Language and Proofs for Higher-Order SMT (Work in Progress)∗

Haniel Barbosa  Jasmin Christian Blanchette  Simon Cruanes

Daniel El Ouraoui  Pascal Fontaine

University of Lorraine, CNRS, Inria, and LORIA, Nancy, France
Vrije Universiteit Amsterdam, Amsterdam, The Netherlands
Max-Planck-Institut für Informatik, Saarbrücken, Germany

{haniel.barbosa,jasmin.blanchette,simon.cruanes,daniel.el-ouraoui,pascal.fontaine}@inria.fr

Satisfiability modulo theories (SMT) solvers have throughout the years been able to cope with increasingly expressive formulas, from ground logics to full first-order logic modulo theories. Nevertheless, higher-order logic within SMT is still little explored. One main goal of the Matryoshka project, which started in March 2017, is to extend the reasoning capabilities of SMT solvers and other automatic provers beyond first-order logic. In this preliminary report, we discuss an extension of the SMT-LIB language, the standard input format of SMT solvers, to handle higher-order constructs. We also discuss how to augment the proof format of the SMT solver veriT to accommodate these new constructs and the solving techniques they require.

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 [8]. Today, the leading automatic provers for first-order classical logic are based either on the superposition calculus [2][12] or on CDCL(∧) [11]. Those based on the latter are usually called satisfiability modulo theory (SMT) solvers and are the focus of this paper.

Our goal, as part of the Matryoshka project[1] is to extend SMT solvers to natively handle higher-order problems, thus avoiding completeness and performance issues with clumsy encodings. In this paper, 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

∗This PxTP submission should be considered an addendum of the PxTP presentation-only submission “Scalable Fine-Grained Proofs for Formula Processing.” Although we tried to make the current paper self-contained, a knowledge of this previous work is useful to get the whole picture.

http://matryoshka.gforge.inria.fr/
solvers support SMT-LIB [5] 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 (Section 2). Regrettably, there is no standard yet for proof output; each proof-producing solver has its own proof format. We focus on the proof format of the SMT solver veriT [9]. This solver is known for its very detailed proofs [3, 6], which are reconstructed in the proof assistants Coq [1] and Isabelle/HOL [7] and in the GAPT system [10].

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. [3]. 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 3).

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. [6]. 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 [4]. Since only modifications to the instantiation module are planned, the only rules that must be adapted are those concerned with quantifier instantiation:

$$
\forall x. \varphi[x] \rightarrow \varphi[t] \quad \text{INST}_\forall
$$
$$
\varphi[t] \rightarrow \exists x. \varphi[x] \quad \text{INST}_\exists
$$

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.

## 2 A Syntax Extension for the SMT-LIB Language

By the time of this writing, the SMT-LIB standard is at version 2.5 [5], 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) [13], 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:

$$
\langle \text{sort} \rangle ::= \langle \text{identifier} \rangle \mid (\langle \text{identifier} \rangle \langle \text{sort} \rangle^+) \mid (\Rightarrow \langle \text{sort} \rangle^+ \langle \text{sort} \rangle)
$$

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:
The old rule \((\text{qual}\text{.identifier}\text{.term})^+\) 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 \((\text{declare-fun } f \ (\text{Int}) \ (\rightarrow \text{Int} \ \text{Int}))\). The following example illustrates higher-order functions, terms representing a function, and partial applications:

\[
\begin{align*}
\text{(set-logic UFLIA)} \\
(\text{declare-fun } g \ (\text{Int}) \ (\rightarrow \text{Int} \ \text{Int})) \\
(\text{declare-fun } h \ (\text{Int} \ \text{Int}) \ \text{Int}) \\
(\text{declare-fun } f \ (\rightarrow \text{Int} \ \text{Int})) \ \text{Int}) \\
(\text{assert} \ (\! (f \ (h \ 1)) \ ((g \ 1) \ 2))) \\
\text{(exit)}
\end{align*}
\]

The term \((g \ 1)\) is a function from \text{Int} to \text{Int}, in agreement with the sort of \(g\). Then it is applied to \(2\) in the expression \(((g \ 1) \ 2)\) of sort \text{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 \text{Int}. Note that in our presentation all functions of type \((\rightarrow \text{Int} \ \text{Int} \ .. \ \text{Int})\) are equivalent to \((\rightarrow \text{Int} \ (\rightarrow \text{Int} \ .. \ \text{Int}))\). This implies, in particular, that in the example above writing \(((g \ 1) \ 2)\) is semantically equivalent to \((g \ 1 \ 2)\). More precisely we may considerate the three different declaration of \(f\) below:

\[
\begin{align*}
(\text{declare-fun } f \ () \ (\rightarrow \text{Int} \ (\rightarrow \text{Int} \ \text{Int}))) \\
(\text{declare-fun } f \ (\text{Int}) \ (\rightarrow \text{Int} \ \text{Int})) \\
(\text{declare-fun } f \ (\text{Int} \ \text{Int}) \ \text{Int})
\end{align*}
\]

as the unique form \((\text{declare-fun } f \ () \ (\rightarrow \text{Int} \ \text{Int} \ \text{Int})))\). This follows from \(\rightarrow\) being right associative. The next example features a \(\lambda\)-abstraction:

\[
\begin{align*}
\text{(set-logic UFLIA)} \\
(\text{declare-fun } g \ (\text{Int}) \ (\text{Int})) \\
(\text{assert} \ (\! ((\lambda ((f \ (\rightarrow \text{Int} \ \text{Int})) \ (\text{x} \ \text{Int})) \ f \ \text{x}) \ g \ 1) \ (g \ 1))))) \\
\text{(exit)}
\end{align*}
\]

The term \((\lambda ((f \ (\rightarrow \text{Int} \ \text{Int})) \ (\text{x} \ \text{Int})) \ f \ \text{x})\) 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 \(\beta\)-reduction).
3 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:

\[
M ::= x \mid c \mid MM \mid \lambda x.M \mid \text{let } \bar{x}_n \simeq \bar{M}_n \text{ in } M
\]

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

\[
\models (\lambda x.t[x]) s \simeq t[s] \quad (\beta)
\]

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.:

\[
\Gamma ::= \emptyset \mid \Gamma, x \mid \Gamma, \bar{x}_n \mapsto \bar{t}_n
\]

Our new set of rules is similar to Barbosa et al. The rules TRANS, SKO\(\exists\), SKO\(\forall\), LET, and TAUT\(\forall\) apply unchanged. The BIND rule is modified to accommodate the new \( \lambda \)-binder:

\[
\frac{\Gamma, y, x \mapsto y \triangleright \varphi \simeq \psi}{\Gamma \triangleright (Bx. \varphi) \simeq (By. \psi)} \quad \text{BIND} \quad \text{if } y \not\in FV(Bx. \varphi)
\]

The metavariable \( B \) ranges over \( \forall \), \( \exists \), and \( \lambda \). 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:

\[
\frac{\Gamma \triangleright t \simeq s \quad \Gamma \triangleright u \simeq v}{\Gamma \triangleright t u \simeq s v} \quad \text{CONG}
\]

The only genuinely new rule handles \( \beta \)-reduction—that is, the substitution of an argument in the body of a \( \lambda \)-abstraction. It is similar in form to the LET rule from the first-order calculus [3]:

\[
\frac{\Gamma \triangleright v \simeq s \quad \Gamma, x \mapsto s \triangleright t \simeq u}{\Gamma \triangleright (\lambda x.t) v \simeq u} \quad \text{BETA} \quad \text{if } \Gamma(s) = s
\]

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

**Example 1.** The derivation tree of the normalization of \((\lambda x. p x x) a\) is as follows:

\[
\begin{array}{l}
\Gamma \triangleright a \simeq a \\
\Gamma \triangleright a \triangleright p \simeq p \\
\Gamma \triangleright a \triangleright x \simeq a \\
\Gamma \triangleright (\lambda x.t) p x x \simeq p a a \\
\Gamma \triangleright (\lambda x. p x x) a \simeq p a a
\end{array}
\]
Example 2. The following tree features a $\beta$-redex under a $\lambda$-abstraction. Let $\Gamma_1 = w, x \mapsto w$; $\Gamma_2 = \Gamma_1, y \mapsto f w$; and $\Gamma_3 = \Gamma_2, z \mapsto f w$:

$$
\frac{
\Gamma_1 \triangleright f \simeq f \\
\text{TAUT}
}{
\Gamma_1 \triangleright f x \simeq f w \\
\text{REFL}
}
\quad
\frac{
\Gamma_2 \triangleright y \simeq f w \\
\text{CONG}
}{
\Gamma_2 \triangleright (\lambda z. p z) y \simeq p (f w) \\
\text{REFL}
}
\quad
\frac{
\Gamma_3 \triangleright p z \simeq p (f w) \\
\text{BETA}
}{
\Gamma_1 \triangleright (\lambda y. (\lambda z. p z) y) (f x) \simeq p (f w) \\
\text{BIND}
}
$$

Example 3. The transitivity rule is useful when the applied term reduces to a $\lambda$-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 $\Gamma_6 = \Gamma_4, z \mapsto \lambda w_1. w$:

$$
\frac{
\Gamma_1 \triangleright y \simeq w \\
\text{REFL}
}{
\Gamma_1 \triangleright (\lambda x. p x) y \simeq p w \\
\text{BETA}
}
\quad
\frac{
\Gamma_2 \triangleright p x \simeq p w \\
\text{REFL}
}{
\Gamma_1 \triangleright (\lambda z. w_1) (p w) \simeq w \\
\text{BETA}
}
\quad
\frac{
\Gamma_4 \triangleright x \simeq (\lambda w_1. w) \\
\text{REFL}
}{
\Gamma_4 \triangleright (\lambda x. z) x \simeq (\lambda w_1. w) \\
\text{BETA}
}
\quad
\frac{
\Gamma_6 \triangleright z \simeq (\lambda w_1. w) \\
\text{REFL}
}{
\Gamma_1 \triangleright (\lambda y. (\lambda z. z) x) (\lambda y. y) \simeq (\lambda w_1. w) \\
\text{BIND}
}
$$

where $\Pi$ stands for the subtree

$$
\frac{
\Gamma_3 \triangleright y \simeq w \\
\text{REFL}
}{
\Gamma_1 \triangleright (\lambda y. (\lambda z. z) x) (\lambda y. y) \simeq (\lambda w_1. w) \\
\text{BETA}
}
$$

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 $\lambda$-calculus, based on the following grammar:

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

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

$$
\frac{
M[u] \simeq N[v] \\
\text{CONG}
}{
M[t u] \simeq N[s v]
}
\quad
\frac{
M[\bar{t}] \simeq N[s] \\
\text{BETA}
}{
M[\bar{t} u] \simeq N[s v]
}
\quad
\frac{
M[\lambda y. (\lambda x. \bar{t}) x] \simeq N[u] \\
\text{BETA}
}{
M[\lambda y. (\lambda x. \bar{t}) x] \simeq N[u]
}
\quad
\frac{
M[\lambda y. (\lambda x. \bar{t}) x] \simeq N[u] \\
\text{BETA}
}{
M[\lambda y. (\lambda x. \bar{t}) x] \simeq N[u]
}
$$

Lemma 1. If the judgment $M \simeq N$ is derivable using the encoded inference system with the theories $\mathcal{T}_1 \ldots \mathcal{T}_n$, then $\models_{\mathcal{S}} \text{reify}(M \simeq N)$ with $\mathcal{S} = \mathcal{T}_1 \cup \cdots \cup \mathcal{T}_n \cup \alpha \cup \beta$.

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

CASE $\text{BIND } B = \lambda$: The induction hypothesis is $\models_{\mathcal{S}} \text{reify}(M[\lambda y. (\lambda x. \bar{t}) x] \simeq N[\lambda y. \psi[y]])$. Using ($\beta$) and the side condition of the rule, we can also deduce that $\models_{\mathcal{S}} \text{reify}(M[\lambda y. \psi[y]]) \simeq N[\lambda y. \psi[y]]$. Hence by $\alpha$-conversion this is equivalent to $\models_{\mathcal{S}} \text{reify}(M[\lambda x. \psi[x]] \simeq N[\lambda y. \psi[y]])$. 


CASE CONG: This case follows directly from equality in a higher-order setting.

CASE BETA: This case follows directly from $(\beta)$ 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 $\Gamma \vdash t \simeq u$. They are defined over the structure of the context, as follows:

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

**Lemma 2.** If the judgment $\Gamma \vdash t \simeq u$ 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 $\Gamma \vdash t \simeq u$, 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 3 (Soundness of Inferences).** If the judgment $\Gamma \vdash t \simeq u$ is derivable using the original inference system with the theories $\mathcal{T}_1 \ldots \mathcal{T}_n$, then $\models \mathcal{F}(\Gamma) \simeq u$ with $\mathcal{F} = \mathcal{T}_1 \cup \ldots \cup \mathcal{T}_n \cup \simeq \cup \varepsilon \cup \text{let} \cup \beta$.

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

## 4 Conclusion and Future Work

We have presented a preliminary extension of the SMT-LIB syntax and of the veriT proof format to support higher-order constructs in SMT problems and proofs. Partial applications, $\lambda$-abstractions, and quantification over functional variables can now be understood by a solver compliant with these languages. 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. [3]. 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.

With the foundations laid down, the next step will be to implement the automatic reasoning machinery for higher-order formulas and properly evaluating its effectiveness. Moreover, when providing support for techniques involving, for example, inductive datatypes, we will need to augment the proof format accordingly.
References


