Extending Smt-Lib v2 with λ-Terms and Polymorphism

Richard Bonichon, David Déharbe, and Cláudia Tavares

Universidade Federal do Rio Grande do Norte
Natal, Brazil
`richard@dimap.ufrn.br`, `david@dimap.ufrn.br`, `claudia@ppgsc.ufrn.br`

Abstract

This paper describes two syntactic extensions to Smt-Lib scripts: lambda-expressions and polymorphism. After extending the syntax to allow these expressions, we show how to update the typing rules of the Smt-Lib to check the validity of these new terms and commands. Since most Smt-solvers only deal with many-sorted first-order formulas, we detail a monomorphization mechanism to allow to use polymorphism in Smt-Lib syntax while retaining a monomorphic solver core.

1 Introduction

Dissemination of Smt-solvers requires they are powerful, trustable and open (i.e. easy to interface). The Smt-Lib format [BST10] is an initiative from the Smt community to address the last aspect, by offering both a common language to describe problems and a command language to interact with the solver.

The Smt-Lib format has evolved in the recent years from version 1.2 to version 2.0 to simplify the definition of the syntax for the part of the language that is responsible for problem descriptions (i.e. declaration of the signature and assertion of logic expressions) on the one hand, and to enrich the commands that allow third-party tools to interact with complying SMT solvers.

veriT is an open-source solver jointly developed at INRIA and UFRN. Early versions of the veriT solver implemented several extensions to version 1.2 of the Smt-Lib format: macro-definitions, λ-expressions and β-reduction. A later extension included polymorphic sorts, signatures and assertions. A successful application of these extensions has been to apply SMT solving to verify proof obligations stemming from set-based formalisms (namely, B and Event-B) using standard Smt-Lib logics (AUFLIA) [Dé10, DFGV12, Dé13].

The rationale of this application is essentially the following:

- A set is encoded as its characteristic predicate. For instance, $\{x \cdot 0 \leq x \leq 9\}$ is encoded as:

  (**lambda** ((x Int)) (**and** (<= 0 x) (<= x 9))).

- Set operations are encoded as higher-order functions. For instance, set intersection is encoded as a (polymorphic) macro called inter defined as:

  (**define−fun** (**par** (X) (inter (**lambda** ((f (X Bool)) (g (X Bool)) (x X)) (**and** (f x) (g x))))) ,

  and set membership by the macro named member and defined as:

  (**define−fun** (**par** (X) (member (**lambda** ((x X) (f (X Bool))) (f x))))) ,

  in both definitions, X denotes a type variable, its scope being the definition itself.

To handle such expressions, veriT implements a processor that performs macro-expansion and β-reduction steps as well as type inference. For instance, the formula $0 \in (\{x{\cdot}x \geq 0\} \cap \{x{\cdot}x \leq 0\})$ would be encoded as `(member 0 (union (lambda (x Int) (>= x 0)) (lambda (x Int) (<= x 0))))`, and the result of processing this expression is `(and (>= 0 0) (<= 0 0))`. In addition, this processor rewrites equalities between lambda expressions as universal quantifications. Once all steps have been applied, if the goal is not first-order logic, then veriT emits an error message and halts.

Some of these extensions were included in the SMT-LIB format: polymorphic sorts , though restricted to theory files, and macro-definition (named function definitions). In this paper, we thus discuss modifications to the SMT-LIB format 2.0 corresponding to the extensions to SMT-LIB format 1.2 that were implemented in the solver veriT. It is noteworthy that these modifications maintain backward compatibility with the existing definition of the SMT-LIB format. Also we discuss how to rewrite a problem expressed with the proposed extensions to plain SMT-LIB 2.0.

This paper is organized as follows. Sec. 2 presents the extensions made to SMT-LIB, introducing polymorphism at the SMT-LIB script level. This leads to an updated set of typing rules, mainly for SMT-LIB terms, which is discussed in Sec. 3. Using these rules, we detail a strategy to generate a monomorphic version of our target problem in Sec. 4. Finally, we discuss the pros and cons of our solution in the context of related work in Sec. 5 and detail ongoing and further work in Sec. 6.

2 Extensions to SMT-LIB

We propose two extensions to the SMT-LIB format:

- anonymous functions (i.e., λ-abstractions) and their applications;
- parametric polymorphism for assertions and function types.

The SMT-LIB already features two flavors of polymorphism:

- parametric polymorphism for sorts and function signatures, but only for background theories;
- ad-hoc polymorphism because functions can be overloaded.

The extension adds parametric polymorphism to assertions and function definitions and declarations. The additional introduction of λ gives a very functional, higher-order, flavor to this extended SMT-LIB. The typing rules presented in Sec. 3.1 even allow let-polymorphism inside the SMT-LIB.

However, λ-abstractions as we envision them will not add much expressiveness as we want to be able to get a first-order problem only through the application of β-reduction. Thus, we will not handle a reduced problem with residual higher-order or partial applications. We see the addition of λ-abstractions as a convenient mechanism to encode certain problems.

Also, the introduction of polymorphism at the syntactic level does not fundamentally change the expressive power available for SMT-LIB scripts. Indeed, the combination of type schemes to express background theories and overloading of functions permitted by the SMT-LIB standard already covers most functionalities which ML-style let-polymorphism permits. Nonetheless, we argue that polymorphism in scripts is syntactically more convenient than writing every ground instances of the expressions we are interested in.

The next section presents the concrete syntax for the two extensions mentioned above.

2.1 Syntax extensions for Smt-Lib

Anonymous functions are terms introduced using **lambda**, which becomes a keyword. Polymorphic terms reuse the same **par** keyword as the polymorphic elements of the Smt-Lib theories. The syntactic extensions are summarized in the BNF extract of Fig. 1.

```
⟨par_function_args⟩  ::=  par  (⟨symbol⟩+)

⟨par_command⟩  ::=  (define−fun  (⟨par_function_args⟩  (⟨symbol⟩
                          (⟨sorted_var⟩*)  ⟨sort⟩  ⟨term⟩)))
                 |  (declare−fun  (⟨par_function_args⟩  (⟨symbol⟩  (⟨sort⟩*)  ⟨sort⟩)))
                 |  (assert  (⟨par_function_args⟩  ⟨term⟩))
```

Figure 1: BNF extensions for Smt-Lib

Function types are allowed as the return type of functions, due to the inclusion of λ-term. The choice made is to declare a function type as a list of types. For example the function declaration (**declare−fun** f (Int Int) Int) is not the same as the function declaration (**declare−fun** f ((Int Int)) Int). The former is a function which expects two integers and returns an integer, the latter expects only one argument — a function from integer to integer — and returns an integer. Now, the only possible problem is to distinguish in a sort expression (X Y) between a sort meant to express a function type and a sort which is the application of a sort of arity ≥ 1 to its argument, like (Array Int). This is not a practical problem as sort arity is explicitly stated. Therefore, if the arity of X is greater than 1, we say it is a sort application, otherwise it is a function type.

3 Typing extended Smt-Lib

This section details the rules for typing extended Smt-Lib scripts. These rules extend the current set of rules for Smt-Lib[1].

Typing polymorphic terms is a necessary step towards uncovering monomorphic ground instances of polymorphic terms. Various type instantiations of the same polymorphic functions will lead in Sec. 4 to the generation of multiple monomorphic versions of the same function. Fortunately, ad-hoc polymorphism through overloading is permitted by the Smt-Lib standard.

3.1 Typing terms

The type of polymorphism we introduce in the extended Smt-Lib is ML-style prenex polymorphism [Pie02]. The typing rules of the system are described in Fig. 3. They can be seen as an adaptation of a Damas-Hindley-Milner [Mil78, Hin69] type system to the Smt-Lib syntax. These rules manipulates Smt-Lib sorts, type variables, tuple types and function types:

Definition 1 (Types)**.** *Let $\mathcal{S}$ be a set of sorts, $\mathcal{V}$ be a set of (type) variables. The set of well-formed types $\mathbb{T}$ is defined inductively as follows:*

1. *if $s \in \mathcal{S}$, then $s \in \mathbb{T}$*

[1]Detailed in Section 4.2.2 of the Smt-Lib Standard [BST10]

```
(set-logic AUFLIA)

(define-fun (par (X) (empty ((e X)) Bool false )))

(define-fun (par (Y)
    (insert ((e Y) (s (Y Bool))) (Y Bool)
        (lambda ((e1 Y)) (or (= e e1) (s e1))))))

(declare-fun (par (Z) (flat (((Z Bool) Bool)) (Z Bool))))

(assert (= (flat empty) empty))

(assert (par (X)
        (forall ((ss ((X Bool) Bool)))
        (= (flat (insert empty ss)) (flat ss)))))

(assert (par (X)
        (forall ((e X) (s (X Bool)) (ss ((X Bool) Bool)))
                (= (flat (insert (insert e s) ss))
                   (insert e (flat (insert s ss)))))))

(assert (forall ((s (Int Bool))) (= (flat (insert s empty)) s)))

(check-sat)
```

Figure 2: A polymorphic specification

2. *if* $v \in \mathcal{V}$, *then* $v \in \mathbb{T}$
3. *if* $T_1 \in \mathbb{T}, T_2 \in \mathbb{T}$, $T_1 \times T_2 \in \mathbb{T}$
4. *if* $T_1 \in \mathbb{T}, T_2 \in \mathbb{T}$, $T_1 \to T_2 \in \mathbb{T}$

Notations. Type variables will be denoted by α. $\bar{\alpha}$ represent a set of type variables (possibly empty). A type is said to be *ground* if it has no type variables. A *type substitution* is a mapping from type variables to types (ground or not). The application of a type substitution is denoted using := and extended to variable sets. Hence $T[\bar{\alpha} := \bar{T}']$ substitutes in T every member of $\bar{\alpha}$ by a corresponding type in $\bar{T}$. A *typing environment* Γ is a mapping from identifiers to types. Universal quantification over type variables is denoted $\forall_{\mathbb{T}}$, in order to separate this quantification from the universal quantifier ranging over term variable. For terms, x or x_i will stand for variables, t or t_i will stand for any term.

Typing rules. The rules of Fig. 3 use two additional functions:

- a function to compute the set of free type variables of a type, denoted fv_T;
- a generalization function Gen, which helps compute the most general type possible for a **let**-bound variable. This function is defined as follows:

$$\mathrm{Gen}(T, \Gamma) = \forall_{\mathbb{T}}(\mathrm{fv}_T(T) \backslash \mathrm{fv}_T(\Gamma)).T$$

The rules distinguish between curried functions ($\rightarrow$ type), where partial application is allowed, and uncurried functions, where the arguments are treated as a tuple, thus disallowing partial application. Curried functions come from explicit λ-abstractions whereas the SMT-LIB's **define−fun** define functions which can only be totally applied.

$$\frac{\Gamma(x) = \forall_{\mathbb{T}}\bar{\alpha}.T}{\Gamma \vdash x : T[\bar{\alpha} := \bar{T}']}\ \textsc{Ax}_{\forall} \qquad \frac{}{\Gamma \vdash x \text{ as } T : T}\ \textsc{Ax}_{\text{AS}}$$

$$\frac{\Gamma, x_1 : T_1, \dots, x_n : T_n \vdash t : \mathsf{bool} \qquad Q \in \{\forall, \exists\}}{\Gamma \vdash Q\ x_1 \dots x_n t : \mathsf{bool}}\ \textsc{Qua}$$

$$\frac{\Gamma \vdash x_1 : T_1 \qquad \dots \qquad \Gamma \vdash x_n : T_n \qquad \Gamma \vdash t : T}{\Gamma \vdash \lambda x_1 \dots x_n.t : T_1 \rightarrow \dots \rightarrow T_n \rightarrow T}\ \textsc{Lam}$$

$$\frac{\Gamma \vdash t_1 : T_1 \qquad \dots \qquad \Gamma \vdash t_n : T_n \qquad \Gamma \vdash f : T_1 \rightarrow \dots T_n \rightarrow T}{\Gamma \vdash f\ t_1\ \dots\ t_n : T}\ \textsc{App}$$

$$\frac{\Gamma \vdash t_1 : T_1 \qquad \dots \qquad \Gamma \vdash t_n : T_n \qquad \Gamma, x_1 : \mathrm{Gen}(T_1, \Gamma), \dots, x_n : \mathrm{Gen}(T_n, \Gamma) \vdash t : T}{\Gamma \vdash \text{let } ((x_1\ t_1) \dots (x_n\ t_n))\ t : T}\ \textsc{Let}$$

Figure 3: Typing rules

3.2 Typing commands

SMT-LIB commands can — and often do — change the global typing environment by introducing new function names. Some commands can change the typing environment by introducing new sorts, new variable names and new functions. In particular, assertions have to be type checked to verify that the term being asserted is indeed a boolean.

Some commands, such as declaring or defining new sorts can also have an indirect influence on the typing environment, as they modify the set of well-formed types. In this section, we suppose that checking the well-formedness of types has been done prior to typing terms.

A SMT-LIB script is formalized as an ordered set of commands $\mathcal{C}$. We represent a program as a couple $\langle \Gamma, \mathcal{C} \rangle$ where Γ represents the current typing environment, initially empty. In the rules below, we only detail commands whose syntax have been changed.

Note that FUNDEF rule follows the transformation explained in the SMT-LIB Standard (p.59).

$$\frac{\langle \Gamma, \{\textbf{define-fun}\ \forall_{\mathbb{T}}\bar{\alpha}(f\ ((x_1\ T_1) \dots (x_n\ T_n))\ T\ t)\} \cup \mathcal{C} \rangle}{\langle \Gamma \cup \{f : \forall_{\mathbb{T}}\bar{\alpha}.T_1 \times \dots \times T_n \rightarrow T\}, \{\textbf{assert}\ \forall_{\mathbb{T}}\bar{\alpha}\ ((\forall x_1 : T_1 \dots x_n : T_n\ f\ x_1 \dots x_n) = t)\} \cup \mathcal{C} \rangle}\ \textsc{fundef}$$

$$\frac{\langle \Gamma, \{\textbf{declare-fun}\ \forall_{\mathbb{T}}\bar{\alpha}(f\ (T_1 \dots T_n)\ T)\} \cup \mathcal{C} \rangle}{\langle \Gamma \cup \{f : \forall_{\mathbb{T}}\bar{\alpha}.T_1 \times \dots \times T_n \rightarrow T\}, \mathcal{C} \rangle}\ \textsc{fundec}$$

$$\frac{\langle \Gamma, \{\textbf{assert}\ \forall_{\mathbb{T}}\bar{\alpha}\ t\} \cup \mathcal{C} \rangle \qquad \Gamma \vdash t : \mathsf{bool}}{\langle \Gamma, \mathcal{C} \rangle}\ \textsc{Assert}$$

The typing system is used in particular to find monomorphic occurrences of polymorphic terms. In the next section, we will use it in a monomorphization procedure.

4 Monomorphization

Once we have checked that a set of commands is well-typed, we need to generate a Smt-Lib-compatible version of it, since we do not want to have to change the core of the solver. It means first that λ-terms must be eliminated and second, that the problem must be monomorphized. The elimination of λ-terms has been discussed in Sec. 1 and uses β-reduction. Only if the problem is still first-order after this elimination can we then try to compute a monomorphic version of it. Otherwise we will simply discard it.

Bobot and Paskevich [BP11] have shown the undecidability of computing a minimal monomorphic set formulas equivalent to an original set of polymorphic formulas. However, we still aim to present here a monomorphization method for polymorphic formulas. If needed, it should compute an over-approximation of the minimal monomorphic set. Our implied goal is that we hope monomorphization will be *good enough* in practice for most of our problems. The proposed monomorphization is expected to be sound with respect to the original polymorphic types. Hence, the unsatisfiability of the newly generated problem implies the unsatisfiability of the original problem but its satisfiability does not in general imply the satisfiability of the original problem.

Monomorphization example. Let us detail the example of Fig. 2 to show what we would like to achieve on this specific case. On this example, monomorphization is expected to fail, so that we can also explain how the procedure works and how we deal with failure. In the example, three function signatures are defined, where type variables are implicitly universally quantified:

- $\mathsf{emptyset}$: $\alpha_e \to \mathbb{B}$
- insert: $((\alpha_i \times (\alpha_i \to \mathbb{B})) \to \alpha_i \to \mathbb{B})$
- flat :$(\alpha_f \to \mathbb{B} \to \mathbb{B} \to \alpha_f \to \mathbb{B})$

The first step identifies polymorphic and monomorphic instances of terms through typing. In the example, we have three polymorphic formulas. These formulas are detailed below. For the sake of readability, we write type annotations only for co-domains, according to the initial function signatures, and hide annotations for term variables.

$$\mathsf{flat}^{\langle \alpha\to\mathbb{B}\rangle}(\mathsf{emptyset}^{\langle \alpha\to\mathbb{B}\to\mathbb{B}\rangle}) = \mathsf{emptyset}^{\langle \alpha\to\mathbb{B}\rangle} \tag{4.1a}$$

$$\forall ss : \alpha\to\mathbb{B}\to\mathbb{B} \\ (\mathsf{flat}^{\langle \alpha\to\mathbb{B}\rangle}(\mathsf{insert}^{\langle \alpha\to\mathbb{B}\to\mathbb{B}\rangle}(\mathsf{emptyset}^{\langle \alpha\to\mathbb{B}\rangle}, ss)) = \mathsf{flat}^{\langle \alpha\to\mathbb{B}\rangle}(ss)) \tag{4.1b}$$

$$\forall e : \alpha, s : \alpha\to\mathbb{B}, ss : \alpha\to\mathbb{B}\to\mathbb{B} \\ (\mathsf{flat}^{\langle \alpha\to\mathbb{B}\rangle}(\mathsf{insert}^{\langle \alpha\to\mathbb{B}\to\mathbb{B}\rangle}(\mathsf{insert}^{\langle \alpha\to\mathbb{B}\rangle}(e, s), ss)) \\ = \mathsf{insert}^{\langle \alpha\to\mathbb{B}\rangle}(e, \mathsf{flat}^{\langle \alpha\to\mathbb{B}\rangle}(\mathsf{insert}^{\langle \alpha\to\mathbb{B}\to\mathbb{B}\rangle}(s, ss)))) \tag{4.1c}$$

In Fig. 2, the unique monomorphic assertion is:

$$\forall s : \mathbb{Z}\to\mathbb{B}\ (\mathsf{flat}^{\langle \mathbb{Z}\to\mathbb{B}\rangle}(\mathsf{insert}^{\langle \mathbb{Z}\to\mathbb{B}\to\mathbb{B}\rangle}(s, \mathsf{emptyset}^{\langle \mathbb{Z}\to\mathbb{B}\to\mathbb{B}\rangle})) = s)$$

Such monomorphic assertions drive the procedure. Basically, they provide the set of ground types that forms the basis for the generation of monomorphic instances of polymorphic functions.

The second step consists in computing the set of new terms derived from the injection of monomorphic elements. Using type substitutions, we can generate monomorphic specializations $\mathcal{F} = \{\mathsf{emptyset}_{[\alpha_e := \mathbb{Z}\to\mathbb{B}]}, \mathsf{insert}_{[\alpha_i := \mathbb{Z}\to\mathbb{B}]}, \mathsf{flat}_{[\alpha_f := \mathbb{Z}]}\}$ of the polymorphic functions. We try to substitute the polymorphic occurrences of the three terms emptyset, insert, flat by their monomorphic counterpart whenever we can in the polymorphic terms 4.1a, 4.1b and 4.1c, using a leftmost-innermost strategy. This generates the following new set of monomorphic assertions:

$$\mathsf{flat}^{\langle\mathbb{Z}\to\mathbb{B}\rangle}(\mathsf{emptyset}^{\langle\mathbb{Z}\to\mathbb{B}\to\mathbb{B}\rangle}) = \mathsf{emptyset}^{\langle\mathbb{Z}\to\mathbb{B}\rangle}_{[\boldsymbol{\alpha_e}:=\mathbb{Z}]} \tag{4.2a}$$

$$\begin{aligned}&\forall ss : \mathbb{Z}\to\mathbb{B}\to\mathbb{B}\\ &\quad(\mathsf{flat}^{\langle\mathbb{Z}\to\mathbb{B}\rangle}(\mathsf{insert}^{\langle\mathbb{Z}\to\mathbb{B}\to\mathbb{B}\rangle}(\mathsf{emptyset}^{\langle\mathbb{Z}\to\mathbb{B}\rangle}_{[\boldsymbol{\alpha_e}:=\mathbb{Z}]}, ss)) = \mathsf{flat}^{\langle\mathbb{Z}\to\mathbb{B}\rangle}(ss))\end{aligned} \tag{4.2b}$$

$$\begin{aligned}&\forall e : \mathbb{Z}, s : \mathbb{Z}\to\mathbb{B}, ss : \mathbb{Z}\to\mathbb{B}\to\mathbb{B}\\ &\quad(\mathsf{flat}^{\langle\mathbb{Z}\to\mathbb{B}\rangle}(\mathsf{insert}^{\langle\mathbb{Z}\to\mathbb{B}\to\mathbb{B}\rangle}(\mathsf{insert}^{\langle\mathbb{Z}\to\mathbb{B}\rangle}_{[\boldsymbol{\alpha_i}:=\mathbb{Z}]}(e, s), ss))\\ &\quad= \mathsf{insert}^{\langle\mathbb{Z}\to\mathbb{B}\rangle}_{[\boldsymbol{\alpha_i}=\mathbb{Z}]}(e, \mathsf{flat}^{\langle\mathbb{Z}\to\mathbb{B}\rangle}(\mathsf{insert}^{\langle\mathbb{Z}\to\mathbb{B}\to\mathbb{B}\rangle}(s, ss))))\end{aligned} \tag{4.2c}$$

The procedure uses all monomorphic terms, and thus generates a number of new monomorphic assertions. At this point, one full pass of the procedure has been executed. This is repeated while new monomorphic type instances are created. For example, the problem after the first full pass now has uncovered new possible monomorphizations: $\mathsf{emptyset}_{[\mathbb{Z}/\alpha_e]}$ (in 4.2a and 4.2b) and $\mathsf{insert}_{[\mathbb{Z}/\alpha_i]}$ (in 4.2c).

The procedure thus stops on a final problem if it does not uncover any more monomorphic instances. There, only two things can happen:

- if polymorphic terms are only found in their original place (i.e. as subterms of the original problem) then we have found a monomorphic expression of the original polymorphic problem, if we remove the original polymorphic assertions and functions from the final problem.
- if polymorphic terms are still present, then we declare that we have failed to find a monomorphic expression of the original problem and halt there.

On the example, the procedural steps we have applied will never terminate because in the term (flat emptyset) = emptyset, a new monomorphic type can be inferred for the leftmost emptyset at any instantiation of the rightmost emptyset , which will be fed to the rightmost one, thus looping forever.

Monomorphization fixpoint. The proposed monomorphization procedure is now summarized. Let T represent the set of term occurrences of the original problem. Initially, the sets of monomorphic and polymorphic term occurrences are empty.

1. Apply type inference and divide the terms in T into two sets M and P of monomorphic and polymorphic term occurrences. If they are the same as the previous M and P, we have reached a fixpoint and can stop. Otherwise, go to step 2.
2. Let $M = \{m_1, m_2, \dots m_n\}$ and $P = \{p_1, p_2, \dots p_m\}$. For each (m_i, p_j), such that $m_i \in M$, $p_j \in P$ and $m_i = p_j$ (i.e. m_i and p_j are two occurrences of the same term), substitute the

polymorphic p_j by its monomorphic m_i in the term t where p_j occurs as subterm, thus deriving a new term $t_{ij} = t[p_j := m_i]$.

At the end of this step, we will have a new set of terms from the various possible pairings between monomorphic and polymorphic occurrences of the same term, generating a new set of term occurrences T'. Let the new problem be represented by $T \cup T'$.

Non-termination. The fact that our procedure is possibly non-terminating is mitigated by the fact that, in practice, we impose restrictions on time or in this case, on the number of full passes. However, we would like to be able to guess possibly infinite expansions because we know we will not be able to guarantee a sound and complete monomorphization. To this effect, we have conjectured the following criterion, which depends on unification [Rob65]:

Conjecture 1 (Non-termination criterion)**.** *Let t and s be terms and s_1 and s_2 be two occurrences of s in the term t. Let T_1 be the type inferred for s_1 and T_2 the one of s_2. If T_1 and T_2* cannot *be unified* because of a failing occur check *then the monomorphization procedure will not terminate.*

5 Related work

The use of polymorphic logics on top of many-sorted or mono-sorted logics has received specific attention in the last few years.

In the context of the Caduceus and Why [FM07, BFMP11], Couchot and Lescuyer [CL07] describe how to translate ML-style polymorphic formulas into untyped and multi-sorted versions of the original problem.

This method is refined by Bobot and Paskevich [BP11] who show a 3-staged treatment of polymorphic formulas to translate them into many-sorted versions, including various possibilities for the last translating step. Their proposals particularly take care of protecting data types which are known to be handled by decision procedures by the targeted SMT-solvers. This work is further detailed in Bobot's thesis [Bob11].

Leino and Rümmer [LR10] consider the higher-order polymorphic specification language of the Boogie2 tool [Lei08] that has to be translated to SMT-solvers which in general do not handle polymorphism. They present two translations, one using type guards, the other adding types as further function arguments.

These two last approaches already present various advanced techniques to translate polymorphism formulas for many-sorted SMT-solvers, each one coming from their experience and needs, but do not tackle monomorphization. In the case of Bobot and Paskevich, their proof of the undecidability of the monomorphization makes clear the reason why, while Leino and Rümmer leave it as a possible further optimization.

Bobot *et al.* [BCCL08] have added built-in support for polymorphism inside the Alt-Ergo prover [BCC$^+$08]. Supporting polymorphic types at the solver level would indeed simplify the addition of polymorphism at the specification level. We chose to keep things separated (for now).

There is also a large body of work on the translation of typed higher-order-logic into untyped first-order logic. In the context of using automation to help discharge proofs Hurd [Hur03] or Meng and Paulson [MP08] can however rely on the type-checking capabilities of higher-order provers to verify automated but untyped first-order proofs. Therefore they can even use unsound translations and leave to the higher-order prover the task of checking the soundness of the proof. In order to use SMT-solvers inside Isabelle/HOL [Pau94], the monomorphization step of Blanchette *et al.* [BBP11] bears a strong similarity to our proposal.

6 Conclusion and further work

We have presented a summary of two backward compatible syntactic extensions made to the SMT-LIB standard: λ-terms and polymorphism. We have shown how to deal with λ-terms through β-reduction. We also have presented how we attempt to generate a monomorphic version of our polymorphic problem using a fixpoint-like procedure. This procedure heavily uses a Damas-Hindley-Milner like type inference algorithm to discover relevant monomorphic instances of polymorphic terms.

The support for the proposed syntactic extensions is currently being implemented in the development version of veriT. λ-terms are already supported and there is preliminary support for polymorphic SMT-LIB scripts. We are currently testing the monomorphization process to see how it behaves in practice.

The preservation of satisfiability means that, even in the case of a non-terminating monomorphization, we could devise a strategy to correctly use an unsatisfiable result at any step of the monomorphization process. Indeed, this would mean that we have found an unsatisfiable subset of the initial problem.

This strategy would be similar to depth-first *iterative deepening* [Kor85], which has been heavily used in provers based on the tableau method [Smu95, DGHP99]. In tableaux, this is used to generate possible term instantiations for universally quantified variables in order to find a model refuting the original formula. In our case, after each deeper monomorphization step, the SMT-solver would get to try a partial and new monomorphic problem containing only a subset of possible type instantiations. If it can prove the unsatisfiability of this new problem, we can stop. If not, we can — and should to preserve completeness — continue. In practice, this iterative procedure will be bounded either by a time limit or by a given number of steps.

We believe this work is a step towards a more generalized use of polymorphism in the context of SMT-solvers. These efforts might converge into an Alt-Ergo-like solution for provers supporting SMT-LIB, indeed building polymorphism support *in* the prover and not only at the syntactic level.

Acknowledgments. We warmly thank the anonymous reviewers for their helpful feedback and constructive criticism, which already show promises of future fruitful discussions about the subject of this paper.

References

[BBP11] Jasmin C. Blanchette, Sascha Böhme, and Lawrence C. Paulson. Extending Sledgehammer with SMT solvers. In Nikolaj Børner and Viorica Sofronie-Stokkermans, editors, *Automated Deduction*, volume 6803 of *LNCS*, pages 116–130. Springer, 2011.

[BCC$^+$08] François Bobot, Sylvain Conchon, Évelyne Contejean, Mohamed Iguernelala, Stéphane Lescuyer, and Alain Mebsout. The Alt-Ergo Automated Theorem Prover, 2008. `http://alt-ergo.lri.fr/`.

[BCCL08] François Bobot, Sylvain Conchon, Évelyne Contejean, and Stéphane Lescuyer. Implementing Polymorphism in SMT solvers. In Clark Barrett and Leonardo de Moura, editors, *SMT 2008: 6th International Workshop on Satisfiability Modulo*, volume 367 of *ACM International Conference Proceedings Series*, pages 1–5, 2008.

[BFMP11] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. Why3: Shepherd Your Herd of Provers. In *Boogie 2011: First International Workshop on Intermediate Verification Languages*, pages 53–64, Wrocław, Poland, August 2011.

[Bob11] François Bobot. *Logique de séparation et vérification déductive.* Phd thesis, Université Paris-Sud, December 2011.

[BP11] François Bobot and Andrey Paskevich. Expressing polymorphic types in a many-sorted language. In Cesare Tinelli and Viorica Sofronie-Stokkermans, editors, *FroCoS*, volume 6989 of *Lecture Notes in Computer Science*, pages 87–102. Springer, 2011.

[BST10] Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB Standard: Version 2.0. In A. Gupta and D. Kroening, editors, *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK)*, 2010.

[CL07] Jean-François Couchot and Stéphane Lescuyer. Handling Polymorphism in Automated Deduction. In *21th International Conference on Automated Deduction (CADE-21)*, volume 4603 of *LNCS (LNAI)*, pages 263–278, Bremen, Germany, July 2007.

[Dé10] David Déharbe. Automatic Verification for a Class of Proof Obligations with SMT-Solvers. In *Abstract State Machines, Alloy, B and Z (ABZ 2010)*, volume 5977 of *Lecture Notes in Computer Science*, pages 217–230. Springer, 2010.

[Dé13] David Déharbe. Integration of SMT-solvers in B and Event-B development environments. *Science of Computer Programming*, 78(3):310–316, 2013.

[DFGV12] David Déharbe, Pascal Fontaine, Yoann Guyot, and Laurent Voisin. SMT Solvers for Rodin. In *Proc. Abstract State Machines, Alloy, B, VDM, and Z (ABZ 2012)*, volume 7316 of *Lecture Notes in Computer Science*, pages 194–207. Springer, 2012.

[DGHP99] Marcello D'Agostino, Dov M. Gabbay, Reiner Hähnle, and Joachim Posegga, editors. *Handbook of Tableau Methods.* Springer, 1999.

[FM07] Jean-Christophe Filliâtre and Claude Marché. The Why/Krakatoa/Caduceus Platform for Deductive Program Verification. In Werner Damm and Holger Hermanns, editors, *CAV*, volume 4590 of *Lecture Notes in Computer Science*, pages 173–177. Springer, 2007.

[Hin69] Roger Hindley. The principle type-scheme of an object in combinatory logic. *Trans. Amer. Math. Soc.*, 146:29–60, 1969.

[Hur03] Joe Hurd. First-Order Proof Tactics in Higher-Order Logic Theorem Provers. In Myla Archer, Ben Di Vito, and César Muñoz, editors, *Design and Application of Strategies/Tactics in Higher Order Logics (STRATA 2003)*, number NASA/CP-2003-212448 in NASA Technical Reports, pages 56–68, September 2003.

[Kor85] Richard E. Korf. Depth-First Iterative-Deepening: An Optimal Admissible Tree Search. *Artif. Intell.*, 27(1):97–109, 1985.

[Lei08] K. Rustan. M. Leino. *This is Boogie 2*, 2008.

[LR10] K. Rustan M. Leino and Philipp Rümmer. A Polymorphic Intermediate Verification Language: Design and Logical Encoding. In Javier Esparza and Rupak Majumdar, editors, *TACAS*, volume 6015 of *Lecture Notes in Computer Science*, pages 312–327. Springer, 2010.

[Mil78] Robin Milner. A Theory of Type Polymorphism in Programming. *J. Comput. Syst. Sci.*, 17(3):348–375, 1978.

[MP08] Jia Meng and Lawrence C. Paulson. Translating Higher-Order Clauses to First-Order Clauses. *J. Autom. Reasoning*, 40(1):35–60, 2008.

[Pau94] Lawrence C. Paulson. *Isabelle - A Generic Theorem Prover (with a contribution by T. Nipkow)*, volume 828 of *Lecture Notes in Computer Science.* Springer, 1994.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages.* MIT Press, 2002.

[Rob65] John Alan Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, 1965.

[Smu95] R.M. Smullyan. *First-order Logic.* Dover books on advanced mathematics. Dover, 1995.