

Combining Type Theory and Untyped Set Theory

Chad E. Brown

Universität des Saarlandes, Saarbrücken, Germany, cebrown@ags.uni-sb.de

Abstract. We describe a second-order type theory with proof irrelevance. Within this framework, we give a representation of a form of Mac Lane set theory and discuss automated support for constructing proofs within this set theory. One of the novel aspects of the representation is that one is allowed to use any class (in the set theory) as a type (in the type theory). Such class types allow a natural way of representing partial functions (e.g., the first and second operators on the class of Kuratowski ordered pairs). We also discuss how automated search can be used to construct proofs. In particular, the first-order prover Vampire can be called to solve a challenge problem (the injective Cantor Theorem) which is notoriously difficult for higher-order automated provers.

1 Introduction

In order to resolve mathematical conjectures, either a person or a system must know enough mathematics. Very few conjectures can be resolved by going back to first principles. One can imagine having a large library of mathematical definitions and theorems. An automated prover for mathematics should be able to make effective use of this library. First-order provers are designed to deal with large numbers of clauses. On the other hand, it is often difficult and unnatural to force higher-order or set-theoretic concepts into first-order logic.

What is the best language for automated reasoning in mathematics? We sketch two answers:

1. Experience shows first-order logic is the appropriate language for automated reasoning. Some forms of set theory such as von Neumann-Bernays-Gödel (NBG) are finitely axiomatizable in first-order logic and sufficient for representing much of mathematics. The best language is first-order logic with a finitely axiomatized set theory as in [6, 20, 5].
2. Higher-order logic (e.g., Church's type theory) allows more natural representations of mathematical propositions. In particular, λ -terms allow a more computational treatment of sets and functions than first-order set theory. Furthermore, higher-order logic supports type distinctions mathematicians make implicitly. While automated reasoning in higher-order logic is more complicated than in first-order logic, systems such as TPS can automatically prove some theorems which are difficult to even represent in first-order [2].

Of course, a third answer is simply that we do not yet know.

We introduce a new alternative for automated reasoning in mathematics. Instead of trying to force mathematics into first-order logic, we use an LF-style dependent type theory (or, logical framework) with proof irrelevance. This type theory is implemented in a mathematical assistance system Scunak [7].

The Scunak type theory itself is too weak to represent interesting mathematics. We must “axiomatize” (give a signature for) a theory strong enough to represent mathematical propositions and proofs. The axiomatic theory presented here is a form of Mac Lane set theory (as advocated by Mac Lane in [14]). If we insisted on implementing this set theory in first-order logic, then we would need to use axiom schemes.

Another advantage of using a logical framework is that one can use proof terms to naturally represent partial functions by insisting that a proof of a certain condition is required before a term exists. One can further improve the treatment of partial functions by including certain (very restricted) product types and the notion of proof irrelevance in the logical framework. Essentially, one can make any predicate (or, in semantic terms, “class”) into a type at the meta-level.

By noting certain properties of the language, we can argue that automated reasoning is possible in this setting. However, a complete theorem prover has not yet been written for our version of Mac Lane set theory in the Scunak type theory. We have written an interface to the first-order prover Vampire [22] which has been able to find a proof of a challenge problem discussed in [3].

2 A Type Theory with Proof Irrelevance

The type theory of Scunak is a modified version of the type theory LF (also called λP) as implemented in Twelf [19]. A thorough development of the LF meta-theory can be found in [12] and of a similar version of Martin-Löf type theory can be found in [16]. Proof irrelevance in type theory is considered as a modal judgment in [18] and [21]. We conjecture that the type theory we are presenting can be interpreted in the presumably more generally setting investigated in [21]. A more thorough investigation of the Scunak type theory is planned for future work. Here we sketch an example of the intended semantics, then we give the syntax, the typing judgments and corresponding rules.

We begin with a type `obj` which contains all the (untyped) mathematical objects of interest. We take \mathcal{D}_{obj} to be V_{ω^2} in the usual von Neumann hierarchy of sets: $V_\emptyset = \emptyset$, $V_{\epsilon+1} = \mathcal{P}(V_\epsilon)$ and $V_\gamma = \bigcup_{\epsilon < \gamma} V_\epsilon$ for each limit ordinal γ .

Next we want a type `prop` to contain “propositions.” In set theory, the important properties of sets are given by the membership and equality relations. Fix four distinct values $\dot{\in}$, $\dot{\notin}$, $\dot{=}$, $\dot{\neq}$. We take $\mathcal{D}_{\text{prop}}$ to be the set

$$\{(R, x, y) \mid R \in \{\dot{\in}, \dot{\notin}, \dot{=}, \dot{\neq}\}, x, y \in V_{\omega^2}\}$$

Intuitively, for any $x, y \in V_{\omega^2}$, $(\dot{\notin}, x, y)$ represents the proposition that $x \notin y$. Only some propositions should be “true” and true propositions should have

“proofs”. Since we wish to model proof irrelevance, a true proposition should have exactly *one* proof. An easy way to model this is to take the set of proofs to be a subset of the set of propositions. We take \mathcal{D}_ρ to be the subset

$$\begin{aligned} & \{(\dot{=}, x, y) \mid x, y \in V_{\omega^2}, x \in y\} \cup \{(\dot{\neq}, x, y) \mid x, y \in V_{\omega^2}, x \notin y\} \\ & \cup \{(\dot{=}^{\cdot}, x, x) \mid x \in V_{\omega^2}\} \cup \{(\dot{\neq}^{\cdot}, x, y) \mid x, y \in V_{\omega^2}, x \neq y\}. \end{aligned}$$

of $\mathcal{D}_{\text{prop}}$. The intention is that $p \in \mathcal{D}_\rho$ is a (the) proof of p .

Consider a function ϕ from \mathcal{D}_{obj} to $\mathcal{D}_{\text{prop}}$. Note that this determines a subset $\{x \in \mathcal{D}_{\text{obj}} \mid \phi(x) \in \mathcal{D}_\rho\}$ of \mathcal{D}_{obj} . Intuitively this is the “class” (relative to V_{ω^2}) of $x \in V_{\omega^2}$ such that $\phi(x)$ has a proof. This subset is isomorphic to the set $\{\langle x, p \rangle \in \mathcal{D}_{\text{obj}} \times \mathcal{D}_\rho \mid p = \phi(x)\}$, motivating consideration of the product type $\text{obj} \times \rho$.

Let \mathcal{ST} be the following set of simple types: We have three base simple types obj , prop , and ρ . We allow one product type $(\text{obj} \times \rho) \in \mathcal{ST}$. Finally, for any $\alpha, \beta \in \mathcal{ST}$, $(\alpha \rightarrow \beta) \in \mathcal{ST}$. Taking $\mathcal{D}_{\text{obj} \times \rho} := \mathcal{D}_{\text{obj}} \times \mathcal{D}_\rho$ and $\mathcal{D}_{\alpha \rightarrow \beta}$ to be the set of all functions from \mathcal{D}_α to \mathcal{D}_β for any α and β we obtain a standard frame for these types. The frame \mathcal{D} is sufficient to evaluate simply typed λ -terms (with pairing between obj and ρ).

We have modelled proofs using the simple type ρ . However, this is not sufficient. We care about proofs *of propositions*. That is, for any $p \in \mathcal{D}_{\text{prop}}$, we care about the proofs of p in ρ . We use a dependent type $\text{pf } P$ for this purpose. Semantically, dependent types are partial equivalence relations (pers) over the domains \mathcal{D}_α . Only valid types will correspond to pers, the rest will simply be binary relations. For each α , let \mathcal{R}_α denote the set of all binary relations over \mathcal{D}_α . For a relation R , let $|R|$ be $\{x \mid \langle x, x \rangle \in R\}$.

We will interpret $\text{pf } P$ using a function $\overline{\text{pf}}$ from $\mathcal{D}_{\text{prop}}$ to \mathcal{R}_ρ taking a proposition p to the per $\{\langle p, p \rangle\}$ if $p \in \mathcal{D}_\rho$ and to the empty per otherwise. Hence $\overline{\text{pf}}(p)$ is a per with one equivalence class if p has a proof, and $\overline{\text{pf}}(p)$ is empty if p has no proof.

The other basic dependent type depends on $\phi \in \mathcal{D}_{\text{obj} \rightarrow \text{prop}}$. As described above, such a “predicate” ϕ determines a subset of \mathcal{D}_{obj} , which is a “class” relative to \mathcal{D}_{obj} . This class can be represented by the domain of the per $\overline{\text{class}}(\phi)$ on $\mathcal{D}_{\text{obj}} \times \mathcal{D}_\rho$ given by $\{\langle \langle x, \phi(x) \rangle, \langle x, \phi(x) \rangle \rangle \mid x \in \mathcal{D}_{\text{obj}}, \phi(x) \in \mathcal{D}_\rho\}$. The remaining dependent types correspond to Π -types. Semantically, given $R \in \mathcal{R}_\alpha$ and $F : \mathcal{D}_\alpha \rightarrow \mathcal{R}_\beta$, we define $\overline{\Pi}(R, F) \in \mathcal{R}_{\alpha \rightarrow \beta}$ by $f(\overline{\Pi}(R, F))g$ iff $f(x) F(x) g(y)$ for all $x R y$.

We now describe the syntax. Let \mathcal{V} be an infinite set of variables and \mathcal{C} be a set of constants. We use x, y, z, x^1, \dots to denote variables and c, d, c^1, \dots to denote constants. We define terms and types as follows:

Terms $M, N, P, Q, R, \phi, \dots := x \mid c \mid (\lambda x. M) \mid (M N) \mid \langle M, N \rangle \mid \pi_1(M) \mid \pi_2(M)$
Types $A, B, C, \dots := \text{obj} \mid \text{prop} \mid (\text{pf } P) \mid (\text{class } \phi) \mid (\Pi x : A. B)$

As usual we identify terms and types up to α -conversion. We assume all the usual notions of λ -calculus: substitution, β -reduction, η -reduction and the following pairing reductions:

$$\begin{aligned}
(\pi_1) : \pi_1(\langle M, N \rangle) &\rightarrow_{\pi_1} M & (\pi_2) : \pi_2(\langle M, N \rangle) &\rightarrow_{\pi_2} N \\
(\pi) : \langle \pi_1(M), \pi_2(M) \rangle &\rightarrow_{\pi} M
\end{aligned}$$

We say a term or type is normal if it contains no redexes. We write W^\downarrow for the normal form of W , if a unique normal form of W exists. In practice we will consider terms and types which are of a certain class (respecting simple types) which satisfy strong normalization and the Church-Rosser property. For such terms and types, W^\downarrow exists (see, for example, [13]).

For some rules in the typing judgment we will η - or π -expand on the fly. We introduce some notation to facilitate this.

If M is a term of the form $(\lambda x N)$, then let \mathbf{x}_λ^M denote x and \mathbf{B}_λ^M denote N . If M is any other term, then let \mathbf{x}_λ^M be a variable not occurring in M and \mathbf{B}_λ^M denote $(M\mathbf{x}_\lambda^M)$. Note that in the first case, $(\lambda\mathbf{x}_\lambda^M\mathbf{B}_\lambda^M)$ is identical to M . In the second case, $(\lambda\mathbf{x}_\lambda^M\mathbf{B}_\lambda^M)$ η -reduces to M .

If M is a term of the form $\langle N, P \rangle$, then let \mathbf{fst}^M denote N and \mathbf{snd}^M denote P . If M is any other term, then let \mathbf{fst}^M denote $\pi_1(M)$ and \mathbf{snd}^M denote $\pi_2(M)$. In the first case, $\langle \mathbf{fst}^M, \mathbf{snd}^M \rangle$ is identical to M . In the second case, $\langle \mathbf{fst}^M, \mathbf{snd}^M \rangle$ π -reduces to M .

A signature Σ is a list of distinct constants associated with types, and a type context Γ is a list of distinct variables associated with types. A simple type signature Ξ is a list of distinct constants associated with simple types, and a simple type context Δ is a list of distinct variables associated with simple types.

We can interpret $c : \alpha \in \Xi$ by giving $\llbracket c \rrbracket \in \mathcal{D}_\alpha$. We can interpret $\llbracket \Delta \rrbracket$ as $\{0\} \times \mathcal{D}_{\alpha^1} \times \dots \times \mathcal{D}_{\alpha^n}$, when Δ is $x^1 : \alpha^1, \dots, x^n : \alpha^n$. (Special Case: $\llbracket \cdot \rrbracket := \{0\}$.)

The dependencies of types on objects can be erased to obtain a simple type as follows: $\llbracket \mathbf{obj} \rrbracket := \mathbf{obj}$, $\llbracket \mathbf{prop} \rrbracket := \mathbf{prop}$, $\llbracket \mathbf{pf} M \rrbracket := \rho$, $\llbracket \mathbf{class} M \rrbracket := (\mathbf{obj} \times \rho)$, and $\llbracket (IIx : A.B) \rrbracket := (\llbracket A \rrbracket \rightarrow \llbracket B \rrbracket)$. We can use this operation to obtain a simple type signature Ξ (resp., simple type context Δ) from a signature Σ (resp., type context Γ).

In Scunak, terms and types are always given in $\beta\pi_1\pi_2$ -normal form, so that the types of λ -abstractions and pairs can be inferred from the given intended type.

We begin with two simple typing judgments:

- “ $\Delta \vdash_{\Xi} M : \alpha$ ” In words, M has simple type α . The rules for this judgment are standard and omitted here. The main reason to consider this judgment is to guarantee strong normalization and Church-Rosser. Semantically, assume we have fixed $\llbracket c \rrbracket \in \mathcal{D}_\alpha$ for each $c : \alpha \in \Xi$. Then, when $\Delta \vdash_{\Xi} M : \alpha$ holds, we can define $\llbracket \Delta | M : \alpha \rrbracket$ to be a function from $\llbracket \Delta \rrbracket$ to \mathcal{D}_α (in the obvious way). For $d \in \llbracket \Delta \rrbracket$, we write $\llbracket \Delta | M : \alpha \rrbracket_d$ instead of $\llbracket \Delta | M : \alpha \rrbracket(d)$.
- “ $\Delta \vdash_{\Xi} A \text{ sv}$ ” In words, A is *simply valid*. The rules are omitted. The idea is that if A is $\mathbf{pf} P$ (resp., $\mathbf{class} \phi$), then $\Delta \vdash_{\Xi} P : \mathbf{prop}$ (resp., $\Delta \vdash_{\Xi} \phi : \mathbf{obj} \rightarrow \mathbf{prop}$) must hold. Semantically, when this judgment holds, we can define $\llbracket \Delta | A \rrbracket$ to be a function from $\llbracket \Delta \rrbracket$ to \mathcal{R}_α . In particular, $\llbracket \Delta | \mathbf{obj} \rrbracket$ and $\llbracket \Delta | \mathbf{prop} \rrbracket$ are the identity relations on $\mathcal{D}_{\mathbf{obj}}$ and $\mathcal{D}_{\mathbf{prop}}$. $\llbracket \Delta | \mathbf{pf} P \rrbracket(d)$ is $\mathit{pf}(\llbracket \Delta | P : \mathbf{prop} \rrbracket_d)$, \mathbf{class} types are interpreted using $\overline{\mathit{class}}$, and function types are interpreted using $\overline{\Pi}$. For $d \in \llbracket \Delta \rrbracket$, we write $\llbracket \Delta | A \rrbracket_d$ instead of $\llbracket \Delta | A \rrbracket(d)$.

The main typing judgments are as follows:

- “ $\vdash \Sigma$ sig” Intuitively, Σ is a valid signature. The idea is to ensure $\vdash_{\Sigma} A : Type$ before adding $c : A$ to Σ . We omit the rules. Note, however, that unlike LF, new families are never added to the signature. Semantically, an interpretation respects Σ if for all $c : A \in \Sigma$, $\llbracket c \rrbracket \in \llbracket \cdot | A \rrbracket_0$. We will assume below that the interpretation respects Σ .
- “ $\vdash_{\Sigma} \Gamma$ ctx” Intuitively, Γ is a valid context. The idea is to ensure $\Gamma \vdash_{\Sigma} A : Type$ holds before adding $x : A$ to Γ . We omit the rules. Semantically, $\llbracket \Gamma \rrbracket$ is a per over $\llbracket \llbracket \Gamma \rrbracket \rrbracket$ inductively given by taking $\llbracket \cdot \rrbracket$ to be the identity over $\{0\}$ and taking $\llbracket \Gamma, x : A \rrbracket$ to be $\{\langle \langle d, y \rangle, \langle e, z \rangle \rangle | d(\llbracket \Gamma \rrbracket)e \text{ and } y\llbracket \llbracket \Gamma \rrbracket | A \rrbracket_d z\}$
- “ $\Gamma \vdash_{\Sigma} M \sim N \uparrow A$ ” Intuitively, M can be checked to be A -related to N . The rules are given in Figure 1. Semantically, $\langle \llbracket \llbracket \Gamma \rrbracket | M \rrbracket_d, \llbracket \llbracket \Gamma \rrbracket | N \rrbracket_e \rangle \in \llbracket \llbracket \Gamma \rrbracket | A \rrbracket_d$ whenever $d\llbracket \llbracket \Gamma \rrbracket \rrbracket e$.
- “ $\Gamma \vdash_{\Sigma} M \sim N \downarrow A$ ” Intuitively, the type A can be extracted as a type in which M is A -related to N . The rules are given in Figure 1. As above, this means for all $d\llbracket \llbracket \Gamma \rrbracket \rrbracket e$, $\langle \llbracket \llbracket \Gamma \rrbracket | M \rrbracket_d, \llbracket \llbracket \Gamma \rrbracket | N \rrbracket_e \rangle \in \llbracket \llbracket \Gamma \rrbracket | A \rrbracket_d$.
- “ $\Gamma \vdash_{\Sigma} A : Type$ ” In words, A is a valid type. The rules are given in Figure 1. Semantically, for all $d\llbracket \llbracket \Gamma \rrbracket \rrbracket e$, $\llbracket \llbracket \Gamma \rrbracket | A \rrbracket_d$ is a per on $\mathcal{D}_{\llbracket \Gamma \rrbracket A}$ and $\llbracket \llbracket \Gamma \rrbracket | A \rrbracket_d = \llbracket \llbracket \Gamma \rrbracket | A \rrbracket_e$.

We usually omit the dependence on Σ in the judgments. Note that the rule *coercepf* which normalizes terms includes premisses to ensure the terms are simply typable, hence that unique normal forms exist. Under certain conditions, one can eliminate these premisses.

One can easily show by induction on derivations:

- If $\Gamma \vdash_{\Sigma} M \sim N \downarrow A$, then $\llbracket \Gamma \rrbracket \vdash_{\llbracket \Sigma \rrbracket} M : \llbracket A \rrbracket$ and $\llbracket \Gamma \rrbracket \vdash_{\llbracket \Sigma \rrbracket} N : \llbracket A \rrbracket$.
- If $\Gamma \vdash_{\Sigma} M \sim N \uparrow A$, then $\llbracket \Gamma \rrbracket \vdash_{\llbracket \Sigma \rrbracket} M : \llbracket A \rrbracket$ and $\llbracket \Gamma \rrbracket \vdash_{\llbracket \Sigma \rrbracket} N : \llbracket A \rrbracket$.
- If $\Gamma \vdash_{\Sigma} A : Type$, then $\llbracket \Gamma \rrbracket \vdash_{\llbracket \Sigma \rrbracket} A$ **sv**.

Hence when the dependent type judgments hold, we can interpret the relevant terms or type.

In Scunak, we further restrict the type theory to second-order. We consider **obj**, **prop**, **pf** P and **class** ϕ to be first-order types. For all $x : B \in \Gamma$, we insist B is a second-order type, i.e., of the form $\Pi x^1 : C^1 \dots \Pi x^n : C^n. C^{n+1}$ where each C^i is a first-order type. For all $c : A \in \Sigma$, A must be a third-order type of the form $\Pi x^1 : B^1 \dots \Pi x^n : B^n. C$ where each B^i is a second-order type and C is a first-order type.

3 Mac Lane Set Theory with Universes

The axiomatic kernel of Mac Lane set theory with Universes (abbreviated **MU**) is implemented in Scunak using a signature of 29 constants. (In particular, the signature is finite.) There are three constants for constructing propositions.

- $\neg : \text{prop} \rightarrow \text{prop}$, i.e., $\neg M$ is a proposition whenever M is a proposition.

$$\begin{array}{c}
\frac{x : A \in \Gamma}{\Gamma \vdash x \sim x \downarrow A} xv \quad \frac{c : A \in \Sigma}{\Gamma \vdash c \sim c \downarrow A} xs \quad \frac{\Gamma \vdash M \sim N \downarrow \text{class } \phi}{\Gamma \vdash \pi_2(M) \sim \pi_2(N) \downarrow \text{pf } (\phi \pi_1(M))} xpi2 \\
\frac{\Gamma \vdash M \sim P \downarrow (\Pi x : A.B) \quad \Gamma \vdash N \sim Q \uparrow A}{\Gamma \vdash (MN) \sim (PQ) \downarrow ([N/x]B)} xa \quad \frac{\Gamma \vdash M \sim N \downarrow \text{class } \phi}{\Gamma \vdash \pi_1(M) \sim \pi_1(N) \downarrow \text{obj}} xpi1 \\
\frac{\Gamma, z : A \vdash [z/\mathbf{x}_\lambda^M] \mathbf{B}_\lambda^M \sim [z/\mathbf{x}_\lambda^N] \mathbf{B}_\lambda^N \uparrow [z/x]B \quad z \in \mathcal{V} \text{ fresh}}{\Gamma \vdash M \sim N \uparrow (\Pi x : A.B)} c\lambda^z \\
\frac{\Gamma \vdash_\Sigma \text{fst}^M \sim \text{fst}^N \uparrow \text{obj} \quad \Gamma \vdash_\Sigma \text{snd}^M \sim \text{snd}^N \uparrow \text{pf } (\phi \text{fst}^M)}{\Gamma \vdash_\Sigma M \sim N \uparrow \text{class } \phi} cp \\
\frac{\Gamma \vdash M \sim N \downarrow B \quad B \in \{\text{obj}, \text{prop}\}}{\Gamma \vdash M \sim N \uparrow B} coerce \\
\frac{\Gamma \vdash M \sim M \downarrow \text{pf } Q \quad [L] \vdash_{[\Sigma]} Q : \text{prop} \quad \Gamma \vdash Q^\perp \sim P \uparrow \text{prop}}{\Gamma \vdash_\Sigma M \sim N \uparrow \text{pf } P} \frac{\Gamma \vdash N \sim N \downarrow \text{pf } R \quad [L] \vdash_{[\Sigma]} R : \text{prop} \quad \Gamma \vdash R^\perp \sim P \uparrow \text{prop}}{coercepf} \\
\frac{\Gamma \vdash A : \text{Type} \quad \Gamma, z : A \vdash [z/x]B : \text{Type} \quad z \in \mathcal{V} \text{ fresh}}{\Gamma \vdash (\Pi x : A.B) : \text{Type}} vt\Pi \quad \frac{}{\Gamma \vdash \text{obj} : \text{Type}} vto \\
\frac{}{\Gamma \vdash \text{prop} : \text{Type}} vtp \quad \frac{\Gamma \vdash M \sim M \uparrow \text{prop}}{\Gamma \vdash \text{pf } M : \text{Type}} vtpf \quad \frac{\Gamma \vdash M \sim M \uparrow (\text{obj} \rightarrow \text{prop})}{\Gamma \vdash \text{class } M : \text{Type}} vtcl
\end{array}$$

Fig. 1. Rules for Typing Judgments

- $\in : \text{obj} \rightarrow \text{obj} \rightarrow \text{prop}$, i.e., if x and y are objects (sets), then $(x \in y)$ (infix for $(\in yx)$) is a proposition.¹
- $= : \text{obj} \rightarrow \text{obj} \rightarrow \text{prop}$, i.e., if x and y are objects (sets), then $(x = y)$ (infix for $(= xy)$) is a proposition.

Note that we take negation as the *only* logical connective. The other connectives will be definable (making use of sets). Bounded quantifiers will also be definable.

There are six constants corresponding to basic set constructors.

- $\emptyset : \text{obj}$, the empty set is an set.
- $\text{dsetconstr} : \Pi A : \text{obj}.(\text{class } (\in A) \rightarrow \text{prop}) \rightarrow \text{obj}$, i.e., if A is a set and ϕ is a predicate on the set A , then $(\text{dsetconstr } A \phi)$ (informally written $\{x \in A \mid \phi(x)\}$) is a set. In the future, we will often write the class type $\text{class } (\in A)$ induced by the set A simply as A . The constant dsetconstr corresponds to the separation axiom.

¹ The order of the arguments are reversed so that $(\in y)$, representing the predicate version of the set y , has a nice η -short form.

- **setadjoin** : $\mathbf{obj} \rightarrow \mathbf{obj} \rightarrow \mathbf{obj}$, i.e., if A and B are sets, then $\{A\} \cup B$ is a set. We take this operation of adjoining to sets (or, *cons* operation) as primitive instead of the more common primitive: unordered pair.
- \mathcal{P} : $\mathbf{obj} \rightarrow \mathbf{obj}$, i.e., if A is a set, then the powerset $\mathcal{P}(A)$ of A is a set.
- \bigcup : $\mathbf{obj} \rightarrow \mathbf{obj}$, i.e., if A is a set, then the union $\bigcup A$ of A is a set.
- **univ** : $\mathbf{obj} \rightarrow \mathbf{obj}$, i.e., if A is a set then the “universe” of A is a set. A universe of a set A is a set containing A and closed under **dsetconstr**, **setadjoin**, \mathcal{P} and \bigcup . Universes are especially useful when one begins to represent so-called “large” categories.

Note that one can easily represent any finite enumeration $\{x_1, \dots, x_n\}$ using the empty set and the **setadjoin** operation: $(\{x_1\} \cup \dots (\{x_n\} \cup \emptyset) \dots)$.

While one could take more constructors for propositions and objects as primitive, there is a very important reason to prefer a minimal set: *primitive substitutions*. When forming a complete automated reasoning procedure, it seems inevitable that sometimes one will need to “guess” the use of one of the basic constants in an instantiation. In higher-order theorem proving, such guessing is performed by primitive substitutions (or, *primsubs*). In Church’s type theory, the set of possible primsubs is infinite since there is a logical constant Π^α for each type α . In formulating **MIU**, we have attempted to keep this set not only finite but also as small as possible.

We can easily interpret the constants above in our semantics. Briefly,

$$\begin{aligned} \llbracket \neg \rrbracket(\dot{\in}, x, y) &:= (\dot{\notin}, x, y) & \llbracket \neg \rrbracket(\dot{\notin}, x, y) &:= (\dot{\in}, x, y) & \llbracket \neg \rrbracket(\dot{=}, x, y) &:= (\dot{\neq}, x, y) \\ \llbracket \neg \rrbracket(\dot{\neq}, x, y) &:= (\dot{=}, x, y) & \llbracket \in \rrbracket(x)(y) &:= (\dot{\in}, x, y) & \llbracket = \rrbracket(x)(y) &:= (\dot{=}, x, y) \\ \llbracket \emptyset \rrbracket &:= \emptyset & \llbracket \mathcal{P} \rrbracket(A) &:= \mathcal{P}(A) & \llbracket \bigcup \rrbracket(A) &:= \bigcup(A). \end{aligned}$$

Also, $\llbracket \mathbf{setadjoin} \rrbracket(A)(B) := \{A\} \cup B$. Interpreting **dsetconstr** requires a bit more explanation. Let $A \in \mathcal{D}_{\mathbf{obj}}$ and $\phi \in \mathcal{D}_{\mathbf{obj} \times \rho \rightarrow \mathbf{prop}}$ be given. The intention is that ϕ is a property of the set A . As such, ϕ depends on an object $x \in \mathcal{D}_{\mathbf{obj}}$ and a proof that x is in A . In our frame, $(\dot{\in}, x, A)$ is the only possible proof, and is only a proof if $(\dot{\in}, x, A) \in \mathcal{D}_\rho$ (i.e., x is actually in A). We let $\llbracket \mathbf{dsetconstr} \rrbracket(A)(\phi) := \{x \in A \mid \phi(\langle x, (\dot{\in}, x, A) \rangle) \in \mathcal{D}_\rho\}$. Also, we interpret $\llbracket \mathbf{univ} \rrbracket(A)$ to be $V_{\delta+\omega} \in V_{\omega^2}$ where $\delta < \omega^2$ is an ordinal such that $A \in V_\delta$.

Finally, there are 20 constants corresponding to natural deduction proof rules. These are shown in natural deduction style in Figure 2. To ease the presentation, some of the rules in Figure 2 are simplified. For example, the premiss $\Gamma \vdash P, Q : \mathbf{prop}$ of the rule **xcases** stands for two premisses: $\Gamma \vdash P : \mathbf{prop}$ and $\Gamma \vdash Q : \mathbf{prop}$. Variables named A, B, C and D are always of type **obj** and variables named P and Q are always of type **prop**. When we add these variables to the context Γ , we do not explicitly write the type. Also, we sometimes write $a : \phi$ for $a : \mathbf{class} \ \phi$ and we write $b : A$ for $b : \mathbf{class} \ (\in A)$. These rules are “sound” in our intended semantics. That is, there is an obvious interpretation (respecting the relevant pers). For example, we can define **setunionI** so that $\llbracket \mathbf{setunionI} \rrbracket(A)(B)(C)(\dot{\in}, B, C)(\dot{\in}, C, A) := (\dot{\in}, B, \bigcup A)$ for $A, B, C \in V_{\omega^2}$ and $(\dot{\in}, B, C), (\dot{\in}, C, A) \in \mathcal{D}_\rho$. Note that given these arguments, $(\dot{\in}, B, \bigcup A) \in \mathcal{D}_\rho$ precisely because $B \in C$ and $C \in A$ and so $B \in \bigcup A$. Given arguments from

\mathcal{D}_ρ which do not fall into the pattern above, `setunionI` can be arbitrary in \mathcal{D}_ρ . Only the pattern above is used to check `[[setunionI]]` is in the domain of the per determined by the type

$$\mathit{II}A : \mathit{obj}.\mathit{II}B : \mathit{obj}.\mathit{II}C : \mathit{obj}.\mathit{pf} (B \in C) \rightarrow \mathit{pf} (C \in A) \rightarrow \mathit{pf} (B \in \bigcup A)$$

so that the interpretation will respect the signature.

The rules `xmcases` and `notE` are natural deduction rules for classical negation. Consider the typed constants corresponding to these rules:

$$\begin{aligned} \mathit{xmcases}: \mathit{II}P : \mathit{prop}.\mathit{II}Q : \mathit{prop}.\mathit{pf} P \rightarrow \mathit{pf} Q &\rightarrow (\mathit{pf} (\neg P) \rightarrow \mathit{pf} Q) \rightarrow \mathit{pf} Q \\ \mathit{notE}: \mathit{II}P : \mathit{prop}.\mathit{II}Q : \mathit{prop}.\mathit{pf} P \rightarrow \mathit{pf} (\neg P) &\rightarrow \mathit{pf} Q \end{aligned}$$

One can use these two basic rules to derive a natural deduction rule for negation introduction `notIp` as well as the classical double negation rule `dnegE`. In terms of the type theory, we make two abbreviations using terms of the appropriate types:

$$\begin{aligned} \mathit{notIp}: (\mathit{II}P : \mathit{prop}.\mathit{pf} P \rightarrow \mathit{II}Q : \mathit{prop}.\mathit{pf} Q) &\rightarrow \mathit{pf} (\neg P) \\ &= (\lambda P \lambda u (\mathit{xmcases} P (\neg P) (\lambda v (u v (\neg P)))) (\lambda w w)) \\ \mathit{dnegE}: (\mathit{II}P : \mathit{prop}.\mathit{pf} (\neg \neg P) \rightarrow \mathit{pf} P) & \\ &= (\lambda P \lambda u (\mathit{xmcases} P P (\lambda v v) (\lambda w (\mathit{notE} (\neg P) P w u)))) \end{aligned}$$

This demonstrates how one represents **MU** theorems and proofs in Scunak. A “theorem” is a function type which returns a proof type and a “proof” is a term which can be judged to be a member of this type.

The rule `eqCE` allows one to replace equals by equals even when the objects are used in a “typed” way. Suppose ϕ is a predicate and a and b are in the class type of ϕ . Technically, a is a pair of an untyped set $\pi_1(a)$ and a proof $\pi_2(a)$ of $(\phi \pi_1(a))$. The “typed” object b is a similar pair. We cannot directly represent the proposition that a and b are equal, since we only have equality between objects (i.e., untyped sets). The proposition that a and b are equal as *untyped sets* is $\pi_1(a) = \pi_1(b)$ (one of the premisses of `eqCE`). Suppose ψ is a predicate that is only defined relative to the class ϕ . If we know ψ is true for the ϕ -object a , and we know a and b are equal as untyped sets, then `eqCE` allows us to conclude ψ is true for the ϕ -object b .

The only other rules which may require any explanation are `dsetconstrI`, `dsetconstrEL` and `dsetconstrER` for introducing and eliminating the (dependent) set constructor. In each case we have premisses indicating A is an object (i.e., untyped set) in context and ϕ is a predicate on A . Since ϕ can only be applied to members of the class type `class` $(\in A)$, we must take care.

In the introduction rule `dsetconstrI`, we assume we have b as a member of this class type and a proof that (ϕb) holds. In such a case we wish to conclude that b is in the set $\{x \in A | (\phi x)\}$. This is not quite correct since \in is a relation between objects, and b is a pair. Hence we recover the object using π_1 and conclude $\pi_1(b) \in \{x \in A | (\phi x)\}$.

The elimination rules `dsetconstrEL` and `dsetconstrER` form a kind of converse. Assume we have an untyped set B and a proof that B is in $\{x \in A | (\phi x)\}$. The first rule `dsetconstrEL` allows us to conclude $B \in A$. The second rule `dsetconstrER` intuitively allows us to conclude B satisfies ϕ . However, ϕ cannot be simply applied to B . Instead we *coerce* the untyped object B to be in

$\frac{\Gamma \vdash P, Q : \text{prop} \quad \Gamma, P \vdash Q \quad \Gamma, \neg P \vdash Q}{\Gamma \vdash Q} \text{ xmcases}$	$\frac{\Gamma \vdash P, Q : \text{prop} \quad \Gamma \vdash P \quad \Gamma \vdash \neg P}{\Gamma \vdash Q} \text{ notE}$
$\frac{\Gamma, A \vdash (\phi A) : \text{prop} \quad \Gamma \vdash a, b : \phi \quad \Gamma, c : \phi \vdash (\psi c) : \text{prop} \quad \Gamma \vdash \pi_1(a) = \pi_1(b) \quad \Gamma \vdash \psi a}{\Gamma \vdash \psi b} \text{ eqCE}$	
$\frac{\Gamma \vdash A, B : \text{obj} \quad \Gamma, C, C \in A \vdash C \in B \quad \Gamma, D, D \in B \vdash D \in A}{\Gamma \vdash A = B} \text{ setext}$	
$\frac{\Gamma \vdash A : \text{obj} \quad \Gamma \vdash A \in \emptyset \quad \Gamma \vdash P : \text{prop}}{\Gamma \vdash P} \text{ emptysetE}$	
$\frac{\Gamma \vdash A : \text{obj} \quad \Gamma, a : A \vdash (\phi a) : \text{prop} \quad \Gamma \vdash b : A \quad \Gamma \vdash \phi b}{\Gamma \vdash \pi_1(b) \in \{x \in A \mid \phi x\}} \text{ dsetconstrI}$	
$\frac{\Gamma \vdash A : \text{obj} \quad \Gamma, a : A \vdash (\phi a) : \text{prop} \quad \Gamma \vdash B : \text{obj} \quad \Gamma \vdash B \in \{x \in A \mid \phi x\}}{\Gamma \vdash B \in A} \text{ dsetconstrEL}$	
$\frac{\Gamma \vdash A : \text{obj} \quad \Gamma, a : A \vdash (\phi a) : \text{prop} \quad \Gamma \vdash B : \text{obj} \quad \Gamma \vdash u : (B \in \{x \in A \mid \phi x\})}{\Gamma \vdash \phi(B, (\text{dsetconstrEL } A \phi B u))} \text{ dsetconstrER}$	
$\frac{\Gamma \vdash A, B : \text{obj}}{\Gamma \vdash A \in \{A\} \cup B} \text{ setadjoinIL}$	$\frac{\Gamma \vdash A, B : \text{obj} \quad \Gamma \vdash C : \text{obj} \quad \Gamma \vdash C \in B}{\Gamma \vdash C \in \{A\} \cup B} \text{ setadjoinIR}$
$\frac{\Gamma \vdash A, B, C : \text{obj} \quad \Gamma \vdash C \in \{A\} \cup B \quad \Gamma \vdash P : \text{prop} \quad \Gamma, C = A \vdash P \quad \Gamma, C \in B \vdash P}{\Gamma \vdash P} \text{ setadjoinE}$	
$\frac{\Gamma \vdash A, B : \text{obj} \quad \Gamma, C, C \in B \vdash C \in A}{\Gamma \vdash B \in \mathcal{P}(A)} \text{ powersetI}$	
$\frac{\Gamma \vdash A, B, C : \text{obj} \quad \Gamma \vdash B \in \mathcal{P}(A) \quad \Gamma \vdash C \in B}{\Gamma \vdash C \in A} \text{ powersetE}$	
$\frac{\Gamma \vdash A, B, C : \text{obj} \quad \Gamma \vdash B \in C \quad \Gamma \vdash C \in A}{\Gamma \vdash B \in \bigcup A} \text{ setunionI}$	
$\frac{\Gamma \vdash A, B : \text{obj} \quad \Gamma \vdash B \in \bigcup A \quad \Gamma \vdash P : \text{prop} \quad \Gamma, C, B \in C, C \in A \vdash P}{\Gamma \vdash P} \text{ setunionE}$	
$\frac{\Gamma \vdash A : \text{obj}}{\Gamma \vdash A \in \text{univ}(A)} \text{ univHas}$	$\frac{\Gamma \vdash A : \text{obj} \quad \Gamma \vdash a : \text{univ}(A) \quad \Gamma, b : \pi_1(a) \vdash (\phi b) : \text{prop}}{\Gamma \vdash \{x \in \pi_1(a) \mid \phi x\} \in \text{univ}(A)} \text{ univSep}$
$\frac{\Gamma \vdash A : \text{obj} \quad \Gamma \vdash a : \text{univ}(A) \quad \Gamma \vdash b : \text{univ}(A)}{\Gamma \vdash \{\pi_1(a)\} \cup \pi_1(b) \in \text{univ}(A)} \text{ univAdj}$	
$\frac{\Gamma \vdash A : \text{obj} \quad \Gamma \vdash a : \text{univ}(A)}{\Gamma \vdash \mathcal{P}(\pi_1(a)) \in \text{univ}(A)} \text{ univPow}$	$\frac{\Gamma \vdash A : \text{obj} \quad \Gamma \vdash a : \text{univ}(A)}{\Gamma \vdash \bigcup \pi_1(a) \in \text{univ}(A)} \text{ univSU}$

Fig. 2. Basic Set Theory Deduction Rules

the class determined by A using the previous rule `dsetconstrEL`. The term `(dsetconstrEL A ϕ B)` takes a proof of $B \in \{x \in A \mid (\phi x)\}$ (the premiss named by u in `dsetconstrER`) to a proof of $B \in A$. Hence $\langle B, (\text{dsetconstrEL } A \phi B u) \rangle$ is in the class type `(class (\in A))`.

The basic set theoretic concepts and rules satisfy a kind of *logical purity*. Unlike most presentations of set theory, the basic rules only mention the basic concepts. In particular, all “axioms” are given before any definitions are made.

Although one starts with negation, one can define the other logical connectives and derive the appropriate rules. For example, for any propositions P and Q , one can define disjunction of P and Q using the term

$$\{\emptyset\} \in \{\{x \in \{\emptyset\} \mid P\}, \{x \in \{\emptyset\} \mid Q\}\}.$$

Once the appropriate rules for disjunction are proven, one need not unfold this definition of disjunction.

One can also define bounded quantifiers. For example, for a set A and predicate $\phi(x)$ depending on an element $x \in A$, $\forall x \in A. \phi(x)$ can be defined by $(\{x \in A. \phi(x)\} = A)$. Note that the predicate ϕ is relative to the set A . That is, $\phi(x)$ is only defined for x in the class type corresponding to A . Thus one could sensibly represent a proposition such as $\forall x \in \textit{Positive}. \frac{1}{x} > 0$ where *Positive* represents the set of positive real numbers. Forming the term $\frac{1}{x}$ would require a proof that x is nonzero and such a proof would depend on the assumption $x \in \textit{Positive}$.

Note that two common set theory axioms, choice and foundation, are omitted from **MU**.

Starting from the basic concepts and rules defining the theory **MU**, we can make new definitions using terms of certain types. If the type returns a proof type, then we can interpret the type as a derived rule (or theorem) and we can interpret the term as a proof. We briefly outline some definitions which have been constructed interactively in Scunak. After getting starting, this follows the usual development of basic mathematics in an axiomatic set theory.

- Propositional connectives and bounded quantifiers are defined and corresponding rules are derived.
- The usual set theoretic notions are defined (\subset , binary union and intersection, etc.) are defined and relevant rules are derived.
- A description operator is defined on the class of singleton sets.
- Ordered pairs (as Kuratowski pairs) are defined, along with “first” and “second” operations which are only defined on the class of Kuratowski pairs. (To prevent confusion with pairing in the Scunak type theory, we write $\langle\langle x, y \rangle\rangle$ for the Kuratowski pair of x and y .)
- Using Kuratowski pairs, Cartesian products $(A \times B)$ of sets A and B are defined.
- Given any sets A and B and (meta-level) relation $\phi : A \rightarrow B \rightarrow \mathbf{prop}$, we can define the subset of $A \times B$ of Kuratowski pairs $\langle\langle a, b \rangle\rangle$ such that (ϕab) holds. In Scunak, we include notation of the form $\{\langle\langle x, y \rangle\rangle : A \times B \mid \dots\}$ for

specifying such sets of pairs. This is especially useful when defining constructors for functions and relations.

- Using Cartesian products, binary relations are defined.
- For any two sets A and B , a function from A to B is defined as a functional (untyped) binary relation on A and B .
- For sets A and B , we define the *set* B^A of functions from A to B .
- Using the description operator, an application operator `ap2` can be defined taking an element f of the set B^A and an element a of A to (intuitively) $f(a)$ in B .
- Given a “meta-level” function g from A to B (i.e., a term g of type $A \rightarrow B$), we can obtain an “object-level” function (`lam2 A B g`) from A to B (i.e., a member of B^A) by $\{\langle x, y \rangle \in (A \times B) \mid (\pi_1(gx) = \pi_1(y))\}$. This is an internalized λ binder and (along with `ap2`) allows us to internalize standard models of Church’s type theory.

Of particular note are the operations that are “partial” (i.e., only defined on subclasses of the untyped universe). Consider first the description operator. The predicate `singleton` is defined in the obvious way. One can easily prove if U is in the class of singletons, then $(\bigcup U) \in U$. Using the previous fact, we define a description operator `the` taking a singleton $U : (\text{class singleton})$ into (the class determined by) U . The important fact is that the description operator is defined *precisely* on the class of singleton sets. The type of `the` is $\Pi U : (\text{class singleton}).U$. The definition of `the` is $(\lambda U (\bigcup U, (\text{theprop } U)))$ where `theprop` is an abbreviation for the proof, given a singleton U , that $(\bigcup U) \in U$.

Two other examples of such “partial” functions are given by the first and second operators for Kuratowski pairs. Both `kfst` and `ksnd` take an argument u of type `(class iskpair)` and return an object (i.e., an element of type `obj`).

4 Proving the Injective Cantor Theorem

A common example which has been considered many times is Cantor’s Theorem. Intuitively, Cantor’s Theorem states that $\mathcal{P}(A)$ is bigger than A . One way to formally state this property is that there is no surjection from A onto $\mathcal{P}(A)$. This is the *surjective Cantor Theorem* which was one of the earliest interesting theorems proven automatically by TPS [3]. The relevant diagonal set can be constructed by TPS using higher-order unification.

Otter was also able to prove the surjective Cantor theorem formalized in NBG. As discussed in [20], the diagonal set was defined by the user and certain lemmas about this diagonal set were explicitly given. For this reason, Quaife describes the proof as “semi-automatic.”

An alternative formulation of Cantor’s Theorem states that there is no injection from $\mathcal{P}(A)$ into A . This is the *injective Cantor Theorem*. As discussed in [3], this is a very challenging problem for higher-order theorem provers. The only known cut-free proofs of the injective Cantor Theorem are of quantificational depth at least 3. (Intuitively, one must do a `primsb` for a variable introduced by a quantifier which itself was introduced by a `primsb`.)

A different approach was suggested by Dana Scott: reduce the injective version to the surjective version by using the fact that an injection h from $\mathcal{P}(A)$ to A induces a surjection g from A to $\mathcal{P}(A)$. It is not reasonable to expect a theorem prover to “guess” such a lemma, but it may be reasonable for a theorem prover to find such a lemma in a library and use it, along with the surjective Cantor Theorem, to prove the injective Cantor Theorem. This is the “semi-automatic” approach we have taken here.

First we must represent the main lemma about one-sided inverse functions. A general version is not true: there can be an injection from A to B without there existing a surjection from B onto A . The counterexample is when A is empty and B is nonempty. We could formulate the lemma as follows: If A is nonempty and there is an injection from A to B , then there is a surjection from B onto A . Given a default value $a \in A$ and an injection h from A to B , we can actually *define* the relevant surjection g as follows:

$$\{\langle y, x \rangle \mid \langle x, y \rangle \in h\} \cup \{\langle y, a \rangle \mid \neg \exists x \in A. \langle x, y \rangle \in h\}. \quad (1)$$

That is, $g(y) = x$ if $h(x) = y$ and $g(y)$ is the default value a if no such x exists.

Let $iF(A, B)$ denote the set of injective functions from A to B and let $sF(A, B)$ denote the set of surjective functions from A to B . We can represent the construction above in Scunak by defining an abbreviation `leftInvOfInj` which takes two sets A and B , a member h of the set $iF(A, B)$ and a member a of the set A and returns a member of the set $sF(B, A)$. Since `leftInvOfInj` returns a member of a class type, it returns a pair. The object part of the pair is defined as indicated by (1) above. The proof part of the pair is a proof that (1) defines a member of $sF(B, A)$ (a surjective function from B to A). Instead of representing the main lemma as a proposition, we define an operation, which we can denote by $(h)_a^{-1}$ taking $h \in iF(A, B)$ and $a \in A$ to a member of $sF(B, A)$.

By the time we state the injective Cantor Theorem in Scunak, we have defined 57 concepts and proven 222 lemmas. Furthermore, 15 of the defined concepts are functions which return elements of class types and hence have proof content (namely, the proof that the resulting untyped object belongs to the class). For each definition (not counting lemmas), there are two rules for folding and unfolding abbreviations, giving 114 more facts. Combining this with the 20 basic proof rules from Figure 2, we have 371 facts which can be used to prove the injective Cantor Theorem.

Among these facts, we only send 114 to Vampire. We do not translate any fact which depends on variables of propositional type. This filters out, for example, the basic rules `xmcases`, `notE` and `eqCE` for negation and equality. However, since we translate negation and equality in Scunak to negation and equality in Vampire, these rules need not be translated. We also only translate types which are first-order (in the λ -calculus sense). This is more restrictive than necessary and filters out some important facts such as the basic set extensionality rule `setext`. Of course, the fewer clauses we send to Vampire, the less likely it becomes that there is a proof, but the more likely it becomes that Vampire will find a proof if one exists. Dependence of object terms on proof terms are deleted in the translation.

In addition to the 114 facts, two formulas corresponding to the injective Cantor theorem are sent to Vampire. Namely, the “axiom” that a constant h is in the set $A^{\mathcal{P}(A)}$ and the “conjecture” that h is not injective.

Vampire (version 8.0) was called with 116 first-order formulas and a five minute time limit. In less than 5 seconds, Vampire generated over 80,000 clauses and found a refutation. Most of the given 116 clauses are not used in the refutation. We describe the relevant clauses used by Vampire:

1. $h \in A^{\mathcal{P}(A)}$.
2. h is injective (the negation of the conclusion).
3. (Surjective Cantor Theorem) If $g \in \mathcal{P}(X)^X$, then g is not surjective.
4. (Main Lemma) If $f \in iF(X, Y)$ and $a \in X$, then $(f)_a^{-1} \in sF(Y, X)$.
5. For any X , $\emptyset \in \mathcal{P}(X)$.
6. If $g \in sF(X, Y)$, then g is surjective.
7. If $g \in sF(X, Y)$, then $g \in Y^X$.
8. If an element f is in Y^X and is injective from X to Y , then $f \in iF(X, Y)$.
9. If $X \subseteq Y$ and $x \in X$, then $x \in Y$.
10. For any X , $\bigcup\{X\} \subseteq X$.
11. (Basic rule `setunionI`) If $x \in X$ and $X \in Y$, then $x \in \bigcup Y$.
12. (Basic rule `setadjoinIL`) For any x and Y , $x \in \{x\} \cup Y$.

The first facts (1-4) are at the heart of the refutation. Fact 5 is used so that the empty set can act as the necessary default value in $\mathcal{P}(A)$. Facts (6-8) must be used to pass from the sets $iF(X, Y)$ or $sF(X, Y)$ to the properties defining these sets. The remaining facts (9-12) are not strictly necessary for the proof, but are used by Vampire. If one modifies the input file for Vampire to include only facts (1-8), then Vampire finds a simpler refutation after generating only 16 clauses.

Upon inspection, the original refutation found by Vampire is a bit round-about. Part of the refutation is clear: By the surjective Cantor theorem, nothing is in $sF(X, \mathcal{P}(X))$ for any X . Using this with main lemma, if something is in $\mathcal{P}(X)$, then nothing is in $iF(\mathcal{P}(X), X)$. At this point, we could finish the refutation using $\emptyset \in \mathcal{P}(A)$ (from 5) and $h \in iF(\mathcal{P}(A), A)$ (from 1, 2 and 8). However, the actual refutation found by Vampire proceeds as follows.

Using 8 and 11, if $f \in Y^X$ is injective and $iF(X, Y) \in Z$, then $f \in \bigcup Z$. Hence (using our assumptions 1 and 2), we conclude $h \in \bigcup Z$ whenever $iF(\mathcal{P}(A), A) \in Z$. Since $iF(\mathcal{P}(A), A) \in \{iF(\mathcal{P}(A), A)\} \cup Y$ by 12, we have $h \in \bigcup(\{iF(\mathcal{P}(A), A)\} \cup Y)$. In particular, $h \in \bigcup\{iF(\mathcal{P}(A), A)\}$. Using 9 and 10, $h \in iF(\mathcal{P}(A), A)$. Hence nothing is in $\mathcal{P}(A)$, contradicting $\emptyset \in \mathcal{P}(A)$.

5 Related Work

A similar idea of axiomatizing set theory in higher-order logic, as well as an internalization of higher-order logic within the set theory, was explored in [11]. The ZF-theory in Isabelle also encodes set theory within a form of simple type theory [17]. More recently, a variety of foundational systems including ZFC (Zermelo-Fraenkel set theory with the axiom of choice) were finitely represented

and compared in Automath [24]. Decidable fragments of set theory have been studied extensively (see, e.g., [8]). Several different forms of mechanized set theory have been formulated and investigated in recent years, including [9], [10], [23], [4] and [24].

The idea of calling a first-order theorem prover to solve subgoals has been explored in several papers, including [15, 1]. The work in [15] describes extensive experiments with a connection between Vampire and Isabelle-ZF. The work in [1] connects a first-order prover with a logical framework.

6 Conclusion

We have presented a type theory with proof irrelevance and class types implemented in the system Scunak. This type theory is designed with the intention of supporting both the construction of a large mathematical library and automated search using the library. Within this type theory, a form of Mac Lane set theory has been implemented. We argue this is a practical representation language for mathematics by noting that we have interactively built enough theory to define function spaces as well as object-level versions of application and abstraction. In particular, we have represented the injective Cantor theorem and mapped this representation (along with much of the previously constructed theory) to the first-order prover Vampire. Vampire proves this challenge problem quickly. Hopefully, in the future we will have a larger library of mathematics in Scunak which can be used effectively during automated search.

References

1. Andreas Abel, Thierry Coquand, and Ulf Norell. Connecting a logical framework to a first-order logic prover. In Bernhard Gramlich, editor, *5th International Workshop on Frontiers of Combining Systems, FroCoS'05, Vienna, Austria, September 19-21, 2005*, Lecture Notes in Computer Science, 2005.
2. Peter B. Andrews and Matthew Bishop. On sets, types, fixed points, and checkers. In Pierangelo Miglioli, Ugo Moscato, Daniele Mundici, and Mario Ornaghi, editors, *Theorem Proving with Analytic Tableaux and Related Methods. 5th International Workshop. (TABLEAUX '96)*, volume 1071 of *Lecture Notes in Artificial Intelligence*, pages 1–15, Terrasini, Italy, May 1996. Springer-Verlag.
3. Peter B. Andrews, Matthew Bishop, and Chad E. Brown. System description: TPS: A theorem proving system for type theory. In David McAllester, editor, *Proceedings of the 17th International Conference on Automated Deduction*, volume 1831 of *Lecture Notes in Artificial Intelligence*, pages 164–169, Pittsburgh, PA, USA, 2000. Springer-Verlag.
4. Arnon Avron. Formalizing set theory as it is actually used. In Andrea Asperti, Grzegorz Bancerek, and Andrzej Trybulec, editors, *MKM*, volume 3119 of *Lecture Notes in Computer Science*, pages 32–43. Springer, 2004.
5. J. G. F. Belinfante. Computer Proofs in Gödel's class theory with equational definitions for composite and cross. *Journal of Automated Reasoning*, 22:311–339, 1999.

6. Robert Boyer, Ewing Lusk, William McCune, Ross Overbeek, Mark Stickel, and Lawrence Wos. Set theory in first-order logic: Clauses for Gödel's axioms. *Journal of Automated Reasoning*, 2:287–327, 1986.
7. Chad E. Brown. A Short Introduction to Scunak, 2006. in preparation, <http://gtps.math.cmu.edu/cebrown/shortintro-scunak.ps>.
8. Domenico Cantone, Calogero G. Zarba, and Rosa Ruggeri-Cannata. A tableau-based decision procedure for a fragment of set theory with iterated membership. *Journal of Automated Reasoning*, 34(1):49–72, 2005.
9. Gilles Dowek. Collections, sets and types. *Mathematical Structures in Computer Science*, 9(1):109–123, 1999.
10. William M. Farmer. Stmm: A set theory for mechanized mathematics. *J. Autom. Reasoning*, 26(3):269–289, 2001.
11. Michael J. C. Gordon. Set theory, higher order logic or both? In Joakim von Wright, Jim Grundy, and John Harrison, editors, *TPHOLs*, volume 1125 of *Lecture Notes in Computer Science*, pages 191–201. Springer, 1996.
12. Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, January 1993.
13. J. Lambek and P. Scott. *Introduction to Higher Order Categorical Logic*. Cambridge University Press, Cambridge, UK, 1986.
14. Saunders Mac Lane. *Mathematics, Form and Function*. Springer-Verlag, 1986.
15. Jia Meng. Integration of interactive and automatic provers. In Manuel Carro and Jesus Correias, editors, *Second CologNet Workshop on Implementation Technology for Computational Logic Systems*, 2003. <http://www.cl.cam.ac.uk/users/jm318/papers/integration.pdf>.
16. Bengt Nordström, Kent Petersson, and Jan Smith. Martin-löf's type theory. In Samson Abramsky et al., editors, *Handbook of Logic in Computer Science*, volume 5. Oxford University Press, 2000.
17. Lawrence C. Paulson. Set Theory for Verification: II. Induction and Recursion. *Journal of Automated Reasoning*, 15(2):167–215, 1995.
18. Frank Pfenning. Intensionality, extensionality, and proof irrelevance in modal type theory. In Joseph Halpern, editor, *Proceedings of the Sixteenth Annual IEEE Symp. on Logic in Computer Science, LICS 2001*, pages 221–. IEEE Computer Society Press, June 2001.
19. Frank Pfenning and Carsten Schürmann. System Description: Twelf—A Meta-Logical Framework for Deductive Systems. In Harald Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction*, volume 1632 of *Lecture Notes in Artificial Intelligence*, pages 202–206, Trento, Italy, 1999. Springer-Verlag.
20. Art Quaipe. *Automated Development of Fundamental Mathematical Theories*. Kluwer Academic Publishers, Norwell, MA, USA, 1992.
21. Jason Reed. Proof irrelevance and strict definitions in a logical framework. Technical Report 02-153, School of Computer Science, Carnegie Mellon University, 2002.
22. A. Riazanov and A. Voronkov. Vampire 1.1. (system description). In *Automated Reasoning, First International Joint Conference, IJCAR*, volume LNAI 2083, pages 376–380. Springer Verlag, January 2001.
23. Raymond Turner. Type inference for set theory. *Theor. Comput. Sci.*, 266(1-2):951–974, 2001.
24. Freek Wiedijk. Is ZF a hack? Comparing the complexity of some (formalist interpretations of) foundational systems for mathematics. *Journal of Applied Logic*, 4, 2006. to appear.