Encoding TLA+ proof obligations safely for SMT

Rosalie Defourné

▶ **To cite this version:**

Rosalie Defourné. Encoding TLA+ proof obligations safely for SMT. Science of Computer Programming, 2025, Selected Papers From the Rigorous State-Based Methods 9th International Conference(ABZ 2023), 239 (103178), 10.1016/J.SCICO.2024.103178 . hal-04701040

HAL Id: hal-04701040
https://inria.hal.science/hal-04701040v1

Submitted on 18 Sep 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

Encoding TLA$^+$ Proof Obligations Safely for SMT

Rosalie Defourné

Université de Lorraine, CNRS, Inria, LORIA, Nancy, France
rosalie.defourne@inria.fr

Abstract. The TLA$^+$ Proof System (TLAPS) allows users to verify proofs with the support of automated theorem provers, including SMT solvers. To increase trust in TLAPS, we revisited the encoding of TLA$^+$ for SMT, whose implementation had become too complex. Our approach is based on a first-order axiomatization with E-matching patterns. The new encoding is available with TLAPS and achieves performances similar to the previous version, despite its simpler design.

Keywords: Automated Theorem Proving · SMT · TLA$^+$ · TLAPS

1 Introduction

TLA$^+$ is a specification language based on the Temporal Logic of Actions and Zermelo-Fraenkel set theory [18,19,32]. It is mostly used in industry for modelling distributed systems [27]. TLA$^+$ includes a syntax for proofs, which are handled by the TLA$^+$ Proof System (TLAPS) [9]. Given a TLA$^+$ specification with user-written proofs, TLAPS generates a number of *proof obligations* which are then sent to backend solvers. At this time, the backends available are Isabelle (through a custom formalization of TLA$^+$'s set theory) [28], Zenon [7], SMT solvers CVC4 [5], veriT [8] and Z3 [12], and finally the LS4 prover for temporal logic [31].

Sending proof obligations to a theorem prover requires encoding TLA$^+$ into the input logic of that prover. We argue that a good encoding should meet two requirements: *soundness* and *efficiency*. An unsound encoding would permit the derivation of faulty statements in TLAPS, rendering it useless. An inefficient encoding would translate easy proof obligations into harder ones, making it harder to work with the backend. Users will call TLAPS expecting proof obligations to be solved, so each failure means they have to spend time simplifying their own proofs (assuming they are correct).

In this paper, we focus on TLAPS's encoding for SMT solvers [25]. The pre-existing version of this encoding is based on an efficient preprocessing phase reducing TLA$^+$ formulas to formulas of first-order logic with uninterpreted symbols and integer arithmetic. The procedure is centered around a rewriting system, whose purpose is to eliminate as many TLA$^+$ primitives as possible. This approach faces some challenges: due to the untyped nature of the language,

some rewriting rules require side conditions to be verified; most of the time, expressions are not structured in such a way that rewriting can eliminate enough primitives, so the rewriting module gets stuck too early. To help the rewriting module progress, the encoding implements several auxiliary techniques. These techniques include the elimination of simple equalities through substitution, and a type synthesis mechanism assigning types to variables, including types for sets and functions.

The implementation of the SMT encoding is particularly complex, making it especially difficult to verify its soundness. Furthermore, proof obligations are heavily transformed by the preprocessing phase, which only increases the need for ensuring the soundness of the encoding. Most SMT-LIB files produced by the encoding cannot be exploited for debugging or proof reconstruction, because they are too different from the source proof obligations. In our experimental evaluation, we found that 11 % of proof obligations were reduced to trivial assertions that SMT solvers can discharge immediately. This is good from an efficiency perspective, but problematic from a safety perspective, because proof obligations are essentially solved by the encoding itself, whose implementation is not verified.

The present work introduces a new design for the SMT encoding, which has been implemented and made available in a newer version of TLAPS. Rather than attempting to simplify TLA$^+$ expressions, our encoding translates them faithfully using uninterpreted SMT symbols and encodes the semantics of TLA$^+$ as an axiomatic theory. We optimize this procedure with custom E-matching patterns, also known as "triggers" [15, 22, 26]. Triggers are heuristics that SMT solvers can use to find relevant instances for first-order formulas. They do not compromise soundness in any way.

This article is structured as follows. We first provide more background on TLA$^+$ and TLAPS (Section 2). Then we formalize the part of TLA$^+$ we support, and also SMT's multi-sorted first-order logic (MS-FOL) (Section 3). The next section describes the successive steps of our encoding, with justifications for its soundness (Section 4). This is followed by a description of our strategy for trigger selection, which covers TLA$^+$'s theory of sets, functions and arithmetic (Section 5). Finally, we present our evaluation, which demonstrates that our encoding performs as well as the previous one, and that our custom triggers impact performances positively (Section 6).

This article is an extended version of work published in [13]. We extend our previous work with more details about the semantics of TLA$^+$ and some finer points of the encoding, a fuller presentation of our selection strategy for SMT triggers, and an extended evaluation including new data about the impact of SMT triggers on solvers' performances.

MODULE *SimpleClock*

EXTENDS *Naturals*
VARIABLE *hour*

$$Init \triangleq hour \in 1 \,..\, 12$$

$$Increment \triangleq \land hour < 12 \\ \land hour' = hour + 1$$

$$Reset \triangleq \land hour = 12 \\ \land hour' = 1$$

$$Next \triangleq \lor Increment \\ \lor Reset$$

$$Spec \triangleq Init \land \Box[Next]_{hour}$$

Fig. 1. Specification of a Simple Clock (Specifying Systems)

2 Background

2.1 TLA$^+$

Similar to the B Method [1], TLA$^+$ is a language for modelling systems and their properties. The Temporal Logic of Actions (TLA) provides the logical basis for describing systems that evolve over discrete time. The "+" of TLA$^+$ refers to untyped set theory, seen as a collection of interpreted operators (symbols) that make the language more expressive.

TLA$^+$ specifications are organized into individual modules. A module consists mostly of definitions. As an example, we introduce the *SimpleClock* module of Figure 1, adapted from the TLA$^+$ reference manual [19]. This module specifies a system counting from 1 to 12 in a circular way.

The module starts with a line importing the *Naturals* module from the standard library of TLA$^+$, which contains symbols and definitions about natural numbers. The next line declares *hour* to be a *temporal* variable of the module. The rest is a series of definitions following a very common structure for TLA$^+$ specifications: a system is described by an initial state, then one or several possible actions. These elements are finally combined in one temporal formula *Spec* representing the system.

Expressions not referring to any temporal variable are called *constant*. Predicates that refer to at least one temporal variable are called *state predicates*, because they are evaluated relatively to some particular state of the system. *Init* is the only state predicate in our example. Using the prime operator, one can refer to the value of an expression in the *next* state. Predicates that depend on two successive states are called *transition predicates*, or *actions*. Predicates

Increment and *Reset* are the two actions that define our system. By defining the action *Next* as their disjunction, we specify that either *Increment* or *Reset* can occur between states. Note that TLA$^+$ features a convenient notation for conjunctions and disjunctions of arbitrary lengths: clauses prefixed by $\lor$ with the same level of indentation are part of the same disjunction—similarly for $\land$ and conjunctions.

The formula *Spec* is interpreted relatively to a *behavior*, which is an infinite succession of states. The notation $\Box[Next]_{hour}$ means that for every transition of the system, either *Next* is true, or the variable *hour* does not change. Transitions during which the system does not progress are called *stuttering steps*. Allowing these steps to occur makes it possible to combine modules into larger specifications, or specify modules as refinements of others.

TLA$^+$ includes a variety of operators from set theory and other areas of mathematics. Here, the set membership operator $\in$ is used in the definition of *Init*, which also features an operator for intervals of integers. The expression $1 \,..\, 12$ is equivalent to $\{x \in Int : 1 \leq x \land x \leq 12\}$. More operators from the theory of integer arithmetic are used in *Increment* and *Reset*. The meaning of these symbols is standard, however one must be aware that TLA$^+$ is an untyped language. For instance, *Increment* is true if $hour < 12$ is true in the current state, and $hour' = hour + 1$ in the next; but the definition does not assume *hour* to be an integer. *Increment* would also be true if $hour = \sqrt{2}$ and $hour' = \sqrt{2} + 1$, for example.

The principle underlying the semantics of TLA$^+$ is *underspecification* [16]. Taking again the example of $hour + 1$, if *hour* is not a number, then the expression's value is simply unknown. It is not undefined; it is just that the semantics does not specify its definition. It is still possible to assert some facts about the expression, but they are essentially tautologies. For instance, $\emptyset + 1$ is a nonsensical expression, but $\emptyset + 1 = \emptyset + 1$ is regardless a valid formula.

Even if *hour* cannot be specified as an integer, we can prove that *hour* is always an integer for every behavior satisfying *Spec*. This can be expressed using temporal logic with the formula

$$Spec \Rightarrow \Box(hour \in Int)$$

This formula expresses a property of the system that we can try to verify. Verification in TLA$^+$ is achieved through either model-checking (TLC [32], Apalache [17]) or proof using TLAPS [9].

2.2 TLAPS

TLA$^+$ includes a syntax for proofs. Figure 2 shows a module extending *SimpleClock* with a proof. The theorem states that $hour \in 1 \,..\, 12$ is an invariant of *Spec*—this is slightly stronger than $hour \in Int$ but still easy to prove.

A TLA$^+$ proof is either a list of relevant facts and definitions, or a sequence of intermediate steps ending in a QED step. Facts are introduced by the keyword BY and definitions by DEF. Intermediate steps must be justified with nested proofs.

```
MODULE SimpleClockProof

EXTENDS SimpleClock, Naturals, TLAPS

Inv ≜ hour ∈ 1 .. 12

THEOREM Spec ⇒ □Inv
PROOF
⟨1⟩1. Init ⇒ Inv
    BY DEF Init, Inv
⟨1⟩2. Next ∧ Inv ⇒ Inv′
    ⟨2⟩1. Increment ∧ Inv ⇒ Inv′
        BY DEF Increment, Inv
    ⟨2⟩2. Reset ∧ Inv ⇒ Inv′
        BY DEF Reset, Inv
    ⟨2⟩.QED
        BY ⟨2⟩1, ⟨2⟩2 DEF Next
⟨1⟩3. UNCHANGED hour ∧ Inv ⇒ Inv′
    BY DEF Inv
⟨1⟩.QED
    BY PTL, ⟨1⟩1, ⟨1⟩2, ⟨1⟩3 DEF Spec
```

Fig. 2. Proof of a Simple Invariant

Each step is introduced by a label $\langle n \rangle l$, where n indicates the current depth of the proof, and l is an optional name. Named steps can be invoked as facts in subsequent proofs of the same depth or deeper. Users may also indicate which backend solver TLAPS should use for a proof using special keywords. In this example, the PTL backend (Propositional Temporal Logic) is explicitly invoked for the last step.

TLAPS generates a *proof obligation* for every proof that starts with BY. Obligations are preprocessed and encoded for each backend solver. Preprocessing consists in inserting relevant facts in the context of the proof obligation, expanding symbols following DEF with their definitions, and reducing formulas to first-order ones without temporal logic (except for PTL). For instance, the proof obligation generated from step $\langle 2 \rangle 1$ can be expressed by the formula

$$hour < 12 \land hour' = hour + 1 \land hour \in 1 \,..\, 12 \Rightarrow hour' \in 1 \,..\, 12$$

To obtain this proof obligation, TLAPS replaces every occurrence of *Increment* and *Inv* with their definitions. Moreover, the expression Inv' is simplified by distributing the prime operator over the expanded expression. This simply results in the variable *hour* being replaced by *hour'*. It suffices to treat *hour'* as a new variable, distinct from *hour*, to eliminate all aspects of temporal logic in this proof obligation.

We call the fragment of TLA$^+$ that excludes temporal logic the *constant fragment*. A typical TLA$^+$ proof will always consist of proof obligations that can be reduced to the constant fragment, possibly ending with a step requiring temporal reasoning. For this reason, all of TLAPS's backend solvers except PTL only support the constant fragment. These backend solvers currently include Isabelle, Zenon, and the SMT solvers CVC4, veriT and Z3. In the case of Isabelle, the target is not Isabelle/HOL, but a custom logic Isabelle/TLA$^+$.

3 Formalizing MS-FOL and TLA$^+$

In this section, we introduce the two logics that correspond to the source and target of the encoding. We start with *multi-sorted first-order logic* with equality (MS-FOL), as it is more standard (Sub-section 3.1). Actually, we present a variant of MS-FOL that extends the language with second-order applications. This will be useful to describe proof obligations at some intermediate stages of the encoding. The constant fragment of TLA$^+$ is formalized next (Sub-section 3.2). Our formalism is based on the view that TLA$^+$ is a theory (collection of primitive symbols and axioms) on top of a variant of unsorted first-order logic. The base logic gives the semantics of Boolean connectives; the theory is largely based on ZFC set theory.

3.1 MS-FOL and Axiomatic Theories

Definition 1 (Types and Signatures). *We assume fixed some collection of sort symbols which include the Boolean sort o. Types are defined by the single-rule grammar*

$$\tau ::= \tau \times \cdots \times \tau \to s$$

where s is a sort symbol. If $s = o$, the type is called a predicate type. If the list of types on the left of the arrow is empty, the type is simply denoted s.

Let $\tau = \tau_1 \times \cdots \times \tau_n \to s$. We define the order $\mathrm{ord}(\tau)$ as 0 if $n = 0$, else $\max(\mathrm{ord}(\tau_i))_{1 \le i \le n} + 1$. If $\mathrm{ord}(\tau) \le 1$, τ is called a first-order type and n is called its arity.

A signature is a map Σ assigning types to function symbols.

Let ι be some sort symbol. Then $\iota \times \iota \to \iota$ is a first-order binary type. All the types we will consider are second-order at most. Second-order types are necessary to characterize some TLA$^+$ primitives, like set comprehension, which expects one first-order argument and one unary predicate argument; its type will be $\iota \times (\iota \to o) \to \iota$. TLA$^+$ also allows users to declare and define their own second-order symbols.

Definition 2 (Terms). *We assume fixed some collection of variable symbols with assigned sorts. Let Σ be a signature such that $\mathrm{ord}(\Sigma(F)) \le 2$ for all F in the domain of Σ. Terms are defined by the grammar*

$$t ::= x^s \mid F(f, \ldots, f) \mid t =_s t \mid \textsc{false} \mid t \Rightarrow t \mid \forall x^s : t \qquad \text{(Terms)}$$

$$f ::= t \mid F \mid \lambda x_1^{s_1}, \ldots, x_n^{s_n} : t \qquad \text{(Arguments)}$$

where x^s is a variable symbol of the sort s and F is in the domain of Σ.

In regular first-order logic, every argument is just a term, and all applications are of the form $F(t_1, \ldots, t_n)$. Terms that only include such applications are called first-order. In the general case, F may be a second-order symbol expecting some higher-order arguments. A higher-order argument is either a symbol or a lambda-term.

In Definition 2, we only allow higher-order arguments to be passed to second-order operators, because this reflects the actual user language of TLA$^+$. Higher-order arguments can only be passed to predefined operators, which can be primitive or user-defined. An example of a primitive second-order operator in TLA$^+$ is $\{x \in t : \phi\}$, which takes the higher-order argument $\lambda x : \phi$. As for user operators, one can for example define

$$\mathit{Twice}(F, a) \triangleq F(F(a))$$

One can then use an expression like $\mathit{Twice}(\lambda n : n + 1, 0)$ in TLA$^+$. If such an expression appears in a proof, then, depending on whether the user wants to expand the definition of *Twice* (using the keyword DEF), either the expression will be left as it is, or it will be replaced by its definition, and TLAPS will also apply the necessary beta-reductions: $(0 + 1) + 1$.

The next definition concerns well-typed terms and arguments. Before that, we introduce some useful terminology. In a term $\forall x^s : t$, the inner term t is called the *scope* of the quantifier $\forall x^s$. In this situation, we impose that no other quantifier binding x can occur in t. If a variable $x^{s'}$ occurs in t, the sorts must correspond, *i.e.* $s = s'$. These occurrences of variables are called *bound*. Any occurrence of a variable not bound by a quantifier is called *free*. Terms without free variables are called *ground*. All these rules also apply to the binding notation λx^s. A variable is bound by at most one $\forall$ or λ, and the sort annotations must correspond in all cases.

Definition 3 (Well-typed Terms). *We define two typing relations $t : s$ and $f : \tau$ by mutual recursion.*

- $x^s : s$
- *If* $\Sigma(F) = \tau_1 \times \cdots \times \tau_n \to s$ *and* $f_i : \tau_i$ *for all* i, *then* $F(f_1, \ldots, f_n) : s$
- *If* $t_1 : s$ *and* $t_2 : s$, *then* $t_1 =_s t_2 : o$
- FALSE $: o$
- *If* $t_1 : o$ *and* $t_2 : o$, *then* $t_1 \Rightarrow t_2 : o$
- *If* $t : o$, *then* $(\forall x^s : t) : o$
- $F : \Sigma(F)$
- *If* $t : s$, *then* $\lambda x_1^{s_1}, \ldots, x_n^{s_n} : t : s_1 \times \cdots \times s_n \to s$

If $t : s$, we say that t is well-typed with s. If $f : \tau$, we say that f is well-typed with τ. Terms well-typed with o are called formulas, and denoted by the letter ϕ.

Let *setst* be a symbol with type $\iota \times (\iota \to o) \to \iota$. For all $t : \iota$ and $\phi : o$ such that every occurrence of x in ϕ has the sort ι, the term $\mathit{setst}(t, \lambda x^\iota : \phi)$ is well-typed. If *setst* represents set comprehension, then the latter is a representation of the set $\{x \in t : \phi\}$, where x is bound in ϕ.

All the terms and arguments we will consider are well-typed. We may omit sort annotations on variables and equality symbols, since they can always be inferred from the context: in $\forall x^s : t$, all variables x in t have the sort s; in an equality $t_1 = t_2$, the annotation is the sort shared by t_1 and t_2.

Some Boolean connectives are not included in the grammar. We define them with the following notations:

$$\begin{aligned}
\neg\phi &\triangleq \phi \Rightarrow \text{FALSE} \\
\text{TRUE} &\triangleq \neg\text{FALSE} \\
t_1 \neq_s t_2 &\triangleq \neg(t_1 =_s t_2) \\
\phi_1 \vee \phi_2 &\triangleq (\neg\phi_1) \Rightarrow \phi_2 \\
\phi_1 \wedge \phi_2 &\triangleq \neg((\neg\phi_1) \vee (\neg\phi_2)) \\
\phi_1 \Leftrightarrow \phi_2 &\triangleq (\phi_1 \Rightarrow \phi_2) \wedge (\phi_2 \Rightarrow \phi_1) \\
\exists x^s : \phi &\triangleq \neg\forall x^s : \neg\phi
\end{aligned}$$

Let $C_1, \ldots, C_n, C$ be collections of elements. We denote $C_1 \times \cdots \times C_n \to C$ the collection of n-ary functions taking their i^{th} argument in C_i and returning elements of C.

Definition 4 (Interpretations). *Let D_o be the collection with two elements $\top$ and $\bot$. Let (D_s) be a family of collections indexed by sort symbols other than o. For all $\tau = \tau_1 \times \cdots \times \tau_n \to s$, we define the collection D_τ as $D_{\tau_1} \times \cdots \times D_{\tau_n} \to D_s$.*

Let Σ be a signature. A Σ-interpretation $\mathcal{I}$ consists of a family of domains D_s for each non-Boolean sort s and a family of functions $F^{\mathcal{I}}$ for each F in Σ such that $F^{\mathcal{I}}$ is an element of $D_{\Sigma(F)}$. A valuation is a function θ that assigns every variable x^s to some element $\theta(x^s)$ in D_s. For all valuations θ, variables x and elements v, we denote θ^x_v the valuation that reassigns x to v, assuming v is an element of the appropriate domain.

Given an interpretation $\mathcal{I}$ and a valuation θ, the interpretation of well-typed terms and arguments is defined recursively:

$$\begin{aligned}
[\![x^s]\!]^{\mathcal{I}}_\theta &\triangleq \theta(x^s) \\
[\![F(f_1, \ldots, f_n)]\!]^{\mathcal{I}}_\theta &\triangleq F^{\mathcal{I}}([\![f_1]\!]^{\mathcal{I}}_\theta, \ldots, [\![f_n]\!]^{\mathcal{I}}_\theta) \\
[\![t_1 =_s t_2]\!]^{\mathcal{I}}_\theta &\triangleq \top \text{ if } [\![t_1]\!]^{\mathcal{I}}_\theta = [\![t_2]\!]^{\mathcal{I}}_\theta, \text{ otherwise } \bot \\
[\![\text{FALSE}]\!]^{\mathcal{I}}_\theta &\triangleq \bot \\
[\![t_1 \Rightarrow t_2]\!]^{\mathcal{I}}_\theta &\triangleq \top \text{ if } [\![t_1]\!]^{\mathcal{I}}_\theta = \bot \text{ or } [\![t_2]\!]^{\mathcal{I}}_\theta = \top, \text{ otherwise } \bot \\
[\![\forall x^s : t]\!]^{\mathcal{I}}_\theta &\triangleq \top \text{ if } [\![t]\!]^{\mathcal{I}}_{\theta^x_v} = \top \text{ for all } v \text{ in } D_s, \text{ otherwise } \bot
\end{aligned}$$

For all v_1 in D_{s_1}, …, v_n in D_{s_n},

$$[\![F]\!]^{\mathcal{I}}_\theta(v_1, \ldots, v_n) \triangleq F^{\mathcal{I}}(v_1, \ldots, v_n) \quad \textit{where } \Sigma(F) = s_1 \times \cdots \times s_n \to s$$

$$[\![\lambda x^{s_1}_1, \ldots, x^{s_n}_n : e]\!]^{\mathcal{I}}_\theta(v_1, \ldots, v_n) \triangleq [\![e]\!]^{\mathcal{I}}_{\theta^{x_1,\ldots,x_n}_{v_1,\ldots,v_n}}$$

We admit without proof the following result: if $t : s$, and if $\theta(x^{s'})$ is an element of $D_{s'}$ for all $x_{s'}$, then $[\![t]\!]^{\mathcal{I}}_{\theta}$ is well-defined as an element of D_s. This could be proved by induction on the structure of well-typed terms and arguments.

We also admit that the interpretation of a term $[\![t]\!]^{\mathcal{I}}_{\theta}$ does not depend on the value of $\theta(x)$ if x has no free occurrence in t. If t is ground, we simply denote $[\![t]\!]^{\mathcal{I}}$ its interpretation.

Definition 5 (Satisfiability and Theories). *For all ground formulas ϕ and interpretations $\mathcal{I}$, if $[\![\phi]\!]^{\mathcal{I}} = \top$ we say that ϕ is* satisfied *by $\mathcal{I}$ and write $\mathcal{I} \models \phi$. Otherwise, we write $\mathcal{I} \not\models \phi$.*

We call theory *a pair (Σ, T) where Σ is a signature and T a collection of ground Σ-formulas. For all Σ-interpretation $\mathcal{I}$, we write $\mathcal{I} \models T$ if every formula in T is satisfied by $\mathcal{I}$. In that case, $\mathcal{I}$ is called a* model *of T.*

For all T and ϕ, we write $T \models \phi$ if every model of T satisfies ϕ. We write $\models \phi$ when T is empty, and say that ϕ is valid.

By restricting the logic to first-order terms and formulas, we obtain MS-FOL. Let us denote MS-FOL$^+$ the unrestricted version of the logic, which does permit second-order applications. Remark that MS-FOL$^+$ is still essentially first-order, since quantifiers $\forall$ can only bind variables and not functional symbols.

SMT's logic is based on MS-FOL. The SMT paradigm admits several *theories*, which are not theories in the sense of Definition 5. An SMT theory is a collection of sorts and functional symbols with predefined interpretations. We refer to those sorts and symbols as *interpreted*. For example, several theories include the sort int and arithmetic operators such as $+$. In these theories, the domain D_{int} is always the collection of integers, and $+$ is always interpreted as addition. Users are always free to declare their own sorts and functional symbols, which are then called *uninterpreted*.

3.2 Syntax and Semantics of TLA$^+$

Unsorted first-order logic is essentially MS-FOL with only one sort besides o. TLA$^+$ also removes the sort o, allowing for new combinations of Boolean connectives with other expressions. As a consequence, the semantics of Boolean connectives is extended.

It is customary in TLA$^+$ to avoid the word "function" when referring to logical symbols, because a TLA$^+$ function is already a kind of set with function-like properties. The word "operator" and the letter K are used instead. Even though TLA$^+$ is untyped, applications must be well-formed, so operators are still assigned types. Let ι be some sort symbol. A TLA$^+$ signature is a mapping Σ of operators K to types $\Sigma(K)$ of order 2 at most and containing only the sort ι.

Definition 6 (Expressions). *Let Σ be a TLA$^+$ signature. The syntax of TLA$^+$ expressions and arguments is defined by the following grammar:*

$$e ::= x \mid K(f, \ldots, f) \mid e = e \mid \text{FALSE} \mid e \Rightarrow e \mid \forall x : e \qquad \text{(Expressions)}$$

$$f ::= e \mid K \mid \lambda x, \ldots, x : e \qquad \text{(Arguments)}$$

where x is a variable symbol and K an operator symbol in the domain of Σ. Each argument is assigned an arity: an expression e has arity 0, the arity of a symbol K is the arity of $\Sigma(K)$, and the arity of $\lambda x_1, \ldots, x_n : e$ is n. A term $K(f_1, \ldots, f_n)$ is well-formed if $\Sigma(K) = \tau_1 \times \cdots \times \tau_n \to \iota$ and the arity of f_i matches the arity of the first-order type τ_i for all i. An expression or argument is well-formed if it only contains well-formed applications.

The logical connectives TRUE, $\neq$, $\neg$, $\wedge$, $\vee$, $\Leftrightarrow$, $\exists$ are defined as notations like in MS-FOL$^+$. The well-formedness condition on applications corresponds to MS-FOL$^+$ well-typedness in a context where ι is the only sort allowed.

Boolean connectives may be combined with other operators in unconventional ways. Assuming a signature containing the symbols of arithmetic, the following expressions are well-formed: $2 = \text{TRUE}$, $1 + (\forall x : x)$, $5 \Rightarrow \neg 6$. Many of the new expressions have no use, but it is possible to extend the semantics of MS-FOL$^+$ slightly to account for them.

Definition 7 (Interpretations). *A TLA$^+$ domain is a collection D that contains at least two distinct values denoted $\top^D$ and $\bot^D$. We fix $D_\iota = D$ and define D_τ for every TLA$^+$ type τ as before. Given a TLA$^+$ signature Σ, a Σ-interpretation $\mathcal{I}$ is defined as a domain D and a family of functions $K^\mathcal{I}$ satisfying the requirement that $K^\mathcal{I}$ is an element of $D_{\Sigma(K)}$. Valuations are defined as before.*

For all interpretations $\mathcal{I}$ and valuations θ, the interpretation of expressions and arguments is defined recursively:

$$
\begin{aligned}
[\![x]\!]^\mathcal{I}_\theta &\triangleq \theta(x) \\
[\![K(f_1, \ldots, f_n)]\!]^\mathcal{I}_\theta &\triangleq K^\mathcal{I}([\![f_1]\!]^\mathcal{I}_\theta, \ldots, [\![f_n]\!]^\mathcal{I}_\theta) \\
[\![e_1 = e_2]\!]^\mathcal{I}_\theta &\triangleq \top^D \text{ if } [\![e_1]\!]^\mathcal{I}_\theta = [\![e_2]\!]^\mathcal{I}_\theta, \text{ otherwise } \bot^D \\
[\![\text{FALSE}]\!]^\mathcal{I}_\theta &\triangleq \bot^D \\
[\![e_1 \Rightarrow e_2]\!]^\mathcal{I}_\theta &\triangleq \top^D \text{ if } [\![e_1]\!]^\mathcal{I}_\theta \neq \top^D \text{ or } [\![e_2]\!]^\mathcal{I}_\theta = \top^D, \text{ otherwise } \bot^D \\
[\![\forall x : e]\!]^\mathcal{I}_\theta &\triangleq \top^D \text{ if } [\![e]\!]^\mathcal{I}_{\theta^x_v} = \top^D \text{ for all } v \text{ in } D, \text{ otherwise } \bot^D
\end{aligned}
$$

For all $v_1, \ldots, v_n$ in D,

$$
\begin{aligned}
[\![K]\!]^\mathcal{I}_\theta\,(v_1, \ldots, v_n) &\triangleq K^\mathcal{I}(v_1, \ldots, v_n) \\
[\![\lambda x_1, \ldots, x_n : e]\!]^\mathcal{I}_\theta\,(v_1, \ldots, v_n) &\triangleq [\![e]\!]^\mathcal{I}_{\theta^{x_1,\ldots,x_n}_{v_1,\ldots,v_n}}
\end{aligned}
$$

We admit that $[\![e]\!]^\mathcal{I}_\theta$ is an element of D if e is well-formed, and that $[\![f]\!]^\mathcal{I}_\theta$ is an element of $D^n \to D$ if f is well-formed with arity n. Bound and free variables are defined as usual. We admit that the interpretation of ground expressions does not depend on the valuation. The satisfaction relation is defined by $\mathcal{I} \models e$ iff $[\![e]\!]^\mathcal{I} = \top^D$. Note that TLA$^+$ does not admit a notion of "formula", so $\models$ is defined for every expression of the language.

A rule of thumb for understanding TLA$^+$ expressions that use Boolean connectives in unusual ways is to replace expressions e with $e = \text{TRUE}$ when e occurs

in a Boolean context. For instance, the interpretation of $5 \Rightarrow \neg 6$ is the same as $(5 = \text{TRUE}) \Rightarrow \neg(6 = \text{TRUE})$. The latter is clearly valid in the theory of arithmetic, because from the hypotheses $5 = \text{TRUE}$ and $6 = \text{TRUE}$, it would follow that $5 = 6$. Therefore, $5 \Rightarrow \neg 6$ is valid. Note that whether $5 = \text{TRUE}$ or $6 = \text{TRUE}$ holds is irrelevant, and not specified in TLA$^+$. One can verify that all the usual tautologies of first-order logic hold under this semantics; for instance $e \vee \neg e$ is valid for all e, and $\forall x : x \vee \neg x$ is valid as well.

3.3 Axiomatic Theory of TLA$^+$

The constant fragment of TLA$^+$ is defined as a theory on top of the base logic described in the previous section. The theory consists of a signature and several axioms including the axioms of set theory. Primitive operators are not given a fixed interpretation; rather, all models of the theory are considered equally legitimate.

We argue that the lack of a canonical interpretation for operators captures the underspecified semantics of the language. For instance, the expression $7 + \emptyset$ is nonsensical and its value is unspecified. But there are valid formulas that involve this expression, like $(7 + \emptyset) \in \{7 + \emptyset\}$. We simply consider this to be a consequence of the axiom defining the singleton set.

The rest of this section is an overview of the primitive operators and axioms of TLA$^+$, focussing on the choice operator and the theories of sets, functions, and integer arithmetic. The theories of tuples, records and sequences (lists) are presented briefly. Not included in this presentation are the axioms for bags (multisets) and reals. They are currently not supported by our encoding.

Hilbert's Choice Operator The signature includes an operator *choose* whose type is $(\iota \to \iota) \to \iota$. Most TLA$^+$ primitives admit a readable notation; for all e, the expression $\mathit{choose}(\lambda x : e)$ is usually denoted $\text{CHOOSE } x : e$.

The operator CHOOSE is Hilbert's choice operator, also called the ϵ operator. Its meaning is given by the following axiom:

$$\forall x : \big[P(x) \Rightarrow P(\text{CHOOSE } x : P(x))\big]$$

Since this axiom is parameterized by a unary operator P, it is actually an axiom schema. Intuitively, the axiom states that $\text{CHOOSE } x : P(x)$ is a witness for P, if such a witness exists. It is possible that no witness exist, in which case the expression is unspecified.

The operator is also specified by the following principle of determinacy, which holds for all unary P and Q:

$$\big[\forall x : P(x) \Leftrightarrow Q(x)\big] \Rightarrow (\text{CHOOSE } x : P(x)) = (\text{CHOOSE } x : Q(x))$$

This axiom scheme lets us rewrite expressions below CHOOSE as long as the expressions are equivalent.

Set Theory The signature of TLA$^+$ includes the operators $\in$ and $\subseteq$. Enumeration sets $\{e_1, \ldots, e_n\}$ have a distinct operator for every n. Set comprehension is written $\{x \in S : e\}$, and set replacement $\{e : x \in S\}$, where the variable x is bound in e in both cases. The ambiguous notation $\{x \in S : x \in T\}$ is an instance of set comprehension. UNION S is the reunion of all sets in S and SUBSET S is the set of all subsets of S. Finally, the operators $\cup$, $\cap$ and $\setminus$ are included.

The axioms themselves are standard and very close to those of ZFC. For example, here is the schema of set comprehension (for the parameter P):

$$\forall a, x : x \in \{y \in a : P(y)\} \Leftrightarrow x \in a \wedge P(x)$$

The axiom of choice does not need to be included, because it is easily proved using CHOOSE. The axiom of regularity is included, even though it is seldom used in practical proofs—users must invoke it explicitly when needed.

Theory of Functions Functions are defined axiomatically in TLA$^+$. This is an important difference with the B Method, in which every function f is identified with its underlying graph, *i.e.* $y = f(x)$ iff $(x, y) \in f$.

The axioms involve a special operator *isafcn*, which is not part of the user language. The class of functions is characterized by this operator. Every function f has a domain DOMAIN f, and for all x in the domain, $f[x]$ is the result of applying f to x. The function of domain S that maps x to $F(x)$ is denoted $[x \in S \mapsto F(x)]$. The set of functions from set S to set T is denoted $[S \to T]$.

Here are the four axioms (including three schemas) that specify these operators.

$$\forall a : \mathit{isafcn}([x \in a \mapsto F(x)])$$

$$\forall a : \text{DOMAIN } [x \in a \mapsto F(x)] = a$$

$$\forall a, x : x \in a \Rightarrow [y \in a \mapsto F(y)][x] = F(x)$$

$$\begin{aligned} \forall a, b, f : f \in [a \to b] \Leftrightarrow\ & \wedge \mathit{isafcn}(f) \\ & \wedge \text{DOMAIN } f = a \\ & \wedge (\forall x : x \in a \Rightarrow f[x] \in b) \end{aligned}$$

A fifth axiom is added to make functional extensionality hold between functions of the same domain:

$$\begin{aligned} \forall f, g : \big[\ & \wedge \mathit{isafcn}(f) \\ & \wedge \mathit{isafcn}(g) \\ & \wedge \text{DOMAIN } f = \text{DOMAIN } g \\ & \wedge (\forall x : x \in \text{DOMAIN } f \Rightarrow f[x] = g[x])\ \big] \\ \Rightarrow\ & f = g \end{aligned}$$

Integer Arithmetic The set of integers is denoted *Int* in TLA$^+$. The operators of integer arithmetic include numeric constants for all natural numbers, unary and binary substraction $-$, addition $+$, multiplication $*$, Euclidian division $\div$, remainder $\%$, and comparisons $\leq$, $<$, $\geq$, $>$. We assume *Int* to be a model of integer arithmetic, *i.e.* the theory includes axioms that specify all arithmetic operators on *Int*.

Some additional operators are specified by definition. The set *Nat* is defined as $\{z \in Int : z \geq 0\}$ and the set $x \mathinner{..} y$ is defined as $\{z \in Int : x \leq z \wedge z \leq y\}$.

We refer the reader to Section 18.4 of the reference book for details about the formalization of arithmetic in TLA$^+$ [19]. Briefly, the construction starts from a model of Peano's axioms for natural arithmetic, then goes to building a model of the reals that contains the naturals. The integers are defined last. The formalization is done this way to ensure that $Nat \subset Int \subset Real$ with operators overloaded appropriately—for instance, addition is denoted $+$ for each set.

Other Axioms TLA$^+$ defines strings as special kinds of functions. The string "`foo`" is the function of domain $1 \mathinner{..} 3$ mapping indexes to the respective characters 'f', 'o', 'o'. There are no operations on strings in TLA$^+$.

Tuples, records and sequences are also defined from functions. For instance, the ordered pair $\langle x, y \rangle$ is defined as the function that maps 1 to x and 2 to y. The product set $a \times b$ is the set of pairs $\langle x, y \rangle$ such that $x \in a$ and $y \in b$. The record $[foo \mapsto x, bar \mapsto y]$ is a function of domain {"`foo`", "`bar`"}. If $x \in a$ and $y \in b$ then $[foo \mapsto x, bar \mapsto y]$ is an element of the set $[foo : a, bar : b]$.

Finite lists are called sequences in TLA$^+$. A sequence of length n is a function of domain $1 \mathinner{..} n$. If s is such a sequence, then its length is denoted $Len(s)$. If all the elements of s are in a, then $s \in Seq(a)$. Because of the way operators are overloaded, every tuple is technically a sequence: if $x_1, \ldots, x_n \in a$, then $\langle x_1, \ldots, x_n \rangle \in Seq(a)$. The signature of TLA$^+$ includes several operators for manipulating sequences: *Head* and *Tail* for getting the head and tail of a sequence, *Append* for appending elements, *Cat* for concatenating sequences, *Subseq* for getting the subrange of a sequence as another sequence, and *Selectseq* for filtering elements through an arbitrary predicate.

4 Encoding TLA$^+$ for SMT

In this section, we detail the main steps of our encoding of TLA$^+$ into SMT's language, formalized as MS-FOL. We start with a short overview of the encoding (Sub-section 4.1). The design of the encoding is summed up in three steps, which are recovering formulas (Sub-section 4.2), axiom selection and insertion (Sub-section 4.3), and the elimination of second-order applications (Sub-section 4.4).

4.1 Overview

To give a brief overview of our encoding, we will use a concrete TLA$^+$ proof obligation to illustrate each step:

$$\begin{array}{ll} \textsc{assume} & \textsc{new}\ P(\_),\ \textsc{new}\ S, \\ & \textsc{new}\ c \in S, \\ & P(c) \\ \textsc{prove} & c \in \{x \in S : P(x)\} \end{array}$$

We use TLA$^+$'s sequent notation for proof obligations: the keyword ASSUME is followed by declarations and hypotheses, and PROVE is followed by the goal. A declaration is introduced by NEW. The bound declaration NEW $c \in S$ is just the declaration of c followed by the hypothesis $c \in S$. The declaration NEW $P(\_)$ introduces a unary operator.

First, to clarify the proof obligation's structure, let us indicate the type of every declared variable and remove some syntactic sugar. We also rewrite every application into the form $K(f_1, \ldots, f_n)$.

$$\begin{array}{ll} \textsc{assume} & \textsc{new}\ P(\_) : \iota \to \iota, \\ & \textsc{new}\ S : \iota, \\ & \textsc{new}\ c : \iota, \\ & mem(c, S), \\ & P(c) \\ \textsc{prove} & mem(c, setst(S, \lambda x : P(x))) \end{array}$$

The primitive operator *mem* has type $\iota \times \iota \to \iota$ and *setst* has type $\iota \times (\iota \to \iota) \to \iota$.

The first step of the encoding consists in correcting ambiguities between terms and formulas to recover the usual semantics for Boolean connectives (Subsection 4.2). The target language is MS-FOL$^+$. Only two sorts are used: the Boolean sort o, and the uninterpreted sort ι. The result of this step is the following proof obligation:

$$\begin{array}{ll} \textsc{assume} & \textsc{new}\ cast_o : o \to \iota, \\ & \textsc{new}\ P(\_) : \iota \to \iota, \\ & \textsc{new}\ S : \iota, \\ & \textsc{new}\ c : \iota, \\ & mem(c, S), \\ & P(c) =_\iota cast_o(\textsc{true}) \\ \textsc{prove} & mem(c, setst(S, \lambda x^\iota : P(x) =_\iota cast_o(\textsc{true}))) \end{array}$$

The transformation does not make assumptions about how declared variables must be used; this is why P's type is preserved, even if its intended use seems to be as a predicate. Exceptions are made for some TLA$^+$ primitives: the type

of *mem* is changed to $\iota \times \iota \to o$ and the type of *setst* is changed to $\iota \times (\iota \to o) \to \iota$. In order to convert between terms and formulas, the encoding introduces the operator $cast_o$. Most notably, any term t can be turned into a formula by rewriting it $t =_\iota cast_o(\text{TRUE})$.

The next step finds primitive operators in the proof obligation and inserts their declarations and axioms in the context (Sub-section 4.3). When axioms contain new primitives, the procedure repeats recursively. Typically, every primitive operator is defined by 1 to 3 axioms. Here, only *setst* is specified by an axiom:

$$\forall a^\iota, x^\iota : mem(x, setst(a, P)) \Leftrightarrow mem(x, a) \wedge P(x) \qquad \text{for all } P : \iota \to o$$

As this is an axiom schema, parameterized by a higher-order argument P, it is not possible to insert it in full in the final SMT problem.

Removing all second-order features from the proof obligation is the focus of the next step, which maps formulas of MS-FOL$^+$ to MS-FOL (Sub-section 4.4). Our procedure parses expressions in a bottom-up way to find second-order applications. For every application $K(f_1, \ldots, f_n)$ with at least one higher-order argument, a specialized first-order operator $K^\bullet$ is created. For this example, $setst^\bullet$ is a specialized version of *setst* for the parameter $\lambda x^\iota : P(x) =_\iota cast_o(\text{TRUE})$. The appropriate instance of *setst*'s axiom schema is selected and inserted in the context. The proof obligation after this step is:

ASSUME NEW $cast_o : o \to \iota$,
NEW $mem : \iota \times \iota \to o$,
NEW $setst^\bullet : \iota \to \iota$,
$\ldots$
$mem(c, S)$,
$P(c) =_\iota cast_o(\text{TRUE})$
$\forall a^\iota, x^\iota : mem(x, setst^\bullet(a)) \Leftrightarrow mem(x, a) \wedge P(x) =_\iota cast_o(\text{TRUE})$
PROVE $mem(c, setst^\bullet(S))$

At this point, the proof obligation is a MS-FOL problem. It can be directly translated to SMT-LIB. Every fact in the context is encoded as an assertion; the negation of the goal is inserted as an assertion. The proof obligation will be considered proved if one SMT solver finds the problem to be unsatisfiable.

Unlike in this example, some axioms may involve arithmetic operators that must be translated as their SMT counterparts on the interpreted sort int. In general, the SMT logic used for the final problem is UFNIA (uninterpreted functions and non-linear integer arithmetic), but UFLIA (uninterpreted functions and linear integer arithmetic) is selected for veriT, since the solver only supports linear integer arithmetic.

4.2 Recovering Formulas

TLA$^+$'s interpretation of Boolean connectives is not standard, as the syntax allows for unconventional combinations of Boolean connectives with expressions.

However, the usual interpretation can be recovered by applying a simple transformation, which revolves around the insertion of conversions. In this section, we define this transformation as a function $\mathcal{B}^o$ mapping TLA$^+$ expressions to formulas of MS-FOL$^+$, and prove it both sound and complete.

The source TLA$^+$ signature is extended with the new operator $cast_o$. Intuitively, the transformation consists in applying the following rewriting rules on appropriate subexpressions:

$$e \longrightarrow cast_o(e) \qquad \text{(Injection)}$$
$$e \longrightarrow e =_\iota cast_o(\text{TRUE}) \qquad \text{(Projection)}$$

Expressions that appear to be formulas but occur in a non-Boolean context are injected into ι. Conversely, expressions that do not appear to be formulas but occur in a Boolean context are projected onto o. This is illustrated by the example below:

$$\forall x : (x = \text{FALSE}) \Rightarrow \neg x \quad \xrightarrow{\mathcal{B}^o} \quad \forall x^\iota : (x_\iota =_\iota cast_o(\text{FALSE})) \Rightarrow \neg(x_\iota =_\iota cast_o(\text{TRUE}))$$

The expression on the left is valid according to the semantics of TLA$^+$. The formula on the right is also valid, as long as $cast_o$ is interpreted as an injective function.

Formal Definition Given a TLA$^+$ signature Σ, we define $\Sigma^{\mathcal{B}}$ by adding $cast_o$ with type $o \to \iota$. All other operators are preserved with their types, with the following exceptions:

$$\begin{aligned} mem &: \iota \times \iota \to o \\ subseteq &: \iota \times \iota \to o \\ choose &: (\iota \to o) \to \iota \\ setst &: \iota \times (\iota \to o) \to \iota \end{aligned}$$

where *subseteq* is set inclusion, usually noted $\subseteq$. Some other exceptions are the operator $isafcn : \iota \to o$ and the operators for arithmetical comparisons, which are encoded as binary predicates; they are handled exactly like *mem* or *subseteq*, so we ignore them to make the presentation shorter. The functions defined below take their inputs from the core logic of TLA$^+$ under Σ and return terms, formulas or arguments in MS-FOL$^+$, under the signature $\Sigma^{\mathcal{B}}$.

Definition 8. *We define by mutual recursion the functions $\mathcal{B}^\iota$ and $\mathcal{B}^o$ on TLA$^+$ expressions and the function $\mathcal{B}^{op}$ on TLA$^+$ arguments:*

$$
\begin{aligned}
\mathcal{B}^\iota(x) &\triangleq x_\iota \\
\mathcal{B}^\iota(\mathit{setst}(e_1, \lambda x : e_2)) &\triangleq \mathit{setst}(\mathcal{B}^\iota(e_1), \lambda x^\iota : \mathcal{B}^o(e_2)) \\
\mathcal{B}^\iota(\mathit{choose}(\lambda x : e)) &\triangleq \mathit{choose}(\lambda x^\iota : \mathcal{B}^o(e)) \\
\mathcal{B}^\iota(K(f_1, \ldots, f_n)) &\triangleq K(\mathcal{B}^{op}(f_1), \ldots, \mathcal{B}^{op}(f_n)) && \text{if } K \neq \mathit{mem}, \mathit{subseteq} \\
\mathcal{B}^\iota(e) &\triangleq \mathit{cast}_o(\mathcal{B}^o(e)) && \text{if no rule above applies}
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{B}^o(\mathit{mem}(e_1, e_2)) &\triangleq \mathit{mem}(\mathcal{B}^\iota(e_1), \mathcal{B}^\iota(e_2)) \\
\mathcal{B}^o(\mathit{subseteq}(e_1, e_2)) &\triangleq \mathit{subseteq}(\mathcal{B}^\iota(e_1), \mathcal{B}^\iota(e_2)) \\
\mathcal{B}^o(e_1 = e_2) &\triangleq \mathcal{B}^\iota(e_1) =_\iota \mathcal{B}^\iota(e_2) \\
\mathcal{B}^o(\text{FALSE}) &\triangleq \text{FALSE} \\
\mathcal{B}^o(e_1 \Rightarrow e_2) &\triangleq \mathcal{B}^o(e_1) \Rightarrow \mathcal{B}^o(e_2) \\
\mathcal{B}^o(\forall x : e) &\triangleq \forall x^\iota : \mathcal{B}^o(e) \\
\mathcal{B}^o(e) &\triangleq \mathcal{B}^\iota(e) =_\iota \mathit{cast}_o(\text{TRUE}) && \text{if no rule above applies}
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{B}^{op}(e) &\triangleq \mathcal{B}^\iota(e) \\
\mathcal{B}^{op}(K) &\triangleq K \\
\mathcal{B}^{op}(\lambda x_1, \ldots, x_n : e) &\triangleq \lambda x_1^\iota, \ldots, x_n^\iota : \mathcal{B}^\iota(e)
\end{aligned}
$$

We call *injection* the last rule for $\mathcal{B}^\iota$ and *projection* the last rule for $\mathcal{B}^o$. To see why the definition is well-founded, remark that the injection and projection rules can be replaced by new rules that only make recursive calls of $\mathcal{B}^\iota$, $\mathcal{B}^o$ and $\mathcal{B}^{op}$ on subexpressions of e. This follows by a simple case analysis on the formation of expressions. For instance, if $\mathcal{B}^o(e)$ must be obtained by projection, it can be shown that e is either a variable x or an application $K(f_1, \ldots, f_n)$ with $K \neq \mathit{mem}, \mathit{subseteq}$. The projection rule could be replaced by the following four rules:

$$
\begin{aligned}
\mathcal{B}^o(x) &\triangleq x_\iota =_\iota \mathit{cast}_o(\text{TRUE}) \\
\mathcal{B}^o(\mathit{setst}(e_1, \lambda x : e_2)) &\triangleq \mathit{setst}(\mathcal{B}^\iota(e_1), \lambda x^\iota : \mathcal{B}^o(e_2)) =_\iota \mathit{cast}_o(\text{TRUE}) \\
\mathcal{B}^o(\mathit{choose}(\lambda x : e)) &\triangleq \mathit{choose}(\lambda x^\iota : \mathcal{B}^o(e)) =_\iota \mathit{cast}_o(\text{TRUE}) \\
\mathcal{B}^o(K(f_1, \ldots, f_n)) &\triangleq K(\mathcal{B}^{op}(f_1), \ldots, \mathcal{B}^{op}(f_n)) =_\iota \mathit{cast}_o(\text{TRUE}) \\
&\qquad \text{if } K \neq \mathit{mem}, \mathit{subseteq}
\end{aligned}
$$

Similarly, expanding $\mathcal{B}^o(e)$ in the injection rule will result in six new rules that only make recursive calls on subexpressions of e.

We admit that the three functions result in well-typed expressions. More precisely, $\mathcal{B}^\iota$ results in terms of the sort ι, $\mathcal{B}^o$ results in formulas of the sort o, and if f has arity n then $\mathcal{B}^{op}(f)$ has the n-ary type $\iota \times \cdots \times \iota \to \iota$.

Correctness For all Σ-interpretation $\mathcal{I}$, we define a $\Sigma^{\mathcal{B}}$-interpretation $\mathcal{I}^{\mathcal{B}}$ on the same domain D. We recall that interpretations in the target logic involve the Boolean domain D_o consisting of the two elements $\top$ and $\bot$. The interpretation of operators *mem*, *subseteq*, *choose* and *setst* is redefined. For all v_1, v_2 in D, $mem^{\mathcal{I}^{\mathcal{B}}}(v_1, v_2)$ is defined as $\top$ iff $mem^{\mathcal{I}}(v_1, v_2) = \top^D$, otherwise $\bot$. Similarly for *subseteq*. Given a function p in $D \to D_o$, we denote $\hat{p}$ the function in $D \to D$ that is defined by $\hat{p}(v) = \top^D$ if $p(v) = \top$, otherwise $\hat{p}(v) = \bot^D$. Then we define the new interpretation for *choose* and *setst* as follows: for all p, $choose^{\mathcal{I}^{\mathcal{B}}}(p)$ is $choose^{\mathcal{I}}(\hat{p})$; for all v and p, $setst^{\mathcal{I}^{\mathcal{B}}}(v, p)$ is $setst^{\mathcal{I}}(v, \hat{p})$. Finally, $cast_o^{\mathcal{I}^{\mathcal{B}}}$ is the function mapping $\top$ to $\top^D$ and $\bot$ to $\bot^D$.

Theorem 1. *Let $\mathcal{I}$ be a TLA$^+$ interpretation. The following propositions hold for all expressions e and arguments f:*

i) $[\![\mathcal{B}^\iota(e)]\!]^{\mathcal{I}^{\mathcal{B}}} = [\![e]\!]^{\mathcal{I}}$
ii) $[\![\mathcal{B}^o(e)]\!]^{\mathcal{I}^{\mathcal{B}}} = \top$ *iff* $[\![e]\!]^{\mathcal{I}} = \top^D$
iii) $[\![\mathcal{B}^o(e)]\!]^{\mathcal{I}^{\mathcal{B}}} = \bot$ *implies* $[\![e]\!]^{\mathcal{I}} = \bot^D$ *when $\mathcal{B}^o(e)$ is not obtained by applying the projection rule;*
iv) $[\![\mathcal{B}^{op}(f)]\!]^{\mathcal{I}^{\mathcal{B}}} = [\![f]\!]^{\mathcal{I}}$

Before going into the proof, let us justify the theorem's statement, especially properties (ii) and (iii). In place of property (iii), one could expect a property analogous to (ii), adapted for falsehood:

$$[\![\mathcal{B}^o(e)]\!]^{\mathcal{I}^{\mathcal{B}}} = \bot \text{ iff } [\![e]\!]^{\mathcal{I}} = \bot^D$$

This property fails for the case where e is a TLA$^+$ expression whose value is neither $\top^D$ nor $\bot^D$ and a projection occurs, *i.e.* $\mathcal{B}^o(e)$ is $\mathcal{B}^\iota(e) =_\iota cast_o(\text{TRUE})$. In that case, the formula $\mathcal{B}^o(e)$ is evaluated as $\bot$, thus the property does not hold. However, for all cases other than projection, the property does hold, hence the condition on (iii). As for why that property only specifies an implication and not an equivalence, it is a matter of convenience: the dual implication follows from property (ii) and the fact that $\top$ and $\bot$ are the two possible values for $[\![\mathcal{B}^o(e)]\!]^{\mathcal{I}^{\mathcal{B}}}$. Since it is simpler to prove an implication than an equivalence, we have chosen this form to simplify the proof below.

Proof. Since Definition 8 is well-founded, we can prove the result by induction on the construction of $\mathcal{B}^\iota(e)$, $\mathcal{B}^o(e)$, $\mathcal{B}^{op}(f)$. Each rule corresponds to a case. The induction hypothesis states that the properties hold for every recursive call preceding the current one.

Injection. Let $\mathcal{B}^\iota(e) \triangleq cast_o(\mathcal{B}^o(e))$. We must prove property (i) for $\mathcal{B}^\iota(e)$. The induction hypothesis applies to $\mathcal{B}^o(e)$, where the relevant properties are (ii) and (iii).

By definition:

$$[\![\mathcal{B}^\iota(e)]\!]_\theta^{\mathcal{I}^{\mathcal{B}}} = cast_o^{\mathcal{I}^{\mathcal{B}}}([\![\mathcal{B}^o(e)]\!]_\theta^{\mathcal{I}^{\mathcal{B}}}) = \begin{cases} \top^D & \text{if } [\![\mathcal{B}^o(e)]\!]_\theta^{\mathcal{I}^{\mathcal{B}}} = \top \\ \bot^D & \text{if } [\![\mathcal{B}^o(e)]\!]_\theta^{\mathcal{I}^{\mathcal{B}}} = \bot \end{cases}$$

According to the induction hypothesis, property (ii), if $[\![\mathcal{B}^o(e)]\!]_\theta^{\mathcal{I}^\mathcal{B}} = \top$ then $[\![e]\!]_\theta^{\mathcal{I}} = \top^D$. Furthermore, it is not possible that $\mathcal{B}^o(e)$ is obtained by applying the projection rule, because an injection never follows a projection. Therefore, property (iii) is applicable as part of the induction hypothesis: if $[\![\mathcal{B}^o(e)]\!]_\theta^{\mathcal{I}^\mathcal{B}} = \bot$ then $[\![e]\!]_\theta^{\mathcal{I}} = \bot^D$. In any case, we have $[\![\mathcal{B}^\iota(e)]\!]_\theta^{\mathcal{I}^\mathcal{B}} = [\![e]\!]_\theta^{\mathcal{I}}$.

Projection. Let $\mathcal{B}^o(e) \triangleq \mathcal{B}^\iota(e) =_\iota cast_o(\text{TRUE})$. We must prove property (ii). Property (iii) is immediate since this is the projection case. The induction hypothesis applies to $\mathcal{B}^\iota(e)$, where the relevant property is (i).

We have the following equivalences:

$$\begin{aligned}
[\![\mathcal{B}^o(e)]\!]_\theta^{\mathcal{I}^\mathcal{B}} = \top \textit{ iff } & [\![\mathcal{B}^\iota(e)]\!]_\theta^{\mathcal{I}^\mathcal{B}} = \top^D && (\text{since } [\![cast_o(\text{TRUE})]\!]_\theta^{\mathcal{I}^\mathcal{B}} = \top^D) \\
\textit{iff } & [\![e]\!]_\theta^{\mathcal{I}} = \top^D && (\text{by property (i) on } \mathcal{B}^\iota(e))
\end{aligned}$$

Implication. Let $e \triangleq e_1 \Rightarrow e_2$ and $\mathcal{B}^o(e) \triangleq \mathcal{B}^o(e_1) \Rightarrow \mathcal{B}^o(e_2)$. We must prove properties (ii) and (iii). The induction hypothesis applies to $\mathcal{B}^o(e_1)$ and $\mathcal{B}^o(e_2)$, where the relevant property is (ii) in both cases.

Property (ii) is proved by the following series of equivalences:

$$\begin{aligned}
&[\![\mathcal{B}^o(e_1 \Rightarrow e_2)]\!]_\theta^{\mathcal{I}^\mathcal{B}} = \top \\
&\quad \textit{iff } [\![\mathcal{B}^o(e_1)]\!]_\theta^{\mathcal{I}^\mathcal{B}} = \bot \text{ or } [\![\mathcal{B}^o(e_2)]\!]_\theta^{\mathcal{I}^\mathcal{B}} = \top && (\text{by MS-FOL's semantics of } \Rightarrow) \\
&\quad \textit{iff } [\![e_1]\!]_\theta^{\mathcal{I}} \neq \top^D \text{ or } [\![e_2]\!]_\theta^{\mathcal{I}} = \top^D && (\text{by property (ii)}) \\
&\quad \textit{iff } [\![e_1 \Rightarrow e_2]\!]_\theta^{\mathcal{I}} = \top^D && (\text{by TLA}^+\text{'s semantics of } \Rightarrow)
\end{aligned}$$

□

Property (iii) follows from property (ii) and the fact that $[\![e]\!]_\theta^{\mathcal{I}} \neq \top^D$ implies $[\![e]\!]_\theta^{\mathcal{I}} = \bot^D$ when e is an implication.

Set membership. Let $e \triangleq mem(e_1, e_2)$ and $\mathcal{B}^o(e) \triangleq mem(\mathcal{B}^\iota(e_1), \mathcal{B}^\iota(e_2))$. We must prove properties (ii) and (iii). The induction hypothesis applies to $\mathcal{B}^\iota(e_1)$ and $\mathcal{B}^\iota(e_2)$, where the relevant property is (i) in both cases.

We have the following equivalences:

$$\begin{aligned}
&[\![\mathcal{B}^o(mem(e_1, e_2))]\!]_\theta^{\mathcal{I}^\mathcal{B}} = \top \\
&\quad \textit{iff } mem^{\mathcal{I}}([\![\mathcal{B}^\iota(e_1)]\!]_\theta^{\mathcal{I}^\mathcal{B}}, [\![\mathcal{B}^\iota(e_2)]\!]_\theta^{\mathcal{I}^\mathcal{B}}) = \top^D && (\text{by definition of } mem^{\mathcal{I}^\mathcal{B}}) \\
&\quad \textit{iff } mem^{\mathcal{I}}([\![e_1]\!]_\theta^{\mathcal{I}}, [\![e_2]\!]_\theta^{\mathcal{I}}) = \top^D && (\text{by property (i) twice}) \\
&\quad \textit{iff } [\![mem(e_1, e_2)]\!]_\theta^{\mathcal{I}} = \top^D
\end{aligned}$$

This proves property (ii). For property (iii), like in the implication case, we can use the fact that $[\![e]\!]_\theta^{\mathcal{I}} \neq \top^D$ implies $[\![e]\!]_\theta^{\mathcal{I}} = \bot^D$ when e is an application of the set membership operator. Indeed, a statement $x \in y$ is specified as either TRUE or FALSE in TLA$^+$.

Set comprehension. Let $e \triangleq setst(e_1, \lambda x : e_2)$ and $\mathcal{B}^\iota(e) \triangleq setst(\mathcal{B}^\iota(e_1), \lambda x^\iota : \mathcal{B}^o(e_2))$. We must prove property (i). The induction hypothesis lets us apply property (i) to $\mathcal{B}^\iota(e_1)$ and property (ii) to $\mathcal{B}^o(e_2)$.

Let p be the function $[\![\lambda x^\iota : \mathcal{B}^o(e_2)]\!]_\theta^{\mathcal{I}^\mathcal{B}}$, *i.e.* p is the function of $D \to D_o$ such that $p(v) = [\![\mathcal{B}^o(e_2)]\!]_{\theta^x_v}^{\mathcal{I}^\mathcal{B}}$. By definition:

$$\begin{aligned}
&[\![\mathcal{B}^\iota(setst(e_1, \lambda x : e_2))]\!]_\theta^{\mathcal{I}^\mathcal{B}} \\
&\quad = setst^\mathcal{I}([\![\mathcal{B}^\iota(e_1)]\!]_\theta^{\mathcal{I}^\mathcal{B}}, \hat{p}) \quad \text{(by definition of } setst^{\mathcal{I}^\mathcal{B}}\text{)} \\
&\quad = setst^\mathcal{I}([\![e_1]\!]_\theta^\mathcal{I}, \hat{p}) \qquad \text{(by property (i))}
\end{aligned}$$

Let q be the function $[\![\lambda x : e_2]\!]_\theta^\mathcal{I}$, *i.e.* q is the function of $D \to D$ such that $q(v) = [\![e_2]\!]_{\theta^x_v}^\mathcal{I}$. By property (ii), we have $p(v) = \top$ iff $q(v) = \top^D$ for all v in D. Therefore, $\hat{p}(v) = \top^D$ iff $q(v) = \top^D$. It follows that

$$setst^\mathcal{I}([\![e_1]\!]_\theta^\mathcal{I}, \hat{p}) = setst^\mathcal{I}([\![e_1]\!]_\theta^\mathcal{I}, q)$$

by set extensionality. Since the right member is equal to $[\![setst(e_1, \lambda x : e_2)]\!]_\theta^\mathcal{I}$ by definition, property (i) is proved.

We omit the other cases from the proof. The set inclusion case is similar to the set membership case. The choose case is similar to the set comprehension case; the principle of determinacy for CHOOSE must be used instead of set extensionality. Every other case is either immediate or straightforward by application of the induction hypothesis. □

Soundness follows trivially from the property (ii) of Theorem 1. Completeness also follows if the interpretation of $cast_o$ is constrained in the target logic. This is achieved by inserting the following axiom in the final problem:

$$cast_o(\text{TRUE}) \neq_\iota cast_o(\text{FALSE}) \tag{B}$$

Theorem 2 (Soundness and Completeness of $\mathcal{B}^o$). *Let e be a TLA$^+$ expression. Then e is satisfiable iff $\mathcal{B}^o(e)$ is satisfiable by a model of* (B).

Proof. If $\mathcal{I} \models e$ then $\mathcal{I}^\mathcal{B} \models \mathcal{B}^o(e)$ by Theorem 1. Clearly $\mathcal{I}^\mathcal{B}$ satisfies (B).

Conversely, if $\mathcal{J} \models \mathcal{B}^o(e)$ with $\mathcal{J}$ model of (B), then it suffices to define an interpretation $\mathcal{I}$ such that $\mathcal{I}^\mathcal{B} = \mathcal{J}$. Then, by Theorem 1, we will have $\mathcal{I} \models e$.

$\mathcal{I}$ is defined as follows. First, we define $\top^D \triangleq [\![cast_o(\text{TRUE})]\!]^\mathcal{J}$ and $\bot^D \triangleq [\![cast_o(\text{FALSE})]\!]^\mathcal{J}$. Since $\mathcal{J}$ satisfies (B) by hypothesis, these are indeed distinct values. Next, we define the interpretations $mem^\mathcal{I}$, $subseteq^\mathcal{I}$, $setst^\mathcal{I}$ and $choose^\mathcal{I}$. For all other operators K, it suffices to take $K^\mathcal{I} \triangleq K^\mathcal{J}$. We define $mem^\mathcal{I}(v_1, v_2) = \top^D$ if $mem^\mathcal{J}(v_1, v_2) = \top$, otherwise $\bot^D$. Then clearly $mem^{\mathcal{I}^\mathcal{B}} = mem^\mathcal{J}$. For all v in D and k in $D \to D$, we define $setst^\mathcal{I}(v, k) \triangleq setst^\mathcal{J}(v, p)$ where p is the predicate on D defined by $p(u) = \top$ iff $k(u) = \top^D$. In particular, if k is some $\hat{q}$, then $p = q$. Then $setst^{\mathcal{I}^\mathcal{B}}(v, p) = setst^\mathcal{I}(v, \hat{p}) = setst^\mathcal{J}(v, p)$ for all v and p, therefore $setst^{\mathcal{I}^\mathcal{B}} = setst^\mathcal{J}$. For $subseteq^\mathcal{I}$ and $choose^\mathcal{I}$, the method is analogous to $mem^\mathcal{I}$ and $setst^\mathcal{I}$ respectively. □

4.3 Axiom Selection

The objective of this step is to make explicit declarations for the relevant TLA$^+$ primitives and insert their axioms in the final proof obligation. To each TLA$^+$ primitive, we have assigned a number of axioms (typically 1–3). Which axioms are relevant to a given primitive is most of the time obvious. For example, the intersection operation *cap* is specified by the axiom

$$\forall a^\iota, b^\iota, x^\iota : mem(x, cap(a, b)) \Leftrightarrow mem(x, a) \wedge mem(x, b)$$

The algorithm for axiom insertion is straightforward. The proof obligation is parsed until a TLA$^+$ primitive is found. The appropriate declaration for the primitive is inserted in the context of the proof obligation, and its axioms are inserted as new hypotheses. Axioms may contain occurrences of TLA$^+$ primitives that did not occur in the proof obligation before, so the procedure is repeated recursively. It terminates when no TLA$^+$ primitives remain in the proof obligation.

Let T be the collection of expressions $\mathcal{B}^o(e)$ for all TLA$^+$ axioms e. Then the procedure is sound as long as all the inserted axioms are consequences of T. Most of the axioms we use are just elementary reformulations of those found in the documentation, so it is easy to verify them by comparison. One essential difference is that we favor axioms containing fewer references to other primitive operators. For instance, $a \mathinner{..} b$ is defined as $\{z \in Int : a \leq z \wedge z \leq b\}$ in the reference manual of TLA$^+$. To avoid the reference to set comprehension, we insert the axiom

$$\forall a^\iota, b^\iota, z^\iota : z \in a \mathinner{..} b \Leftrightarrow z \in Int \wedge a \leq z \wedge z \leq b$$

Our axioms for SMT are discussed in more details in Section 5, which is about the optimization of SMT's instantiation module through trigger annotations. The only part of the axiomatization we detail now is integer arithmetic, as this part of the theory is handled in a special way.

Axioms for Integer Arithmetic The encoding features a special set of axioms for integer arithmetic. The idea is to link TLA$^+$'s arithmetic with SMT's builtin arithmetic, in order to reason more efficiently on integers. Intuitively, a correspondence is established between the predicate $n \in Int$ and the sort int. Here is a summary of the axioms for integer arithmetic:

$$cast_{\mathsf{int}} : \mathsf{int} \to \iota$$
$$proj_{\mathsf{int}} : \iota \to \mathsf{int}$$

$$\forall z^{\mathsf{int}} : mem(cast_{\mathsf{int}}(z), Int) \qquad \text{(CastImage)}$$
$$\forall x^\iota : mem(x, Int) \Rightarrow x =_\iota cast_{\mathsf{int}}(proj_{\mathsf{int}}(x)) \qquad \text{(CastSurjective)}$$
$$\forall z^{\mathsf{int}} : z =_{\mathsf{int}} proj_{\mathsf{int}}(cast_{\mathsf{int}}(z)) \qquad \text{(CastInjective)}$$
$$\forall z_1^{\mathsf{int}}, z_2^{\mathsf{int}} : plus(cast_{\mathsf{int}}(z_1), cast_{\mathsf{int}}(z_2)) =_\iota cast_{\mathsf{int}}(z_1 +_{\mathsf{int}} z_2) \qquad \text{(PlusTyping)}$$

The operator $cast_{\mathsf{int}}$ gives the TLA$^+$ integer corresponding to a SMT integer. $proj_{\mathsf{int}}$ converts in the other direction; it is specified for elements of the set Int, unspecified for other elements. By declaring $proj_{\mathsf{int}}$ as a left-inverse of $cast_{\mathsf{int}}$, we specify $cast_{\mathsf{int}}$ as injective. This way of specifying an operator as injective is interesting as it uses only one universal quantifier, making it easier for SMT to instantiate the axiom [11].

The axiom (PlusTyping) specifies a correspondence between TLA$^+$'s addition *plus* and SMT's interpreted addition $+_{\mathsf{int}}$. There is another similar axiom for every TLA$^+$ primitive that has a counterpart in SMT. In the case of Euclidian division and remainder, the correspondence is restricted to positive numbers:

$$\forall z_1^{\mathsf{int}}, z_2^{\mathsf{int}} : z_2 >_{\mathsf{int}} 0 \Rightarrow quotient(cast_{\mathsf{int}}(z_1), cast_{\mathsf{int}}(z_2)) =_\iota cast_{\mathsf{int}}(z_1 \div_{\mathsf{int}} z_2)$$

As for the TLA$^+$ constants 0, 1, 2, it is far simpler to just translate them as $cast_{\mathsf{int}}(0)$, $cast_{\mathsf{int}}(1)$, $cast_{\mathsf{int}}(2)$ where 0, 1 and 2 refer to the SMT constants. This is performed at the very last moment when translating the preprocessed proof obligations to SMT-LIB.

Those axioms are not derived from the theory of TLA$^+$, but extend it conservatively. The soundness of the method relies on the fact that the arithmetics of TLA$^+$ and SMT are assumed to agree. More precisely, if a TLA$^+$ expression can be derived using the axioms above and SMT's interpretation of integer arithmetic, it can also be derived from the theory of TLA$^+$ alone.

4.4 Elimination of Second-order Applications

Axiom selection may insert axioms parameterized by operator symbols, and there may be second-order applications left in the proof obligation. The final step before translation to SMT-LIB reduces terms to first-order ones. Translations of higher-order logic to first-order logic have been developed for proof assistants [4, 23]. Compilers of functional programming languages also implement techniques to translate higher-order functions into a first-order language [10]. While we took inspiration from these works, the scope of our implementation is more restrained, as TLA$^+$ is not truly a higher-order logic, just a first-order logic with second-order applications.

The first half of this section describes our procedure in the general case; the second half will illustrate with a concrete example. First, we introduce some notations for substitutions. Let $t\,[x_1, \ldots, x_n]$ denote a term t containing the free variables $x_1, \ldots, x_n$. Then the result of substituting terms u_i for each x_i is denoted $t\,[u_1, \ldots, u_n]$.

The procedure parses expressions in a bottom-up way until a second-order application is found. We consider the particular case with one higher-order parameter

$$K(t_1, \ldots, t_m, \lambda x_1^\iota, \ldots, x_n^\iota : t)$$

Generalizing to multiple higher-order arguments is straightforward. If the higher-order parameter is a symbol F, we may treat it as $\lambda x_1^\iota, \ldots, x_n^\iota : F(x_1, \ldots, x_n)$ where n is the arity of F.

The inner term t is parsed to identify first-order subterms to abstract away. Formally, the procedure finds a term $t'[x_1, \ldots, x_n, y_1, \ldots, y_p]$ and terms u_j for $1 \leq j \leq p$ such that $t = t'[x_1, \ldots, x_n, u_1, \ldots, u_p]$, with each y_j occurring once in t', and such that no u_j contains a variable among $x_1, \ldots, x_n$. This is implemented by parsing subterms of t in a top-down way (to consider the largest subterms first); subterms not containing the variables $x_1, \ldots, x_n$ are replaced by fresh variables and returned as part of the result.

Having abstracted away subterms from t, we can define a new operator $K^\bullet$ which is specified as a specialized version of K. This can be expressed by the following definition:

$$K^\bullet(z_1, \ldots, z_m, y_1, \ldots, y_p) \triangleq K(z_1, \ldots, z_m, \lambda x_1^\iota, \ldots, x_n^\iota : t'[x_1, \ldots, x_n, y_1, \ldots, y_p])$$

With this definition, the initial second-order application can be rewritten as

$$K^\bullet(t_1, \ldots, t_m, u_1, \ldots, u_p)$$

To actually specify $K^\bullet$ in the SMT problem, we have to insert axioms. Typically, second-order applications result from TLA$^+$ expressions like $\{x \in S : e\}$. In this case, the axiom is just an instance of some general axiom schema and can be generated by applying a substitution. Other second-order applications result from user-defined second-order operators; we reduce these applications to first-order expressions, but currently offer no way of specifying the created operators. This will allow SMT solvers to attempt the proof, but they will likely fail if the second-order operator is relevant to the proof obligation.

Let us now look at a concrete example. Consider the TLA$^+$ expression

$$\exists i : \{n \in Int : n \neq 0\} = \{n \in Int : n \neq i\}$$

The expression is trivially valid by providing 0 as witness. At this stage of the encoding, the expression is in the form

$$\exists i^\iota : setst(Int, \lambda n^\iota : n \neq_\iota cast_{\mathsf{int}}(0)) =_\iota setst(Int, \lambda n^\iota : n \neq_\iota i)$$

We first consider the leftmost second-order application. The higher-order parameter is $\lambda n^\iota : n \neq_\iota cast_{\mathsf{int}}(0)$. The subterm $cast_{\mathsf{int}}(0)$ does not contain the variable n. It is replaced by a fresh variable of the same type ι to obtain the lambda-term $\lambda n^\iota : n \neq_\iota y$. The procedure declares a new operator $setst^\bullet$ which is specified as *setst* specialized for this lambda-term. $setst^\bullet$ is assigned the type $\iota \times \iota \to \iota$. The first ι is for the original first-order parameter, the second is for the new parameter y.

Since *setst* is specified by the schema of set comprehension, $setst^\bullet$ is specified by one of its instances, in this case

$$\forall y^\iota, a^\iota, x^\iota : mem(x, setst^\bullet(a, y)) \Leftrightarrow mem(x, a) \wedge x \neq_\iota y$$

This axiom is inserted in the current problem.

The original application is rewritten as $setst^{\bullet}(Int, cast_{\mathsf{int}}(0))$. Next, when the procedure resumes with the other second-order application, whose higher-order parameter is $\lambda n^{\iota} : n \neq_{\iota} i$, the same lambda-term $\lambda n^{\iota} : n \neq_{\iota} y$ is found. In a situation like this, there is no need to create and specify a new symbol, because we can reuse a previous one. Ultimately, both applications are rewritten using $setst^{\bullet}$:

$$\exists i^{\iota} : setst^{\bullet}(Int, cast_{\mathsf{int}}(0)) =_{\iota} setst^{\bullet}(Int, i)$$

Our procedure keeps track of what second-order applications were reduced and how, in order to detect these situations, which are rather common in TLA$^+$ proofs. Only the largest possible subterms are selected when the body of a lambda-term is inspected to identify first-order parameters. While this may result in specialized symbols to feature irrelevant parameters, this method ensures that they are general enough for potential reuses.

5 Trigger Instantiation

5.1 General Definition

SMT solvers are primarily designed for problems of first-order logic without quantifiers. For problems containing quantifiers, the SMT procedure can be modeled as the interaction between a *ground solver* and an *instantiation module*. The solver manages a set of ground formulas E and a set of quantified ground formulas Q, which are assumed to be of the form $\forall x_1, \ldots, x_n : \phi$. The purpose of the instantiation module is to generate instances $\phi\sigma$, where σ is a ground substitution. These ground instances are then added to E so that the ground solver can progress.

Consider, for example, the following SMT problem involving some uninterpreted sort s, two functions $f, g : s \to s$ and two constants $a, b : s$. The problem consists of the single quantified formula

$$\forall x^{s} : g(f(x)) =_{s} x$$

and two ground formulas

$$f(a) =_{s} f(b), \qquad a \neq_{s} b$$

By itself, the ground problem is satisfiable, so SMT's ground solver will either find a model for it, or fail to conclude. The instantiation module is called next to generate more ground formulas. Suppose the two following instances are generated:

$$g(f(a)) =_{s} a, \qquad g(f(b)) =_{s} b$$

The new ground problem, which contains four formulas, is now unsatisfiable—it is possible to infer $a =_{s} b$, resulting in a contradiction. SMT's ground solver will detect this, and the procedure will terminate.

Deciding which instances are relevant to a given problem is very difficult in general. SMT solvers implement different strategies to overcome this difficulty.

The approach we focus on is called heuristics-based and features annotations known as *triggers*. The general form of a formula with a trigger is

$$\forall x_1, \ldots, x_n : \{p_1, \ldots, p_k\} \quad \phi$$

where each p_i is a term called a pattern. Free variables of every pattern must be included in the set $\{x_1, \ldots, x_n\}$, and every x_i must occur in at least one pattern.

SMT solvers use triggers to infer relevant instances from terms matching the patterns [12]. Let E be the current ground problem. Given the trigger $\{p_1, \ldots, p_k\}$, a match is a list of *known* terms $t_1, \ldots, t_k$ and a ground substitution σ such that, for all i, $E \models t_i = p_i\sigma$ in the theory of equality and uninterpreted functions. Known terms are typically those that appear in E. Intuitively, all patterns p_i must be matched simultaneously with ground terms in the problem. The substitution σ is then used to generate an instance.

Here is a concrete example in the context of set theory:

$$\begin{aligned}&\forall a^\iota, b^\iota, x^\iota : \{mem(x, cap(a, b))\}\\ &\quad mem(x, cap(a, b)) \Leftrightarrow mem(x, a) \wedge mem(x, b)\end{aligned}$$

Whenever a term $mem(t_1, cap(t_2, t_3))$ occurs in the ground problem, SMT will detect a match and use the substitution $\{x \mapsto t_1, a \mapsto t_2, b \mapsto t_3\}$ to generate the instance

$$mem(t_1, cap(t_2, t_3)) \Leftrightarrow mem(t_1, t_2) \wedge mem(t_1, t_3)$$

In general, quantified formulas can be annotated with several triggers, in which case any individual trigger can generate an instance.

SMT solvers implement different strategies for instantiating quantified formulas, including the automatic generation of triggers [29,30]. Through trial and error by inspecting difficult proof obligations, we found that more problems could be solved if we manually selected triggers for our TLA$^+$ theory. In the rest of this section, we present the ideas underlying our strategy for the axioms of set theory, functions, and integer arithmetic. The full list of SMT axioms with triggers used in our encoding includes 85 axioms and can be found in Appendix A. We conclude the section with a detailed look at a concrete TLA$^+$ proof obligation, to illustrate how triggers lead SMT solvers to the solution in practice.

5.2 Triggers for Set Theory

Let us start with some general remarks about axioms and triggers. For any formula ϕ, all subformulas of ϕ are assigned a polarity: positive or negative. The whole formula, ϕ, has the positive polarity. The polarity is only reversed when going down on the left of implications: if $\phi_1 \Rightarrow \phi_2$ has the positive (resp. negative) polarity, then ϕ_1 has the negative (resp. positive) polarity. We call *weak* any quantifier $\forall x^s$ that occurs in a positive context (the subformula has the positive polarity). Given the way we defined $\neg$ and $\exists x$ as notations, a weak quantifier can also be a $\exists x$ that occurs in a negative context. Any quantifier that is not weak is called *strong*.

Weak quantifiers are essentially universal, while strong quantifiers are essential existential. This means that all weak quantifiers, and only those, are handled by SMT's instantiation module. Strong quantifiers are handled in other ways, for instance they can be eliminated by SMT through Skolemization [2]. To ensure all weak quantifiers are covered by at least one trigger, we have formulated axioms in such a way that these quantifiers are put at the beginning of formulas. Thus, the general form of an axiom in our case is

$$\forall x_1, \ldots, x_n : \{p_1, \ldots, p_k\} \quad \phi$$

where ϕ only contains strong quantifiers.

Let us illustrate with the axiom specifying set inclusion:

$$\forall a^\iota, b^\iota : \mathit{subseteq}(a, b) \Leftrightarrow \left[\forall x^\iota : \mathit{mem}(x, a) \Rightarrow \mathit{mem}(x, b)\right]$$

Since an equivalence is actually a conjunction of implications, there are actually two nested quantifiers in this formula. When the equivalence is broken down in two implications, it becomes obvious that one quantifier is weak and the other strong. To move the weak quantifier at the top, it is best to also break down the axiom in two parts. The axioms we use to specify set inclusion are:

$$\forall a^\iota, b^\iota : \{\mathit{subseteq}(a, b)\} \qquad \text{(SubseteqIntro)}$$
$$\left[\forall x^\iota : \mathit{mem}(x, a) \Rightarrow \mathit{mem}(x, b)\right] \Rightarrow \mathit{subseteq}(a, b)$$

$$\forall a^\iota, b^\iota, x^\iota : \{\mathit{subseteq}(a, b), \mathit{mem}(x, a)\} \qquad \text{(SubseteqElim)}$$
$$\mathit{subseteq}(a, b) \wedge \mathit{mem}(x, a) \Rightarrow \mathit{mem}(x, b)$$

By ensuring all weak quantifiers are annotated with a trigger, we increase the predictability of our approach. If we had chosen a trigger for the original axiom, for instance $\{\mathit{subseteq}(a, b)\}$, then any time the axiom is instantiated, a new quantified formula would be introduced in the problem. Whereas by providing SMT with the axiom (SubseteqElim) and a trigger including the variable x, we ensure all generated instances are ground.

We now detail how triggers are selected. For all axioms $\forall x_1^{s_1}, \ldots, x_n^{s_n} : \phi$, we select triggers $\{p_1, \ldots, p_k\}$ where each pattern p_i is a subterm of ϕ. For any potential trigger, we consider the subterms of ϕ that are not subterms of any p_i. We say that these terms can be *produced* by the trigger. Intuitively, produced terms are terms that may occur in the new ground problem as a result of the instance being generated. For instance, the trigger of axiom (SubseteqElim) can only produce the term $\mathit{mem}(x, b)$. Note that, when a produced term happens to be Boolean, the fact that it is produced does not entail its truth.

Terms produced by triggers are terms that may become known as a result of instantiating axioms. Therefore, they determine how axioms trigger each other through the insertion of ground instances into the current problem. If triggers produce irrelevant terms, then SMT will generate irrelevant instances. Conversely, producing relevant terms increases the chance SMT generates relevant instances.

For set theory in the context of TLA$^+$, we make the assumption that most proof obligations can be solved through elementary reasoning on the sets that are already mentioned in them, and only those sets. From this assumption, we derive our strategy for set theory:

1. No trigger should produce a term $C(t_1, \ldots, t_n)$ where C is a set-theoretic constructor (enumeration, comprehension, union, intersection, etc.)
2. Triggers should only produce terms $mem(x, y)$, and we should select as many triggers as possible for every axiom.

Let us see how these principles apply on a concrete example: the axiom specifying set intersection, displayed below.

$$\begin{aligned} \forall a^\iota, b^\iota : \; & \{mem(x, cap(a, b))\} \\ & \{cap(a, b), mem(x, a)\} \\ & \{cap(a, b), mem(x, b)\} \\ & mem(x, cap(a, b)) \Leftrightarrow mem(x, a) \wedge mem(x, b) \end{aligned} \tag{CapDef}$$

The first trigger is the most straightforward: when a term $mem(x, cap(a, b))$ appears, the appropriate instance of the definition is added to the problem. That trigger can produce the terms $mem(x, a)$ and $mem(x, b)$. It will never produce the proposition $mem(x, cap(a, b))$ itself, because this term has to be known already for the match to happen.

It is interesting to have at least one trigger that does produce the term $mem(x, cap(a, b))$, to detect potential elements of known sets $cap(a, b)$. The second and third triggers are included for that reason. The second trigger produces $mem(x, cap(a, b))$ and also $mem(x, b)$. Note that $mem(x, b)$ does not need to be known to hold, or even to occur in the problem, for the match to happen. Note also that the potential trigger $\{cap(a, b), mem(x, a), mem(x, b)\}$ is not included, because the knowledge of either $mem(x, a)$ or $mem(x, b)$ alongside $cap(a, b)$ gives enough information to find a match for the variables x, a and b.

The following trigger is rejected:

$$\{mem(x, a), mem(x, b)\}$$

Indeed, this trigger can produce the term $cap(a, b)$, which is built from a set-theoretic operator. Thus, there is a risk that the trigger would introduce irrelevant terms in the problem. Moreover, this particular trigger suffers from a more critical problem as it can make the instantiation procedure loop on itself: given some known term $mem(S, T)$, the match $\{x \mapsto S, a \mapsto T, b \mapsto T\}$ results in the production of $mem(S, cap(T, T))$, leading to a new match, and so on.

Set Extensionality The method described so far does not apply to the axiom of set extensionality, because there is no subterm in it that can serve to make a trigger. However, we still want some control over the instantiation of that axiom, because SMT solvers struggle to find relevant instances by themselves.

Our trigger for set extensionality involves a new operator $\mathit{appext} : \iota \times \iota \to o$. Its only purpose is to trigger an instance of extensionality. By controlling the generation of terms $\mathit{appext}(x, y)$, we prompt SMT to generate an instance of set extensionality for x and y. In our case, we will only generate an instance for certain equalities that occur in the proof obligation. Since it is not possible to use a term $x = y$ in a trigger, we rewrite relevant equalities $x = y$ as $\mathit{equals}(x, y)$, where *equals* is defined as an alias for equality. Another axiom lets SMT generate a term $\mathit{appext}(x, y)$ for every occurrence of $\mathit{equals}(x, y)$. The three axioms are:

$$\forall x^\iota, y^\iota : \{\mathit{appext}(x, y)\} \qquad \text{(SetExtensionality)}$$
$$\big[\forall z^\iota : \mathit{mem}(z, x) \Leftrightarrow \mathit{mem}(z, y)\big] \Rightarrow x = y$$

$$\forall x^\iota, y^\iota : \{\mathit{equals}(x, y)\} \qquad \text{(EqualsDef)}$$
$$\mathit{equals}(x, y) \Leftrightarrow x = y$$

$$\forall x^\iota, y^\iota : \{\mathit{equals}(x, y)\} \qquad \text{(EqualsTriggersExt)}$$
$$\mathit{appext}(x, y)$$

For the method to work, the relevant equalities $x = y$ must be rewritten as $\mathit{equals}(x, y)$ in the proof obligation. This is done through a simple pass over the proof obligation. All equalities are not relevant; we only rewrite equalities that occur in a positive context, and only if one of the two member is built from a set-theoretic operator. The restriction to positive contexts ensures we only generate instances for equalities that must be proved; for example, it is not necessary to use set extensionality to solve the goal $\forall x, y : x = \emptyset \Rightarrow y \notin x$. The second restriction helps identifying equalities between sets; otherwise an instance would be generated for goals like $1 + 1 = 2$.

To give a concrete example, consider the TLA$^+$ goal $a = b \Rightarrow (a \cap c) = (c \cap b)$. This expression is ultimately encoded as $a = b \Rightarrow \mathit{equals}(\mathit{cap}(a, c), \mathit{cap}(c, b))$. The first equality occurs in a negative context (left of $\Rightarrow$) so it is translated with SMT's builtin equality. The second equality is a positive occurrence and at least one member is built from the operator *cap*, so it is encoded with *equals*.

5.3 Triggers for Functions

We start with the axioms that specify the operators $[x \in S \mapsto F(x)]$ (explicit function), DOMAIN f (domain) and $f[x]$ (application). On SMT's end, the theory features four symbols:

$$\mathit{fcn}_F : \iota^{n+1} \to \iota \quad \text{where } F : \iota^{n+1} \to \iota \qquad \mathit{isafcn} : \iota \to o$$
$$\mathit{domain} : \iota \to \iota \qquad \mathit{fcnapp} : \iota \times \iota \to \iota$$

DOMAIN f corresponds to the term $\mathit{domain}(f)$, $f[x]$ corresponds to the term $\mathit{fcnapp}(f, x)$. The operator *isafcn* has no corresponding TLA$^+$ notation. The symbol fcn_F is actually a collection of symbols; for every second-order TLA$^+$ expression $[x \in S \mapsto F(x, c_1, \ldots, c_n)]$, a first-order symbol fcn_F is generated during our reduction to first-order logic in order to represent the expression as

$fcn_F(x, c_1, \ldots, c_n)$. The parameters $c_1, \ldots, c_n$ are those identified by our procedure (see Section 4.4).

The operators are specified by three axiom schemas (for the parameter F), displayed below with our triggers. They are usually quantified over several parameters $c_1^\iota, \ldots, c_n^\iota$; we display the case for $n = 1$ to simplify the presentation.

$$\forall c^\iota, a^\iota : \{fcn_F(a, c)\} \qquad \text{(FcnIsafcn)}$$
$$isafcn(fcn_F(a, c))$$

$$\forall c^\iota, a^\iota : \{fcn_F(a, c)\} \qquad \text{(FcnDomain)}$$
$$domain(fcn_F(a, c)) = a$$

$$\forall c^\iota, a^\iota, x^\iota : \{fcnapp(fcn_F(a, c), x)\} \qquad \text{(FcnApp)}$$
$$\{fcn_F(a, c), mem(x, a)\}$$
$$mem(x, a) \Rightarrow fcnapp(fcn_F(a, c), x) = F(x, c)$$

To select those triggers, we applied a method analogous to the one we described for set theory. We assume proof obligations can be solved by reasoning with the functions already mentioned. The consequence for triggers is that we reject triggers that produce the term $fcn_F(a, c)$. Moreover, we can see that each axiom defines a term $K(fcn_F(a, c), \ldots)$ where K is either *isafcn*, *domain* or *fcnapp*. We ensure there is always at least one trigger that can produce that term, to help SMT infer relevant information about the functions it knows about.

The theory of TLA$^+$ functions also involves the set $[a \to b]$, which is represented by $arrow(a, b)$ where $arrow : \iota \times \iota \to \iota$. For all f, $f \in [a \to b]$ iff f is a function such that DOMAIN $f = a$ and $f[x] \in b$ for all $x \in a$. This is specified by an axiom, to which we can apply the method already described in the section about set theory:

$$\forall a^\iota, b^\iota, f^\iota : \{mem(f, arrow(a, b))\} \qquad \text{(ArrowIntro)}$$
$$\wedge\ isafcn(f)$$
$$\wedge\ domain(f) = a$$
$$\wedge\ \left[\forall x^\iota : mem(x, a) \Rightarrow mem(fcnapp(f, x), b)\right]$$
$$\Rightarrow mem(f, arrow(a, b))$$

$$\forall a^\iota, b^\iota, f^\iota : \{mem(f, arrow(a, b))\} \qquad \text{(ArrowElim}_1\text{)}$$
$$mem(f, arrow(a, b)) \Rightarrow\ \wedge\ isafcn(f)$$
$$\wedge\ domain(f) = a$$

$$\forall a^\iota, b^\iota, f^\iota, x^\iota : \{mem(f, arrow(a, b)), mem(x, a)\} \qquad \text{(ArrowElim}_2\text{)}$$
$$\{mem(f, arrow(a, b)), fcn(f, x)\}$$
$$mem(f, arrow(a, b)) \wedge mem(x, a) \Rightarrow mem(fcnapp(f, x), b)$$

The original axiom is an equivalence with a nested quantifier $\forall x^\iota$; breaking the equivalence in two, and the conjunction in two for the elimination part, allows

us to move the weak quantifier $\forall x^\iota$ to the front of the formula in (ArrowElim$_2$). Triggers are selected so that no term $\mathit{arrow}(a, b)$ can be produced.

Axiom (ArrowIntro) is used in proofs to infer a proposition $f \in [a \to b]$. However, in order for the axiom to be instantiated by the associated trigger, the proposition must already be known; this happens typically when the goal to prove is precisely $f \in [a \to b]$. For situations where it is not obvious that $f \in [a \to b]$ must be derived to finish the proof, we need a trigger that can actually produce that term. However, there is no obvious way to extend (ArrowIntro) with a good trigger satisfying this requirement. One option would be the trigger $\{\mathit{isafcn}(f), \mathit{arrow}(a, b)\}$, but this would result in every function f to be paired up with every set $[a \to b]$ in the problem, leading to many irrelevant instances.

Assuming a set $[a \to b]$ is known, good candidates for elements of that set are all terms of the form $[x \in a \mapsto F(x)]$. To complement the theory of functions, we also add the following axiom, which does produce terms of the form $\mathit{mem}(f, \mathit{arrow}(a, b))$, only f has to be an explicit function.

$$\forall c^\iota, a^\iota, b^\iota : \{\mathit{fcn}_F(a, c), \mathit{arrow}(a, b)\} \qquad \text{(FcnTyping)}$$
$$\big[\forall x^\iota : \mathit{mem}(x, a) \Rightarrow \mathit{mem}(F(x, c), b)\big] \Rightarrow \mathit{mem}(\mathit{fcn}_F(a, c), \mathit{arrow}(a, b))$$

Finally, functional extensionality is specified in a separate axiom:

$$\begin{aligned}&\forall f^\iota, g^\iota : \{\mathit{isafcn}(f), \mathit{isafcn}(g)\} \qquad \text{(FcnExtensionality)}\\ &\quad \wedge \mathit{isafcn}(f) \wedge \mathit{isafcn}(g)\\ &\quad \wedge \mathit{domain}(f) =_\iota \mathit{domain}(g)\\ &\quad \wedge \big[\forall x^\iota : \mathit{mem}(x, \mathit{domain}(f)) \Rightarrow \mathit{fcnapp}(f, x) =_\iota \mathit{fcnapp}(g, x)\big]\\ &\quad \Rightarrow f =_\iota g\end{aligned}$$

This trigger leads to one instance of extensionality being generated for every pair of known functions f, g. If a problem involves many different functions, it is likely that many instances will be irrelevant, so we admit that this is not a very efficient solution.

The theories of tuples, records and sequences are handled very similarly to the theory of functions. Each of these theories introduces a new constructor operator (several in the case of sequences), a new set-theoretic operator, and reuses the operators *isafcn*, *domain* and *fcnapp*. For instance, a tuple $\langle x_1, \ldots, x_n \rangle$ is represented by $\mathit{tup}_n(x_1, \ldots, x_n)$ where tup_n plays a role analogous to fcn_F in the theory. The product set $a \times b$ is represented by $\mathit{product}(a, b)$, and *product* is analogous to *arrow*.

5.4 Triggers for Arithmetic

There are only three axioms that specify *cast*, and one axiom for each TLA$^+$ operator that has a counterpart in SMT's arithmetic. Here is a summary of the

axioms with their triggers:

$$\forall z^{\mathsf{int}} : \{cast_{\mathsf{int}}(z)\} \qquad \text{(CastImage)}$$
$$\quad mem(cast_{\mathsf{int}}(z), Int)$$

$$\forall x^{\iota} : \{mem(x, Int)\} \qquad \text{(CastSurjective)}$$
$$\quad mem(x, Int) \Rightarrow x =_{\iota} cast_{\mathsf{int}}(proj_{\mathsf{int}}(x))$$

$$\forall z^{\mathsf{int}} : \{cast_{\mathsf{int}}(z)\} \qquad \text{(CastInjective)}$$
$$\quad z =_{\mathsf{int}} proj_{\mathsf{int}}(cast_{\mathsf{int}}(z))$$

$$\forall z_1^{\mathsf{int}}, z_2^{\mathsf{int}} : \{plus(cast_{\mathsf{int}}(z_1), cast_{\mathsf{int}}(z_2))\} \qquad \text{(PlusTyping)}$$
$$\quad plus(cast_{\mathsf{int}}(z_1), cast_{\mathsf{int}}(z_2)) =_{\iota} cast_{\mathsf{int}}(z_1 +_{\mathsf{int}} z_2)$$

Each axiom has a specific use in proofs, which is reflected in the triggers. (CastImage) is used to infer $n \in Int$ for every known integer n. (CastSurjective) is used when $n \in Int$ is known, to produce the term $proj_{\mathsf{int}}(n)$, which is the SMT integer corresponding to n. (CastInjective) generates the equalities that specify $cast_{\mathsf{int}}$ as injective, so computations in ι are reflected in int. (PlusTyping) produces the SMT addition corresponding to a TLA$^+$ addition between two integers, so that computations can be carried in int by the SMT engine.

5.5 Practical Example

To illustrate how triggers work in the context of our encoding, we will use a simple but realistic problem of set theory. Starting from a TLA$^+$ proof obligation and a few axioms with triggers, we describe how SMT's procedure unfolds to solve the problem.

The proof obligation is displayed below on the left. On the right, it is shown as encoded in MS-FOL—as we can see, the structure of the proof obligation is preserved.

ASSUME NEW S,	ASSUME NEW $S : \iota$,
$\quad (S \cap Int) \subseteq \emptyset$	$\quad subseteq(cap(S, Int), empty)$
PROVE $1 \notin S$	PROVE $\neg mem(cast_{\mathsf{int}}(1), S)$

The intuitive proof goes as follows. Assuming $1 \in S$, we derive a contradiction. Of course, $1 \in Int$, therefore $1 \in (S \cap Int)$. But $(S \cap Int) \subseteq \emptyset$, thus $1 \in \emptyset$, which is absurd.

Here are the axioms required for this proof:

$$\forall a^{\iota}, b^{\iota}, x^{\iota} : \{subseteq(a,b), mem(x,a)\} \quad \text{(SubseteqIntro)}$$
$$subseteq(a,b) \wedge mem(x,a) \Rightarrow mem(x,b)$$

$$\forall a^{\iota}, b^{\iota}, x^{\iota} : \{cap(a,b), mem(x,a)\} \quad \text{(CapIntro)}$$
$$mem(x, cap(a,b)) \Leftrightarrow mem(x,a) \wedge mem(x,b)$$

$$\forall x^{\iota} : \{mem(x, empty)\} \quad \text{(EmptyElim)}$$
$$\neg mem(x, empty)$$

$$\forall n^{\mathsf{int}} : \{cast_{\mathsf{int}}(n)\} \quad \text{(CastImage)}$$
$$mem(cast_{\mathsf{int}}(n), Int)$$

Only the axioms important for the proof are shown. Analogously, only the triggers that will play a role are shown. This applies to axiom (CapIntro), which we previously introduced as (CapDef) with more triggers.

Finally, here is the proof obligation encoded as a ground problem:

$$subseteq(cap(S, Int), empty) \quad \text{(E1)}$$
$$mem(cast_{\mathsf{int}}(1), S) \quad (\neg\text{G})$$

We may now follow SMT's procedure step by step. Initially, the ground problem is not found unsatisfiable by the ground solver, so the instantiation module is called to generate instances from the quantified formulas.

There are two matches for triggers, one for the axiom (CapIntro), the other for (CastImage). Let us start with the latter; the matching term is $cast_{\mathsf{int}}(1)$, found in the negated goal ($\neg$G). Applying the substitution $\{n \mapsto 1\}$ to the axiom, SMT's instantiation module generates the ground instance

$$mem(cast_{\mathsf{int}}(1), Int) \quad \text{(E2)}$$

The instance is inserted into the ground problem.

The pair of terms $cap(S, Int)$, $mem(cast_{\mathsf{int}}(1), S)$ are a match for the trigger $\{cap(a,b), mem(x,a)\}$ through the substitution $\{a \mapsto S, b \mapsto Int, x \mapsto cast_{\mathsf{int}}(1)\}$. The instance generated from (CapIntro) is:

$$mem(cast_{\mathsf{int}}(1), cap(S, Int)) \Leftrightarrow mem(cast_{\mathsf{int}}(1), S) \wedge mem(cast_{\mathsf{int}}(1), Int) \quad \text{(E3)}$$

Consider now the trigger of axiom (SubseteqIntro). The pattern $subseteq(a,b)$ has a match in the initial problem: the term $subseteq(cap(S, cast_{\mathsf{int}}(1)), empty)$. With the introduction of (E3), there is now an appropriate match for $mem(x,a)$ in the form of the term $mem(cast_{\mathsf{int}}(1), cap(S, Int))$. The instantiation procedure may then progress with the following instance of (SubseteqIntro):

$$subseteq(cap(S, Int), empty) \wedge mem(cast_{\mathsf{int}}(1), cap(S, Int)) \quad \text{(E4)}$$
$$\Rightarrow mem(cast_{\mathsf{int}}(1), empty)$$

The formula (E4) contains an obvious match for the axiom (EmptyElim). The new instance is:

$$\neg mem(cast_{\mathsf{int}}(1), empty) \tag{E5}$$

The ground problem composed of (E1), (¬G), (E2), (E3), (E4), (E5) is unsatisfiable. Each step of the instantiation phase actually corresponds to a step in the intuitive proof we detailed. When SMT's ground solver is called again, the problem is found unsatisfiable, and the procedure terminates.

6 Evaluation

Our SMT encoding is implemented in TLAPS and available on GitHub.[1] We now present its evaluation. The main purpose of this evaluation is to compare our encoding with the original SMT backend. We are also interested in the impact of our triggers on the performances of SMT.

6.1 Experiment and Results

Our starting data is a collection of TLA$^+$ specifications, taken from three different sources: the TLA$^+$ Examples,[2] the library of examples from the TLAPS distribution, and a recent specification of Lamport's Deconstructed Bakery algorithm [20]. We generated SMT benchmarks by calling TLAPS on all files; since we are interested in comparing different SMT encodings, we generated several SMT benchmarks by calling TLAPS with different parameters. Specifically, we invoked the following encodings:

- The new SMT encoding described in this paper;
- The old SMT encoding with no type reconstruction (T_0);
- The old SMT encoding with elementary type reconstruction (T_1).

T_0 and T_1 refer to the two possible modes for the previous SMT encoding. T_0 uses a simple encoding without type reconstruction, while T_1 uses elementary type reconstruction to optimize the process. There is a third mode T_2 based on a dependent type system, but we decided to discard it after realizing that it is unsound (type-checking conditions that should be verified by an external automated procedure seem to be systematically ignored and assumed to be true).

All proof steps in the original TLA$^+$ files were originally verified by some backend of TLAPS, SMT or other. Included in our benchmarks are all proof obligations except the ones that were solved by LS4. Those proof obligations require temporal logic, which the SMT backend does not support.[3]

[1] https://github.com/tlaplus/tlapm

[2] https://github.com/tlaplus/Examples

[3] The TLA$^+$ specifications and SMT benchmarks used for this evaluation can be found at https://github.com/adef-inr/SafeTLAEncodingBenchmarks

We used the following SMT solvers for the evaluation: CVC4 [5], cvc5 [3], veriT [8] and Z3 [12]. To use veriT, we modified input files by replacing the SMT logic UFNIA by UFLIA, as veriT only supports linear arithmetic. All solvers are called with a timeout of 5 seconds, which is the default value in TLAPS. The experiment was carried out on a Dell Latitude laptop with an Intel Core i7 processor (1.90 GHz).

The results are presented in Table 1 and Table 2. For both tables, we have regrouped specifications according to their sources (TLA$^+$ Examples, TLAPS distribution or Deconstructed Bakery); the last line shows the overall results. The old and new encodings are compared in Table 1. An encoded proof obligation is considered solved if at least one SMT solver managed to solve it. An proof obligation is solved by the old encoding if either the version encoded with T_0 or the one encoded with T_1 was solved. There are two numbers in each cell reporting solved proof obligations: the top number is the number of proof obligations solved by the encoding; the bottom number is the number of proof obligations solved *uniquely* by that encoding.

Table 2 focusses on the new encoding. It details the performances of each SMT solver and the impact of our custom triggers. For each solver, the left column reports the number of proof obligations solved when the solver is asked to ignore our triggers; the right column reports the number of proof obligations solved with triggers enabled. As before, the bottom numbers are the numbers of proof obligations solved uniquely with triggers either disabled or enabled, for that solver. The last column presents the results of a virtual best solver for this evaluation.

Category	Size (# POs)	Encoding	
		Old	New
TLA$^+$ Examples	1996	1688	1821
		56	189
TLAPS Distribution	1402	1281	1333
		44	96
Deconstructed Bakery	1214	1030	1188
		26	184
Total	4612	3999	4342
		126	469

Table 1. Proof obligations (POs) solved by each SMT encoding. Top numbers report how many POs are solved, bottom numbers report how many POs are solved uniquely.

Category	Size	Solver									
		Z3		CVC4		cvc5		veriT		**portfolio**	
TLA$^+$ Ex.	1996	1516	1754	1695	1750	1686	1722	748	995	1745	1821
		21	259	14	69	22	58	18	265	4	80
TLAPS Dist.	1402	1120	1278	1155	1256	1156	1240	510	854	1232	1333
		12	170	4	105	5	89	3	347	0	101
D. Bakery	1214	927	1104	1027	1109	1036	1116	199	236	1093	1188
		17	194	48	130	42	122	5	42	2	97
Total	4612	3563	4136	3877	4115	3878	4078	1457	2085	4070	4342
		50	623	66	304	69	269	26	654	6	278

Table 2. Impact of our custom triggers on SMT solvers. Right numbers: triggers enabled. Left numbers: SMT's default procedure for instantiation. Top numbers are POs solved, bottom numbers are POs solved uniquely.

6.2 Discussion

According to Table 1, out of all 4612 proof obligations, 3999 (86.7%) were solved using the old SMT encoding, and 4342 (94.1%) using the new encoding. Although the new encoding performs as well as the old one, they do not cover exactly the same proof obligations: 469 proof obligations are only solved with the new version, and 126 are not solved anymore. Further investigations into these unsolved 126 proof obligations might reveal flaws in our encoding and show areas of improvement.

One reasonable explanation for the 26 unsolved proof obligations of the *DeconstructedBakery* specification is our treatment of functional extensionality. Our trigger for the axiom generates one instance for every pair of functions in the problem. This becomes problematic in this specification because it involves matrices, which are defined as functions returning functions. For any two matrices M and N in the problem, the generated instance of functional extensionality introduces new terms $M[i]$ and $N[i]$, which are themselves functions. This results in even more instances of functional extensionality, many of which are irrelevant.

As for why our encoding surpasses the previous one in many cases, there are several possible explanations. We noticed that the simplification phase of the old encoding sometimes introduces quantifiers that SMT might struggle to instantiate correctly. For instance, the rewriting rule

$$x \in \text{UNION } a \quad \longrightarrow \quad \exists y^{\iota} : y \in a \wedge x \in y$$

introduces a new quantifier into the problem. The *DeconstructedBakery* specification features the following definition for partial functions:

$$PFunc(A, B) \triangleq \text{UNION } \{[X \to B] : X \in \text{SUBSET } A\}$$

To prove a fact $f \in PFunc(A, B)$, one needs to find the set X such that $f \in [X \to B]$. With the old encoding, this task is left to SMT in the form of a new quantifier to instantiate. It might be difficult to find the relevant instance, and simplification seems to aggravate this in some cases, because relevant terms are rewritten and sometimes eliminated from the problem altogether. In comparison, our encoding steers SMT toward the instance with triggers annotating the axioms of UNION and set replacement. The instance is found if some facts about f are known; for example, if $f \in [X \to B]$ is known, or if f is known to be a function of domain X and the set $[X \to B]$ occurs somewhere.

Many failures of the old encoding are the result of implementation issues. The old encoding did not produce a result for 281 proof obligations with T_0, 451 with T_1. We can speculate that many more proof obligations would be solved by the old encoding if the implementation were fixed. This would account for a portion of the 469 proof obligations solved uniquely by our encoding, and could also explain why our encoding outperforms the old one even without triggers (Table 2).

Let us now focus on Table 2. Our triggers impact the performances of the solvers positively: 278 proof obligations are solved with triggers enabled only. Z3 and veriT benefit the most from our triggers, with respectively 623 and 654 proof obligations solved with them only. By contrast, triggers allow CVC4 and cvc5 to respectively solve 304 and 269 additional proof obligations only. It is also worth noting that each solver has solved a number of proof obligations uniquely without our triggers: 50 for Z3, 66 for CVC4, 69 for cvc5 and 26 for veriT.

The fact that many proof obligations are still solved without using our triggers, and sometimes *only* without them, is not entirely surprising. SMT solvers' default procedures may involve the generation of their own triggers, which may end up similar to ours, or more adapted than ours in particular cases. Unfortunately, we did not inspect automatically generated triggers as part of our evaluation, as examining the inner workings of individual SMT solvers goes beyond the scope of this paper.

7 Related Work

Encodings of set theory for interactive theorem proving have been explored in the context of the B Method [1]. An encoding of B's set theory into SMT-LIB has been implemented by Déharbe for the Rodin platform [14]. Mentré et al developed an encoding for Atelier B which targets the polymorphic first-order logic of Why3 to leverage SMT solvers [6, 24]. A crucial difference between B and TLA$^+$ is that the former is typed while the latter is not. For this reason, it is not possible to implement encodings for TLAPS in a similar way.

The core idea of our encoding is the use of an axiomatic theory with custom triggers. SMT triggers have been studied from a theoretical and practical point of view. Dross et al propose a method for deriving reasoning procedures from theories with triggers, and introduce appropriate notions of termination and completeness [15]. Moskal reports on the integration of Z3 for a C verifier using

triggers [26]. Leino et al describe a strategy for trigger selection in the context of the Dafny verifier [22]. Many ideas for building our own strategy were borrowed from all these works, particularly Leino's. Nonetheless, the application of triggers to a theory as expressive as untyped set theory seems rather new. The idea of only allowing certain symbols to be produced by triggers is derived from our practical experience with typical TLA$^+$ proofs.

8 Conclusion

We presented an encoding of TLA$^+$'s constant fragment into SMT-LIB. Our approach is based on the view that TLA$^+$ is a standard theory on top of a core logic where terms and formulas are not distinguished. Proof obligations are encoded into SMT's logic by first applying a transformation to recover formulas, then inserting declarations and axioms for all relevant TLA$^+$ primitives.

Our encoding faithfully translates expressions of TLA$^+$'s untyped set theory. We argue that our implementation is safer to use than the previous one because it is more lightweight and based on a simpler design. Still, more could be done to increase trust in the new encoding. Most importantly, our axioms are adapted from the official documentation, but they have not been formally verified.

The most innovative part of this encoding is our use of triggers to optimize SMT's instantiation procedure for TLA$^+$. With these heuristics, we achieved performances similar to the previous version of the SMT encoding, which used simplification techniques to eliminate occurrences of TLA$^+$ primitives. This is possible because most TLA$^+$ proof obligations require only elementary reasoning on sets and functions that are explicitly mentioned, and triggers suffice to model this kind of reasoning.

Acknowledgment I thank Jasmin Blanchette, Pascal Fontaine, and Stephan Merz for their support and guidance through the development of this work. This research is funded by the European Research Council (ERC) under the European's Union Horizon 2020 research and innovation program (grant agreement No. 713999, Matryoshka), and by the Région Grand Est.

References

1. Jean-Raymond Abrial. *The B-book - assigning programs to meanings.* Cambridge University Press, 1996.
2. Matthias Baaz, Uwe Egly, and Alexander Leitsch. Normal form transformations. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning (in 2 volumes)*, pages 273–333. Elsevier and MIT Press, 2001.
3. Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. cvc5: A versatile and industrial-strength SMT solver. In Dana Fisman and Grigore Rosu, editors, *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, volume 13243 of *Lecture Notes in Computer Science*, pages 415–442. Springer, 2022.

4. Haniel Barbosa, Andrew Reynolds, Daniel El Ouraoui, Cesare Tinelli, and Clark W. Barrett. Extending SMT solvers to higher-order logic. In Pascal Fontaine, editor, *Automated Deduction - CADE 27 - 27th International Conference on Automated Deduction, Natal, Brazil, August 27-30, 2019, Proceedings*, volume 11716 of *Lecture Notes in Computer Science*, pages 35–54. Springer, 2019.
5. Clark W. Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer, 2011.
6. 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. https://hal.inria.fr/hal-00790310.
7. Richard Bonichon, David Delahaye, and Damien Doligez. Zenon : An extensible automated theorem prover producing checkable proofs. In Nachum Dershowitz and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning, 14th International Conference, LPAR 2007, Yerevan, Armenia, October 15-19, 2007, Proceedings*, volume 4790 of *Lecture Notes in Computer Science*, pages 151–165. Springer, 2007.
8. Thomas Bouton, Diego Caminha Barbosa De Oliveira, David Déharbe, and Pascal Fontaine. veriT: An open, trustable and efficient smt-solver. In Renate A. Schmidt, editor, *Automated Deduction - CADE-22, 22nd International Conference on Automated Deduction, Montreal, Canada, August 2-7, 2009. Proceedings*, volume 5663 of *Lecture Notes in Computer Science*, pages 151–156. Springer, 2009.
9. Denis Cousineau, Damien Doligez, Leslie Lamport, Stephan Merz, Daniel Ricketts, and Hernán Vanzetto. TLA$^+$ proofs. *CoRR*, abs/1208.5933, 2012.
10. Olivier Danvy and Lasse R. Nielsen. Defunctionalization at work. In *Proceedings of the 3rd international ACM SIGPLAN conference on Principles and practice of declarative programming, September 5-7, 2001, Florence, Italy*, pages 162–174. ACM, 2001.
11. Leonardo de Moura and Nikolaj Bjørner. Z3 – A Tutorial. Technical report, Microsoft Research, 2010.
12. Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, pages 337–340, 2008.
13. Rosalie Defourné. Encoding TLA$^+$ proof obligations safely for smt. In Uwe Glässer, Jose Creissac Campos, Dominique Méry, and Philippe Palanque, editors, *Rigorous State-Based Methods*, pages 88–106, Cham, 2023. Springer Nature Switzerland.
14. David Déharbe, Pascal Fontaine, Yoann Guyot, and Laurent Voisin. Integrating SMT solvers in Rodin. *Sci. Comput. Program.*, 94:130–143, 2014.

15. Claire Dross, Sylvain Conchon, Johannes Kanig, and Andrei Paskevich. Reasoning with triggers. In *10th International Workshop on Satisfiability Modulo Theories, SMT 2012, Manchester, UK, June 30 - July 1, 2012*, pages 22–31, 2012.
16. David Gries and Fred B. Schneider. Avoiding the undefined by underspecification. In Jan van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments*, volume 1000 of *Lecture Notes in Computer Science*, pages 366–373. Springer, 1995.
17. Igor Konnov, Jure Kukovec, and Thanh-Hai Tran. TLA$^+$ model checking made symbolic. *Proc. ACM Program. Lang.*, 3(OOPSLA):123:1–123:30, 2019.
18. Leslie Lamport. The temporal logic of actions. *ACM Trans. Program. Lang. Syst.*, 16(3):872–923, 1994.
19. Leslie Lamport. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.
20. Leslie Lamport. Deconstructing the bakery to build a distributed state machine. *Commun. ACM*, 65(9):58–66, 2022.
21. Leslie Lamport and Lawrence C. Paulson. Should your specification language be typed. *ACM Trans. Program. Lang. Syst.*, 21(3):502–526, 1999.
22. K. Rustan M. Leino and Clément Pit-Claudel. Trigger selection strategies to stabilize program verifiers. In *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I*, pages 361–381, 2016.
23. Jia Meng and Lawrence C. Paulson. Translating higher-order clauses to first-order clauses. *J. Autom. Reason.*, 40(1):35–60, 2008.
24. David Mentré, Claude Marché, Jean-Christophe Filliâtre, and Masashi Asuka. Discharging proof obligations from atelier B using multiple automated provers. In John Derrick, John S. Fitzgerald, Stefania Gnesi, Sarfraz Khurshid, Michael Leuschel, Steve Reeves, and Elvinia Riccobene, editors, *Abstract State Machines, Alloy, B, VDM, and Z - Third International Conference, ABZ 2012, Pisa, Italy, June 18-21, 2012. Proceedings*, volume 7316 of *Lecture Notes in Computer Science*, pages 238–251. Springer, 2012.
25. Stephan Merz and Hernán Vanzetto. Encoding TLA$^+$ into unsorted and many-sorted first-order logic. *Sci. Comput. Program.*, 158:3–20, 2018.
26. Michal Moskal. Programming with triggers. *ACM International Conference Proceeding Series*, 01 2009.
27. Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. How Amazon web services uses formal methods. *Commun. ACM*, 58(4):66–73, 2015.
28. Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.
29. Andrew Reynolds. Conflicts, models and heuristics for quantifier instantiation in SMT. In Laura Kovács and Andrei Voronkov, editors, *Vampire@IJCAR 2016. Proceedings of the 3rd Vampire Workshop, Coimbra, Portugal, July 2, 2016*, volume 44 of *EPiC Series in Computing*, pages 1–15. EasyChair, 2016.
30. Andrew Reynolds, Haniel Barbosa, and Pascal Fontaine. Revisiting enumerative instantiation. In Dirk Beyer and Marieke Huisman, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Part II*, volume 10806 of *Lecture Notes in Computer Science*, pages 112–131. Springer, 2018.

31. Martin Suda and Christoph Weidenbach. A pltl-prover based on labelled superposition with partial model guidance. In Bernhard Gramlich, Dale Miller, and Uli Sattler, editors, *Automated Reasoning*, pages 537–543, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.
32. Yuan Yu, Panagiotis Manolios, and Leslie Lamport. Model checking TLA$^+$ specifications. In *Correct Hardware Design and Verification Methods, 10th IFIP WG 10.5 Advanced Research Working Conference, CHARME '99, Bad Herrenalb, Germany, September 27-29, 1999, Proceedings*, pages 54–66, 1999.

A SMT Symbols and Axioms

This section documents the symbols and axioms used for the SMT encoding. Many symbols and axioms are parameterized by some value. We admit the following notational conventions: the letter n denotes a natural number; p denotes a positive natural number; s denotes a string of characters; F, P and Q denote functional symbols. When a symbol or axiom is parameterized by a symbol, the arity of that symbol will be implied by the context. When writing types, the notation ι^n denotes $\iota \times \cdots \times \iota$ where ι occurs n times.

Functional Symbols

The full list of functional symbols that may be inserted in the final SMT problem is displayed below. Symbols are provided with their SMT types. On the right, the correspondence between a TLA$^+$ notation and a MS-FOL term is given, if one exists.

$$
\begin{array}{rll}
\mathit{choose}_P : \iota^n \to \iota & \text{CHOOSE } x : P(x, c_1, \ldots, c_n) & \triangleq \mathit{choose}_P(c_1, \ldots, c_n) \\
\\
\mathit{mem} : \iota \times \iota \to o & x \in y & \triangleq \mathit{mem}(x, y) \\
\mathit{subseteq} : \iota \times \iota \to o & x \subseteq y & \triangleq \mathit{subseteq}(x, y) \\
\mathit{enum}_n : \iota^n \to \iota & \{x_1, \ldots, x_n\} & \triangleq \mathit{enum}_n(x_1, \ldots, x_n) \\
\mathit{union} : \iota \to \iota & \text{UNION } a & \triangleq \mathit{union}(a) \\
\mathit{subset} : \iota \to \iota & \text{SUBSET } a & \triangleq \mathit{subset}(a) \\
\mathit{setst}_P : \iota \times \iota^n \to \iota & \{x \in a : P(x, c_1, \ldots, c_n)\} & \triangleq \mathit{setst}_P(a, c_1, \ldots, c_n) \\
\mathit{setof}_{p,F} : \iota^p \times \iota^n \to \iota & \{F(x_1, \ldots, x_p, c_1, \ldots, c_n) : x_1 \in a_1, \ldots, x_p \in a_p\} & \\
 & & \triangleq \mathit{setof}_{p,F}(a_1, \ldots, a_p, c_1, \ldots, c_n) \\
\mathit{cup} : \iota \times \iota \to \iota & a \cup b & \triangleq \mathit{cup}(a, b) \\
\mathit{cap} : \iota \times \iota \to \iota & a \cap b & \triangleq \mathit{cap}(a, b) \\
\mathit{diff} : \iota \times \iota \to \iota & a \setminus b & \triangleq \mathit{diff}(a, b) \\
\mathit{Boolean} : \iota & \text{BOOLEAN} & \triangleq \mathit{Boolean} \\
\mathit{cast}_o : o \to \iota & & \\
\\
\mathit{isafcn} : \iota \to o & & \\
\mathit{fcn}_F : \iota \times \iota^n \to \iota & [x \in a \mapsto F(x, c_1, \ldots, c_n)] & \triangleq \mathit{fcn}_F(a, c_1, \ldots, c_n) \\
\mathit{except} : \iota \times \iota \times \iota \to \iota & [f \text{ EXCEPT } ![x] = y] & \triangleq \mathit{except}(f, x, y) \\
\mathit{domain} : \iota \to \iota & \text{DOMAIN } f & \triangleq \mathit{domain}(f) \\
\mathit{fcnapp} : \iota \times \iota \to \iota & f[x] & \triangleq \mathit{fcnapp}(f, x)
\end{array}
$$

$$
\begin{array}{rll}
arrow : \iota \times \iota \to \iota & & [a \to b] \triangleq arrow(a, b) \\
\\
cast_{\mathsf{int}} : \mathsf{int} \to \iota & & \\
proj_{\mathsf{int}} : \iota \to \iota & & \\
Int : \iota & & \\
Nat : \iota & & \\
plus : \iota \times \iota \to \iota & & x + y \triangleq plus(x, y) \\
uminus : \iota \to \iota & & -x \triangleq uminus(x) \\
minus : \iota \times \iota \to \iota & & x - y \triangleq minus(x, y) \\
times : \iota \times \iota \to \iota & & x \times y \triangleq times(x, y) \\
quotient : \iota \times \iota \to \iota & & x \iota y \triangleq quotient(x, y) \\
remainder : \iota \times \iota \to \iota & & x \,\%\, y \triangleq remainder(x, y) \\
lteq : \iota \times \iota \to o & & x \leq y \triangleq lteq(x, y) \\
 & & x < y \triangleq lteq(x, y) \wedge x \neq y \\
 & & x \geq y \triangleq lteq(y, x) \\
 & & x > y \triangleq lteq(y, x) \wedge x \neq y \\
range : \iota \times \iota \to \iota & & x \,..\, y \triangleq range(x, y) \\
\\
tup_n : \iota^n \to \iota & & \langle x_1, \ldots, x_n \rangle \triangleq tup_n(x_1, \ldots, x_n) \\
product_p : \iota^p \to \iota & & a_1 \times \cdots \times a_p \triangleq product_p(a_1, \ldots, a_p) \\
\\
str_s : \iota & & \text{“}\texttt{foo}\text{”} \triangleq str_{foo} \\
String : \iota & & \textsc{string} String \\
\\
record_{s_1,\ldots,s_p} : \iota^p \to \iota & & [s_1 \mapsto x_1, \ldots, s_p \mapsto x_p] \triangleq record_{s_1,\ldots,s_p}(x_1, \ldots, x_p) \\
rect_{s_1,\ldots,s_p} : \iota^p \to \iota & & [s_1 : a1, \ldots, s_p : a_p] \triangleq rect s_1, \ldots, s_p(a_1, \ldots, a_p) \\
\\
Seq : \iota \to \iota & & \\
Len : \iota \to \iota & & \\
Cat : \iota \times \iota \to \iota & & \\
Append : \iota \times \iota \to \iota & & \\
Head : \iota \to \iota & & \\
Tail : \iota \to \iota & & \\
Subseq : \iota \times \iota \times \iota \to \iota & & \\
Selectseq_P : \iota \times \iota^n \to \iota & & Selectseq(s, \lambda x : P(x, c_1, \ldots, c_n)) \\
 & & \quad \triangleq Selectseq_P(s, c_1, \ldots, c_n)
\end{array}
$$

Hilbert's Choice

$$\forall c_1^\iota, \ldots, c_n^\iota, x^\iota : \qquad \text{(ChooseDef)}$$
$$P(x, c_1, \ldots, c_n) \Rightarrow P(choose_P(c_1, \ldots, c_n), c_1, \ldots, c_n)$$

$$\forall c_1^\iota, \ldots, c_n^\iota : \qquad \text{(ChooseExt)}$$
$$\left[\forall x^\iota : P(x, c_1, \ldots, c_n) \Leftrightarrow Q(x, c_1, \ldots, c_n)\right]$$
$$\Rightarrow choose_P(c_1, \ldots, c_n) = choose_Q(c_1, \ldots, c_n)$$

Set Theory

$$\forall a^\iota, b^\iota : \{subseteq(a, b)\} \qquad \text{(SubseteqIntro)}$$
$$\left[\forall x^\iota : mem(x, a) \Leftrightarrow mem(x, b)\right] \Rightarrow subseteq(a, b)$$

$$\forall a^\iota, b^\iota, x^\iota : \{subseteq(a, b), mem(x, a)\} \qquad \text{(SubseteqElim)}$$
$$subseteq(a, b) \wedge mem(x, a) \Rightarrow mem(x, b)$$

$$\forall a_1^\iota, \ldots, a_p^\iota : \{enum_p(a_1, \ldots, a_p)\} \qquad \text{(EnumIntro)}$$
$$mem(a_1, enum_p(a_1, \ldots, a_p)) \wedge \cdots \wedge mem(a_p, enum_p(a_1, \ldots, a_p))$$

$$\forall a_1^\iota, \ldots, a_p^\iota, x^\iota : \{mem(x, enum_p(a_1, \ldots, a_p))\} \qquad \text{(EnumElim)}$$
$$mem(x, enum_p(a_1, \ldots, a_p)) \Rightarrow x = a_1 \vee \cdots \vee x = a_p$$

$$\forall x^\iota : \{mem(x, enum_0)\} \qquad \text{(EmptyElim)}$$
$$\neg mem(x, enum_0)$$

$$\forall a^\iota, x^\iota : \{mem(x, subset(a))\} \qquad \text{(SubsetDef)}$$
$$\{subseteq(x, a), subset(a)\}$$
$$mem(x, subset(a)) \Leftrightarrow subseteq(x, a)$$

$$\forall a^\iota, x^\iota, y^\iota : \{mem(y, a), mem(x, union(a))\} \qquad \text{(UnionIntro)}$$
$$\{mem(x, y), mem(x, union(a))\}$$
$$\{mem(x, y), mem(y, a), union(a)\}$$
$$mem(x, y) \wedge mem(y, a) \Rightarrow mem(x, union(a))$$

$$\forall a^\iota, x^\iota : \{mem(x, union(a))\} \qquad \text{(UnionElim)}$$
$$mem(x, union(a)) \Rightarrow \exists y^\iota : mem(x, y) \wedge mem(y, a)$$

$$\forall c_1^\iota, \ldots, c_n^\iota, a^\iota, x^\iota : \{mem(x, setst_P(a, c_1, \ldots, c_n))\} \qquad \text{(SetstDef)}$$
$$\{mem(x, a), setst_P(a, c_1, \ldots, c_n)\}$$
$$mem(x, setst_P(a, c_1, \ldots, c_n)) \Leftrightarrow mem(x, a) \wedge P(x, c_1, \ldots, c_n)$$

$$\forall c_1^\iota, \ldots, c_n^\iota, a_1^\iota, \ldots, a_p^\iota, y_1^\iota, \ldots, y_p^\iota : \qquad \text{(SetofIntro)}$$
$$\{F(y_1, \ldots, y_p, c_1, \ldots, c_n), setof_F(a_1, \ldots, a_p, c_1, \ldots, c_n)\}$$
$$\{mem(y_1, a_1), \ldots, mem(y_p, a_p), setof_F(a_1, \ldots, a_p, c_1, \ldots, c_n)\}$$
$$mem(y_1, a_1) \wedge \cdots \wedge mem(y_p, a_p)$$
$$\Rightarrow mem(F(y_1, \ldots, y_p, c_1, \ldots, c_n), setof_F(a_1, \ldots, a_p, c_1, \ldots, c_n))$$

$$\forall c_1^\iota, \ldots, c_n^\iota, a_1^\iota, \ldots, a_p^\iota, x^\iota : \qquad \text{(SetofElim)}$$
$$\{mem(x, setof_F(a_1, \ldots, a_p, c_1, \ldots, c_n))\}$$
$$mem(x, setof_F(a_1, \ldots, a_p, c_1, \ldots, c_n))$$
$$\Rightarrow \exists y_1^\iota, \ldots, y_p^\iota : mem(y_1, a_1) \wedge \cdots \wedge mem(y_p, a_p) \wedge x = F(y_1, \ldots, y_p)$$

$$\forall a^\iota, b^\iota, x^\iota : \{mem(x, cup(a, b))\} \qquad \text{(CupDef)}$$
$$\{mem(x, a), cup(a, b)\}$$
$$\{mem(x, b), cup(a, b)\}$$
$$mem(x, cup(a, b)) \Leftrightarrow mem(x, a) \vee mem(x, b)$$

$$\forall a^\iota, b^\iota, x^\iota : \{mem(x, cap(a, b))\} \qquad \text{(CapDef)}$$
$$\{mem(x, a), cap(a, b)\}$$
$$\{mem(x, b), cap(a, b)\}$$
$$mem(x, cap(a, b)) \Leftrightarrow mem(x, a) \wedge mem(x, b)$$

$$\forall a^\iota, b^\iota, x^\iota : \{mem(x, diff(a, b))\} \qquad \text{(SetminusDef)}$$
$$\{mem(x, a), diff(a, b)\}$$
$$\{mem(x, b), diff(a, b)\}$$
$$mem(x, diff(a, b)) \Leftrightarrow mem(x, a) \wedge \neg mem(x, b)$$

$$cast_o(\text{TRUE}) \neq cast_o(\text{FALSE}) \qquad \text{(BoolCastInj)}$$

$$mem(cast_o(\text{TRUE}), Boolean) \wedge mem(cast_o(\text{FALSE}), Boolean) \qquad \text{(BooleanIntro)}$$

$$\forall x^\iota : \{mem(x, Boolean)\} \qquad \text{(BooleanElim)}$$
$$mem(x, Boolean) \Rightarrow x = cast_o(\text{TRUE}) \vee x = cast_o(\text{FALSE})$$

Functions

$$\forall f^\iota, g^\iota : \{isafcn(f), isafcn(g)\} \qquad \text{(FcnExt)}$$
$$\wedge\ isafcn(f) \wedge isafcn(g)$$
$$\wedge\ domain(f) = domain(g)$$
$$\wedge\ \left[\forall x^\iota : mem(x, domain(f)) \Rightarrow fcnapp(f, x) = fcnapp(g, x)\right]$$
$$\Rightarrow f = g$$

$$\forall c_1^\iota, \ldots, c_n^\iota, a^\iota : \{fcn_F(a, c_1, \ldots, c_n)\} \quad \text{(FcnIsafcn)}$$
$$isafcn(fcn_F(a, c_1, \ldots, c_n))$$

$$\forall c_1^\iota, \ldots, c_n^\iota, a^\iota : \{fcn_F(a, c_1, \ldots, c_n)\} \quad \text{(FcnDom)}$$
$$domain(fcn_F(a, c_1, \ldots, c_n)) = a$$

$$\forall c_1^\iota, \ldots, c_n^\iota, a^\iota, x^\iota : \{fcnapp(fcn_F(a, c_1, \ldots, c_n), x)\} \quad \text{(FcnApp)}$$
$$\{mem(x, a), fcn_F(a, c_1, \ldots, c_n)\}$$
$$mem(x, a) \Rightarrow fcnapp(fcn_F(a, c_1, \ldots, c_n), x) = F(x, c_1, \ldots, c_n)$$

$$\forall c_1^\iota, \ldots, c_n^\iota, a^\iota, b^\iota : \{fcn_F(a, c_1, \ldots, c_n), arrow(a, b)\} \quad \text{(FcnTyping)}$$
$$\left[\forall x^\iota : mem(x, a) \Rightarrow mem(F(x, c_1, \ldots, c_n), b)\right]$$
$$\Rightarrow mem(fcn_F(a, c_1, \ldots, c_n), arrow(a, b))$$

$$\forall a^\iota, b^\iota, f^\iota : \{mem(f, arrow(a, b))\} \quad \text{(ArrowIntro)}$$
$$\wedge\ isafcn(f)$$
$$\wedge\ domain(f) = a$$
$$\wedge \left[\forall x^\iota : mem(x, a) \Rightarrow mem(fcnapp(f, x), b)\right]$$
$$\Rightarrow mem(f, arrow(a, b))$$

$$\forall a^\iota, b^\iota, f^\iota : \{mem(f, arrow(a, b))\} \quad \text{(ArrowElim}_1\text{)}$$
$$mem(f, arrow(a, b)) \Rightarrow\ \wedge\ isafcn(f)$$
$$\wedge\ domain(f) = a$$

$$\forall a^\iota, b^\iota, f^\iota, x^\iota : \{mem(f, arrow(a, b)), mem(x, a)\} \quad \text{(ArrowElim}_2\text{)}$$
$$\{mem(f, arrow(a, b)), fcnapp(f, x)\}$$
$$mem(f, arrow(a, b)) \wedge mem(x, a) \Rightarrow mem(fcnapp(f, x), b)$$

$$\forall f^\iota, x^\iota, y^\iota : \{except(f, x, y)\} \quad \text{(ExceptIsafcn)}$$
$$isafcn(except(f, x, y))$$

$$\forall f^\iota, x^\iota, y^\iota : \{except(f, x, y)\} \quad \text{(ExceptDom)}$$
$$domain(except(f, x, y)) = domain(f)$$

$$\forall f^\iota, x^\iota, y^\iota : \{except(f, x, y)\} \quad \text{(ExceptApp}_1\text{)}$$
$$mem(x, domain(f)) \Rightarrow fcnapp(except(f, x, y), x) = y$$

$$\forall f^\iota, x^\iota, y^\iota, z^\iota : \{fcnapp(except(f, x, y), z)\} \quad \text{(ExceptApp}_2\text{)}$$
$$\{except(f, x, y), fcnapp(f, z)\}$$
$$mem(z, domain(f)) \wedge z \neq x \Rightarrow fcnapp(except(f, x, y), z) = fcnapp(f, z)$$

$$\forall f^{\iota}, x^{\iota}, y^{\iota}, a^{\iota}, b^{\iota} : \quad \text{(ExceptTyping)}$$
$$\{except(f, x, y), mem(f, arrow(a, b))\}$$
$$mem(f, arrow(a, b)) \wedge \left[mem(x, a) \Rightarrow mem(y, b)\right]$$
$$\Rightarrow mem(except(f, x, y), arrow(a, b))$$

Arithmetic

$$\forall z^{\mathsf{int}} : \{cast_{\mathsf{int}}(z)\} \quad \text{(IntCastInjective)}$$
$$z = proj_{\mathsf{int}}(cast_{\mathsf{int}}(z))$$

$$\forall z^{\mathsf{int}} : \{cast_{\mathsf{int}}(z)\} \quad \text{(IntIntro)}$$
$$mem(cast_{\mathsf{int}}(z), Int)$$

$$\forall x^{\iota} : \{mem(x, Int)\} \quad \text{(IntElim)}$$
$$mem(x, Int) \Rightarrow x = cast_{\mathsf{int}}(proj_{\mathsf{int}}(x))$$

$$\forall z^{\mathsf{int}} : \{cast_{\mathsf{int}}(z)\} \quad \text{(NatIntro)}$$
$$z \geq 0 \Rightarrow mem(cast_{\mathsf{int}}(z), Nat)$$

$$\forall x^{\iota} : \{mem(x, Nat)\} \quad \text{(NatElim)}$$
$$mem(x, Nat) \Rightarrow x = cast_{\mathsf{int}}(proj_{\mathsf{int}}(x)) \wedge proj_{\mathsf{int}}(x) \geq 0$$

$$\forall z_1^{\mathsf{int}}, z_2^{\mathsf{int}} : \{plus(cast_{\mathsf{int}}(z_1), cast_{\mathsf{int}}(z_2))\} \quad \text{(PlusTyping)}$$
$$plus(cast_{\mathsf{int}}(z_1), cast_{\mathsf{int}}(z_2)) = cast_{\mathsf{int}}(z_1 +_{\mathsf{int}} z_2)$$

$$\forall z^{\mathsf{int}} : \{uminus(cast_{\mathsf{int}}(z))\} \quad \text{(UminusTyping)}$$
$$uminus(cast_{\mathsf{int}}(z)) = cast_{\mathsf{int}}(-_{\mathsf{int}} z)$$

$$\forall z_1^{\mathsf{int}}, z_2^{\mathsf{int}} : \{minus(cast_{\mathsf{int}}(z_1), cast_{\mathsf{int}}(z_2))\} \quad \text{(MinusTyping)}$$
$$minus(cast_{\mathsf{int}}(z_1), cast_{\mathsf{int}}(z_2)) = cast_{\mathsf{int}}(z_1 -_{\mathsf{int}} z_2)$$

$$\forall z_1^{\mathsf{int}}, z_2^{\mathsf{int}} : \{times(cast_{\mathsf{int}}(z_1), cast_{\mathsf{int}}(z_2))\} \quad \text{(MultTyping)}$$
$$times(cast_{\mathsf{int}}(z_1), cast_{\mathsf{int}}(z_2)) = cast_{\mathsf{int}}(z_1 \times_{\mathsf{int}} z_2)$$

$$\forall z_1^{\mathsf{int}}, z_2^{\mathsf{int}} : \{quotient(cast_{\mathsf{int}}(z_1), cast_{\mathsf{int}}(z_2))\} \quad \text{(QuotientTyping)}$$
$$z_2 > 0 \Rightarrow quotient(cast_{\mathsf{int}}(z_1), cast_{\mathsf{int}}(z_2)) = cast_{\mathsf{int}}(z_1 \div_{\mathsf{int}} z_2)$$

$$\forall z_1^{\mathsf{int}}, z_2^{\mathsf{int}} : \{remainder(cast_{\mathsf{int}}(z_1), cast_{\mathsf{int}}(z_2))\} \quad \text{(RemainderTyping)}$$
$$z_2 > 0 \Rightarrow remainder(cast_{\mathsf{int}}(z_1), cast_{\mathsf{int}}(z_2)) = cast_{\mathsf{int}}(z_1 \%_{\mathsf{int}} z_2)$$

$$\forall z_1^{\mathsf{int}}, z_2^{\mathsf{int}} : \{lteq(cast_{\mathsf{int}}(z_1), cast_{\mathsf{int}}(z_2))\} \quad \text{(LteqTyping)}$$
$$lteq(cast_{\mathsf{int}}(z_1), cast_{\mathsf{int}}(z_2)) \Leftrightarrow z_1 \leq_{\mathsf{int}} z_2$$

$$\forall a^{\iota}, b^{\iota}, z^{\mathsf{int}} : \{mem(cast_{\mathsf{int}}(z), range(a, b))\} \quad \text{(RangeIntro)}$$
$$lteq(a, cast_{\mathsf{int}}(z)) \wedge lteq(cast_{\mathsf{int}}(z), b) \Rightarrow mem(cast_{\mathsf{int}}(z), range(a, b))$$

$$\forall a^{\iota}, b^{\iota}, x^{\iota} : \{mem(x, range(a, b))\} \quad \text{(RangeElim)}$$
$$mem(x, range(a, b)) \Rightarrow x = cast_{\mathsf{int}}(proj_{\mathsf{int}}(x)) \wedge lteq(a, x) \wedge lteq(x, b)$$

Tuples

$$\forall x_1^{\iota}, \ldots, x_n^{\iota} : \{tup_n(x_1, \ldots, x_n)\} \quad \text{(TupIsafcn)}$$
$$isafcn(tup_n(x_1, \ldots, x_n))$$

$$\forall x_1^{\iota}, \ldots, x_n^{\iota} : \{tup_n(x_1, \ldots, x_n)\} \quad \text{(TupDom)}$$
$$domain(tup_n(x_1, \ldots, x_n)) = enum_n(cast_{\mathsf{int}}(1), \ldots, cast_{\mathsf{int}}(n))$$

$$\forall x_1^{\iota}, \ldots, x_p^{\iota} : \{tup_p(x_1, \ldots, x_p)\} \quad \text{(TupApp)}$$
$$\wedge\, fcnapp(tup_p(x_1, \ldots, x_p), cast_{\mathsf{int}}(1)) = x_1$$
$$\wedge \ldots$$
$$\wedge\, fcnapp(tup_p(x_1, \ldots, x_p), cast_{\mathsf{int}}(p)) = x_p$$

$$\forall x_1^{\iota}, \ldots, x_p^{\iota}, x^{\iota} : \{except(tup_p(x_1, \ldots, x_p), cast_{\mathsf{int}}(p'), x)\} \quad \text{(TupExcept)}$$
$$except(tup_p(x_1, \ldots, x_{p'}, \ldots, x_p), cast_{\mathsf{int}}(p'), x) = tup_p(x_1, \ldots, x, \ldots, x_p)$$

$$\forall a_1^{\iota}, \ldots, a_p^{\iota}, x_1^{\iota}, \ldots, x_p^{\iota} : \{tup_p(x_1, \ldots, x_p), product_p(a_1, \ldots, a_p)\} \quad \text{(ProdIntro)}$$
$$mem(x_1, a_1) \wedge \cdots \wedge mem(x_p, a_p)$$
$$\Rightarrow mem(tup_p(x_1, \ldots, x_p), product_p(a_1, \ldots, a_p))$$

$$\forall a_1^{\iota}, \ldots, a_p^{\iota}, x^{\iota} : \{mem(x, product_p(a_1, \ldots, a_p))\} \quad \text{(ProdElim)}$$
$$mem(x, product_p(a_1, \ldots, a_p))$$
$$\Rightarrow x = tup_p(fcnapp(x, cast_{\mathsf{int}}(1)), \ldots, fcnapp(x, cast_{\mathsf{int}}(p)))$$

Strings

$$mem(str_s, String) \quad \text{(StringIntro)}$$

For distinct strings s_1 and s_2:

$$str_{s_1} \neq str_{s_2} \quad \text{(StringsDistinct)}$$

Records

$$\forall x_1^\iota, \ldots, x_p^\iota : \left\{ record_{s_1,\ldots,s_p}(x_1, \ldots, x_p) \right\} \quad \text{(RecordIsafcn)}$$
$$isafcn(record_{s_1,\ldots,s_p}(x_1, \ldots, x_p))$$

$$\forall x_1^\iota, \ldots, x_p^\iota : \left\{ record_{s_1,\ldots,s_p}(x_1, \ldots, x_p) \right\} \quad \text{(RecordDom)}$$
$$domain(record_{s_1,\ldots,s_p}(x_1, \ldots, x_p)) = enum_p(str_{s_1}, \ldots, str_{s_p})$$

$$\forall x_1^\iota, \ldots, x_p^\iota : \left\{ record_{s_1,\ldots,s_p}(x_1, \ldots, x_p) \right\} \quad \text{(RecordApp)}$$
$$\begin{aligned} &\wedge fcnapp(record_{s_1,\ldots,s_p}(x_1, \ldots, x_p), str_{s_1}) = x_1 \\ &\wedge \ldots \\ &\wedge fcnapp(record_{s_1,\ldots,s_p}(x_1, \ldots, x_p), str_{s_p}) = x_p \end{aligned}$$

$$\forall x_1^\iota, \ldots, x_p^\iota, x^\iota : \left\{ except(record_{s_1,\ldots,s_p}(x_1, \ldots, x_p), str_{s_{p'}}, x) \right\} \quad \text{(RecordExcept)}$$
$$\begin{aligned} &except(record_{s_1,\ldots,s_p}(x_1, \ldots, x_{p'}, \ldots, x_p), str_{s_{p'}}, x) \\ &\quad = record_{s_1,\ldots,s_p}(x_1, \ldots, x, \ldots, x_p) \end{aligned}$$

$$\forall a_1^\iota, \ldots, a_p^\iota, x_1^\iota, \ldots, x_p^\iota : \left\{ record_{s_1,\ldots,s_p}(x_1, \ldots, x_p), rect_{s_1,\ldots,s_p}(a_1, \ldots, a_p) \right\} \quad \text{(RectIntro)}$$
$$\begin{aligned} &mem(x_1, a_1) \wedge \cdots \wedge mem(x_p, a_p) \\ &\Rightarrow mem(record_{s_1,\ldots,s_p}(x_1, \ldots, x_p), rect_{s_1,\ldots,s_p}(a_1, \ldots, a_p)) \end{aligned}$$

$$\forall a_1^\iota, \ldots, a_p^\iota, x^\iota : \left\{ mem(x, rect_{s_1,\ldots,s_p}(a_1, \ldots, a_p)) \right\} \quad \text{(RectElim)}$$
$$\begin{aligned} &mem(x, rect_{s_1,\ldots,s_p}(a_1, \ldots, a_p)) \\ &\Rightarrow x = record_{s_1,\ldots,s_p}(fcnapp(x, str_{s_1}), \ldots, fcnapp(x, str_{s_p})) \end{aligned}$$

Sequences

$$\forall a^\iota, s^\iota : \{ mem(s, Seq(a)) \} \quad \text{(SetIntro)}$$
$$\begin{aligned} &\wedge isafcn(s) \\ &\wedge mem(Len(s), Nat) \\ &\wedge \left[\forall i^\iota : mem(i, domain(s)) \Leftrightarrow \begin{array}{l} \wedge mem(i, Int) \\ \wedge 1 \leq_{\mathsf{int}} proj_{\mathsf{int}}(i) \\ \wedge proj_{\mathsf{int}}(i) \leq_{\mathsf{int}} proj_{\mathsf{int}}(Len(s)) \end{array} \right] \\ &\wedge \left[\forall i^{\mathsf{int}} : 1 \leq_{\mathsf{int}} i \wedge i \leq_{\mathsf{int}} proj_{\mathsf{int}}(Len(s)) \Rightarrow mem(fcnapp(s, cast_{\mathsf{int}}(i)), a) \right] \\ &\Rightarrow mem(s, Seq(a)) \end{aligned}$$

$$\forall a^{\iota}, s^{\iota} : \{mem(s, Seq(a))\} \qquad \text{(SetElim}_1\text{)}$$
$$mem(s, Seq(a)) \Rightarrow \ \wedge\, isafcn(s)$$
$$\wedge\, mem(Len(s), Nat)$$
$$\wedge\, domain(s) = range(cast_{\mathsf{int}}(1), Len(s))$$

$$\forall a^{\iota}, s^{\iota}, i^{\mathsf{int}} : \{mem(s, Seq(a)), fcnapp(s, cast_{\mathsf{int}}(i))\} \qquad \text{(SeqElim}_2\text{)}$$
$$mem(s, Seq(a)) \wedge 1 \leq_{\mathsf{int}} i \wedge i \leq_{\mathsf{int}} proj_{\mathsf{int}}(Len(s))$$
$$\Rightarrow mem(fcnapp(s, cast_{\mathsf{int}}(i)), a)$$

$$\forall a^{\iota}, s^{\iota}, t^{\iota} : \{mem(s, Seq(a)), Cat(s, t)\} \qquad \text{(CatTyping)}$$
$$\{mem(t, Seq(a)), Cat(s, t)\}$$
$$mem(s, Seq(a)) \wedge mem(t, Seq(a)) \Rightarrow mem(Cat(s, t), Seq(a))$$

$$\forall s^{\iota}, t^{\iota} : \{Cat(s, t)\} \qquad \text{(CatLen)}$$
$$mem(Len(s), Nat) \wedge mem(Len(t), Nat)$$
$$\Rightarrow Len(Cat(s, t)) = cast_{\mathsf{int}}(proj_{\mathsf{int}}(Len(s)) +_{\mathsf{int}} proj_{\mathsf{int}}(Len(t)))$$

$$\forall s^{\iota}, t^{\iota}, i^{\mathsf{int}} : \{fcnapp(Cat(s, t), cast_{\mathsf{int}}(i))\} \qquad \text{(CapApp}_1\text{)}$$
$$\{Cat(s, t), fcnapp(s, cast_{\mathsf{int}}(i))\}$$
$$\wedge\, mem(Len(s), Nat) \wedge mem(Len(t), Nat)$$
$$\wedge\, 1 \leq_{\mathsf{int}} i \wedge i \leq_{\mathsf{int}} proj_{\mathsf{int}}(Len(s))$$
$$\Rightarrow fcnapp(Cat(s, t), cast_{\mathsf{int}}(i)) = fcnapp(s, cast_{\mathsf{int}}(i))$$

$$\forall s^{\iota}, t^{\iota}, i^{\mathsf{int}} : \{fcnapp(Cat(s, t), cast_{\mathsf{int}}(i))\} \qquad \text{(CapApp}_2\text{)}$$
$$\wedge\, mem(Len(s), Nat) \wedge mem(Len(t), Nat)$$
$$\wedge\, i \leq_{\mathsf{int}} proj_{\mathsf{int}}(Len(s)) +_{\mathsf{int}} proj_{\mathsf{int}}(Len(t)) \wedge proj_{\mathsf{int}}(Len(s)) < i$$
$$\Rightarrow fcnapp(Cat(s, t), cast_{\mathsf{int}}(i)) = fcnapp(t, cast_{\mathsf{int}}(i -_{\mathsf{int}} proj_{\mathsf{int}}(Len(s))))$$

$$\forall a^{\iota}, s^{\iota}, x^{\iota} : \{mem(s, Seq(a)), Append(s, x)\} \qquad \text{(AppendTyping)}$$
$$mem(s, Seq(a)) \wedge mem(x, a) \Rightarrow mem(Append(s, x), Seq(a))$$

$$\forall s^{\iota}, x^{\iota} : \{Append(s, x)\} \qquad \text{(AppendLen)}$$
$$mem(Len(s), Nat)$$
$$\Rightarrow Len(Append(s, x)) = cast_{\mathsf{int}}(proj_{\mathsf{int}}(Len(s)) +_{\mathsf{int}} 1)$$

$$\forall s^{\iota}, x^{\iota}, i^{\mathsf{int}} : \{fcnapp(Append(s, x), cast_{\mathsf{int}}(i))\} \qquad \text{(AppendApp}_1\text{)}$$
$$\{Append(s, x), fcnapp(s, cast_{\mathsf{int}}(i))\}$$
$$\wedge\, mem(Len(s), Nat)$$
$$\wedge\, 1 \leq_{\mathsf{int}} i \wedge i \leq_{\mathsf{int}} proj_{\mathsf{int}}(Len(s))$$
$$\Rightarrow fcnapp(Append(s, x), cast_{\mathsf{int}}(i)) = fcnapp(s, cast_{\mathsf{int}}(i))$$

$$\forall s^{\iota}, x^{\iota} : \{Append(s, x)\} \qquad \text{(AppendApp}_2\text{)}$$
$$mem(Len(s), Nat)$$
$$\Rightarrow fcnapp(Append(s, x), cast_{\mathsf{int}}(proj_{\mathsf{int}}(Len(s)) +_{\mathsf{int}} 1)) = x$$

$$\forall s^{\iota} : \{Head(s)\} \qquad \text{(HeadDef)}$$
$$Head(s) = fcnapp(s, cast_{\mathsf{int}}(1))$$

$$\forall a^{\iota}, s^{\iota} : \{mem(s, Seq(a)), Tail(s)\} \qquad \text{(TailTyping)}$$
$$\wedge\ mem(s, Seq(a))$$
$$\wedge\ proj_{\mathsf{int}}(Len(s)) \neq 0$$
$$\Rightarrow mem(Tail(s), Seq(a))$$

$$\forall s^{\iota} : \{Tail(s)\} \qquad \text{(TailLen)}$$
$$\wedge\ mem(Len(s), Nat)$$
$$\wedge\ proj_{\mathsf{int}}(Len(s)) \neq 0$$
$$\Rightarrow Len(Tail(s)) = cast_{\mathsf{int}}(proj_{\mathsf{int}}(Len(s)) -_{\mathsf{int}} 1)$$

$$\forall s^{\iota}, i^{\mathsf{int}} : \{fcnapp(Tail(s), cast_{\mathsf{int}}(i))\} \qquad \text{(TailApp)}$$
$$\wedge\ mem(Len(s), Nat)$$
$$\wedge\ proj_{\mathsf{int}}(Len(s)) \neq 0$$
$$\wedge\ 1 \leq_{\mathsf{int}} i \wedge i \leq_{\mathsf{int}} proj_{\mathsf{int}}(Len(s)) -_{\mathsf{int}} 1$$
$$\Rightarrow fcnapp(Tail(s), cast_{\mathsf{int}}(i)) = fcnapp(s, cast_{\mathsf{int}}(i +_{\mathsf{int}} 1))$$

$$\forall a^{\iota}, s^{\iota}, x^{\mathsf{int}}, y^{\mathsf{int}} : \qquad \text{(SubseteqTyping)}$$
$$\{mem(s, Seq(a)), Subseq(s, cast_{\mathsf{int}}(x), cast_{\mathsf{int}}(y))\}$$
$$\wedge\ mem(s, Seq(a))$$
$$\wedge\ 1 \leq_{\mathsf{int}} x$$
$$\wedge\ y \leq_{\mathsf{int}} proj_{\mathsf{int}}(Len(s))$$
$$\Rightarrow mem(Subseq(s, cast_{\mathsf{int}}(x), cast_{\mathsf{int}}(y)), Seq(a))$$

$$\forall s^{\iota}, x^{\mathsf{int}}, y^{\mathsf{int}} : \{Subseq(s, cast_{\mathsf{int}}(x), cast_{\mathsf{int}}(y))\} \qquad \text{(SubseqLen)}$$
$$\wedge\ x \leq_{\mathsf{int}} y +_{\mathsf{int}} 1$$
$$\Rightarrow Len(Subseq(s, cast_{\mathsf{int}}(x), cast_{\mathsf{int}}(y))) = cast_{\mathsf{int}}((y +_{\mathsf{int}} 1) -_{\mathsf{int}} x)$$
$$\wedge\ x > y +_{\mathsf{int}} 1 \Rightarrow Len(Subseq(s, cast_{\mathsf{int}}(x), cast_{\mathsf{int}}(y))) = cast_{\mathsf{int}}(0)$$

$$\forall s^{\iota}, x^{\mathsf{int}}, y^{\mathsf{int}}, z^{\mathsf{int}} : \qquad \text{(SubseqApp)}$$
$$\{fcnapp(Subseq(s, cast_{\mathsf{int}}(x), cast_{\mathsf{int}}(y)), cast_{\mathsf{int}}(z))\}$$
$$\wedge\ 1 \leq_{\mathsf{int}} x$$
$$\wedge\ 1 \leq_{\mathsf{int}} z \wedge z \leq_{\mathsf{int}} (y +_{\mathsf{int}} 1) -_{\mathsf{int}} x$$
$$\Rightarrow fcnapp(Subseq(s, cast_{\mathsf{int}}(x), cast_{\mathsf{int}}(y)), cast_{\mathsf{int}}(z))$$
$$= fcnapp(s, cast_{\mathsf{int}}((z +_{\mathsf{int}} x) -_{\mathsf{int}} 1))$$

$\forall c_1^\iota, \ldots, c_n^\iota, a^\iota, s^\iota :$ (SelectseqTyping)
$\quad \{mem(s, Seq(a)), Selectseq_T(s, c_1, \ldots, c_n)\}$
$\quad mem(s, Seq(a)) \Rightarrow mem(Selectseq_T(s, c_1, \ldots, c_n), Seq(a))$

$\forall c_1^\iota, \ldots, c_n^\iota, s^\iota : \{Selectseq_T(s, c_1, \ldots, c_n)\}$ (SelectseqLen)
$\quad mem(Len(s), Nat)$
$\quad \Rightarrow proj_{\mathsf{int}}(Len(Selectseq_T(s, c_1, \ldots, c_n))) \leq_{\mathsf{int}} proj_{\mathsf{int}}(Len(s))$

$\forall c_1^\iota, \ldots, c_n^\iota, s^\iota, x^\iota : \{fcnapp(Selectseq_T(s, c_1, \ldots, c_n), x)\}$ (SelectseqApp)
$\quad mem(x, domain(Selectseq_T(s, c_1, \ldots, c_n)))$
$\quad \Rightarrow T(fcnapp(Selectseq_T(s, c_1, \ldots, c_n), x))$

$Selectseq_T(tup_0, c_1, \ldots, c_n) = tup_0$ (SelectseqNil)

$\forall c_1^\iota, \ldots, c_n^\iota, s^\iota, x^\iota : \{Selectseq_T(Append(s, x), c_1, \ldots, c_n)\}$ (SelectseqAppend)
$\quad \wedge T(x, c_1, \ldots, c_n) \Rightarrow$
$\quad\quad Selectseq_T(Append(s, x), c_1, \ldots, c_n)$
$\quad\quad\quad = Append(Selectseq_T(s, c_1, \ldots, c_n), x)$
$\quad \wedge \neg T(x, c_1, \ldots, c_n) \Rightarrow$
$\quad\quad Selectseq_T(Append(s, x), c_1, \ldots, c_n)$
$\quad\quad\quad = Selectseq_T(s, c_1, \ldots, c_n)$

$\forall a^\iota, x_1^\iota, \ldots, x_n^\iota :$ (TupSeqTyping)
$\quad \{mem(x_1, a), \ldots, mem(x_n, a), tup_n(x_1, \ldots, x_n)\}$
$\quad mem(x_1, a) \wedge \cdots \wedge mem(x_n, a) \Rightarrow mem(tup_n(x_1, \ldots, x_n), Seq(a))$

$\forall x_1^\iota, \ldots, x_n^\iota : \{tup_n(x_1, \ldots, x_n)\}$ (TupSeqLen)
$\quad Len(tup_n(x_1, \ldots, x_n)) = cast_{\mathsf{int}}(n)$