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 report on 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, Miz$\mathbb{AR}$, 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 [1,12] or on CDCL($\mathscr{T}$) [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

[*]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.

[†]This work has been partially supported by the ANR/DFG project STU 483/2-1 SMArT ANR-13-IS02-0001 of the Agence Nationale de la Recherche, by the H2020-FETOPEN-2016-2017-CSA project SC^2 (712689), and by the European Research Council (ERC) starting grant Matryoshka (713999).

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

Submitted to:
PxTP 2017

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 solvers support SMT-LIB [5] as an input format. We report on 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 [2,6], which are reconstructed in the proof assistants 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. [2]. 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($\mathscr{T}$) 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($\mathscr{T}$) 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:

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

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.

2 A Syntax Extension for the SMT-LIB Language

By the time of starting this writing, the SMT-LIB standard was at version 2.5 [5], and version 2.6 was in preparation. Although some discussions to extend the SMT-LIB 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 report on an extension of 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. This 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. The language is first extended so that functional sorts can be built:

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

The second line is the addition to the original grammar. We use $\texttt{(->}\,\langle sort \rangle^{+}\,\langle sort \rangle\,\texttt{)}$ rather than a special case of $\texttt{(}\,\langle identifier \rangle^{+}\,\langle sort \rangle\,\texttt{)}$ to avoid ambiguities with parametric sorts and to have the same notation as the one generally used for functional sorts.

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:

```
⟨term⟩         ::=  ⟨spec_constant⟩
                 |  ⟨qual_identifier⟩
                 |  ( ⟨term⟩ ⟨term⟩+ )
                 |  ( lambda ( ⟨sorted_var⟩+ ) ⟨term⟩+ )
                 |  ( let ( ⟨var_binding⟩+ ) ⟨term⟩ )
                 |  ( forall ( ⟨sorted_var⟩+ ) ⟨term⟩ )
                 |  ( exists ( ⟨sorted_var⟩+ ) ⟨term⟩ )
                 |  ( match ⟨term⟩ ( ⟨match_case⟩+ ) )
                 |  ( ! ⟨term⟩ ⟨attribute⟩+ )
⟨sorted_var⟩   ::=  ( ⟨symbol⟩ ⟨sort⟩ )
```

The old rule (⟨*qual_identifier*⟩⟨*term*⟩$^+$) is now redundant. Higher-order quantification requires no new syntax, since sorts have been extended to accommodate functions.

Semantically, the arrow constructor `->` and the lambda abstraction use the following typing rules:

$$\frac{\Sigma[x:\sigma] \vdash t : \tau}{\Sigma \vdash \lambda x.t : \sigma \to \tau}\ \text{LAMBDA} \qquad \frac{\Sigma \vdash u : \sigma \to \tau \quad \Sigma \vdash v : \sigma}{\Sigma \vdash u\,v : \tau}\ \text{APP}$$

Where a judgment is composed of two items. On the left hand side, a signature Σ, which is a tuple of function and constant symbols. On the right hand side, a term annotated by its type. The notation $\Sigma[x:\tau]$ stands for the signature that maps x to the type τ.

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`. Note that in our presentation all functions of type `(-> Int Int ... Int)` are equivalent to `(-> Int (-> Int (-> ... Int)))`. This implies, in particular, that in the example above `((g 1) 2)` is semantically equivalent to `(g 1 2)`. More precisely we may considerate the three different declaration of f below:

```
(declare-fun f () (-> Int (-> Int Int)))
(declare-fun f (Int) (-> Int Int))
(declare-fun f (Int Int) Int)
```

as the unique form `(declare-fun f () (-> Int Int Int))`. This follows from `->` being right associative. The next example features 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).

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:

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

where formulas are terms of Boolean type. We rely on the meta theory defined in Barbosa et al. [2]. Besides the axioms for characterizing Hilbert choice and 'let' described there, we add the following axiom for λ-abstraction, where $\simeq$ denotes the equality predicate:

$$\models (\lambda x.\, t[x])\; s \;\simeq\; t[s] \qquad (\beta)$$

The notation $t[\bar{x}_n]$ stands for a term that may depend on distinct variables $\bar{x}_n$; $t[\bar{s}_n]$ is the corresponding term where the terms $\bar{s}_n$ are simultaneously substituted for $\bar{x}_n$; bound variables in t are renamed to avoid capture. 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* $\{\bar{x}_n \mapsto \bar{t}_n\}$. Abstractly, a context Γ fixes a set of variables and specifies a substitution $\mathit{subst}(\Gamma)$. The substitution is the identity for $\varnothing$ and is defined as follows in the other cases:

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

In the first equation, the $[x \mapsto x]$ update shadows any replacement of x induced by Γ. We write $\Gamma(t)$ to abbreviate the capture-avoiding substitution $\mathit{subst}(\Gamma)(t)$.

Our new set of rules is similar to that in Barbosa et al. The rules Trans, Sko$_\exists$, Sko$_\forall$, Let, and Taut$_{\mathcal{T}}$ are unchanged. The Bind rule is modified to accommodate the new λ-binder:

$$\frac{\Gamma, y, x \mapsto y \rhd s \simeq t}{\Gamma \rhd (\mathrm{B}x.\, s) \simeq (\mathrm{B}y.\, t)}\;\textsc{Bind} \quad \text{if } y \notin FV(\mathrm{B}x.\, s)$$

The metavariable B ranges over $\forall$, $\exists$, and λ. The Cong rule is also 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, since it can involve more complicated terms. Rewriting can now occur also on these complicated terms. The updated Cong rule is as follows:

$$\dfrac{\Gamma \rhd t \simeq s \qquad \Gamma \rhd u \simeq v}{\Gamma \rhd t\,u \simeq s\,v}\;\textsc{Cong}$$

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:

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

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

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

$$\dfrac{\dfrac{}{\rhd \mathsf{a} \simeq \mathsf{a}}\;\textsc{Cong} \qquad \dfrac{\dfrac{\dfrac{}{x \mapsto \mathsf{a} \rhd \mathsf{p} \simeq \mathsf{p}}\;\textsc{Refl} \quad \dfrac{}{x \mapsto \mathsf{a} \rhd x \simeq \mathsf{a}}\;\textsc{Refl}}{x \mapsto \mathsf{a} \rhd \mathsf{p}x \simeq \mathsf{p}\,\mathsf{a}}\;\textsc{Cong} \quad \dfrac{}{x \mapsto \mathsf{a} \rhd x \simeq \mathsf{a}}\;\textsc{Refl}}{x \mapsto \mathsf{a} \rhd \mathsf{p}xx \simeq \mathsf{p}\,\mathsf{a}\,\mathsf{a}}\;\textsc{Cong}}{\rhd (\lambda x.\, \mathsf{p}xx)\mathsf{a} \simeq \mathsf{p}\,\mathsf{a}\,\mathsf{a}}\;\textsc{Beta}$$

Example 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 $\Gamma_3 = \Gamma_2, z \mapsto f\,w$:

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

Example 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 $\Gamma_6 = \Gamma_4, z \mapsto \lambda w_1.\, w$:

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

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 soundness of the extended calculus is a simple extension of the soundness proof in the longer version of Barbosa et al. [3]. 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:

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

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

Lemma 1. *If the judgment $M \simeq N$ is derivable using the encoded inference system with the theories $\mathscr{T}_1 \ldots \mathscr{T}_n$, then $\models_{\mathscr{T}} \mathit{reify}(M \simeq N)$ with $\mathscr{T} = \mathscr{T}_1 \cup \cdots \cup \mathscr{T}_n \cup \simeq \cup\, \varepsilon_1 \cup \varepsilon_2 \cup \mathrm{let} \cup \beta$.*

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 $\models_{\mathscr{T}} \mathit{reify}(M[\lambda y.\,(\lambda x.\,s[x])\;y] \simeq N[\lambda y.\,t[y]])$. Using (β) and the side condition of the rule, we can also deduce that $\models_{\mathscr{T}} \mathit{reify}(M[\lambda y.\,s[y]])] \simeq N[\lambda y.\,t[y]])$. Hence by α-conversion this is equivalent to $\models_{\mathscr{T}} \mathit{reify}(M[\lambda x.\,s[x]] \simeq N[\lambda y.\,t[y]])$.
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 2. *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 $M[\lambda y.\,(\lambda x.\;s)\;y] \simeq N[\lambda y.\;t]$ (i.e., $L(\Gamma, y, x \mapsto y)[s] \simeq R(\Gamma, y, x \mapsto y)[t]$), and the encoded succedent is $M[\lambda x.\;s] \simeq N[\lambda y.\;t]$. 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 $\Gamma \rhd t \simeq u$ is derivable using the original inference system with the theories $\mathscr{T}_1 \ldots \mathscr{T}_n$, then $\models_{\mathscr{T}} \Gamma(t) \simeq u$ with $\mathscr{T} = \mathscr{T}_1 \cup \ldots \cup \mathscr{T}_n \cup \simeq \cup\, \varepsilon_1 \cup \varepsilon_2 \cup \mathrm{let} \cup \beta$.*

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, λ-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. [2]. 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.

Acknowledgment We would like to thank the anonymous reviewers for their comments. Between the initial version of this document and the current one, the SMT-LIB extension has been greatly influenced by discussions with Clark Barrett and Cesare Tinelli (the SMT-LIB managers, together with Pascal Fontaine) and they should also be considered authors of this syntax extension.

References

[1] Leo Bachmair & Harald Ganzinger (1994): *Rewrite-Based Equational Theorem Proving with Selection and Simplification*. *Journal of Logic and Computation* 4(3), pp. 217–247.

[2] Haniel Barbosa, Jasmin Christian Blanchette & Pascal Fontaine (2017): *Scalable Fine-Grained Proofs for Formula Processing*. In Leonardo de Moura, editor: *Conference on Automated Deduction (CADE)*, *LNCS* 10395, Springer, pp. 398–412.

[3] Haniel Barbosa, Jasmin Christian Blanchette & Pascal Fontaine (2017): *Scalable Fine-Grained Proofs for Formula Processing*. Research Report, Inria. Available at `https://hal.inria.fr/hal-01526841`.

[4] Haniel Barbosa, Pascal Fontaine & Andrew Reynolds (2017): *Congruence Closure with Free Variables*. In Axel Legay & Tiziana Margaria, editors: *Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, *LNCS* 10206, pp. 214–230.

[5] Clark Barrett, Pascal Fontaine & Cesare Tinelli (2015): *The SMT-LIB Standard: Version 2.5*. Technical Report, Department of Computer Science, The University of Iowa. Available at `www.SMT-LIB.org`.

[6] Frédéric Besson, Pascal Fontaine & Laurent Théry (2011): *A Flexible Proof Format for SMT: a Proposal*. In Pascal Fontaine & Aaron Stump, editors: *Workshop on Proof eXchange for Theorem Proving (PxTP)*.

[7] Jasmin Christian Blanchette, Sascha Böhme, Mathias Fleury, Steffen Juilf Smolka & Albert Steckermeier (2016): *Semi-intelligible Isar Proofs from Machine-Generated Proofs*. *Journal of Automated Reasoning* 56(2), pp. 155–200.

[8] Jasmin Christian Blanchette, Cezary Kaliszyk, Lawrence C. Paulson & Josef Urban (2016): *Hammering towards QED*. *Journal of Formalized Reasoning* 9(1), pp. 101–148.

[9] Thomas Bouton, Diego Caminha B. de Oliveira, David Déharbe & Pascal Fontaine (2009): *veriT: An Open, Trustable and Efficient SMT-Solver*. In Renate A. Schmidt, editor: *Conference on Automated Deduction (CADE)*, *LNCS* 5663, Springer, pp. 151–156.

[10] Gabriel Ebner, Stefan Hetzl, Giselle Reis, Martin Riener, Simon Wolfsteiner & Sebastian Zivota (2016): *System Description: GAPT 2.0*. In Nicola Olivetti & Ashish Tiwari, editors: *International Joint Conference on Automated Reasoning (IJCAR)*, *LNCS* 9706, Springer, pp. 293–301.

[11] Robert Nieuwenhuis, Albert Oliveras & Cesare Tinelli (2006): *Solving SAT and SAT Modulo Theories: From an Abstract Davis–Putnam–Logemann–Loveland Procedure to DPLL(T)*. *Journal of the ACM* 53(6), pp. 937–977.

[12] Robert Nieuwenhuis & Albert Rubio (2001): *Paramodulation-Based Theorem Proving*. In Alan Robinson & Andrei Voronkov, editors: *Handbook of Automated Reasoning*, I, pp. 371–443.

[13] Dan Rosén & Nicholas Smallbone (2015): *TIP: Tools for Inductive Provers*. In Martin Davis, Ansgar Fehnker, Annabelle McIver & Andrei Voronkov, editors: *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, Springer, pp. 219–232.