Unification in Matching Logic Extended Version

Andrei Arusoaie and Dorel Lucanu

Alexandru Ioan Cuza University, Iaşi, Romania, {arusoaie.andrei,dlucanu}@info.uaic.ro

Abstract. Matching Logic is a framework for specifying programming language semantics and reasoning about programs. Its formulas are called patterns and are built with variables, symbols, connectives and quantifiers. A pattern is a combination of structural components (term patterns), which must be matched, and constraints (predicate patterns), which must be satisfied. Dealing with more than one structural component in a pattern could be cumbersome because it involves multiple matching operations. A source for getting patterns with many structural components is the conjunction of patterns. Here, we propose a method that uses a syntactic unification algorithm to transform conjunctions of structural patterns into equivalent patterns having only one structural component and some additional constraints. We prove the soundness of our approach, we discuss why the approach is not complete and we provide sound strategies to generate certificates for the equivalences, validated using Coq.

Keywords: Matching Logic \cdot Syntactic term unification \cdot Semantic unification \cdot Certification.

1 Introduction

Matching Logic [22] (hereafter shorthanded as ML) is a novel framework which is currently used for specifying programming languages semantics [11,12,19,8] and for reasoning about programs [23,10,9,26,27,14,5]. The logic is inspired from the domain of programming language semantics and it aims to use the operational semantics of a programming language as a basis for both *execution* and *verification* of programs.

On the program verification side, ML has some advantages over the existing program verification logics. The logic is *parametric* in the operational semantics of a language. One can *execute* the semantics against test suites and then use the *same* semantics for verification. Therefore, one can detect issues in the semantics at an early stage and fix them right away, thus, providing additional trust in the semantics. The proof system of ML is proved sound and (relatively) complete for *all* languages, unlike in the existing Floyd-Hoare logics, where the soundness of proof systems needs to be proved separately for each language. Moreover, ML eliminates the need to prove consistency relations between the operational

semantics (used for execution) and the axiomatic semantics (used for verification) as it is often the case when using the traditional approaches.

The ML formulas, called *patterns*, are built using variables, symbols, connectives and quantifiers. A pattern is evaluated to the set of values that *matches* it. ML makes no distinction between function symbols and predicate symbols. Not having this distinction increases the expressivity of the language, where various notions (e.g., *function*, *equality*) can be specified using symbols that satisfy some axioms.

An example of such a ML formula is φ_1 below: it matches over the set of lists that start at address p+2 and store the sequence a which contains an even number on the third position:

$$\varphi_1 \triangleq \mathtt{list}(p+2,a) \land \exists k.(select\ a\ 3) = 2*k$$

Basically, the novelty in ML w.r.t. first-order logics is that structural components are formulas as well. In our example, φ_1 is a conjunction of a structural component $\mathtt{list}(p+2,a)$ – that is, a list that starts at address p+2 which stores a sequence implemented as an array (encoded using the select-store axioms), – and a constraint $\exists k.(select\ a\ 3) = 2*k$. In ML, the structural components are called $term\ patterns$, whereas the constraints are called $predicates\ patterns$.

The conjunction of two ML patterns may produce a new pattern with more than one structural component, as shown here:

$$\underbrace{\underbrace{\mathtt{list}(p+2,a)}_{\varphi_1} \wedge \underbrace{\exists k.(select\ a\ 3) = 2*k}_{\varphi_2} \wedge \underbrace{\underbrace{\mathtt{list}(q,(store\ b\ 3\ y))}_{\varphi_2} \wedge \underbrace{\underbrace{}_{\varphi_2}^{\text{constraint}}}_{}$$

Finding a set of elements that matches the conjunction $\varphi_1 \wedge \varphi_2$ is not necessarily an easy task mainly because both structural components (list(p + 2, a)) and (list($q, (store \ b \ 3 \ y)$)) need to be matched simultaneously. In theory, this set is the intersection of the sets that match φ_1 and φ_2 independently.

In practice, dealing with multiple structural components in one formula is cumbersome. Reasoning with such formulas is a burden for larger formulas. Also, when mixing multiple structural components in one formula we lose the separation between structure and constraint. This separation is essential when implementing a ML prover, where the constraints can be handled separately using existing SMT solvers. In our examples above, the constraints of both φ_1 and φ_2 can be dealt with using existing SMT solvers like Z3 [18] or CVC4 [7] since they provide theories for handling arrays and quantifiers. A more convenient approach would be to work with formulas that have only one structural component.

In ML, the semantics of $\varphi_1 \wedge \varphi_2$ is the largest set of elements matching φ_1 and φ_2 . Thus, the conjunction of two patterns can be seen as a semantic unification of the two patterns. So, it makes sense to relate syntactic unification to this notion of semantic unification [22]. Let us consider the particular case when $\varphi_i \triangleq t_i \wedge \phi_i$,

where t_i is a term pattern and ϕ_i is a predicate pattern, $i \in \{1, 2\}$. In this case:¹

$$\varphi_1 \wedge \varphi_2 = t_1 \wedge \phi_1 \wedge t_2 \wedge \phi_2 = t_1 \wedge t_2 \wedge \phi_1 \wedge \phi_2 = t_1 \wedge (t_1 = t_2) \wedge \phi_1 \wedge \phi_2$$

The predicate patterns expressing the equality of two term patterns $t_1 = t_2$ cannot be handled, e.g., by SMT solvers. Therefore, it would be more convenient to reduce it to a simpler equivalent predicate $\phi^{t_1=t_2}$, which can be handled using external provers. In addition, it would be worth to produce a formal proof of the equivalence between $t_1 = t_2$ and $\phi^{t_1=t_2}$.

At a first sight, unification of terms seems to be useful here. If σ is the most general unifier of t_1 and t_2 , seen as first-order terms, then $t_1\sigma=t_2\sigma$. Unifiers are substitutions, and substitutions can be transformed into ML formulas [4].

In our list example, $\mathtt{list}(p+2,a)$ and $\mathtt{list}(q,(store\ b\ 3\ y))$ have $\sigma=\{q\mapsto p+2,a\mapsto(store\ b\ 3\ y)\}$ as the most general unifier. Translating σ to a formula results in $\phi^\sigma\triangleq(q=p+2)\land(a=(store\ b\ 3\ y))$. For this particular case, the term pattern equality $\mathtt{list}(p+2,a)=\mathtt{list}(q,(store\ b\ 3\ y))$ is equivalent to ϕ^σ . Moreover, the semantic unifier $\varphi_1\land\varphi_2$ is also equivalent to $\mathtt{list}(p+2,a)\land\phi^\sigma\land(\exists k.(select\ a\ 3)=2*k)\land y>0$. This form is now convenient since it has only one structural component and a constraint manageable by an SMT solver.

Contributions. We show that $\phi^{t_1=t_2}$ can be obtained using the most general unifier σ of t_1 and t_2 , whenever it exists. The proof of the equivalence between $t_1=t_2$ and ϕ^{σ} is not trivial and, surprisingly, it depends on the algorithm used to compute the most general unifier. Our proof uses the syntactic unification algorithm proposed by Martelli and Montanari [16]. Since the equivalence is proved only for the case when the most general unifier exists, we say that this algorithm is sound for semantic unification in ML.

Unfortunately, this algorithm is not *complete* for semantic unification: if the terms t_1 and t_2 are not syntactically unifiable, then there are no guarantees that $t_1 \wedge t_2$ is a "contradiction" in ML. We present a detailed analysis of this aspect and we provide a counterexample.

Finally, a provableness property of the Martelli-Montanari unification algorithm is shown: we provide a sound strategy to generate a proof certificate of the equivalence between $t_1 \wedge t_2$ and $t_1 \wedge \phi^{\sigma}$ with σ the most general unifier of t_1 and t_2 . This proof uses the rules of the ML proof system [22], and the main idea is to transform the steps of the unification algorithm into sequences of proof steps. The proposed approach is validated by a Coq encoding, which mechanically checks the correctness of the applied strategy.

All these contributions explicitly establish the relationship between syntactic unification and semantic unification in ML, as summarised be the next table:

¹ For the sake of presentation, we assume here that all patterns have the same sort. Also, the last equality in the sequence $t_1 \wedge t_2 \wedge \phi_1 \wedge \phi_2 = t_1 \wedge (t_1 = t_2) \wedge \phi_1 \wedge \phi_2$ holds because of a lemma which is presented in the technical section of the paper.

Syntactic term unification	Semantic unification in ML	Where to find it
unification of t_1 and t_2	$t_1 \wedge t_2$	=
substitution σ	ϕ^{σ}	Definition 12
$\sigma(t_1) = \sigma(t_2)$	$\phi^{\sigma} \to t_1 = t_2$	Lemma 5
$\sigma = mgu(t_1, t_2)$	$t_1 \wedge t_2 = t_1 \wedge \phi^{\sigma} = t_2 \wedge \phi^{\sigma}$	Theorem 1
syntactic unification algorithm	proof certificates	Section 4

Paper organisation. In Section 2.1 we recall the main notions and notations from the unification theory that we use in this paper. Section 2.2 includes a concise presentation of Matching Logic based on [22]. In Section 3 we show how to find the convenient representation of our semantic unifiers using the syntactic unification algorithm. We prove that the unification algorithm is sound for semantic unification and we discuss why this algorithm is not complete for semantic unification. In Section 4 we describe sound strategies for generating proofs that can be further used to generate proof certificates.

2 Preliminaries

2.1 Syntactic Unification

We recall from [6] the notions related to unification that we use in this paper. We also recall the algorithm for finding the *most general unifier* presented in [16].

Let S be a set of sorts. We consider a (countably) infinite S-indexed set of variables Var and a signature, i.e., a (finite or countably infinite) S-indexed set of function symbols, Σ . By T_{Σ} we denote the algebra of ground terms and by $T_{\Sigma}(Var)$ the corresponding term algebra generated by Σ . To keep the presentation simple (as in [6]) we do not explicitly show the sorts of the terms unless they cannot be inferred from context. This does not restrict in any way the generality and will be handled properly when transferring all these to Matching Logic.

We use the typical conventions and notations. Letters x, y, z denote variables and c, f, g denote symbols. Terms are either variables or compound terms of the form $f(t_1, \ldots, t_n)$; $f \in \Sigma_{s_1 \ldots s_n, s}$ means that f has $arity \ s_1 \ldots s_n, s$, that is, for each $i = \overline{1, n}$, the subterm t_i is of sort s_i and the sort of $f(t_1, \ldots, t_n)$ is s. If n = 0 then f is a constant and the term f() is simply denoted by f. By var(t) we denote the set of variables occurring in a term t. Substitutions are denoted by symbols σ, η, θ or directly as a set of bindings $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$. We use ι to denote the identity substitution. The application of a substitution σ to a term t is denoted $t\sigma^2$. The composition of substitutions σ and η is denoted as $\sigma\eta$. If $\sigma = \{x \mapsto y, y \mapsto y, z \mapsto 4\}$ and $\eta = \{y \mapsto 3\}$ then $\sigma\eta = \{x \mapsto y, y \mapsto y, z \mapsto 4\}\eta = \{x \mapsto (y\eta), y \mapsto (y\eta), z \mapsto (4\eta)\} = \{x \mapsto 3, y \mapsto 3, z \mapsto 4\}$. Two substitutions σ and η are equal, written $\sigma = \eta$, if they are extensionally

² Although substitutions are defined only over a set of variables, it is well-known that they can be extended to terms. Also, if a substitution σ is not defined for a variable, say x, then we consider $\sigma(x) = \iota(x) = x$.

equal: $x\sigma = x\eta$ for every variable x. A substitution σ is more general than a substitution η , written as $\sigma \leq \eta$, if there is a substitution θ such that $\sigma\theta = \eta$.

Example 1. Let us consider a sort s and a signature Σ that includes the symbols f,g,c, where $f,g\in \Sigma_{ss,s}$ and $c\in \Sigma_{,s}$. Then $g(c,c)\in T_{\Sigma}$ is a ground term, $f(g(x,c),y)\in T_{\Sigma}(Var)$ is a term with variables and $var(f(g(x,c),y))=\{x,y\}\subseteq Var$. A substitution $\eta=\{x\mapsto c,y\mapsto g(x,z)\}$ applied to f(g(x,a),y) produces $(f(g(x,c),y))\eta=f(g(c,c),g(x,z))$. If $\sigma=\{x\mapsto x',y\mapsto g(x,z)\}$ then $\sigma\leq \eta$, because there is $\theta=\{x'\mapsto c\}$ such that $\sigma\theta=\eta$.

Definition 1 (Unifier, Most General Unifier). A substitution σ is a unifier of two terms t and t' if $t\sigma = t'\sigma$. A unifier σ is the most general unifier (hereafter shorthanded as mgu) if for every unifier σ' of t and t' we have $\sigma \leq \sigma'$.

Example 2. If $t \triangleq f(g(x,c),y)$ and $t' \triangleq f(z,y')$ are terms then $\sigma = \{z \mapsto g(x,c), y \mapsto y'\}$ is a unifier of t and $t' : t\sigma = f(g(x,c),y') = t'\sigma$.

Whenever there exists a unifier for two given terms we say that the terms are unifiable. It is not always the case that, given two terms, we can find unifiers for them. For example, recall t from Example 2 and consider $t'' \triangleq g(g(x,c),y)$. Then t and t'' are not unifiable because it is impossible to find a substitution σ such that $t\sigma = t''\sigma$. In the particular context of syntactic unification, for every two unifiable terms there exists a $most\ general\ unifier$.

Definition 2 (Unification problem, Solution, Solved form). An unification problem P is either a set of pairs of terms $\{t_1 = t'_1, \ldots, t_n = t'_n\}$ or a special symbol \bot . A substitution σ is a solution of a unification problem $P = \{t_1 = t'_1, \ldots, t_n = t'_n\}$ if σ is a unifier of t_i and t'_i , for every $i = \overline{1, n}$. A unification problem P is in solved form if $P = \bot$ or $P = \{x_1 = t'_1, \ldots, x_n = t'_n\}$ with $x_i \notin var(t_i)$ for all $i, j = \overline{1, n}$.

Let $unifiers(P) = \{ \sigma \mid \sigma \text{ is a solution of } P \}$ denote the set of solutions of P. If $P = \bot$ then $unifiers(P) = \emptyset$. Each unification problem $P = \{x_1 = t'_1, \dots, x_n = t'_n\}$ in solved form defines a substitution $\sigma_P = \{x_1 \mapsto t'_1, \dots, x_n \mapsto t'_n\}$.

Among the well-known algorithms for finding the most general unifier we encounter the unification by recursive descent [21], and a rule-based approach for finding the mgu [13,16]. The latter is presented in Figure 1 and it consists of a set of transformation rules of the form $P \Rightarrow P'$ applied over unification problems P and P'.

Remark 1. We recall from [6] the main properties of the unification algorithm in Figure 1. 3 If P as a unification problem then:

- 1. Progress: If P is not in solved form, then there exists P' such that $P \Rightarrow P'$.
- 2. Solution preservation: If $P \Rightarrow P'$ then unifiers (P) = unifiers(P').
- 3. Termination: There is no infinite sequence $P \Rightarrow P_1 \Rightarrow P_2 \Rightarrow \cdots$.

³ It is not the purpose of this paper to prove these results. The interested reader is referred to [6] for complete proofs and details.

```
\begin{array}{ll} \textbf{Delete:} & P \cup \{t \stackrel{.}{=} t\} \Rightarrow P \\ \textbf{Decomposition:} & P \cup \{f(t_1, \dots, t_n) \stackrel{.}{=} f(t'_1, \dots, t'_n)\} \Rightarrow P \cup \{t_1 \stackrel{.}{=} t'_1, \dots, t_n \stackrel{.}{=} t'_n\} \\ \textbf{Symbol clash:} & P \cup \{f(t_1, \dots, t_n) \stackrel{.}{=} g(t'_1, \dots, t'_n)\} \Rightarrow \bot \\ \textbf{Orient:} & P \cup \{f(t_1, \dots, t_n) \stackrel{.}{=} x\} \Rightarrow P \cup \{x \stackrel{.}{=} f(t_1, \dots, t_n)\} \\ \textbf{Occurs check:} & P \cup \{x \stackrel{.}{=} f(t_1, \dots, t_n)\} \Rightarrow \bot, \text{ if } x \in var(f(t_1, \dots, t_n)) \\ \textbf{Elimination:} & P \cup \{x \stackrel{.}{=} t\} \Rightarrow P\{x \mapsto t\} \cup \{x \stackrel{.}{=} t\} \text{ if } x \not\in var(t), x \in var(P) \\ \end{array}
```

Fig. 1. A rule-based algorithm for syntactic unification

4. Most general unifier: If θ is a solution for P, then for any maximal sequence of transformations $P \Rightarrow^! P'$ either P' is \bot or $\sigma_{P'} \leq \theta$. If there is no solution for P then P' is \bot .

The properties listed in Remark 1 essentially say that the algorithm in Figure 1 produces the most general unifier when it exists. Note that this algorithm does not impose any strategy to apply the rules.

Example 3. Recall $t \triangleq f(g(x,c),y)$ and $t' \triangleq f(z,y')$ from Example 2. Consider the unification problem $P = \{t = t'\}$. Using the unification algorithm we obtain:

$$\begin{array}{ll} P = \{t \stackrel{.}{=} t'\} \triangleq \{f(g(x,c),y) \stackrel{.}{=} f(z,y')\} & \Rightarrow & \textbf{(Decomposition)} \\ \{g(x,c) \stackrel{.}{=} z, y \stackrel{.}{=} y'\} & \Rightarrow & \textbf{(Orient)} \\ \{z \stackrel{.}{=} g(x,c), y \stackrel{.}{=} y'\} & \triangleq P' \end{array}$$

The obtained unification problem P' is in solved form; the corresponding substitution $\sigma_{P'} = \{z \mapsto g(x,c), y \mapsto y'\}$ is the most general unifier of t and t'.

When it exists, the most general unifier is not unique. By composition with renaming substitutions we can generate an infinite set of mgus. In general, we say that mgus are unique up to a composition with a renaming substitution.

2.2 Matching Logic

Matching Logic [22,24] started as a logic over a particular case of constrained terms [23,26,9,25,27,5,14], but now it is developed as a solid program logic framework. Here we recall from [22] the particular definitions and notions of ML that we use in this paper. This subsection is longer than an usual one for preliminaries. Since Matching Logic is a quite recent research contribution including new atypical concepts and results, we decided to present it with more details and examples. This makes the paper self-content.

ML formulas are defined over a many-sorted signature (S, Σ) , where Σ is a $S^* \times S$ -indexed set of symbols. The formulas in ML are patterns:

Definition 3 (ML Formula). A pattern Σ -pattern φ_s of sort s is defined by:

$$\varphi_s ::= x_s \mid f(\varphi_{s_1}, \dots, \varphi_{s_n}) \mid \neg \varphi_s \mid \varphi_s \wedge \varphi_s \mid \exists x. \varphi_s$$

where x_s ranges over the variables of sort s ($x_s \in \mathcal{X}_s \subseteq Var_s$), f ranges over $\Sigma_{s_1...s_n,s}$, and x ranges over the set of variables (of any sort).

The derived patterns are defined as expected: $\exists x.x \ (x \text{ of sort } s), \perp_s \triangleq \neg \exists^4, \varphi_1 \lor \varphi_2 \triangleq \neg (\neg \varphi_1 \land \neg \varphi_2), \varphi_1 \rightarrow \varphi_2 \triangleq \neg \varphi_1 \lor \varphi_2, \varphi_1 \leftrightarrow \varphi_2 \triangleq (\varphi_1 \rightarrow \varphi_2) \land (\varphi_2 \rightarrow \varphi_1).$

Example 4. Let Nat be a sort and Σ a signature which includes symbols $o \in \Sigma_{Nat}$ and $succ \in \Sigma_{Nat,Nat}$. Then, $o, succ(o), succ(x), o \land succ(o), \neg(o \land succ(o)), o \lor \exists x. succ(x)$ are all ML patterns.

When sorts are not relevant or can be inferred from the context we drop the sort subscript $(\varphi_s \text{ becomes } \varphi)$.

Definition 4 (ML model). A ML model Σ -model M consist of:

- S-sorted sets $\{M_s\}$ for each $s \in S$, where M_s is the carrier of sort s of M;
- a function $f_M: M_{s_1} \times \cdots \times M_{s_n} \to \mathcal{P}(M_s)$ (note the use of the powerset $\mathcal{P}(M_s)$ as the co-domain) for each symbol $f \in \Sigma_{s_1...s_n,s}$.

Example 5. Recall the signature Σ from Example 4. A possible Σ -model M includes a set $M_{Nat} = \mathbb{N}$, a constant function o_M which evaluates to the singleton set $\{0\}$, and a function $succ_M : \mathbb{N} \to \mathcal{P}(\mathbb{N})$ which returns a singleton set containing the successor of the given natural number. Here, the interpretation functions have only singleton sets as results. This is not always the case. Let us enrich Σ with a new symbol $\leq \in \Sigma_{NatNat,Nat}$. We can choose the following interpretation \leq_M function for the \leq symbol: $\leq_M : \mathbb{N} \times \mathbb{N} \to \mathcal{P}(\mathbb{N})$, such that $\leq_M (x,y) = \mathbb{N}$ if x is less or equal than y, and $\leq_M (x,y) = \emptyset$ otherwise.

The meaning of patterns is given by using valuations ρ as in first-order logic, but the result of the interpretation is a set of elements that the pattern "matches", similar to the worlds in modal logic.

Definition 5 (M-valuations). If $\rho: \mathcal{X} \to M$ is a variable valuation and φ a pattern, then the extension of ρ to patterns $\overline{\rho}(\varphi)$ is inductively defined as follows:

```
1. \overline{\rho}(x) = {\rho(x)};
```

- 2. $\overline{\rho}(f(\varphi_1,\ldots,\varphi_n)) = \bigcup \{f_M(v_1,\ldots,v_n) \mid v_i \in \overline{\rho}(\varphi_i), i=1,\ldots,n\};$
- 3. $\overline{\rho}(\neg \varphi) = M_s \setminus \overline{\rho}(\varphi)$, where the sort of φ is s;
- 4. $\overline{\rho}(\varphi_1 \wedge \varphi_2) = \overline{\rho}(\varphi_1) \cap \overline{\rho}(\varphi_2)$, where φ_1 and φ_2 have the same sort;
- 5. $\overline{\rho}(\exists x.\varphi) = \bigcup_{v \in M_s} \overline{\rho}[v/x](\varphi)$, where $x \in \mathcal{X}_s$ and $\overline{\rho}[v/x]$ is the valuation ρ' s.t. $\rho'(y) = \rho(y)$ for all $y \neq x$, and $\rho'(x) = v$.

When a functional symbol is a constant c (case 2 in Def. 5) we let $\overline{\rho}(c) = c_M$. Additional constructs can be handled similarly (e.g. $\overline{\rho}(\varphi_1 \vee \varphi_2) = \overline{\rho}(\varphi_1) \cup \overline{\rho}(\varphi_2)$).

Example 6. Recall the signature Σ from Example 4 and the model M from Example 5. Also, consider a valuation $\rho: \mathcal{X} \to M$ such that $\rho(x) = 0$. The pattern succ(x) matches over $\{1\}$ because $\overline{\rho}(succ(x)) = succ_M(\rho(x)) = succ_M(0) = \{1\}$.

An interesting pattern is $o \vee \exists x.succ(x)$ since it matches over the entire set \mathbb{N} . Indeed, if we consider any valuation $\varrho : \mathcal{X} \to M_{Nat}$, then $\overline{\varrho}(o \vee \exists x.succ(x)) = \{0\} \cup \bigcup_{n \in \mathbb{N}} \overline{\varrho[n/x]}(succ(x)) = \{0\} \cup \bigcup_{n \in \mathbb{N}} succ_M(n) = M_{Nat} = \mathbb{N}$.

⁴ Note that \perp is different from the (bold) symbol \perp used in Section 3.

A particular type of patterns are M-predicates. These are meant to capture the usual meaning of predicates, i.e., patterns that can be either true or false.

Definition 6 (M-predicates). The pattern φ_s is an M-predicate iff for any valuation $\rho: \mathcal{X} \to M$, $\overline{\rho}(\varphi_s)$ is either M_s or \emptyset . Also, φ_s is called a predicate iff it is a M-predicate in all models M.

Example 7. The pattern $o \vee \exists x.succ(x)$ (from Example 6) is an M-predicate because for all $\varrho : \mathcal{X} \to M$ we have $\overline{\varrho}(o \vee \exists x.succ(x)) = M_{Nat} = \mathbb{N}$.

The pattern $o \wedge succ(o)$ is also an M-predicate because $\overline{\varrho}(o \wedge succ(o)) = \overline{\varrho}(o) \cap \overline{\varrho}(succ(o)) = \{0\} \cap \{succ_M(0)\} = \emptyset$.

Definition 7 (Satisfaction relation, validity). A model M satisfies φ , written $M \models \varphi_s$, if $M_s = \overline{\rho}(\varphi_s)$ for each variable valuation ρ . A pattern φ is valid (written $\models \varphi$) iff $M \models \varphi$ for all models M.

Example 8. Recall the model M from Example 5. $M \models o \lor \exists x.succ(x)$ since, for all $\rho : \mathcal{X} \to M$ we have $\overline{\rho}(o \lor \exists x.succ(x)) = M_{Nat}$.

Proposition 1 (Proposition 2.6 in [22]). Let φ_1 and φ_2 be two ML formulas and M a ML model. Then:

```
-M \models \varphi_1 \to \varphi_2 \text{ iff } \overline{\rho}(\varphi_1) \subseteq \overline{\rho}(\varphi_2) \text{ for all } \rho : Var \to M.
- M \models \varphi_1 \leftrightarrow \varphi_2 \text{ iff } \overline{\rho}(\varphi_1) = \overline{\rho}(\varphi_2) \text{ for all } \rho : Var \to M.
```

Definition 8 (ML specifications). A matching logic specification is a triple (S, Σ, F) , where F contains Σ -patterns. The Σ -patterns in F are axiom patterns. We say that φ is a semantical consequence of F, written $F \models \varphi$, iff $M \models F$ implies $M \models \varphi$, for each Σ -model M.

An important ingredient of ML is the definedness symbol $\lceil _ \rceil_{s_1}^{s_2} \in \Sigma_{s_1,s_2}$, with the following intuitive meaning: if φ is matched by some values of sort s_1 then $\lceil \varphi \rceil_{s_1}^{s_2}$ is $\lceil s_2 \rceil$, otherwise it is $\rfloor s_2 \rceil$. This interpretation is enforced by including the axiom pattern $\lceil x \rceil_{s_1}^{s_2}$ in the set of axioms F. This symbol and its associated pattern are used to define:

- conjunction of patterns with different sorts: for instance, if the symbol $\leq_b \in \Sigma_{Nat\ Nat\ Bool}$, then the pattern $x \wedge o \leq_b x$ is not syntactically correct, because x has sort Nat whereas $o \leq_b x$ has sort Bool. Using definedness we can now write a syntactically correct formula $x \wedge \lceil o \leq_b x \rceil_{Bool}^{Nat}$;
- membership pattern: $x \in_{s_1}^{s_2} \varphi \triangleq [x \land \varphi]_{s_1}^{s_2}$ with $x \in Var_{s_1}$, where x is another pattern that evaluates to a single value;
- equality pattern: $\varphi = \stackrel{s_2}{s_1} \varphi' \triangleq \neg \lceil \neg (\varphi \leftrightarrow \varphi') \rceil \stackrel{s_2}{s_1}$.

In ML there is no distinction between function and predicate symbols. However, there is a way to specify that certain symbols are interpreted as functions. These symbols are called *functional* symbols.

Definition 9 (Functional patterns). A pattern φ is functional in a model M iff $|\overline{\rho}(\varphi)| = 1$ for any valuation $\rho: Var \to M$. The pattern φ is functional in F iff it is functional in all models M such that $M \models F$.

Remark 2. In [22] (more precisely, Proposition 5.17 in [22]) it is shown that functional patterns are interpreted as total functions in models, and their interpretation contains precisely one element. Moreover, given a ML specification (S, Σ, F) , a pattern φ is functional in all (S, Σ, F) -models iff $F \models \exists y. (\varphi = y)$.

Example 9. Recall the o and succ(x) patterns from Example 4. Both patterns are functional in M_{Nat} (from Example 5) since they are interpreted as functions (i.e., o_M and $succ_M$) which return a singleton set. If we want to have functional interpretation for o and succ in all models, then we have to add to F the axioms $\exists y.(c = y)$ and $\exists y.(succ(x) = y)$.

The following technical result was proved in [22] and establishes the link between equivalence and equality of functional patterns:

Proposition 2 (Proposition 5.9 in [22]). If φ , φ' are patterns of sort s_1 then:

```
-\overline{\rho}(\varphi =_{s_1}^{s_2} \varphi') = \emptyset \text{ iff } \overline{\rho}(\varphi) \neq \overline{\rho}(\varphi'), \text{ for any } \rho : Var \to M.
-\overline{\rho}(\varphi =_{s_1}^{s_2} \varphi') = M_{s_2} \text{ iff } \overline{\rho}(\varphi) = \overline{\rho}(\varphi'), \text{ for any } \rho : Var \to M.
-M \models \varphi =_{s_1}^{s_2} \varphi' \text{ iff } M \models \varphi \leftrightarrow \varphi', \text{ for any model } M.
- \models \varphi =_{s_1}^{s_2} \varphi' \text{ iff } \models \varphi \leftrightarrow \varphi'.
```

It is worth noting that the Proposition 2 holds only for functional patterns. When functional patterns have the same sort, the proposition below holds:

Proposition 3 (Proposition 5.24 in [22]). *If* φ *and* φ' *are two functional patterns of the same sort then* $\models (\varphi \land \varphi') = \varphi \land (\varphi = \varphi')$.

Definition 10 (Term patterns). If $f \in \Sigma_{s_1,...,s_n,s}$ is a symbol such that F contains the pattern $\exists y. (f(x_1,...,x_n)=y)$ then f is a functional symbol. Term patterns are formulas containing only functional symbols.

Example 10. If 1, f, g are symbols in Σ and x and z are variables in Var, then $t \triangleq f(x, g(1), g(z))$ is a term pattern if $\exists y. f(x_1, x_2) = y$, $\exists y. g(x_1) = y$, and $\exists y. 1 = y$ are semantical consequences of the axioms F.

Substitution. Sometimes we need to use substitution over ML patterns directly. We use $\varphi[\varphi'/x]$ to denote the formula obtained by substituting φ' for variable x in φ (we assume φ' and x have the same sort):

```
1. x[\varphi'/x] = \varphi'; \quad y[\varphi'/x] = y \text{ when } x \neq y.

2. f(\varphi_1, \dots, \varphi_n)[\varphi'/x] = f(\varphi_1[\varphi'/x], \dots, \varphi_n[\varphi'/x])

3. (\neg \varphi)[\varphi'/x] = \neg(\varphi[\varphi'/x])

4. (\varphi_1 \wedge \varphi_2)[\varphi'/x] = \varphi_1[\varphi'/x] \wedge \varphi_2[\varphi'/x]
```

5. $(\exists y.\varphi)[\varphi'/x] = \exists y.\varphi[\varphi'/x]$, if $y \notin var(\varphi')$; otherwise, a renaming is required.

Our main result use the following technical lemma. For the particular case when the equivalence and the equality are the same it is a consequence of Proposition 5.10 from [22]. We include its proof here as an example of Matching Logic reasoning.

Lemma 1. If φ is a pattern, t is a term pattern, and x is a variable such that $F \models x = t$, then $F \models \varphi[t/x] = \varphi$.

Proof. By induction on φ , we show: for all M and $\rho: \mathcal{X} \to M$, $\overline{\rho}(\varphi[t/x]) = \overline{\rho}(\varphi)$:

- $-\overline{\rho}(x[t/x]) = \rho(t) = \rho(x)$, which (by Proposition 2) holds since x = t;
- $-\overline{\rho}(y[t/x]) = \rho(y)$ when $x \neq y$;
- $-\overline{\rho}(f(\varphi_1,\ldots,\varphi_n)[t/x]) = \overline{\rho}(f(\varphi_1[t/x],\ldots,\varphi_n[t/x])) \text{ which, by Definition 3 is } \bigcup \{f_M(v_1,\ldots,v_n) \mid v_i \in \overline{\rho}(\varphi_i[t/x]), i = \overline{1,n}\}. \text{ Here, we use the inductive hypothesis which says that } \overline{\rho}(\varphi_i) = \overline{\rho}(\varphi_i[t/x]) \text{ for all } i = \overline{1,n}, \text{ and we obtain } \bigcup \{f_M(v_1,\ldots,v_n) \mid v_i \in \overline{\rho}(\varphi_i), i = \overline{1,n}\} = \overline{\rho}(f(\varphi_1,\ldots,\varphi_n));$
- $-\overline{\rho}((\neg\varphi')[t/x]) = \overline{\rho}(\neg(\varphi'[t/x])) = M \setminus \overline{\rho}(\varphi'[t/x]) = M \setminus \overline{\rho}(\varphi') = \overline{\rho}(\neg\varphi') \text{ using } \overline{\rho}(\varphi'[t/x]) = \overline{\rho}(\varphi') \text{ from the inductive hypothesis;}$
- $-\overline{\rho}((\varphi_1 \wedge \varphi_2)[t/x]) = \overline{\rho}(\varphi_1[t/x] \wedge \varphi_2[t/x]) = \overline{\rho}(\varphi_1[t/x]) \cap \overline{\rho}(\varphi_2[t/x]) = \overline{\rho}(\varphi_1) \cap \overline{\rho}(\varphi_2) = \overline{\rho}(\varphi_1 \wedge \varphi_2) \text{ by the inductive hypothesis: } \overline{\rho}(\varphi_i[t/x]) = \overline{\rho}(\varphi_i), i \in \{1, 2\};$
- $-\overline{\rho}((\exists y.\varphi')[t/x]) = \bigcup_{v \in M} \{\overline{\rho}[v/y](\varphi'[t/x])\} = \bigcup_{v \in M} \{\overline{\rho}[v/y](\varphi')\} = \overline{\rho}(\exists y.\varphi'),$ with $y \notin var(t)$ and $\overline{\rho}[v/y](\varphi'[t/x]) = \overline{\rho}[v/y](\varphi')$ the inductive hypothesis.

Since for all M and $\rho: \mathcal{X} \to M$, $\overline{\rho}(\varphi[t/x]) = \overline{\rho}(\varphi)$ (by Prop. 2) we have $\varphi[t/x] = \varphi$.

P1.
$$\vdash \varphi \to (\varphi' \to \varphi)$$

P2. $\vdash (\varphi \to (\varphi' \to \varphi'')) \to ((\varphi \to \varphi') \to (\varphi \to \varphi''))$
P3. $\vdash (\neg \varphi' \to \neg \varphi) \to (\varphi \to \varphi')$

Fig. 2. Rules for propositional reasoning in ML.

The proof system of Matching Logic. Matching Logic provides a proof system that is sound and complete (Figure 3). The notation $\varphi[\varphi'/x]$ denotes the pattern obtained from φ by replacing all free occurrences of x with φ' . Note that the propositional calculus reasoning is subsumed by rules R1-R2 of the proof system. According to [2], R1 is in fact a set of rules that includes a version of the implicational propositional calculus (proposed by Łukasievicz [15]) shown in Fig. 2.

Unification in Matching Logic. In [22], unification has a semantical definition. More precisely, it is defined in terms of conjunctions of patterns. In order to explain this better, let us consider two ML patterns: φ and φ' . Both patterns can be matched by (possibly infinite) sets of elements, say $\overline{\rho}(\varphi)$ and $\overline{\rho}(\varphi')$, given some variable valuation ρ . In this context, finding a unifier is the same as finding a pattern φ_u that matches over a set of elements included in both $\overline{\rho}(\varphi)$ and $\overline{\rho}(\varphi')$, that is, $\overline{\rho}(\varphi_u) \subseteq \overline{\rho}(\varphi) \cap \overline{\rho}(\varphi')$, for any ρ . The most general pattern φ_u that corresponds to the largest set with this property (i.e., $\overline{\rho}(\varphi) \cap \overline{\rho}(\varphi')$), is (by Definition 5) the pattern $\varphi \wedge \varphi'$.

```
R1.
          ⊢ propositional tautologies
 R2.
          Modus ponens: \vdash \varphi_1 and \vdash \varphi_1 \rightarrow \varphi_2 imply \vdash \varphi_2
 R3.
          \vdash (\forall x.\varphi_1 \to \varphi_2) \to \varphi_1 \to (\forall x.\varphi_2), when x does not occur free in \varphi_1
R4.
          Universal generalization: \vdash \varphi implies \vdash \forall x.\varphi
R5.
          Functional substitution: \vdash (\forall x.\varphi) \land (\exists y.\varphi' = y) \rightarrow \varphi[\varphi'/x]
R5.
          Functional variable: \vdash \exists y.x = y
R6.
          Equality introduction: \vdash \varphi = \varphi
          Equality elimination: \vdash \varphi_1 = \varphi_2 \land \varphi[\varphi_1/x] \rightarrow \varphi[\varphi_2/x]
R7.
R8. \quad \vdash \forall x.x \in \varphi \text{ iff } \vdash \varphi
R9. \vdash x \in y = (x = y) when x, y \in Var
R10. \vdash x \in \neg \varphi = \neg (x \in \varphi)
R11. \vdash x \in \varphi_1 \land \varphi_2 = (x \in \varphi_1) \land (x \in \varphi_2)
R12. \vdash (x \in \exists y.\varphi) = \exists y.(x \in \varphi), with x and y distinct
R13. \vdash x \in f(\varphi_1, ..., \varphi_{i-1}, \varphi_i, \varphi_{i+1}, ..., \varphi_n) = \exists y. (y \in \varphi_i \land f(\varphi_1, ..., \varphi_{i-1}, y, \varphi_{i+1}, ..., \varphi_n)
```

Fig. 3. Sound and complete proof system of Matching Logic [22]

3 From Unification Theory to Matching Logic

This section is concerned with finding, for two given term patterns t_1 and t_2 , a pattern of the form $t \wedge \phi$ and having the following properties: 1) t is a term pattern, 2) ϕ is a predicate pattern that captures the idea of the most general unifier of t_1 and t_2 , and 3) $\models t_1 \wedge t_2 = t \wedge \phi$. This particular form $(t \wedge \phi)$ has some very practical advantages compared to $t_1 \wedge t_2$. First, there only one structural part of the formula held by t which is separated from the constraints ϕ . Second, as we show in this section, having a single structural component in a formula allows implementations to reuse existing work on unification. Finally, the separation of constraints ϕ enables the use of SMT solvers for reasoning.

The idea of transforming the pattern $t_1 \wedge t_2$ into an equivalent one $t \wedge \phi$ was suggested in [22], using an example. Here we propose a general solution that involves the unification algorithm shown in Figure 1. Example 11 illustrates how the rules of the unification algorithm are simulated by pattern transformations. Except the step (2) - which is a direct consequence of Proposition 3 applied to (1) - the rest of the equations correspond to the steps of the algorithm: **Decomposition** for (3,4,5), **Orient** for (6), and **Elimination** for (7,8).

Example 11. $t_1 \wedge t_2$ can be transformed into an equivalent formula $t \wedge \phi$:

$$t_{1} \wedge t_{2} = f(x, g(1), g(z)) \wedge f(g(y), g(y), g(g(x))$$

$$= f(x, g(1), g(z)) \wedge (f(x, g(1), g(z)) = f(g(y), g(y), g(g(x)))$$

$$= f(x, g(1), g(z)) \wedge (x = g(y)) \wedge (g(1) = g(y)) \wedge (g(z) = g(g(x)))$$

$$= f(x, g(1), g(z)) \wedge (x = g(y)) \wedge (1 = g(y)) \wedge (g(z) = g(g(x)))$$

$$= f(x, g(1), g(z)) \wedge (x = g(y)) \wedge (1 = y) \wedge (z = g(x))$$

$$= f(x, g(1), g(z)) \wedge (x = g(y)) \wedge (y = 1) \wedge (z = g(x))$$

$$= f(x, g(1), g(z)) \wedge (x = g(y)) \wedge (y = 1) \wedge (z = g(x))$$

$$= f(x, g(1), g(z)) \wedge (x = g(y)) \wedge (y = 1) \wedge (z = g(x))$$

$$= f(x, g(1), g(z)) \wedge (x = g(y)) \wedge (y = 1) \wedge (z = g(x))$$

$$= f(x, g(1), g(z)) \wedge (x = g(y)) \wedge (y = 1) \wedge (z = g(x))$$

$$= f(x, g(1), g(z)) \wedge (x = g(y)) \wedge (y = 1) \wedge (z = g(x))$$

$$= f(x, g(1), g(z)) \wedge (x = g(y)) \wedge (y = 1) \wedge (z = g(x))$$

$$= f(x, g(1), g(z)) \wedge (x = g(y)) \wedge (y = 1) \wedge (z = g(x))$$

$$= f(x, g(1), g(z)) \wedge (x = g(y)) \wedge (y = 1) \wedge (z = g(x))$$

$$= f(x, g(1), g(z)) \wedge (x = g(y)) \wedge (y = 1) \wedge (z = g(x))$$

$$= f(x, g(1), g(z)) \wedge (x = g(y)) \wedge (y = 1) \wedge (z = g(x))$$

$$= f(x, g(1), g(z)) \wedge (x = g(y)) \wedge (y = 1) \wedge (z = g(x))$$

$$= f(x, g(1), g(z)) \wedge (x = g(y)) \wedge (y = 1) \wedge (z = g(x))$$

$$= f(x, g(1), g(z)) \wedge (x = g(y)) \wedge (y = 1) \wedge (z = g(x))$$

$$= f(x, g(1), g(z)) \land (x = g(1)) \land (y = 1) \land (z = g(x))$$
 (7)

$$=\underbrace{f(x,g(1),g(z))}_{t} \wedge \underbrace{(x=g(1)) \wedge (y=1) \wedge (z=g(g(1)))}_{\phi} \tag{8}$$

Mainly, the idea illustrated in Example 11 is to use the syntax unification algorithm to determine ϕ . In the rest of this section we introduce several notions and prove some intermediate technical results that we use to formally prove the equality (in the ML sense) of $t \wedge \phi$ and $t_1 \wedge t_2$.

3.1 Encoding unification in ML

Terms can be naturally expressed in ML as term patterns provided the following set of axiom patterns which we consider implicitly included in (S, Σ, F) :

- 1. The definedness patterns, needed to define equality and membership;
- 2. Axioms ensuring that the structural patterns are built only with functional symbols (cf. Definition 10);
- 3. Axioms ensuring that all functional symbols used in structural patterns are interpreted as injections:

$$f(x_1, \dots, x_n) = f(y_1, \dots, y_n) \to x_1 = y_1 \wedge \dots \wedge x_n = y_n. \tag{9}$$

Let us explain here why this particular axiom ensures injectivity. Indeed, in any model M and for any $\rho: \mathcal{X} \to M$, $\overline{\rho}(f(x_1, \dots, x_n) = f(y_1, \dots, y_n) \to x_1 = y_1 \wedge \dots \wedge x_n = y_n) = M$ (cf. Prop. 1) implies that $\overline{\rho}(f(x_1, \dots, x_n) = f(y_1, \dots, y_n)) \subseteq \overline{\rho}(x_1 = y_1 \wedge \dots \wedge x_n = y_n)$). If $\overline{\rho}(f(x_1, \dots, x_n) = f(y_1, \dots, y_n)) = \emptyset$ (recall that equality is a predicate), then (cf. Prop. 2) $\overline{\rho}(f(x_1, \dots, x_n)) \neq \overline{\rho}(f(y_1, \dots, y_n))$, which implies that $f_M(v_1, \dots, v_n) \neq f_M(u_1, \dots, u_n)$, for all $v_i = \rho(x_i)$, $u_i = \rho(y_i)$, $i = \overline{1,n}$. Therefore, for all v_i , u_i , $v_i \neq u_i$ implies $f_M(v_1, \dots, v_n) \neq f_M(u_1, \dots, u_n)$ holds as well, and hence the function f_M is injective. On the other hand, if $\overline{\rho}(f(x_1, \dots, x_n)) = f(y_1, \dots, y_n)) = M$, then $\overline{\rho}(f(x_1, \dots, x_n)) = \overline{\rho}(f(y_1, \dots, y_n))$ and $\overline{\rho}(x_1 = y_1 \wedge \dots \wedge x_n = y_n) = M$. Therefore, $f_M(v_1, \dots, v_n) = f_M(u_1, \dots, u_n)$, for all $v_i \in \{\rho(x_i)\}$, $u_i \in \{\rho(y_i)\}$, $i = \overline{1,n}$ implies $\bigwedge_i v_i = u_i$ (because, in particular $\overline{\rho}(x_i = y_i) = M$), and f_M is injective.

As suggested by Example 11, our solution for finding an equivalent form $t \wedge \phi$ for $t_1 \wedge t_2$ requires the simulation of the unification algorithm shown in Figure 1 in ML. First, we have to encode unification problems as ML formulas:

Definition 11. For each unification problem $P = \{v_1 = u_1, \dots, v_n = u_n\}$ we define a corresponding ML predicate $\phi^P \triangleq \bigwedge_{i=1}^n v_i = u_i$. Also, $\phi^{\perp} = \perp$.

A unification problem in solved form has a corresponding substitution. These substitutions can be encoded as ML predicates called *substitution patterns*:

Definition 12. A substitution pattern that corresponds to a substitution $\sigma = \{x_i \mapsto u_i \mid i = 1, ..., n\}$ is a predicate of the form $\phi^{\sigma} \triangleq \bigwedge_{i=1}^m x_i = u_i$.

For the particular case when σ corresponds to a unification problem P in solved form we have $\phi^{\sigma} = \phi^{P}$. For a term pattern t, we use the same notation $t\sigma$ to denote the corresponding term pattern obtained after applying substitution σ to t, as follows: $x_{i}\sigma = u_{i}$ if $(x_{i} \mapsto u_{i}) \in \sigma$; $x\sigma = x$ if $(x \mapsto \bot) \notin \sigma$; finally, $f(t_{1}, \ldots, t_{n})\sigma = f(t_{1}\sigma, \ldots, t_{n}\sigma)$.

Example 12. Terms $t_1 \triangleq f(x, g(1), g(z))$ and $t_2 \triangleq f(g(y), g(y), g(g(x)))$ are term patterns in ML. For the unifier $\sigma = \{x \mapsto g(1), y \mapsto 1, z \mapsto g(g(1))\}$ of t_1 and t_2 the corresponding substitution pattern ϕ^{σ} is $x = g(1) \land y = 1 \land z = g(g(1))$. Now, both $t_1\sigma$ and $t_2\sigma$ are the same with the ML pattern f(g(1), g(1), g(g(1))).

One may be tempted to say that for every term t, $t\sigma$ is equal to $t \wedge \phi^{\sigma}$. However, this is not always true. For instance, if t is a variable $x \in \mathcal{X}$ such that $x\sigma = x$ and $\overline{\rho}(\phi^{\sigma}) = \emptyset$, and $\rho : \mathcal{X} \to M$ is a valuation, then $\overline{\rho}(x \wedge \phi^{\sigma}) = \rho(x) \cap \overline{\rho}(\phi^{\sigma}) = \emptyset \neq \{\rho(x)\} = \overline{\rho}(x) = \overline{\rho}(x\sigma)$. The following lemma formalises the precise relation between $t\sigma$ and $t \wedge \phi^{\sigma}$:

Lemma 2. If t is a term pattern and σ a substitution then $F \models t\sigma \wedge \phi^{\sigma} \leftrightarrow t \wedge \phi^{\sigma}$.

Proof. Let us choose an arbitrary model M. We have to prove that $M \models t\sigma \land \phi^{\sigma} \leftrightarrow t \land \phi^{\sigma}$. By Proposition 2, $M \models t\sigma \land \phi^{\sigma} \leftrightarrow t \land \phi^{\sigma}$ iff $\overline{\rho}(t\sigma \land \phi^{\sigma}) = \overline{\rho}(t \land \phi^{\sigma})$ for any $\rho : Var \to M$. By Proposition 1, this holds iff $\overline{\rho}(t) \cap \overline{\rho}(\phi^{\sigma}) = \overline{\rho}(t\sigma) \cap \overline{\rho}(\phi^{\sigma})$. If $\overline{\rho}(\phi^{\sigma}) = \emptyset$ then this equality holds trivially. If $\overline{\rho}(\phi^{\sigma}) = M^5$ then $\overline{\rho}(t\sigma) \cap M = \overline{\rho}(t) \cap M$ iff $\overline{\rho}(t\sigma) = \overline{\rho}(t)$. We proceed by structural induction on t:

- Base case. t=x. Recall that $\phi^{\sigma} \triangleq \bigwedge_{i=1}^{m} x_i = u_i$. We have two sub-cases: 1. $x \in \{x_1, \dots, x_m\}$: since $\overline{\rho}(\phi^{\sigma}) = M$ then $\overline{\rho}(\bigwedge_{i=1}^{m} x_i = u_i) = M$ which implies $\bigcap_{i}^{m} \overline{\rho}(x_i = u_i) = M$. Thus, $\overline{\rho}(x_i = u_i) = M$ iff $\overline{\rho}(x_i) = \overline{\rho}(u_i)$ (using Proposition 2) and in particular $\rho(x) = \overline{\rho}(u) = \overline{\rho}(x\sigma)$. 2. $x \notin \{x_1, \dots, x_m\}$: in this case $x\sigma = x$ and $\overline{\rho}(x\sigma) = \rho(x)$.
- Inductive step. $t = f(t_1, ..., t_n)$ and the inductive hypothesis holds for all subterm patterns $t_1, ..., t_n$. Then: $\overline{\rho}(f(t_1, ..., t_n)\sigma) = \overline{\rho}(f(t_1\sigma, ..., t_n\sigma)) = f_M(\overline{\rho}(t_1\sigma), ..., \overline{\rho}(t_n\sigma)) = f_M(\overline{\rho}(t_1), ..., \overline{\rho}(t_n)) = \overline{\rho}(f(t_1, ..., t_n))$, using the inductive hypothesis and Definitions 9 and 5.

Lemma 3 shows that the steps performed by the unification algorithm (Figure 1) can be encoded as implications in ML. Note that we only consider the case when the most general unifier exists.

⁵ This should be M_s where $s \in S$ is the sort of $t\sigma$, but we choose not to show the sort explicitly.

Lemma 3. In the context of Figure 1, if $P \Rightarrow P'$ and $P' \neq \bot$ then $F \models \phi^P \rightarrow \phi^{P'}$, for all unification problems P and P'.

Proof. We have to prove that for all models M and for all valuations $\rho: \mathcal{X} \to M$, $\overline{\rho}(\phi^P \to \phi^{P'}) = M$, that is (by Proposition 1), $\overline{\rho}(\phi^P) \subseteq \overline{\rho}(\phi^{P'})$. Since ϕ^P is a predicate, $\overline{\rho}(\phi^P)$ is either \emptyset (in this case the lemma holds trivially) or M. Let $\overline{\rho}(\phi^P) = M$; we proceed by case analysis on the rule applied for step $P \Rightarrow P'$:

- 1. Delete: $\overline{\rho}(\phi^{P \cup \{t \stackrel{.}{=} t\}}) = \overline{\rho}(\phi^P \wedge t = t) = \overline{\rho}(\phi^P) \cap \overline{\rho}(t = t) \subseteq \overline{\rho}(\phi^P)$.
- 2. **Decomposition**: On the one hand we have $\overline{\rho}(\phi^P \cup \{f(t_1, \dots, t_n) = f(t'_1, \dots, t'_n)\}) = \overline{\rho}(\phi^P \wedge f(t_1, \dots, t_n) = f(t'_1, \dots, t'_n)) = \overline{\rho}(\phi^P) \cap \overline{\rho}(f(t_1, \dots, t_n) = f(t'_1, \dots, t'_n)).$ On the other hand, $\overline{\rho}(\phi^{P \cup \{t_1 = t'_1, \dots, t_n = t'_n\}}) = \overline{\rho}(\phi^P \wedge t_1 = t'_1 \wedge \dots \wedge t_n = t'_n) = \overline{\rho}(\phi^P) \cap \overline{\rho}(t_1 = t'_1) \cap \dots \cap \overline{\rho}(t_n = t'_n).$ If $\overline{\rho}(f(t_1, \dots, t_n) = f(t'_1, \dots, t'_n)) = \emptyset$ then the inclusion holds trivially. If $\overline{\rho}(f(t_1, \dots, t_n) = f(t'_1, \dots, t'_n)) = M$, then $\overline{\rho}(f(t_1, \dots, t_n)) = \overline{\rho}(f(t'_1, \dots, t'_n))$ (by Proposition 2) iff $f_M(\overline{\rho}(t_1), \dots, \overline{\rho}(t_n)) = f_M(\overline{\rho}(t'_1), \dots, \overline{\rho}(t'_n))$ (recall that
- $\overline{\rho}(t_1) = \overline{\rho}(t'_1), \dots, \overline{\rho}(t_n) = \overline{\rho}(t'_n). \text{ Thus, } \overline{\rho}(t_1 = t'_1) = \dots = \overline{\rho}(t_n = t'_n) = M.$ 3. Orient: $\overline{\rho}(\phi^{P \cup \{f(t_1, \dots, t_n) = x\}}) = \overline{\rho}(\phi^P \wedge f(t_1, \dots, t_n) = x) = \overline{\rho}(\phi^P \wedge x = f(t_1, \dots, t_n)) = \overline{\rho}(\phi^P \cup \{x = f(t_1, \dots, t_n)\}).$

f is a functional symbol and f_M is an injective function) which implies

- 4. **Elimination**: we have $x \notin var(t), x \in var(P)$ and we have to show that $\overline{\rho}(\phi^{P \cup \{x \doteq t\}}) = \overline{\rho}(\phi^P) \cap \overline{\rho}(x=t)$ is included in the set $\overline{\rho}(\phi^{P\{x \mapsto t\} \cup \{x \doteq t\}}) = \overline{\rho}(\phi^{P\{x \mapsto t\}}) \cap \overline{\rho}(x=t) = \overline{\rho}(\phi^P[t/x]) \cap \overline{\rho}(x=t)$. If $\overline{\rho}(x=t) = \emptyset$ the inclusion holds trivially. The interesting case is when $\overline{\rho}(x=t) = M$, i.e., x=t. In this case it is sufficient to prove $\overline{\rho}(\phi^P) = \overline{\rho}(\phi^P[t/x])$ which follows from Lemma 1.
- 5. Occurs check and Symbol clash cannot be applied because $P' \neq \bot$.

Lemma 4. If σ is the most general unifier of t_1 and t_2 then $F \models (t_1 = t_2) \rightarrow \phi^{\sigma}$.

Proof. Note that σ is obtained using the unification algorithm in Figure 1. The algorithm generates a finite sequence $\{t_1 = t_2\} \Rightarrow \cdots \Rightarrow P^{\sigma}$ where P^{σ} is in solved form and corresponds to mgu σ . If we apply Lemma 3 for each step in this sequence we also have a sequence of valid implications $(t_1 = t_2) \rightarrow \cdots \rightarrow \phi^{\sigma}$.

The reversed implication is given by the following lemma:

Lemma 5. If σ is a unifier of term patterns t_1 and t_2 then $\models \phi^{\sigma} \rightarrow (t_1 = t_2)$.

Proof. We have to prove that for all models M and for all valuations $\rho: \mathcal{X} \to M$, $\overline{\rho}(\phi^{\sigma} \to (t_1 = t_2)) = M$. By Proposition 1, we have to prove that $\overline{\rho}(\phi^{\sigma}) \subseteq \overline{\rho}(t_1 = t_2)$). The case $\overline{\rho}(\phi^{\sigma}) = \emptyset$ is trivial. When $\overline{\rho}(\phi^{\sigma}) = M$ it is sufficient to prove that $\overline{\rho}(t_1 = t_2) = M$, namely, $\overline{\rho}(t_1) = \overline{\rho}(t_2)$.

From Lemma 2 we have $F \models t_1 \sigma \wedge \phi^{\sigma} \leftrightarrow t_1 \wedge \phi^{\sigma}$ and $F \models t_2 \sigma \wedge \phi^{\sigma} \leftrightarrow t_2 \wedge \phi^{\sigma}$. This implies that $\overline{\rho}(t_1 \sigma) \cap \overline{\rho}(\phi^{\sigma}) = \overline{\rho}(t_1) \cap \overline{\rho}(\phi^{\sigma})$ and $\overline{\rho}(t_2 \sigma) \cap \overline{\rho}(\phi^{\sigma}) = \overline{\rho}(t_2) \cap \overline{\rho}(\phi^{\sigma})$. Because $\overline{\rho}(\phi^{\sigma}) = M$, we have $\overline{\rho}(t_1) = \overline{\rho}(t_1 \sigma)$ (\spadesuit) and $\overline{\rho}(t_2 \sigma) = \overline{\rho}(t_2)$ (\clubsuit).

Since σ is a unifier, then $t_1\sigma$ and $t_2\sigma$ are syntactical equal. This implies $\overline{\rho}(t_1\sigma) = \overline{\rho}(t_2\sigma)$; by (\clubsuit) and (\spadesuit) we obtain $\overline{\rho}(t_1) = \overline{\rho}(t_2\sigma) = \overline{\rho}(t_2\sigma) = \overline{\rho}(t_2)$. \square

Lemma 6. If σ is the mgu of t_1 and t_2 then $F \models (t_1 = t_2) \leftrightarrow \phi^{\sigma}$.

Proof. Consequence of Lemmas 5 and 4.

Now we are ready establish the main contribution of this section, namely that the syntactic unification algorithm is *sound* for semantic unification in ML:

Theorem 1 (Soundness). Let σ be the most general unifier of t_1 and t_2 obtained by applying the algorithm shown in Figure 1 to the unification problem $\{t_1 = t_2\}$. Then $F \models t_1 \land t_2 = t_1 \land \phi^{\sigma}$ and $F \models t_1 \land t_2 = t_2 \land \phi^{\sigma}$.

Proof. We have $F \models t_1 \land t_2 = t_1 \land (t_1 = t_2)$ by Proposition 3, and $F \models (t_1 = t_2) \leftrightarrow \phi^{\sigma}$ by Lemma 6. Therefore, $F \models t_1 \land t_2 = t_1 \land \phi^{\sigma}$. The second conclusion is obtained in a similar way.

Theorem 1 states that if the unification algorithm successfully terminates, then the most general unifier supplies the constraint pattern needed to express the semantic unifier as a conjunction of a structural pattern and a constraint.

Completeness. An interesting question to ask here is what happens when the input term patterns are not unifiable? In such a case, the unification algorithm fails and the sequence of transformations over the term patterns ends with \bot . In fact, the condition $P' \neq \bot$ in Lemma 3 prevents exactly this situation to happen. In order to remove this condition, one needs to prove $\phi^P \Rightarrow \bot$ when **Occurs check** and **Symbol clash** apply.

The injectivity axiom is not enough to prove these properties and stronger axioms are needed. Obviously, a tempting alternative is to use constructors instead of injections. In [22], the *constructors* are defined as follows:

- No junk: $F \models \bigvee_{s} \exists x_1 : s_1 \dots \exists x_m : s_m . c(x_1, \dots, x_m)$, where $c \in \Sigma_{s_1 \dots s_m, s}$;
- No confusion, different constructors:
 - $F \models \neg(c(x_1,\ldots,x_m) \land c'(y_1,\ldots,y_n)), \text{ with } c \neq c', c \in \Sigma_{s_1\ldots s_m,s}, \text{ and } c \in \Sigma_{s_1\ldots s_n,s}.$
- No confusion, same constructors:

```
F \models c(x_1, \ldots, x_m) \land c(y_1, \ldots, y_m) \rightarrow c(x_1 \land y_1, \ldots, x_m \land y_m), \text{ with } c \in \Sigma_{s_1 \ldots s_m, s}.
```

No junk ensures the that constructors can be used to construct all the elements of the target domain. No confusion, different constructors ensures that constructors yield a unique way to construct each element of the target domain. No confusion, same constructors says that constructors are injective.

The **no confusion, different constructors** axiom is sufficient to prove Lemma 3 for the **Symbol clash** case. Unfortunately, none of these axioms is enough to prove the lemma for the **Occurs check** case. The main issue is that $x = f(t_1, ..., t_n)$ cannot be proved equal to \bot when $x \in var(f(t_1, ..., t_n))$. Recall that the condition $x \in var(f(t_1, ..., t_n))$ implies that x occurs at least in a term t_i .

The axioms for constructors cannot prevent to have $M \models x = f(t_1, ..., t_n)$ for some (S, Σ, F) -model M, when $x \in var(f(t_1, ..., t_n))$. Here is a counterexample.

Let s be a sort and and Σ a signature which includes only a functional symbol $f \in \Sigma_{s,s}$. Also, let M be a ML model where $f_M(a) = a$, with a the only element in M. Note that any valuation $\rho : Var \to M$, assigns to variables a set equal to $\{a\}$.

Also, note that f satisfies the axioms above: first, the **no confusion, different constructors** is satisfied trivially since there is no other symbol in Σ ; second, the **no confusion, same constructors** holds, since $\overline{\rho}(f(x) \land f(y)) = \overline{\rho}(f(x)) \cap \overline{\rho}(f(y)) = \{a\} \cap \{a\} = \{f_M(a)\} \subseteq \overline{\rho}(\underline{f(x \land y)})$; finally, the **no junk** axiom $\exists x. f(x)$ holds, since $\overline{\rho}(\exists x. f(x)) = \bigcup_{a \in M} \overline{\rho[a/x]}(f(x)) = \bigcup_{a \in M} f_M(a) = M$. However, x and x are unifiable in the sense of ML.

In our opinion, there are two choices to handle such situations. First, we can modify the syntactic unification algorithm such that it reports also the mappings $x \mapsto t(x)$ when **Occurs check** is applicable (here, t(x) denotes a term that has x as subterm). If we want to consider only models where the equalities x = t(x) do not hold, then we simply add the axioms $\neg(x = t(x))$ to F. The problem here is that we do not know a priori these axiom patterns. Second, if we want to consider models where the equalities x = t(x) may hold, then we define $\phi^{t_1=t_2}$ as being $(\bigwedge x = t(x)) \implies \phi^{\sigma}$, where $(\bigwedge x = t(x))$ is the conjunction over all mappings introduced by **Occurs check**, and σ is the substitution defined by the **Elimination** mappings. The price paid in this case is that we may get formulas that SMT solvers might not be able to handle.

4 Generating proofs

In this section we present a sound strategy to generate formal proofs of equivalence between $t_1 \wedge t_2$ and $t_1 \wedge \phi^{\sigma}$ with σ the most general unifier of t_1 and t_2 . This strategy uses the rules of the ML proof system [22] and some derived rules that mimic the steps of the unification algorithm.

We first explain the main idea of our strategy using Example 11. The equations (1-8) correspond to the steps of the unification algorithm shown in Figure 1: **Decomposition** for equations (3,4,5), **Orient** for (6), and **Elimination** for (7,8). The only exception is the equation (2), which is justified by Proposition 3. This particular example suggests that successive transformations over the initial pattern produce a conjunction of a term pattern and a constraint. The fact that in ML we can express the mgu of term patterns t_1 and t_2 as a ML pattern $t_1 \wedge t_2$ (i.e., at the *object level*) is important: using the transformations above we can actually generate a *proof certificate* that the obtained constrained term pattern $t_1 \wedge \phi^{\sigma}$ is equal to $t_1 \wedge t_2$.

Our current approach is to generate proofs in two stages: first, we start with $t_1 \wedge t_2$ and we derive $t_1 \wedge \phi^{\sigma}$ using several derived proof rules which mimic the steps of the unification algorithm; these will be we proved separately using the ML proof system; second, we start with $t_1 \wedge \phi^{\sigma}$ and we derive $t_1 \wedge t_2$ using the original proof system of ML. For both stages we have strategies that always produce proofs when the most general unifier exists.

Stage 1 The list of derived rules that we use in the first stage is shown below. For each rule we indicate the corresponding rule from the unification algorithm:

```
 \Delta 1. \ F \vdash \varphi \land (t=t) \rightarrow \varphi  Delete  \Delta 2. \ F \vdash \varphi \land (f(t_1,..,t_n) = f(t_1',..,t_n')) \rightarrow \varphi \land t_1 = t_1' \land .. \land t_n = t_n'  Decomposition  \Delta 3. \ F \vdash \varphi \land (f(t_1,..,t_n) = x) \rightarrow \varphi \land (x = f(t_1,..,t_n))  Orient  \Delta 4. \ F \vdash \varphi \land (x = t) \rightarrow \varphi[t/x] \land (x = t), \text{ if } x \not\in var(t), x \in var(\varphi)  Elimination
```

These rules are proved (semantically) in the proof of Lemma 3, but we also prove them using the ML proof system (Table 4). Note that there are no corresponding rules for **Occurs check** and **Symbol clash**, because we are interested in generating proofs only for the cases when the most general unifier exists. An example of a proof that uses the derived rules is shown below:

```
f(x,g(1),g(z)) \wedge f(g(y),g(y),g(g(x))
                                                                                  hypothesis
ii
                                                                                  Prop 3: i
     f(x,g(1),g(z)) \wedge (f(x,g(1),g(z))) = f(g(y),g(y),g(g(x)))
    f(x,g(1),g(z)) \land (x=g(y)) \land (g(1)=g(y)) \land (g(z)=g(g(x))) \Delta 2: ii
                                                                                  \Delta 2: iii
    f(x, g(1), g(z)) \land (x = g(y)) \land (1 = g(y)) \land (g(z) = g(g(x)))
     f(x, g(1), g(z)) \land (x = g(y)) \land (1 = y) \land (z = g(x))
                                                                                  \Delta 2: iv
   f(x, g(1), g(z)) \land (x = g(y)) \land (y = 1) \land (z = g(x))
                                                                                  ∆3: v
vii f(x, g(1), g(z)) \land (x = g(1)) \land (y = 1) \land (z = g(x))
                                                                                  \Delta 4: vi
viii f(x, g(1), g(z)) \land (x = g(1)) \land (y = 1) \land (z = g(g(1)))
                                                                                  \Delta 4:vii
```

Each line represents a proof step annotated with a justification specified as \langle the applied proof rule \rangle : \langle references to previous steps \rangle . We intentionally omit $F \vdash$ before each proof step and we prefer to add some useful annotations at the end.

The first line is our hypothesis. The pattern derived at the second line is obtained by applying Proposition 3 to pattern i. Then, the strategy is given by the unification algorithm. The third line is obtained by applying $\Delta 2$ to ii, that is, **Decomposition** for symbol f. To keep the above proof simple, we silently use the associativity and commutativity of \wedge . Starting with the formula at step i we are able to derive the formula from step viii.

It is easy to see that the strategy of the first stage is dictated by the unification algorithm shown in Figure 1. Its soundness is given by Proposition 3 and Lemma 3. However, we provide proofs that use the rules of the ML proof system for $\Delta 1$ - $\Delta 4$ and Proposition 3 in Table 4 . It is worth noting that we used only a few rules of the ML proof system: R1, R2, R7, R9.

To validate the proofs from Table 4, we have encoded the definitions, the proof rules and the needed axioms in Coq. Then we checked our proofs mechanically. We have formulated and proved a deduction theorem which holds for the ML proof system fragment that we use (R1, R2, R7, R9). Also, we provide Coq proofs for rather trivial steps (i.e., \land elimination, \land introduction, and other simple propositional lemmas) using the rules in Figure 2. The (assertive) proof style that we used is intended to improve source code reading for non-expert Coq users. The Coq code can be found at [3].

Stage 2 We start explaining our strategy for stage 2 by proving the reversed implication of our example:

$\Delta 1$	(Delete):	
i	$\varphi \wedge t = t$	hypothesis
ii	arphi	R1: i
$\Delta 2$	(Decomposition):	
i	$\varphi \wedge (f(t_1,,t_n) = f(t'_1,,t'_n))$	hypothesis
ii	arphi	<i>R</i> 1: i
iii	$f(t_1,,t_n) = f(t'_1,,t'_n)$	R1: i
iv	$f(t_1,,t_n) = f(t'_1,,t'_n) \to t_1 = t'_1 \land \land t_n = t'_n$	inj axiom in F
v	$t_1 = t_1' \wedge \wedge t_n = t_n'$	R2: iii, iv
vi	$\varphi \wedge t_1 = t_1' \wedge \wedge t_n = t_n'$	R1: ii, v
$\Delta 3$	(Orient):	,
i	$\varphi \wedge (f(t_1,, t_n) = x)$	hypothesis
ii	φ	<i>R</i> 1: i
iii	$f(t_1,,t_n) = x$	R1: i
iv	$x = f(t_1,, t_n)$	(symmetry of $=$): iii
v	$\varphi \wedge (x = f(t_1,, t_n))$	R1: ii, iv
$\Delta 4$	(Elimination):	
i	$\varphi \wedge (x=t)$	hypothesis
ii	φ	R1: i
iii	x = t	R1: i
iv	$\varphi[x/x]$	ii: $(\varphi = \varphi[x/x])$
v	$x = t \wedge \varphi[x/x]$	R1: iii, iv
vi	$(x = t) \land \varphi[x/x] \to \varphi[t/x]$	R7
vii	$\varphi[t/x]$	R2: v, vii
viii	$\varphi[t/x]$ $\varphi[t/x] \wedge x = t$	R1: vii, iii
Prop 3		7t1. VII, III
i		hypothesis
ii	$\varphi \wedge (\varphi = \varphi')$ $\varphi = \varphi'$	R1: i
iii	. ,	R1: i
	$arphi = arphi' \ arphi'$	
iv	•	R1: ii, iii, Prop. 2
v.	$\varphi \wedge \varphi'$	R1: iii, iv
vi 	$(\varphi \wedge \varphi') \to (\varphi \wedge (\varphi = \varphi') \to (\varphi \wedge \varphi'))$	R1: P1
vii	$\varphi \wedge (\varphi = \varphi') \to (\varphi \wedge \varphi')$	R2: v, vi
viii	$(\varphi \wedge \varphi')$	R2: i, vii
Prop 3		
i	$(\varphi \wedge \varphi')$	hypothesis
ii	$[\varphi \wedge \varphi']$	$definedness \ axiom \ in \ F$
iii	$[\varphi \wedge \varphi'] \to ((\varphi \wedge \varphi') \to [\varphi \wedge \varphi'])$	R1 (P1)
iv	$(\varphi \wedge \varphi') \to [\varphi \wedge \varphi']$	R2: ii, iii
v	$(\varphi \land \varphi') \to \varphi \in \varphi'$	definition of \in : iv
vi	$(\varphi \wedge \varphi') \to (\varphi = \varphi')$	R9: v
vii	$(\varphi \wedge \varphi') \to \varphi$	$R1^+$
viii	$(\varphi \wedge \varphi') \to \varphi \wedge (\varphi = \varphi')$	$R1^+$: vi, vii
ix	$\varphi \wedge (\varphi = \varphi')$	R2: i, viii

Table 1. Proofs of the derived rules $\varDelta 1\text{-}\varDelta 4$ and Proposition 3.

```
f(x, g(1), g(z)) \land (x = g(1)) \land (y = 1) \land (z = g(g(1)))
                                                                     hypothesis
Х
                                                                     R1: x
xi
      f(x, g(1), g(z))
      (x = g(1)) \land (y = 1) \land (z = g(g(1)))
                                                                     R1: x
xii
                                                                     R1: xii
xiii
      x = g(1)
xiv
      (y=1) \land z = g(g(1))
                                                                     R1: xii
                                                                     R1: xiv
xv
      y = 1
      z = g(g(1))
                                                                     R1: xiv
xvi
      f(x, g(1), g(z)) = f(x, g(1), g(z))
                                                                     R6
      f(g(y), g(y), g(g(x))) = f(g(y), g(y), g(g(x)))
                                                                     R6
      f(g(1), g(1), g(z)) = f(x, g(1), g(z))
                                                                     R7: xvii,xiii
xix
      f(g(1), g(1), g(g(g(1)))) = f(x, g(1), g(z))
                                                                     R7: xix, xvi
XX
      f(g(1),g(y),g(g(x))=f(g(y),g(y),g(g(x))
                                                                     R7: xviii,xv
xxi
      f(g(1), g(1), g(g(x))) = f(g(y), g(y), g(g(x)))
                                                                     R7: xxi, xv
xxiii f(g(1), g(1), g(g(g(1))) = f(g(y), g(y), g(g(x)))
                                                                     R7: xxii,xiii
                                                                     R7: xx, xxiii
xxiv f(x, g(1), g(z)) = f(g(y), g(y), g(g(x)))
xxv f(x,g(1),g(z)) \land (f(x,g(1),g(z)) = f(g(y),g(y),g(g(x))) R1: xi, xxiv
                                                                     Prop 3: xxv
xxvi f(x,g(1),g(z)) \wedge f(g(y),g(y),g(g(x))
```

Now, we present the strategy corresponding to this stage, which has five steps:

- 1. start with $t_1 \wedge \phi^{\sigma}$ as hypothesis;
- 2. use R1 to break the large conjunction from the hypothesis (e.g., steps xi-xvi)
- 3. use R6 to introduce equalities $t_1 = t_1$ and $t_2 = t_2$ (e.g., steps xvii, xviii);
- 4. use R7 to replace the variables occurring in the left hand sides of the equalities (e.g., xx, xxiii);
- 5. use R7 to equate the right hand sides of the equalities produced by the previous step(e.g. xxiv); then apply R1 (∧ introduction, e.g., xxv), and finally Proposition 3 (e.g., xxvi).

This strategy essentially rebuilds the semantic unifier $t_1 \wedge t_2$ starting with $t_1 \wedge \phi^{\sigma}$. Because ϕ^{σ} has the form $\bigwedge_{i=1}^{n} x_i = \underline{u_i}$ the step 2 will always produce equalities of the form $x_i = u_i$ for all $i = \overline{1, n}$. In the left hand sides of the equalities introduced by step 3 we can always substitute x_i by u_i . Since σ is the most general unifier, the left hand sides will become equal after substitutions performed by step 4. Finally, we can always apply R7, R1, and Proposition 3 conveniently to obtain $t_1 \wedge t_2$. Because it uses only rules from the original proof system of ML (check Table 4 for proof of Proposition 3), this strategy is sound.

5 Conclusions

Previous verification efforts with ML [23,26,25,27,5,14,10,17,20,9] were based on unification. However, unification was always considered a trusted component.

In this paper we finally tackle down this issue by proposing a sound method for unification which involves a syntactic unification algorithm. More precisely, we show that the syntactic unification algorithm proposed by Martelli and Montanari [16] is *sound* for semantic unification in ML. We explain by means of a counterexample, why this algorithm is not *complete* for semantic unification.

Finally, we show a *provableness* property of the same algorithm: we provide a sound strategy to generate a proof certificate when the most general unifier exists. This proof uses some derived rules (which we encode and prove in Coq) and the rules of the ML proof system [22].

Related work. We include here only the comparison with the closest related work Kore [2]: an implementation of ML which is currently under development [1]. They handle conjunctions via a set of transformations over patterns intended to serve a more general purpose, for instance, to deal with partiality and injections (subsort relations). The approach that we proposed here focuses on how the syntactic unification algorithms can be used to help reasoning in ML.

Future work. The fact that the proof of the soundness of our approach depends on the unification algorithm is intriguing. We intend to explore whether there is an independent proof, which uses only the definition of the most general unifier.

A topic that also needs further investigation is the completeness of the algorithm with respect to semantic unification. In Section 3.1 we discuss the completeness issue and we sketch two solutions, but a deeper investigation is required.

Via private communication with the Kore team we learned that a slightly modified proof system is implemented in Kore for which a deduction theorem can be proved. We intend to adapt our proof generation strategy to use this new proof system since it seems that the deduction theorem can simplify some steps.

Finally, a completely new ground to explore is unification modulo axioms (e.g., commutativity, associativity, and so on). Obviously, it is more challenging to use the existing unification modulo axioms algorithms in the same manner as we have done for syntactic unification.

Acknowledgements. We would like to especially thank the Kore developers and researchers: Phillip Harris, Traian Şerbănuţă and Virgil Şerbănuţă for their valuable assistance and feedback. They helped us with our proof generation strategy and they suggested improvements for our current work. We also want to specially thank Grigore Roşu for the fruitful discussions that we had about this topic at FROM 2018. This work was supported by a grant of the "Alexandru Ioan Cuza" University of Iaşi, within the Research Grants program, Grant UAIC, code GI-UAIC-2017-08.

References

- The Kore language (github repository, last accessed 2018-11-07): https://github.com/kframework/kore, https://github.com/kframework/kore
- The semantics of K (online document, last accessed 2018-09-03), https://github.com/kframework/kore/blob/master/docs/semantics-of-k.pdf
- 3. Arusoaie, A.: Coq proofs for unification, https://github.com/andreiarusoaie/proof-generation-for-unification
- Arusoaie, A., Lucanu, D., Rusu, V.: Symbolic execution based on language transformation. Comp. Lang., Systems & Structures 44, 48–71 (2015)

- Arusoaie, A., Nowak, D., Rusu, V., Lucanu, D.: A Certified Procedure for RL Verification. In: SYNASC 2017. pp. 129–136. IEEE CPS, Timisoara, Romania (Sep 2017), https://hal.inria.fr/hal-01627517
- 6. Baader, F., Snyder, W.: Unification theory (1999)
- Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanovic, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: CAV 2011. pp. 171–177 (2011)
- 8. Bogdănaş, D., Roşu, G.: K-Java: A Complete Semantics of Java. In: POPL 2015. pp. 445–456 (2015). https://doi.org/10.1145/2676726.2676982
- Ştefănescu, A., Ciobâcă, Ş., Mereuţă, R., Moore, B.M., Şerbănuţă, T.F., Roşu, G.: All-path reachability logic. In: RTA-TLCA. LNCS, vol. 8560, pp. 425–440 (2014)
- Ştefănescu, A., Park, D., Yuwen, S., Li, Y., Roşu, G.: Semantics-based program verifiers for all languages. In: OOPSLA 2016 (to appear)
- 11. Ellison, C., Roşu, G.: An executable formal semantics of C with applications. In: POPL 2012. pp. 533–544 (2012)
- Hathhorn, C., Ellison, C., Rosu, G.: Defining the Undefinedness of C. In: PLDI 2015. pp. 336–345 (2015). https://doi.org/10.1145/2737924.2737979
- 13. J, H.: Recherches sur la théorie de la démonstration. Logical Writings, in W.D. Goldfarb (1971)
- Lucanu, D., Rusu, V., Arusoaie, A., Nowak, D.: Verifying reachability-logic properties on rewriting-logic specifications. In: Logic, Rewriting, and Concurrency Essays dedicated to José Meseguer on the Occasion of His 65th Birthday. LNCS, vol. 9200, pp. 451–474. Springer (2015)
- 15. Lukasiewicz, J.: The shortest axiom of the implicational calculus of propositions. Proceedings of the Royal Irish Academy. Section A: Mathematical and Physical Sciences **52**, 25–33 (1948), http://www.jstor.org/stable/20488489
- Martelli, A., Montanari, U.: An efficient unification algorithm. ACM Transactions on Programming Languages and Systems 4(2), 258–282 (Apr 1982). https://doi.org/10.1145/357162.357169
- 17. Moore, B., Peña, L., Roşu, G.: Program verification by coinduction. In: 27th European Symposium on Programming (ESOP) (April 2018)
- 18. de Moura, L.M., Bjørner, N.: Z3: An Efficient SMT Solver. In: TACAS 2008. pp. 337–340 (2008)
- Park, D., Ştefănescu, A., Roşu, G.: KJS: A complete formal semantics of JavaScript. In: PLDI 2015. pp. 346–356 (2015)
- 20. Park, D., Zhang, Y., Saxena, M., Daian, P., Roşu, G.: A Formal Verification Tool for Ethereum VM Bytecode. In: Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'18). ACM (November 2018)
- 21. Robinson, J.A.: A machine-oriented logic based on the resolution principle. J. ACM 12(1), 23–41 (Jan 1965). https://doi.org/10.1145/321250.321253
- 22. Roşu, G.: Matching logic. Logical Methods in Computer Science **13**(4), 1–61 (December 2017). https://doi.org/http://arxiv.org/abs/1705.06312
- 23. Roşu, G., Ştefănescu, A.: From Hoare Logic to Matching Logic Reachability. In: FM 2012. LNCS, vol. 7436, pp. 387–402 (2012)
- Rosu, G.: Matching logic extended abstract (invited talk). In: 26th International Conference on Rewriting Techniques and Applications, RTA 2015, June 29 to July 1, 2015, Warsaw, Poland. pp. 5–21 (2015). https://doi.org/10.4230/LIPIcs.RTA.2015.5
- 25. Rosu, G., Stefanescu, A.: Matching logic: a new program verification approach. In: Proceedings of the 33rd International Conference on Software Engineering,

- ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011. pp. 868–871 (2011). https://doi.org/10.1145/1985793.1985928
- 26. Rosu, G., Stefanescu, A., Ştefan Ciobâcă, Moore, B.M.: One-path reachability logic. In: 28th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2013, New Orleans, LA, USA, June 25-28, 2013. pp. 358–367 (2013). https://doi.org/10.1109/LICS.2013.42
- Rusu, V., Arusoaie, A.: Proving reachability-logic formulas incrementally. In: Rewriting Logic and Its Applications - 11th International Workshop, WRLA 2016, Held as a Satellite Event of ETAPS, Eindhoven, The Netherlands, April 2-3, 2016, Revised Selected Papers. pp. 134–151 (2016)