } \varphi_2$
Frame reasoning	
PROPAGATION $_{\perp}$	$C^{\$}[\perp] \rightarrow \perp$
PROPAGATION $_{\vee}$	$C^{\$}[\varphi_1 \vee \varphi_2] \rightarrow C^{\$}[\varphi_1] \vee C^{\$}[\varphi_2]$
PROPAGATION $_{\exists}$	$C^{\$}[\exists.\varphi] \rightarrow \exists.C^{\$}[\varphi]$
FRAMING	$\frac{\varphi_1 \rightarrow \varphi_2}{C^{\$}[\varphi_1] \rightarrow C^{\$}[\varphi_2]}$
Fixpoint reasoning	
SUBSTITUTION	$\frac{\varphi}{\varphi[\psi/X]}$
PRE-FIXPOINT	$\varphi[(\mu.\varphi)/\underline{0}] \rightarrow \mu.\varphi$
KNASTER-TARSKI	$\frac{\varphi[\psi/\underline{0}] \rightarrow \psi}{(\mu.\varphi) \rightarrow \psi}$
Technical rules	
EXISTENCE	$\exists.\underline{0}$
SINGLETON	$\neg(C_1^{\$}[x \wedge \varphi] \wedge C_2^{\$}[x \wedge \neg\varphi])$

Table 2.1: Hilbert-style proof system for matching logic under locally nameless representation

We also define a relation called provability, written as $\Gamma \vdash \varphi$. Here Γ is a set of patterns called a theory and its members called are axioms. When the relation holds we say that φ is provable from Γ . Whenever a smaller theory is included in the theory we are working with, it is assumed that its symbols are also part of the signature.

Definedness

One of the theories that is used heavily in unification is that of definedness. It introduces only a single symbol, $\lceil _ \rceil$. When it is applied to pattern $(\lceil _ \rceil \varphi)$ we instead use the notation $\lceil \varphi \rceil$. This is a predicate pattern, evaluating to the full set if φ is matched by anything (i.e. it is defined), and to the empty set if it is not.

With this we can also define a related concept, called totality. It uses the symbol $\lfloor _ \rfloor$, and similarly we may use the notation $\lfloor \varphi \rfloor$ when applying it to a pattern. Totality is also a predicate pattern, that evaluates to the full set if φ does too, and to the empty set, if it evaluates to anything else. Totality can be expressed with definedness as $\lfloor \varphi \rfloor := \neg \lceil \neg \varphi \rceil$.

Using totality it is now possible to define equality. $\varphi_1 = \varphi_2 := \lfloor \varphi_1 \leftrightarrow \varphi_2 \rfloor$. This intuitively means that φ_1 and φ_2 need to be matched by the same set. We can also define the usual set operations, such as $\in$ and $\subseteq$. The negations of these can also be defined trivially. See below for more details.

We present the formal specification of the theory of definedness. The section Symbol includes the new symbols introduced, which will be part of the signature, but not the derived ones. Notation describes the usage of these symbols, and introduces the derived patterns, while Axiom describes the axioms that will be part of the theory.

spec DEFINEDNESS	
Symbol: $\lceil _ \rceil$	
Notation:	
$\lceil \varphi \rceil \equiv \lceil _ \rceil \varphi$	$\lfloor \varphi \rfloor \equiv \neg \lceil \neg \varphi \rceil$
$\varphi_1 = \varphi_2 \equiv \lfloor \varphi_1 \leftrightarrow \varphi_2 \rfloor$	$\varphi_1 \neq \varphi_2 \equiv \neg(\varphi_1 = \varphi_2)$
$\varphi_1 \in \varphi_2 \equiv \lceil \varphi_1 \wedge \varphi_2 \rceil$	$\varphi_1 \notin \varphi_2 \equiv \neg(\varphi_1 \in \varphi_2)$
$\varphi_1 \subseteq \varphi_2 \equiv \lfloor \varphi_1 \rightarrow \varphi_2 \rfloor$	$\varphi_1 \not\subseteq \varphi_2 \equiv \neg(\varphi_1 \subseteq \varphi_2)$
Axiom:	
(DEFINEDNESS) $\forall. \lceil \underline{0} \rceil$	
endspec	

We may also use equality, $\exists$, and $\forall$ s to define functional patterns. A constant σ is functional if $\exists. \sigma = \underline{0}$. A symbol f may be thought of as a binary function if $\forall. \forall. \exists. f \underline{2} \underline{1} = \underline{0}$.

Functional patterns may also be referred to as term patterns. Intuitively term patterns can be thought of as either variables, symbols, or terms applied to other terms. We can also think of them as patterns that are matched by singleton sets. Unification will only be defined over term patterns.

Predicate patterns can also be expressed using equality, syntactically expressing that φ either matches nothing, or everything. If φ is a predicate then the following holds:

$$\varphi = \perp \vee \varphi = \top$$

There are a number of theorems of the formalization which this work relies on:

Theorem 2. Let Γ be a theory and φ a pattern. Then $\Gamma \vdash \varphi \rightarrow [\varphi]$.

This is a consequence of the DEFINEDNESS axiom and $\exists$ -GENERALIZATION.

Theorem 3. Let Γ be a theory and φ_1 and φ_2 functional patterns. Then $\Gamma \vdash \varphi_1 \in \varphi_2 \rightarrow \varphi_1 = \varphi_2$.

The following is used to extract and insert an extra condition into both sides of an equivalence.

Theorem 4. Let Γ be a theory and φ_1, φ_2 , and φ_3 be patterns. Then $\Gamma \vdash (\varphi_1 \wedge \varphi_2 \leftrightarrow \varphi_1 \wedge \varphi_3) \leftrightarrow (\varphi_1 \rightarrow \varphi_2 \leftrightarrow \varphi_3)$.

The above theorem is also true if φ_1 is on the right side, by the commutativity of conjunction.

Deduction theorem

In first order logic, the deduction theorem is used to move part of the conclusion into the theory. This way it is possible to continue with the rest of the conclusion as the goal without losing any information, because it is still accessible in the theory. The following is the formal statement of the deduction theorem in first order logic:

Theorem 5 (Deduction theorem). Let Γ be a theory and φ_1 and φ_2 formulas. If $\Gamma \cup \{\varphi_1\} \vdash \varphi_2$ then $\Gamma \vdash \varphi_1 \rightarrow \varphi_2$.

However, in first order logic all formulas are predicates, which is not the case in matching logic. One possible option to adapt the theorem would be to restrict its usage to predicate patterns, but this is not necessary, as a less restrictive version can also be proven that uses totality on the pattern, which always makes it a predicate pattern. The following deduction theorem holds in matching logic. Its proof can be found in [12].

Theorem 6 (Deduction theorem in matching logic). Let Γ be a matching logic theory and φ_1 and φ_2 patterns. If $\Gamma \cup \{\varphi_1\} \vdash \varphi_2$ then $\Gamma \vdash [\varphi_1] \rightarrow \varphi_2$.

However, there are some restrictions to this theorem. It can only be used if

- the proof premise (i.e. $\Gamma \cup \{\varphi_1\} \vdash \varphi_2$) does not use $\exists$ -GENERALIZATION on the free variables of φ_1 and
- the proof premise does not use SUBSTITUTION for free set variables of φ_1 and
- if the proof premise uses KNASTER-TARSKI, then $\underline{0}$ cannot appear under a μ binder in φ_2 .

Recall that equality is defined as the totality of an equivalence, therefore this theorem can also be used to extract the equivalence from an object level equality. This can be combined, for example, with the congruence theorem to get an object level version with equality:

Corollary 1 (Congruence with equality). Let Γ be a theory, φ_1 and φ_2 patterns, and C a pattern context. Then $\Gamma \vdash \varphi_1 = \varphi_2 \rightarrow C[\varphi_1] \leftrightarrow C[\varphi_2]$.

This will serve as the basis for object level rewriting.

Theorem 4 can also be extended to work with equality using the deduction theorem:

Corollary 2. Let Γ be a theory, φ_1 a predicate pattern, and φ_2 , and φ_3 be patterns. Then $\Gamma \vdash (\varphi_1 \wedge \varphi_2 = \varphi_1 \wedge \varphi_3) \leftrightarrow (\varphi_1 \rightarrow \varphi_2 = \varphi_3)$.

2.3 Formalization

In the formalization [13, 14], pseudo-patterns are defined as an inductive type, simply called `Pattern`, and well-formedness is expressed as a boolean predicate named `well_formed`, as a conjunction of three helper predicates, `well_formed_positive`, `well_formed_closed_ex_aux`, and `well_formed_closed_mu_aux`, corresponding to the three well-formedness conditions in Section 2.2. Here the latter two also take a natural parameter, allowing us to express that a pattern is mostly closed,

save for some pre-determined amount of free element or set variables. This is useful for stating that the body of a quantified expression is partially well-formed, even when the quantifier will only get added later.

While freestanding **Patterns** are allowed to not be well-formed, every theorem, lemma and definition that is expressed on a **Pattern** also requires that it is well-formed. This is normally done through an additional parameter representing the proof of well-formedness, although most of the time a dependent pair (written in the code as `sig well_formed`, or using the easier to understand equivalent notation `{p : Pattern | well_formed p}`) would also suffice. We may refer to this form as a “well-formed pattern” or **WFPattern** in the code and it can also be used to return a pattern and a well-formedness proof at the same time from a definition.

The provability relation defined above serves as the basis for the proof mode defined in Coq. The proof mode depends on the Hilbert-style system, however, in most cases it is difficult to rely solely on that to prove complicated statements. For this reason the proof mode employs a single-conclusion sequent calculus that features several user-facing tactics that mirror Coq’s own tactic system, making it simple for a user familiar with Coq to prove matching logic theorems in the formalization.

The proof mode can manipulate the proof context by handling object level hypotheses. This is done by treating the left sides of implications as hypotheses. They can be introduced, named, and used just like hypotheses at the meta level. It is also possible to make use of already proven theorems available as a meta-level hypothesis inside the object level proof mode.

2.4 Unification

Unification [10] is the process of trying to identify expressions by replacing variables with other expressions until they are syntactically equal. For this thesis, the expressions will be term patterns.

Arusoaie and Lucanu define an alternative “set of bindings” notation for substitutions, e.g. $\sigma = \{x_1 \mapsto t_1, x_2 \mapsto t_2, \dots, x_n \mapsto t_n\}$. Applying this substitution to a pattern t is denoted $t\sigma$. Whether this application is sequential or parallel and if they

are sequential, what the order is, or if they are parallel, what to do with conflicts (i.e. when a variable is substituted for two different patterns) is unspecified in the paper. We will discuss how we chose to resolve this issue in Chapter 3.

Before we introduce the unification algorithm, we need to specify common operations on these substitutions. First, how to compose two substitutions.

Definition 9 (Composition of substitutions). Let $\sigma = \{x_1 \mapsto t_1, \dots, x_n \mapsto t_n\}$ and η be two substitutions. The composition of these, written as $\sigma\eta$ can be defined as $\sigma\eta := \{x_1 \mapsto t_1\eta, \dots, x_n \mapsto t_n\eta\}$, i.e. we apply the substitution η to every pattern on the right side of σ .

Next we define how to determine if two substitutions are equal.

Definition 10 (Equality of substitutions). Two substitutions σ and η are equal if they are extensionally equal: $\forall t. t\sigma = t\eta$, that is if their effect is the same when applied to any term.

We can combine these two operations to define when one substitution is more general than another.

Definition 11 (More general substitution). We say that σ is more general than η , written as $\sigma \leq \eta$ if there is a substitution θ such that $\sigma\theta = \eta$.

The algorithm will produce a substitution that, when applied to the input terms, produces the same term. We call this substitution a unifier.

Definition 12 (Unifier, unifiable terms). A substitution σ is called a unifier of two patterns t_1 and t_2 if $t_1\sigma = t_2\sigma$, using the definition of equality above. If there exists a unifier of two terms, we call the terms unifiable.

However, there may be infinitely many unifiers that only differ in some insignificant details. For example, if the terms are fx and fy , we could replace x and y with any term, but all that matters is that they need to be the same. To solve this, we introduce the concept of the most general unifier, using the more general relation from above.

Definition 13 (Most general unifier). A unifier σ is called the most general if for any unifier θ , σ is more general than θ ($\sigma \leq \theta$).

Delete	$P \cup \{(t, t)\} \Rightarrow P$
Decomposition	$P \cup \{(f(t_1, \dots, t_n), f(u_1, \dots, u_n))\} \Rightarrow$ $P \cup \{(t_1, u_1), \dots, (t_n, u_n)\}$
Symbol clash	$P \cup \{(f(t_1, \dots, t_n), g(u_1, \dots, u_m))\} \Rightarrow \perp$
Orient	$P \cup \{(t, x)\} \Rightarrow P \cup \{(x, t)\}$ if t is not a variable
Occurs check	$P \cup \{(x, t)\} \Rightarrow \perp$ if $x \in FV(t)$
Elimination	$P \cup \{(x, t)\} \Rightarrow P\{x \mapsto t\} \cup \{(x, t)\}$ if $x \notin FV(t)$

Table 2.2: Rules of the unification algorithm. Note that these rules are for polyadic matching logic. We refer to Table 3.1 for how we have rephrased them for applicative matching logic.

It follows from the definition of the more general unifier that any other solution may be obtained from this by instantiation. In this thesis we will not consider higher order unification (i.e. variables may not stand in for functions).

There are several algorithms for solving unification problems [15] but in this thesis we will be utilizing a rule based approach. For this we will require a data structure, appropriately named unification problem. This is either a set of pairs of term patterns, or a special symbol $\perp$, representing an unsolvable problem.

The algorithm is represented by a “unification step” relation, denoted as $P \Rightarrow P'$ for some P and P' unification problems. This relation is inductively defined by the rules of the algorithm. See Table 2.2 for the rules. The reflexive-transitive closure of this relation is written as $P \Rightarrow^* P'$. It is assumed that the left side of $\Rightarrow$, $\Rightarrow^*$, and $\cup$ is not $\perp$.

Definition 14 (Solved form). We say that the unification problem is in solved form if it is $\perp$ or a set $\{(x_1, t_1), \dots, (x_n, t_n)\}$, where $\forall i. x_i$ is a variable $\wedge \forall j. x_i \notin t_j$ (i.e. the first term in the pairs is a variable that does not appear in the second term of any of the pairs).

A unification problem that is in solved form and is not $\perp$ can be seen as a substitution. If we start with a unification problem with just the two terms we want to unify as a pair, then apply the rules of the algorithm, we will either end up with $\perp$, in which case the terms are not unifiable, or a unification problem whose corresponding substitution is the most general unifier of the terms. These claims are proven in [10] and formalized in this thesis.

The opposite of the above is also true:

Theorem 7. If σ is the most general unifier of t_1 and t_2 , then $\exists P, \{(t_1, t_2)\} \Rightarrow^* P$ and P is in solved form, with its corresponding substitution being σ .

This is essentially treated as an axiom in the paper, a consequence of properties proven in [15], so we will treat it as an axiom in the formalization as well.

Example 1. The following is an example run of the algorithm with terms $f(x, g(1), g(z))$ and $f(g(y), g(y), g(g(x)))$.

$$\begin{aligned}
& \{\underline{f(x, g(1), g(z))}, \underline{f(g(y), g(y), g(g(x)))}\} && \text{Decomposition} \\
& \{(x, g(y)), \underline{(g(1), g(y))}, (g(z), g(g(x)))\} && \text{Decomposition} \\
& \{(x, g(y)), (1, y), \underline{(g(z), g(g(x)))}\} && \text{Decomposition} \\
& \{(x, g(y)), \underline{(1, y)}, (z, g(x))\} && \text{Orient} \\
& \{(x, g(y)), \underline{(y, 1)}, (z, g(x))\} && \text{Elimination} \\
& \{\underline{(x, g(1))}, (y, 1), (z, g(x))\} && \text{Elimination} \\
& \{(x, g(1)), (y, 1), (z, g(g(1)))\}
\end{aligned}$$

Substitutions and unification problems as predicates

It is possible to turn substitutions of the form $\{x_1 \mapsto t_1, \dots, x_n \mapsto t_n\}$ and unification problems of the form $\{(x_1, t_1), \dots, (x_n, t_n)\}$ into the predicate pattern $\bigwedge_{i=1}^n x_i = t_i$. These predicate patterns are denoted ϕ^σ for a substitution σ or ϕ^P for a unification problem P .

In case a substitution σ is the substitution corresponding to a unification problem P , it is true that $\phi^\sigma = \phi^P$. Since ϕ^P is used almost exclusively in lemmas, we will utilize this fact to connect a unification problem to its substitution, instead of trying to do type conversions to extract a substitution from one directly.

3 Unification in Applicative Matching Logic

In this chapter we present the design decisions and details of the formalization of the algorithm.

3.1 Substitutions

An important decision that needed to be made early in the project is what to do with substitutions. The original paper [10] does not mention whether the substitutions are sequential or parallel and it is not immediately obvious if one is better than the other.

Sequential substitutions are the ones already existing in the formalization so it would be much preferred if these could work, as there are many lemmas regarding these that would all need to be redefined and reproven if we had to use parallel substitutions. The sequential version is also easier to use and reason about, as the parallel ones would have had to be defined as a function, and built one substitution at a time. And this still would not solve the issue of what to do if the same variable is substituted by two different terms.

An example of the differences between sequential and parallel substitutions is the following pattern: $(x \wedge y)\{x \mapsto y, y \mapsto a\}$. With parallel substitution, this would simplify to $y \wedge a$, as both variables are substituted at the same time. With the sequential version, the result would be either $a \wedge a$ or $y \wedge a$, depending on what order we choose. Ideally we would want $a \wedge a$, as applying the substitution to $y \wedge a$ would also produce that, suggesting that we did not fully apply the substitution.

However, during the unification procedure, this sort of problem should never actually surface. A variable that is on the left of a pair should not appear on the right side of any other pair, because **Occurs check** and **Elimination** will get rid of it, and the same variable should not be substituted twice, as **Elimination** and **Orient** can make them equal and because we are using sets, the duplicates will be

Delete	$P \cup \{(t, t)\} \Rightarrow P$
Decomposition	$P \cup \{(f t, g u)\} \Rightarrow P \cup \{(f, g), (t, u)\}$
Symbol clash L	$P \cup \{(f, t)\} \Rightarrow \perp$ if $t \neq f \wedge t \notin EV$
Symbol clash R	$P \cup \{(t, f)\} \Rightarrow \perp$ if $t \neq f \wedge t \notin EV$
Orient	$P \cup \{(t, x)\} \Rightarrow P \cup \{(x, t)\}$ if t is not a variable
Occurs check	$P \cup \{(x, t)\} \Rightarrow \perp$ if $x \in FV(t)$
Elimination	$P \cup \{(x, t)\} \Rightarrow P\{x \mapsto t\} \cup \{(x, t)\}$ if $x \notin FV(t)$

Table 3.1: The new rules for the unification algorithm.

erased. In the above example, we can use **Elimination** on the second term to get the substitution $\{(x, a), (y, a)\}$, which will always produce the desired result. If we instead had a substitution like $\{x \mapsto y, x \mapsto a\}$, then by using **Elimination** on the second one, we get $\{(a, y), (x, a)\}$, in which case we also need **Orient** on the first to reach the same result as before.

Because of this we decided to use a chain of sequential substitutions. This is defined as a fold over a list of pairs with the pattern as the base and the sequential substitution as the function, however, in most cases where this is used, the list is constant, so the fold simplifies neatly.

3.2 Going from polyadic to applicative

The first thing that needed to be modified about the original description is the rules of the unification algorithm. The ones described in Table 2.2 are in polyadic matching logic, where functions receive all of their arguments at once, but the formalization uses applicative matching logic, where function application is a binary operation.

It is for this reason that we decided to replace the two rules where this matters, **Decomposition** and **Symbol clash**. The main idea is that we try decomposing the two patterns one argument at a time, adding both sides of the application to the unification problem. Once we reach the leftmost application in the shorter pattern, there are 3 possibilities:

- We have reached the leftmost application in both patterns, meaning that both had the same number of arguments. In this case the two sides are added as normal. If the symbols were not the same, the offending pair will be caught by either of the next two cases. If they were the same, the pair may be deleted with the **Delete** rule.
- We reached the leftmost application in only the left pattern. In this case the left side is a symbol that is not the same as the right side, either because the right side is still an application or a different symbol, added by the above step. In this case we also need to make sure that the right side is not a free variable, as in the applicative case, that would also be possible. If both conditions hold, we can relate the unification problem with $\perp$, as the two terms are not unifiable.
- This case is similar to the above one, except we reach the leftmost application on the right side first. The reason we need this is because **Orient** only works if the right side is a free variable, however, we explicitly forbid that. Otherwise this rule is the same as **Orient** followed by the above one.

The new rules are described with mathematical notation in Table 3.1. Note how with this change all $\cup$ operations only do one or two singular inserts instead of an arbitrary amount.

3.3 Injectivity axiom

The paper and therefore this formalization depends on the axiom of injectivity, which in classical logic states that some function f is injective if

$$\forall x_1, \dots, x_n, y_1, \dots, y_n. f(x_1, \dots, x_n) = f(y_1, \dots, y_n) \rightarrow x_1 = y_1 \wedge \dots \wedge x_n = y_n$$

Simply postulating this axiom could cause inconsistency, although not in the way it is used in the paper. They provide a semantic proof of this fact. In this formalization it is simply treated as a Coq axiom, stating that the injectivity of functional patterns is a provable matching logic pattern, and used only for proving

the correctness of decomposition. Ideally this would be an object level axiom in the theory we are working under, however this requires a lot more machinery that is outside the scope of this thesis.

3.4 Unification problem

For defining the unification problem, we have decided to take a more generalized approach. First we have defined a typeclass which contains all the necessary members, functions, and properties that relate to the unification problem or are necessary for future proofs. The fields of the class are as follows:

- **insertUP**: This is the insert operation, used to add a pair to the unification problem in all of the rules. Recall that with the introduction of the applicative rules, every $\cup$ operation actually boiled down to one or two inserts. For this reason, we decided to use insert in the formalization instead of unions with singletons.
- **bottomUP**: This field requires the underlying type to have a special element that represents $\perp$.
- **toPredicateUP**: This is the ϕ^P function, that produces the predicate pattern from the unification problem.
- **substituteAllUP**: The substitution operation used in the **Elimination** rule.
- **singletonUP**: This creates a new unification problem consisting of a single pair. The algorithm starts with a problem created this way. Note that it would have also been correct to have a method for making an empty unification problem and then inserting into that would have been equivalent to this. However, an empty unification problem does not necessarily make sense in every context.

For example, the pattern $\phi^{(P \cup \{(x, t)\})\{y \mapsto u\}}$ would be expressed as `toPredicateUP (substituteAllUP y u (insertUP P (x, t)))`. For two non-unifiable patterns φ_1 and φ_2 we would say $\{(\varphi_1, \varphi_2)\} \Rightarrow^* \perp$, which could be expressed as `singletonUP  $\varphi_1$   $\varphi_2 \Rightarrow^*$  bottomUP`.

All of the above mentioned fields work with well-formed patterns (i.e. the dependent pairs described in Section 2.3). This is necessary for `toPredicateUP`, because this is the only way to return both the pattern and the well-formedness proof. It would have been possible to add an additional property stating that the pattern returned by this function is always well-formed, however, that would have been more difficult to prove and use later, especially with the mechanisms defined as part of this thesis to work with such pairs. All the other fields use them only for the sake of consistency and ease of use with the mentioned mechanisms, but it is worth noting that these would have worked equally well if they took patterns and proofs separately. The `UP` postfix stands for unification problem and is only added to avoid name collisions and make it easy to recognize when a term is constructing or eliminating a unification problem. The following are the additional properties that each instance of the class must satisfy, in mathematical notation and with the field names above, ignoring the well-formedness notations:

Let Γ be a theory, P a unification problem, t , t_1 , and t_2 terms, x a variable.

Field 1 (`toPredicateInsertUP`). $\Gamma \vdash \phi^{P \cup \{(t_1, t_2)\}} \leftrightarrow t_1 = t_2 \wedge \phi^P$

$\Gamma \vdash \text{toPredicateUP}(\text{insertUP } P (t_1, t_2)) \leftrightarrow ((t_1 = t_2) \wedge (\text{toPredicateUP } P))$

This is a simple consequence of the definition of ϕ^t , that will be required for most rules.

Field 2 (`toPredicateSubstituteAllUP`). $\Gamma \vdash \phi^{P[t/x]} \leftrightarrow (\phi^P)[t/x]$

$\Gamma \vdash \text{toPredicateUP}(\text{substituteAllUP } x t P) \leftrightarrow (\text{toPredicateUP } P)[t/x]$

This also follows from the definition of ϕ^t and substitution and will be required for **Elimination**.

Field 3 (`insertNotBottomUP`). $\Gamma \vdash P \neq \perp \rightarrow P \cup (t_1, t_2) \neq \perp$

$\Gamma \vdash P \neq \text{bottomUP} \rightarrow \text{insertUP } P (t_1, t_2) \neq \text{bottomUP}$

This expresses that we insert into a unification problem that is not bottom, the result will also not be bottom. When paired with the fact that we never try inserting into $\perp$, this just serves as a way to propagate that fact.

Field 4 (`toPredicateSingletonUP`). $\Gamma \vdash \phi^{\{(t_1, t_2)\}} \leftrightarrow (t_1 = t_2)$

$\Gamma \vdash \text{toPredicateUP}(\text{singletonUP } t_1 t_2) \leftrightarrow (t_1 = t_2)$

This is similar to `toPredicateUP`, but specialized to singletons. This would not be necessary if we had gone with the “empty and insert” approach mentioned above. This will be used in the proof of the lemma regarding $\Rightarrow^*$.

Instances of this class

An instance of this class has been created for sets containing pairs of well-formed patterns, wrapped in an `option`. `option A` for some type `A` is an inductive type that has two constructors, `None : option A`, and `Some : A -> option A`. This essentially supplements `A` with an additional value, `None`. This value is used as $\perp$. The rest of the methods are quite self-explanatory:

- All methods start with `option_map`, which takes a function and an `option` value as parameters, and if the latter is `None`, it leaves it as is, but if it is a `Some`, it applies the function to the value wrapped by `Some`.
- `insertUP` and `singletonUP` use the sets `insert` and `singleton` methods as the function
- `toPredicateUP` is using a fold, making the two sides of the pair equal and using a conjunction with the accumulator, with the base being the $\top$ pattern, which always holds and $\forall \varphi, \varphi \wedge \top \leftrightarrow \varphi$, therefore its presence does not cause any additional problems
- `substituteAllUP` uses a map, substituting every pattern in the set

We are using `stdpp`’s set classes [16], therefore this instance is still generic in the exact implementation of the set, as long as it has all required methods and properties. We also attempted a definition for lists, with the `insertUP` method being simply `cons`, however, that does not work most of the time because of the ordering. It is often necessary to access an element that is in the middle of the list, and without the ability to prove that reordering the list does not change the meaning of the unification problem, it is not possible to apply the rules. With an extra property that expresses that the list is unordered, it would be possible, however, `stdpp` includes a set implementation for such lists, which could be covered by a similar instance.

Technical note: The instance defined requires that set equivalence also implies definitional equality, which is obviously not true for lists, however, set equivalence should be enough, the proof would just be more difficult. We did not see a reason to generalize our definition that much, but that is a possible improvement.

It could also be possible to not wrap the type in `option`, however, in that case `bottomUP` would have to be the empty set which would not be technically correct, as an empty unification problem is not necessarily an error. However, if we shed the requirement that `insertUP` and the rules can only be used with non- $\perp$ unification problems, at least some of the current lemmas could be provable with such a malformed instance as well.

4 Proofs

In this chapter we present all lemmas from [10] that were only proven semantically and their new syntactical proofs. For the following lemmas, Γ will be a theory that contains the theory of definedness.

Theorem 8 (Proposition 3 in [10]). Let φ_1 and φ_2 be functional patterns. Then $\Gamma \vdash \varphi_1 \wedge \varphi_2 \leftrightarrow \varphi_1 \wedge (\varphi_1 = \varphi_2)$.

Proof. First we prove the $\leftarrow$ direction. Here we simply rewrite the goal using Corollary 1 on $\varphi_1 = \varphi_2$ to obtain $\varphi_1 \wedge \varphi_1$, which is easily solved by the φ_1 hypothesis.

For the $\rightarrow$ direction, we know from Theorem 2 that $\varphi \rightarrow [\varphi]$. We apply this to the $\varphi_1 \wedge \varphi_2$ hypothesis, that gives us a defined conjunction, which is the definition of $\in$, so our new hypothesis is $\varphi_1 \in \varphi_2$.

Then, we can discharge φ_1 by the hypothesis, and $\varphi_1 \in \varphi_2$ implies $\varphi_1 = \varphi_2$ by Theorem 3, so we can solve that part as well. $\square$

This lemma is used to set up the unification problem, as $\varphi_1 = \varphi_2$ is the predicate of the unification problem $\{(\varphi_1, \varphi_2)\}$.

Lemma 1. Let φ and t be patterns, and x be a variable. Then $\Gamma \vdash x = t \rightarrow \varphi[t/x] = \varphi$.

Proof. We use congruence theorem stated in Corollary 1 on $x = t$, using the pattern context $\varphi[\square/x] = \varphi$. We can assume that $\square$ is a fresh variable in φ . This gives us $\varphi[\square/x][x/\square] = \varphi \leftrightarrow \varphi[\square/x][t/\square] = \varphi$.

In $\varphi[\square/x][x/\square]$ we replace all occurrences of x with a fresh variable $\square$, and then immediately replace all occurrences of that with x . It is clear to see that this operation is the same as $\varphi[x/x]$, which actually does nothing and so the pattern is equivalent to just φ .

We can make a similar argument in the case of $\varphi[\square/x][t/\square]$, however, this time it is t that we place back into $\square$, so this pattern is equivalent to $\varphi[t/x]$.

Rewriting the formula with these two equivalences, we get $\varphi = \varphi \leftrightarrow \varphi[t/x] = \varphi$.

As equivalence is just the conjunction of two implications, we can drop the $\leftarrow$ direction, keeping only the $\rightarrow$ one.

Finally, $\varphi = \varphi$ is proven by reflexivity, so by modus ponens on the implication, we arrive at our goal of $\varphi[t/x] = \varphi$. $\square$

The above lemma makes it easier to simplify substitutions using congruence in future proofs.

Lemma 2. *Let φ be a pattern and σ a substitution. Then $\Gamma \vdash \varphi\sigma \wedge \phi^\sigma \leftrightarrow \varphi \wedge \phi^\sigma$.*

Proof. First we extract the ϕ^σ from both sides of the equivalence using Theorem 4, which yields us $\phi^\sigma \rightarrow \varphi\sigma \leftrightarrow \varphi$.

Then we do induction on σ with any φ . If it is the empty substitution, we can prove $\top \rightarrow \varphi \leftrightarrow \varphi$ by reflexivity. If it is not, then there is at least one x element variable and t pattern that is part of σ ($\sigma = \{x \mapsto t\} \cup \sigma'$), and we also know that $x = t$ is part of ϕ^σ ($\phi^\sigma = (x = t) \wedge \phi^{\sigma'}$). Thus our goal becomes $x = t \wedge \phi^{\sigma'} \rightarrow \varphi[t/x]\sigma' \leftrightarrow \varphi$.

If we specialize the induction hypothesis with $\varphi[t/x]$ as φ , we get $\phi^{\sigma'} \rightarrow \varphi[t/x]\sigma' \leftrightarrow \varphi[t/x]$.

$\phi^{\sigma'}$ is part of our hypothesis. Using the transitivity and symmetry of equivalence, we can reduce the goal to $\varphi[t/x] \leftrightarrow \varphi$, and we still have $x = t$ in the hypothesis.

We can use Lemma 1 with this to obtain $\varphi[t/x] = \varphi$.

Finally, using the deduction theorem, we can extract the equivalence from this equality. $\square$

This is, in a sense, an extension of Lemma 1 to set-based substitutions, with the added requirement for the predicate ϕ^σ .

Lemma 3. *Let P and P' be unification problems. If $P \Rightarrow P'$ and $P' \neq \perp$ then $\Gamma \vdash \phi^P \rightarrow \phi^{P'}$.*

Proof. We do induction on the $\Rightarrow$ relation. In most cases we will use Field 1, which states for every P unification problem and x, y terms that $\phi^{P \cup \{(x, y)\}} \leftrightarrow x = y \wedge \phi^P$.

Using this property in the **Delete** case, we get $t = t \wedge \phi^P \rightarrow \phi^P$ which is trivially proven.

In the case of **Decomposition**, we use this property three times, ultimately transforming the goal to $f t = g u \wedge \phi^P \rightarrow t = u \wedge f = g \wedge \phi^P$. This is the one time where we need to make use of the injectivity axiom, applying it to $f t = g u$ we obtain $f = g \wedge t = u$. With this now we can easily prove the goal.

With **Symbol clash L**, **Symbol clash R**, and later **Occurs check** we actually have a contradiction, as in both cases $P' = \perp$ and we have a hypothesis that says $P' \neq \perp$, therefore these cases are immediately disputed.

In the **Orient** case we use the property twice to transform the goal to $x = t \wedge \phi^P \rightarrow t = x \wedge \phi^P$ and this is solved using the symmetry of equality.

Finally, in the case of **Elimination** we will once again use this property twice, however, this time that will not be enough. We can reduce the goal to $x = t \wedge \phi^P \rightarrow x = t \wedge \phi^{P\{x \mapsto t\}}$ and the $x = t$ part is solvable like before but for the other one we need to take advantage of another property, Field 2. This says that $\phi^{P\{x \mapsto t\}} \leftrightarrow \phi^P[t/x]$. Rewriting with this, we get $x = t \wedge \phi^P \rightarrow x = t \wedge \phi^P[t/x]$. If we choose σ to be $\{(x, t)\}$ and φ to be ϕ^P , then this is a direct consequence of Lemma 2. $\square$

This lemma establishes the connection between unification problems and matching logic patterns.

Lemma 4. *If P and P' are unification problems, and $P \Rightarrow^* P'$, where $P' \neq \perp$, then $\Gamma \vdash \phi^P \rightarrow \phi^{P'}$.*

Proof. First we do induction on the $\Rightarrow^*$ relation.

In the reflexive case, we have $\phi^P \rightarrow \phi^P$, which is trivially provable.

In the transitive case, we have an inductive hypothesis that if $P' \neq \perp$, then $\phi^{P''} \rightarrow \phi^{P'}$, and from the hypothesis we know that $P' \neq \perp$, so it is enough to prove that $\phi^P \rightarrow \phi^{P''}$, using the fact that $P \Rightarrow P''$ and $P'' \Rightarrow^* P'$.

For this we are going to use Lemma 3 with $P \Rightarrow P''$, but in order to do that, we first need to prove that $P'' \neq \perp$.

For that we are going to do another induction, this time on $P'' \Rightarrow^* P'$. In the reflexive case, when $P'' = P'$, we can once again use the $P' \neq \perp$ hypothesis. In the transitive case we get that $P'' \Rightarrow P'''$, and we know that in every case of $\Rightarrow$, the left side is an insertion into some unification problem that is not $\perp$. That is, there is some P_0 , such that $P_0 \neq \perp$ and $P'' = P_0 \cup \{\dots\}$. But thanks to Field 3, that said $P_0 \neq \perp \rightarrow P_0 \cup \{\dots\} \neq \perp$, we know that P'' cannot be $\perp$.

With this condition satisfied, we can use Lemma 3 to finish the proof. $\square$

The above is actually a more general version of the fourth lemma in [10], which we treat as just a consequence of the above:

Corollary 3. If a substitution σ is the most general unifier of two patterns t_1 and t_2 , then $\Gamma \vdash t_1 = t_2 \rightarrow \phi^\sigma$.

Proof. Here we can use Theorem 7 with the most general unifier σ to obtain a P' unification problem that is not $\perp$ and $\phi^{P'} = \phi^\sigma$. We also know that $t_1 = t_2$ is the same as $\phi^{\{(t_1, t_2)\}}$, from Field 4. By rewriting with these equalities, we have transformed the goal to be solvable by the general version. $\square$

The general version of this lemma extends Lemma 3 to the reflexive-transitive closure of the relation, while the other version specializes it to the singleton pattern that the algorithm starts with.

Lemma 5. Let σ be a unifier of two terms t_1 and t_2 . Then $\Gamma \vdash \phi^\sigma \rightarrow t_1 = t_2$.

Proof. First we take the ϕ^σ condition and add it to both sides of the equality using Corollary 2, resulting in $t_1 \wedge \phi^\sigma = t_2 \wedge \phi^\sigma$.

Next, we use Lemma 2 on both sides of the equality, which gives us $t_1\sigma \wedge \phi^\sigma = t_2\sigma \wedge \phi^\sigma$.

Now, we can remove the ϕ^σ from both sides by using Corollary 2 again, and the resulting $t_1\sigma = t_2\sigma$ is exactly the definition of σ being a unifier of t_1 and t_2 , which is in our hypothesis. $\square$

This is the same as the above, but in the other direction.

Lemma 6. If a substitution σ is the most general unifier of two patterns t_1 and t_2 , then $\Gamma \vdash t_1 = t_2 \leftrightarrow \phi^\sigma$.

Proof. This is the conjunction of Corollary 3 and Lemma 5. $\square$

Lemma 4 and Lemma 5 can be combined together to make an equivalence that serves as the basis for the soundness theorem.

Theorem 9 (Soundness, Theorem 1 in [10]). Let σ be the most general unifier of two patterns t_1 and t_2 . Then $\Gamma \vdash t_1 \wedge t_2 \leftrightarrow t_1 \wedge \phi^\sigma$ and $\Gamma \vdash t_1 \wedge t_2 \leftrightarrow t_2 \wedge \phi^\sigma$.

Proof. From Theorem 8 we know that $t_1 \wedge t_2 \leftrightarrow t_1 \wedge t_1 = t_2$. Using the commutativity of $\wedge$ and symmetry of $=$, we can also get $t_1 \wedge t_2 \leftrightarrow t_2 \wedge t_1 = t_2$ from the same theorem.

We also know from Lemma 6 that $t_1 = t_2 \leftrightarrow \phi^\sigma$. Rewriting the previous two statements with this we get the two statements of this theorem. $\square$

Finally, we demonstrate the algorithm and its soundness on an example.

Example 2. We present the example shown in Example 1, generate its unifier and prove the first statement of Theorem 9 with its terms (the second may be proven similarly): $\exists \sigma. \Gamma \vdash \underbrace{f\ x\ (g\ 1)\ (g\ z)}_{t_1} \wedge \underbrace{f\ (g\ y)\ (g\ y)\ (g\ (g\ x))}_{t_2} \leftrightarrow \underbrace{f\ x\ (g\ 1)\ (g\ z)}_{t_1} \wedge \phi^\sigma$. We use $\exists \sigma$ because we know that the terms are unifiable and σ will be automatically inferred as we run the algorithm.

Proof. We begin with the $\rightarrow$ direction. First we use Theorem 8 to rewrite $t_1 \wedge t_2$ as $t_1 \wedge t_1 = t_2$.

Then, we use Lemma 4, since we do not have a proof that σ is the most general unifier (since we do not even know what σ is yet), we need to use the general version. For this, we need to prove that there is some non- $\perp$ P unification problem, such that $\{(t_1, t_2)\} \Rightarrow^* P$. We will also not state what P is ahead of time, as that too can be inferred.

This proof follows the steps outlined in Example 1, but this time we need to use the applicative version of the **Decomposition** rule. We use the transitive constructor of $\Rightarrow^*$ with each step, and when we reach the solved form, end with the reflexive constructor. The steps are as follows:

$$\begin{aligned}
& \{\underline{(f\ x\ (g\ 1)\ (g\ z),\ f\ (g\ y)\ (g\ y)\ (g\ (g\ x)))}\} \Rightarrow \text{Decomposition} \\
& \{\underline{(f\ x\ (g\ 1),\ f\ (g\ y)\ (g\ y))},\ (g\ z,\ g\ (g\ x))\} \Rightarrow \text{Decomposition} \\
& \{\underline{(f\ x,\ f\ (g\ y))},\ (g\ 1,\ g\ y),\ (g\ z,\ g\ (g\ x))\} \Rightarrow \text{Decomposition} \\
& \{\underline{(f,\ f)},\ (x,\ g\ y),\ (g\ 1,\ g\ y),\ (g\ z,\ g\ (g\ x))\} \Rightarrow \text{Delete} \\
& \{(x,\ g\ y),\ \underline{(g\ 1,\ g\ y)},\ (g\ z,\ g\ (g\ x))\} \Rightarrow \text{Decomposition} \\
& \{(x,\ g\ y),\ \underline{(g,\ g)},\ (1,\ y),\ (g\ z,\ g\ (g\ x))\} \Rightarrow \text{Delete} \\
& \{(x,\ g\ y),\ (1,\ y),\ \underline{(g\ z,\ g\ (g\ x))}\} \Rightarrow \text{Decomposition} \\
& \{(x,\ g\ y),\ (1,\ y),\ \underline{(g,\ g)},\ (z,\ g\ x)\} \Rightarrow \text{Delete} \\
& \{(x,\ g\ y),\ \underline{(1,\ y)},\ (z,\ g\ x)\} \Rightarrow \text{Orient} \\
& \{(x,\ g\ y),\ (y,\ 1),\ (z,\ g\ x)\}
\end{aligned}$$

We could have used **Elimination** at the end, however, it will not help us in this proof, so we omitted those steps from the original example.

Now that we have successfully inferred what P is, we can easily prove that it not $\perp$ and we have a hypothesis that says $t_1 = t_2 \rightarrow x = g\ y \wedge y = 1 \wedge z = g\ x \wedge \top$. We can apply this to our hypothesis of $t_1 = t_2$.

With this, we can actually already solve our goal. The t_1 part is trivial, and the ϕ^σ part is unknown, because σ is uninstantiated, however, because it is a predicate, we know that is a chain of conjunctions ending in $\top$. If we solve this with the one from above, it will be inferred that σ is indeed $\{(x \mapsto g\ y), (y \mapsto 1), (z \mapsto g\ x)\}$.

Now that we know this, the $\leftarrow$ direction can be solved. Here, the now known ϕ^σ gives us a series of equalities which we can use to repeatedly rewrite subterms in our goal of $t_1 \wedge t_2$ and the t_1 hypothesis, until the t_1 and t_2 in them become the same pattern. From this the solution is trivial.

We intuitively know that this rewriting is possible, as σ is a unifier of t_1 and t_2 , therefore $t_1\sigma = t_2\sigma$, and although we did not formally prove this, it is easy to manually check in this example. In this manual proof it is also not necessary to do the substitution fully (i.e. to calculate the actual $t_1\sigma$ and $t_2\sigma$), it is enough to go until the two terms match. $\square$

5 Conclusion

In this thesis we presented syntactical proofs for theorems and lemmas that were only proven semantically before in [10]. We not only showed that these proofs are possible to construct, but we have also provided a machine checked implementation [17] in Coq using an existing formalization [9]. With this it is now possible to use this algorithm in providing external, verified proofs for the $\mathbb{K}$ framework.

We have also explored possible ways to make the algorithm and its data structures more generic to facilitate wider usecases.

5.1 Discussion

Side conditions. A mayor challenge throughout this project has been maintaining the well-formedness side condition. This is an especially significant issue for the functions defined using `fold`s, for which we have had to define a separate induction principle. This states, that if some property P holds for the base value, some other property Q holds for every member of the list, and the combining function, given input values that satisfy these two, preserves P , then both directions of `fold` will preserve P as well. We mostly use `well_formed` for both P and Q . The Q property is not strictly necessary to prove this statement, but without it, it would be essentially useless, as the functions preserving property usually requires both. As an example, if our function is $\wedge$, then to prove that $\varphi_1 \wedge \varphi_2$ is well-formed, we need to know that both φ_1 and φ_2 are well-formed, knowing just one is not enough.

It also would have been enough to say that one argument of the function is an element of the list, and then use the context where this induction principle is used to get the proof for Q , but that would have been more complicated, and this is actually a more specific case of what we described, with Q being the “in list” relation, and the trivial proof for “every element of a list is in the list” being omitted.

Well-formed pattern. In Section 2.3 we describe an alternative notation for well-formed patterns using dependent pairs. This notation is not widely used in the existing formalization, but because of the unification problem class, it was necessary for us to use them and develop some machinery around it. This mostly consisted of creating wrappers for existing operators, such as disjunction, conjunction, implication, equality and substitutions. Proving the well-formedness of these patterns was really easy, as the proof was always bundled with the patterns. The big challenge with them was that the proof mode is only set up for use with regular patterns, and none of the existing lemmas used these pairs, so whenever we tried to use something like that, we had to recursively unwrap the patterns. This did create a lot of boilerplate, which could be abstracted away by defining tactics that unwrap patterns, and either enter proof mode or apply theorems, and rewrap the patterns, however, for this project it did not seem necessary to create these.

Alternative to the applicative rules. For the $\Rightarrow$ relation, instead of changing the rules to the applicative version, it would have also been possible to try and simulate polyadic matching logic, by defining a function which takes as arguments a function symbol and a list of patterns, and uses a left `fold` on the list with the symbol as the base and the binary application as the combining function to apply the patterns in the list to the function symbol to produce a single pattern. This would have been helpful in that if we have the function symbol and all of the arguments in a list, when we try unifying two terms of this form we can simply check the symbol and the lengths of the lists in both patterns, and if either of them does not match, we can reject the unification problem (by relating it with $\perp$) in a single step. If both of them match, then it is enough to pair up the arguments in the lists with the `zip` function, and add them all to the unification problem. However, this would have required an additional fold with the `insert` operation.

The trouble with this approach is that the use of so many folds introduces a lot of unnecessary complexity into the proofs as reasoning about folds, especially in the context of sets is difficult, while providing little to no additional benefits. Furthermore, as the formalization is of applicative matching logic, this would have been really out of place, both in terms of the definition and the usage.

Other possible instances of unification problem. In the case of the unification problem class, we have not investigated the possibility of instances which do not depend on some underlying container, though it might be possible that such instances exist. One example would be accumulating the pairs directly into a pattern and relying on the commutativity and associativity of the conjunction to reorder the terms as needed. This would of course have the same problems as a list would, but the interesting property of this instance would be that in the end, the pattern would be the predicate pattern itself without the need for folds or other modifications (i.e. `toPredicateUP` would be the identity function).

5.2 Future work

As mentioned above, the current implementation of the unification problem class requires that set equivalence is the same as definitional equality, which is a restriction that seems to be able to be relaxed and therefore the number of possible instances of that class could be increased. That would allow investigating a simpler instance using lists. Looking into what alternative instances are possible (such as the pattern based one described above) is also an interesting topic.

Upon describing and formalizing the unification problems as a class, we have noticed that they share a lot of properties and methods with substitutions, which were also described as sets of pairs in the original work. We theorize that it would be possible to define a similar generalized, container-independent class for substitutions as well. Even the method of applying it to a pattern might be able to become a function of the class, which could eliminate the need to commit to either a sequential or parallel method.

If this class existed, it could be possible to define a method in the unification problem to convert to an instance of the substitution class when in solved form, instead of relying on their predicate patterns being equal.

The thesis relies on two axioms, one for injectivity, and Theorem 7. The latter should be possible to formalize based on [15], while it should be possible to express the former in a better way, using a matching logic theory.

After starting work on this thesis, a new version of the unification paper [18] has been released which uses a new, $\exists$ -based notation for unifiers, and uses this to define soundness. That paper also includes a completeness theorem. It would be interesting to formalize the lemmas following that paper and compare the differences with this one.

There is a reverse algorithm for unification, called antiunification and a paper explaining a matching logic implementation for that also exists [19]. That also uses the new notation mentioned above. An implementation of that would nicely complement this work.

Acknowledgements

We would like to express our gratitude to Runtime Verification Inc. and the ADA Foundation for their financial support.

Bibliography

- [1] Xiaohong Chen, Dorel Lucanu, and Grigore Roşu. Matching logic explained. *Journal of Logical and Algebraic Methods in Programming*, 120, April 2021. ISSN 23522208. doi: 10.1016/j.jlamp.2021.100638. URL <https://linkinghub.elsevier.com/retrieve/pii/S2352220821000018>.
- [2] Chris Hathhorn, Chucky Ellison, and Grigore Roşu. Defining the undefinedness of C. *ACM SIGPLAN Notices*, 50(6):336–345, August 2015. ISSN 0362-1340, 1558-1160. doi: 10.1145/2813885.2737979. URL <https://dl.acm.org/doi/10.1145/2813885.2737979>.
- [3] 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. ACM, January 2015. ISBN 9781450333009. doi: 10.1145/2676726.2676982. URL <https://dl.acm.org/doi/10.1145/2676726.2676982>.
- [4] Dwight Guth. *A formal semantics of Python 3.3*. PhD thesis, University of Illinois at Urbana-Champaign, August 2013. URL <https://hdl.handle.net/2142/45275>.
- [5] Feng Wang, Fu Song, Min Zhang, Xiaoran Zhu, and Jun Zhang. KRust: A Formal Executable Semantics of Rust. 2018. doi: 10.48550/ARXIV.1804.10806. URL <https://arxiv.org/abs/1804.10806>.
- [6] 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. ACM, June 2019. ISBN 9781450367127. doi: 10.1145/3314221.3314601. URL <https://dl.acm.org/doi/10.1145/3314221.3314601>.

- [7] Everett Hildenbrandt, Manasvi Saxena, Nishant Rodrigues, Xiaoran Zhu, Philip Daian, Dwight Guth, Brandon Moore, Daejun Park, Yi Zhang, Andrei Stefanescu, and Grigore Rosu. KEVM: A Complete Formal Semantics of the Ethereum Virtual Machine. In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, pages 204–217. IEEE, July 2018. ISBN 9781538666807. doi: 10.1109/CSF.2018.00022. URL <https://ieeexplore.ieee.org/document/8429306/>.
- [8] 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. ACM, April 2021. ISBN 9781450383172. doi: 10.1145/3445814.3446751. URL <https://dl.acm.org/doi/10.1145/3445814.3446751>.
- [9] Release v1.0.15 - harp-project/AML-Formalization. URL <https://github.com/harp-project/AML-Formalization/releases/tag/v1.0.15>.
- [10] Andrei Arusoae and Dorel Lucanu. Unification in Matching Logic - Extended Version, November 2018. URL <http://arxiv.org/abs/1811.02835>.
- [11] Arthur Charguéraud. The Locally Nameless Representation. *Journal of Automated Reasoning*, 49(3):363–408, October 2012. ISSN 0168-7433, 1573-0670. doi: 10.1007/s10817-011-9225-2.
- [12] Xiaohong Chen and Grigore Rosu. Matching μ -Logic. In *2019 34th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–13, June 2019. ISBN 9781728136080. doi: 10.1109/LICS.2019.8785675. URL <https://ieeexplore.ieee.org/document/8785675/>.
- [13] Péter Berczky, Xiaohong Chen, Dániel Horpácsi, Lucas Peña, and Jan Tušil. Mechanizing Matching Logic In Coq. *Electronic Proceedings in Theoretical Computer Science*, 369:17–36, September 2022. ISSN 2075-2180. doi: 10.4204/EPTCS.369.2. URL <http://arxiv.org/abs/2201.05716>.

- [14] Jan Tušil, Péter Berczky, and Dániel Horpácsi. Interactive Matching Logic Proofs in Coq. In *Theoretical Aspects of Computing – ICTAC 2023*, volume 14446, pages 139–157. 2023. ISBN 9783031479625 9783031479632. URL https://link.springer.com/10.1007/978-3-031-47963-2_10.
- [15] Franz Baader and Wayne Snyder. Unification theory. *Handbook of automated reasoning*, 1:445–532, 2001.
- [16] stdpp.base. URL <https://plv.mpi-sws.org/coqdoc/stdpp/stdpp.base.html#lab26>.
- [17] Unification - Pull Request #431 - harp-project/AML-Formalization. URL <https://github.com/harp-project/AML-Formalization/pull/431>.
- [18] Andrei Arusoaie and Dorel Lucanu. Proof-carrying parameters in certified symbolic execution. *Logic Journal of the IGPL*, May 2023. ISSN 1367-0751, 1368-9894. doi: 10.1093/jigpal/jzad008. URL <https://academic.oup.com/jigpal/advance-article/doi/10.1093/jigpal/jzad008/7146273>.
- [19] Andrei Arusoaie and Dorel Lucanu. Proof-Carrying Parameters in Certified Symbolic Execution: The Case Study of Antiunification. *Electronic Proceedings in Theoretical Computer Science*, 369:1–16, September 2022. ISSN 2075-2180. doi: 10.4204/EPTCS.369.1. URL <http://arxiv.org/abs/2110.11700>.