Higher-Order in SMT

Daniel El Ouraoui, supervised by Pascal Fontaine and Jasmin Christian Blanchette,

Team VeriDis, Inria, and LORIA, Nancy, France

August 21, 2017

General context

Improving the security of many applications in business, industry and scientific research requires procedures to determine the satisfiability of logical formulas with respect to some specific theory, e.g. arithmetic on integers and reals, arrays or bit-vectors. One usual way to reach this aim is to use Satisfiability Modulo Theories (SMT) solvers. SMT stems from the propositional satisfiability problem(SAT), i.e. checking whether or not a propositional formula is satisfied. Modern SMT solvers extend this propositional decision procedure to more expressive logics. Originally, these tools were developed to check the satisfiability of first-order formulas. Between the late 1970s and early 1980s, Nelson and Oppen [35] developed the first practical decision procedure for the theory of uninterpreted function symbols. In the meantime, Shostak [43] progressed on a practical decision procedure for arithmetic. Later, Boyer and Moore [19] provided one of the first theorem provers based on the concept of a theory reasoner. The Boyer-Moore theorem prover, was able to handle quantifiers, arithmetic, induction and some rewriting problems. In the late 1990s, the interest for SMT was increasing. Several research projects were developed, in academia as well as in industry. Modern SMT solvers now rely on state-of-the-art techniques based on the advances in SAT technology. These advances contributed to make them usable tools to solve specific first-order problems. Today the usages of SMT solvers are various. They can go from the verification of systems (programs), e.g. Why3, F* or TLA+, to poof assistants, e.g. Coq (SMTCoq), Isabelle (Sledgehammer), HOL or PVS. Recently, some programming languages like Haskell or F*, use SMT features to build more expressive type systems — i.e. refinement types or LIQUID types. There are two major approaches to handle the SMT problem: the "eager" and "lazy" ones. The eager approach decomposes an input formula, expressed in some first-order logic, into a set of propositional clauses which is equisatisfiable. Several specialized translations and relevant consequences of the based theory are used. In some way, this method can be considered like a reduction from the SMT problem to the SAT problem. This method is thus only based on the ability of the SAT solver to quickly solve a formula. The main argument in favor of this technique is that SAT is a decidable problem, with practically efficient solvers. However, this is an NP-complete problem (without known low complexity decision procedure) and the translation might furthermore explode the size of formulas. The lazy approach is an ad hoc procedure specialized on a background theory $\mathcal{T}$. Typically, this method is built around a CDCL (Conflict-Driven Clause Learning) algorithm extended to understand the background theory. The common practice is to write a theory solver able to handle conjunctive sets of literals (that is, atomic formulas and their negations) and leave the Boolean structure of the formula to the underlying SAT solver. This architecture allows to use the theory solvers as separate sub-modules, and leads to a greater flexibility.

The research problem

Now, most of the efficient SMT solvers provide a suitable method to understand first-order (FO) formula with respect to the theory. Some of them offer useful extensions. For example, Alt-Ergo allows polymorphism and CVC4 provides datatypes, induction and function synthesis. But none of them are today able to reason on higher-order (HO) formulas. The main goal of my internship (and my coming PhD) is to lift up the reasoning capabilities of SMT solvers towards

higher-order. In order to propose more efficient tactics and richer higher-order languages, most of the interactive verification tools (e.g. Coq, Why3, Isabelle) use SMT solvers as backend reasoning engines. Unfortunately, the language of these tools cannot be directly understood by the SMT solvers. Most of the time, a translation should be defined to interface SMT solvers with the more expressive tools. Often these translations rely on problems that are theoretically or practically unprovable. Generally, SMT solvers cannot perform induction, which is typically the corner stone of a higher-order reasoning. To our best knowledge, only CVC4 provides support for induction but in a limited form. However, we believe it is important to develop higher-order reasoning in SMT. First, this would allow to by-pass the actual translation which could lead to solve many more problems. Support for full inductive reasoning could also significantly improve the reasoning capability. Higher-order logic yields a concise syntax to express problems in much simple terms. Obviously, a native understanding could lead to better performances.
One of the first initiative to automating higher-order logic was the LEO-II [10] (now LEO-III) prover which relies on a higher-order resolution calculus. Recently, Satallax [20] which is also a higher-order theorem prover has achieved remarkable results at the CASC competition [44] in the higher-order division. Developing higher-order reasoning for SMT is in the line of these previous works. This ambitious initiative is one of the aims of the Matryoshka [1] project.

Contributions

As a starting point, I studied the different encoding to first-order from higher-order logic. In particular, we investigated the encoding of λ-free higher-order [14]. This was helpful to understand the differences between HO and FO logic. We have extended the language and the typing rules of the SMT-LIB standard [8] to handle higher-order terms. In a second phase, we focused on two different axes, the solving part and the proof production. The solving part has been extended in two ways:

- the pre-processing of all formulas by a module using rewriting to eliminate intricate cases;
- extend the congruence closure algorithm to deal with higher-order function symbols.

The proof-producing module of the SMT solver has been extended to deal with higher-order formulas. In the last part of my internship I investigated instantiation of quantifiers.

Arguments supporting its validity

This report is a collection of several works pursued during my internship. Some of them have been implemented in the veriT solver [18]. In particular the language, the typing system, the pre-processing (higher-order congruence, normalization) and the congruence-closure algorithm have been implemented. A part of this work was the object of an article accepted to the PxTP workshop 2017 which includes the result on proof of processing. The work performed on the instantiation is still in progress and not yet fully evaluated.

Summary and future work

This internship covered the first steps required to build a higher-order reasoning module for SMT. In theory, this provides a complete procedure for simple type theory. It remains however to implement the last piece of the puzzle, that is the E-matching algorithm with λ-patterns — i.e see section 7. In this way we'll able to implement the triggers based instantiation to tackle higher-order quantification. We hope that we will be able to handle most of our benchmarks with this method. A suitable approach would be to extend and implement the E-unification algorithm to λ-patterns and generalizes this result to Conflict based instantiation. We could then compare the results with the previous implementations. A fundamental question is how to instantiate efficiently the universal quantified function symbols. Unification is one of the way to reach this aim but there may be others. Another direction of research could be to study the synthesis of functions [41] to develop a new method of instantiation for function symbols. As a side quest, it could be stimulating to explore other tracks. In particular, how to manage thousands of well know lemmas during the instantiation, which of theme are really relevant for the current goal and on which criteria are they selected? Some of these strips have been studied — i.e see [26]. But adapt these solutions in SMT to use libraries like TPTP or Isabelle [2] directly with the solvers, could be an interesting work.

[1] http://matryoshka.gforge.inria.fr/
[2] https://www.isa-afp.org/

1 Introduction

Higher-order (HO) logic is a pervasive setting for reasoning about numerous real-world applications. In particular, it is widely used in proof assistants (also known as interactive theorem provers) to provide trustworthy, machine-checkable formal proofs of theorems. A major challenge in these applications is to automate as much as possible the production of these formal proofs, thereby reducing the "burden of proof" on the users.
An effective approach towards stronger automation is to rely on less expressive but more automatic theorem provers to discharge some of the proof obligations. Systems such as HOLYHammer, MizAR, Sledgehammer, and Why3, which provide a one-click connection from proof assistants to first-order provers, have led in recent years to considerable improvements in proof assistant automation [15]. Today, the leading automatic provers for first-order classical logic are based either on the superposition calculus [4, 37] or on CDCL($\mathcal{T}$) [36]. Those based on the latter are usually called satisfiability modulo theory (SMT) solvers and are the focus of this report.
Our goal, as part of the Matryoshka project, is to extend SMT solvers to natively handle higher-order problems, thus avoiding completeness and performance issues with clumsy encodings. In this reporte, we present our first steps towards two contributions within our established goal: to extend the input (problems) and output (proofs) of SMT solvers to support higher-order constructs. Most SMT solvers support SMT-LIB [8] as an input format. We provide a syntax extension for augmenting SMT-LIB with higher-order functions with partial applications, λ-abstractions, and quantification on higher-order variables.

2 Background

As in [21], we use a classical higher-order logic based on the formulation of simple typed λ-calculus. The simple types are freely generated from the basic types ι and o. Usually, ι represents a non-empty domain of individuals (in SMT there are generally some basic types for individuals like `Int` or `Real`), while o denotes the type of thrust values (more commonly called Boolean values). The simple types theory allows one to express functional types by the arrow constructor $\to$. So if τ_1 and τ_2 are simple types then $\tau_1 \to \tau_2$ is a simple type. The set of terms of higher-order logic is built inductively over the following extended grammar, where we represent the constant symbols in the grammar by c and the variables by x. Additionally, we extend the grammar with the $\mathbf{let}\, x = u\, \mathbf{in}\, t$ construction whose semantics is exactly that of the following β-redex $(\lambda x.t)\, u$.

$$M \ ::= \ \ x \ \mid \ c \ \mid \ M\ M \ \mid \ (\lambda x.M) \ \mid \ \mathbf{let}\, x \ = \ M\, \mathbf{in}\, M$$

Notations In the following report, the letters a, c, f, g, p, q stand for function symbols. F, G, H denote free function variables. a and b can be used to denote sets but it will always be made explicit before. w, x, y, z stand for variables, r, s, t, u, v for well-typed terms, φ, ψ for formulas, $\mathcal{B}$ for a binder in the set $\{\forall,\ \exists,\ \lambda\}$ and $\mathcal{Q}$ in $\{\forall,\ \exists\}$. We note $\overline{a}_n$ or $(a_i)_{i=1}^n$ the vector with n elements, and we write [n] for $\{1, \ldots, n\}$. We express the fact that a term may depend on the distinct variables $\overline{x}_n$ by the notation t[$\overline{x}_n$]. Generally, we use the Greek letters τ, σ to express any simple types (e.g o, ι, $o \to o$, $\iota \to \iota$ ...), α the simple types which are neither propositions nor predicates and θ that is propositions or predicates. In this report, we'll use curried and uncurried formulations of functions. We suppose implicitly the existence of two functions $\mathbf{curry}\ f\ (t_1, \ldots, t_n)$ and $\mathbf{uncurry}\ f\ t_1 \ldots t_n$, where t_i has a type τ_i, respectively $\mathbf{curry}$ is of type $(\tau_1 \times \ldots \times \tau_n \to \tau) \to (\tau_1 \times \ldots \times \tau_n) \to (\tau_1 \to \ldots \to \tau_n \to \tau)$ then $\mathbf{curry}\ f\ (t_1, \ldots, t_n) = ((f\ t_1) \ldots t_n)$ and $\mathbf{uncurry}$ is of type $(\tau_1 \to \ldots \to \tau_n \to \tau) \to \tau_1 \to \ldots \to \tau_n \to (\tau_1 \times \ldots \times \tau_n \to \tau)$ hence $\mathbf{uncurry}\ f\ t_1 \ldots t_n = f(t_1, \ldots, t_n)$. We assume both notations are equivalent. In particular, the λ-terms of the shape $\lambda x_1 \ldots \lambda x_n.\ t$ are written $\lambda \overline{x}_n.\ t$.

Terms Every variable or constant of type τ are terms. If f is function symbol of type $(\tau_1 \to \ldots \to \tau_n \to \sigma)$ and t_{τ_1}, ..., t_{τ_n} are terms of types τ_i, then $f\ t_{\tau_1}\ \ldots\ t_{\tau_m}$ is a term of type σ.

[1] `http://matryoshka.gforge.inria.fr/`

Formulas Every term of type o is a formula. With respect to the λ-calculus formulation, all logical connectors are declared as constant symbols as follows: $\neg_{o\to o}$, $\vee_{o\to o\to o}$, $\wedge_{o\to o\to o}$, $\Rightarrow_{o\to o\to o}$, $\forall_{(\iota\to o)\to o}$, $\exists_{(\iota\to o)\to o}$. In addition, we have a family of symbols indexed by all types of the theory $(=_{\tau\to\tau\to o})$ which stand for the standard equality. To distinguish between equality terms and the equivalence relation we give another family of equality symbols that we write $(\simeq_{\tau\to\tau\to o})$. $\varphi \oplus \psi$ is short way to write binary operator. $\oplus$ range in $\{\vee, \wedge, \Rightarrow\}$. If p is a symbol of type $(\tau_1 \to \ldots \to \tau_n \to o)$ and $t_{\tau_1}, \ldots, t_{\tau_n}$ are terms of the type τ_i, then $p\ t_{\tau_1}\ \ldots\ t_{\tau_m}$ is formula, and p is a predicate.

Atoms (or literals) An atom is a formula of the form $p\ t_1\ \ldots\ t_m$ where p is predicate symbol.

Ground formulas (terms) A ground formula (term) has no variables or quantifiers.

Closed formulas A closed formula has no free variables, whereas an open formula does have free variables.

Free variables and substitutions We denote by $FV(t)$ free variables of the term t. The substitution is defined as meta-function over terms. Substitution is defined up to α-renaming —i.e here before reducing $(\lambda x.(\lambda y.x\,y))\ y$ we need to rename y in z thereby $(\lambda x.(\lambda y.x\,y))\ y \simeq_\alpha (\lambda x.(\lambda z.x\,z))\,y$. We write $t\{x \mapsto u\}$ to say we substitute all occurrences of x in t by the term u. We write $t\{\overline{x}_n \mapsto \overline{u}_n\}$ for $t\{x_1 \mapsto u_1\}\ldots\{x_n \mapsto u_n\}$.

higher-order logic $\mathcal{F}^\omega$ denote the system of ω-order logic that have all finite order logics as subsystems. This is a common notation of the higher-order logical system [2].

Signature of $\mathcal{F}^\omega$ Let S a set of symbols, and τ some type, $S = \bigcup_\tau S_\tau^{const} \cup \bigcup_\tau S_\tau^{var}$. (see [27])

logical system Let $\mathcal{L}$ a logical system, we denote by $\mathcal{L}(S)$ a logic in $\mathcal{L}$ defined by the signature S. Likewise we define t a term of type τ over the signature S_τ by $t \in T(S_\tau)$.

map set Let $A_1, \ldots A_n, B$ be sets, then $\mathcal{F}(A_1, \ldots A_n; B)$ is the set of all function from $(A_1 \times \ldots \times A_n)$ to B.

Frame A frame is a collection $\{\mathcal{D}_\tau\}_\tau$ of non empty set $\mathcal{D}_\tau$, for each type τ. In particular, $\mathcal{D}_o = \{\top, \bot\}$ and $\mathcal{D}_{\tau_1\times\ldots\times\tau_n\to\sigma} \subseteq \mathcal{F}(\mathcal{D}_{\tau_1}, \ldots \mathcal{D}_{\tau_n}; \mathcal{D}_\sigma)$ — i.e. unlike in the Tarski's model where $\mathcal{D}_{\tau_1\times\ldots\times\tau_n\to\sigma} = \mathcal{F}(\mathcal{D}_{\tau_1}, \ldots \mathcal{D}_{\tau_n}; \mathcal{D}_\sigma)$ (standard model).

Interpretation $(\ \{\mathcal{D}_\tau\}_\tau\ , \mathcal{I}\ )$ consists of a frame and the map $\mathcal{I} : S_\tau^{const} \to \mathcal{D}_\tau$ for each type τ.

Valuation is a map $\mathcal{V} : S_\tau^{var} \to \mathcal{D}_\tau$ for all type τ.

General model Let $\mathcal{M}$ = ($\{\mathcal{D}_\tau\}_\tau$, $\mathcal{I}$) is called a general model for $\mathcal{F}^\omega$ iff there is a map $[\![.]\!]^\mathcal{M}$ such that for all valuation $\mathcal{V}$ and term t of type τ $[\![t]\!]^\mathcal{M}_\mathcal{V} \in \mathcal{D}_\tau$ and :

- $\forall$ x $\in S_\tau^{var}$. $[\![x]\!]^\mathcal{M}_\mathcal{V} = \mathcal{V}(x)$
- $\forall$ c $\in S_\tau^{const}$. $[\![c]\!]^\mathcal{M}_\mathcal{V} = \mathcal{I}(c)$
- $\forall$ f $\in S_{\tau_1\times\ldots\times\tau_n\to\sigma}, t_1 \in T(S_{\tau_1}), \ldots, t_n \in T(S_{\tau_n})$ $[\![f(t_1, \ldots, t_n)]\!]^\mathcal{M}_\mathcal{V} = [\![f]\!]^\mathcal{M}_\mathcal{V}\ ([\![t_1]\!]^\mathcal{M}_\mathcal{V}, \ldots, [\![t_n]\!]^\mathcal{M}_\mathcal{V})$
- for all formula φ, ψ $[\![\varphi\ \oplus\ \psi]\!]^\mathcal{M}_\mathcal{V} = [\![\varphi]\!]^\mathcal{M}_\mathcal{V} \oplus [\![\psi]\!]^\mathcal{M}_\mathcal{V}$
- for all formula φ, $[\![\neg\varphi]\!]^\mathcal{M}_\mathcal{V} = \neg\ [\![\varphi]\!]^\mathcal{M}_\mathcal{V}$
- for all formula φ, $[\![\forall x_\tau \varphi]\!]^\mathcal{M}_\mathcal{V} = \Pi_{d\in\mathcal{D}_\tau}\ [\![\varphi]\!]^\mathcal{M}_{\mathcal{V}[x_\tau\mapsto d]}$
  We note by $(\Pi_{d\in\mathcal{D}_\tau}\varphi)$ the conjunction $d_1 \in \mathcal{D}_\tau, \ldots, d_n \in \mathcal{D}_\tau,\ \varphi\{x \mapsto d_1\} \wedge \ldots \wedge \varphi\{x \mapsto d_n\}$

– for all formula φ, $[\![\exists x_\tau \varphi]\!]^{\mathcal{M}}_{\mathcal{V}} = \Sigma_{d \in \mathcal{D}_\tau} [\![\varphi]\!]^{\mathcal{M}}_{\mathcal{V}[x_\tau \mapsto d]}$
We note by $(\Sigma_{d \in \mathcal{D}_\tau} \varphi)$ the disjunction $d_1 \in \mathcal{D}_\tau, \ldots, d_n \in \mathcal{D}_\tau,\ \varphi\{x \mapsto d_1\} \vee \ldots \vee \varphi\{x \mapsto d_n\}$

– $\forall\, t_1, t_2 \in T(S_\tau)$, $[\![t_1 \equiv_{\tau \times \tau \to o} t_2\,]\!]^{\mathcal{M}}_{\mathcal{V}} = [\![t_1]\!]^{\mathcal{M}}_{\mathcal{V}} \equiv_{\mathcal{D}_\tau} [\![t_2\,]\!]^{\mathcal{M}}_{\mathcal{V}}$
Note $\equiv_{\tau \times \tau \to o}$ is constant a symbol in the signature then it has a special interpretation; in fact, we consider terms equal iff they are equal in the domain of the type τ then if they are the same interpretation they are equal in the same domain.

3 A Syntax Extension for the SMT-LIB Language

Currently, the SMT-LIB standard is at version 2.5 [8], and version 2.6 is in preparation. Although some discussions to extend this language to higher-order logic have occurred in the past, notably to include λ-abstractions, the format is currently based on many-sorted first-order logic. We here propose to extend the language in a pragmatic way to accommodate higher-order constructs: higher-order functions with partial applications, λ-abstractions, and quantifiers ranging over higher-order variables. Our extension is inspired by the work on TIP (Tools for Inductive Provers) [42], which is another pragmatic extension of SMT-LIB.
SMT-LIB contains commands to define atomic sorts and functions, but no functional sorts. We first extend the language so that functional sorts can be built:

$$\begin{array}{rcl} \langle \textit{sort} \rangle & ::= & \langle \textit{identifier} \rangle \mid \texttt{(}\ \langle \textit{identifier} \rangle\ \langle \textit{sort} \rangle^+\ \texttt{)} \\ & \mid & \texttt{(}\ \texttt{->}\ \langle \textit{sort} \rangle^+\ \langle \textit{sort} \rangle\ \texttt{)} \end{array}$$

The second line is the addition to the original grammar.
The next modification is in the grammar for terms, which essentially adds a rule for λ-abstractions and generalizes the application so that any term can be applied to other terms:

$$\begin{array}{rcl} \langle \textit{term} \rangle & ::= & \langle \textit{spec\_constant} \rangle \\ & \mid & \langle \textit{qual\_identifier} \rangle \\ & \mid & \texttt{(}\ \langle \textit{term} \rangle\ \langle \textit{term} \rangle^+\ \texttt{)} \\ & \mid & \texttt{(}\ \texttt{lambda}\ \texttt{(}\ \langle \textit{sorted\_var} \rangle^+\ \texttt{)}\ \langle \textit{term} \rangle^+\ \texttt{)} \\ & \mid & \texttt{(}\ \texttt{let}\ \texttt{(}\ \langle \textit{var\_binding} \rangle^+\ \texttt{)}\ \langle \textit{term} \rangle\ \texttt{)} \\ & \mid & \texttt{(}\ \texttt{forall}\ \texttt{(}\ \langle \textit{sorted\_var} \rangle^+\ \texttt{)}\ \langle \textit{term} \rangle\ \texttt{)} \\ & \mid & \texttt{(}\ \texttt{exists}\ \texttt{(}\ \langle \textit{sorted\_var} \rangle^+\ \texttt{)}\ \langle \textit{term} \rangle\ \texttt{)} \\ & \mid & \texttt{(}\ \texttt{match}\ \langle \textit{term} \rangle\ \texttt{(}\ \langle \textit{match\_case} \rangle^+\ \texttt{)}\ \texttt{)} \\ & \mid & \texttt{(}\ \texttt{!}\ \langle \textit{term} \rangle\ \langle \textit{attribute} \rangle^+\ \texttt{)} \\ \langle \textit{sorted\_var} \rangle & ::= & \texttt{(}\ \langle \textit{symbol} \rangle\ \langle \textit{sort} \rangle\ \texttt{)} \end{array}$$

The old rule $\texttt{(}\langle \textit{qual\_identifier} \rangle \langle \textit{term} \rangle^+ \texttt{)}$ is now redundant. Higher-order quantification requires no new syntax, since sorts have been extended to accommodate functions.
If we want to define a function taking an integer as argument and returning a function from integers to integers, it is now possible to write `(declare-fun f (Int) (-> Int Int))`. The following example illustrates higher-order functions, terms representing a function, and partial applications:

```
(set-logic UFLIA)
(declare-fun g (Int) (-> Int Int))
(declare-fun h (Int Int) Int)
(declare-fun f ((-> Int Int)) Int)
(assert (= (f (h 1)) ((g 1) 2)))
(exit)
```

The term `(g 1)` is a function form `Int` to `Int`, in agreement with the sort of `g`. Then it is applied to `2` in the expression `((g 1) 2)` of sort `Int`. The term `(h 1)` is a partial application of the binary function `h`, and is thus a unary function. The term `(f (h 1))` is therefore well-typed and is an `Int`. The next example features a λ-abstraction:

```
(set-logic UFLIA)
(declare-fun g (Int) (Int))
(assert
  (= ((lambda ((f (-> Int Int)) (x Int)) f x) g 1) (g 1)))
(exit)
```

The term `(lambda ((f (-> Int Int)) (x Int)) f x)` is an anonymous function that takes a function `f` and an integer `x` as arguments. It is applied to `g` and `1`, and the fully applied term is stated to be equal to `(g 1)`. The assertion is a tautology (thanks to β-reduction).

Typing rules In the following, we give the details for extending typing judgment with respect to the SMT-LIB. These rules extend the current set of rules in the SMT-LIB. This extension is straightforward. Only the typing rules for the λ-abstraction and the generalization of application are new. With these two rules we get all the power of the simply typed λ-calculus. A judgment is composed of two items. On the left hand, the **signature** Σ that is a tuple composed of function symbols and constant symbols. These symbols are annotated by their types. On the right hand is the term annotated by its type. We call a such term an **annotated term**. The notation $\Sigma[x : \tau]$ denotes the signature that maps x to the type τ in the signature Σ.

$$\frac{}{\Sigma[x:\tau] \vdash x : \tau}\ \texttt{var} \qquad \frac{\Sigma[x:\sigma] \vdash t : \tau}{\Sigma \vdash \lambda x.t : \sigma \to \tau}\ \texttt{lambda}$$

$$\frac{\Sigma \vdash u : \sigma \quad \Sigma[x:\sigma] \vdash t : \tau}{\Sigma \vdash \textbf{let}\ x = u\ \textbf{in}\ t : \tau}\ \texttt{let} \qquad \frac{\Sigma \vdash u : \sigma \to \tau \quad \Sigma \vdash v : \sigma}{\Sigma \vdash u\ v : \tau}\ \texttt{app}$$

Implementation In the veriT solver, the transition was a bit less simple. In fact as we have discussed above there is no ambiguity when two terms are equal. Especially for the binary application, which is left associative. Left associative means, that if we omit the parentheses we read first the leftmost term. For example, the term $(\lambda x.x)\, t\, u$ should be parsed as $((\lambda x.x)\, t)\, u$. In the λ-calculus all terms are in curried form. More precisely, the application is part of the syntax. Consequently, it is not required to declare some specific function symbols for each occurrence of application. For example, if f is a function symbol with n arguments and there is an application $f(a_1, \ldots, a_n)$ in the formula then these informations should be in the signature. While than in λ-calculus only the information that f is a symbol is required. This is why its formulation is so simple. Though, veriT is a first-order prover its representation of terms is flexible. More exactly, veriT use a Dag representation of terms with maximum sharing. This representation allows it to be extended to a higher-order syntax in simple way. Thereby, to deal with simple types and handle equalities of functional types we have chosen to prepossess types before saving functional types in the typing environment and in order to resolve the problem. By typing environment we mean the Σ signature in the abstract rules above. We give the idea implemented in veriT with the simple following Ocaml code. We denote by the constructor `Arrow of sort list` the arrow type $\to$. The function `last_of l` returns the last element of `l` and `unhook_last l` return the list l without the last element. This function is called at the parsing level.

```
(** lty = [t_1 ; ... ; t_n ; return_type]  **)
let rec flatting_type : sort -> sort = function
    | Arrow lty ->
    let return_type = last_of lty in
    match return_type with
     | Arrow l as arrow ->
       let append_type = flatting_type arrow in
       let args_ty = unhook_last lty in
       Arrow(args_ty @ append_type)
     | _ -> Arrow lty
    | ty -> ty
```

4 Pre-processing formulas

One way to deal with higher-order formulas could be to handle the higher-order constructions before solving the formula. The approach aims to avoid complicated cases such as partial-application or β-redex before the solving.

α-conversion α-conversion is the least equivalence relation $\simeq_\alpha$ on λ-terms such that for any λ-term t, and variables x, y where $y \notin FV(t)$, we have:

$$\lambda x.t \simeq_\alpha \lambda y.t\{x \mapsto y\}. \tag{α}$$

β-reduction The β-rule is the following reduction relation between λ-terms, such that for any λ-terms t,u, and variable x, we have:

$$(\lambda x.t)\, u \to_\beta t\{x \mapsto u\}. \tag{β}$$

The sub-term $(\lambda x.t)\, u$ which is transformed by the β-rule is called the **redex** (or **β-redex**).

η-reduction The η-rule is the following reduction relation, such that for any λ-term t, and variable x we have:

$$(\lambda x.t\, x) \to_\eta t. \qquad \text{if } x \notin FV(t). \tag{η}$$

Normal form An expression is in normal form if it cannot be reduced further by use of β and η reduction.
For more details about properties of the λ-calculus and the rewriting systems the reader could be refer to [1].

Order The **order** of the simple type τ, is denoted by the function ord(τ) and is defined inductively over the structure of the simple type:

$$\begin{aligned} \text{ord}(\tau) &= 0 \qquad \text{if } \tau \text{ is non functional} \\ \text{ord}(\sigma \to \tau) &= \max(\text{ord}(\sigma) + 1, \text{ord}(\tau)) \end{aligned}$$

The first higher-order processing implemented in the veriT solver is β-reduction with respect to the above definitions. Therefore, all terms must be in normal-form before being handled by the solver itself. Then, we have considered how to handle partially applied terms. A partial application, provides strictly less arguments than the full number excepted by the function. One example of this is given in section 3. This phenomenon can not appear in first-order for the simple reason that the **order** of each function is at most one. Therefore, any function can't take function in argument. However, the way to avoid this particular form is to use systematically the higher-order congruence propertie on partially applied terms.

Rewriting with functional congruence The higher-order logic with equality is obtained by adding a family of constant symbols $(=_{\tau\to\tau\to o})$. Beyond that, a stronger sense of equality is obtained if some form of Leibniz's law is added as an axiom. The axiom states that two things are equal if they have all and only the same properties. Formally:

$$\forall(x\, y : \tau).\ x = y \ \Rightarrow\ \forall(P : \tau \to o)\ P\, x \Leftrightarrow P\, y \qquad \text{(Leibniz's)}$$

But, instead of considering Leibniz's law as an axiom, it can also be taken as the definition of equality. This is what we do. Thanks to the above properties it follows:

$$\forall(f : \sigma \to \tau)\forall(g : \sigma \to \tau)\ f = g \Rightarrow (\forall(x : \sigma)\ f\, x = g\, x) \qquad \text{(hocong)}$$

Which is the right to left side of the extensionality axiom (generally deduced from the substitutivity property). Assuming extensionality we deduce the following rewriting rules:

$$\forall(f g : \tau \to \sigma)\ (f = g) = \forall(x : \tau)(f\, x = g\, x) \qquad (3^{\alpha\beta})$$

One consequence of the above equality, where A and B are terms of type τ is:

$$(\lambda\overline{x}_n.A) = (\lambda\overline{x}_n.B) = \forall\overline{x}_n.\ A = B \qquad (3^{\lambda})$$

Sketch. The proof [2] is by induction n:

CASE n = 1: $(\lambda x.A) = (\lambda x.B) = \forall x.\ A = B$ follow by $3^{\alpha\beta}$

CASE n + 1: $(\lambda\overline{x}_n.(\lambda x.A)) = (\lambda\overline{x}_n.(\lambda x.B)) = \forall\overline{x}_n.(\lambda x.A) = (\lambda x.B)$ by inductive hypothesis □

Let the one hole context formula be defined as follows where t is a λ-term as defined in 2 and ψ a well-formed formula:

$$C\ ::=\ [] \ \mid\ \neg C \ \mid\ C \oplus \psi \ \mid\ \psi \oplus C \ \mid\ \forall x.C \ \mid\ \exists x.C$$

Then to ensure that our transformation is correct we prove the following theorem

Theorem 4.1. *For all context C and function symbols f, g: $C[f = g]$ is satisfiable iff $C[\forall\overline{x}_n.\ f\,\overline{x}_n = g\,\overline{x}_n]$ is satisfiable.*

Sketch. The proof is by induction on the context C and is performed by usage of 3^{λ} and $3^{\alpha\beta}$. □

It appears now obvious how these properties can be useful to avoid all partially applied terms. As before, we provide an abstract view of the usage of these properties in order to simplify higher-order formulas: we call `preprocess` the function which takes a formula and uses the properties above to avoid partial-application; `reduce` is the function which takes a term and applies the β-reduction until normal-form; `get_type` is the function which takes a term and returns its type; `eq_type` stands for the equality between two types; `arity` returns the arity of the type; `mk_freshs` takes the type and the list of arguments of the function and returns the list of missing ones (e.g if f is a function symbol of type $\sigma_1 \to \sigma_2 \to \sigma_3 \to \tau$ and t is a term of type σ_1 then the result of `mk_freshs f [t]` is `([x;y],2)` where $x : \sigma_2$ and $y : \sigma_3$); `mk_forall` takes a list of variables and a formula and returns a quantified representation of the formula — i.e **`Forall of`** `var list * formula`. Then the following Ocaml code expresses in simpler terms exactly how this procedure was implemented in veriT. We give only the cases for equality, implication and conjunction respectively denoted by the constructors **`Eq of`** `term * term`, **`Impl of`** `formula * formula`, **`And of`** `formulas * formulas`. The other cases are treated in the same way:

```
1 let rec preprocess : formula -> formula = function
2     | Eq(e1,e2) as eq ->
3     match e1, e2 with begin
4       | App(FunId f,args_f), App(FunId g,args_g)
5         when eq_type (get_type f) (get_type g) ->
6           let fresh_args_f,delta_f =
```

```
7               mk_freshs (get_type f) args
8           let fresh_args_g,delta_g =
9               mk_freshs (get_type g) args
10           mk_forall fresh_args_g (Eq(App(FunId(f),args_f@fresh_args_f),
11                                      App(FunId(g),args_g@fresh_args_g))) end
12        (** works in the same way for the symmetric case-i.e
13            partial = constant, constant = partial,... and lambda abstraction**) end
14    | Impl(f1,f2) -> Impl(preprocess f1, preprocess f2)
15    | And(f1,f2) -> Impl(preprocess f1, preprocess f2)
```

Skolemization and choice In the solving step, several methods can be used to encode quantifier dependencies. Of course, one must be careful to obtain a sound framework. For instance, in Isabelle the existential quantifiers are encoded by using ϵ-terms [32]. Typically, we have an additional constant symbol in our signature that we write ϵx. Then we add to the theory these two following axiom schemas where A is a formula:

$$\exists(x:\tau)A[x] \Leftrightarrow A(\epsilon x A[x]) \qquad \forall(x:\tau)A[x] \Leftrightarrow A(\epsilon x \neg A[x])$$

The epsilon ϵ-term expresses the witness of the quantified formula. Another way to deal with existential quantifiers is the Skolemization procedure. Together with the result of the Herbrand theorems this technique allows one to prove the (un)satisfiability of existentially quantified formulas. In particular, this method makes possible to create a ground model, also called Herbrand model, which is at the origin of many instantiation techniques used in the SMT solvers. Without going back to semantic considerations, below we give in really simple terms the principle of Skolemization. Given the following formula, where p is a predicate symbol with $n+1$ arguments:

$$\forall(x_1:\tau_1)\ldots\forall(x_n:\tau_n)\ \exists(y:\sigma)\ p(x_1,\ldots,x_n,y)$$

We call a Skolem extension of our theory $\mathcal{T}$, the new theory built by adding a new function symbol f of arity n, called the **Skolem symbol**. The formula obtained after Skolemization is:

$$\forall(x_1:\tau_1)\ldots\forall(x_n:\tau_n)\ p(x_1,\ldots,x_n,f(x_1,\ldots,x_n))$$

For more details about Herbrand models and Skolemization the reader can refer to [2]. The existence of a Skolem function follows from the choice axiom. This axiom, is inherited from the set theory. If we choose to accept this axiom in the set theory, we get the so called theory **ZFC**. One way of articulating the axiom of choice is to say that there exists a function (called the choice function) which from a collection of sets, selects an element in each of these sets. Formally, let R be a relation between two elements of types σ and τ:

$$\forall(x_1:\sigma)\exists(y:\tau)\ R(x,y) \Rightarrow \exists(f:\sigma\to\tau)\forall(z:\sigma)R(z,f(z))$$

Since we are in classical logic, we assume the axiom of choice. Yet, it is important to note that other systems may not: in constructive logics (e.g. Coq, agda) this axiom is not assumed. Then, it follows that the Skolem symbols should not be allowed without any restriction (see e.g [31]). There are some other problems with Skolemization. In particular, when a type may have zero or one inhabitant prior to Skolemization and it may have an infinite number of inhabitants afterwards. Another problems can appear during the unification of typed λ-terms those are describes in [11, 9].

Conclusions and implementation In accordance with the previous points, we provide a pre-treatment for higher-order formulas. So far, this has been implemented and tested in the veriT solver over more than fifty benchmarks. Some of them come from Sledgehammer [13] (and are translated by hand to our new syntax) from Isabelle formalizations. With the given method, we got a correct answer on some of our examples. But since we didn't provide yet an implementation of the instantiation technique, it is of course not excepted to get answers for the ones that need instantiation. The downside of the approach described here is the systematic introduction of new quantifiers in the original formula. This results in an overhead due to more calls to the instantiation module. The instantiation in SMT is a challenging problem

generally tackled with considerable cost. Nevertheless, the usage of several heuristics help to minimize this cost. In conclusion, it is common sense to add new symbols almost for free (and consequently more work for the instantiation module) should be unacceptable. This is a temporary solution, while waiting for a full higher-order update of the veriT source code.

5 A decision procedure for a higher-order QF_UF theory

The QF_UF theory is related to the equality problem over (quantifier free) uninterpreted function symbols. We present here an extension of the algorithm of Nelson and Oppen [35]. This decision procedure uses the congruence closure algorithm based on the union-find data structure. This method implements the equality theory that we'll give bellow.

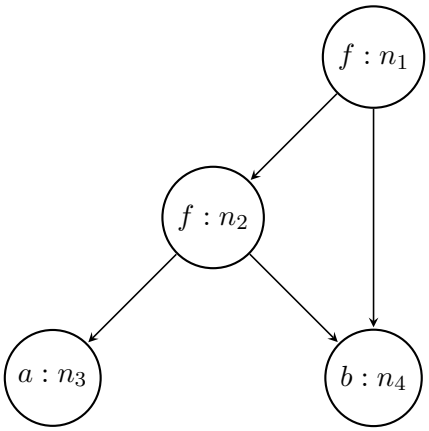
Theory of equality The equality theory is defined axiomatically as follows, where f is a constant symbol in the theory: Let $\simeq$ be a binary relation, an equivalence closure of $\simeq$ is the smallest relation closed under reflexivity, symmetry, transitivity which is an **equivalence relation**. It follows that an **equivalence class** is a set $\mathbf{T}$ of terms closed under equivalence relation. In particular, we denote by $[t]$ the **representative** term of its class. Notably, thanks to this construction we get that two terms t, u, are equals (i.e $t \simeq u$) iff $[t]$ and $[u]$ are equals (i.e $[t] \simeq [u]$). Then the **congruence closure** of a relation is the smallest relation that is closed under **equivalence** and **congruence**. **Congruence class** is the set E^{cc} of terms closed under congruence closure.

$$\forall(x : \tau)\ x = x \qquad \text{(reflexivity)}$$
$$\forall(xy : \tau)\ x = y \Rightarrow y = x \qquad \text{(symmetry)}$$
$$\forall(xyz : \tau)\ (x = y \Rightarrow y = z) \Rightarrow x = z \qquad \text{(transitivity)}$$
$$\forall(xyz : \tau)\ (x = y) \Rightarrow fx = fz \qquad \text{(congruence)}$$

Computing the congruence closure As we have seen above, the congruence closure provides a suitable structure to rank similar objects among themselves. This structure can be used to establish which terms are equal and consequently provide a way to gather them all together. To reach this goal, we will give ourselves an abstract representation of terms, which is called **DAGs** — i.e Directed Acyclic Graphs. Usually, the DAG representation enjoys the sharing property. While a tree data structure includes many duplications of a node, DAG data structure contains only one occurrence of each node. This means that during the construction of the structure, a fresh node is generated if and only if it does not appear in the structure. Generally, each node has a unique identifier stored in a table interrogated before any production of new node. Adapted to the terms we call this kind of representation a **maximum sharing of terms**. We encode a term in a graph structure where nodes represent an application (or in uncurried form a function symbol of n-arity respectively by a node with n branches) and leafs describe variable or constant symbols — i.e 0-arity. Typically, we can represent the term $f(f(a, b), b)$ by the following DAG in the right figure bellow. Toward an equality algorithm, we give to each node a unique identifier, as in the figure. Formally, let $G = (V, E)$ be a graph, the set of identifiers is represented by V and the set of edges E describes the equivalence relation $\simeq$ between the nodes. We define two operations overs this structure: for all terms u, v, `union u v` combines the equivalence class of u with the class of v; `find u` takes a term and returns its representative. Now, we have enough material to build the relation $\simeq$. For example, taking the term $f(f(a, b), b)$ represented by the right DAG right figure below. The proposition $f(f(a, b), b) = a$ can be written as the relation $\{(n_1, n_3)\}$. Now, if we want a decision procedure we start by building the initial relation containing all sub-terms of the formula. These sub-terms are initially in relation with themselves. For example, the proposition $f(a, b) = a \wedge f(f(a, b), b) = a$ we build the initial relation that is $\simeq = \{\{n_3\}, \{n_4\}, \{n_2\}, \{n_1\}\}$ which is the representation of $\{\{a\}, \{b\}, \{f(a, b)\}, \{f(f(a, b), b)\}\}$. First we take $f(a, b) = a$ that is $\simeq \cup \{n_3, n_2\}$, then we deduce $\{n_3, n_2\}$ and $\{n_4, n_4\}$ by congruence $\{n_1, n_2\}$, then $\{n_3, n_1\}$, finally the relation is $\simeq = \{\{n_3, n_2, n_1\}, \{n_4\}\}$ that is the representation of $\{\{a, f(a, b), f(f(a, b), b)\}, \{b\}\}$. Finally, if $\simeq$ is a relation on the nodes of G, that $\simeq$ is closed under congruence and u, v are nodes of G then, the function `merge u v` builds the congruence closure by adding (u, v) and all its outbuildings at the current relation $\simeq$. The Ocaml code given at the left describe the complete

algorithm to compute the congruence closure. The function `ccpar u` computes the set `pu` of all predecessors of all nodes equivalents to u — i.e informally this function computes the **c**ongruence **c**losure **p**arents which are the parents of all members of the class of u. For example, assume that our relation is the result of the previous example and on the diagram above we choose $u = n_3$. Then by the previous analyse n_2 and n_3 are equivalent then the predecessor of n_3 is n_2 and of n_3 is n_1. Thus $\texttt{pu} = \{n_2; n_1\}$. Trivially, the function `congruent u v` checks if u, v have all their children in common.

```
let rec merge u v =
if find u <> find v
then begin
 let pu = ccpar u in
 let pv = ccpar v in
 union u v;
 List.iter (
    fun t1 ->
    List.iter (
       fun  t2 ->
       if (find t1) <> (find t2)
         && congruent t1 t2 then
        merge t1 t2
      ) (IdSet.elements pv)
   ) (IdSet.elements pu)
 end
```



Higher-order congruence closure Similarly, we can build the higher-order congruence closure of two terms u, v by adding the property **hocong** to our equality theory. In fact, while the first-order congruence property only allows us to reason between the arguments of the same function, the higher-order congruence hocong is stronger and allows one to reason on different functions. In particular, it makes possible to deduce more sub-terms in the relation $\simeq$. Therefore, we can exploit equality between functional symbols. From this observation, it becomes easy to see that to extend the decision procedure described above it is enough to consider the full application of functions as a particular case. If we have an equality which involves partial application for some symbol f, then we merge all the children of f which appear in the relation $\simeq_+$ from its current arity. We denote by $\simeq_+$ the higher-order closure, which is simply the congruence closure plus the hocong property given above. Suppose we have the following equalities, with $(f : (\tau \to \tau) \to \tau \to \tau)$,$(g : (\tau \to \tau) \to \tau \to \tau)$, $(a : \tau \to \tau)$, $(b : \tau)$, $(x : \tau)$: let $E_1 : f \simeq_+ g$; $E_2 : (f\ a\ b) \simeq_+ x$; $E_3 : (g\ a\ b) \simeq_+ (a\ b)$. By hocong from E_1 we deduce $(f\ a\ b) \simeq_+ (g\ a\ b)$ then by transitivity we deduce $a\ b \simeq_+ x$. Furthermore, if we add the conjunctive proposition $E_1 \wedge E_2 \wedge E_3$ the disequality $\neg(f\ a \simeq_+ g\ a)$, then the proposition becomes unsatisfiable. The reason is simple. If we add $\neg(f\ a \simeq_+ g\ a)$ then by E_1 all sub-terms of the shape $f\ \overline{x}_i$ should be equal to all sub-terms $g\ \overline{x'}_i$. More precisely, we have $f\ \overline{x}_i \in [g\ \overline{x'}_i]$ and $g\ \overline{x}_i \in [f\ \overline{x'}_i]$ which is absurd since $(f\ a) \simeq_+ (g\ a)$. By consequence, the higher-order closure allow us to refute the proposition $E_1 \wedge E_2 \wedge E_3 \wedge \neg(f\ a \simeq_+ g\ a)$. It's enough to consider for all equalities $f\ \overline{x}_i \simeq g\ \overline{y}_i$, all occurrences of sub-terms t, t' in $\simeq_+$ such that t has the shape $f\ \overline{x'}_n$ and t' is $g\ \overline{y'}_n$ with $i \leq n$. We consider the sets: $E_f = \{\ t \mid \bigcup_{t \in \simeq_+} symbol(t) = f \wedge i \leq arity\ t\}$ and $E_g = \{\ t' \mid \bigcup_{t' \in \simeq_+} symbol(t') = g \wedge i \leq arity\ t'\}$. Let $[f^i]$, $[g^i]$, respectively denote the classes of terms with the shape: $f\ \overline{x'}_i$ and $g\ \overline{y'}_i$. Then from E_f and E_g we build for all t, t' the new class $[f^i] \cup [g^i]$ and adding at $\simeq_+$ all pairs $(f\ \overline{x'}_i, g\ \overline{y'}_i)$. Therefore, all partial application of the same arity become congruent.

Now, it remains only to add this condition in the code of `merge u v`. The function `ecchil u (arity u)` built exactly the set E_u — i.e Closure Children. The function `ho_congruent u v` check if the terms u, v have the same parents or if the terms have the same children — i.e we consider a hierarchy of terms in the Dags structure, for example the term f is the parent of $f\ a$ and $f\ a$ of $f\ a\ b$ see example below. Finally, the function `arity u` given a term returns its arity, in other word return the number of arguments applied to the symbol function at the head of the term.

```
let rec merge u v =
if find u <> find v then begin
 let cu = cchil u (arity u) in
 let cv = cchil v (arity v) in
 let pu = ccpar u in
 let pv = ccpar v in
 union u v;
 List.iter ( fun t1 ->
    List.iter ( fun  t2 ->
       if (find t1) <> (find t2)
       && ho_congruent t1 t2 then
          merge t1 t2
      ) (IdSet.elements (IdSet.union pv cv))
   ) (IdSet.elements (IdSet.union pu cu))
  end
```

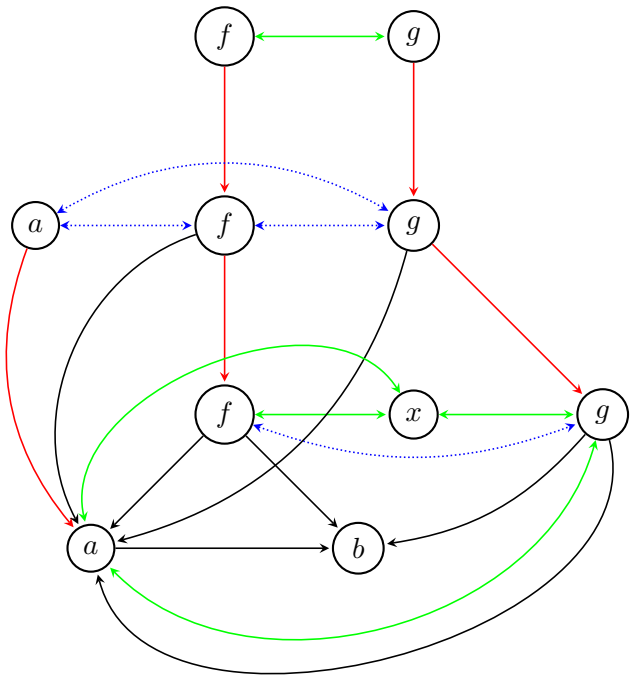


Figure 5.1 – At the left the Dags representation of $f \simeq_+ g \wedge (f\ a\ b) \simeq_+ x \wedge (g\ a\ b) \simeq_+ (a\ b)$

congruent parent
higher-order closure relation
structure of term
equalities

6 Proof producing

Regrettably, there is no standard yet for proof output; each proof-producing solver has its own proof format. We focus in this section on the proof format of the SMT solver veriT [18]. This solver is known for its very detailed proofs [12, 5], which are reconstructed in the proof assistants Coq [3] and Isabelle/HOL [16] and in the GAPT system [22]. Proofs in veriT accommodate the formula processing and the proof search performed by the solver. Processing steps are represented using an extensible set of inference rules described in Barbosa et al. [5]. Here, we extend the processing calculus by Barbosa et al. to support transformations such as β-reduction and congruence with λ-abstractions, which are required by the new constructs that can appear in higher-order problems (Section 6.1).

The CDCL($\mathcal{T}$) reasoning performed by veriT is represented by a resolution proof, which consists of the resolution steps performed by the underlying SAT solver and the lemmas added by the theory solvers and the instantiation module. These steps are described in Besson et al. [12]. The part of the proof corresponding to the actual proving will change according to how we solve higher-order problems. In keeping with the CDCL($\mathcal{T}$) setting, the reasoning is performed in a stratified manner. Currently, the SAT solver handles the propositional reasoning, a combination of theory solvers tackle the ground (variable-free) reasoning, and an instantiation module takes care of the first-order reasoning. Our initial plan is to adapt the instantiation module so that it can heuristically instantiate quantifiers with functional variables, and to extend veriT's underlying modular engine for computing substitutions [7]. Since only modifications to the instantiation module are planned, the only rules that must be adapted are those concerned with quantifier instantiation:

$$\frac{}{\forall x.\ \varphi[x] \to \varphi[t]}\ \textsc{Inst}_\forall \qquad \frac{}{\varphi[t] \to \exists x.\ \varphi[x]}\ \textsc{Inst}_\exists$$

These rules are generic enough to be suitable also for higher-order instantiation. Here, we focus on adapting the rules

necessary to suit the new higher-order constructs in the formula processing steps.

6.1 An Extension for the veriT Proof Format

Our setting is classical higher-order logic as defined by the extended SMT-LIB language above, or abstractly described by the following grammar:

$$M \;::=\; x \mid c \mid M\ M \mid \lambda x.\, M \mid \text{let } \bar{x}_n \simeq \overline{M}_n \text{ in } M$$

where formulas are terms of Boolean type. For λ-abstraction, we can now rely on the following axiom, where $\simeq$ denotes the equality predicate:

$$\models \exists x.\varphi[x] \Rightarrow \varphi[\epsilon x.\varphi] \tag{ϵ_1}$$
$$\models (\forall x.\varphi = \psi) \Rightarrow (\epsilon x.\varphi) = (\epsilon x.\psi) \tag{ϵ_2}$$
$$\models (\text{let } \bar{x_n} = \bar{s_n} \text{ in } t[\bar{x_n}]) = t[\bar{s_n}] \tag{let}$$
$$\models (\lambda x.t[x])\ s = t[s] \tag{β}$$

For readability, and because it is natural with a higher-order calculus, we present the rules in curried form—that is, functions can be partially applied, and rules must only consider unary functions.
The notion of context is as in Barbosa et al.:

$$\Gamma \;::=\; \varnothing \mid \Gamma,\, x \mid \Gamma,\, \bar{x}_n \mapsto \bar{t}_n$$

Each context entry either *fixes* a variable x or defines a *substitution* $\{\bar{x}_n \mapsto \bar{s}_n\}$. Any variables arising in the terms $\bar{s}_n$ will normally have been introduced in the context Γ on the left. If a context introduces the same variable several times, the rightmost entry shadows the others. Abstractly, a context Γ fixes a set of variables and specifies a substitution $subst(\Gamma)$ defined by:

$$subst(\varnothing) = \{\}, subst(\Gamma,\, x) = subst(\Gamma)[x \mapsto x], and subst(\Gamma,\, \bar{x}_n \mapsto \bar{t}_n) = subst(\Gamma) \circ \{\bar{x}_n \mapsto \bar{t}_n\}.$$

The substitution is the identity for $\varnothing$ and is defined as follows in the other cases:

$$subst(\Gamma,\, x) = subst(\Gamma)[x \mapsto x] \qquad subst(\Gamma,\, \bar{x}_n \mapsto \bar{t}_n) = subst(\Gamma) \circ \{\bar{x}_n \mapsto \bar{t}_n\}$$

The $[x \mapsto x]$ update shadows any replacement of x induced by Γ. The examples below illustrate this subtlety:

$$subst(x \mapsto 7,\, x \mapsto \mathsf{g}(x)) = \{x \mapsto \mathsf{g}(7)\} \qquad subst(x \mapsto 7,\, x,\, x \mapsto \mathsf{g}(x)) = \{x \mapsto \mathsf{g}(x)\}$$

We write $\Gamma(t)$ to abbreviate the capture-avoiding substitution $subst(\Gamma)(t)$.
Transformations of terms (and formulas) are justified by judgments of the form $\Gamma \rhd t \simeq u$, where Γ is a context, t is an unprocessed term, and u is the corresponding processed term. The free variables in t and u must appear in the context Γ. Semantically, the judgment expresses the equality of the terms $\Gamma(t)$ and u for all variables fixed by Γ. Crucially, the substitution applies only on the left-hand side of the equality. The inference rules for the transformations covered in this report are presented below, followed by explanations.

$$\frac{}{\Gamma \rhd t \simeq u}\ \textsc{Taut}_{\mathcal{T}} \quad \text{if } \models_{\mathcal{T}} \Gamma(t) \simeq u \qquad \frac{\Gamma \rhd s \simeq t \quad \Gamma \rhd t \simeq u}{\Gamma \rhd s \simeq u}\ \textsc{Trans} \quad \text{if } \Gamma(t) = t$$

$$\frac{\left(\Gamma \rhd t_i \simeq u_i\right)_{i=1}^{n}}{\Gamma \rhd \mathsf{f}(\bar{t}_n) \simeq \mathsf{f}(\bar{u}_n)}\ \textsc{Cong} \qquad \frac{\Gamma,\, y,\, x \mapsto y \rhd \varphi \simeq \psi}{\Gamma \rhd (Qx.\,\varphi) \simeq (Qy.\,\psi)}\ \textsc{Bind} \quad \text{if } y \notin FV(Qx.\,\varphi)$$

$$\frac{\Gamma,\, x \mapsto (\varepsilon x.\,\varphi) \rhd \varphi \simeq \psi}{\Gamma \rhd (\exists x.\,\varphi) \simeq \psi}\ \textsc{Sko}_{\exists} \qquad \frac{\Gamma,\, x \mapsto (\varepsilon x.\,\neg\varphi) \rhd \varphi \simeq \psi}{\Gamma \rhd (\forall x.\,\varphi) \simeq \psi}\ \textsc{Sko}_{\forall}$$

$$\frac{\left(\Gamma \rhd r_i \simeq s_i\right)_{i=1}^{n} \quad \Gamma,\, \bar{x}_n \mapsto \bar{s}_n \rhd t \simeq u}{\Gamma \rhd (\text{let } \bar{x}_n \simeq \bar{r}_n \text{ in } t) \simeq u}\ \textsc{Let} \quad \text{if } \Gamma(s_i) = s_i \text{ for all } i \in [n]$$

Our new set of rules is similar to Barbosa et al. The rules TRANS, SKO$_\exists$, SKO$_\forall$, LET, and TAUT$_\mathcal{T}$ apply unchanged. The BIND rule is modified to accommodate the new λ-binder:

$$\frac{\Gamma,\, y,\, x \mapsto y \rhd \varphi \simeq \psi}{\Gamma \rhd (B x.\, \varphi) \simeq (B y.\, \psi)}\ \textsc{Bind} \quad \text{if } y \notin FV(B x.\, \varphi)$$

The metavariable B ranges over $\forall$, $\exists$, and λ. Another rule is modified to suit new cases. With respect to the first-order calculus, the left hand side of an application is not always a simple function or predicate symbol anymore, but can involve more complicated terms. Rewriting can now occur also on these complicated terms. The updated CONG rule is as follows:

$$\frac{\Gamma \rhd t \simeq s \quad \Gamma \rhd u \simeq v}{\Gamma \rhd t\, u \simeq s\, v}\ \textsc{Cong}$$

The only genuinely new rule handles β-reduction—that is, the substitution of an argument in the body of a λ-abstraction. It is similar in form to the LET rule from the first-order calculus [5]:

$$\frac{\Gamma \rhd v \simeq s \quad \Gamma,\, x \mapsto s \rhd t \simeq u}{\Gamma \rhd (\lambda x.\, t)\, v \simeq u}\ \textsc{Beta} \quad \text{if } \Gamma(s) = s$$

Indeed, $(\text{let } x \simeq u \text{ in } t)$ and $(\lambda x.\, t)\, u$ are equal semantically.

Example 6.1. The derivation tree of the normalization of $(\lambda x.\, p\ x\ x)\ a$ is as follows:

$$\dfrac{\dfrac{}{\rhd a \simeq a}\ \textsc{Cong} \qquad \dfrac{\dfrac{\dfrac{}{x \mapsto a \rhd p \simeq p}\ \textsc{Taut}_\mathcal{T} \quad \dfrac{}{x \mapsto a \rhd x \simeq a}\ \textsc{Refl}}{x \mapsto a \rhd p\ x \simeq p\ a}\ \textsc{Cong} \quad \dfrac{}{x \mapsto a \rhd x \simeq a}\ \textsc{Refl}}{x \mapsto a \rhd p\ x\ x \simeq p\ a\ a}\ \textsc{Cong}}{\rhd (\lambda x.\, p\ x\ x)\ a \simeq p\ a\ a}\ \textsc{Beta}$$

Example 6.2. The following tree features a β-redex under a λ-abstraction. Let $\Gamma_1 = w,\, x \mapsto w$; $\Gamma_2 = \Gamma_1,\, y \mapsto f\ w$; and $\Gamma_3 = \Gamma_2,\, z \mapsto f\ w$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma_1 \rhd f \simeq f}\ \textsc{Taut}_\mathcal{T} \quad \dfrac{}{\Gamma_1 \rhd x \simeq w}\ \textsc{Refl}}{\Gamma_1 \rhd f\ x \simeq f\ w}\ \textsc{Cong} \quad \dfrac{\dfrac{}{\Gamma_2 \rhd y \simeq f\ w}\ \textsc{Refl} \quad \dfrac{}{\Gamma_3 \rhd p\ z \simeq p\ (f\ w)}\ \textsc{Refl}}{\Gamma_2 \rhd (\lambda z.\, p\ z)\ y \simeq p\ (f\ w)}\ \textsc{Beta}}{\Gamma_1 \rhd (\lambda y.\, (\lambda z.\, p\ z)\ y)\ (f\ x) \simeq p\ (f\ w)}\ \textsc{Beta}}{\rhd (\lambda x.\, (\lambda y.\, (\lambda z.\, p\ z)\ y)\ (f\ x)) \simeq (\lambda w.\, p\ (f\ w))}\ \textsc{Bind}$$

Example 6.3. The transitivity rule is useful when the applied term reduces to a λ-abstraction. Let $\Gamma_1 = w,\, y \mapsto w$; $\Gamma_2 = \Gamma_1,\, x \mapsto w$; $\Gamma_3 = \Gamma_1,\, w_1 \mapsto p\ w$; $\Gamma_4 = \Gamma_1,\, x \mapsto \lambda w_1.\, w$; $\Gamma_5 = \Gamma_1,\, w_1,\, x \mapsto w_1$; and $\Gamma_6 = \Gamma_4,\, z \mapsto \lambda w_1.\, w$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma_1 \rhd y \simeq w}\ \textsc{Refl} \quad \dfrac{}{\Gamma_2 \rhd p\ x \simeq p\ w}\ \textsc{Refl}}{\Gamma_1 \rhd (\lambda x.\, p\ x)\ y \simeq p\ w}\ \textsc{Beta} \quad \Pi}{\Gamma_1 \rhd ((\lambda x.\, (\lambda z.\, z)\ x)\ (\lambda x.\, y))\ ((\lambda x.\, p\ x)\ y) \simeq (\lambda w_1.\, w)\ (p\ w)}\ \textsc{Cong} \quad \dfrac{\dfrac{}{\Gamma_1 \rhd p\ w \simeq p\ w}\ \textsc{Refl} \quad \dfrac{}{\Gamma_3 \rhd w \simeq w}\ \textsc{Cong}}{\Gamma_1 \rhd (\lambda w_1.\, w)\ (p\ w) \simeq w}\ \textsc{Beta}}{\Gamma_1 \rhd ((\lambda x.\, (\lambda z.\, z)\ x)\ (\lambda x.\, y))\ ((\lambda x.\, p\ x)\ y) \simeq w}\ \textsc{Trans}}{\rhd (\lambda y.\, (\lambda x.\, (\lambda z.\, z)\ x)\ (\lambda x.\, y))\ ((\lambda x.\, p\ x)\ y)) \simeq (\lambda w.\, w)}\ \textsc{Bind}$$

where Π stands for the subtree

$$\dfrac{\dfrac{\dfrac{}{\Gamma_5 \rhd y \simeq w}\ \textsc{Refl}}{\Gamma_1 \rhd (\lambda x.\, y) \simeq (\lambda w_1.\, w)}\ \textsc{Bind} \quad \dfrac{\dfrac{}{\Gamma_4 \rhd x \simeq (\lambda w_1.\, w)}\ \textsc{Refl} \quad \dfrac{}{\Gamma_6 \rhd z \simeq (\lambda w_1.\, w)}\ \textsc{Refl}}{\Gamma_4 \rhd (\lambda z.\, z)\ x \simeq (\lambda w_1.\, w)}\ \textsc{Beta}}{\Gamma_1 \rhd (\lambda x.\, (\lambda z.\, z)\ x)\ (\lambda x.\, y) \simeq (\lambda w_1.\, w)}\ \textsc{Beta}$$

The correctness of the extended calculus is a simple extension of the correctness proof in Barbosa et al. We focus on the extensions. Recall that the proof uses an encoding of terms and context in λ-calculus, based on the following grammar:

$$M \ ::= \ \boxed{t} \ \mid \ (\lambda x.M) \ \mid \ (\lambda \bar{x}_n.M)\ \bar{t}_n$$

As previously $\mathit{reify}(M \simeq N)$ is defined as $\forall \bar{x}_n.\ t \simeq u$ if $M =_{\alpha\beta} \lambda x_1 \ldots \lambda x_n.\boxed{t}$ and $N =_{\alpha\beta} \lambda x_1 \ldots \lambda x_n.\boxed{u}$. The encoded rules are as follows:

$$\frac{M[u] \simeq N[v] \quad M[t] \simeq N[s]}{M[t\,u] \simeq N[s\ v]}\ \textsc{Cong} \qquad \frac{M[\lambda y.\ (\lambda x.\ \varphi)\ y] \simeq N[\lambda y.\ \psi]}{M[Bx.\ \varphi] \simeq N[By.\ \psi]}\ \textsc{Bind} \quad \text{if } y \notin FV(Bx.\ \varphi)$$

$$\frac{M[v] \simeq N[s] \quad M[(\lambda x.\ t)\ s] \simeq N[u]}{M[(\lambda x.\ t)\ v] \simeq N[u]}\ \textsc{Beta} \quad \text{if } M[v] =_{\alpha\beta} N[s]$$

Lemma 6.4. *If the judgment $M \simeq N$ is derivable using the encoded inference system with the theories $\mathcal{T}_1 \ldots \mathcal{T}_n$, then $\models_{\mathcal{T}} \mathit{reify}(M \simeq N)$ with $\mathcal{T} = \mathcal{T}_1 \cup \cdots \cup \mathcal{T}_n \cup \simeq \cup\ \epsilon \cup \mathrm{let} \cup \beta$.*

Proof. The proof is by induction over the derivation $M \simeq N$. We only provide here the three new cases:

Case Bind $B = \lambda$: The induction hypothesis is $\models_{\mathcal{T}} \mathit{reify}(M[\lambda y.\ (\lambda x.\ \varphi[x])\ y] \simeq N[\lambda y.\ \psi[y]])$. Using (β) and the side condition of the rule, we can also deduce that $\models_{\mathcal{T}} \mathit{reify}(M[\lambda y.\ \varphi[y])] \simeq N[\lambda y.\ \psi[y]])$. Hence by α-conversion this is equivalent to $\models_{\mathcal{T}} \mathit{reify}(M[\lambda x.\ \varphi[x])] \simeq N[\lambda y.\ \psi[y]])$.

Case Cong: This case follows directly from equality in a higher-order setting.

Case Beta: This case follows directly from (β) and equality in a higher-order setting.

The remaining cases are similar to Barbosa et al. □

The auxiliary functions $L(\Gamma)[t]$ and $R(\Gamma)[u]$ are used to encode the judgment of the original inference system $\Gamma \rhd t \simeq u$. They are defined over the structure of the context, as follows:

$$\begin{aligned} L(\varnothing)[t] &= \boxed{t} & R(\varnothing)[u] &= \boxed{u} \\ L(x,\ \Gamma)[t] &= \lambda x.\ L(\Gamma)[t] & R(x,\ \Gamma)[u] &= \lambda.\ L(\Gamma)[u] \\ L(\bar{x}_n \mapsto \bar{s}_n, \Gamma)[t] &= (\lambda \bar{x}_n.\ L(\Gamma)[t])\ \bar{s}_n & R(\bar{x}_n \mapsto \bar{s}_n, \Gamma)[u] &= (\lambda \bar{x}_n.\ L(\Gamma)[u])\ \bar{s}_n \end{aligned}$$

Lemma 6.5. *If the judgment $\Gamma \rhd t \simeq u$ is derivable using the original inference system, the equality $L(\Gamma)[t] \simeq R(\Gamma)[u]$ is derivable using the encoded inference system.*

Proof. The proof is by induction over the derivation $\Gamma \rhd t \simeq u$, we give only the three new cases:

Case Bind with $B = \lambda$: The encoded antecedent is $M[\lambda y.\ (\lambda x.\ \varphi)\ y] \simeq N[\lambda y.\ \psi]$ (i.e., $L(\Gamma,\ y,\ x \mapsto y)[\varphi] \simeq R(\Gamma,\ y,\ x \mapsto y)[\psi]$), and the encoded succedent is $M[\lambda x.\ \varphi] \simeq N[\lambda y.\ \psi]$. By the induction hypothesis, the encoded antecedent is derivable. Thus, by the encoded Bind rule, the encoded succedent is derivable.

Case Cong: Similar to Bind.

Case Beta: Similar to Let with $n = 1$.

The remaining cases are similar to Barbosa et al. □

Lemma 6.6 (Soundness of Inferences). *If the judgment $\Gamma \rhd t \simeq u$ is derivable using the original inference system with the theories $\mathcal{T}_1 \ldots \mathcal{T}_n$, then $\models_{\mathcal{T}} \Gamma(t) \simeq u$ with $\mathcal{T} = \mathcal{T}_1 \cup \ldots \cup \mathcal{T}_n \cup \simeq \cup\ \epsilon \cup \mathrm{let} \cup \beta$.*

Proof. Using the above updated lemmas, the proof is identical to the one for the original calculus. □

7 Quantifier instantiation

Substitutions are finite mappings from variables to terms. In this section substitutions are denoted by σ. The application of σ to t is written $t\sigma$, and $t\sigma \downarrow_\beta$ express that $t\sigma$ is in β-normal form. An individual substitution is still denoted by $t\{x \mapsto t\}$ (where x could be free or not). The environments are represented by a list of substitutions at left, and equality at right. Respectively, we represent the empty list at left by $\top$, which is equivalent to the empty conjunction, and by $\bot$ at right which is equivalent to the empty disjunction. We introduce the notation $E_\sigma = \{x \simeq x\sigma \,|\, x \in dom(\sigma)\}$. We say that a formula φ is true in a model $\mathcal{M}$ (or $\mathcal{M}$ is model of φ) : $\mathcal{M} \models \varphi$.

Checking the satisfiability of quantified formulas with respect to the underlying theories is generally a hard problem. Most of my work during the internship was focused around the veriT solver. This is an example of a system based on the CDCL($\mathcal{T}$) framework [36]. In recent years, many heuristic instantiation techniques were developed around the CDCL($\mathcal{T}$) framework. Although the problem is most of the time undecidable the purpose of the game is to be able to derive a set of a ground instances of the quantified formula and afterwards to delegate all generated instances of the original formulas to an efficient ground solver. Generally, the satisfiability of a quantified formula is checked through a feedback loop which at each step activates the instantiation module until either it finds an instance which refutes the formula, finds a complete instantiation, loops forever or answers unknown. Usually, all input formulas are in Skolem form, and by consequence, all the generated instantiation lemmas are of the form:

$$\forall \overline{x}_n \in dom(\sigma).\ \psi \Rightarrow \psi\sigma$$

Currently, there are three main methods to perform quantifier instantiation in the CDCL($\mathcal{T}$) framework. These methods are sometimes more efficient on some particular problems, then generally these techniques are combined to fill the gaps in each of them.

- **Trigger based instantiation** was originally developed in the thesis of Nelson [34] and is roughly based on the observation that a universal quantifier on a variable $(x : \tau)$ can most of the times (in first order) be transformed to a conjunction of the values of the domain of τ with some shape reminiscent of the terms already appearing in the formula.

- **Conflict based instantiation** [40] was originally created to produce relevant sets of instances, unlike the trigger based instantiation method, which sometimes may produce many irrelevant instances for the solving. The idea is that given a ground model $\mathcal{M}$ and a quantified formula $\forall(\overline{x}_n : \overline{\tau}_n).\varphi$, we find a substitution σ such that $\mathcal{M} \models \neg\varphi\sigma$. In this way, once the corresponding instantiation lemmas are added, we will prevent the derivation of that same candidate model.

- **Model based instantiation** [23] was introduced to provide a complete instantiation method for CDCL($\mathcal{T}$). The basic purpose aims to build an interpretation over a ground model $\mathcal{M}$ for all uninterpreted functions — e.g if f is a function symbol then we build its interpretation for an arbitrary predicate P (usually inferred from the formula) with respect to the current theory. $[\![f^n]\!]_\mathcal{M} = \lambda\overline{x}_n.if\ P(x_1)\ then\ \top\ else \ldots\ if\ P(x_n)\ then\ \top\ else\ \bot$. If the interpretation satisfies all formulas in $\mathcal{Q}$ then we can answer SAT, $\mathcal{Q}$ stands for a set of quantified formulas.

In the following we focus on trigger based instantiation and provide an extension to tackle higher-order quantification. Notice that we aim for a higher-order logic with the general semantic, more commonly called Henkin semantics [24]. Henkin showed that higher-order logic interpreted in the general model is completely equivalent to its counter-part in first-order. Then higher-order logic with Henkin semantic is equivalent to first-order logic. However, that is the origin of the current translations employed in verification tools to turn the higher proposition into a first-order one. Thanks to this observation we can extend without worries the trigger based instantiation which relies on semantic observations.

7.1 trigger based instantiation

Trigger instantiation is a combination of two procedures. First, we select all relevant terms occurring in the quantified formula. Then, we batch these terms in a collection of sets that we call triggers. Secondly, we try to match the terms which appears in triggers with terms occurring in the ground model, in order to derive instantiation lemmas. This process is performed through E-matching.

E-matching Given a conjunctive set of equality literals E and terms u and t, with t ground, the E-matching problem is that of finding a substitution σ such that $E \models u\sigma \simeq t$.

Triggers A trigger T for a quantified formula $\forall \overline{x}_n.\psi$ is a set of non-ground terms $u_1, \ldots, u_n \in \mathbf{T}(\psi)$ such that $\{\overline{x}\} \subseteq FV(u_1) \cup \ldots \cup FV(u_n)$. Given a ground model $\mathcal{M}$ and a set of quantified formulas $\mathcal{Q}$, for each formula $\psi \in \mathcal{Q}$ we do:

1. choose a collection of trigger $T_1, \ldots, T_n$ from ψ;
2. build the set of instantiation lemma:

$$I = \bigcup_{i=1}^{n} \{\ \forall \overline{x}_n.\psi \Rightarrow \psi\sigma \mid T_i = \{u_1, \ldots, u_n\} \text{ and } t_1, \ldots, t_n \in \mathcal{M},\ \mathcal{M} \models u_1\sigma \simeq t_1,\ \ldots,\ \mathcal{M} \models u_n\sigma \simeq t_n\}$$

3. adding I to the original problem.

The substitution σ computed in the step 2 of the above procedure is the solution to the E-matching problem for the specific formula ψ. In order to compute this substitution Leonardo de Moura and Nikolaj Bjørner provided an efficient algorithm [33], given a term u in a trigger and a ground term t infer the substitution σ such that $u\sigma = t$. This algorithm has been implemented in the Z3 SMT solver. We propose to extend this algorithm to compute substitutions of higher-order formulas. Consider the well-typed literal $F(a) \simeq g(a,\, a)$, with F being a functional variable. There are four unifiers for this problem:

$$\begin{aligned}
\sigma_1 &= \{F \mapsto \lambda w.\, g(w,\, w)\} \\
\sigma_2 &= \{F \mapsto \lambda w.\, g(w,\, a)\} \\
\sigma_3 &= \{F \mapsto \lambda w.\, g(a,\, w)\} \\
\sigma_4 &= \{F \mapsto \lambda w.\, g(a,\, a)\}
\end{aligned}$$

These unifiers can be found systematically with Huet's algorithm [25]. Higher-order matching is decidable, unification is semi-decidable (if the problem has unifiers they will be found, however the procedure may loop indefinitely if no solution exists). A good alternative to Huet's algorithm, which is quite explosive, is pattern unification [29], based on subclass of λ-terms which behave almost like first-order terms. More exactly this is a decidable and deterministic fragment of λ-calculus for unification algorithm (there exist most general unifiers). This observation come first from Dale Miller [28]. Surprisingly, although restrictive this technique is still effective in practice, being the default higher-order unification procedure used in systems such as λProlog and Isabelle [39]. When considering how to handle higher-order quantifiers in CDCL($\mathcal{T}$) we have to decide how to tackle E-ground (dis)unification [7] in a higher-order context, since it is the base for the major instantiation techniques. Here we start with the simplest case: how to perform E-matching in a higher-order, belong knowing that this will be enough to perform trigger based instantiation. Moreover, we start investigating how to do so using only pattern matching, in the above sense, hoping that this fragment will be enough to yield effective procedures. We proceed by defining pattern unification in lambda calculus, then how it works in the context where quantifiers are present (HOSMT).

7.2 Pattern unification

The following observations come from Nipkow [38], which provides a simple yet efficient implementation of unification of both untyped and simply typed patterns: This can be seen as a specialised and improved version of Huet's algorithm. The fact that patterns turn out so frequently in practice is probably the reason that Huet's algorithm works so surprisingly well in practice (given its explosive nature, that is). The specialisation sums up to: imitation and projection coincide and flex-flex pairs can be solved. An interesting detail: it is possible, as shown by Miller [30], to perform full higher-order unification by means of pattern unification if one implements an external search procedure on top of it. A pattern is defined below. Furthermore, we choose to hold the same formalism of Nipkow [38] in the syntax. Accordingly, the n-applied terms $((\ldots(a\, s_1)\ldots)s_n)$ is written $a(\bar{s}_n)$.

Pattern A term t in β-normal form is a *(higher-order) pattern* if and only if every free occurrence of a variable F is in a subterm $F(\bar{u}_n)$ of t such that $\bar{u}_n$ is η-equivalent to a list of distinct bound variables.

Example 7.1. *Examples of patterns are $\lambda x.\, c(x)$, $\lambda x.\, F(\lambda z.\, x(z))$, and $\lambda xy.\, F(x,\, y)$. Examples of non-patterns are $F(c)$, $\lambda x.\, F(x,\, x)$, and $\lambda x.\, F(F(x))$. More in details the first λ-term $F(c)$ is obviously not a pattern because c is not a bounded variable. The second $\lambda x.\, F(x,\, x)$ is not a pattern because: F should be applied to a list of distinct bound variables. Finally the last one may seem well-formed pattern but we have F which is a free-variable, hence in $F(F(x))$ we have F which is not a bound variable.*

The rules for pattern unification:

$$\frac{E_\sigma \Vdash ((\lambda x.\, s) \simeq (\lambda x.\, t)) :: L}{E_\sigma \Vdash (s \simeq t) :: L} \qquad \frac{E_\sigma \Vdash (a(\bar{s}_n) \simeq a(\bar{t}_n)) :: L}{E_\sigma \Vdash [s_1 \simeq t_1,\, \ldots,\, s_n \simeq t_n] \mathbin{@} L}$$

$$\frac{E_\sigma \Vdash (F(\bar{x}_m) \simeq a(\bar{s}_n)) :: L}{E_\sigma \cup \{F \simeq (\lambda \bar{x}_m.\, a(\bar{H}_n(\bar{x}_m)))\} \Vdash [H_1(\bar{x}_m) \simeq s_1,\, \ldots,\, H_n(\bar{x}_m) \simeq s_n] \mathbin{@} L}$$

where $F \notin FV(\bar{s}_n)$ and a is constant or $a \in \{\bar{x}_m\}$

$$\frac{E_\sigma \Vdash (F(\bar{x}_m) \simeq F(\bar{y}_n)) :: L}{E_\sigma \cup \{F \simeq (\lambda \bar{x}_m.\, H(\bar{z}_p))\} \Vdash L} \qquad \frac{E_\sigma \Vdash (F(\bar{x}_m) \simeq G(\bar{y}_n)) :: L}{E_\sigma \cup \{F \simeq (\lambda \bar{x}_m.\, H(\bar{z}_p)),\, G \simeq (\lambda \bar{y}_n.\, H(\bar{z}_p))\} \Vdash L}$$

where $\{\bar{z}_p\} = \{x_i \mid x_i \simeq y_i\}$ — where $F \neq G$ and $\{\bar{z}_p\} = \{\bar{x}_m\} \cap \{\bar{y}_n\}$

Table 1 – Pattern unification

These rules has been adapted to match our needs. Then, in the above figure a judgment is composed of two elements: at the left a set of equivalents literals which is equivalent of the original presentation with substitution; at the right a list of pattern equalities which is strictly adapted to [38]. Finally, the relation $\Vdash$ stand for the entailment relation. The $@$ is the concatenation and :: perform the appending in head of the list.

Theorem 7.2. *A list of pattern equalities L has a solution iff from the judgment $\top \Vdash L$ we derive $E_\sigma \Vdash \bot$ where E_σ is the most general unifier of L.*

Proof. Completeness and correctness follows because the above rules cover all solvable cases and because the antecedent (modulo the resulting computed substitution after the rule) and succedent have the same set of unifiers. [38] □

Example 7.3. *Consider $\lambda xy.\, F(x)$ and $\lambda xy.\, c(G(y,\, x))$. The derivation is:*

$$\frac{\dfrac{\dfrac{\top \Vdash [(\lambda xy.\, F(x)) \simeq (\lambda xy.\, c(G(y,\, x)))]}{\top \Vdash [F(x) \simeq c(G(y,\, x))]}}{\{F \simeq \lambda x.\, c(H(x))\} \Vdash [H(x) \simeq G(y,\, x)]}}{\{F \simeq \lambda x.\, c(H'(x)),\, H \simeq \lambda x.\, H'(x),\, G \simeq \lambda yx.\, H'(x)\} \Vdash \bot}$$

[1] $(\lambda x.\, Fx) \simeq F$ whenever x does not appear free in F. This conversion embodies extensionality.

7.3 Pattern E-matching on HOSMT

An abstract version of the de Moura and Bjørner algorithm is presented in the following with three new rules which allow one to tackle higher-order patterns. This calculus performs higher-order E-matching, where the following call $ematch(s, t, S)$ is composed of a pattern s in the first component, a ground term t and a set S of a substitutions. Then the rules are:

$$
\begin{aligned}
ematch(x, t, S) &= \{\sigma \cup \{x \mapsto t\} \,|\, \sigma \in S,\, s \notin dom(\sigma))\} \cup \{\sigma \,|\, \sigma \in S,\, E \models x\sigma \downarrow_{beta} \simeq t\} \\
ematch(\lambda x.\, s, \lambda x.\, t, S) &= ematch(s, t, S) \\
ematch(t', t, S) &= \begin{cases} S & \text{if } E \models t' \simeq t \\ \varnothing & \text{otherwise} \end{cases} \\
ematch(a(\overline{s_n}), t, S) &= \bigcup_{a(\overline{t_n}) \in \mathbf{T}(E),\, E \models a(\overline{t_n}) \simeq t} ematch(s_n, t_n, \ldots, ematch(s_1, t_1, S)\ldots) \\
ematch(F(\overline{x_n}), t, S) &= \begin{cases} ematch(H_m(\overline{x_n}), t_m, \ldots, ematch(H_1(\overline{x_n}), t_1, S)\ldots) & \\ \bigcup_{a(\overline{t_m}) \in \mathbf{T}(E),\, E \models a(\overline{t_m}) \simeq t} \{F \mapsto \lambda\overline{x_n}.\, a(\overline{H_m(\overline{x_n})})\} & \\ \text{see the note} & \end{cases}
\end{aligned}
$$

Such that all instantiations for a quantified formula with a trigger $T = \{u_1, \ldots, u_n\}$ can be obtained with the set of substitutions:

$$S = \bigcup_{t_i \in \mathbf{T}(E)} ematch(u_n, t_n, \ldots, ematch(u_1, t_1, S)\ldots)$$

The correctness of these rules follow first from the correctness of the original subset [33] of rules, the first, the third and a sub case of the forth — i.e originally we didn't allow application of bound variables. Secondly, the correctness and completeness of the new added rules directly follows from 7.2, since E-matching is a sub problem of unification — i.e because by definition one side of the equality is ground.

8 Conclusion and Future Work

We have presented a preliminary extension of the SMT-LIB syntax, a decision procedure to ground higher order equality, an instantiation technique and an extension of the veriT proof format to support higher-order constructs in SMT proofs. Partial applications, λ-abstractions, and quantification over functional variables can now be understood by a solver compliant with these languages. For the proof production, the only relatively challenging element of these extensions so far concerns the rules for representing detailed proofs of formula processing. The next step is to extend the generic proof-producing formula processing algorithm from Barbosa et al. [5]. Given the structural similarity between the introduced extensions and the previous proof calculus, we expect this to be straightforward.

A more interesting challenge will be to reconstruct these new proofs in proof assistants, to allow full integration of a higher-order SMT solver. Since detailed proofs are produced, with proof checking being guaranteed to have reasonable complexity, we are confident to be able to produce effective implementations. Independently, we have extend the so called CCFV (i.e. **c**ongruence **c**losure **w**ith free **v**ariables) calculus [6] to λ-patterns. This calculus provides a suitable and efficient framework to find instantiations of quantified formulas. Furthermore, it one allows to handle almost all the main instantiation techniques in a flexible way. Due to the space constraint and because this is still a work in progress I preferred to avoid this part of my report. In parallel, we have planned to carry on the SMTpp [17] and Dolmen [1] projects in order to provide generic tools for syntactic manipulations with SMT and HOSMT e.g. translate TPTP [45] benchmarks. With the foundations laid down, the next step will be to implement the automatic reasoning machinery for higher-order formulas and properly evaluate its effectiveness.Moreover, when providing support for techniques involving, for example, inductive datatypes, we will need to augment the proof format. This internship was for me a good introduction to a SMT and it allowed me to get acquainted with this research field.

[1] https://github.com/Gbury/dolmen

9 Remerciments

Je tiens tout d'abord à remercier profondément Pascal fontaine et Jasmin Blanchette pour m'avoir accueilli au sein du projet Matryoshka et ainsi m'avoir donné l'opportunité d'effectuer ce stage. Je remercie personnellement Pascal pour son aide, son soutien et sa pédagogie qui au jour le jour m'a poussé à corriger mes erreurs et à évoluer. Je remercie personnellement Jasmin pour son aide dans mon travail ainsi que pour ses conseils qui m'ont aidé à me perfectionner. Je tiens à remercier Haniel pour son aide et ses conseils quotidiens. Mais aussi Simon et Martin pour leur sympathie journalière, leurs points de vue aiguisés et leurs très bonnes explications. Je remercie pour sa gentillesse Stephan Merz, et le remercie de m'avoir accueilli dans son équipe. Je remercie tous les membres de l'équipe VeriDis: Sophie, Marie, Thomas, Margaux, Tung, Punam pour leur bonne humeur quotidienne. Je souhaite finir en remerciant mes parents qui sans eux rien n'aurait été possible, je les remercie pour leur soutien et leur aide durant ces 5 années me permettant ainsi de me consacrer essentiellement à ma passion l'informatique.

References

[1] Roberto M. Amadio. "Operational methods in semantics". Lecture. Paris, France, Dec. 2016. URL: https://hal.archives-ouvertes.fr/cel-01422101.

[2] Peter B Andrews. *An introduction to mathematical logic and type theory*. Vol. 27. Springer Science & Business Media, 2002.

[3] Michaël Armand et al. "A Modular Integration of SAT/SMT Solvers to Coq through Proof Witnesses". In: *Certified Programs and Proofs*. Ed. by Jean-Pierre Jouannaud and Zhong Shao. Vol. 7086. Lecture Notes in Computer Science. Springer, 2011, pp. 135–150. DOI: 10.1007/978-3-642-25379-9_12.

[4] Leo Bachmair and Harald Ganzinger. "Rewrite-Based Equational Theorem Proving with Selection and Simplification". In: *Journal of Logic and Computation* 4.3 (1994), pp. 217–247. DOI: 10.1093/logcom/4.3.217. URL: http://dx.doi.org/10.1093/logcom/4.3.217.

[5] Haniel Barbosa, Jasmin Christian Blanchette, and Pascal Fontaine. "Scalable fine-grained proofs for formula processing". In: *Proc. Conference on Automated Deduction (CADE)*. Ed. by Leonardo de Moura. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2017.

[6] Haniel Barbosa, Pascal Fontaine, and Andrew Reynolds. *Congruence Closure with Free Variables*. Tech. rep. https://hal.inria.fr/hal-01442691. Inria, 2017.

[7] Haniel Barbosa, Pascal Fontaine, and Andrew Reynolds. "Congruence Closure with Free Variables". In: *Tools and Algorithms for Construction and Analysis of Systems (TACAS)*. Ed. by Axel Legay and Tiziana Margaria. Vol. 10206. Lecture Notes in Computer Science. 2017, pp. 214–230. DOI: 10.1007/978-3-662-54580-5. URL: http://dx.doi.org/10.1007/978-3-662-54580-5.

[8] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. *The SMT-LIB Standard: Version 2.5*. Tech. rep. Available at www.SMT-LIB.org. Department of Computer Science, The University of Iowa, 2015.

[9] Christoph Benzmüller et al. "Automation of Higher-Order Logic." In: *Computational Logic*. Vol. 9. 2014, pp. 215–254.

[10] Christoph Benzmüller et al. "The Higher-Order Prover Leo-II". In: *Journal of Automated Reasoning* 55.4 (2015), pp. 389–404. ISSN: 1573-0670. DOI: 10.1007/s10817-015-9348-y. URL: https://doi.org/10.1007/s10817-015-9348-y.

[11] Christoph E Benzmuller and Chad E Brown. "A structured set of higher-order problems". In: *TPHOLs*. Springer. 2005, pp. 66–81.

[12] Frédéric Besson, Pascal Fontaine, and Laurent Théry. "A Flexible Proof Format for SMT: a Proposal". In: *Workshop on Proof eXchange for Theorem Proving (PxTP)*. 2011.

[13] Jasmin Christian Blanchette. *Hammering Away: A Users Guide to Sledgehammer for Isabelle/HOL. 2013*.

[14] Jasmin Christian Blanchette, Uwe Waldmann, and Daniel Wand. "A lambda-free higher-order recursive path order". In: *International Conference on Foundations of Software Science and Computation Structures*. Springer. 2017, pp. 461–479.

[15] Jasmin Christian Blanchette et al. "Hammering towards QED". In: *Journal of Formalized Reasoning* 9.1 (2016), pp. 101–148.

[16] Jasmin Christian Blanchette et al. "Semi-intelligible Isar Proofs from Machine-Generated Proofs". In: *Journal of Automated Reasoning* 56.2 (2016), pp. 155–200.

[17] Richard Bonichon et al. "SMTpp: preprocessors and analyzers for SMT-LIB". In: *Proceedings of the 13th International Workshop on Satisfiability Modulo Theories (SMT 2015)*. 2015.

[18] Thomas Bouton et al. "veriT: An Open, Trustable and Efficient SMT-Solver". In: *Proc. Conference on Automated Deduction (CADE)*. Ed. by Renate A. Schmidt. Vol. 5663. Lecture Notes in Computer Science. Springer, 2009, pp. 151–156. DOI: 10.1007/978-3-642-02959-2_12. URL: http://dx.doi.org/10.1007/978-3-642-02959-2_12.

[19] Robert S Boyer and J Strother Moore. "A theorem prover for a computational logic". In: *International Conference on Automated Deduction*. Springer. 1990, pp. 1–15.

[20] Chad E. Brown. "Satallax: An Automated Higher-Order Prover". In: *6th International Joint Conference on Automated Reasoning (IJCAR 2012)*. Ed. by Ulrike Sattler Bernhard Gramlich Dale Miller. Accepted. Springer, 2012, pp. 111 –117.

[21] Alonzo Church. "A formulation of the simple theory of types". In: *The journal of symbolic logic* 5.2 (1940), pp. 56–68.

[22] Gabriel Ebner et al. "System Description: GAPT 2.0". In: *International Joint Conference on Automated Reasoning (IJCAR)*. Ed. by Nicola Olivetti and Ashish Tiwari. Vol. 9706. Lecture Notes in Computer Science. Springer, 2016, pp. 293–301.

[23] Yeting Ge and Leonardo de Moura. "Complete Instantiation for Quantified Formulas in Satisfiabiliby Modulo Theories". In: *Computer Aided Verification (CAV)*. Ed. by Ahmed Bouajjani and Oded Maler. Vol. 5643. Lecture Notes in Computer Science. Springer, 2009, pp. 306–320. DOI: 10.1007/978-3-642-02658-4_25. URL: http://dx.doi.org/10.1007/978-3-642-02658-4_25.

[24] Leon Henkin. "Completeness in the theory of types". In: *The Journal of Symbolic Logic* 15.2 (1950), pp. 81–91.

[25] G.P. Huet. "A unification algorithm for typed -calculus". In: *Theoretical Computer Science* 1.1 (1975), pp. 27 –57. ISSN: 0304-3975. DOI: http://dx.doi.org/10.1016/0304-3975(75)90011-0. URL: http://www.sciencedirect.com/science/article/pii/0304397575900110.

[26] Cezary Kaliszyk and Josef Urban. "Learning-assisted theorem proving with millions of lemmas". In: *J. Symb. Comput.* 69 (2015), pp. 109–128. DOI: 10.1016/j.jsc.2014.09.032. URL: https://doi.org/10.1016/j.jsc.2014.09.032.

[27] Manfred Kerber. "How to prove higher order theorems in first order logic". In: (1999).

[28] Tomer Libal and Dale Miller. "Functions-as-Constructors Higher-Order Unification". In: *Formal Structures for Computation and Deduction (FSCD)*. Ed. by Delia Kesner and Brigitte Pientka. Vol. 52. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016, 26:1–26:17.

[29] Dale Miller. "A Logic Programming Language with Lambda-Abstraction, Function Variables, and Simple Unification". In: *J. Log. Comput.* 1.4 (1991), pp. 497–536. DOI: 10.1093/logcom/1.4.497. URL: https://doi.org/10.1093/logcom/1.4.497.

[30] Dale Miller. "Unification of Simply Typed Lamda-Terms as Logic Programming". In: *International Conference on Logic Programming (ICLP)*. Ed. by Koichi Furukawa. MIT Press, 1991, pp. 255–269.

[31] Dale A. Miller. "A compact representation of proofs". In: *Studia Logica* 46.4 (1987), pp. 347–370. ISSN: 1572-8730. DOI: 10.1007/BF00370646. URL: https://doi.org/10.1007/BF00370646.

[32] Georg Moser and Richard Zach. "The Epsilon Calculus". In: *Computer Science Logic: 17th International Workshop CSL 2003, 12th Annual Conference of the EACSL, 8th Kurt Gödel Colloquium, KGC 2003, Vienna, Austria, August 25-30, 2003. Proceedings*. Ed. by Matthias Baaz and Johann A. Makowsky. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 455–455. ISBN: 978-3-540-45220-1. DOI: 10.1007/978-3-540-45220-1_36. URL: https://doi.org/10.1007/978-3-540-45220-1_36.

[33] Leonardo de Moura and Nikolaj Bjørner. "Efficient E-Matching for SMT Solvers". In: *Proc. Conference on Automated Deduction (CADE)*. Ed. by Frank Pfenning. Vol. 4603. Lecture Notes in Computer Science. Springer, 2007, pp. 183–198. ISBN: 978-3-540-73594-6. DOI: 10.1007/978-3-540-73595-3_13. URL: http://dx.doi.org/10.1007/978-3-540-73595-3_13.

[34] Charles Gregory Nelson. "Techniques for Program Verification". PhD thesis. Stanford, CA, USA, 1980.

[35] Greg Nelson and Derek C Oppen. "Fast decision procedures based on congruence closure". In: *Journal of the ACM (JACM)* 27.2 (1980), pp. 356–364.

[36] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. "Solving SAT and SAT Modulo Theories: From an Abstract Davis–Putnam–Logemann–Loveland Procedure to DPLL(T)". In: *J. ACM* 53.6 (Nov. 2006), pp. 937–977. ISSN: 0004-5411. DOI: 10.1145/1217856.1217859. URL: http://doi.acm.org/10.1145/1217856.1217859.

[37] Robert Nieuwenhuis and Albert Rubio. "Paramodulation-Based Theorem Proving". In: *Handbook of automated reasoning*. Ed. by Alan Robinson and Andrei Voronkov. Vol. 1. 2001, pp. 371–443.

[38] Tobias Nipkow. "Functional Unification of Higher-Order Patterns". In: *Logic In Computer Science (LICS)*. IEEE Computer Society, 1993, pp. 64–74.

[39] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Vol. 2283. LNCS. Springer, 2002.

[40] Andrew Reynolds, Cesare Tinelli, and Leonardo Mendonça de Moura. "Finding conflicting instances of quantified formulas in SMT". In: *Formal Methods In Computer-Aided Design (FMCAD)*. IEEE, 2014, pp. 195–202. URL: http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=6975680.

[41] Andrew Reynolds et al. "Counterexample-Guided Quantifier Instantiation for Synthesis in SMT". In: *Computer Aided Verification (CAV)*. Ed. by Daniel Kroening and Corina S. Pasareanu. Vol. 9207. Lecture Notes in Computer Science. Springer, 2015, pp. 198–216.

[42] Dan Rosén and Nicholas Smallbone. "TIP: Tools for Inductive Provers". In: *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*. Ed. by Martin Davis et al. Springer, 2015, pp. 219–232.

[43] Robert E Shostak. "A practical decision procedure for arithmetic with function symbols". In: *Journal of the ACM (JACM)* 26.2 (1979), pp. 351–360.

[44] G. Sutcliffe. "The CADE ATP System Competition - CASC". In: *AI Magazine* 37.2 (2016), pp. 99–101.

[45] G. Sutcliffe. "The TPTP Problem Library and Associated Infrastructure: The FOF and CNF Parts, v3.5.0". In: *Journal of Automated Reasoning* 43.4 (2009), pp. 337–362.