# Safely Encoding B Proof Obligations in SMT-LIB*

Vincent Trélat[0009−0006−4143−3939]

Université de Lorraine, CNRS, Inria, LORIA, Nancy, France
vincent.trelat@inria.fr

**Abstract.** This paper presents a sound encoding of B proof obligations in SMT-LIB 2.7, leveraging the recent extensions of SMT-LIB to higher-order logic. Our encoding improves upon previous approaches by eliminating uninterpreted membership predicates and by avoiding the complexity of encoding functions as relations. Through SMT-LIB's support for higher-order constructs, we achieve a more natural representation of B's set theory, while ensuring soundness of the translation. Preliminary experimental results are promising and indicate that our encoding allows certain proof obligations that previously failed to be discharged.

**Keywords:** B Proof Obligations · SMT-LIB · Higher-Order Logic · Set Theory

## 1 Introduction

Formal specification techniques like the B method [1] have been increasingly used in the development of high-assurance software systems, particularly in safety-critical domains such as transportation. Verifying properties of B models has been facilitated by integrated development environments such as Atelier B [8] and Rodin [3], which provide a framework for development and verification. Proof obligations are systematically generated from B components and translated into machine-verifiable formats to be fed into automated solvers [10]. While automated theorem proving has made steady progress over the past decades, a small but significant proportion of applications still require human intervention for verification in practice. Recent advances in automated reasoning have opened the door to higher-order logic [4] and have led to the introduction of higher-order constructs in SMT-LIB 2.7, thus creating new opportunities for the encoding.

This paper presents a novel approach to encoding B proof obligations that leverages higher-order logic, addressing limitations in previous translation methods and offering a more natural and computationally efficient encoding. Background context on B and SMT is provided in Sec. 2, followed by the formalization of both languages with their respective type systems in Sec. 3 and Sec. 4. Section 5 introduces an encoding that leverages SMT-LIB 2.7's higher-order features to represent sets and functions. Preliminary experimental results are shown in Sec. 6.

* This work is supported by the ANR project BLASST (ANR-21-CE25-0010).

## 2 Background

This section provides an overview of the B method and SMT-LIB, focusing on the aspects relevant to the encoding of B proof obligations.

### 2.1 The B Method

The B Method is a formal approach to software development, enabling the specification of abstract machines that can be progressively refined into concrete, implementable code while preserving correctness through proof obligations. From specification to implementation, all artifacts are expressed in a single formalism, the B language. At its core, the B language uses set theory [18,13], treating all objects—including relations and functions—as sets.

This theoretical basis enables rigorous reasoning about software properties while remaining practical for industrial use. Notable applications include various Communication-Based Train Control (CBTC) systems worldwide such as the Métro Line 14 in Paris, which generated about 27,800 proof obligations, among which 2,250 required manual intervention [15].

Atelier B and Rodin provide tools for generating proof obligations from B components and interacting with external provers [12], which involves translating proof obligations into machine-verifiable formats such as SMT-LIB. Atelier B's Proof Obligation Generator (POG) derives proof obligations from refinement steps, well-formedness and consistency checks on B components.

*Example 1.* Refining a set to an array would generate obligations ensuring that (a) the array bounds can accommodate the maximum set size, (b) array operations preserve the abstract set's properties and (c) the invariant linking abstract and concrete representations is maintained. □

These obligations are stored in an XML-based format [9], essentially representing B expressions in a structured way. A detailed example of this format is provided in Appendix A.

### 2.2 Satisfiability Modulo Theories

The Satisfiability Modulo Theories (SMT) problem extends Boolean satisfiability by incorporating background theories such as arithmetic, arrays, and uninterpreted functions. SMT solvers determine whether a formula is satisfiable with respect to these theories. Such formulas are written in the SMT-LIB language, which provides a standard interface for interacting with SMT solvers. The expressiveness of SMT-LIB was recently extended from multi-sorted first-order logic to higher-order logic in version 2.7 [6]. In particular, the introduction of the arrow type for functions and the lambda binder significantly enhances the expressiveness of the language, allowing for more direct representations of higher-order constructs.

Solvers are gradually adapting to these new features. Thanks to recent advances in higher-order instantiation [4,20,21], the cvc5 solver [5] now provides partial support for higher-order reasoning, as shown in the following example.

*Example 2.* Consider the following SMT-LIB 2.7 script, which declares a higher-order predicate `P` that takes functions from integers to integers as arguments—notice the directive `(set-logic HO_ALL)`, which enables higher-order reasoning. The script asks for a satisfiability check of the formula asserting the existence of a function `f` such that `P(f)` is true.

```
(set-logic HO_ALL)
(declare-const P (-> (-> Int Int) Bool))
(assert (exists ((f (-> Int Int))) (= (P f) true)))
(check-sat)
```

cvc5 returns `sat` and provides the following model for `P`:

```
(define-fun P ((f (-> Int Int))) Bool (= (lambda ((x Int)) 0) f))
```

□

## 3 Formalizing B Proof Obligations

The following section formalizes the B language and its type system.

### 3.1 Syntax

The grammar formalized below is intentionally simplified compared to the full syntax of the B language, though it is not minimal, to avoid overly complex definitions for simple constructs. Let $\mathcal{V}$ represent a collection of variables. A term $t$ is then defined inductively as one of these constructs:

- literals, where $v$ is a variable in $\mathcal{V}$, $n$ is an integer, and $b$ is a Boolean:
$$v \mid n \mid b \mid \mathbb{Z} \mid \mathbb{B}$$
- arithmetic operations:
$$t +^{\mathrm{B}} t \mid t -^{\mathrm{B}} t \mid t *^{\mathrm{B}} t \mid t \leq^{\mathrm{B}} t \mid t =^{\mathrm{B}} t$$
- Boolean operations:
$$t \wedge^{B} t \mid \neg^{\mathrm{B}} t$$
- set operations:
$$t \mapsto^{\mathrm{B}} t \mid t \in^{\mathrm{B}} t \mid \mathcal{P}^{\mathrm{B}}(t) \mid t \times^{\mathrm{B}} t \mid t \cap^{\mathrm{B}} t \mid t \cup^{\mathrm{B}} t$$
- functions:
$$t \nrightarrow^{\mathrm{B}} t \mid t(t) \mid \min t \mid \max t \mid |t|^{\mathrm{B}}$$
- binders, where $v_1, \ldots, v_n$ are variables in $\mathcal{V}$:
$$\{v_1, \ldots, v_n \in t \mid t\}^{\mathrm{B}} \mid \lambda^{\mathrm{B}} v_1, \ldots, v_n \cdot (t \mid t) \mid \forall^{\mathrm{B}} v_1, \ldots, v_n \in t \cdot t$$

Additional syntax constructs are defined in terms of these base constructs, such as disjunction ($\vee^{\mathrm{B}}$), implication ($\Rightarrow^{\mathrm{B}}$), existential quantifier ($\exists^{\mathrm{B}}$), and a variety of interrelated constructs defining specific classes of functions, encompassing all combinations of partial and total functions with injective, surjective and bijective properties. They are provided in Appendix B.

### 3.2 Type system

Formally, the B language is untyped. While the B-Book [2] introduces a relation associated with a typing judgment, this relation is fundamentally part of the language semantics rather than a separate type system layer. Instead, it should be understood as a membership constraint on terms, thus forming the foundation for the generation of proof obligations. Strictly speaking, the language does not require a type system to be well-defined, as it is grounded in set theory. However, to make the language practical for software specification and development, it is necessary to introduce facilities for manipulating machine integers and implementable objects that cannot be easily represented within pure set theory, as illustrated in the following example.

*Example 3.* If the integer 3 is to be treated as an immediate value, in set theory it is represented as follows:

$$3 \stackrel{\text{def}}{=} \{\varnothing, \{\varnothing\}, \{\varnothing, \{\varnothing\}\}, \{\varnothing, \{\varnothing\}, \{\varnothing, \{\varnothing\}\}\}\} \tag{1}$$

Moreover, even a seemingly simple statement such as $3 = 2 + 1$ requires a non-trivial proof involving extensionality and case analysis. □

That being said, the B language can indeed be equipped with a type system, which serves to enforce the well-formedness of expressions. Atelier B includes a well-formedness checker that computes variable domains and verifies that expressions conform to the well-formedness rules of the B language, as shown in the following example.

*Example 4.* In ZFC set theory, the expression $1 + \top$ is well-formed and definitionally equal to the set $\mathbb{B}$ of Booleans, or the natural number 2, but it is not a valid expression in B. □

The fact that integer and Boolean literals are assigned distinct constructs in the syntax motivates the introduction of distinct types for integers and Booleans, as mentioned in the last two examples.

Additional constructs, such as the minimum, maximum, and cardinality functions were included in the syntax. While these could theoretically be defined using more fundamental constructs, it would complicate the encoding unnecessarily. Contrary to the rules presented in the B Book, in our setting they are simply typed as functions into integers, and the premises for their correct application—for instance, that the cardinal is applied to finite sets—is handled at the semantics level.

It is therefore sufficient to consider a type system with four base types: integers, Booleans, (power) sets, and (cartesian) products.

$$\tau ::= \texttt{int} \mid \texttt{bool} \mid \texttt{set}\ \tau \mid \tau \times^B \tau$$

*Remark 1.* This type system could be extended to also model user-defined sets, which can be used as type parameters for functions and relations. For the moment, such sets, along with enumerations, are assimilated to sets of integers.

$$\frac{\Gamma \vdash^{\mathrm{B}} A : \mathtt{set}(\alpha) \qquad \Gamma \vdash^{\mathrm{B}} B : \mathtt{set}(\beta)}{\Gamma \vdash^{\mathrm{B}} A \nrightarrow^{\mathrm{B}} B : \mathtt{set}(\mathtt{set}(\alpha \times^{\mathrm{B}} \beta))} \text{ (pfun)}$$

$$\frac{\Gamma \vdash^{\mathrm{B}} x : \alpha \qquad \Gamma \vdash^{\mathrm{B}} f : \mathtt{set}(\alpha \times^{\mathrm{B}} \beta)}{\Gamma \vdash^{\mathrm{B}} f(x) : \beta} \text{ (app)}$$

$$\frac{\forall i \in [\![1, n]\!],\ \Gamma \vdash^{\mathrm{B}} D_i : \mathtt{set}(\alpha_i) \qquad \Gamma, v_1 : \alpha_1, \ldots, v_n : \alpha_n \vdash^{\mathrm{B}} P : \mathtt{bool}}{\Gamma \vdash^{\mathrm{B}} \{v_1, \ldots, v_n \in^{\mathrm{B}} D_1 \times^{\mathrm{B}} \ldots \times^{\mathrm{B}} D_n \mid P\} : \mathtt{set}(\alpha_1 \times^{\mathrm{B}} \ldots \times^{\mathrm{B}} \alpha_n)} \text{ (collect)}$$

**Fig. 1.** Typing rules for partial functions, function application and set comprehension. In the last rule, variables $v_1, \ldots v_n$ are fresh variables and must not appear in $\Gamma$.

### 3.3 Typing rules

Typing is then naturally defined inductively on terms within a typing context, which is a map associating variables with their representive types. The typing rules are formally expressed judgments $\Gamma \vdash^{\mathrm{B}} t : \tau$, indicating that term $t$ has type $\tau$ in context $\Gamma$. A selection of three such rules is illustrated in Fig. 1, covering partial functions, function application, and set comprehension. The full set of rules is provided in Appendix C.

*Remark 2.* A POG file always represents closed terms: therefore, any proof obligation $PO$ must verify the judgment with any typing context, and in particular, the empty context:

$$\vdash^{\mathrm{B}} PO : \mathtt{bool}$$

## 4 Formalizing a subset of SMT-LIB

Similarly, a subset of the SMT-LIB language is formalized, based on the official language specification [6], with simplifications.

### 4.1 Syntax

Let $\mathcal{V}$ be a collection of variables. A term $t$ is then defined inductively as one of these constructs:

- literals, where $v$ is a variable in $\mathcal{V}$, $n$ is an integer, and $b$ is a Boolean:

$$v \mid n \mid b$$

- arithmetic operations:

$$t +^{\mathrm{S}} t \mid t -^{\mathrm{S}} t \mid t *^{\mathrm{S}} t$$

- Boolean operations:

$$t \wedge^{\mathrm{S}} t \mid t \vee^{\mathrm{S}} t \mid \neg^{\mathrm{S}} t \mid t =^{\mathrm{S}} t \mid t \leq^{\mathrm{S}} t \mid t \Rightarrow^{\mathrm{S}} t \mid \text{if } t \text{ then } t \text{ else } t$$

$$\frac{\Gamma \vdash^{\mathrm{S}} x : \alpha}{\Gamma \vdash^{\mathrm{S}} \texttt{some}\ x : \texttt{Option}\ \alpha}\ (\text{some}) \qquad \frac{}{\Gamma \vdash^{\mathrm{S}} \texttt{as none}\ \alpha : \texttt{Option}\ \alpha}\ (\text{none})$$

$$\frac{\Gamma \vdash^{\mathrm{S}} x : \alpha \qquad \Gamma \vdash^{\mathrm{S}} y : \beta}{\Gamma \vdash^{\mathrm{S}} \texttt{pair}\ x\ y : \texttt{Pair}\ \alpha\ \beta}\ (\text{pair}) \qquad \frac{\Gamma, v_1 : \alpha_1, \ldots, v_n : \alpha_n \vdash^{\mathrm{S}} P : \texttt{bool}}{\Gamma \vdash^{\mathrm{S}} \forall^{\mathrm{S}} v_1 : \alpha_1 \ldots, v_n : \alpha_n, P : \texttt{Bool}}\ (\text{forall})$$

**Fig. 2.** Typing rules for options, pairs and forall. In the last rule, variables $v_1, \ldots v_n$ are fresh variables and must not appear in $\Gamma$.

– binders, where $v_1, \ldots, v_n$ are variables in $\mathcal{V}$ and $\alpha_1, \ldots, \alpha_n$ are SMT-LIB types (which are defined below in Sec. 4.2):

$$\lambda^{\mathrm{S}} v_1 : \alpha_1, \ldots, v_n : \alpha_n, t \mid \forall^{\mathrm{S}} v_1 : \alpha_1, \ldots, v_n : \alpha_n, t \mid \exists^{\mathrm{S}} v_1 : \alpha_1, \ldots, v_n : \alpha_n, t$$

– other constructs, where $\alpha$ is an SMT-LIB type:

$$t(t) \mid \text{as } t\ \alpha \mid \text{some } t \mid \text{none} \mid \text{the } t \mid \text{pair } t\ t \mid \text{fst } t \mid \text{snd } t$$

*Remark 3.* Note that the some, none, the, pair, fst and snd constructs are not part of the official SMT-LIB language, but are introduced here to simplify the encoding of certain constructs. They are formally defined as constructors of a polymorphic datatype in the following section.

### 4.2 Type system

We formalize a simplified version of SMT-LIB's type system covering the restricted syntax defined in the previous section. Recalling that SMT-LIB functions are total, a standard approach to representing partial functions within this framework involves encoding them as total functions whose codomain includes an additional element denoting the absence of value, as commonly achieved via the so-called `Option` type. The type system of SMT-LIB is simplified to a minimal form that accommodates the requirements of the formalization as follows:

$$\tau ::= \texttt{Int} \mid \texttt{Bool} \mid \texttt{Unit} \mid \tau \to^{\mathrm{S}} \tau \mid \texttt{Option}\ \tau \mid \texttt{Pair}\ \tau\ \tau$$

*Remark 4.* The `Option` and `Pair` types, along with the `some`, `none`, and `pair` constructors are included as terms of the syntax to circumvent unnecessary complexity in the formalization of the language. Formally, they are defined as the following polymorphic datatypes:

```
(declare-datatype Pair (par (T1 T2) ((pair (fst T1) (snd T2)))))
(declare-datatype Option (par (T) ((some (the T)) (none))))
```

The judgment $\Gamma \vdash^{\mathrm{S}} t : \tau$, denotes that term $t$ has type $\tau$ in context $\Gamma$. The complete typing rules are provided in Appendix D. Four of these rules are presented in Fig. 2, covering the `some`, `none`, `pair`, and `forall` constructs.

Similarly to the B language, the semantics of SMT-LIB is defined in a set-theoretic setting for type-correct terms, but is out of the scope of this paper.

## 5 Encoding B in SMT-LIB

With the formalizations of B and SMT-LIB established, the encoding rules can now be formulated. The translation of POG files in SMT-LIB format has historically been handled by the tool *ppTransSMT* [16]. This approach requires complex encodings, particularly for set-theoretic constructs and functions. The limitations of this encoding strategy, along with the new possibilities offered by SMT-LIB 2.7's higher-order features, are presented in the following. The encoding rules are then detailed, followed by a discussion on soundness.

### 5.1 Prior approach

Versions of SMT-LIB prior to 2.7 were limited to multi-sorted first-order logic; therefore higher-order elements required encoding through first-order reductions, leaving certain components uninterpreted and only partially specified. The encoding performed by *ppTransSMT* was therefore based on a first-order reduction and monomorphization of polymorphic constructs like membership, leaving any higher-order terms uninterpreted. The set-theoretic constructs of the B language were encoded using two distinct sorts: a unary sort `P` representing the powerset operator, and a binary sort `C` encoding the cartesian product.

```
(declare-sort P 1)
(declare-sort C 2)
```

This formalization approach, while operational, presents two limitations. The first limitation stems from its conflation of set-theoretic concepts (specifically the powerset and cartesian product operations) with type-theoretic notions, two paradigms that are based on divergent foundational principles [7]: *collections* and *constructions* respectively.

*Example 5.* To visualize this theoretical divergence in practice, consider the type of integers that are equal to their successor and the type of prime numbers between 13 and 17, which are both uninhabited. For the sake of readability, they can be informally expressed as:

$$\{x\colon \texttt{int} \mid x = x + 1\} \quad \text{and} \quad \{x\colon \texttt{int} \mid x \in \mathbb{P} \wedge 13 < x < 17\}$$

where $\mathbb{P}$ denotes the set of prime numbers. For these to be definitionally equal, their constructions must be definitionally equal as well:

$$\forall x\colon \texttt{int}, x = x + 1 \stackrel{\text{def}}{=} x \in \mathbb{P} \wedge 13 < x < 17$$

This clearly does not hold, showing that both of these uninhabited types are not definitionally equal, despite being propositionally equal (since both sides of the equation are logically equivalent to $\bot$). □

The second limitation arises from the necessity to declare distinct uninterpreted membership predicates for each sort combination. These predicates must then be axiomatically specified to define concrete sets.

*Example 6.* Let $S := \{a, b, c\}$ be an enumeration of three sets.[1] The encoding of $S$ via the unary sort `P` is as follows:

```
(declare-const a (P (Int)))
(declare-const b (P (Int)))
(declare-const c (P (Int)))
(declare-const S (P (P (Int))))
(declare-const ∈₁ ((P (Int)) (P (P (Int)))) Bool)
(assert (forall ((x (P (Int))))
  (= (∈₁ x S) (or (= x a) (= x b) (= x c)))))
```

With such a definition, the claim "$a \in S$" is not true by definition but is an assertion to be proven; it is relevant to the specification of $S$ which is not a definition. □

These limitations motivate an alternative encoding approach that capitalizes on SMT-LIB 2.7's support for higher-order logic to achieve a more faithful and direct representation of set-theoretic constructs.

### 5.2 Sets as characteristic predicates

The primary notable distinction in the design of this new encoding is the representation of sets as predicates, i.e., total functions that evaluate to the truth value of any element being a member of a set. Note that this approach is only feasible thanks to the support of higher-order constructs brought by SMT-LIB 2.7, since sets of sets are encoded as higher-order functions.

**Definition 1 (Characteristic predicate).** *Let $S$ be a set. The characteristic predicate $\hat{S}$ of $S$ is defined as the following total function:*

$$\forall x, \hat{S}(x) \stackrel{def}{=} x \in S$$

**Lemma 1 (Existence and uniqueness).** *In any set theory embedding extensional reasoning on sets, the characteristic predicate of a set as defined above describes the set correctly and is unique up to logical equivalence.*

*Proof.* Any set $S$ is extensionally equal to $\{x \mid \hat{S}(x)\}$. Let then $P$ be a predicate such that the following holds:

$$\forall x,\ x \in S \Leftrightarrow P(x)$$

Then by definition of $\hat{S}$, $\forall x, \hat{S}(x) \Leftrightarrow P(x)$. ■

It follows that for any type $\alpha$, any type representing homogeneous sets containing elements of type $\alpha$ is isomorphic to the type $\alpha \to^{\text{S}} \texttt{Bool}$. Note that B

[1] Note that the constants $a, b, c$ are not assigned a specific type *a priori* and are encoded as sets of integers for simplicity. Alternatively, they could be declared as sets of an abstract nullary sort.

types can be inductively embedded into SMT-LIB types in an axiomatic way, via the following mapping $\xi$:

$$\xi(\texttt{int}) = \texttt{Int} \qquad \xi(\texttt{bool}) = \texttt{Bool}$$
$$\xi(\texttt{set}\ \alpha) = \xi(\alpha) \to^{\mathrm{S}} \texttt{Bool} \qquad \xi(\alpha \times^{\mathrm{B}} \beta) = \texttt{Pair}\ \xi(\alpha)\ \xi(\beta)$$

where $\alpha$ and $\beta$ are B types. This implies that B integers (resp. Booleans) can be semantically interpreted as SMT-LIB integers (resp. Booleans). For instance, B sets of integers are represented as functions mapping integers to Booleans. This is the basis of the encoding of sets in SMT-LIB. Moreover, the mapping $\xi$ can be extended to accommodate uninterpreted sorts, as in the example below.

*Example 7.* A set `S` of elements of type `A` is declared in SMT-LIB as follows:

```
(declare-sort A 0)
(declare-const S (-> A Bool))
```

□

Note that thanks to the `lambda` binder, any concrete set may be defined either via a specification or directly as a function.

*Example 8.* The encoding of the set of natural numbers, first specified, then inlined, yields:

```
(declare-const Nat (-> Int Bool))
(assert (forall ((x Int)) (= (Nat x) (>= x 0))))
```

```
(define-const Nat_λ (-> Int Bool) (lambda ((x Int)) (>= x 0)))
```

□

The latter option is chosen, so that set membership can be verified through definitional equality with one $\beta$-reduction.

*Example 9.* Revisiting the set $S = \{a, b, c\}$ from example 6, its encoding using characteristic predicates is as follows:

```
(declare-const a (-> Int Bool))
(declare-const b (-> Int Bool))
(declare-const c (-> Int Bool))
(define-const S (-> (-> Int Bool) Bool)
  (lambda ((x Int)) (or (= x a) (= x b) (= x c))))
```

□

This encoding offers the advantage of being more direct: it eliminates the need to declare membership predicates, reducing membership to function application, which is polymorphic. It suffers from a notable drawback, as it erases the structure of sets, which can be a hindrance when attempting proof reconstruction.

### 5.3 Functions as functions

Another notable distinction in our encoding is the direct representation of functions in B as actual SMT-LIB functions rather than encoding them as functional relations, i.e., sets of pairs. This is achieved by leveraging the inherent function type (`->`) introduced in SMT-LIB 2.7 and allowing for a more natural representation of functions. This approach not only simplifies the encoding but also reduces the burden on SMT solvers by avoiding the overhead of quantifiers that would be necessary when encoding functions as relations. Quantifier elimination and instantiation are among the most significant challenges in SMT solving, motivating the choice of avoiding them whenever feasible.

Functions in B, being rooted in set theory, are fundamentally represented as relations rather than natively as found in similar languages such as TLA$^+$ [17] or Alloy [14]. In particular, they are characterized as binary relations satisfying some property, as defined below.

**Definition 2 (Partial function).** *Let $f$ be a binary relation over two arbitrary sets $S$ and $T$, i.e. $f \subseteq S \times T$. $f$ verifies the functional property if:*

$$\forall x \in S, y \in T, z \in T, (x \mapsto y \in f \land x \mapsto z \in f) \Rightarrow y = z$$

*In this case, $f$ is said to be a partial function, denoted by $f \in S \nrightarrow T$.*

*Remark 5.* In the case of a functional relation $f$, the notation $x \mapsto y \in f$ should be understood as $f(x) = y$. The definition above simply states that each element from the domain of $f$ is mapped to at most one element in its codomain.

*Example 10.* The overhead of encoding functions as relations can be readily grasped by examining the encoding of the B expression `FINITE(S)`, where `S` is an arbitrary set. For improved readability, the superscript annotations of the B operators are omitted in this example only and all expressions below are written in B syntax.

By definition of the `FINITE` predicate in B, the expression expands to:

$$\forall a \in \mathbb{Z} \cdot (\exists b, f \in \mathbb{Z} \times (\mathtt{S} \leftrightarrow \mathbb{Z}) \cdot f \in \mathtt{S} \rightarrowtail a..b)$$

This expresses that the set `S` can be injectively mapped onto a finite interval of integers, i.e., it can be enumerated. Unfolding all definitions yields the following:

$$\begin{aligned}
&\forall a \in \mathbb{Z} \cdot \exists b, f \in \mathbb{Z} \times (\mathtt{S} \leftrightarrow \mathbb{Z}) \cdot \\
&\qquad (\forall x, y \in \mathtt{S} \times \mathbb{Z} \cdot x \mapsto y \in f \Rightarrow a \leq y \land y \leq b) \qquad \land \\
&\qquad (\forall x, y, z \in \mathtt{S} \times \mathbb{Z} \times \mathbb{Z} \cdot x \mapsto y \in f \land x \mapsto z \in f \Rightarrow y = z) \qquad \land \\
&\qquad (\forall z \in \mathtt{S} \cdot \exists w \in \mathbb{Z} \cdot z \mapsto w \in f) \qquad \land \\
&\qquad (\forall x, y, z \in \mathtt{S} \times \mathbb{Z} \times \mathbb{Z} \cdot x \mapsto z \in f \land y \mapsto z \in f \Rightarrow x = y)
\end{aligned}$$

Encoding this expression with *ppTransSMT* yields the following SMT-LIB script, where $\tau$ is the SMT-LIB type of the elements in the set `S`:

```
(assert (forall ((a Int)) (exists ((b Int) (f (P (C τ Int)))) (and
  (forall ((x τ) (y Int)) (=> (mem2 x y f) (and (<= a y) (<= y b))))
  (forall ((x τ) (y Int) (z Int))
    (=> (and (mem2 x y f) (mem2 x y f)) (= y z)))
  (forall ((z τ)) (exists ((w Int)) (mem2 z w f)))
  (forall ((x τ) (y τ) (z Int))
    (=> (and (mem2 x z f) (mem2 y z f)) (= x y)))))))
```

It can be seen that the encoding obtained via a first-order reduction produces a large expression with many quantifiers. In particular, the variable $f$ bound under the existential quantifier has the SMT-LIB type `P (C τ Int)`, which is already a complex type and depends on $\tau$. □

With the arrow type, B functions can be directly encoded as SMT-LIB functions, thereby avoiding the need to rely on relations satisfying some property.

*Example 11.* Returning to the B expression `FINITE(S)` from Ex. 10, let $\tau$ denote an SMT-LIB type representing the type of the elements in the set `S`. By directly encoding the function $f$ as a function from $\tau$ to `Option Int`, the expression is encoded as follows in SMT-LIB, omitting the superscript annotations for readability:

$$\begin{aligned}\exists N : \texttt{Int}, f : \tau \to \texttt{Option Int} \cdot\ & (\forall x : \tau, (\neg f(x) = \texttt{none}) = \hat{S}(x)) \quad \wedge \\ & (\forall x : \tau, y : \tau, z : \texttt{Int} \cdot f(x) = \texttt{some}\ z \wedge f(y) = \texttt{some}\ z \Rightarrow x = y) \quad \wedge \\ & (\forall x : \tau \cdot \hat{S}(x) \Rightarrow 0 \leq \texttt{the}\ f(x) \wedge \texttt{the}\ f(x) < N)\end{aligned}$$

□

Ultimately, expressions involving operators such as `FINITE` or `CARD`, which rely on functions embedded within their definitions, can be encoded in a more direct and efficient manner.

This idea can be systematically extended to all B terms involving functions, rather than being restricted solely to those derived from fixed definitions. To ensure the soundness of the proof obligations generated by the encoding, it is essential to establish the conditions under which the encoding remains valid. This is safe in the particular case of invariants enforcing functional properties on terms. These properties are indeed easily broken in other contexts, as illustrated in the following example.

*Example 12.* Consider the following B operation:

```
op (x, y) =
PRE
  x : INTEGER & y : INTEGER      // x ∈ᴮ ℤ ∧ y ∈ᴮ ℤ
THEN
  f := f \/ {x |-> y}            // f := f ∪ᴮ {x ↦ᴮ y}
END
```

and assume `f` has been initialized to the partial function `{0 |-> 1, 1 |-> 2}`.

After a call to `op(2, 3)`, the value of `f` is `{0 |-> 1, 1 |-> 2, 2 |-> 3}`, which is still a partial function. After another call to `op(2, 4)` however, the value of `f` is `{0 |-> 1, 1 |-> 2, 2 |-> 3, 2 |-> 4}`, which is no longer a partial function, as it maps 2 to both 3 and 4. Encoding `f` as a function in SMT-LIB would be unsound.

Now, consider that the operation `op` occurs within the following B machine:

```
VARIABLES
  f
INVARIANT
  f : INTEGER +-> INTEGER          // f ∈ ℤ ⇸ ℤ
INITIALISATION
  f := {0 |-> 1, 1 |-> 2}          // f := {0 ↦ 1, 1 ↦ 2}
```

This machine requires `f` to be a partial function as an invariant. In this case, two proof obligations are generated, one for the initialization and one for the invariant preservation. Written using the syntax defined in Sec. 3.1, they are:

$$\{0 \mapsto^{\mathrm{B}} 1, 1 \mapsto^{\mathrm{B}} 2\} \in^{\mathrm{B}} \mathbb{Z} \nrightarrow^{\mathrm{B}} \mathbb{Z} \qquad \text{(init)}$$

$$f \in^{\mathrm{B}} \mathbb{Z} \nrightarrow^{\mathrm{B}} \mathbb{Z} \Rightarrow^{\mathrm{B}} \forall^{\mathrm{B}} x, y \in^{\mathrm{B}} \mathbb{Z} \times^{\mathrm{B}} \mathbb{Z} \cdot f \cup^{\mathrm{B}} \{x \mapsto^{\mathrm{B}} y\} \in^{\mathrm{B}} \mathbb{Z} \nrightarrow^{\mathrm{B}} \mathbb{Z} \qquad \text{(invariant)}$$

The first proof obligation is trivially satisfied. However, the second one is false: consider the counterexample $f := \{0 \mapsto^{\mathrm{B}} 1, 1 \mapsto^{\mathrm{B}} 2, 2 \mapsto^{\mathrm{B}} 3\}$, $x := 2$, and $y := 4$. Then $f \cup^{\mathrm{B}} \{x \mapsto^{\mathrm{B}} y\}$ is not a partial function.

As will be shown in the following section, the encoding of relation-related operations must vary depending on whether neither, one, or both operands are encoded as functions. In this example, the union $f \cup^{\mathrm{B}} \{x \mapsto^{\mathrm{B}} y\}$ of a partial function and a singleton has to be encoded.[2] Since no additional constraints are provided on variables $x$ and $y$, a safe choice is to encode it as a relation, based on the following consideration:

$$f \cup^{\mathrm{B}} \{x \mapsto^{\mathrm{B}} y\} =^{\mathrm{B}} \{p \in^{\mathrm{B}} \mathbb{Z} \times^{\mathrm{B}} \mathbb{Z} \mid p =^{\mathrm{B}} x \mapsto^{\mathrm{B}} y \vee^{\mathrm{B}} \exists^{\mathrm{B}} a \in^{\mathrm{B}} \mathbb{Z} \cdot p =^{\mathrm{B}} a \mapsto^{\mathrm{B}} f(a)\}$$

The SMT-LIB encoding of $f \cup^{\mathrm{B}} \{x \mapsto^{\mathrm{B}} y\}$ is then:

$$\lambda^{\mathrm{S}} p : \texttt{Pair Int Int},\ (\exists^{\mathrm{S}} a : \texttt{Int}, b : \texttt{Int},\ p =^{\mathrm{S}} \texttt{pair}\ a\ b \wedge^{\mathrm{S}} f(a) =^{\mathrm{S}} \texttt{some}\ b) \vee^{\mathrm{S}} p =^{\mathrm{S}} \texttt{pair}\ x\ y$$

Using this expression in the encoding of the proof obligation, stating that this relation is a partial function, would then result in an SMT-LIB expression that can be proven to be false; however the semantics of the B expression would be preserved by the encoding. □

[2] The fact that the proof obligation encodes the fact that $f \cup^{\mathrm{B}} \{x \mapsto^{\mathrm{B}} y\}$ is a partial function is a semantic property of the expression, and has nothing to do with how the encoding is done.

### 5.4 Encoding rules

With these foundational considerations established, the encoding rules can be defined. To establish a sound encoding, it is imperative to formalize the source and target languages, thereby enabling rigorous reasoning about their structure and properties. While the scope of this paper is confined to encoding B proof obligations in SMT-LIB, it is noteworthy that a POG file encapsulates typing information that can be checked at runtime. Soundness of the encoding is therefore contingent upon correctness of the typing and well-definedness of the encoding rules, among other factors.

**Literals, Arithmetic and Boolean operations** Literal values are encoded directly, associating B types with SMT-LIB types via the mapping $\xi$. Since arithmetic and Boolean operations are natively supported in both B and SMT-LIB, they can be directly encoded, namely $\odot^{\mathrm{B}}$ is encoded as $\odot^{\mathrm{S}}$, where $\odot$ is one of the operators $+, -, *, =, \wedge, \vee, \Rightarrow$ and $\neg$. Binders are also encoded directly, the only difference being that the SMT-LIB types of the bound variables must be computed from the domain of the B quantifier.

**Set operations** Maplets are encoded as pairs, i.e. $\mapsto^{\mathrm{B}}$ is encoded as `pair`, and membership is encoded as function application, benefiting from the encoding of sets as characteristic predicates. Powerset is encoded as the set of all subsets. Cartesian product is encoded with pairs as follows: let $S$ and $T$ be B terms such that $\Gamma \vdash^{\mathrm{B}} S : \texttt{set}\ \alpha$ and $\Gamma \vdash^{\mathrm{B}} T : \texttt{set}\ \beta$ for some context $\Gamma$ and types $\alpha$ and $\beta$. Let $\hat{S}$ and $\hat{T}$ be their respective encodings. The expression $S \times^{\mathrm{B}} T$ is encoded as:

$$\lambda^{\mathrm{S}} p\colon \texttt{Pair}\ \xi(\alpha)\ \xi(\beta), \hat{S}(\texttt{fst}\ p) \wedge^{\mathrm{S}} \hat{T}(\texttt{snd}\ p)$$

More complex operations like union are trickier to encode and are discussed later.

**Partial functions** Let $f, \mathcal{A}$ and $\mathcal{B}$ be B terms. The expression $f \in^{\mathrm{B}} \mathcal{A} \nrightarrow^{\mathrm{B}} \mathcal{B}$, assuming it is well-typed in the type system of B, is encoded in SMT-LIB as follows:

- the symbol $f$ is declared with the type $\alpha \to^{\mathrm{S}} \texttt{Option}\ \beta$,
- the following assertion is added:

$$(\forall^{\mathrm{S}} x\colon \alpha, f\ x \neq \texttt{none} \Rightarrow^{\mathrm{S}} \hat{\mathcal{A}}\ x)\ \wedge^{\mathrm{S}} (\forall^{\mathrm{S}} x\colon \alpha, f\ x \neq \texttt{none} \Rightarrow^{\mathrm{S}} \hat{\mathcal{B}}\ (\texttt{the}\ f\ x))$$

  where $\alpha$ (resp. $\beta$) is the SMT-LIB type corresponding to the B type of elements of $\mathcal{A}$ (resp. $\mathcal{B}$), and $\hat{\mathcal{A}}$ (resp. $\hat{\mathcal{B}}$) is the encoding of $\mathcal{A}$ (resp. $\mathcal{B}$) as a characteristic predicate.

This rule is, in fact, quite straightforward: the characterization of a function is determined by two components—its domain and codomain—such that each element in the domain is associated with exactly one corresponding element in the codomain.

*Example 13.* Returning to the B expression from Example 10, the SMT-LIB code obtained from encoding the expression `FINITE(S)` where `S` is a set of integers is as follows:

```
(declare-const S (-> Int Bool))
(assert (exists ((N Int) (f (-> Int (Option Int)))) (and
  (forall ((x Int) (y Int)) (=>
    (and (S x) (S y) (= (f x) (f y)))
    (= x y)))
  (forall ((x Int)) (= (not (= (f x) none)) (S x)))
  (forall ((x Int)) (=> (S x) (and (<= 0 (f x)) (<= (f x) N)))))))
```

□

**Union** As mentioned in Ex. 12, the main challenge stemming from juggling relations and functions is to ensure that operations dealing with relations are correctly handled in the case of functions.

Let $f, g$ be two B terms and $\tilde{f}, \tilde{g}$ their respective encoding in SMT-LIB such that $\Gamma \vdash^{\mathrm{S}} \tilde{f} : \alpha$ and $\Gamma \vdash^{\mathrm{S}} \tilde{g} : \beta$ for some context $\Gamma$ and SMT-LIB types $\alpha$ and $\beta$. The expression $f \cup^{\mathrm{B}} g$ is encoded as follows, distinguishing between the following cases on $\alpha$ and $\beta$:

- if $\alpha = \beta = \sigma \to^{\mathrm{S}} \texttt{Bool}$,
$$\lambda^{\mathrm{S}} x : \sigma,\ \tilde{f}(x) \vee^{\mathrm{S}} \tilde{g}(x)$$
- if $\alpha = \beta = \sigma \to^{\mathrm{S}} \texttt{Option}\ \gamma$,
$$\lambda^{\mathrm{S}} p : \texttt{Pair}\ \sigma\ \gamma,\ (\tilde{f}(\texttt{fst}\ p) =^{\mathrm{S}} \texttt{some}\ (\texttt{snd}\ p)) \vee^{\mathrm{S}} (\tilde{g}(\texttt{fst}\ p) =^{\mathrm{S}} \texttt{some}\ (\texttt{snd}\ p))$$
- if $\alpha = \texttt{Pair}\ \sigma\ \gamma \to^{\mathrm{S}} \texttt{Bool}$ and $\beta = \sigma \to^{\mathrm{S}} \texttt{Option}\ \gamma$ (the symmetric case is omitted),
$$\lambda^{\mathrm{S}} p : \texttt{Pair}\ \sigma\ \gamma,\ \tilde{f}(p) \vee^{\mathrm{S}} (\tilde{g}(\texttt{fst}\ p) =^{\mathrm{S}} \texttt{some}\ (\texttt{snd}\ p))$$

The rules described above cover the subset of the B language formalized in Sec. 3.1 while ensuring type correctness of the SMT-LIB translation under the assumption that the B input is well typed. While additional B operators can be encoded following similar principles by relying on the base constructs we have defined, new encoding rules may be established to handle more complex constructs in a more efficient manner.

### 5.5 Towards soundness

Establishing soundness—i.e., proving that the SMT-LIB encoding preserves the semantics of B proof obligations—is a non-trivial task and remains future work. However, the key ideas of the proof are outlined, without formally defining semantics, for the more intricate rules: partial functions and unions. The non-trivial aspect of these rules is that the SMT-LIB types must be correctly computed from the B types. The notations used in the rules detailed in Sec. 5.4 are kept.

**Partial functions** The idea is to reason by induction on B terms. Assuming that $f \in^{\mathrm{B}} \mathcal{A} \nrightarrow^{\mathrm{B}} \mathcal{B}$ is well-typed in B, there exists a context $\Gamma$ such that:

$$\Gamma \vdash^{\mathrm{B}} f \in^{\mathrm{B}} \mathcal{A} \nrightarrow^{\mathrm{B}} \mathcal{B} : \texttt{bool}$$

By definition of the typing rules, this implies that there exist two B types $\alpha'$ and $\beta'$ such that:

$$\Gamma \vdash^{\mathrm{B}} f : \texttt{set}\ (\alpha' \times^{\mathrm{B}} \beta') \quad \text{and} \quad \Gamma \vdash^{\mathrm{B}} \mathcal{A} : \texttt{set}\ \alpha' \quad \text{and} \quad \Gamma \vdash^{\mathrm{B}} \mathcal{B} : \texttt{set}\ \beta'$$

Consequently, $\mathcal{A}$ (resp. $\mathcal{B}$) is encoded to $\hat{\mathcal{A}}$ (resp. $\hat{\mathcal{B}}$) via its characteristic predicate and the type $\alpha$ (resp. $\beta$) is shown to be equal to $\xi(\alpha')$ (resp. $\xi(\beta')$) where $\xi$ is the mapping embedding B types into SMT-LIB types defined in Sec. 5.2. This encoding rule is therefore well-defined. ■

**Union** The cases listed in the encoding rule are shown to be exhaustive. Assuming that $\Gamma \vdash^{\mathrm{B}} f \cup^{\mathrm{B}} g : \texttt{set}\ \tau'$, for some context $\Gamma$ and B type $\tau'$, the typing rules of B imply that:

$$\Gamma \vdash^{\mathrm{B}} f : \texttt{set}\ \tau', \quad \Gamma \vdash^{\mathrm{B}} g : \texttt{set}\ \tau'$$

If $\tau' = \sigma' \times^{\mathrm{S}} \gamma'$, then $f$ and $g$ can be encoded either as functions or relations, which constitutes the different cases. We only consider the case where only one operand is encoded as a function—suppose it is $f$—as the other cases are similar. By induction, the types $\alpha$ and $\beta$ of $\tilde{f}$ and $\tilde{g}$ are shown to be such that:

$$\alpha = \sigma \to^{\mathrm{S}} \texttt{Option}\ \gamma \quad \text{and} \quad \beta = \texttt{Pair}\ \sigma\ \gamma \to^{\mathrm{S}} \texttt{Bool} \quad \text{for some types } \sigma \text{ and } \gamma$$

which are shown to be equal to $\xi(\sigma')$ and $\xi(\gamma')$ respectively. ■

## 6 Evaluation

Although our encoding currently covers only a limited subset of the B language, it can already be tested against real-world B proof obligations. An executable version of the encoding, developed in Lean 4.15.0, is available at [22] with instructions and example proof obligations. A publicly available dataset of B proof obligations [11] from industrial applications was used to compare our encoding to *ppTransSMT*.

For the evaluation, 133 POG files that fall within the subset of the syntax that can be handled by our encoding were selected from the dataset and encoded into SMT-LIB using both our encoding and *ppTransSMT*, yielding a total of 2,195 proof obligations. These proof obligations were processed using cvc5 [5], with model-based quantifier instantiation enabled and a timeout of 3 seconds per proof obligation. Table 1 indicates that, on average, cvc5 discharges proof obligations encoded using our approach twice as fast as those encoded with *ppTransSMT*, likely due to the more direct encoding of functions. As shown in Table 2, the

| | Time (s) | Time / PO (ms) |
|---|---|---|
| **PP** | 296 | 135 |
| **HO** | 116 | 53 |

**Table 1.** Total time taken by cvc5 to solve the proof obligations encoded using *ppTransSMT* (PP) and our encoding (HO).

| | $(u, u)$ | $(r, u)$ | $(u, r)$ |
|---|---|---|---|
| #PO | 22 | 14 | 28 |

**Table 2.** Results of the proof obligations on both encodings, where $r$ denotes either `sat` or `unsat` and $u$ denotes `unknown`.

solver exhibits consistent behavior across both encodings, except for 64 proof obligations. Among these, 28 are successfully solved with our encoding but not with *ppTransSMT*, which is a promising result. However, 14 proof obligations are solved by *ppTransSMT* but not by our encoding, suggesting that our approach may still overwhelm the solver in certain cases, especially when dealing with complex lambda expressions that cannot be reduced. Finally, 22 proof obligations remain unsolved by both encodings, indicating that further work is needed to address these challenges. Inspection of these proof obligations reveals that they are large and complex, involving multiple nested quantifiers that are difficult to simplify in either encoding. It could be beneficial to consider splitting these proof obligations into smaller, more manageable parts and it requires a good understanding of the internal workings of the solvers.

## 7 Conclusion

This paper introduces an encoding of B proof obligations into SMT-LIB, laying the groundwork for formal verification. The encoding rules are carefully designed to guarantee soundness and are evaluated against a dataset of real-world B proof obligations. The results are promising, demonstrating that our encoding is efficient. However, further work is required to resolve the remaining challenges.

The encoding is being implemented in the Lean theorem prover [19], along with a formalization of the B language and SMT-LIB semantics to ensure the correctness of the encoding and to facilitate the verification of further properties. Future work will focus on proving the soundness of the encoding in Lean, on extending the encoding to cover more constructs of the B language, and on improving the efficiency of the encoding, both in terms of solving time and number of discharged proof obligations.

*Acknowledgments.* I thank Stephan Merz, Sophie Tourret and Ghilain Bergeron for providing valuable feedback on my work, and David Déharbe for discussions on the B method and ppTransSMT.

## References

1. Abrial, J.R., Lee, M.K.O., Neilson, D.S., Scharbach, P.N., Sørensen, I.H.: The b-method. In: Prehn, S., Toetenel, H. (eds.) VDM '91 Formal Software Development Methods. pp. 398–405. Springer Berlin Heidelberg, Berlin, Heidelberg (1991)

2. Abrial, J.R.: The B-Book: Assigning Programs to Meanings. Cambridge University Press, USA (1996)
3. Abrial, J.R., Butler, M., Hallerstede, S., Hoang, T.S., Mehta, F., Voisin, L.: Rodin: an open toolset for modelling and reasoning in Event-B. International Journal on Software Tools for Technology Transfer **12**(6), 447–466 (Nov 2010). https://doi.org/10.1007/s10009-010-0145-y
4. Barbosa, H., Reynolds, A., Ouraoui, D.E., Tinelli, C., Barrett, C.: Extending SMT solvers to higher-order logic. In: Fontaine, P. (ed.) Proceedings of the 27th International Conference on Automated Deduction (CADE '19). Lecture Notes in Artificial Intelligence, vol. 11716, pp. 35–54. Springer (Aug 2019), http://theory.stanford.edu/~barrett/pubs/BREO+19.pdf, natal, Brazil
5. Barbosa, H., et al.: cvc5: A versatile and industrial-strength SMT solver. In: Fisman, D., Rosu, G. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I. Lecture Notes in Computer Science, vol. 13243, pp. 415–442. Springer (2022). https://doi.org/10.1007/978-3-030-99524-9_24, https://doi.org/10.1007/978-3-030-99524-9_24
6. Barrett, C., Fontaine, P., Tinelli, C.: The SMT-LIB Standard: Version 2.7. Tech. rep., Department of Computer Science, The University of Iowa (2025), available at www.SMT-LIB.org
7. Bruijn, de, N.: On the roles of types in mathematics, pp. 27–54. Cahiers du centre de logique, Academia-Erasme (1995)
8. Clearsy: Atelier B. https://www.atelierb.eu
9. Clearsy: PO XML Format Documentation (2023), https://www.atelierb.eu/wp-content/uploads/2023/10/pog-1.0.html
10. Déharbe, D.: Integration of SMT-solvers in B and Event-B development environments. Science of Computer Programming **78**, 310–326 (03 2013). https://doi.org/10.1016/j.scico.2011.03.007
11. Déharbe, D.: Proof obligations from the B formal method (September 2022). https://doi.org/10.5281/zenodo.7050797
12. Déharbe, D., Fontaine, P., Guyot, Y., Voisin, L.: Integrating SMT solvers in Rodin. Science of Computer Programming **94** (11 2014). https://doi.org/10.1016/j.scico.2014.04.012
13. Fraenkel, A.A., Bar-Hillel, Y.: Foundations of Set Theory. Elsevier, Atlantic Highlands, NJ, USA (1973)
14. Jackson, D.: Alloy: A logical modelling language. In: Bert, D., Bowen, J.P., King, S., Waldén, M. (eds.) ZB 2003: Formal Specification and Development in Z and B, Third International Conference of B and Z Users, Turku, Finland, June 4-6, 2003, Proceedings. Lecture Notes in Computer Science, vol. 2651, p. 1. Springer (2003). https://doi.org/10.1007/3-540-44880-2_1
15. Jacquel, M.: Automatisation des preuves pour la vérification des règles de l'Atelier B. Theses, Conservatoire national des arts et metiers - CNAM (Apr 2013), https://theses.hal.science/tel-00840484
16. Konrad, M.: Translation from Set-Theory to Predicate Calculus. Technical report, ETH Zurich (2012)
17. Lamport, L.: Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley Longman Publishing Co., Inc., USA (2002)
18. Mendelson, E., Fraenkel, A.A.: Axiomatic set theory. Journal of Symbolic Logic **24** (1958). https://doi.org/10.2307/2963801

19. de Moura, L., Kong, S., Avigad, J., van Doorn, F., von Raumer, J.: The Lean Theorem Prover (System Description). In: Felty, A.P., Middeldorp, A. (eds.) Automated Deduction - CADE-25. pp. 378–388. Springer International Publishing, Cham (2015)
20. Reynolds, A., Barbosa, H., Fontaine, P.: Revisiting Enumerative Instantiation, pp. 112–131 (04 2018). https://doi.org/10.1007/978-3-319-89963-3_7
21. Tourret, S., Fontaine, P., El Ouraoui, D., Barbosa, H.: Lifting congruence closure with free variables to $\lambda$-free higher-order logic via SAT encoding. In: SMT 2020 - 18th International Workshop on Satisfiability Modulo Theories. Online COVID-19, France (Jul 2020), https://hal.science/hal-03049088
22. Trélat, V.: Safely Encoding B Proof Obligations in SMT-LIB (Feb 2025). https://doi.org/10.5281/zenodo.14870543

# A POG format

Atelier B features a Proof Obligation Generator (POG), which produces proof obligations derived from well-formedness and consistency checks on a B component. They are stored in a separate file in the POG format, which is essentially an XML-based representation of B expressions. For example, the term `% x. (x : NATURAL | x)` represents the identity function over the natural numbers. The corresponding POG entry is as follows:

```
<Quantified_Exp type="%" typref="3">
  <Variables>
      <Id value="x" typref="1"/>
  </Variables>
  <Pred>
      <Exp_Comparison op=":">
          <Id value="x" typref="1" />
          <Id value="NATURAL" typref="2" />
      </Exp_Comparison>
  </Pred>
  <Body>
      <Id value="x" typref="1" />
  </Body>
</Quantified_Exp>
```

# B Additional B constructs

**Definition 3 (Additional syntactic constructs).** *Additional constructs are defined using the basic constructs of our B syntax. Let $P$ and $Q$ be two B terms. The following constructs are defined:*

$$
\begin{aligned}
P \vee^B Q &:= \neg^B(\neg^B P \wedge^B \neg^B Q) && \text{(disjunction)}\\
P \Rightarrow^B Q &:= \neg^B P \vee^B Q && \text{(implication)}\\
P \Leftrightarrow^B Q &:= (P \Rightarrow^B Q) \wedge^B (Q \Rightarrow^B P) && \text{(equivalence)}\\
P \neq^B Q &:= \neg^B(P =^B Q) && \text{(disequality)}
\end{aligned}
$$

*Let $S$ and $T$ be two B terms. We define the following syntactic macros, for any term $f$:*

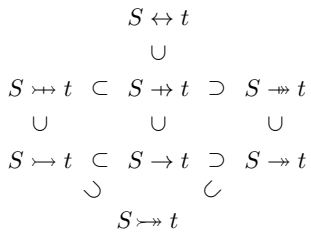
$$
\begin{aligned}
\texttt{inj}\ f &:= \forall^B x, y \in^B S \times^B S \cdot f(x) = f(y) \Rightarrow^B x =^B y\\
\texttt{surj}\ f &:= \forall^B y \in^B T \cdot^B \exists^B x \in^B S \cdot f(x) =^B y\\
\texttt{dom}\ f &:= \{x \in^B S \mid \exists^B y \in^B T \cdot x \mapsto^B y \in^B f\}\\
\texttt{ran}\ f &:= \{y \in^B T \mid \exists^B x \in^B S \cdot x \mapsto^B y \in^B f\}\\
\texttt{finite}\ S &:= \exists^B N \in^B \mathbb{Z} \cdot^B \exists^B \zeta \in^B S \rightarrowtail^B \mathbb{Z} \cdot \forall^B x \in^B S \cdot 0 \leq^B \zeta(x) \wedge^B \zeta(x) \leq^B N
\end{aligned}
$$

*Then, the following constructs are defined:*

$$
\begin{aligned}
f \in^B S \to^B T &:= f \in^B S \pfun^B T \wedge^B \mathtt{dom}\ f = S \\
f \in^B S \rightarrowtail^B T &:= f \in^B S \pfun^B T \wedge^B \mathtt{inj}\ f \\
f \in^B S \twoheadrightarrow^B T &:= f \in^B S \pfun^B T \wedge^B \mathtt{surj}\ f \\
f \in^B S \tfun^B T &:= f \in^B S \to^B T \wedge^B \mathtt{surj}\ f \\
f \in^B S \rightarrowtail\!\!\!\!\twoheadrightarrow^B T &:= f \in^B S \twoheadrightarrow^B T \wedge^B \mathtt{inj}\ f \\
f \in^B S \tbij^B T &:= f \in^B S \tfun^B T \wedge^B \mathtt{inj}\ f
\end{aligned}
$$

**Definition 4 (Classes of functions).** *Let $S$ and $T$ be two $B$ sets. The different classes of functions available in $B$ are listed in the table below, and the relation of inclusion between them is illustrated by the following diagram:*

| **Property** | **Math.** | **ASCII** |
|---|---|---|
| *Partial* | $S \pfun T$ | `+->` |
| *Total* | $S \to T$ | `-->` |
| *Partial and injective* | $S \rightarrowtail T$ | `>+>` |
| *Total and injective* | $S \tinj T$ | `>->` |
| *Partial and surjective* | $S \twoheadrightarrow T$ | `+->>` |
| *Total and surjective* | $S \tfun T$ | `-->>` |
| *Total and bijective* | $S \tbij T$ | `>->>` |




## C Typing rules for B

$$\frac{\Gamma(v) = \tau}{\Gamma \vdash^B v : \tau}\ (\text{var}) \qquad \frac{}{\Gamma \vdash^B n : \mathtt{int}}\ (\text{int}) \qquad \frac{}{\Gamma \vdash^B b : \mathtt{bool}}\ (\text{bool})$$

$$\frac{}{\Gamma \vdash^B \mathbb{Z} : \mathtt{set\ int}}\ (\text{bool}) \qquad \frac{}{\Gamma \vdash^B \mathbb{B} : \mathtt{set\ bool}}\ (\text{bool}) \qquad \frac{\Gamma \vdash^B x : \mathtt{bool}}{\Gamma \vdash^B \neg^B x : \mathtt{bool}}\ (\text{not})$$

$$\frac{\Gamma \vdash^B x : \mathtt{int} \qquad \Gamma \vdash^B y : \mathtt{int}}{\Gamma \vdash^B x \odot y : \mathtt{int}}\ (\text{add, sub, mul}) \quad \text{where } \odot \text{ is one of } +^B, -^B, *^B$$

$$\frac{\Gamma \vdash^B x : \mathtt{bool} \qquad \Gamma \vdash^B y : \mathtt{bool}}{\Gamma \vdash^B x \wedge^B y : \mathtt{bool}}\ (\text{and}) \qquad \frac{\Gamma \vdash^B x : \alpha \qquad \Gamma \vdash^B y : \alpha}{\Gamma \vdash^B x =^B y : \mathtt{bool}}\ (\text{eq})$$

$$\frac{\Gamma \vdash^B x : \mathtt{int} \qquad \Gamma \vdash^B y : \mathtt{int}}{\Gamma \vdash^B x \leq^B y : \mathtt{bool}}\ (\text{le}) \qquad \frac{\Gamma \vdash^B x : \alpha \qquad \Gamma \vdash^B y : \beta}{\Gamma \vdash^B x \mapsto^B y : \alpha \times^B \beta}\ (\text{maplet})$$

$$\frac{\Gamma \vdash^B S : \mathtt{set\ int}}{\Gamma \vdash^B \min\ S : \mathtt{int}}\ (\text{min}) \qquad \frac{\Gamma \vdash^B S : \mathtt{set\ int}}{\Gamma \vdash^B \max\ S : \mathtt{int}}\ (\text{max}) \qquad \frac{\Gamma \vdash^B S : \mathtt{set}\ \alpha}{\Gamma \vdash^B |S|^B : \mathtt{int}}\ (\text{card})$$

$$\frac{\Gamma \vdash^B S : \mathtt{set}(\alpha)}{\Gamma \vdash^B \mathcal{P}^B(S) : \mathtt{set}(\mathtt{set}(\alpha))}\ (\text{pow}) \qquad \frac{\Gamma \vdash^B A : \mathtt{set}(\alpha) \qquad \Gamma \vdash^B B : \mathtt{set}(\beta)}{\Gamma \vdash^B A \pfun^B B : \mathtt{set}(\mathtt{set}(\alpha \times^B \beta))}\ (\text{pfun})$$

$$\frac{\Gamma \vdash^B x : \mathtt{set}\ \alpha \qquad \Gamma \vdash^B y : \mathtt{set}\ \alpha}{\Gamma \vdash^B x \odot y : \mathtt{set}\ \alpha}\ (\text{union, inter}) \quad \text{where } \odot \text{ is one of } \cup^B, \cap^B$$

$$\frac{\Gamma \vdash^{\mathrm{B}} S : \mathtt{set}(\alpha) \qquad \Gamma \vdash^{\mathrm{B}} T : \mathtt{set}(\beta)}{\Gamma \vdash^{\mathrm{B}} S \times^{\mathrm{B}} T : \mathtt{set}(\alpha \times^{\mathrm{B}} \beta)}\ (\text{cprod})$$

$$\frac{\Gamma \vdash^{\mathrm{B}} x : \alpha \qquad \Gamma \vdash^{\mathrm{B}} S : \mathtt{set}(\alpha)}{\Gamma \vdash^{\mathrm{B}} x \in^{\mathrm{B}} S : \mathtt{bool}}\ (\text{mem})$$

$$\frac{\Gamma \vdash^{\mathrm{B}} x : \alpha \qquad \Gamma \vdash^{\mathrm{B}} f : \mathtt{set}(\alpha \times^{\mathrm{B}} \beta)}{\Gamma \vdash^{\mathrm{B}} f(x) : \beta}\ (\text{app})$$

$$\frac{\forall i \in [\![1, n]\!],\ \Gamma \vdash^{\mathrm{B}} D_i : \mathtt{set}(\alpha_i) \qquad \Gamma, v_1 : \alpha_1, \ldots, v_n : \alpha_n \vdash^{\mathrm{B}} P : \mathtt{bool}}{\Gamma \vdash^{\mathrm{B}} \{v_1, ..., v_n \in^{\mathrm{B}} D_1 \times^{\mathrm{B}} \ldots \times^{\mathrm{B}} D_n \mid P\} : \mathtt{set}(\alpha_1 \times^{\mathrm{B}} ... \times^{\mathrm{B}} \alpha_n)}\ (\text{collect})$$

$$\frac{\forall i \in [\![1, n]\!],\ \Gamma \vdash^{\mathrm{B}} D_i : \mathtt{set}(\alpha_i) \qquad \Gamma, v_1 : \alpha_1, \ldots, v_n : \alpha_n \vdash^{\mathrm{B}} t : \beta}{\Gamma \vdash^{\mathrm{B}} \lambda^{\mathrm{B}} v_1, ..., v_n.(v_1 \mapsto^{\mathrm{B}} ... \mapsto^{\mathrm{B}} v_n \in^{\mathrm{B}} \mathcal{D} \mid t) : \mathtt{set}(\alpha_1 \times^{\mathrm{B}} ... \times^{\mathrm{B}} \alpha_n \times^{\mathrm{B}} \beta)}\ (\text{lambda})$$
where $\mathcal{D} = D_1 \times^{\mathrm{B}} \ldots \times^{\mathrm{B}} D_n$

# D Typing rules for SMT-LIB

$$\frac{\Gamma(v) = \tau}{\Gamma \vdash^{\mathrm{S}} v : \tau}\ (\text{var}) \qquad \frac{}{\Gamma \vdash^{\mathrm{S}} n : \mathtt{int}}\ (\text{int}) \qquad \frac{}{\Gamma \vdash^{\mathrm{S}} b : \mathtt{bool}}\ (\text{bool})$$

$$\frac{\Gamma \vdash^{\mathrm{S}} x : \mathtt{int} \qquad \Gamma \vdash^{\mathrm{S}} y : \mathtt{int}}{\Gamma \vdash^{\mathrm{S}} x \leq^{\mathrm{S}} y : \mathtt{int}}\ (\text{le}) \qquad \frac{\Gamma \vdash^{\mathrm{S}} x : \mathtt{bool}}{\Gamma \vdash^{\mathrm{S}} \neg^{\mathrm{S}} x : \mathtt{bool}}\ (\text{not})$$

$$\frac{\Gamma \vdash^{\mathrm{S}} x : \alpha}{\Gamma \vdash^{\mathrm{S}} \mathtt{some}\ x : \mathtt{Option}\ \alpha}\ (\text{some}) \qquad \frac{}{\Gamma \vdash^{\mathrm{S}} \mathtt{as\ none}\ \alpha : \mathtt{Option}\ \alpha}\ (\text{none})$$

$$\frac{\Gamma \vdash^{\mathrm{S}} x : \alpha \qquad \Gamma \vdash^{\mathrm{S}} y : \beta}{\Gamma \vdash^{\mathrm{S}} \mathtt{pair}\ x\ y : \mathtt{Pair}\ \alpha\ \beta}\ (\text{pair}) \qquad \frac{\Gamma \vdash^{\mathrm{S}} x : \alpha \qquad \Gamma \vdash^{\mathrm{S}} y : \alpha}{\Gamma \vdash^{\mathrm{S}} x =^{\mathrm{S}} y : \mathtt{bool}}\ (\text{eq})$$

$$\frac{\Gamma \vdash^{\mathrm{S}} c : \mathtt{bool} \qquad \Gamma \vdash^{\mathrm{S}} x : \alpha \qquad \Gamma \vdash^{\mathrm{S}} y : \alpha}{\Gamma \vdash^{\mathrm{S}} \text{if } c \text{ then } x \text{ else } y : \alpha}\ (\text{ite})$$

$$\frac{\Gamma \vdash^{\mathrm{S}} f : \alpha \to^{\mathrm{S}} \beta \qquad \Gamma \vdash^{\mathrm{S}} x : \alpha}{\Gamma \vdash^{\mathrm{S}} f(x) : \beta}\ (\text{app})$$

$$\frac{\Gamma \vdash^{\mathrm{S}} x : \mathtt{int} \qquad \Gamma \vdash^{\mathrm{S}} y : \mathtt{int}}{\Gamma \vdash^{\mathrm{S}} x \odot y : \mathtt{int}}\ (\text{add, sub, mul}) \quad \text{where } \odot \text{ is one of } +^{\mathrm{S}}, -^{\mathrm{S}}, *^{\mathrm{S}}$$

$$\frac{\Gamma \vdash^{\mathrm{S}} x : \mathtt{bool} \qquad \Gamma \vdash^{\mathrm{S}} y : \mathtt{bool}}{\Gamma \vdash^{\mathrm{S}} x \odot y : \mathtt{bool}}\ (\text{and, or, imp}) \quad \text{where } \odot \text{ is one of } \wedge^{\mathrm{S}}, \vee^{\mathrm{S}}, \Rightarrow^{\mathrm{S}}$$

$$\frac{\Gamma, v_1 : \alpha_1, \ldots, v_n : \alpha_n \vdash^{\mathrm{S}} P : \mathtt{bool}}{\Gamma \vdash^{\mathrm{S}} \mathcal{Q}\, v_1 : \alpha_1, \ldots, v_n : \alpha_n.\ P : \mathtt{bool}}\ (\text{forall, exists}) \quad \text{where } \mathcal{Q} \text{ is one of } \forall^{\mathrm{S}}, \exists^{\mathrm{S}}$$

$$\frac{\Gamma, v_1 : \alpha_1, \ldots, v_n : \alpha_n \vdash^{\mathrm{S}} t : \beta}{\Gamma \vdash^{\mathrm{S}} \lambda^{\mathrm{S}} v_1 : \alpha_1, \ldots, v_n : \alpha_n.\ t : \alpha_1 \to^{\mathrm{S}} \ldots \to^{\mathrm{S}} \alpha_n \to^{\mathrm{S}} \beta}\ (\text{lambda})$$