Challenges and Solutions for Higher-Order SMT Proofs

Chad E. Brown
Czech Technical University in Prague
Prague, Czech Republic

Mikoláš Janota
Czech Technical University in Prague
Prague, Czech Republic
Mikolas.Janota@cvut.cz

Cezary Kaliszyk
University of Innsbruck
Innsbruck, Austria
cezary.kaliszyk@uibk.ac.at

***Abstract*—An interesting goal is for SMT solvers to produce independently checkable proofs. SMT languages already have expressive power that goes beyond first-order languages, and further extensions would give even more expressive power by allowing quantification over function types. Indeed, such extensions are considered in the current proposal for the new standard SMT3. Given the expressive power of SMT and its extensions, careful thought must be given to the intended semantics and an appropriate notion of proof. We propose higher-order set theory as an appropriate interpretation of SMT (and extensions) and obtain an adequate notion of SMT proofs via proof terms in higher-order set theory. To demonstrate the strength of this approach, we give a number of abstract examples that would be difficult to handle by other notions of proof. To demonstrate the practicality of the approach, we describe a family of integer difference logic examples. We give proof terms for each of these examples and check the correctness of the proofs using two proof checkers: the proof checker distributed with the Proofgold system and an alternative checker we have implemented that does not rely on access to the Proofgold block chain.**

***Index Terms*—SMT, proofs, proof checking, semantics**

I. Introduction

A preliminary proposal for SMT-LIB Version 3.0 was recently published online [1]. According to this proposal, there are plans to extend SMT in serious ways, essentially bringing an expressive power somewhere between Church's simple type theory [11] (by including arrow types) and the Calculus of Inductive Constructions [8], [26], [27] (by including dependent types and inductively defined types). In addition, a working group on SMT proofs was announced [7] with the goal of developing a standard for "producing independently checkable proofs." Indeed, producing proofs from SAT and SMT solvers is a long-standing research challenge, cf. [16], [18], [19], [22], [30]. Of course, having a standard notion of proof for SMT3 will require clarifying the intended semantics of SMT3 so that there is precision about what sets of formulas should be unsatisfiable (so there might be a "proof" of inconsistency) or satisfiable (so there might be a "model").

We consider the possibility of using higher-order set theory via the well-known Werner-Aczel semantics of Calculus of Inductive Constructions to provide both a clear semantics and a notion of checkable proof that is likely to be sufficient for SMT3 as well as for its possible future extensions. Proofs are given in the form of *proof terms*, which are terms in typed lambda calculus checkable against a proposition by a small set of rules (trusted kernel). We also give examples to demonstrate the feasibility of this approach. The examples range from the abstract (induction, Pigeonhole principle, and Schroeder-Bernstein) to the concrete (integer difference logic problems). The abstract examples demonstrate that proofs are available for very general kinds of problems. The concrete examples show that proof terms can be practically constructed and checked for more realistic problems.

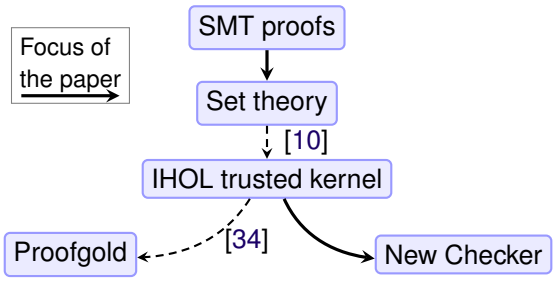

Fig. 1. Schematic outline of the paper

Fig. 1 overviews the paper's main components. Starting from an SMT problem and its proof, we translate them to set theory, which is formalized into *intuitionistic higher-order logic (IHOL)*. Set theory represents a natural target for SMT, e.g. the sort of integers is interpreted as the set of integers, just as expected. Translating to IHOL gives us a small trusted kernel of rules enabling an easy implementation of a proof-checker. Indeed, Section VI compares the performance of the existing Proofgold checker to a new checker. The set theoretical formulations in this paper are all written manually but conceptualized in a way that an SMT solver could automate such translation. In Section V-D we particularly focus on the lazy-SMT approach to solving integer difference logic (IDL).

II. Models and Proofs in General

In the best case scenario a logic provides a clear definition of propositions, a rigorous definition of when a proposition is provable and a class of interpretations with a satisfaction relation. A proposition is considered valid if it is true in every interpretation in the class. The logic satisfies soundness and completeness if provability coincides with validity. The most well-known case is classical first-order logic with any

number of proof systems and interpretations given by Tarski-style semantics.

Church's simple type theory provides another example of such a logic. In Church's original paper [11] there is a clear definition of types, terms (some of which are propositions) and a Hilbert style proof system. Henkin [17] later gave a notion of semantics for which a completeness result could be proven. (Technically Henkin's interpretations were not all sound with respect to Church's functional extensionality axiom, but this was corrected by Andrews [3].) An equality-based version of Church's simple type theory with a Hilbert style proof system and a notion of interpretation (called *general models*) following the Henkin-Andrews approach is presented in [4]. Furthermore in [4] one can find proofs of soundness, completeness and the usual results associated with first-order logic such as the Lowenheim-Skolem Theorem and the Compactness Theorem.

For more serious extensions of Church's simple type theory – such as the Calculus of Inductive Constructions – there does not seem to be an effort to create a Henkin-Andrews notion of interpretation for which one could prove soundness and completeness. Instead research into semantics for type theories has tended to go in the direction of category theory [21], [24] and the most interesting interpretations are not classical.

In terms of soundness alone, there is one well-known set-theoretic interpretation of type theories like the Calculus of Inductive Constructions. The interpretation is classical, extensional and satisfies proof irrelevance.[1] It was described by Werner [33] and Aczel [2] with more details found in the works of Werner, Lee and Barras [5], [25]. In this model, the universe of propositions is interpreted as a two element set – one of which is empty (having no proofs) representing "false" and the other being a singleton (having one proof) representing "true." Being a two element set makes it essentially the same as the interpretation of the type of booleans, as seems to be the intended treatment of propositions as booleans in SMT. Types are interpreted as sets (including the empty set) that live in some universe closed under various set-theoretic operations. Coq is a well-known proof assistant based on the Calculus of Inductive Constructions (CIC). In the calculus of Coq, each type universe is closed under the formation of (dependent) function types and inductively defined types. The Werner-Aczel style of interpretation would interpret each of Coq's universes as a set U closed under the corresponding set-theoretic operations (e.g., if A and B are in the set U, then the set B^A of functions is in the set U).

An alternative to attempting to obtain a Henkin-Andrews style semantics for which soundness and completeness can be proven is to simply take the standard set-theoretic semantics *but* allow the model of the underlying set theory to change. That is, instead of defining a proposition as valid if it is true in every standard set-theoretic interpretation, one could define it as being valid if it is true in every standard set-theoretic interpretation living in a model of, say, first-order ZFC. Validity would then become recursively enumerable again and we clearly have a complete proof system (given by any proof system for first-order ZFC). We explore this possibility in this paper, except we use higher-order Tarski Grothendieck (HOTG) as described in [10] instead of first-order ZFC. The reason for using higher-order instead of first-order is to make the theory finitely axiomatizable. (We still obtain complete calculi via Henkin-Andrews semantics.) The reason for using Tarski Grothendieck instead of Zermelo Fraenkel is to ensure we have sufficient set-theoretic universes for interpreting the type-theoretic universes of CIC. Different possibilities for semantics of and proofs for higher-order SMT are discussed in the unpublished paper [9], from which some of the material for this article was taken.

[1]Proof irrelevance means all proofs of a given proposition are equal.

III. Higher-Order Set Theory with Proof Terms

We begin by giving a formulation of simple type theory with proof terms, which we then extend to include set theory. The types are simple types and the terms are simply typed λ-terms in the style of Church [11]. The proof system is a natural deduction system [28] that admits proof terms in the usual Curry-Howard-de Bruijn style [15], [20], [31]. We additionally include constants and axioms for Tarski-Grothendieck style set theory [32] similar to the formulation described in [10].

We have two base types ι (sets) and o (propositions). All other types are function types of the form $(\alpha\beta)$ of functions from α to β. Such function types are often written as $(\alpha \to \beta)$. When parentheses are omitted they should be replaced to the right, e.g., $\iota\iota o$ is the type $(\iota(\iota o))$.

Let $\mathcal{V}_\alpha$ be the set of variables of type α and $\mathcal{S}_\alpha$ be a set of constants of type α. Assume we have countably many variables at each type. We now define a family $(\Lambda_\alpha)_\alpha$ of terms recursively, where $s \in \Lambda_\alpha$ means s is a term of type α.

- (Variables) If $x \in \mathcal{V}_\alpha$, then $x \in \Lambda_\alpha$.
- (Constants) If $c \in \mathcal{S}_\alpha$, then $c \in \Lambda_\alpha$.
- (Application) If $s \in \Lambda_{\alpha\beta}$ and $t \in \Lambda_\alpha$, then $(st) \in \Lambda_\beta$.
- (Abstraction) If $x \in \mathcal{V}_\alpha$ and $t \in \Lambda_\beta$, then $(\lambda x.t) \in \Lambda_{\alpha\beta}$.
- (Implication) If $s \in \Lambda_o$ and $t \in \Lambda_o$, then $(s \to t) \in \Lambda_o$.
- (Universal Quantification) If $x \in \mathcal{V}_\alpha$ and $t \in \Lambda_o$, then $(\forall x.t) \in \Lambda_o$.

We use common conventions for omitting parentheses and abbreviating multiple binders. Propositions are terms in Λ_o. The set $\mathcal{F}(s)$ for free variables of a term is defined as usual, as is the notion of capture avoiding substitution, denoted s^x_t. We consider two terms to be equal if they are the same up to α-conversion (renaming of bound variables). The notion of $\beta\eta$-conversion, denoted $s{\approx}t$, is also defined in the usual way. When s, t are terms of a common type α, we write $s = t$ as notation for $\forall q.q\ s\ t \to q\ t\ s$ where $q \in \mathcal{V}_{\alpha\alpha o} \setminus (\mathcal{F}(s) \cup \mathcal{F}(t))$. We write $\bot$ for $\forall p.p$ where $p \in \mathcal{V}_o$ and write $\neg s$ for $s \to \bot$.

Given a family of constants $\mathcal{S}$ and a set of propositions $\mathcal{A}$, we can give a notion of provability for intiutionistic higher-order logic (IHOL) via a natural deduction system. We give such a system, annotated with proof terms, in Figure 2. The judgment defined by this system is $\Gamma \vdash \mathcal{D} : t$ (meaning $\mathcal{D}$ is a proof of t in context Γ). We will use this system as a

$$\frac{}{\Gamma \vdash \mathsf{Known}_s : s}\ s \in \mathcal{A} \qquad \frac{}{\Gamma \vdash u : s}\ u : s \in \Gamma$$

$$\frac{\Gamma \vdash \mathcal{D} : s}{\Gamma \vdash \mathcal{D} : t}\ s{\approx}t \qquad \frac{\Gamma, u : s \vdash \mathcal{D} : t}{\Gamma \vdash (\lambda u : s.\mathcal{D}) : s \to t}$$

$$\frac{\Gamma \vdash \mathcal{D} : s \to t \qquad \Gamma \vdash \mathcal{E} : s}{\Gamma \vdash (\mathcal{D}\mathcal{E}) : t}$$

$$\frac{\Gamma \vdash \mathcal{D} : s \qquad x \in \mathcal{V}_\alpha \setminus \mathcal{F}\Gamma}{\Gamma \vdash (\lambda x.\mathcal{D}) : \forall x.s}$$

$$\frac{\Gamma \vdash \mathcal{D} : \forall x.s \qquad x \in \mathcal{V}_\alpha, t \in \Lambda_\alpha}{\Gamma \vdash (\mathcal{D}t) : s_t^x}$$

$$\frac{f, g \in \mathcal{V}_{\alpha\beta}\ \text{distinct}, x \in \mathcal{V}_\alpha}{\Gamma \vdash \mathsf{Ext}_{\alpha,\beta} : (\forall fg.(\forall x.fx = gx) \to f = g)}$$

Fig. 2. Natural Deduction Calculus with Proof Terms

framework in which we can express the axioms of set theory and express and prove set theory theorems. It is straightforward to write a proof checker for such a calculus. Indeed it uses the same ideas as the first implemented proof checker, AUTOMATH [14], [15], dating back to 1968. A particular proof checker is included in software supporting the Proofgold cryptocurrency. Proofgold is an implementation of the idea for a cryptocurrency supporting formal proofs described in the Qeditas white paper [34]. In our examples we will create proofs checkable by the Proofgold proof checker. We will also compare the performance of the checker distributed with the Proofgold Core software to a more efficient alternative implementation due to the third author.

Our primary use case is where $\mathcal{S}$ is a collection of set-theoretic constants (either primitive or defined) and $\mathcal{A}$ is a set of propositions that are either axioms of set theory or follow from those axioms. The particular set theory we have in mind is a form of higher-order Tarski-Grothendieck (HOTG). The primitive constants are those listed in [10], with the exception that we only take the choice operator ε_ι at type ι, rather than at every type. Specifically we have $\varepsilon_\iota : (\iota o)\iota$, $\mathsf{In} : \iota\iota o$, $\mathsf{Empty} : \iota$, $\mathsf{Union} : \iota\iota$, $\mathsf{Power} : \iota\iota$, $\mathsf{Repl} : \iota(\iota\iota)\iota$ and $\mathsf{UnivOf} : \iota\iota$. The axioms we have in mind are those given in [10], again with the exception that we only take a choice axiom at type ι. The axioms are sufficient to ensure the logic is classical and extensional. Functional extensionality is built into the proof system in Figure 2 and we explicitly include an axiom of propositional extensionality: $\forall PQ : o.(P \leftrightarrow Q) \to P = Q$. Via the Diaconescu argument [29], the choice axiom at ι implies excluded middle. As a consequence the proof system is sound and complete with respect to Henkin-Andrews semantics.

From now on we will write set-theoretic propositions in the usual mathematical style, with the understanding that this can be (and is) fully formalized. For example, we write the set extensionality axiom as $\forall XY.X \subseteq Y \to Y \subseteq X \to X = Y$ which means $\forall XY.\mathsf{Subq}\ X\ Y \to \mathsf{Subq}\ Y\ X \to X = Y$ where $\mathsf{Subq} : \iota\iota o$ is defined as $\lambda A.\lambda B.\forall x.\mathsf{In}\ x\ A \to \mathsf{In}\ x\ B$ and In is primitive.

IV. Translating SMT to Set Theory

In this section we describe a translation from SMT problems to propositions of set theory. If the SMT problem is refutable, the set-theoretic proposition will be provable. We begin with a toy example to give an intuition for how the translation will work. Consider the following unsatisfiable SMT problem:

```
(declare-fun p () Bool)
(assert p)
(assert (not p))
```

We will interpret the SMT sort Bool as the set 2 (i.e., $\{0, 1\}$). In the SMT formulation, p is a constant (or equivalently, a nullary function). We translate p as a universally quantified variable restricted to 2. When we wish to use p as a proposition (and not a set) we write $0 \in p$ (since $0 \notin 0$ and $0 \in 1$).[2]

The set-theoretic proposition translated from the SMT problem is

$$\forall p \in 2.0 \in p \to 0 \notin p \to \bot \tag{1}$$

(Note that $0 \notin p$ is notation for $\neg(0 \in p)$ which is notation for $0 \in p \to \bot$.) The proof term for this proposition is

$$\lambda p : \iota.\lambda u : p \in 2.\lambda v : 0 \in p.\lambda w : 0 \notin p.w\ v \tag{2}$$

By saying that (2) is a *proof term* for (1), we mean that $\vdash$(2):(1) is derivable in the calculus given by Fig. 2.

The Werner-Aczel interpretation of the Calculus of Inductive Constructions (CIC) is described elsewhere [2], [5], [25], [33]. For our purposes we simply write $M, N, A, B, D, \ldots$ for terms of CIC and assume we have a partial function which may assign a set $\mathcal{T}_\varphi M$ to M, given an assignment φ for (at least) the variables in M. We assume that for well-typed terms M depending on variables $x_1 : A_1, \ldots x_n : A_n$, $\mathcal{T}_\varphi M$ is defined whenever $\varphi x_i \in \mathcal{T}_\varphi A_i$ for $i \in \{1, \ldots, n\}$. We furthermore assume the values satisfy the expected properties. For example, if M has type A, then $\mathcal{T}_\varphi M \in \mathcal{T}_\varphi A$. In particular, if M is a proposition (has type Prop, where Prop is the universe of propositions in CIC), then $\mathcal{T}_\varphi M \in 2$, where 2 is $\{0, 1\}$. Here, 0 is the empty set and 1 is $\{0\}$. The value 0 is also assigned to every proof. That is, if M is a proposition with proof D, then $\mathcal{T}_\varphi M$ is 1 and $\mathcal{T}_\varphi D$ is 0 (for appropriate assignments φ).

Intuitively $\mathcal{T}$ maps from a type theory (CIC) to the language of mathematics. However, our intention is to use $\mathcal{T}$ to map from CIC to the formal set theory in Section III. This provides both a semantics to CIC and a different (stronger) notion of proof term. While there is no proof of proof irrelevance in CIC, there is a proof of its translation via $\mathcal{T}$ in HOTG.

[2]We could alternatively translate p as $p = 1$ or as $p \neq 0$ since we only care about the behavior for $p \in 2$.

As a starting point for translating SMT to set theory, let us consider sorts and terms in SMT to be corresponding terms in CIC. In that case, $\mathcal{T}$ already provides a method of translating SMT sorts and terms to sets. If we simply consider SMT propositions to be terms of type boolean, then we can also translate SMT propositions to sets (each provably a member of 2) – a set which is "true" if 0 is a member of it and "false" otherwise. However, the SMT propositions will correspond more closely to the set-theoretic propositions if we use $\mathcal{T}$ to define a mapping $\mathcal{T}^p$ sending SMT propositions to set-theoretic propositions. For example, $\mathcal{T}^p_\varphi(\neg P)$ should be $\neg\mathcal{T}^p_\varphi(P)$, $\mathcal{T}^p_\varphi(\forall x : A.P)$ should be $\forall x : \iota.x \in \mathcal{T}_\varphi(A) \to \mathcal{T}^p_{\varphi,x\mapsto x}(P)$ and $\mathcal{T}^p_\varphi(s = t)$ should be $\mathcal{T}_\varphi(s) = \mathcal{T}_\varphi(t)$. If no other case applies, $\mathcal{T}^p_\varphi(P)$ is taken to be $0 \in \mathcal{T}_\varphi(P)$.

It is an oversimplification to consider SMT sorts to be CIC types. Some SMT sorts have a special intended meaning. For example, the SMT sort Int of integers should be interpreted as the set of integers, i.e., $\mathcal{T}_\varphi(\mathsf{Int})$ should be $\omega \cup \{-n \mid n \in \omega\}$, where $-n$ is defined appropriately. In the examples in this paper we will only use the SMT sorts for booleans, integers and arrays. Hence we assume $\mathcal{T}_\varphi(\mathsf{Bool}) = 2$ and $\mathcal{T}_\varphi(\mathsf{Int})$ is the set of integers. The interpretation of arrays is restricted but not fixed by the specification (see Page 39 of [6]), and we will handle this in a special way shown in the next section.

Suppose an SMT problem is given by a set of declarations of sorts $\sigma_1, \ldots, \sigma_n$, typed constants $u_1 : \alpha_1, \ldots, u_m : \alpha_m$ and assertions $P_1, \ldots, P_k$. Let U be a fixed Grothendieck universe, i.e., a set provably satisfying the properties of ZF. We can translate the SMT problem to the set-theoretic proposition

$$\forall\sigma_1 \cdots \sigma_n \in U.\forall u_1 \in \mathcal{T}_{\varphi_1}(\alpha_1) \cdots \forall u_m \in \mathcal{T}_{\varphi_1}(\alpha_m).\\ \mathcal{T}^p_{\varphi_2}(P_1) \to \cdots \to \mathcal{T}^p_{\varphi_2}(P_k) \to \bot$$

where φ_1 takes each α_i to a corresponding variable of type ι (a "set") which we also call α_i and φ_2 extends φ_1 by also taking each u_j to a corresponding variable of type ι (a "set") which we also call u_j. Note that the set-theoretic proposition corresponding to the SMT problem is *provable* if the SMT problem is *unsatisfiable*. As a consequence, if the negation of the set-theoretic proposition is provable, then the SMT problem must be satisfiable. It is, of course, also possible that neither the set-theoretic proposition nor its negation is provable.

V. Examples

We now consider a few examples. In each case we will show the result of translating the problem to a formal set theory and note there is either a formal proof of the set-theoretic proposition or a formal proof of its negation. We briefly describe the proofs in each case. To make definitions and construct proofs in the formal set theory we will use the Megalodon system (the successor to the Egal system [10]). Megalodon can also produce Proofgold proof terms presented in a simple to parse prefix notation.[3] While the Proofgold checker can be used for type checking and proof checking the data, we claim that it is straightforward to implement an independent proof checker and we additionally check the proofs with a faster reimplementation of the checker. We allow ourselves to freely use previous definitions or previously proven results (if they have been previously proven in Megalodon and published in Proofgold documents). That is, we do not need the proof term to contain a justification back to the axioms of set theory, but only back to previously proven results. Later we will also call the proof checkers on a sequence of documents that do contain proofs for each theorem, starting from the axioms. Doing so is where the efficiency of the reimplemented checker is evident.

A. Induction on Natural Numbers

As a first simple example we consider induction on the natural numbers. Here the natural numbers are considered as a predicate over the sort Int.

In SMT2 format we can assert induction fails (which should be unsatisfiable) by giving a predicate p which holds for 0 and is closed under successor but does not hold for all integers $n \geq 0$. The SMT2 specification is as follows:

```
(declare-fun p (Int) Bool)
(assert (p 0))
(assert (forall ((?n Int))
                (=> (<= 0 ?n) (=> (p ?n) (p (+ ?n 1))))))
(assert (not (forall ((?n Int)) (=> (<= 0 ?n) (p ?n))))))
```

To translate this into a set theoretical statement, we must give a specific set representing integers. For natural numbers a reasonable option is to take the finite ordinals (the members of ω). As part of a formalization of Conway's surreal numbers [12] we also have a unary minus operation on all surreal numbers (including ordinals). The details are not important here, but it is sufficient to note that $-0 = 0$, $-n \notin \omega$ if $n \in \omega$ and $--x = x$ for all surreal numbers x. We take int to be the set $\omega \cup \{-n \mid n \in \omega\}$ and use int as the fixed interpretation of the sort Int. In the Megalodon preamble file that we use, this definition appears as follows:

```
Definition int : set := omega :\/: {- n|n :e omega}.
```

Note that the infix :e is Megalodon's notation for $\in$.

We also have defined binary operations $+$ and $*$ on surreal numbers which behave as expected on int, as well as orderings $<$ and $\leq$ on surreal numbers. In general we will not give details about definitions unless they are relevant. We will only state some relevant properties we use, but emphasize that all properties we use have been previously proven in Megalodon and published into the Proofgold chain. There are no goals left open. To make the translation more direct on propositions, we assume $\mathcal{T}^p(s < t)$ is $\mathcal{T}(s) < \mathcal{T}(t)$ and $\mathcal{T}^p(s \leq t)$ is $\mathcal{T}(s) \leq \mathcal{T}(t)$ when s and t are of type Int.

Instead of writing $0 \in \mathcal{T}_\varphi(s)$ we define local notation bp ("bool to prop") in Megalodon as follows:

```
Let bp : set →prop := fun b => 0 :e b.
```

[3]The full data is available at http://grid01.ciirc.cvut.cz/~chad/s.

We briefly consider the behavior of bp when applied to booleans (members of the set $\{0,1\}$). The negation of bp 0 is $0 \notin 0$ which is provable, so bp 0 acts as the false proposition. On the other hand bp 1 is $0 \in 1$ which is provable, so bp 1 acts as the true proposition. Note $\mathcal{T}^p_\varphi(P)$ is bp $\mathcal{T}_\varphi(P)$ if P is not one of the special cases mentioned above.

The statement of the set-theoretic translation of the SMT2 problem appears as follows in Megalodon:

```
Theorem example1ind_unsat: forall p :e 2 :^: int, bp (p 0)
  →(forall n :e int, 0 <= n →bp (p n) →bp (p (n + 1)))
  →~(forall n :e int, 0 <= n →bp (p n))
  →False.
```

The set 2 :^: int denotes the set of functions from integers to booleans: 2^{int}. Essentially the statement says the three (translated) assertions lead to a contradiction.

The proof in Megalodon proceeds as follows: we assume p is in the set 2^{int} and assume the three properties hold. In the preamble there is a predicate nat_p that holds for the finite ordinals. A previously proven induction principle is included:

```
nat_ind : forall p:set→prop, p 0
  →(forall n, nat_p n →p n →p (ordsucc n))
  →forall n, nat_p n →p n.
```

It is straightforward to prove the translated statement from this already known induction principle.

B. Pigeonhole Principle on Arrays

Our second example will be two versions of the Pigeonhole Principle. We use arrays from integers to integers (with some constraints) to play the role of functions from finite ordinals to finite ordinals. In the first version we will state that every array acting as a function from $\{0,\ldots,n\}$ to $\{0,\ldots,n-1\}$ is not injective. In SMT2 format we assert the negation of this statement as follows:

```
(assert (not (forall ((?n Int))
 (=> (>= ?n 0) (forall ((?f (Array Int Int)))
  (=> (forall ((?i Int)) (=> (and (<= 0 ?i) (<= ?i ?n))
       (and (<= 0 (select ?f ?i)) (< (select ?f ?i) ?n))))
   (exists ((?i Int) (?j Int))
       (and (<= 0 ?i) (< ?i ?j) (<= ?j ?n)
            (= (select ?f ?i) (select ?f ?j))))))))))
```

In order to translate this SMT2 problem into a statement of formal set theory we must interpret arrays. We will translate to a statement that universally quantifies over appropriate interpretations of arrays. An interpretation of arrays is a function Array taking two sets X (the set of indices) and Y (the set of values) and returning a set Array X Y such that Array X Y is a set of functions from X to Y that is closed under changing one value.[4] If $f \in$ Array X Y and we want to change the value of f on $x \in X$ to be $y \in Y$, we can represent the corresponding function as $\lambda u \in X.\texttt{if } u = x \texttt{ then } y \texttt{ else } f\ u$. Here $\lambda u \in X.t$ is notation for the set-theoretic encoding of the function taking inputs $u \in X$ to values t (depending on u). The formal details are not important, but $\lambda u \in X.t$ means the HOTG term lam X $(\lambda u.t)$ of type ι, where we assume lam has been previously defined. Likewise, the if-then-else construct corresponds to the use of a defined If of type $o\iota\iota$. Megalodon supports notation for such set-level λ binders and if-then-else notation. In addition, a set f (of type ι) can be directly applied to a set u in Megalodon, where the meaning is the HOTG term ap f u and ap of type $\iota\iota\iota$ is previously defined. We can summarize the properties of an interpretation of Array via the following definition in Megalodon:

```
Definition Array_interp : (set →set →set) →prop
  := fun Array => (forall X Y, Array X Y c= Y :^: X)
 /\ (forall X Y, forall f :e Array X Y,
                   forall x :e X, forall y :e Y,
     (fun u :e X => if u = x then y else f u) :e Array X Y).
```

To deal with arrays, we modify the translation so that $\mathcal{T}_\varphi$Array is a special selected variable Array : $\iota\iota\iota$ and produce the set-theoretic problem

$$\forall Array.\texttt{Array\_interp } Array \to \forall \sigma_1 \cdots \sigma_n \in U.\\ \forall u_1 \in \mathcal{T}_{\varphi_1}(\alpha_1) \cdots \forall u_m \in \mathcal{T}_{\varphi_1}(\alpha_m).\\ \mathcal{T}^p_{\varphi_2}(P_1) \to \cdots \to \mathcal{T}^p_{\varphi_2}(P_k) \to \bot.$$

Translating the Pigeonhole SMT problem to the formal set theory of Megalodon we have the following theorem:

```
Theorem PigeonHoleArrays_1_unsat :
 forall Array:set →set →set, Array_interp Array →
 ~(forall n :e int, 0 <= n →forall f :e Array int int,
  (forall i :e int, 0 <= i /\ i <= n →0 <= f i /\ f i < n)
   →(exists i j :e int, 0 <= i /\ i < j
                /\ j <= n /\ f i = f j)) →False.
```

We can prove the set-theoretic version by reducing to the following previously proven version of the Pigeonhole principle:

```
PigeonHole_nat : forall n, nat_p n →forall f:set →set,
       (forall i :e ordsucc n, f i :e n)
   →~(forall i j :e ordsucc n, f i = f j →i = j).
```

A second version of the Pigeonhole principle states that every (array acting as an) injective function from $\{0,\ldots,n-1\}$ into $\{0,\ldots,n-1\}$ is surjective. As an SMT2 problem this can be stated as follows:

```
(assert (not (forall ((?n Int)) (=> (>= ?n 0)
 (forall ((?f (Array Int Int)))
  (=> (forall ((?i Int)) (=> (and (<= 0 ?i) (< ?i ?n))
       (and (<= 0 (select ?f ?i)) (< (select ?f ?i) ?n))))
  (=> (forall ((?i Int) (?j Int))
       (=> (and (<= 0 ?i) (< ?i ?n) (<= 0 ?j) (< ?j ?n)
            (= (select ?f ?i) (select ?f ?j))) (= ?i ?j)))
   (forall ((?j Int)) (=> (and (<= 0 ?j) (< ?j ?n))
    (exists ((?i Int)) (and (<= 0 ?i) (< ?i ?n)
                            (= (select ?f ?i) ?j))))))))))))
```

The corresponding Megalodon theorem looks as follows:

[4]Note that this allows the set of arrays to be empty. If all types in SMT3 will be assumed to be nonempty, then this definition should be changed.

```
Theorem PigeonHoleArrays_2_unsat :
  forall Array:set →set →set, Array_interp Array →
~(forall n :e int, 0 <= n →forall f :e Array int int,
  (forall i :e int, 0 <= i /\ i < n →0 <= f i /\ f i < n)
  →(forall i j :e int, 0 <= i /\ i < n /\ 0 <= j
                              /\ j < n /\ f i = f j →i = j)
  →(forall j :e int, 0 <= j /\ j < n
           →exists i :e int, 0 <= i /\ i < n /\ f i = j))
 →False.
```

The Megalodon proof proceeds by reducing to a similar previously proven version of the Pigeonhole Principle. It would also be possible to infer the second version from the first version simply by instantiating with an array with one element changed.

C. Failure of Schroeder-Bernstein for Arrays

As a third example, we consider the Schroeder-Bernstein property for arrays. That is, we consider whether or not two types α and β must have a bijection between them if there are injections from α into β and β into α. In this case the negation of the property is satisfiable and we give an interpretation of arrays for which the property fails. Usually in logic there is either a proof on the one hand or a model on the other. However, in this case we can also give a proof term for a proof of the negation of the set theoretical property (where the negation is before the quantifier over possible interpretations of arrays). Since the negation of the translated proposition is provable, the translated proposition is *not* provable (assuming consistency of HOTG) and so the original SMT problem must be satisfiable.

For the SMT2 problem we let f and g be of appropriate array types and assume f and g are injective. (For simplicity we fix the two types to both be Int instead of declaring two sorts.) We then assume there does not exist a bijective array.

```
(declare-fun f () (Array Int Int))
(declare-fun g () (Array Int Int))
(assert (forall ((?m Int) (?n Int))
   (=> (= (select f ?m) (select f ?n)) (= ?m ?n))))
(assert (forall ((?m Int) (?n Int))
   (=> (= (select g ?m) (select g ?n)) (= ?m ?n))))
(assert (not (exists ((?h (Array Int Int)))
 (and (forall ((?m Int) (?n Int))
        (=> (= (select ?h ?m) (select ?h ?n)) (= ?m ?n)))
      (forall ((?n Int))
           (exists ((?m Int)) (= (select ?h ?m) ?n)))))))
```

The translation of this problem to a set-theoretic proposition in Megalodon appears as follows:

```
forall Array:set →set →set, Array_interp Array →
   (forall f g :e Array int int,
       (forall m n :e int, f m = f n →m = n)
     →(forall m n :e int, g m = g n →m = n)
     →~(exists h :e Array int int,
            (forall m n :e int, h m = h n →m = n)
          /\ (forall n :e int, exists m :e int, h m = n))
     →False)).
```

We can prove the negation of this proposition (where we emphasize the negation is before the quantifier over Array). The most important choice for proving this negated proposition is properly instantiating for Array. Fig. 3 gives an informal intuition for the construction.

···	-2	-1	0	1	2	···
···	3	1	0	2	4	···

Fig. 3. Arrays f and g are set to the depicted (injective) array. Any finite number of updates to it will not create a bijection between Int and Int.

We start by defining an injective function from integers to natural numbers which sends negative integers x to $(2(-x))+1$ and nonnegative integers x to $2x$.

```
set int_into_nat : set := (fun x :e int =>
   if x < 0 then ordsucc (2 * (- x)) else 2 * x).
```

We can now inductively define the collection of all functions that are the same as `int_into_nat` except on finitely many elements.

```
set ArrayIntInt_p : set →prop := fun f =>
 forall p:set →prop, p int_into_nat
  →(forall f, forall x y :e int, p f →
        p (fun u :e int => if u = x then y else f u))
  →p f.
```

Finally we can define Array (the term we will use as the instantiation for the quantified variable Array) to be the set of all functions unless both arguments are the set of integers, in which case the functions must satisfy `ArrayIntInt_p`.

```
set Array : set →set →set := fun A B =>
 if A = int /\ B = int
 then {f :e int :^: int | ArrayIntInt_p f} else B :^: A.
```

Intuitively it should be clear that this choice satisfies `Array_interp`. It is also the case that $Array$ int int contains no bijection. Formally we prove that every function satisfying `ArrayIntInt_p` has a lower bound and then use this to conclude that such a function cannot be a surjection.

D. Integer Difference Logic

Our final examples will be integer difference logic problems. Two are from the "job shop" collection from QF_IDL portion of the SMT library. One is satisfiable and the other is unsatisfiable. In both cases we can obtain proof terms for the corresponding set-theoretic proposition.

As described in [13], [23] satisfiability of a set of atoms of the form $x_1 - x_0 \leq v_0$, $x_2 - x_1 \leq v_1$, ..., $x_0 - x_{n-1} \leq v_{n-1}$ (where the variables range over integers) can be decided by forming a certain directed graph with edges labeled by integers and checking if there is a negative cycle. If there is no negative cycle, then values for the variables can be computed from the graph.

We first consider a slightly modified (but equivalent) version of the problem `jobshop2-2-1-1-4-4-11`. In the problem there are five integer variables s_1^1, s_2^1, s_1^2, s_2^2 and ref. The assertion given in the problem is

$$(v_0 \vee v_1) \wedge (v_2 \vee v_3) \wedge s_2^1 - s_1^1 \geq 4 \wedge s_2^2 - s_1^2 \geq 4$$
$$\wedge s_2^1 - \text{ref} \leq 7 \wedge s_2^2 - \text{ref} \leq 7 \wedge s_1^1 - \text{ref} \geq 0 \wedge s_1^2 - \text{ref} \geq 0$$

where v_1, v_0, v_3 and v_2 are locally defined (via a `let`) to be the atoms $s_1^1 - s_1^2 \geq 4$, $s_1^2 - s_1^1 \geq 4$, $s_2^1 - s_2^2 \geq 4$ and $s_2^2 - s_2^1 \geq 4$, respectively. This problem is unsatisfiable. An informal proof of unsatisfiability proceeds by splitting into two cases via the disjunction $v_2 \vee v_3$. In the v_2 case there is a negative cycle given by $s_2^2, s_2^1, s_1^1, \mathrm{ref}$. In the v_3 case there is a negative cycle given by $s_2^1, s_2^2, s_1^2, \mathrm{ref}$.

The set-theoretic version of the problem can be defined as the following proposition in Megalodon.

```
Definition jobshop2_2_1_1_4_4_11 : prop :=
 forall s1_1 s1_2 s2_1 s2_2 ref :e int,
 let v_0:prop := 4 <= s2_1 + -s1_1 in
 let v_1:prop := 4 <= s1_1 + -s2_1 in
 let v_2:prop := 4 <= s2_2 + -s1_2 in
 let v_3:prop := 4 <= s1_2 + -s2_2 in
    ((v_0 \/ v_1) /\ (v_2 \/ v_3)
  /\ 4 <= s1_2 + -s1_1 /\ 4 <= s2_2 + -s2_1
  /\ s1_2 + -ref <= 7 /\ s2_2 + -ref <= 7
  /\ 0 <= s1_1 + -ref /\ 0 <= s2_1 + -ref)
 ->False.
```

Note that we have slightly modified the inequalities to all use $\leq$ for simplicity. Also, we combine the unary $-$ with the binary $+$ operator instead of using a binary $-$ operator. The proposition above corresponds to the unsatisfiability of the original SMT problem and it can be proven in set theory using the negative cycles mentioned above and the following (formally proven) result.

```
Theorem idl_negcycle_4 : forall x y z w v1 v2 v3 v4,
    SNo x ->SNo y ->SNo z ->SNo w
 ->SNo v1 ->SNo v2 ->SNo v3 ->SNo v4
 ->v1 + v2 + v3 + v4 < 0 ->y + -x <= v1 ->z + -y <= v2
 ->w + -z <= v3 ->x + -w <= v4 ->False.
```

The theorem `idl_negcycle_4` is specific to negative cycles of length 4, but is also more general since variables range over values satisfying the predicate `SNo`, a predicate true for integers, real numbers, and more (Conway's extension of the real numbers described in [12]). It is relatively easy to prove `idl_negcycle_4`, as it is for smaller cycles. It is also possible to algorithmically generate a proof of a theorem for $n+1$-cycles given one for n-cycles. We have done this for $n \in \{2, \ldots, 22\}$.

The next example we consider is a slightly modified version of `jobshop2-2-1-1-4-4-12` which is a simple modification of the previous example by changing each 7 to 8.

$$(v_0 \vee v_1) \wedge (v_2 \vee v_3) \wedge s_2^1 - s_1^1 \geq 4 \wedge s_2^2 - s_1^2 \geq 4$$
$$\wedge s_2^1 - \mathrm{ref} \leq 8 \wedge s_2^2 - \mathrm{ref} \leq 8 \wedge s_1^1 - \mathrm{ref} \geq 0 \wedge s_1^2 - \mathrm{ref} \geq 0$$

This makes the problem satisfiable by taking $s_1^1 = 0$, $s_2^1 = 4$, $s_1^2 = 4$, $s_2^2 = 8$ and $\mathrm{ref} = 0$.

The corresponding set-theoretic proposition is given as follows:

```
Definition jobshop2_2_1_1_4_4_12 : prop :=
 forall s1_1 s1_2 s2_1 s2_2 ref :e int,
 let v_0:prop := 4 <= s2_1 + -s1_1 in
 let v_1:prop := 4 <= s1_1 + -s2_1 in
 let v_2:prop := 4 <= s2_2 + -s1_2 in
 let v_3:prop := 4 <= s1_2 + -s2_2 in
    ((v_0 \/ v_1) /\ (v_2 \/ v_3)
  /\ 4 <= s1_2 + -s1_1 /\ 4 <= s2_2 + -s2_1
  /\ s1_2 + -ref <= 8 /\ s2_2 + -ref <= 8
  /\ 0 <= s1_1 + -ref /\ 0 <= s2_1 + -ref)
 ->False.
```

Since the problem is satisfiable, the proposition cannot be proven. However, we can prove its negation by assuming the proposition holds and applying it to the values above. One must then proven v_0 and v_2 hold (giving both the disjunctions) and prove the remaining inequalities hold.

We finally consider a family of integer difference logic examples. For $m \in \{3, \ldots, 16\}$ let idl_m be the integer difference logic problem with integer variables l_0, ..., l_m, r_0, ..., r_m, n and s. Assume $s - n \leq m + 1$, $l_0 - s \leq -1$, $r_0 - s \leq -1$, $n - l_m \leq -1$ and $n - r_m \leq -1$. Additionally we assume disjunctions $l_{i+1} - l_i \leq -1 \vee r_{i+1} - l_i \leq -1$ and $r_{i+1} - r_i \leq -1 \vee l_{i+1} - r_i \leq -1$ for $i \in \{0, \ldots, m-1\}$. The proof requires considering 2^m cases, each of which is contradictory since there would be a negative $m+3$ cycle. We have generated the problems and their solutions algorithmically as Megalodon propositions and proofs.

VI. Results: Checking Proofs

In order to show the practical applicability of the approach, we show that the proofs generated in the previous section can be efficiently verified. The generality of proof terms allows checking them independently. To demonstrate this, as already mentioned in Section III, we not only provide a lower-level proof checker, but also show that the proofs can be verified by the Proofgold checker. In this section we discuss the standalone checker and present the results of the experiments from the previous section.

The checker parses the given ASCII proofs into an in-memory algebraic datatype representation. The representation corresponds to the proof rules of the calculus (as presented in Fig. 2). In order to verify that all the proof rules "KNOWN" are declared and correctly interpreted (are of correct types), already as part of this process, the checker also loads and proof-checks all the declared primitives, definitions, and the proved properties. These can for example correspond to the basic concepts in the used HOTG set theory (introduced in Section III), but also further definitions and the proven consequences of these definitions such as these shown in the last section.

The checking of the individual proof term rules follows their definitions (again as given in Fig. 2). Checking most of those is straightforward, however, they heavily rely on a simply-typed λ-calculus term representation in which checking $\beta\eta$-convertibility needs to be efficient. For this reason, our standalone checker implements a term representation that preserves the invariant that terms are $\beta\eta$ normal. As this operation may additionally be costly, we furthermore provide maximum (sub)term sharing and additionally cache the results of reductions. As the sharing and reductions are the most

TABLE I
TIME TO PARSE AND CHECK ALL PREVIOUSLY PUBLISHED DOCUMENTS

	New Checker	Proofgold Core Checker
ASCII	6s	57.7s
binary	2.5s	44s

costly, we implement a low-level (simply-typed) lambda-calculus representation in the C programming language, which exposes only the high-level term construction and inspection operations. We use this low-level library from an OCaml checker for the individual proof steps.

We report the time taken to check the proofs in Tables I and II. In Proofgold proofs are part of documents, so what we report are the time to check the document. The usual way documents are checked in Proofgold Core relies on knowing what has been previously published so that one can use previous definitions and proven results. In order to do standalone checking we read and check all 317 documents published into the HOTG theory of the Proofgold blockchain, hashing previous results for use when checking the next document. In Table I we report the time to check these 317 documents before checking the new document. We give the results for two formats: one is ASCII format mentioned above (so that the time includes the time to parse) and another is a binary format native to OCaml. The new checker is almost 18 times faster when checking the binary format, showing a significant improvement over the checking time of the Proofgold Core checker.

Table II shows the time taken by the two checkers to check the new document (the one with the relevant proof). On the examples other than the family of integer difference logic examples, the new checker is roughly 4.6 times faster. On the family of examples, as the n gets larger, the new checker begins to take longer than Proofgold Core. The likely reason for this is that in the family of examples the steps are mostly unique and therefore there is almost no possibility for term sharing, so the new checker introduces overhead for these.

VII. CONCLUSION

A finitely axiomatized set theory using Church's type theory instead of first-order logic provides a simple way of obtaining a candidate semantics for SMT3. Via Curry-Howard, we also naturally obtain a candidate notion of independently checkable proof term. We have given examples that show such a notion of proof is sufficient to handle abstract examples that may be out of reach for more specific notions of proof. We have also given a family of integer difference logic examples to demonstrate that such proof terms can be practically constructed and checked. An efficient reimplementation of the checker distributed with Proofgold Core is able to check all the examples in this paper, checking all proofs back to the axioms of set theory within a few seconds.

TABLE II
TIME TO CHECK BINARY VERSIONS THE NEW DOCUMENT

	New Checker	Proofgold Core Checker
Induction	0.0015s	0.005s
Pigeonhole1	0.0017s	0.0065s
Pigeonhole2	0.0018s	0.0075s
Schroeder-Bernstein	0.0049s	0.0268s
Jobshop_{11}	0.0017s	0.0098s
Jobshop_{12}	0.0015s	0.0042s
IDL_3	0.0018s	0.0043s
IDL_4	0.0025s	0.0058s
IDL_5	0.0035s	0.0079s
IDL_6	0.0064s	0.0117s
IDL_7	0.0121s	0.0263s
IDL_8	0.0276s	0.0434s
IDL_9	0.069s	0.1104s
IDL_{10}	0.209s	0.486s
IDL_{11}	0.721s	0.527s
IDL_{12}	1.738s	0.991s
IDL_{13}	4.088s	2.232s
IDL_{14}	9.603s	5.347s
IDL_{15}	22.763s	10.91s
IDL_{16}	46.177s	24.138s

ACKNOWLEDGMENT

The results were supported by the Ministry of Education, Youth and Sports within the dedicated program ERC CZ under the project *POSTMAN* no. LL1902 as well as the ERC starting grant no. 714034 *SMART*. This scientific article is part of the *RICAIP* project that has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 857306.

REFERENCES

[1] SMT-LIB version 3.0 - preliminary proposal, 2021. http://smtlib.cs.uiowa.edu/version3.shtml.
[2] Peter Aczel. On relating type theories and set theories. In Thorsten Altenkirch, Wolfgang Naraschewski, and Bernhard Reus, editors, *TYPES*, volume 1657 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 1998.
[3] Peter B. Andrews. General models and extensionality. *J. Symb. Log.*, 37:395–397, 1972.
[4] Peter B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof*. Kluwer Academic Publishers, 2nd edition, 2002.
[5] Bruno Barras. Sets in Coq, Coq in Sets. *Journal of Formalized Reasoning*, 3(1), 2010.
[6] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The SMT-LIB Standard: Version 2.6. Technical report, Department of Computer Science, The University of Iowa, 2017. Available at `www.SMT-LIB.org`.
[7] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. New working group on SMT proofs, 2021. https://groups.google.com/g/smt-lib/c/a0-U-nU4J68.
[8] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004.
[9] Chad E. Brown. Notes on semantics of and proofs for SMT, Sep 2021.
[10] Chad E. Brown and Karol Pąk. A tale of two set theories. In Cezary Kaliszyk, Edwin C. Brady, Andrea Kohlhase, and Claudio Sacerdoti Coen, editors, *Intelligent Computer Mathematics - 12th International Conference, CICM 2019, Prague, Czech Republic, July 8-12, 2019, Proceedings*, volume 11617 of *Lecture Notes in Computer Science*, pages 44–60. Springer, 2019.
[11] Alonzo Church. A formulation of the simple theory of types. *The Journal of Symbolic Logic*, 5:56–68, 1940.

[12] John H. Conway. *On numbers and games, Second Edition*. A K Peters, 2001.

[13] Scott Cotton and Oded Maler. Fast and flexible difference constraint propagation for DPLL(T). In Armin Biere and Carla P. Gomes, editors, *Theory and Applications of Satisfiability Testing - SAT*, pages 170–183, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[14] N. de Bruijn. The Mathematical language AUTOMATH, its usage, and some of its extensions. In *Symposium on Automatic Demonstration*, pages 29–61. Lecture Notes in Mathematics, 125, Springer, 1970.

[15] N. G. de Bruijn. A survey of the project AUTOMATH. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*, pages 579–606. Academic Press, 1980.

[16] Liana Hadarean, Clark W. Barrett, Andrew Reynolds, Cesare Tinelli, and Morgan Deters. Fine grained SMT proofs for the theory of fixed-width bit-vectors. In Martin Davis, Ansgar Fehnker, Annabelle McIver, and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning - 20th International Conference, LPAR-20*, volume 9450 of *Lecture Notes in Computer Science*, pages 340–355. Springer, 2015.

[17] Leon Henkin. Completeness in the theory of types. *The Journal of Symbolic Logic*, 15:81–91, 1950.

[18] Marijn Heule, Warren A. Hunt Jr., and Nathan Wetzler. Trimming while checking clausal proofs. In *Formal Methods in Computer-Aided Design, FMCAD*, pages 181–188. IEEE, 2013.

[19] Marijn J. H. Heule. Proofs of unsatisfiability. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability - Second Edition*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, pages 635–668. IOS Press, 2021.

[20] W.A. Howard. The formulas-as-types notion of construction. In J.P. Seldin and J.R. Hindley, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490, New York, 1980. Academic Press.

[21] B. Jacobs. *Categorical Logic and Type Theory*. ISSN. Elsevier Science, 1999.

[22] Guy Katz, Clark W. Barrett, Cesare Tinelli, Andrew Reynolds, and Liana Hadarean. Lazy proofs for DPLL(T)-based SMT solvers. In Ruzica Piskac and Muralidhar Talupur, editors, *2016 Formal Methods in Computer-Aided Design, FMCAD*, pages 93–100. IEEE, 2016.

[23] Hyondeuk Kim and Fabio Somenzi. Finite instantiations for integer difference logic. In *2006 Formal Methods in Computer Aided Design*, pages 31–38. IEEE, nov 2006.

[24] Joachim Lambek and Philip Scott. *Introduction to higher order categorical logic*. Cambridge University Press, Cambridge, UK, 1986.

[25] Gyesik Lee and Benjamin Werner. Proof-irrelevant model of CC with predicative induction and judgmental equality. *Logical Methods in Computer Science*, 7(4), 2011.

[26] Zhaohui Luo. *Computation and reasoning: a type theory for computer science*. Oxford University Press, Inc., New York, NY, USA, 1994.

[27] The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2020. Version 8.12.

[28] Dag Prawitz. *Natural deduction: a proof-theoretical study*. Dover, 2006.

[29] R. Diaconescu. Axiom of choice and complementation. *Proceedings of the American Mathematical Society*, 51:176–178, 1975.

[30] Hans-Jörg Schurr, Mathias Fleury, Haniel Barbosa, and Pascal Fontaine. Alethe: Towards a generic SMT proof format (extended abstract). In Chantal Keller and Mathias Fleury, editors, *Proceedings Seventh Workshop on Proof eXchange for Theorem Proving, PxTP*, volume 336 of *EPTCS*, pages 49–54, 2021.

[31] M.H.B. Sørensen and P. Urzyczyn. *Lectures on the Curry-Howard Isomorphism*. Rapport (Københavns universitet. Datalogisk institut). Datalogisk Institut, Københavns Universitet, 1998.

[32] A. Tarski. Der Aussagenkalkül und die Topologie. *Fundamentae Mathematicae*, 31:103–134, 1938.

[33] Benjamin Werner. Sets in types, types in sets. In Martín Abadi and Takayasu Ito, editors, *TACS*, volume 1281 of *Lecture Notes in Computer Science*, pages 530–346. Springer, 1997.

[34] Bill White. Qeditas: A formal library as a bitcoin spin-off, 2016.