Proof-Carrying Parameters in Certified Symbolic Execution: The Case Study of Antiunification

Andrei Arusoaie Dorel Lucanu
Faculty of Computer Science
Alexandru Ioan Cuza, University of Iași, România
{arusoaie.andrei,dlucanu}@info.uaic.ro

Symbolic execution uses various algorithms (matching, (anti)unification), whose executions are parameters for proof object generation. This paper proposes a generic method for generating proof objects for such parameters. We present in detail how our method works for the case of antiunification. The approach is accompanied by an implementation prototype, including a proof object generator and a proof object checker. In order to investigate the size of the proof objects, we generate and check proof objects for inputs inspired from the K definitions of C and Java.

1 Introduction

K (`https://kframework.org`) is a well established framework for programming languages, which brings a different perspective of what such a framework should be. K provides means to give formal definitions for programming languages and aims to automatically derive a series of practical tools for those languages: a parser, an interpreter, a debugger, a symbolic execution tool, a deductive verifier, a model-checker, and others. A formal semantics of a language defined in K consists of syntax declarations, a language configuration, and a set of rewriting rules. The configuration is a constructor term which holds the semantical information needed to execute programs (e.g., the code, environment, stack, program counter, etc). The rewriting rules are pairs $\varphi \Rightarrow \varphi'$ of program configurations with variables which specify how program configurations transit to other program configurations. An example of a tool automatically generated by K is the interpreter, which works as follows: the user provides a concrete configuration (which includes the program and an initial state of that program) and K applies rewriting rules as much as possible to this configuration. Another tool is the K prover, which uses symbolic execution to prove reachability properties.

The theoretical foundation of K is Matching Logic [20, 8] (hereafter shorthanded as ML), a logical framework where the formal definitions of program languages [12, 13, 17] and program reasoning [11, 23, 22, 15, 4] can be done in a uniform way. ML formulas are called *patterns* and they are used to uniformly specify syntax and semantics of programming languages, and the properties of program executions. ML has a *pattern matching semantics*: a pattern is interpreted as the set of elements that *match* it. For example, if a pattern t encodes a symbolic program configuration (with variables), then t is interpreted as the set of concrete program configurations that match it.

ML has a minimal, but expressive, syntax. For example, if one wants to specify configurations t that satisfy a first-order constraint ϕ, then $t \wedge \phi$ is the pattern that captures this intent. Not only constraints can be attached to patterns. Conjunctions $t_1 \wedge t_2$ (and disjunctions $t_1 \vee t_2$) are interpreted as the intersection (and union, respectively) of the elements that match t_1 and t_2. Moreover, it is easy to explain K rules $\varphi \Rightarrow \varphi'$ as implications of patterns $\varphi \rightarrow \bullet\varphi'$: the program configurations that match φ can transit in

Vlad Rusu (Ed.) Sixth Working Formal Methods Symposium (FROM 2022)
EPTCS 369, 2022, pp. 1–16, doi:10.4204/EPTCS.369.1

one step to program configurations that match φ', where $\bullet$ is a special symbol used to specify one step transitions.

ML is equipped with a sound proof system which can derive sequents of the form $\Gamma \vdash \varphi$, where φ is a pattern and Γ is an ML *theory* (i.e., a set of axiom patterns). The ML proof system is the key ingredient of another tool that K aims to generate: a deductive verifier.

Motivation. A fair question that needs to be posed is *how can we trust the proofs produced by the deductive verifier generated by K?* Given the size of the K codebase (about half a million lines of code [6]) and its dynamics (new code committed every week), the formal verification of the implementation of K is out of question. The solution here is to do what other formal verification tools do: instrument K so that its automatically generated tools produce *proof objects* that can be independently checked by a trusted kernel. In our context, proof objects are just proofs that use the ML proof system.

It turns out that all these tools that K aims to generate share several components. For example, *matching* algorithms are useful for concrete execution (interpreter), while *unification* and *antiunification* algorithms are needed for symbolic execution and program verification. Therefore, we can have a uniform approach: if we can find proof object generation mechanisms for each component, then we can simply instantiate those mechanisms whenever needed.

Symbolic execution is a key component in program verification and it has been used in K as well (e.g., [1, 14, 11]). Generating proof objects for symbolic execution is difficult because the parameters of an execution step must carry more proof information than in the concrete executions case. First, instead of matching, proof parameters must include unification information. Second, path conditions need to be carried along the execution.

In ML, there is a natural way to deal with symbolic execution. ML patterns φ have a *normal form* $t \wedge \phi$, where t is a term pattern and ϕ is a predicate pattern, expressing a constraint on variables in t. In particular, t can be the program configuration and ϕ the path condition. Patterns $t \wedge \phi$ are evaluated to the set of values that *match* t and satisfy ϕ. To compute the symbolic successors of a pattern, say $t \wedge \phi$, with respect to a rule, say $t_1 \wedge \phi_1 \Rightarrow t_2 \wedge \phi_2$, we need to unify the patterns $t \wedge \phi$ and $t_1 \wedge \phi_1$. Because unification can be expressed as a conjunction in ML [2, 20], we can say that only the states matched by $(t \wedge \phi) \wedge (t_1 \wedge \phi_1) \equiv (t \wedge t_1) \wedge (\phi \wedge \phi_1)$ transit to states matched by $t_2 \wedge \phi_2$. Expressing unification as a conjunction $(t \wedge t_1)$ is a nice feature of ML, but, in practice, unification algorithms are still needed to compute the general unifying substitution since it is used in symbolic successor patterns. The symbolic successors are obtained by applying the unifying substitution to the right-hand side of a rule (e.g., $t_2 \wedge \phi_2$) and adding the substitution (as an ML formula) to the path condition. Also, unification algorithms are being used to normalise conjunctions of the form $t \wedge t_1$, so that they consist of only one term and a constraint, $t' \wedge \phi'$. Therefore, unification algorithms are parameters of the symbolic execution steps and they must be used to generate the corresponding proof objects.

It is often the case when more than one rule can be applied during symbolic execution. For instance, if an additional rule $t'_1 \wedge \phi'_1 \Rightarrow (t'_2 \wedge \phi'_2)$ can be applied to $t \wedge \phi$, then the set of target states must match $t_2 \wedge \phi_2$ or $t'_2 \wedge \phi'_2$. This set of states is matched by the disjunction $(t_2 \wedge \phi_2) \vee (t'_2 \wedge \phi'_2)$. For the case when $\phi_2 \wedge \phi'_2$ holds, the disjunction reduces to $t_2 \vee t'_2$, which is not a normal form but it can be normalised using antiunification.

Related work. The literature on (anti)unification is vast. Due to the space limit, we recall here the closest related work which addresses proof object generation for concrete and symbolic executions, to understand the context of our work. In [6], the authors propose a method to generate proof objects for program executions $\varphi_{init} \Rightarrow \varphi_{final}$, where φ_{init} is the formula that specifies the initial state of the execu-

tion, φ_{final} specifies the final state, and "$\Rightarrow$" states the rewriting/reachability relation between states. The correctness of an execution,

$$\Gamma \vdash \varphi_{init} \Rightarrow \varphi_{final},$$

is witnessed by a formal proof, which uses the ML proof system. The K interpreter computes the parameters (e.g., execution traces, matching info) needed to generate the proof object.

In [20], the author shows that unification in ML can be represented as a conjunction of ML patterns. A first step into generating proof objects for unification was done in [2]. We proposed a method to normalise conjunctions of patterns $t_1 \wedge t_2$. The K implementation works with patterns in normal form $t \wedge \phi$, which are more efficient: matching/unification algorithms are executed only once on normalised patterns, rather than multiple times on patterns having multiple structural components (e.g., $t_1 \wedge t_2$). In [2], we use the syntactic unification algorithm [16] to (1) find an equivalent normal form $t \wedge \phi$ for conjunctions $t_1 \wedge t_2$, and (2) to generate proof objects for the equivalence between $t \wedge \phi$ and $t_1 \wedge t_2$. The unification algorithm provides the needed parameters (e.g., unifying substitutions) for proof generation.

Contributions. A lesson that we learned from [2] and [6] is that the algorithms implemented in various components of K can be used to compute the parameters needed to generate proof objects. In this paper we address the problem of *generating proof objects for antiunification*, which is used, e.g., in symbolic execution and verification. In ML, the *least general generalisation* of two term patterns t_1 and t_2 is given by their disjunction $t_1 \vee t_2$. We use Plotkin's antiunification algorihm [18, 19] to find normal forms $t \wedge \phi$ for disjunctions $t_1 \vee t_2$, and to generate proof objects for the equivalences between $t \wedge \phi$ and $t_1 \vee t_2$. The execution of the antiunification algorithm provides the parameters (intermediate generalisations and substitutions computed at each step) to generate the proof objects. Our contributions are:

1. We express Plotkin's antiunification algorithm in ML terms and we show that its steps produce equivalent patterns (Lemma 1 and Theorem 2).
2. We propose a proof object generation mechanism for the equivalences computed by the algorithms used in symbolic execution and verification, and we show how it works in the case of antiunification (Section 4).
3. We provide a prototype implementation of our proof object generation mechanism and a proof checker (Section 5);
4. We test our prototype on interesting examples, including inputs inspired from the K definitions of C [12, 13] and Java [5].

Indeed, the most challenging part of this work is the proof object generation mechanism. The main difficulty was to find the right proof object schema generation that precisely captures one step of the antiunification algorithm. Another tricky part was to design the proof object schema so that the proofs generated for each step can be easily composed. The size of the resulted proofs depends on the number of steps performed by the antiunification algorithm.

Paper organisation. Section 2 presents ML, its proof system and the ML theory for many-sorted term algebras. In Section 3 we present antiunification in a ML setting, and we prove that Plotkin's antiunification can be safely used to normalise disjunctions of term patterns. Our proof object generation methodology is presented in Section 4. The prototype implementation is described in Section 5 and we conclude in Section 6.

2 Matching Logic

Matching logic (ML) [20, 9, 8] started as a logic over a particular case of constrained terms [21, 23, 4, 15], but now it is developed as a full logical framework. We recall from [8] the definitions and notions that we use in this paper.

A matching logic *signature* is a triple $(EVar, SVar, \Sigma)$, where $EVar$ is a set of *element variables* $x, y, \ldots$, $SVar$ is a set of *set variables* $X, Y, \ldots$, and Σ is a set of *constant symbols* (or *constants*). The set PATTERN of Σ-*patterns* is generated by the grammar below, where $x \in EVar$, $X \in SVar$, and $\sigma \in \Sigma$:

$$\varphi ::= x \mid X \mid \sigma \mid \varphi_1\ \varphi_2 \mid \bot \mid \lceil \varphi \rceil \mid \varphi_1 \to \varphi_2 \mid \exists x.\varphi \mid \mu X.\varphi \text{ if } \varphi \text{ is positive in } X$$

A pattern φ is *positive* in X if all free occurrences of X in φ are under an even number of negations. The patterns below are derived constructs:

$$\begin{array}{lll} \top \equiv \neg\bot & \lfloor \varphi \rfloor \equiv \neg\lceil\neg\varphi\rceil & \varphi_1 \vee \varphi_2 \equiv \neg\varphi_1 \to \varphi_2 \\ \neg\varphi \equiv \varphi \to \bot & \varphi_1 = \varphi_2 \equiv \lfloor \varphi_1 \leftrightarrow \varphi_2 \rfloor & \varphi_1 \wedge \varphi_2 \equiv \neg(\neg\varphi_1 \vee \neg\varphi_2) \\ \forall x.\varphi \equiv \neg\exists x.\neg\varphi & \varphi_1 \neq \varphi_2 \equiv \neg(\varphi_1 = \varphi_2) & \varphi_1 \leftrightarrow \varphi_2 \equiv (\varphi_1 \to \varphi_2) \wedge (\varphi_2 \to \varphi_1) \end{array}$$

Example 1. *Let* $\Sigma = \{zero, succ, nil, cons\}$ *be an* ML *signature. Then x, zero, succ zero, succ x,* $\exists x.zero = x$, $\mu X.zero \vee (succ\, X)$ *are examples of* ML *patterns*[1].

ML has a pattern matching semantics where patterns are interpreted on a given carrier set, say M. Each pattern is interpreted as the set of elements that match it. *Element variables* x are matched by a singleton set, while *set variables* X are matched by a subset of M. The pattern $\bot$ is matched by the empty set (and hence $\top$ by M). The implication pattern $\varphi_1 \to \varphi_2$ is matched by the elements that do not match φ_1 or match φ_2. A pattern $\exists x.\varphi$ is matched by the instances of φ when x ranges over M. In particular, $\exists x.x$ is matched by M. Note that $\exists$ binds only element variables. Symbols σ (e.g. *zero*, *succ*) are interpreted as subsets $\sigma_M \subseteq M$, and, usually, the needed interpretation for them is obtained using axioms. For instance, the pattern $\exists x.zero = x$ is matched by M if $zero_M$ is a singleton, and by $\bot$ otherwise. This type of pattern is often used as axiom to restrict the interpretation of symbols to singletons. The pattern $\varphi_1\ \varphi_2$ is an *application* and its interpretation is given by means of a function $M \times M \to \mathscr{P}(M)$, which is pointwise extended to a function $\mathscr{P}(M) \times \mathscr{P}(M) \to \mathscr{P}(M)$. Applications are useful to build various structures or relations. For instance, $\forall x.\exists y.\, succ\, x = y$ says that *succ* has a functional interpretation (recall that the element variable y is matched by a singleton set). Applications are left associative. The pattern $\mu X.\varphi$ is matched by the least fixpoint of the functional defined by φ when X ranges $\mathscr{P}(M)$. An example is $\mu X.(zero \vee succ\, X)$, which is matched by the natural numbers $\mathbb{N}$ (up to a surjection), when both *zero* and *succ* have a functional interpretation (as above). Note that μ binds only set variables in *positive* patterns. The pattern $\lceil\varphi\rceil$ is called *definedness*[2] and it is matched by M if φ is matched at least by one element, and by $\emptyset$ otherwise. Such patterns are called *predicate patterns*.

The syntax priorities of the ML constructs is given by this ordered list:

$$\neg\_, \lceil\_\rceil, \lfloor\_\rfloor, \_ = \_, \_ \wedge \_, \_ \vee \_, \_ \to \_, \_ \leftrightarrow \_, \exists\_.\_, \forall\_.\_, \mu\_.\_,$$

where $\neg$ has the highest priority and $\mu\_.\_$ has the lowest priority. By convention, the scope of the binders extends as much as possible to the right, and parentheses can be used to restrict the scope of the binders. We often write $\varphi[\psi/x]$ and $\varphi[\psi/X]$ to denote the pattern obtained by substituting all free occurrences of x and X, respectively, in φ for ψ. In order to avoid variable capturing, we consider that α-renaming happens implicitly.

[1] Of course, *succ nil* or *cons nil zero* are also ML patterns but these can be handled properly using sorts (see Section 2.1).

[2] For convenience, we introduce it directly in the syntax of patterns but it can be axiomatised as in [20].

Hilbert-style proof system	
PROPOSITIONAL	φ, if φ is a propositional tautology over patterns
MODUS PONENS	$\dfrac{\varphi_1 \quad \varphi_1 \to \varphi_2}{\varphi_2}$
$\exists$-QUANTIFIER	$\varphi[y/x] \to \exists x.\varphi$
$\exists$-GENERALISATION	$\dfrac{\varphi_1 \to \varphi_2}{(\exists x.\varphi_1) \to \varphi_2}$ if $x \notin free(\varphi_2)$
PROPAGATION$_\bot$	$C[\bot] \to \bot$
PROPAGATION$_\vee$	$C[\varphi_1 \vee \varphi_2] \to C[\varphi_1] \vee C[\varphi_2]$
PROPAGATION$_\exists$	$C[\exists x.\varphi] \to \exists x.C[\varphi]$ if $x \notin free(C)$
FRAMING	$\dfrac{\varphi_1 \to \varphi_2}{C[\varphi_1] \to C[\varphi_2]}$
SET VARIABLE SUBSTITUTION	$\dfrac{\varphi}{\varphi[\psi/X]}$
PRE-FIXPOINT	$\varphi[\mu X.\varphi/X] \to \mu X.\varphi$
KNASTER-TARSKI	$\dfrac{\varphi[\psi/X] \to \psi}{\mu X.\varphi \to \psi}$
EXISTENCE	$\exists x.x$
SINGLETON	$\neg(C_1[x \wedge \varphi] \wedge C_2[x \wedge \neg\varphi])$

Figure 1: The Hilbert-style ML proof system.

The ML *proof system* [8] is shown in Figure 1. It contains four categories of rules: propositional tautologies, frame reasoning over application contexts, standard fixpoint reasoning, and two rules needed for completeness. An *application context* C is a pattern with a distinguished placeholder variable $\Box$ s.t. the path from the root of C to $\Box$ has only applications. $C[\varphi/\Box]$ is a shorthand for $C[\varphi]$ and $free(\varphi)$ denotes the set of free variables in φ.

2.1 ML Specification of the Term Algebra

A complete ML axiomatization of the many-sorted term algebra is given in [7] and we briefly recall it in Figure 2. The specification SORTS introduces symbols for sorts and their inhabitant sets, and some usual notations for them. The specification MSA includes the axioms corresponding to a given algebraic signature. Finally, TERM(S, F) includes the properties "no confusion" and "no junk" (inductive domains) that characterizes the (initial) term algebra."No confusion" says that the function symbols F are constructors:

- two different constructors will define different terms (NOCONFUSION I);
- a constructor is injective (i.e., the same constructors with different arguments will define different terms), and this is captured by (NOCONFUSION II).

The "no junk" property says that all inhabitants of a sort are generated using the constructors F, and this is captured by the axiom (INDUCT. DOMAIN). For the sake of presentation, this axiom does not include the case of the mutual recursive sorts[3].

Theorem 1 ([7])**.** *The specification* MSA(S,F) *captures the many-sorted* (S,F)*-algebras in the following sense:*

[3]See [8] for a complete definition.

spec SORTS
Symbol: $inh, Sort$
Notation:
$[\![s]\!] \equiv inh\ s$
$\forall x{:}s.\ \varphi \equiv \forall x.\ x \in [\![s]\!] \to \varphi$
$\exists x{:}s.\ \varphi \equiv \exists x.\ x \in [\![s]\!] \wedge \varphi$
$\varphi{:}s \equiv \exists z{:}s.\ \varphi = z$
$\forall x_1,\ldots,x_n{:}s.\ \varphi \equiv \forall x_1{:}s \ldots \forall x_n{:}s.\ \varphi$
$\exists x_1,\ldots,x_n{:}s.\ \varphi \equiv \exists x_1{:}s \ldots \exists x_n{:}s.\ \varphi$
endspec

SORTS

spec MSA(S,F)
Import: SORTS
Symbol: $s \in S$, $f \in F$
Notation:
$f(\varphi_1,\ldots,\varphi_n) \equiv f\,\varphi_1 \ldots \varphi_n$
Axiom:
(SORT) $s{:}Sort$ for each $s \in S$
(NONEMPTY) $[\![s]\!] \neq \bot$ for each $s \in S$
(FUNCTION) $\forall x_1{:}s_1 \ldots \forall x_n{:}s_n.(f\,x_1 \ldots x_n){:}s$ for each $f \in F_{s_1\ldots s_n,s}$
endspec

MSA

spec $TERM(S,F)$
Import: MSA(S,F)
Axiom:
(NOCONFUSION I) $f \neq f' \to \forall x_1{:}s_1 \ldots \forall x_n{:}s_n.\forall x'_1{:}s'_1 \ldots \forall x'_m{:}s'_m.f\,x_1\,..\,x_n \neq f'\,x'_1\,..\,x'_m$
(NOCONFUSION II) $\forall x_1,x'_1{:}s_1 \ldots. \forall x_n,x'_n{:}s_n.(f\,x_1 \cdots x_n) = (f\,x'_1 \cdots x'_n) \to (x_1{=}x'_1) \wedge \cdots \wedge (x_n{=}x'_n)$
(INDUCT. DOMAIN) $[\![s]\!] = \mu X.\ \bigvee_{f:s_1\ldots s_n,s} f\,Y_1 \cdots Y_n$, where $Y_i = \begin{cases} X, & \text{if } s_i = s \\ [\![s_i]\!], & \text{otherwise} \end{cases}$
endspec

TERM(S,F)

Specification 2: ML specifications for sorts, many-sorted algebras, and term algebra

- *from each* MSA(S,F)*-model* M *we may extract an* (S,F)*-algebra* $\alpha(M)$*, and*
- *for each* (S,F)*-algebra* A *there is an* MSA(S,F)*-model* M *s.t.* $\alpha(M) = A$.

If M *is a* $TERM(S,F)$*-model, then* $\alpha(M)$ *is the term* (S,F)*-algebra (up to isomorphism).*

Proposition 1. *The next patterns are semantical consequences of* $TERM(S,F)$*:*

$$\exists z.t \wedge (z = u) \leftrightarrow t[u/z] \qquad \text{if } z \notin var(u)$$
$$z = (f\,\bar{t}) \leftrightarrow \exists \bar{y}.z = (f\,\bar{y}) \wedge \bar{y} = \bar{t} \quad \text{if } \bar{y} \notin var\big((f\,\bar{t})\big) \cup \{z\}$$

The equivalences in Proposition 1 are later used as macro rules for proof object generation. The notation $(f\,\bar{t})$ means $(f\,t_1 \ldots t_n)$. We also use $\exists\bar{y}$ or $\exists\{y_1,\ldots,y_n\}$ instead of $\exists y_1 \ldots \exists y_n$. The equality $\bar{y} = \bar{t}$ is sugar syntax for $\bigwedge_{i=1}^{n} y_i = t_i$.

3 Antiunification in ML

Antiunification is a process dual to unification [16] that computes a *generalisation* t of two input terms t_1 and t_2. A term t is an *antiunifier* of t_1 and t_2 if there are two substitutions σ_1 and σ_2 such that $t\sigma_1 = t_1$ and $t\sigma_2 = t_2$. There is at least one antiunifier for any t_1 and t_2: we can always choose a variable $x \notin var(t_1,t_2)$ and substitutions $\sigma_1 = \{x \mapsto t_1\}$ and $\sigma_2 = \{x \mapsto t_2\}$ s.t. $x\sigma_1 = t_1$ and $x\sigma_2 = t_2$.

A term t' is more *general* than a term t if there is a substitution σ such that $t'\sigma = t$. Given t_1 and t_2, their *least general* antiunifier t satisfies: for any antiunifier t' of t_1 and t_2 we have that t is less general than t' (a.k.a., *least general generalisation*, shorthanded as *lgg*).

Remark 1. *In other words, t is less general than t' iff the set of ground instances of t is included in that of t'. In terms of matching logic, this can be expressed by $\exists var(t).t \subseteq \exists var(t').t'$, where $\varphi_1 \subseteq \varphi_2$ is defined as $\lfloor \varphi_1 \to \varphi_2 \rfloor$.*

Now we present Plotkin's antiunification algorithm [19] for computing the *lgg* over ML term patterns. First, we define *antiunification problems*:

Definition 1. *An* antiunification problem *is a pair $\langle t, P\rangle$ consisting of a term pattern t and a non-empty set P of elements of the form $z \mapsto u \sqcup v$, where z is a variable, and u and v are term patterns.*

Plotkin's algorithm [19] for computing the *lgg* consists in applying a decomposition rule over antiunification problems as much as possible:

$$\langle t, P \cup \{z \mapsto (f\, u_1 \dots u_n) \sqcup (f\, v_1 \dots v_n)\}\rangle \rightsquigarrow \langle t[(f\, z_1 \dots z_n)/z], P \cup \{z_1 \mapsto u_1 \sqcup v_1, \dots, z_n \mapsto u_n \sqcup v_n\}\rangle,$$

where $z_1, \dots, z_n$ are fresh variables. If we want to compute the *lgg* of t_1 and t_2, we build the initial antiunification problem $\langle z, \{z \mapsto t_1 \sqcup t_2\}\rangle$ with $z \notin var(t_1) \cup var(t_2)$ and we apply Plotkin's rule repeatedly. When this rule cannot be applied anymore, we say that the obtained antiunification problem $\langle t', P'\rangle$ is in *solved form*. The obtained t' is the *lgg* of t_1 and t_2, while P' defines the two substitutions $\sigma_1 = \{z \mapsto u \mid z \mapsto u \sqcup v \in P'\}$ and $\sigma_2 = \{z \mapsto v \mid z \mapsto u \sqcup v \in P'\}$ such that $t'\sigma_1 = t_1$ and $t'\sigma_2 = t_2$. Note that the pairs $u \sqcup v$ are not commutative.

Example 2. *Let $t_1 = (cons\,(succ\,x_1)\,(cons\,zero\,l_1))$ and $t_2 = (cons\,x_2\,(cons\,(succ\,x_2)\,l_2))$. Using Plotkin's algorithm on the input $\langle z, \{z \mapsto t_1 \sqcup t_2\}\rangle$ (note that z is fresh w.r.t. $var(t_1) \cup var(t_2)$) we obtain:*

$$\begin{aligned}
&\langle z, \{z \mapsto t_1 \sqcup t_2\}\rangle = \\
&\langle z, \{z \mapsto (cons\,(succ\,x_1)\,(cons\,zero\,l_1)) \sqcup (cons\,x_2\,(cons\,(succ\,x_2)\,l_2))\}\rangle \rightsquigarrow \\
&\langle z[(cons\,z_1\,z_2)/z], \{z_1 \mapsto (succ\,x_1) \sqcup x_2, z_2 \mapsto (cons\,zero\,l_1) \sqcup (cons\,(succ\,x_2)\,l_2)\}\rangle = \\
&\langle (cons\,z_1\,z_2), \{z_1 \mapsto (succ\,x_1) \sqcup x_2, z_2 \mapsto (cons\,zero\,l_1) \sqcup (cons\,(succ\,x_2)\,l_2)\}\rangle \rightsquigarrow \\
&\langle (cons\,z_1\,z_2)[(cons\,z_3\,z_4)/z_2], \{z_1 \mapsto (succ\,x_1) \sqcup x_2, z_3 \mapsto zero \sqcup (succ\,x_2), z_4 \mapsto l_1 \sqcup l_2\}\rangle = \\
&\langle (cons\,z_1\,(cons\,z_3\,z_4)), \{z_1 \mapsto (succ\,x_1) \sqcup x_2, z_3 \mapsto zero \sqcup (succ\,x_2), z_4 \mapsto l_1 \sqcup l_2\}\rangle \not\rightsquigarrow
\end{aligned}$$

The lgg of the term patterns t_1 and t_2 is the term pattern $t \triangleq (cons\,z_1\,(cons\,z_3\,z_4))$ while the substitutions $\sigma_1 = \{z_1 \mapsto (succ\,x_1), z_3 \mapsto zero, z_4 \mapsto l_1\}$ and $\sigma_2 = \{z_1 \mapsto x_2, z_3 \mapsto (succ\,x_2), z_4 \mapsto l_2\}$ satisfy $t\sigma_1 = t_1$ and $t\sigma_2 = t_2$. The generated variables z_1, z_2, z_3, z_4 occur at most once in the computed lgg, and $var(t) = dom(\sigma_1) = dom(\sigma_2)$.

The $\rightsquigarrow^!$ in Example 2 means that $\rightsquigarrow$ has been applied repeatedly until $\langle t, P\rangle$ is in solved form.

Antiunification problems are encoded as ML patterns as below:

Definition 2 (Antiunification problem)**.** *For each antiunification problem $\langle t, P\rangle$ we define an* ML *pattern*

$$\phi^{\langle t,P\rangle} \triangleq \exists \overline{z}.t \wedge \left(\phi^{\sigma_1} \vee \phi^{\sigma_2}\right),$$

where $\sigma_1 = \{z \mapsto u \mid z \mapsto u \sqcup v \in P\}$, $\sigma_2 = \{z \mapsto v \mid z \mapsto u \sqcup v \in P\}$, and $var(t) = dom(\sigma_1) = dom(\sigma_2) = \overline{z}$.

Example 3. *Here are the corresponding encodings for the intermediate antiunification problems that are generated during the execution shown in Example 2:*

1. *$\langle z, \{z \mapsto (cons\,(succ\,x_1)\,(cons\,zero\,l_1)) \sqcup (cons\,x_2\,(cons\,(succ\,x_2)\,l_2))\}\rangle$ is encoded as $\exists z.z \wedge \big(z = (cons\,(succ\,x_1)\,(cons\,zero\,l_1)) \vee z = (cons\,x_2\,(cons\,(succ\,x_2)\,l_2))\big)$;*

2. $\langle (cons\, z_1\, z_2), \{z_1 \mapsto (succ\, x_1) \sqcup x_2, z_2 \mapsto (cons\, zero\, l_1) \sqcup (cons\, (succ\, x_2)\, l_2)\}\rangle$ *is encoded as:*
$\exists\{z_1, z_2\}.(cons\, z_1\, z_2) \wedge \Big(\big(z_1 = (succ\, x_1) \wedge z_2 = (cons\, zero\, l_1)\big) \vee \big(z_1 = x_2 \wedge z_2 = (cons\, (succ\, x_2)\, l_2)\big)\Big)$;

3. $\langle (cons\, z_1\, (cons\, z_3\, z_4)), \{z_1 \mapsto (succ\, x_1) \sqcup x_2, z_3 \mapsto zero \sqcup (succ\, x_2), z_4 \mapsto l_1 \sqcup l_2\}\rangle$ *is encoded as:*
$\exists\{z_1, z_3, z_4\}.(cons\, z_1\, (cons\, z_3\, z_4)) \wedge$
$\Big(\big(z_1 = (succ\, x_1) \wedge z_3 = zero \wedge z_4 = l_1\big) \vee \big(z_1 = x_2 \wedge z_3 = (succ\, x_2) \wedge z_4 = l_2\big)\Big)$.

Note that the encodings shown in Example 3 are all equivalent. Also, remember that the scope of the quantifiers extends as much as possible to the right.

Lemma 1. *If* $\langle t_i, P_i\rangle \rightsquigarrow \langle t_{i+1}, P_{i+1}\rangle$ *is a step performed using Plotkin's antiunification rule, then*

$$TERM(S,F) \models \phi^{\langle t_i, P_i\rangle} \leftrightarrow \phi^{\langle t_{i+1}, P_{i+1}\rangle}.$$

The soundness theorem shown below is a direct consequence of Lemma 1:

Theorem 2. (Soundness) *Let* t_1 *and* t_2 *be two term patterns and* z *a variable such that* $z \notin var(t_1) \cup var(t_2)$. *If* $\langle z, \{z \mapsto t_1 \sqcup t_2\}\rangle \rightsquigarrow^! \langle t, P\rangle$, *then* $TERM(S,F) \models (t_1 \vee t_2) \leftrightarrow \phi^{\langle t, P\rangle}$.

The above results are proved using the semantical ML satisfaction relation ($\models$). Recall that our goal is to generate proof objects, and thus, we want to prove the above results using the ML proof system. We address this challenge in the following section.

4 Generating Proof Objects

Our method for generating proof objects is generic in the sense that it can be used for a larger class of term-algebra-based algorithms (e.g., unification, antiunification).

A proof object is represented by a sequence of lines of the form:

k	derived pattern	justification

where k is the step index and the justification mentions the applied inference rule and the step index of the premises of the rule (if any). The step index of the premises should be smaller than k, i.e., the premises are justified by the previous lines.

Our method is sketched as follows:

1. We consider algorithms that transform a pattern φ into an equivalent one φ'. So, a proof object *proofObj* has to be generated for $\varphi \leftrightarrow \varphi'$.
2. The execution of such algorithms for an input φ produces a sequence of intermediate patterns $\varphi_1, \ldots, \varphi_{n-1}$ such that $\varphi^{i-1} \leftrightarrow \varphi^i$, $0 < i \leq n$, where $\varphi_0 \triangleq \varphi$ and $\varphi_n \triangleq \varphi'$. So, a proof object for each $\varphi^{i-1} \leftrightarrow \varphi^i$ has to be generated in order to build *proofObj*.
3. Assuming that φ^i is obtained from φ^{i-1} by applying a generic step of the algorithm, we design a proof schema for this step s.t. the instance of this schema for φ^{i-1} and φ^i produces a proof object $proofObj_i$ for $\varphi^{i-1} \leftrightarrow \varphi^i$.

4. The proof object *proofObj* for the equivalence $\varphi \leftrightarrow \varphi'$ is obtained by the composition

$$proofObj_1; \ldots; proofObj_n; proofObj_{n+1},$$

where $proofObj_{n+1}$ connects the other proofs objects by transitivity so that the result is a proof object.

The approach from [2] for unification can be formalised now as an instance of this method.

4.1 Generating proof objects for antiunification

The goal is to generate proof objects for the equivalence $TERM(S,F) \models (t_1 \vee t_2) \leftrightarrow \phi^{\langle t,P\rangle}$ (cf. Theorem 2). For an input antiunification problem $\langle t_0, P_0\rangle \triangleq \langle z, z \mapsto t_1 \sqcup t_2\rangle$, Plotkin's algorithm generates a sequence of antiunification problems until it reaches the final pair $\langle t_k, P_k\rangle \triangleq \langle t, P\rangle$ which contains the *lgg*:

$$\langle t_0, P_0\rangle \rightsquigarrow \cdots \rightsquigarrow \langle t_i, P_i\rangle \rightsquigarrow \cdots \rightsquigarrow \langle t_k, P_k\rangle.$$

The key observation is that the ML encodings of the antiunification problems from the above sequence are all equivalent (cf. Lemma 1):

$$\phi^{\langle t_0,P_0\rangle} \leftrightarrow \cdots \leftrightarrow \phi^{\langle t_i,P_i\rangle} \leftrightarrow \cdots \leftrightarrow \phi^{\langle t_k,P_k\rangle}.$$

The proof object for $t_1 \vee t_2 \leftrightarrow \phi^{\langle t,P\rangle}$ is obtained by instantiating the next schema:

$$\vee_{gen}; (\rightsquigarrow_{step})^k; (\leftrightarrow_{tranz})^k,$$

where:

- k is the number of applications of Plotkin's rule;
- $\vee_{gen}$ is the proof schema which corresponds to the initial equivalence $t_1 \vee t_2 \leftrightarrow \phi^{\langle t_0,P_0\rangle}$. Recall that $\phi^{\langle t_0,P_0\rangle} = \phi^{\langle z, z\mapsto t_1\sqcup t_2\rangle} \triangleq \exists z.z \wedge (z = t_1 \vee z = t_2)$;
- $\rightsquigarrow_{step}$ is the proof schema corresponding to the equivalences $\phi^{\langle t_i,P_i\rangle} \leftrightarrow \phi^{\langle t_{i+1},P_{i+1}\rangle}$, with $i \in \{0,\ldots,k-1\}$. All these equivalences are obtained by applying a generic step of the algorithm. The schema $\rightsquigarrow_{step}$ corresponds to this generic step.
- Finally, using the transitivity of $\leftrightarrow$ k times we obtain a proof object for $t_1 \vee t_2 \leftrightarrow \phi^{\langle t_k,P_k\rangle}$.

The proof schema for $\vee_{gen}$ and $\rightsquigarrow_{step}$ are presented in Sections 4.3 and 4.4.3. Both use the proof rules in Figure 1 and the macro rules in Section 4.2. The proof schema for $\rightsquigarrow_{step}$ uses two additional (sub)schema $\exists$-GEN' (shown in Section 4.4.1) and DEC (shown in Section 4.4.2).

Example 4 shows a high-level proof object where we apply $\vee_{gen}$ once, $\rightsquigarrow_{step}$ and $\leftrightarrow_{tranz}$ twice. This is because the antiunification rule has been applied two times to obtain the *lgg*. The exact (low-level) proof object corresponding to this example is obtained by instantiating the proof schemata for $\vee_{gen}$ and $\rightsquigarrow_{step}$ (presented later in Sections 4.3 and 4.4.3).

Example 4. *We show here the proof object corresponding to the execution from Example 2, where we use the encodings in Example 3. The structure of the proof object follows the schema* $\vee_{gen}; (\rightsquigarrow_{step})^k; (\leftrightarrow_{tranz})^k$, *where* $k = 2$*:*

(1)	$t_1 \vee t_2 \leftrightarrow$ $\quad \exists z.z \wedge \big(z = (cons\,(succ\,x_1)\,(cons\,zero\,l_1)) \vee z = (cons\,x_2\,(cons\,(succ\,x_2)\,l_2))\big)$	$\vee_{gen}$
(2.1)	$\exists z.z \wedge$ $\quad \big(z = (cons\,(succ\,x_1)\,(cons\,zero\,l_1)) \vee z = (cons\,x_2\,(cons\,(succ\,x_2)\,l_2))\big) \leftrightarrow$ $\exists\{z_1,z_2\}.(cons\,z_1\,z_2) \wedge$ $\Big(\big(z_1 = (succ\,x_1) \wedge z_2 = (cons\,zero\,l_1)\big) \vee \big(z_1 = x_2 \wedge z_2 = (cons\,(succ\,x_2)\,l_2)\big)\Big)$	$\rightsquigarrow_{step}$
(2.2)	$\exists\{z_1,z_2\}.(cons\,z_1\,z_2) \wedge$ $\Big(\big(z_1 = (succ\,x_1) \wedge z_2 = (cons\,zero\,l_1)\big) \vee \big(z_1 = x_2 \wedge z_2 = (cons\,(succ\,x_2)\,l_2)\big)\Big) \leftrightarrow$ $\exists\{z_1,z_3,z_4\}.(cons\,z_1\,(cons\,z_3\,z_4)) \wedge$ $\Big(\big(z_1 = (succ\,x_1) \wedge z_3 = zero \wedge z_4 = l_1\big) \vee \big(z_1 = x_2 \wedge z_3 = (succ\,x_2) \wedge z_4 = l_2\big)\Big)$	$\rightsquigarrow_{step}$
(3.1)	$t_1 \vee t_2 \leftrightarrow$ $\exists\{z_1,z_2\}.(cons\,z_1\,z_2) \wedge$ $\Big(\big(z_1 = (succ\,x_1) \wedge z_2 = (cons\,zero\,l_1)\big) \vee \big(z_1 = x_2 \wedge z_2 = (cons\,(succ\,x_2)\,l_2)\big)\Big)$	$\leftrightarrow_{tranz}$: 1, 2.1
(3.2)	$t_1 \vee t_2 \leftrightarrow$ $\exists\{z_1,z_3,z_4\}.(cons\,z_1\,(cons\,z_3\,z_4)) \wedge$ $\Big(\big(z_1 = (succ\,x_1) \wedge z_3 = zero \wedge z_4 = l_1\big) \vee \big(z_1 = x_2 \wedge z_3 = (succ\,x_2) \wedge z_4 = l_2\big)\Big)$	$\leftrightarrow_{tranz}$: 3.1, 2.2

Generating proof objects for antiunification turns out to be more complex than in the unification case from [2]. Plotkin's algorithm generates fresh variables at each step. These variables are existentially quantified in the corresponding ML encodings and handling these quantifiers in proofs is difficult. The main difficulty comes from the fact that most of the times the application of a ML proof rule requires a lot of preparation work: you have to isolate the goal that can be proved using a particular ML proof rule, and then find a way to put back the existential quantifiers. Also, you have to make sure that the proof objects generated by $\rightsquigarrow_{step}$ remain composable, so that $\leftrightarrow_{tranz}$ can be applied. This is why we use several macro rules in addition to the ML proof system.

4.2 The Macro Rules

Our approach uses the additional rules shown in Figure 3[4]. The first part includes three macro rules. $\exists$-CTX enables the replacement of a formula with an equivalent one under the $\exists$ quantifier. $\exists$-SCOPE extends the scope of $\exists$ over formulas that do not contain variables that can be captured. $\exists$-COLLAPSE is useful when existentially quantified formulas can be collapsed under a single quantifier.

The second part includes two macro rules which are consequences of the specification $TERM(S,F)$ (cf. Proposition 1). $\exists$-SUBST states that $t \wedge (z = u)$ is equivalent to $t[u/z]$ under the existential quantifier which binds z. $\exists$-GEN allows one to replace subterms $\bar{t} \triangleq t_1 \ldots t_n$ of a term with existentially quantified fresh variables $\bar{y} \triangleq y_1 \ldots y_n$. To obtain an equivalent formula, the constraints $\bar{y} = \bar{t}$ are added.

4.3 Proof object schema $\vee_{gen}$

The first step of our method is to establish the equivalence between the disjunction $t_1 \vee t_2$ and the encoding of the initial unification problem $\langle z, z \mapsto t_1 \sqcup t_2 \rangle$, that is $\big(\exists z.z \wedge (z = t_1)\big) \vee \big(\exists z.z \wedge (z = t_2)\big)$. This is done via the proof object schema $\vee_{gen}$, which is shown below.

[4] Proving these rules using the proof system in Figure 1 is out of scope of this paper.

Additional proof rules	
	$\dfrac{\varphi_2 \leftrightarrow \varphi_2'}{\big(\exists \bar{x}.\varphi_1 \wedge \varphi_2\big) \leftrightarrow \exists \bar{x}.(\varphi_1 \wedge \varphi_2')}$
$\exists$-CTX	
$\exists$-SCOPE	$\big(\big(\exists \bar{x}.\varphi_1\big) \odot \varphi_2\big) \leftrightarrow \exists \bar{x}.(\varphi_1 \odot \varphi_2)$, if $\bar{x} \notin free(\varphi_2)$
$\exists$-COLLAPSE	$\big(\big(\exists \bar{x}.\varphi_1\big) \vee \big(\exists \bar{x}.\varphi_2\big)\big) \leftrightarrow \exists \bar{x}.(\varphi_1 \vee \varphi_2)$
Term algebra specific proof rules	
$\exists$-SUBST	$\exists z.t \wedge (z = u) \leftrightarrow t[u/z]$, if $z \notin var(u)$
$\exists$-GEN	$z = (f\,\bar{t}) \leftrightarrow \exists \bar{y}.z = (f\,\bar{y}) \wedge \bar{y} = \bar{t}$, if $\bar{y} \notin var\big((f\,\bar{t})\big) \cup \{z\}$

Figure 3: The macro rules used to generate proof objects for antiunification. The $\odot$ is a placeholder for one logical operator in the set $\{\wedge, \leftrightarrow\}$. Also, unless explicitly delimited using parentheses, the scope of the quantifiers extends as much as possible to the right.

To shorten our presentation, the MODUSPONENS rule from Figure 1 is applied here directly over a double implication $\leftrightarrow$ (instead of $\rightarrow$). For the steps k+6 and k+8 we use PROPOSITIONAL to justify two trivial equivalences: the first says that $\phi_1 \vee \phi_2 \leftrightarrow \phi_1' \vee \phi_2'$ if $\phi_1 \leftrightarrow \phi_1'$ and $\phi_2 \leftrightarrow \phi_2'$; the second is just a well-known distributivity property.

(k)	$\big(\exists z.t_1 \leftrightarrow z \wedge (z = t_1)\big)$	$\exists$-SUBST (note: $z[t_1/z] = t_1$)
(k+1)	$\big(\exists z.t_1 \leftrightarrow z \wedge (z = t_1)\big) \leftrightarrow \big(t_1 \leftrightarrow \exists z.z \wedge (z = t_1)\big)$	$\exists$-SCOPE
(k+2)	$t_1 \leftrightarrow \exists z.z \wedge (z = t_1)$	MODUSPONENS: k, k+1
(k+3)	$\big(\exists z.t_2 \leftrightarrow z \wedge (z = t_2)\big)$	$\exists$-SUBST (note: $z[t_2/z] = t_2$)
(k+4)	$\big(\exists z.t_2 \leftrightarrow z \wedge (z = t_2)\big) \leftrightarrow \big(t_2 \leftrightarrow \exists z.z \wedge (z = t_2)\big)$	$\exists$-SCOPE
(k+5)	$t_2 \leftrightarrow \exists z.z \wedge (z = t_2)$	MODUSPONENS: k+3, k+4
(k+6)	$t_1 \vee t_2 \leftrightarrow \big(\exists z.z \wedge (z = t_1)\big) \vee \big(\exists z.z \wedge (z = t_2)\big)$	PROPOSITIONAL: k+2, k+5
(k+7)	$\Big(\big(\exists z.z \wedge (z = t_1)\big) \vee \big(\exists z.z \wedge (z = t_2)\big)\Big) \leftrightarrow \exists z.\big(z \wedge (z = t_1)\big) \vee \big(z \wedge (z = t_2)\big)$	$\exists$-COLLAPSE
(k+8)	$\big(z \wedge (z = t_1)\big) \vee \big(z \wedge (z = t_2)\big) \leftrightarrow z \wedge \big((z = t_1) \vee (z = t_2)\big)$	PROPOSITIONAL
(k+9)	$\exists z.\big(z \wedge (z = t_1)\big) \vee \big(z \wedge (z = t_2)\big) \leftrightarrow \exists z.z \wedge \big((z = t_1) \vee (z = t_2)\big)$	$\exists$-CTX: k+8
(k+10)	$t_1 \vee t_2 \leftrightarrow \exists z.\big(z \wedge (z = t_1)\big) \vee \big(z \wedge (z = t_2)\big)$	$\leftrightarrow_{tranz}$: k+6, k+7
(k+11)	$t_1 \vee t_2 \leftrightarrow \exists z.z \wedge \big((z = t_1) \vee (z = t_2)\big)$	$\leftrightarrow_{tranz}$: k+10, k+9

4.4 Proof schema $\rightsquigarrow_{step}$

The proof schema of $\rightsquigarrow_{step}$ used two other (sub)schemata $\exists$-GEN' and DEC. We explain these first, and them we present the proof schema for $\rightsquigarrow_{step}$.

4.4.1 Proof object schema $\exists$-GEN'

Recall that $\exists$-GEN (Figure 3 – term algebra specific rule) establishes an equivalence between $z = (f\,\bar{t})$ and $\exists \bar{y}.z = (f\,\bar{y}) \wedge \bar{y} = \bar{t}$, if $\bar{y} \notin var\big((f\,\bar{t})\big) \cup \{z\}$, which basically describes a generalisation of $(f\,\bar{t})$. However, most of the times $\exists$-GEN is applied under a conjunction. The proof schema $\exists$-GEN$'$ generalises the

$\exists$-GEN macro rule as follows:

$$\big(\varphi \wedge z = (f\,\bar{t})\big) \leftrightarrow \exists \bar{z}.\varphi \wedge z = (f\,\bar{z}) \wedge \bar{z} = \bar{t},$$

where $(f\,\bar{t})$ denotes $(f\,t_1 \ldots t_n)$, $(f\,\bar{z})$ stands for $(f\,z_1 \ldots z_n)$, and $\bar{z} = \bar{t}$ denotes the conjunction $\bigwedge_{i=1}^{n} z_i = t_i$. Note that φ is safely introduced under the existential quantifier. The proof schema $\exists$-GEN' is shown below:

(k)	$z = (f\,\bar{t}) \leftrightarrow \exists \bar{z}.z = (f\,\bar{y}) \wedge \bar{y} = \bar{t}$	$\exists$-GEN
(k+1)	$\big(\varphi \wedge z = (f\,\bar{t})\big) \leftrightarrow$ $\varphi \wedge \exists \bar{z}.z = (f\,\bar{z}) \wedge \bar{z} = \bar{t}$	PROPOSITIONAL:k
(k+2)	$\big(\varphi \wedge \exists \bar{z}.z = (f\,\bar{z}) \wedge \bar{z} = \bar{t}\big) \leftrightarrow$ $\exists \bar{z}.\varphi \wedge z = (f\,\bar{z}) \wedge \bar{z} = \bar{t}$	$\exists$-SCOPE, $var(\varphi) \cap \{z_1, \ldots, z_n\} = \emptyset$
(k+3)	$\big(\varphi \wedge z = (f\,\bar{t})\big) \leftrightarrow$ $\exists \bar{z}.\varphi \wedge z = (f\,\bar{z}) \wedge \bar{z} = \bar{t}$	$\leftrightarrow_{tranz}$: k+1, k+2

At step $k+1$ we use PROPOSITIONAL, in particular, we use this property: if $\varphi_1 \leftrightarrow \varphi_2$ then $\varphi \wedge \varphi_1 \leftrightarrow \varphi \wedge \varphi_2$. This schema is applied in a certain context, where $var(\varphi) \cap \{z_1, \ldots, z_n\} = \emptyset$, because $z_1, \ldots, z_n$ are always fresh variables introduced by Plotkin's antiunification algorithm.

4.4.2 Proof schema DEC

Once we have equivalent forms for $z = (f\,\bar{t})$ (i.e., $\exists \bar{y}.z = (f\,\bar{y}) \wedge \bar{y} = \bar{t}$, cf. $\exists$-GEN'), we are now ready to tackle disjunctions $(f\,\bar{u}) \vee (f\,\bar{v})$. DEC captures a decomposition: $(f\,\bar{u}) \vee (f\,\bar{v})$ is equivalent to a conjunction between $(f\,\bar{z})$ and $(\bar{z} = \bar{u}) \vee (\bar{z} = \bar{v})$, where again $\bar{z} = \{z_1 \ldots z_n\}$ are existentially quantified. In addition, DEC performs the decomposition under a conjunction:

$$\big((\varphi \wedge z = (f\,\bar{u})) \vee (\varphi' \wedge z = (f\,\bar{v}))\big) \leftrightarrow \exists \bar{z}.z = (f\,\bar{z}) \wedge \big((\varphi \wedge \bar{z} = \bar{u}) \vee (\varphi' \wedge \bar{z} = \bar{v})\big),$$

where $(f\,\bar{u})$ means $(f\,u_1 \ldots u_n)$, $(f\,\bar{z})$ stands for $(f\,z_1 \ldots z_n)$, and the equality $\bar{z} = \bar{u}$ denotes $\bigwedge_{i=1}^{n} z_i = u_i$. Similarly, $(f\,\bar{v})$ denotes $(f\,v_1 \ldots v_n)$ and $\bar{z} = \bar{v}$ is $\bigwedge_{i=1}^{n} z_i = v_i$. The schema for DEC uses $\exists$-GEN':

(k)	$\big(\varphi \wedge z = (f\,\bar{u})\big) \leftrightarrow \exists \bar{z}.\varphi \wedge z = (f\,\bar{z}) \wedge \bar{z} = \bar{u}$	$\exists$-GEN'
(k+1)	$\big(\varphi' \wedge z = (f\,\bar{v})\big) \leftrightarrow \exists \bar{z}.\varphi' \wedge z = (f\,\bar{z}) \wedge \bar{z} = \bar{v}$	$\exists$-GEN'
(k+2)	$\big(\varphi \wedge z = (f\,\bar{u})\big) \vee \big(\varphi' \wedge z = (f\,\bar{v})\big) \leftrightarrow$ $\big(\exists \bar{z}.\varphi \wedge z = (f\,\bar{z}) \wedge \bar{z} = \bar{u}\big) \vee \exists \bar{z}.\varphi' \wedge z = (f\,\bar{z}) \wedge \bar{z} = \bar{v}$	PROPOSITIONAL: k+1, k+2
(k+3)	$\big(\exists \bar{z}.\varphi \wedge z = (f\,\bar{z}) \wedge \bar{z} = \bar{u}\big) \vee \exists \bar{z}.\varphi' \wedge z = (f\,\bar{z}) \wedge \bar{z} = \bar{v} \leftrightarrow$ $\exists \bar{z}.\big(\varphi \wedge z = (f\,\bar{z}) \wedge \bar{z} = \bar{u}\big) \vee \varphi' \wedge z = (f\,\bar{z}) \wedge \bar{z} = \bar{v}$	$\exists$-COLLAPSE
(k+4)	$\big(\varphi \wedge z = (f\,\bar{u})\big) \vee \big(\varphi' \wedge z = (f\,\bar{v})\big) \leftrightarrow$ $\exists \bar{z}.\big(\varphi \wedge z = (f\,\bar{z}) \wedge \bar{z} = \bar{u}\big) \vee \varphi' \wedge z = (f\,\bar{z}) \wedge \bar{z} = \bar{v}$	$\leftrightarrow_{tranz}$: k+2, k+3
(k+5)	$\big(\varphi \wedge z = (f\,\bar{z}) \wedge \bar{z} = \bar{u}\big) \vee \varphi' \wedge z = (f\,\bar{z}) \wedge \bar{z} = \bar{v} \leftrightarrow$ $z = (f\,\bar{z}) \wedge \big((\varphi \wedge \bar{z} = \bar{u}) \vee (\varphi' \wedge \bar{z} = \bar{v})\big)$	PROPOSITIONAL
(k+6)	$\exists \bar{z}.\big(\varphi \wedge z = (f\,\bar{z}) \wedge \bar{z} = \bar{u}\big) \vee \varphi' \wedge z = (f\,\bar{z}) \wedge \bar{z} = \bar{v} \leftrightarrow$ $\exists \bar{z}.z = (f\,\bar{z}) \wedge \big((\varphi \wedge \bar{z} = \bar{u}) \vee (\varphi' \wedge \bar{z} = \bar{v})\big)$	$\exists$-CTX: k+5
(k+7)	$\big(\varphi \wedge z = (f\,\bar{u})\big) \vee \big(\varphi' \wedge z = (f\,\bar{v})\big) \leftrightarrow$ $\exists \bar{z}.z = (f\,\bar{z}) \wedge \big((\varphi \wedge \bar{z} = \bar{u}) \vee (\varphi' \wedge \bar{z} = \bar{v})\big)$	$\leftrightarrow_{tranz}$: k+4, k+6

4.4.3 Proof schema $\rightsquigarrow_{step}$

Recall that in each step $\langle t_i, P_i\rangle \rightsquigarrow \langle t_{i+1}, P_{i+1}\rangle$, t_{i+1} is a generalisation of t_i, and both t_i and t_{i+1} are generalisations of the initial term patterns. Also, recall that both $\phi^{\langle t_i, P_i\rangle}$ and $\phi^{\langle t_{i+1}, P_{i+1}\rangle}$ are existentially quantified conjunctions between a term pattern (e.g., the generalisations t_i, t_{i+1}) and a predicate (cf. Definition 2). Our final step is to add the missing existential quantifiers and the missing term patterns (generalisations) to the equivalences obtained using DEC. The schema which does all the above is summarised below:

(k+1)	$\big(\varphi \wedge z=(f\,\overline{u})\big) \vee \big(\varphi' \wedge z=(f\,\overline{v})\big) \leftrightarrow$ $\exists \overline{z}.z=(f\,\overline{z}) \wedge \big((\varphi \wedge \overline{z}=\overline{u}) \vee (\varphi' \wedge \overline{z}=\overline{v})\big)$	DEC
(k+2)	$\exists \overline{x}.t \wedge \big(\varphi \wedge z=(f\,\overline{u})\big) \vee \big(\varphi' \wedge z=(f\,\overline{v})\big) \leftrightarrow$ $\exists \overline{x}.t \wedge \exists \overline{z}.z=(f\,\overline{z}) \wedge \big((\varphi \wedge \overline{z}=\overline{u}) \vee (\varphi' \wedge \overline{z}=\overline{v})\big)$	$z \in \overline{x}=var(t)$ $\exists$-CTX: k+1
(k+3)	$t \wedge \exists \overline{z}.z=(f\,\overline{z}) \wedge \big((\varphi \wedge \overline{z}=\overline{u}) \vee (\varphi' \wedge \overline{z}=\overline{v})\big) \leftrightarrow$ $\exists \overline{z}.t \wedge z=(f\,\overline{z}) \wedge \big((\varphi \wedge \overline{z}=\overline{u}) \vee (\varphi' \wedge \overline{z}=\overline{v})\big)$	$\overline{z}$ fresh $\exists$-SCOPE
(k+4)	$\exists \overline{x}.t \wedge \exists \overline{z}.z=(f\,\overline{z}) \wedge \big((\varphi \wedge \overline{z}=\overline{u}) \vee (\varphi' \wedge \overline{z}=\overline{v})\big) \leftrightarrow$ $\exists \overline{x}.\exists \overline{z}.t \wedge z=(f\,\overline{z}) \wedge \big((\varphi \wedge \overline{z}=\overline{u}) \vee (\varphi' \wedge \overline{z}=\overline{v})\big)$	$\overline{x} \in var(t)$ $\exists$-CTX: k+3
(k+5)	$\exists \overline{x}.t \wedge \big(\varphi \wedge z=(f\,\overline{u})\big) \vee \big(\varphi' \wedge z=(f\,\overline{v})\big) \leftrightarrow$ $\exists \overline{x}.\exists \overline{z}.t \wedge z=(f\,\overline{z}) \wedge \big((\varphi \wedge \overline{z}=\overline{u}) \vee (\varphi' \wedge \overline{z}=\overline{v})\big)$	$\leftrightarrow_{tranz}$: k+2, k+4
(k+6)	$\exists z.t \wedge z=(f\,\overline{z}) \wedge \big((\varphi \wedge \overline{z}=\overline{u}) \vee (\varphi' \wedge \overline{z}=\overline{v})\big) \leftrightarrow$ $\big(\exists z.t \wedge z=(f\,\overline{z})\big) \wedge \big((\varphi \wedge \overline{z}=\overline{u}) \vee (\varphi' \wedge \overline{z}=\overline{v})\big)$	$\exists$-SCOPE
(k+7)	$\exists \{\overline{x},\overline{z}\}.t \wedge z=(f\,\overline{z}) \wedge \big((\varphi \wedge \overline{z}=\overline{u}) \vee (\varphi' \wedge \overline{z}=\overline{v})\big) \leftrightarrow$ $\exists \{\overline{x},\overline{z}\} \setminus \{z\}.\big(\exists z.t \wedge z=(f\,\overline{z})\big) \wedge \big((\varphi \wedge \overline{z}=\overline{u}) \vee (\varphi' \wedge \overline{z}=\overline{v})\big)$	$\exists$-CTX: k+6
(k+8)	$\exists \overline{x}.t \wedge \big(\varphi \wedge z=(f\,\overline{u})\big) \vee \big(\varphi' \wedge z=(f\,\overline{v})\big) \leftrightarrow$ $\exists \{\overline{x},\overline{z}\} \setminus \{z\}.\big(\exists z.t \wedge z=(f\,\overline{z})\big) \wedge \big((\varphi \wedge \overline{z}=\overline{u}) \vee (\varphi' \wedge \overline{z}=\overline{v})\big)$	$\leftrightarrow_{tranz}$: k+5, k+7
(k+9)	$\exists z.t \wedge z=(f\,\overline{z}) \leftrightarrow t[(f\,\overline{z})/z]$	$\exists$-SUBST
(k+10)	$\big(\exists z.t \wedge z=(f\,\overline{z}) \leftrightarrow t[(f\,\overline{z})/z]\big) \leftrightarrow \big(\exists z.t \wedge z=(f\,\overline{z})\big) \leftrightarrow t[(f\,\overline{z})/z]$	$\exists$-SCOPE
(k+11)	$\big(\exists z.t \wedge z=(f\,\overline{z})\big) \leftrightarrow t[(f\,\overline{z})/z]$	MODUSPON.: k+9, k+10
(k+12)	$\big(\exists \{\overline{x},\overline{z}\}.t \wedge z=(f\,\overline{z})\big) \wedge \big(\varphi \wedge z=(f\,\overline{u})\big) \vee \big(\varphi' \wedge z=(f\,\overline{v})\big) \leftrightarrow$ $\exists \{\overline{x},\overline{z}\} \setminus \{z\}.t[(f\,\overline{z})/z] \wedge \big(\varphi \wedge z=(f\,\overline{u})\big) \vee \big(\varphi' \wedge z=(f\,\overline{v})\big)$	$\exists$-CTX: k+11
(k+13)	$\exists \overline{x}.t \wedge \big(\varphi \wedge z=(f\,\overline{u})\big) \vee \big(\varphi' \wedge z=(f\,\overline{v})\big) \leftrightarrow$ $\exists \{\overline{x},\overline{z}\} \setminus \{z\}.t[(f\,\overline{z})/z] \wedge \big(\varphi \wedge z=(f\,\overline{u})\big) \vee \big(\varphi' \wedge z=(f\,\overline{v})\big)$	$\leftrightarrow_{tranz}$: k+8, k+12

This schema uses the fact that Plotkin's algorithm generates fresh variables at each antiunification step. Here, $\overline{x} = \{x_1, \ldots, x_n\} = var(t)$ and $\exists\{\overline{x},\overline{z}\}$ stands for $\exists\{x_1, \ldots, x_n, z_1, \ldots, z_m\}$. At k+7, we directly use $\exists\{\overline{x},\overline{z}\}$ instead of $\exists\{\overline{x},\overline{z}\} \setminus \{z\}.\exists z$.

5 A tool for certifying antiunification

We implement a prototype [3] for our proof object generation mechanism and a checker for the generated proof objects. The proof generator and the proof checker are implemented in Maude [10]. Both tools can be used directly in Maude, but we also created a Python interface in order to facilitate the interaction of the user with the Maude tools.

The Python script takes easy-to-write specifications as inputs, it automatically calls the Maude proof generator and the proof checker behind the scenes, and outputs a proof and a checking status.

Table 1: The results obtained when generating proof objects for large inputs inspired from K definitions of real-life languages (C and Java).

Language	File name	File size (kb)	proof object size (no. of lines)
list of nats	`13_paper_cons_succ.in`	0.122	84
C	`18_c_declare_local.in`	14	5052
Java	`19_java_method_invoke.in`	6	2352

The specifications are minimally specified in an input file whose content is self-explanatory. For example, the antiunification problem from Example 2 is specified as:

```
variables: x1, x2, l1, l2
symbols: cons, succ, zero
problem: cons(succ(x1),cons(zero,l1))=?cons(x2,cons(succ(x2),l2))
```

The Python script parses the input and extracts the variables, the symbols, and the antiunification problem. It infers automatically the arities of the symbols and throws errors when the inputs are not well-formed or arities are inconsistent (e.g., same symbol used with different number of arguments). Then it calls the Maude proof generator and checker in the background. The output from Maude is postprocessed in Python and the user can inspect the proof and the checking status (`true` or `false`):

```
> python3 ml-antiunify.py tests/samples/13_paper_cons_succ.in
Proof of: // goal: ...
// generated proof...
Checked:   true
```

For this particular example, the generated proof has 84 proof lines as shown in the first line of Table 1.

We tested our prototype on larger inputs as well. Our goal was to see if the size of the generated proof objects for real-life language configurations is indeed manageable (e.g., the size should increase linearly, not exponentially). Also, we wanted to check whether our proof-generation schemas are correctly instantiated and they compose as expected to obtain the final proof objects. The inputs that we use are inspired from the K definitions of several languages, including the K definitions of C [12] and Java [5]. In these K definitions, the configurations are quite big (i.e., $\sim$130 nodes for C, $\sim$65 nodes for Java). From these definitions we extracted some large term patterns which we use as inputs for antiunification.

Table 1 shows some of the results that we obtained[5]. The input terms in our specifications represent C and Java symbolic configurations converted into our input format. We use the specification file size to measure the input size and we use the number of proof lines to measure the size of proof object. As expected, the size of the generated proof objects depends on the input size. For language definitions that have larger configurations the generated proofs are big (e.g., 5000 lines for C, and 2300 lines for Java).

For each step of the antiunification algorithm, the proof generator (which implements the schemas discussed in Section 4) produces a fixed number of proof lines. Therefore, the size of the proof object is directly proportional with the number of execution steps of the antiunification algorithm. In the worst-case scenario, when executed on an input $\langle z, z \mapsto t_1 \sqcup t_2 \rangle$ the number of execution steps of the antiunification algorithm is given by the size of the smallest term pattern between t_1 and t_2. An example of worst-case scenario is when one term is an instance of the other, i.e., the lgg t_1 and t_2 is t_i with $i \in \{1,2\}$.

[5]Details can be found here: `https://github.com/andreiarusoaie/certifying-unification-in-aml/tree/master/tests/samples#readme`

The tests that we performed on large inputs show that our proof object generation method can be implemented and used in practice. Our prototype is able to produce a correct output (i.e., the proof objects are successfully checked by the checker) and thus, it shows that there are no corner cases that we missed in our approach.

6 Conclusions

In order to obtain a certified symbolic execution, the parameters of an execution step have to carry the proof object of their actions. Two examples of such parameters are unification and antiunification algorithms. In this paper we proposed a generic method for generating proof objects for the tasks that the K tool performs. We showed how this method works for the case of antiunification by providing schemas for generating proof objects. More precisely, we used Plotkin's antiunification to normalise disjunctions and to generate the corresponding proof objects. We also provided a prototype implementation of our proof object generation technique and a checker for the generated proof objects.

The prototype generates proof objects whose size depends on the number of steps performed by the antiunification algorithm. However, the number of steps is limited by the size of the inputs. We successfully used our prototype on complex inputs inspired from the K semantics of C and Java. This indicates that our approach is practical even for large inputs.

Future work In order to obtain a proof object that uses only the Hilbert-style proof rules, the next step is to find proof schemata for the macro rules in Figure 3. This will allow us to use the newest proof checker for ML implemented in Metamath [24]. Both checkers (the existing Metamath and our Maude implementations) have the same functionality w.r.t. ML proof system. So far, we preferred our Maude implementation because it was easier to handle macro rules. Also, we reused only the ML syntax module written in Maude for both the checker and the proof generator. However, the short term goal is to use the Metamath checker so that the proof object generator and the checker are completely independent.

References

[1] Andrei Arusoaie (2014): *A Generic Framework for Symbolic Execution: Theory and Applications*. Ph.D. thesis, Faculty of Computer Science, Iaşi. Available at `https://tel.archives-ouvertes.fr/tel-01094765`. Alexandru Ioan Cuza, University of Iaşi, Romania.

[2] Andrei Arusoaie & Dorel Lucanu (2019): *Unification in Matching Logic*. In Maurice H. ter Beek, Annabelle McIver & José N. Oliveira, editors: *Formal Methods - The Next 30 Years - Third World Congress, FM 2019, Porto, Portugal, October 7-11, 2019, Proceedings*, *Lecture Notes in Computer Science* 11800, Springer, pp. 502–518, doi:10.1007/978-3-030-30942-8_30.

[3] Andrei Arusoaie & Dorel Lucanu (2021): *Certifying (anti)unification in Matching Logic*. Available at `https://github.com/andreiarusoaie/certifying-unification-in-aml`.

[4] Andrei Arusoaie, David Nowak, Vlad Rusu & Dorel Lucanu (2017): *A Certified Procedure for RL Verification*. In: *SYNASC 2017*, IEEE CPS, Timişoara, Romania, pp. 129–136. Available at `https://hal.inria.fr/hal-01627517`.

[5] Denis Bogdanas & Grigore Roşu (2015): *K-Java: A Complete Semantics of Java*. In: *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ACM, New York, NY, USA, pp. 445–456. Available at `http://doi.acm.org/10.1145/2676726.2676982`.

[6] Xiaohong Chen, Zhengyao Lin, Minh-Thai Trinh & Grigore Roşu (2021): *Towards a Trustworthy Semantics-Based Language Framework via Proof Generation*. In: *Proceedings of the 33rd International Conference on Computer-Aided Verification*, ACM, p. 477–499, doi:10.1007/978-3-030-81688-9_23.

[7] Xiaohong Chen, Dorel Lucanu & Grigore Roşu (2020): *Initial Algebra Semantics in Matching Logic*. Technical Report http://hdl.handle.net/2142/107781, University of Illinois at Urbana-Champaign. Submitted.

[8] Xiaohong Chen, Dorel Lucanu & Grigore Roşu (2021): *Matching logic explained*. *Journal of Logical and Algebraic Methods in Programming* 120, p. 100638, doi:10.1016/j.jlamp.2021.100638.

[9] Xiaohong Chen & Grigore Roşu (2019): *Matching μ-logic*. In: *Proceedings of the 34th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS'19)*, IEEE, Vancouver, Canada, pp. 1–13, doi:10.1109/LICS.2019.8785675.

[10] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer & C. L. Talcott. (2007): *All About Maude, A High-Performance Logical Framework*. *Lecture Notes in Computer Science* 4350, Springer, doi:10.1007/978-3-540-71999-1_10.

[11] Andrei Ştefănescu, Daejun Park, Shijiao Yuwen, Yilong Li & Grigore Roşu (2016): *Semantics-Based Program Verifiers for All Languages*. In: *OOPSLA'16*, ACM, pp. 74–91, doi:10.1145/2983990.2984027.

[12] Chucky Ellison & Grigore Rosu (2012): *An Executable Formal Semantics of C with Applications*. In: *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, New York, USA, pp. 533–544. Available at `http://doi.acm.org/10.1145/2103656.2103719`.

[13] Chris Hathhorn, Chucky Ellison & Grigore Roşu (2015): *Defining the Undefinedness of C*. In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, New York, NY, USA, pp. 336–345. Available at `http://doi.acm.org/10.1145/2737924.2737979`.

[14] Dorel Lucanu, Vlad Rusu & Andrei Arusoaie (2016): *A Generic Framework for Symbolic Execution: A Coinductive Approach*. *Journal of Symbolic Computation*, pp. –, doi:10.1016/j.jsc.2016.07.012.

[15] Dorel Lucanu, Vlad Rusu, Andrei Arusoaie & David Nowak (2015): *Verifying Reachability-Logic Properties on Rewriting-Logic Specifications*. In Narciso Martí-Oliet, Peter Csaba Ölveczky & Carolyn L. Talcott, editors: *Logic, Rewriting, and Concurrency - Essays dedicated to José Meseguer on the Occasion of His 65th Birthday*, *Lecture Notes in Computer Science* 9200, Springer, pp. 451–474, doi:10.1007/978-3-319-23165-5_21.

[16] Alberto Martelli & Ugo Montanari (1982): *An Efficient Unification Algorithm*. *ACM Transactions on Programming Languages and Systems* 4(2), pp. 258–282, doi:10.1145/357162.357169.

[17] Daejun Park, Andrei Ştefănescu & Grigore Roşu (2015): *KJS: A Complete Formal Semantics of JavaScript*. In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, New York, NY, USA, pp. 346–356. Available at `http://doi.acm.org/10.1145/2737924.2737991`.

[18] G. Plotkin (1972): *Building in equational theories*. *Machine Intelligence* 7, pp. 73–90. Available at `http://www.cs.york.ac.uk/mlg/MI/mi.html`. Http://www.cs.york.ac.uk/mlg/MI/mi.htmlJournal Webpage.

[19] Gordon D. Plotkin (1970): *A Note on Inductive Generalization*. *Machine Intelligence* 5, pp. 153–163.

[20] Grigore Rosu (2017): *Matching Logic*. *Log. Methods Comput. Sci.* 13(4), pp. 1–61, doi:10.23638/LMCS-13(4:28)2017.

[21] Grigore Roșu, Andrei Ștefănescu, Ştefan Ciobâcă & Brandon M. Moore (2013): *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, doi:10.1109/LICS.2013.42.

[22] Vlad Rusu & Andrei Arusoaie (2016): *Proving Reachability-Logic Formulas Incrementally*. In Dorel Lucanu, editor: *Rewriting Logic and Its Applications - 11th International Workshop, WRLA 2016, April 2-3, 2016, Revised Selected Papers*, *Lecture Notes in Computer Science* 9942, Springer, pp. 134–151, doi:10.1007/978-3-319-44802-2_8.

[23] Andrei Stefanescu, Ştefan Ciobâcă, Radu Mereuta, Brandon M. Moore, Traian-Florin Serbanuta & Grigore Rosu (2014): *All-Path Reachability Logic*. In Gilles Dowek, editor: *RTA-TLCA 2014, 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, *LNCS* 8560, Springer, pp. 425–440, doi:10.1007/978-3-319-08918-8_29.

[24] K Team: *Matching Logic Proof Checker*. Available at `https://github.com/kframework/matching-logic-proof-checker/blob/main/theory/matching-logic-250-loc.mm`.