Automated Deduction in the B Set Theory using Deduction Modulo*

Guillaume Bury[1], David Delahaye[1], Damien Doligez[2], Pierre Halmagrand[1], and Olivier Hermant[3]

[1] Cedric/Cnam/Inria, Paris, France,
Guillaume.Bury@inria.fr
David.Delahaye@cnam.fr
Pierre.Halmagrand@inria.fr
[2] Inria, Paris, France,
Damien.Doligez@inria.fr
[3] CRI, MINES ParisTech, Fontainebleau, France,
Olivier.Hermant@mines-paristech.fr

Abstract. We introduce an encoding of the set theory of the B method based on deduction modulo, and to be used for the automated verification of B proof obligations in the framework of the BWare project. The theory of deduction modulo is an extension of predicate calculus that includes rewriting on both terms and propositions. It is well suited for proof search in axiomatic theories because it turns many axioms into rewrite rules. We also present the associated automated theorem prover Zenon Modulo, an extension of Zenon to deduction modulo, along with its backend to the Dedukti universal proof checker, which also relies on deduction modulo, and which allows us to verify the proofs produced by Zenon Modulo. To deal with a large part of proof obligations, Zenon Modulo includes other extensions as well, such as polymorphic types and arithmetic. Finally, we assess our approach with experimental results obtained on the proof obligations of the benchmark of BWare.

Keywords: Automated Deduction, Set Theory, B Method, Deduction Modulo, Typed Proof Search, Zenon Modulo, Dedukti.

1 Introduction

Reasoning within theories, whether decidable or not, has become a crucial point in automated theorem proving. A theory, commonly formulated as a collection of axioms, is often necessary to specify, in a concise and understandable way, the properties of objects manipulated in software proofs, such as lists or arrays. Each theory has its own features and specificities, but a small number of them appear recurrently, among which arithmetic and set theory. For example, the B method relies on a variant of set theory [1], and this theory is supported by some

* This work is supported by the BWare project [12, 15] (ANR-12-INSE-0010) funded by the INS programme of the French National Research Agency (ANR).

tool sets, such as Atelier B, which are used in industry to specify and build, by stepwise refinements, software that is correct by design.

The Atelier B tool set still lacks automation: it comes with built-in automated theorem provers but in general a lot of Proof Obligations (POs), often generated during the refinement process, are left to the user who must discharge them using the interactive theorem prover or by coming, with new proof rules that must be verified at a later stage, to the rescue of the automated theorem provers. Due to the large practical impact of the B method, the BWare project [12, 15] has committed itself to solve this issue by providing a proof platform with several Automated Theorem Provers (ATPs) aiming to support the verification of POs coming from the development of industrial applications.

When dealing with proofs, leaving the axioms and definitions of set theory wandering among the hypotheses is not a reasonable option: first, it induces a combinatorial explosion in the proof search space and second, axioms do not bear any specific meaning that an ATP can take advantage of. To avoid these drawbacks, we propose to replace axioms by rewrite rules, along the lines of deduction modulo [13]. This framework combines a first order proof system with rewrite rules on terms and propositions. Rewriting on propositions is a feature that allows us to go beyond pure first order reasoning without using any axiom. However, we must be careful with the desired properties of the theory, among which consistency, cut elimination, and completeness of proof search methods.

In this paper, we define an encoding of the set theory of the B method, which is formulated as a theory modulo, i.e. a rewrite system rather than a set of axioms. As deduction modulo applies to proof search methods in classical first order logic [6, 13], we also provide this encoding with an extension of the Zenon tableau-based ATP [7] to deduction modulo, getting a tool called Zenon Modulo [11]. In addition, this tool features a backend to Dedukti [5], an universal proof checker that also relies on deduction modulo, in order to verify the automated proofs that are produced.

To cope properly with the POs provided by the industrial partners of the BWare project, we introduce, in this paper, more extensions to Zenon. Among those, there is the ability to perform typed proof search. We therefore provide Zenon with a polymorphic type system, which offers more flexibility in the expressiveness of the theories, and in particular which allows us to deal with theories that rely on elaborate type systems, like the B set theory, which is a typed set theory. Thanks to types, it is possible to extend Zenon further to arithmetic, which is used in many POs of BWare. This extension targets linear arithmetic, and relies on the simplex algorithm [10] for rational problems, as well as on the branch-and-bound method for integer problems.

The paper is organized as follows: in Sec. 2, we describe the extension of Zenon to typed proof search using a polymorphic type system in particular, and we also briefly present the extension to arithmetic; we then introduce, in Sec. 3, the adaptation of Zenon to deduction modulo, as well as our formulation of the B set theory as a theory modulo; finally, in Sec. 4, we describe our implementation and the experimental results obtained on the benchmark of BWare.

2 Typed Proof Search for Zenon

In this section, we describe an extension of the Zenon ATP to typed proof search with a polymorphic type system. This extension is the ground for further extensions, such as arithmetic.

2.1 Polymorphic Type System

Before showing the inference rules of Zenon, which deal with classical first order logic with equality, we extend expressions to polymorphic types à la ML, through a type system in the spirit of [2]. The language of first order expressions with polymorphic types is provided in Fig. 1, where τ is a term-level type, σ a type scheme that may bind type variables, used for polymorphic types, terms and formulas (of type o), and e an expression (term or formula), in which functions and predicates may now be polymorphic and bear type arguments. In this context, formulas may be quantified over types, as long as these quantifications occur before any quantification over terms (not enforced in our simplified Fig. 1). It should be noted that the very expressions of Fig. 1 are used during the proof search. This explains why there are metavariables (capitalized here, often named free variables in tableau-related literature), which are used to find instantiations by unification, as well as ϵ-terms ($\epsilon(x).P(x)$ is a term/type meaning some x that satisfies $P(x)$, if it exists), an alternative to Skolem terms. In the sequel, we write $e \neq_\tau e'$ for $\neg(e =_\tau e')$, and f for $f()$ when f has arity 0.

To introduce typing judgments, we need contexts. They contain pairs of symbols with a type: the global context Γ_G contains function/predicate symbols and constructors, while the local context Γ_L contains term and type variables. A typing context Γ is a pair $\Gamma_G; \Gamma_L$.

We present in Fig. 2 the inference rules for a single (as customary for PTS) typing judgment for expressions, types, constructors, function/predicate symbols, and the Type constant. The relation $\Gamma \vdash t : \sigma$ means that, in the context Γ the term (type, expression, etc.) t is well-typed of type (scheme) σ. To factorize rules (e.g., $\forall$, Meta or Sym), we embed the term-level types into Kind by the Sub rule. So κ : Kind is either a mere term-level type, or Type. Similarily, κ_o is either κ, or o. The usual freshness proviso holds in the rules $\mathrm{WF}_{1/2}$, App, Var, $\forall$, $\exists$, ϵ, where in the four last cases, $\Gamma, x : \kappa$ denotes $\Gamma_G; \Gamma_L, x : \kappa$.

Γ_G being provided by the built-in theory once and for all, we often elide it, writing $t : \sigma$ for $\Gamma_G; \emptyset \vdash t : \sigma$, and $t_1, \ldots, t_n : \sigma$ for $t_i : \sigma, i = 1 \ldots n$. In addition, we write $\forall x_1, \ldots, x_n : \kappa.e$ for $\forall x_1 : \kappa. \ldots . \forall x_n : \kappa.e$, where κ is a Kind; as well as for the other binders $\exists$ and Π.

2.2 Proof Search Rules

The inference rules summarized in Figs. 3 and 4 are an adaptation of the rules of Zenon [7] to typed formulas. For the sake of simplicity, the unfolding and extension rules have been omitted. The "|" symbol is used to separate two distinct

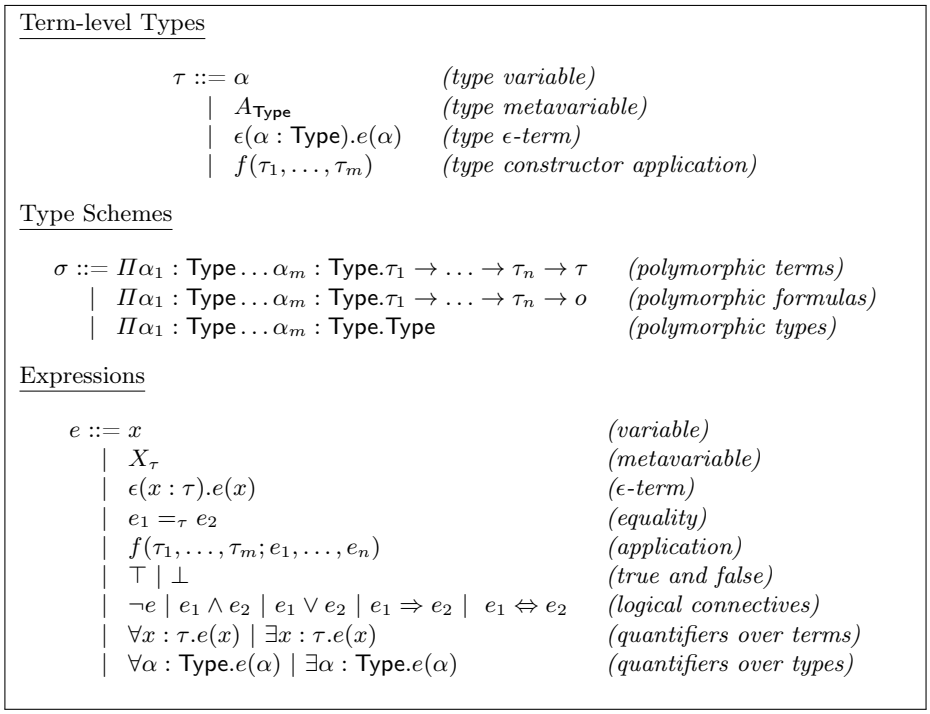


Fig. 1. Types, Type Schemes, and Expressions

nodes to be created and R_r, R_s, R_t, and R_{ts} denote respectively reflexive, symmetric, transitive, and transitive-symmetric relations (that includes equality). Note that rules $\delta_\exists$, $\delta_{\neg\forall}$, $\gamma_{\forall M}$ and $\gamma_{\neg\exists M}$ also deal with quantification on types.

The proof search algorithm uses the usual tableau method: starting from the negation of the goal, apply the rules in a top-down fashion to build a tree. When all branches are closed (i.e. end with a closure rule), the tree is closed, and this closed tree is a proof of the goal. This is done in strict depth-first order: we close the current branch before we start working on another branch. Moreover, we work in a non-destructive way: extending a branch will never change the formulas on another branch.

Given an initially well-typed formula, it should be noted that the inference rules generate only well-typed formulas, that all can be typed in the empty local context (we use Church-style ϵ-terms, decorating the bound variable with its type, and the metavariables carry their types), which explains the simplified form of the typing side conditions.

In presence of polymorphic theories, this typed version of Zenon allows us to narrow the search space and produce smaller proofs, since the typing constraints are handled at the metalevel, while the untyped version of Zenon needs a first order encoding of the polymorphic layer, as in [4]. This will become clear in

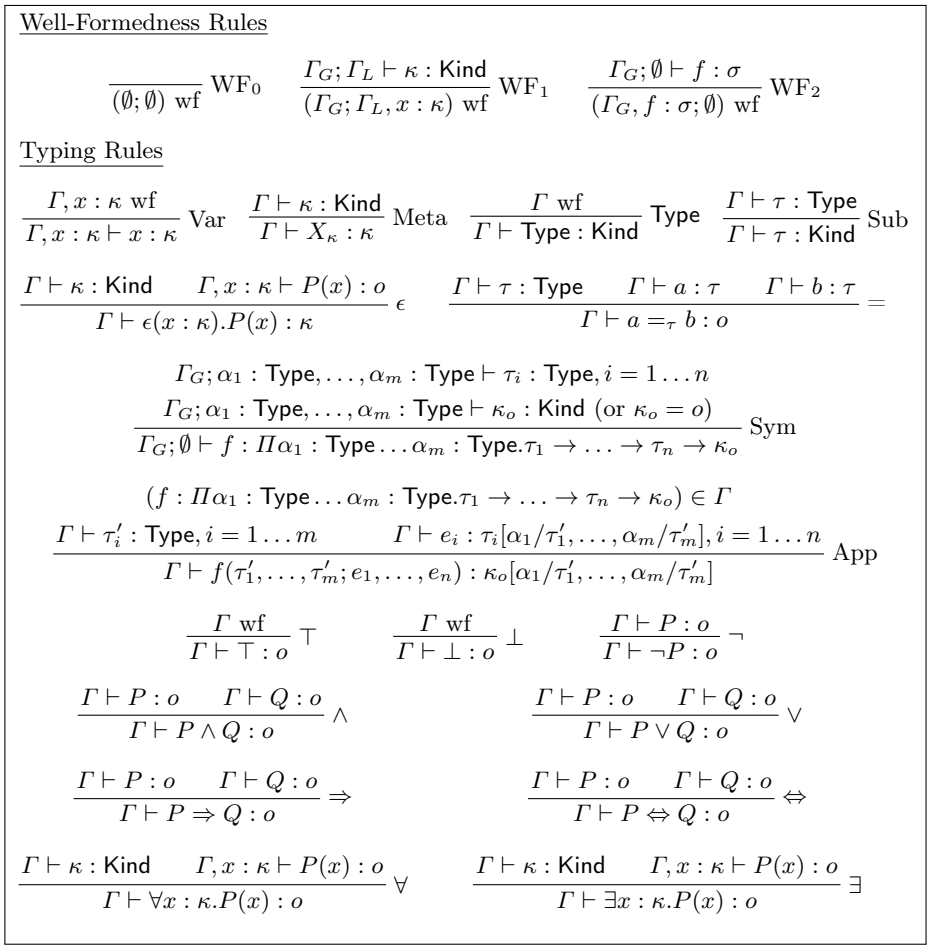


Fig. 2. Typing Inference Rules

Sec. 4 with the comparative benchmark provided by the BWare project in the framework of the B set theory, which is typed.

2.3 Handling of Metavariables

In Zenon, metavariables play a special role. They are actually not variables as they are never substituted. They are introduced by the rules $\gamma_{\forall M}$ and $\gamma_{\neg\exists M}$, and only serve when faking a closure rule to determine a substitution by unification. But instead of applying this substitution in all the tableau, we instantiate the initial formulas that introduced the metavariables (using now the rules $\gamma_{\forall\text{inst}}$ and $\gamma_{\neg\exists\text{inst}}$). A single metavariable may therefore be instantiated many times.

In presence of polymorphism, type metavariables may also be introduced (by the same rules). In contrast with term metavariables, we can afford more

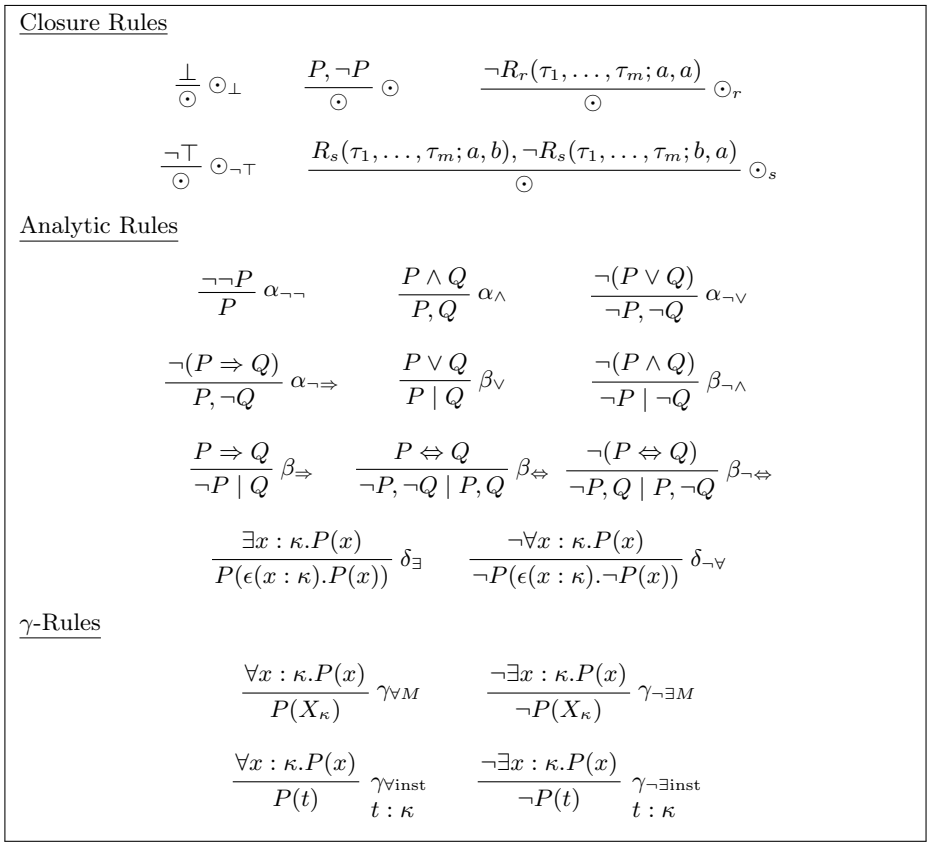


Fig. 3. Proof Search Rules (Part 1)

simplicity. Figs. 3 and 4 have rules with non-linearity constraints (e.g., in the pred rule) and side conditions (e.g., in the $\gamma_{\forall\text{inst}}$ rule) on types. Type constraints do not turn into regular branch formulas, and generate instantiations. When trying to apply a rule with type constraints, we look for a (type metavariable) substitution that satisfies them, rather than postponing this to closure. In case of success, as for terms, we instantiate the initial formulas. This shortcut minimizes both the search space and the size of proof trees.

As an example, let us consider a relation $P : \Pi\alpha : \mathsf{Type}.\alpha \to \alpha \to o$, a type constant $\tau : \mathsf{Type}$, and two constants $a, b : \tau$. We want to prove $P(\tau; a, b)$ under the hypothesis $\forall\alpha : \mathsf{Type}.\forall x, y : \alpha.P(\alpha; x, y)$. The proof is given in Fig. 5 (before pruning of useless formulas; see [7] for more information about pruning) where, once the formula $P(A_{\mathsf{Type}}; X_{A_{\mathsf{Type}}}, Y_{A_{\mathsf{Type}}})$ is introduced, we try to apply the pred rule, which requires first instantiating the type metavariable A_{Type} with τ.

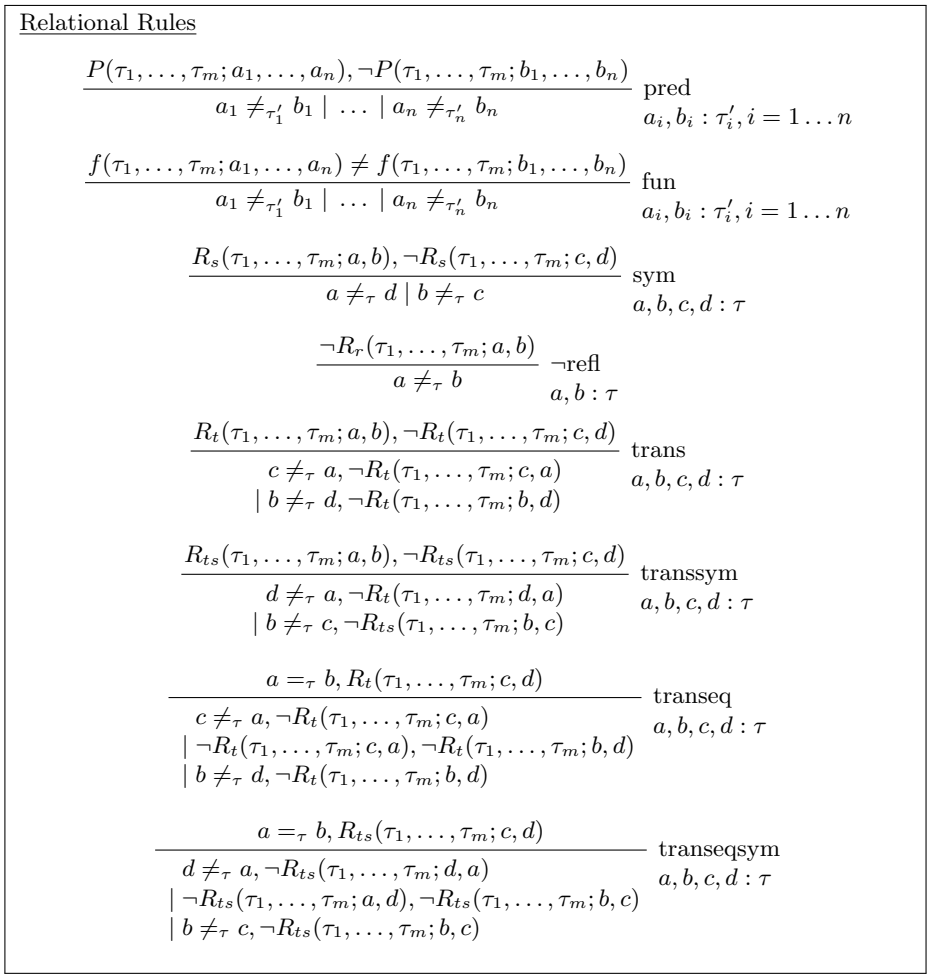


Fig. 4. Proof Search Rules (Part 2)

2.4 Extension to Arithmetic

On this baseline typed system, we can build typed extensions. This is how we have integrated linear arithmetic (i.e. formulas involving equalities or inequalities of linear combinations of real or integer variables) to Zenon. This extension has been essentially designed to handle the POs involving arithmetic of the benchmark provided by the BWare project (see Sec. 4).

In this case, the need for typing is, however, rather loose as we only have to distinguish arithmetic expressions from other expressions. This extension of Zenon consists of an additional set of proof search rules (due to lack of space, we do not provide them in this paper), which can be interleaved with the other rules. These new rules rely on the general simplex algorithm [10] (i.e. the usual

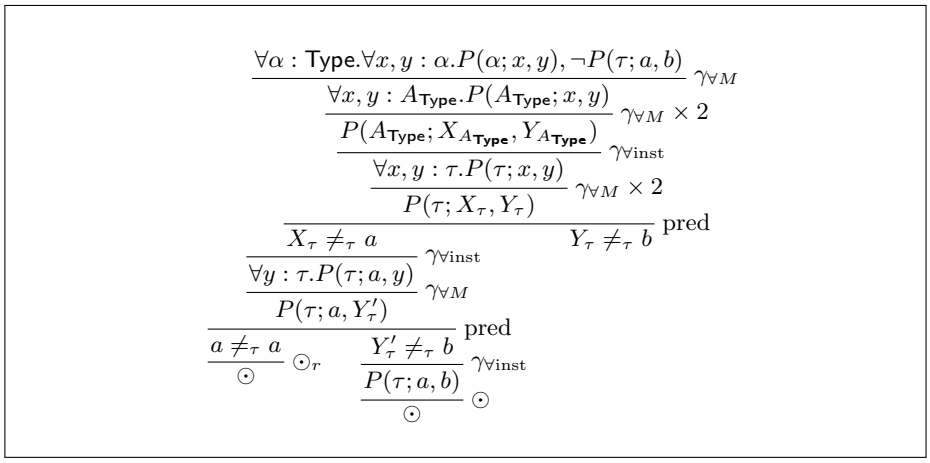


Fig. 5. Proof Search Example with Metavariables

simplex algorithm with no objective function to optimize) to compute solutions for problems over rationals, as well as on the branch-and-bound method to deal with integer problems.

3 A Theory Modulo for the B Set Theory

In this section, we introduce the notion of deduction modulo, as well as the corresponding extension of Zenon, and show how to express the set theory of the B method in this framework.

3.1 From Axioms to Deduction Modulo

Deduction modulo [13] reasons on equivalence classes of formulas under a congruence generated by rewrite rules, which replace axioms. For example, the usual (untyped) inclusion axiom $\forall X, Y.X \subseteq Y \Leftrightarrow \forall x.x \in X \Rightarrow x \in Y$ can be turned into the rewrite rule $X \subseteq Y \longrightarrow \forall x.x \in X \Rightarrow x \in Y$. A tableau proof of $A \subseteq A$ using the regular (untyped) rules of Zenon [7] would then be the following, where t is a shorthand for the term $\epsilon(x).x \in A \Rightarrow x \in A$.

$$\dfrac{\dfrac{\dfrac{\neg(A \subseteq A)}{\neg(t \in A \Rightarrow t \in A)}\,\delta_{\neg\forall},\ A \subseteq A \longrightarrow \forall x.x \in A \Rightarrow x \in A}{t \in A, \neg(t \in A)}\,\alpha_{\neg\Rightarrow}}{\odot}\,\odot$$

We can observe that computation (rewriting) is interleaved with the connective rules and is omitted from the proof tree. We get a simpler and shorter proof than the usual axiom-based proof, which we leave as an exercise (this proof would have two branches, one δ , one β, two α, and two γ rules, which are in fact boilerplate rules to apply the axiom).

3.2 Extension of Zenon to Deduction Modulo

Let us first introduce the notion of rewrite system. It turns out that we do not need the full generality of deduction modulo (in particular, we do not consider the equational axioms in the congruence). We therefore present only the relevant fragment (in the following, $\mathrm{FV}(t)$ stands for the set of free variables of t):

Definition 1 (Rewrite System). *A term rewrite rule is a pair of terms denoted by $l \longrightarrow r$, where $\mathrm{FV}(r) \subseteq \mathrm{FV}(l)$. A proposition rewrite rule is a pair of formulas denoted by $l \longrightarrow r$, where l is an atomic formula and r is an arbitrary formula, and where $\mathrm{FV}(r) \subseteq \mathrm{FV}(l)$.*

A rewrite system is a set $\mathcal{R}$ of proposition rewrite rules along with a set $\mathcal{E}$ of term rewrite rules.

Given a rewrite system $\mathcal{RE}$, the relations $=_{\mathcal{E}}$ and $=_{\mathcal{RE}}$ denote the congruences generated respectively by $\mathcal{E}$ and $\mathcal{R} \cup \mathcal{E}$.

Extending Zenon to deduction modulo then consists in adding to the proof search rules of Figs. 3 and 4 the following conversion rule:

$$\frac{P}{Q}\ \mathrm{conv},\ {\scriptstyle P =_{\mathcal{RE}} Q}$$

This presentation is more modular than the one described in [11], where the congruence $=_{\mathcal{RE}}$ is part of every proof search rule.

The metavariable instantiation mechanism needs adaptation: we look for formulas P and Q s.t. $P =_{\mathcal{RE}} P'$, $Q =_{\mathcal{RE}} \neg Q'$, and there exists a metavariable unifier σ s.t. $\sigma(P') =_{\mathcal{E}} \sigma(Q')$. To have a complete rewriting algorithm, we also extend this mechanism to propositional narrowing: we look for a formula P and a substitution σ s.t. $P =_{\mathcal{RE}} P'$, and there exist $P'_{|\omega}$ and a rule $l \longrightarrow r$ of $\mathcal{RE}$ s.t. $\sigma(P'_{|\omega}) =_{\mathcal{E}} \sigma(l)$ (where $P'_{|\omega}$ is the term or formula at occurrence ω in P').

3.3 Rules for a B Set Theory Modulo

Expressing the B set theory as a theory modulo amounts to building an adequate rewrite system. To do so, we turn whenever possible the axioms and definitions of Chap. 2 of the B-Book [1] into rewrite rules, following the pattern of the previous example regarding the inclusion in set theory. The resulting theory is summarized in Figs. 6 and 7, where we omit the BIG set (an arbitrary infinite set, only used to build natural numbers in the foundational theory), and the sets defined in extension (we only consider the singleton set, which can be used to derive sets defined in extension, and which is also used in other definitions). As can be observed, this theory is typed. The type constructors, i.e. tup for tuples and set for sets, and type schemes of the considered set constructs are provided in Fig. 8 of Appx. A. It should also be noted that we use the infix notations of the B-Book, and that type arguments are subscript annotations of the construct; for example, given two expressions $s, t : \mathsf{set}(\tau)$, where τ is a type, the intersection of s and t is noted $s \cap_\tau t$, and is s.t. $s \cap_\tau t : \mathsf{set}(\tau)$. To improve readability, we

Axioms of Set Theory

$(x, y) \in_{\mathsf{tup}(\alpha_1,\alpha_2)} s \times_{\alpha_1,\alpha_2} t \longrightarrow x \in_{\alpha_1} s \wedge y \in_{\alpha_2} t$
$s \in_{\mathsf{set}(\alpha)} \mathbb{P}_\alpha(t) \longrightarrow \forall x : \alpha . x \in_\alpha s \Rightarrow x \in_\alpha t$
$s =_{\mathsf{set}(\alpha)} t \longrightarrow \forall x : \alpha . x \in_\alpha s \Leftrightarrow x \in_\alpha t$

Set Inclusion

$s \subseteq_\alpha t \longrightarrow s \in_{\mathsf{set}(\alpha)} \mathbb{P}_\alpha(t) \qquad s \subset_\alpha t \longrightarrow s \subseteq_\alpha t \wedge s \neq_{\mathsf{set}(\alpha)} t$

Derived Constructs

$x \in_\alpha s \cup_\alpha t \longrightarrow x \in_\alpha s \vee x \in_\alpha t \qquad x \in_\alpha s \cap_\alpha t \longrightarrow x \in_\alpha s \wedge x \in_\alpha t$
$x \in s -_\alpha t \longrightarrow x \in_\alpha s \wedge x \notin_\alpha t \qquad x \in_\alpha \emptyset_\alpha \longrightarrow \bot$
$x \in_\alpha \{a\}_\alpha \longrightarrow x =_\alpha a \qquad \mathbb{P}_{1\,\alpha}(s) \longrightarrow \mathbb{P}_\alpha(s) -_\alpha \{\emptyset_\alpha\}_{\mathsf{set}(\alpha)}$

Binary Relation Constructs: First Series

$p \in_{\mathsf{set}(\mathsf{tup}(\alpha_1,\alpha_2))} u \leftrightarrow_{\alpha_1,\alpha_2} v \longrightarrow$
$\quad \forall x : \alpha_1 . \forall y : \alpha_2 . (x, y) \in_{\mathsf{tup}(\alpha_1,\alpha_2)} p \Rightarrow x \in_{\alpha_1} u \wedge y \in_{\alpha_2} v$
$(y, x) \in_{\mathsf{tup}(\alpha_2,\alpha_1)} p^{-1}_{\alpha_1,\alpha_2} \longrightarrow (x, y) \in_{\mathsf{tup}(\alpha_1,\alpha_2)} p$
$x \in_{\alpha_1} \mathsf{dom}_{\alpha_1,\alpha_2}(p) \longrightarrow \exists b : \alpha_2 . (x, b) \in_{\mathsf{tup}(\alpha_1,\alpha_2)} p$
$x \in_{\alpha_2} \mathsf{ran}_{\alpha_1,\alpha_2}(p) \longrightarrow \exists a : \alpha_1 . (a, x) \in_{\mathsf{tup}(\alpha_1,\alpha_2)} p$
$(x, y) \in_{\mathsf{tup}(\alpha_1,\alpha_3)} p ;_{\alpha_1,\alpha_2,\alpha_3} q \longrightarrow \exists b : \alpha_2 . (x, b) \in_{\mathsf{tup}(\alpha_1,\alpha_2)} p \wedge (b, y) \in_{\mathsf{tup}(\alpha_2,\alpha_3)} q$
$q \circ_{\alpha_1,\alpha_2,\alpha_3} p \longrightarrow p ;_{\alpha_1,\alpha_2,\alpha_3} q$
$(x, y) \in_{\mathsf{tup}(\alpha,\alpha)} \mathsf{id}_\alpha(u) \longrightarrow x \in_\alpha u \wedge x =_\alpha y$
$(x, y) \in_{\mathsf{tup}(\alpha_1,\alpha_2)} s \lhd_{\alpha_1,\alpha_2} p \longrightarrow (x, y) \in_{\mathsf{tup}(\alpha_1,\alpha_2)} p \wedge x \in_{\alpha_1} s$
$(x, y) \in_{\mathsf{tup}(\alpha_1,\alpha_2)} p \rhd_{\alpha_1,\alpha_2} t \longrightarrow (x, y) \in_{\mathsf{tup}(\alpha_1,\alpha_2)} p \wedge y \in_{\alpha_2} t$
$(x, y) \in_{\mathsf{tup}(\alpha_1,\alpha_2)} s \lhd\!\!\!\cdot\, _{\alpha_1,\alpha_2} p \longrightarrow (x, y) \in_{\mathsf{tup}(\alpha_1,\alpha_2)} p \wedge x \notin_{\alpha_1} s$
$(x, y) \in_{\mathsf{tup}(\alpha_1,\alpha_2)} p \rhd\!\!\!\cdot\, _{\alpha_1,\alpha_2} t \longrightarrow (x, y) \in_{\mathsf{tup}(\alpha_1,\alpha_2)} p \wedge y \notin_{\alpha_2} t$

Fig. 6. Rules of the B Set Theory Modulo (Part 1)

remove the type annotations in tuples when they are used with the membership construct as they are redundant, i.e. given the expressions e_1, e_2, and e_3, s.t. $e_1 : \tau_1$ and $e_2 : \tau_2$, where τ_1 and τ_2 are two types, $(e_1, e_2)_{\tau_1,\tau_2} \in_{\mathsf{tup}(\tau_1,\tau_2)} e_3$ is simply noted $(e_1, e_2) \in_{\mathsf{tup}(\tau_1,\tau_2)} e_3$.

As can be seen, we only consider first order constructs. This means that we do not handle comprehension (Axiom SET 3 of the B-Book) or lambda abstractions. We therefore must eliminate comprehension from the axioms where it appears. For example, the left-hand side formula below is the initial definition of the intersection construct, while the right-hand side formula is its unraveled variant:

$$s \cap_\alpha t \mathrel{\hat{=}} \{x : \alpha \mid x \in_\alpha s \wedge x \in_\alpha t\} \qquad \forall x : \alpha . x \in_\alpha s \cap_\alpha t \Leftrightarrow x \in_\alpha s \wedge x \in_\alpha t$$

This comprehension-free axiom (the right-hand side formula) can then be easily turned into the following rewrite rule:

$$x \in_\alpha s \cap_\alpha t \longrightarrow x \in_\alpha s \wedge x \in_\alpha t$$

Binary Relation Constructs: Second Series

$x \in_{\alpha_2} p[w]_{\alpha_1,\alpha_2} \longrightarrow \exists a : \alpha_1.a \in_{\alpha_1} w \wedge (a,x) \in_{\mathsf{tup}(\alpha_1,\alpha_2)} p$
$(x,y) \in_{\mathsf{tup}(\alpha_1,\alpha_2)} q <+_{\alpha_1,\alpha_2} p \longrightarrow$
$\quad ((x,y) \in_{\mathsf{tup}(\alpha_1,\alpha_2)} q \wedge x \notin_{\alpha_1} \mathsf{dom}_{\alpha_1,\alpha_2}(p)) \vee (x,y) \in_{\mathsf{tup}(\alpha_1,\alpha_2)} p$
$(x,(y,z)) \in_{\mathsf{tup}(\alpha_1,\mathsf{tup}(\alpha_2,\alpha_3))} f \otimes_{\alpha_1,\alpha_2,\alpha_3} g \longrightarrow (x,y) \in_{\mathsf{tup}(\alpha_1,\alpha_2)} f \wedge (x,z) \in_{\mathsf{tup}(\alpha_1,\alpha_3)} g$
$((x,y),z) \in_{\mathsf{tup}(\mathsf{tup}(\alpha_1,\alpha_2),\alpha_1)} \mathsf{prj}_{1\ \alpha_1,\alpha_2}(s,t) \longrightarrow$
$\quad ((x,y),z) \in_{\mathsf{tup}(\mathsf{tup}(\alpha_1,\alpha_2),\alpha_1)} (s \times_{\alpha_1,\alpha_2} t) \times_{\mathsf{tup}(\alpha_1,\alpha_2),\alpha_1} s \wedge x =_{\alpha_1} z$
$((x,y),z) \in_{\mathsf{tup}(\mathsf{tup}(\alpha_1,\alpha_2),\alpha_2)} \mathsf{prj}_{2\ \alpha_1,\alpha_2}(s,t) \longrightarrow$
$\quad ((x,y),z) \in_{\mathsf{tup}(\mathsf{tup}(\alpha_1,\alpha_2),\alpha_1)} (s \times_{\alpha_1,\alpha_2} t) \times_{\mathsf{tup}(\alpha_1,\alpha_2),\alpha_1} t \wedge y =_{\alpha_1} z$
$((x,y),(z,w)) \in_{\mathsf{tup}(\mathsf{tup}(\alpha_1,\alpha_3),\mathsf{tup}(\alpha_2,\alpha_4))} h||_{\alpha_1,\alpha_2,\alpha_3,\alpha_4} k \longrightarrow$
$\quad (x,z) \in_{\mathsf{tup}(\alpha_1,\alpha_2)} h \wedge (y,w) \in_{\mathsf{tup}(\alpha_3,\alpha_4)} k$

Function Constructs: First Series

$f \in_{\mathsf{set}(\mathsf{tup}(\alpha_1,\alpha_2))} s \mathrel{+\!\!\!\rightarrow}_{\alpha_1,\alpha_2} t \longrightarrow f \in_{\mathsf{set}(\mathsf{tup}(\alpha_1,\alpha_2))} s \leftrightarrow_{\alpha_1,\alpha_2} t \wedge$
$\quad \forall x : \alpha_1.\forall y,z : \alpha_2.(x,y) \in_{\mathsf{tup}(\alpha_1,\alpha_2)} f \wedge (x,z) \in_{\mathsf{tup}(\alpha_1,\alpha_2)} f \Rightarrow y =_{\alpha_2} z$
$f \in_{\mathsf{set}(\mathsf{tup}(\alpha_1,\alpha_2))} s \rightarrow_{\alpha_1,\alpha_2} t \longrightarrow$
$\quad f \in_{\mathsf{set}(\mathsf{tup}(\alpha_1,\alpha_2))} s \mathrel{+\!\!\!\rightarrow}_{\alpha_1,\alpha_2} t \wedge \mathsf{dom}_{\alpha_1,\alpha_2}(f) =_{\mathsf{set}(\alpha_1)} s$
$f \in_{\mathsf{set}(\mathsf{tup}(\alpha_1,\alpha_2))} s \rightarrowtail_{\alpha_1,\alpha_2} t \longrightarrow$
$\quad f \in_{\mathsf{set}(\mathsf{tup}(\alpha_1,\alpha_2))} s \mathrel{+\!\!\!\rightarrow}_{\alpha_1,\alpha_2} t \wedge f^{-1}_{\alpha_1,\alpha_2} \in_{\mathsf{set}(\mathsf{tup}(\alpha_2,\alpha_1))} t \mathrel{+\!\!\!\rightarrow}_{\alpha_2,\alpha_1} s$
$f \in_{\mathsf{set}(\mathsf{tup}(\alpha_1,\alpha_2))} s \rightarrowtail_{\alpha_1,\alpha_2} t \longrightarrow$
$\quad f \in_{\mathsf{set}(\mathsf{tup}(\alpha_1,\alpha_2))} s \rightarrowtail_{\alpha_1,\alpha_2} t \wedge f \in_{\mathsf{set}(\mathsf{tup}(\alpha_1,\alpha_2))} s \rightarrow_{\alpha_1,\alpha_2} t$
$f \in_{\mathsf{set}(\mathsf{tup}(\alpha_1,\alpha_2))} s \twoheadrightarrow_{\alpha_1,\alpha_2} t \longrightarrow$
$\quad f \in_{\mathsf{set}(\mathsf{tup}(\alpha_1,\alpha_2))} s \mathrel{+\!\!\!\rightarrow}_{\alpha_1,\alpha_2} t \wedge \mathsf{ran}_{\alpha_1,\alpha_2}(f) =_{\mathsf{set}(\alpha_2)} t$
$f \in_{\mathsf{set}(\mathsf{tup}(\alpha_1,\alpha_2))} s \rightarrow\!\!\!\!\rightarrow_{\alpha_1,\alpha_2} t \longrightarrow$
$\quad f \in_{\mathsf{set}(\mathsf{tup}(\alpha_1,\alpha_2))} s \twoheadrightarrow_{\alpha_1,\alpha_2} t \wedge f \in_{\mathsf{set}(\mathsf{tup}(\alpha_1,\alpha_2))} s \rightarrow_{\alpha_1,\alpha_2} t$
$f \in_{\mathsf{set}(\mathsf{tup}(\alpha_1,\alpha_2))} s \rightarrowtail\!\!\!\!\rightarrow_{\alpha_1,\alpha_2} t \longrightarrow$
$\quad f \in_{\mathsf{set}(\mathsf{tup}(\alpha_1,\alpha_2))} s \rightarrowtail_{\alpha_1,\alpha_2} t \wedge f \in_{\mathsf{set}(\mathsf{tup}(\alpha_1,\alpha_2))} s \twoheadrightarrow_{\alpha_1,\alpha_2} t$
$f \in_{\mathsf{set}(\mathsf{tup}(\alpha_1,\alpha_2))} s \rightarrowtail\!\!\!\!\!\rightarrow_{\alpha_1,\alpha_2} t \longrightarrow$
$\quad f \in_{\mathsf{set}(\mathsf{tup}(\alpha_1,\alpha_2))} s \rightarrowtail_{\alpha_1,\alpha_2} t \wedge x \in_{\mathsf{set}(\mathsf{tup}(\alpha_1,\alpha_2))} s \rightarrow\!\!\!\!\rightarrow_{\alpha_1,\alpha_2} t$

Fig. 7. Rules of the B Set Theory Modulo (Part 2)

We also modify the definition of function evaluation, defined in the B-Book in terms of the choice operator, itself defined via the axiom of choice (Axiom SET 5). Such a definition is not well-suited for automated deduction, and we replace it with two derived properties (Properties 2.5.2 and 2.5.4 in the B-Book):

$\forall \alpha_1,\alpha_2 : \mathsf{Type}.\forall s : \mathsf{set}(\alpha_1).\forall t : \mathsf{set}(\alpha_2).\forall f : \mathsf{set}(\mathsf{tup}(\alpha_1,\alpha_2)).\forall x : \alpha_1.$
$\quad f \in_{\mathsf{set}(\mathsf{tup}(\alpha_1,\alpha_2))} s \mathrel{+\!\!\!\rightarrow}_{\alpha_1,\alpha_2} t \wedge x \in_{\alpha_1} \mathsf{dom}_{\alpha_1,\alpha_2}(f) \Rightarrow$
$\quad (x, f(x)_{\alpha_1,\alpha_2}) \in_{\mathsf{tup}(\alpha_1,\alpha_2)} f$

$\forall \alpha_1,\alpha_2 : \mathsf{Type}.\forall s : \mathsf{set}(\alpha_1).\forall t : \mathsf{set}(\alpha_2).\forall f : \mathsf{set}(\mathsf{tup}(\alpha_1,\alpha_2)).\forall x : \alpha_1.\forall y : \alpha_2.$
$\quad f \in_{\mathsf{set}(\mathsf{tup}(\alpha_1,\alpha_2))} s \mathrel{+\!\!\!\rightarrow}_{\alpha_1,\alpha_2} t \Rightarrow (x,y) \in_{\mathsf{tup}(\alpha_1,\alpha_2)} f \Leftrightarrow$
$\quad x \in_{\alpha_1} \mathsf{dom}_{\alpha_1}(f) \wedge y =_{\alpha_2} f(x)_{\alpha_1,\alpha_2}$

Turning these axioms into rewrite rules has been shown possible only recently [8]. It requires conditional rewriting (the guard being, in the first case, $x \in_{\alpha_1} \mathsf{dom}_{\alpha_1,\alpha_2}(f)$), which is not yet implemented. For the time being, those axioms are therefore left as regular axioms in our theory.

To sum up, our formulation of the B set theory sticks closely to the B-Book, for the BIG- and extension-free fragment. The slight differences deserve to be justified, at least informally. What we need is conservativity results, which boil down to cross-derivability of the initial axioms and definitions in our system (completeness) and the formulas associated to our rewrite rules in the initial system (correctness). The minor variations over function evaluation are harmless, the main point being comprehension. Dropping this axiom is hazardous only for completeness, the correctness of the "inlined" rewrite rules that we propose follows from immediate applications of the comprehension axiom itself. Without further considerations, completeness fails: sometimes, it might be necessary to introduce a set defined by comprehension, for example in Cantor's proof that "a set is not in bijection with its power set". However, this kind of smart trick is seldom met in practice, and we leave this question for future work.

4 Experimental Results

To assess the extensions of Zenon and the design of the B set theory modulo (see Secs. 2 and 3), we use a benchmark of POs provided by the industrial partners of the BWare project [12, 15]. To run these tests, we rely on the BWare verification platform, which we outline briefly here. The POs are initially produced by Atelier B. They are then translated into Why3 files [3], using a Why3 encoding of the B set theory [14]. Next, the Why3 platform generates the POs, in the form of valid statements, and feeds the automated deduction tools. Why3's B set theory is interpreted as rewrite rules (according to Sec. 3) for tools compliant with deduction modulo, otherwise as axioms. As this theory appeals to polymorphism, the output format may be either the new TFF1 format [2] of the TPTP community for the first-order polymorphic ATPs, or the regular FOF format with a first-order encoding of the polymorphic layer [4] for the other first- order ATPs, or the SMT-LIB format with the same encoding for SMT solvers, except for Alt-Ergo, which features a native format to deal with polymorphism.

The benchmark consists in 12,876 POs[4], and the experiment was run on an Intel Xeon E5-2660 v2 2.20GHz computer, with a timeout of 120s and a memory limit of 1GiB. The results are summarized in Tab. 1. In these results, the first table focuses on the results for five different versions of Zenon, mainly based on Zenon 0.8.0, compared to the main prover (mp) of Atelier B 4.0. The second table compares all the tools of BWare, i.e. mp, Zenon, Alt-Ergo 0.99.1, and iProver Modulo v0.7+0.2 (an extension of iProver v0.7 to deduction modulo), with a representative panel of first order ATPs, such as Vampire 2.6 and E 1.8, and SMT solvers, like CVC4 1.4 and Z3 4.3.2. Among these tools, only Zenon

[4] This benchmark is publicly available at: `http://bware.lri.fr/`.

	All Tools (12,738/98.9%)					
#POs: 12,876	mp	Zenon	Zenon (T)	Zenon (T+A)	Zenon (T+M)	Zenon (T+M+A)
Proofs	10,995	337	6,251	7,406	10,340	12,281
Rate	85.4%	2.6%	48.5%	57.5%	80.3%	95.4%
Time (s)	-	2,316	14,452	18,514	31,665	31,689
Unique	329	0	0	0	34	946

	All Tools (12,797/99.4%)							
	BWare Tools (12,772/99.2%)				Other Tools			
#POs: 12,876	mp	Zenon (T+M+A)	iProver Modulo	Alt-Ergo	Vampire	E	CVC4	Z3
Proofs	10,995	12,281	3,695	12,620	10,154	7,877	12,173	10,880
Rate	85.4%	95.4%	28.7%	98.0%	78.9%	61.2%	94.5%	84.5%
Time (s)	-	31,689	20,156	7,129	118,541	40,215	7,897	3,404
Uniq. (1)	109	4	0	65				
Uniq. (2)	84	0	0	13	0	0	1	12

T ≡ with types M ≡ with deduction modulo A ≡ with arithmetic

Table 1. Experimental Results over the BWare Benchmark

with deduction modulo and iProver Modulo implement deduction modulo, while only Zenon with types and Alt-Ergo have polymorphic types.

For both tables, we provide the number of proved POs, the corresponding rate, and the cumulative time for the successfully proved POs (not measured for mp, since it is not possible to split the timeout by PO). The "Unique" line refers to the number of POs that are only proved by a given prover; in the second table, "Uniq. (1)" ranges over the BWare tools, while "Uniq. (2)" considers all the tools. The coverage is given on top of tables; in the second table, we also distinguish the coverage for all the tools and among the BWare tools. In addition, Fig. 9 of Appx. B presents the cumulative times according to the numbers of proved POs, and shows the trends in terms of proof power w.r.t. the time resource.

In the first table, in addition to the regular version of Zenon, we present the extensions with (polymorphic) types, with types and arithmetic, with types and deduction modulo, and with types, deduction modulo, and arithmetic, which is currently considered as the regular version of Zenon Modulo. As can be observed, the more extensions we plug, the more POs we prove. The most significant gain is provided by the type extension, for which we get an increase of about 1755% compared to plain Zenon. Plugging deduction modulo gives an additional increase of 65%. Finally, connecting arithmetic on top allows us to prove 20% POs more, and to improve by 10 percentage points on the results of mp.

In the second table, we observe that Zenon with types and deduction modulo (but without arithmetic) obtains better results than the first order ATPs Vampire and E. Similarly, Zenon including all the extensions proves more POs than the SMT solvers CVC4 and Z3, except Alt-Ergo. The low results of iProver Modulo can be explained by the encoding of polymorphism, which hampers the analysis of the theory and the generation of a rewrite system similar to the one of Sec. 3.

As described in [11], Zenon Modulo enjoys a backend that outputs certificates for Dedukti [5], a universal proof checker for the $\lambda\Pi$-calculus modulo. Since it also relies on deduction modulo, Dedukti is able to natively deal with rewriting and is well-suited to verify the proofs of Zenon Modulo. In particular, we don't need to record the rewriting steps in the proofs (these steps are implicitly done by Dedukti), which are therefore quite compact. The logic of Dedukti is constructive, and calls for a translation initially based on an optimized double-negation translation (see [11]). We have replaced it by a more syntactical translation using excluded middle explicitly (due to lack of space, we cannot detail this translation in this paper), which is more efficient in practice. Currently, this backend deals with all the proofs not involving arithmetic, and all the 10,340 relevant POs of Tab. 1 have been translated and approved by Dedukti.

5 Conclusion

In this paper, we have observed the benefits of deduction modulo for automated deduction, implemented in the Zenon Modulo tool, and applied to the B set theory. We have assessed this claim on the thousands of POs provided by the BWare project. Our tool obtains better results than the state-of-the-art first order ATPs (without arithmetic), and competes with the best SMT solvers (with arithmetic). Another contribution is to show how providing an ATP with an elaborate type system offers effective flexibility in the expressiveness of theories, and significantly helps the proof search itself. On our benchmark, this is notable when comparing Zenon with its version with polymorphic types.

As future work, we aim to integrate conditional rewrite rules to Zenon Modulo along the lines of [8], in order to define function evaluation by rewriting. We also plan to study theoretical properties, such as cut-free completeness. As there is no general way to ensure this in deduction modulo [9], we will probably use techniques specific to our case. We also need to enhance the Dedukti backend of Zenon Modulo with arithmetic, and make Dedukti understand these computations. We foresee an extension of the conversion rule of Dedukti, shifting from rewriting to reasoning modulo a decision procedure. This would allow us to produce more compact proof certificates. Finally, within the BWare project, we will increase the variety of available domains and applications of the benchmark, through the integration of a new large project. The overall benchmark will consist of more than 80,000 proof obligations, which will be one of the largest academic benchmarks in the domain of program verification. Still in the objectives of BWare, we intend to make this work scale up to industry. A first step has been done in this direction in the latest version of Atelier B, which now proposes

a Why3 output. To obtain a similar integration of Zenon Modulo, we need to certify it, which should be eased by its ability to produce checkable proofs by means of its Dedukti backend.

References

1. J.-R. Abrial. *The B-Book, Assigning Programs to Meanings*. Cambridge University Press, Cambridge (UK), 1996. ISBN 0521496195.
2. J. C. Blanchette and A. Paskevich. TFF1: The TPTP Typed First-Order Form with Rank-1 Polymorphism. In *Conference on Automated Deduction (CADE)*, volume 7898 of *LNCS*, pages 414–420, Lake Placid (NY, USA), June 2013. Springer.
3. F. Bobot, J.-C. Filliâtre, C. Marché, and A. Paskevich. Why3: Shepherd Your Herd of Provers. In *International Workshop on Intermediate Verification Languages (Boogie)*, 2011.
4. F. Bobot and A. Paskevich. Expressing Polymorphic Types in a Many-Sorted Language. In *Frontiers of Combining Systems (FroCoS)*, volume 6989 of *LNCS*, pages 87–102, Saarbrücken (Germany), Oct. 2011. Springer.
5. M. Boespflug, Q. Carbonneaux, and O. Hermant. The $\lambda\Pi$-Calculus Modulo as a Universal Proof Language. In *Proof Exchange for Theorem Proving (PxTP)*, pages 28–43, Manchester (UK), June 2012.
6. R. Bonichon. TaMeD: A Tableau Method for Deduction Modulo. In *International Joint Conference on Automated Reasoning (IJCAR)*, volume 3097 of *LNCS*, pages 445–459, Cork (Ireland), July 2004. Springer.
7. R. Bonichon, D. Delahaye, and D. Doligez. Zenon: An Extensible Automated Theorem Prover Producing Checkable Proofs. In *Logic for Programming Artificial Intelligence and Reasoning (LPAR)*, volume 4790 of *LNCS/LNAI*, pages 151–165, Yerevan (Armenia), Oct. 2007. Springer.
8. G. Burel. Cut Admissibility by Saturation. In *Joint International Conference on Rewriting Techniques and Applications and International Conference on Typed Lambda Calculi and Applications (RTA-TLCA)*, volume 8560 of *LNCS*, pages 124–138, Vienna (Austria), July 2014. Springer.
9. G. Burel and C. Kirchner. Regaining Cut Admissibility in Deduction Modulo using Abstract Completion. *Information and Computation*, 208(2):140–164, Feb. 2010.
10. V. Chvátal. *Linear Programming*. Series of Books in the Mathematical Sciences. W. H. Freeman and Company, New York (USA), 1983. ISBN 0716715872.
11. D. Delahaye, D. Doligez, F. Gilbert, P. Halmagrand, and O. Hermant. Zenon Modulo: When Achilles Outruns the Tortoise using Deduction Modulo. In *Logic for Programming Artificial Intelligence and Reasoning (LPAR)*, volume 8312 of *LNCS/ARCoSS*, pages 274–290, Stellenbosch (South Africa), Dec. 2013. Springer.
12. D. Delahaye, C. Dubois, C. Marché, and D. Mentré. The BWare Project: Building a Proof Platform for the Automated Verification of B Proof Obligations. In *Abstract State Machines, Alloy, B, VDM, and Z (ABZ)*, Lecture Notes in Computer Science (LNCS), pages 126–127, Toulouse (France), June 2014. Springer.
13. G. Dowek, T. Hardin, and C. Kirchner. Theorem Proving Modulo. *Journal of Automated Reasoning (JAR)*, 31(1):33–72, Sept. 2003.
14. D. Mentré, C. Marché, J.-C. Filliâtre, and M. Asuka. Discharging Proof Obligations from Atelier B using Multiple Automated Provers. In *Abstract State Machines, Alloy, B, VDM, and Z (ABZ)*, volume 7316 of *LNCS*. Springer, 2012.
15. The BWare Project, 2012. `http://bware.lri.fr/`.

A Typing of the Theory Modulo for the B Set Theory

Type Constructors

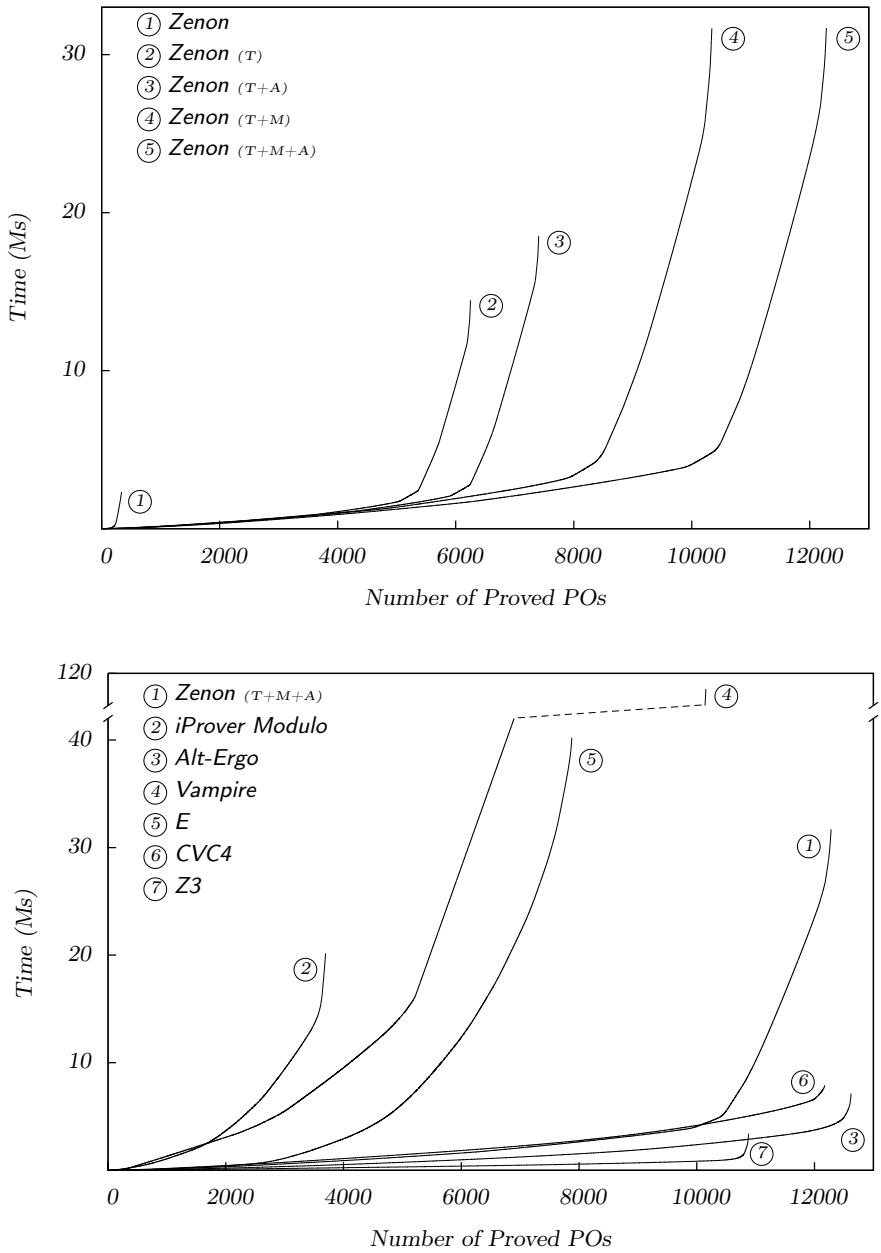
$\mathsf{tup} : \Pi\alpha_1, \alpha_2 : \mathsf{Type}.\mathsf{Type}$ $\quad$ $\mathsf{set} : \Pi\alpha : \mathsf{Type}.\mathsf{Type}$

Type Schemes of the Set Constructs

$$
\begin{array}{ll}
\text{- } \in \text{ -} & : \Pi\alpha : \mathsf{Type}.\alpha \to \mathsf{set}(\alpha) \to o \\
(\text{-},\text{-}) & : \Pi\alpha_1, \alpha_2 : \mathsf{Type}.\alpha_1 \to \alpha_2 \to \mathsf{tup}(\alpha_1, \alpha_2) \\
\text{- } \times \text{ -} & : \Pi\alpha_1, \alpha_2 : \mathsf{Type}.\mathsf{set}(\alpha_1) \to \mathsf{set}(\alpha_2) \to \mathsf{set}(\mathsf{tup}(\alpha_1, \alpha_2)) \\
\mathbb{P}(\text{-}) & : \Pi\alpha : \mathsf{Type}.\mathsf{set}(\alpha) \to \mathsf{set}(\mathsf{set}(\alpha)) \\
\text{- } = \text{ -} & : \Pi\alpha : \mathsf{Type}.\alpha \to \alpha \to o \\
\mathsf{BIG} & : \Pi\alpha : \mathsf{Type}.\mathsf{set}(\alpha) \\
\text{- } \subseteq \text{ -},\ \text{- } \subsetneq \text{ -} : & \\
\qquad & \Pi\alpha : \mathsf{Type}.\mathsf{set}(\alpha) \to \mathsf{set}(\alpha) \to o \\
\text{- } \cup \text{ -},\ \text{- } \cap \text{ -},\ \text{- } - \text{ -} : & \\
\qquad & \Pi\alpha : \mathsf{Type}.\mathsf{set}(\alpha) \to \mathsf{set}(\alpha) \to \mathsf{set}(\alpha) \\
\{\text{-}\} & : \Pi\alpha : \mathsf{Type}.\alpha \to \mathsf{set}(\alpha) \\
\emptyset & : \Pi\alpha : \mathsf{Type}.\mathsf{set}(\alpha) \\
\mathbb{P}_1(\text{-}) & : \Pi\alpha : \mathsf{Type}.\mathsf{set}(\alpha) \to \mathsf{set}(\mathsf{set}(\alpha)) \\
\text{- } \leftrightarrow \text{ -} & : \Pi\alpha_1, \alpha_2 : \mathsf{Type}.\mathsf{set}(\alpha_1) \to \mathsf{set}(\alpha_2) \to \mathsf{set}(\mathsf{set}(\mathsf{tup}(\alpha_1, \alpha_2))) \\
\text{-}^{-1} & : \Pi\alpha_1, \alpha_2 : \mathsf{Type}.\mathsf{set}(\mathsf{tup}(\alpha_1, \alpha_2)) \to \mathsf{set}(\mathsf{tup}(\alpha_2, \alpha_1)) \\
\mathsf{dom}(\text{-}) & : \Pi\alpha_1, \alpha_2 : \mathsf{Type}.\mathsf{set}(\mathsf{tup}(\alpha_1, \alpha_2)) \to \mathsf{set}(\alpha_1) \\
\mathsf{ran}(\text{-}) & : \Pi\alpha_1, \alpha_2 : \mathsf{Type}.\mathsf{set}(\mathsf{tup}(\alpha_1, \alpha_2)) \to \mathsf{set}(\alpha_2) \\
\text{-;-} & : \Pi\alpha_1, \alpha_2, \alpha_3 : \mathsf{Type}.\mathsf{set}(\mathsf{tup}(\alpha_1, \alpha_2)) \to \mathsf{set}(\mathsf{tup}(\alpha_2, \alpha_3)) \to \mathsf{set}(\mathsf{tup}(\alpha_1, \alpha_3)) \\
\text{- } \circ \text{ -} & : \Pi\alpha_1, \alpha_2, \alpha_3 : \mathsf{Type}.\mathsf{set}(\mathsf{tup}(\alpha_2, \alpha_3)) \to \mathsf{set}(\mathsf{tup}(\alpha_1, \alpha_2)) \to \mathsf{set}(\mathsf{tup}(\alpha_1, \alpha_3)) \\
\mathsf{id}(\text{-}) & : \Pi\alpha : \mathsf{Type}.\mathsf{set}(\alpha) \to \mathsf{set}(\mathsf{tup}(\alpha, \alpha)) \\
\text{- } \lhd \text{ -} & : \Pi\alpha_1, \alpha_2 : \mathsf{Type}.\mathsf{set}(\alpha_1) \to \mathsf{set}(\mathsf{tup}(\alpha_1, \alpha_2)) \to \mathsf{set}(\mathsf{tup}(\alpha_1, \alpha_2)) \\
\text{- } \rhd \text{ -} & : \Pi\alpha_1, \alpha_2 : \mathsf{Type}.\mathsf{set}(\mathsf{tup}(\alpha_1, \alpha_2)) \to \mathsf{set}(\alpha_2) \to \mathsf{set}(\mathsf{tup}(\alpha_1, \alpha_2)) \\
\text{- } \trianglelefteq \text{ -} & : \Pi\alpha_1, \alpha_2 : \mathsf{Type}.\mathsf{set}(\alpha_1) \to \mathsf{set}(\mathsf{tup}(\alpha_1, \alpha_2)) \to \mathsf{set}(\mathsf{tup}(\alpha_1, \alpha_2)) \\
\text{- } \trianglerighteq \text{ -} & : \Pi\alpha_1, \alpha_2 : \mathsf{Type}.\mathsf{set}(\mathsf{tup}(\alpha_1, \alpha_2)) \to \mathsf{set}(\alpha_2) \to \mathsf{set}(\mathsf{tup}(\alpha_1, \alpha_2)) \\
\text{-[-]} & : \Pi\alpha_1, \alpha_2 : \mathsf{Type}.\mathsf{set}(\mathsf{tup}(\alpha_1, \alpha_2)) \to \mathsf{set}(\alpha_1) \to \mathsf{set}(\alpha_2) \\
\text{-}\lhd\text{+-} & : \Pi\alpha_1, \alpha_2 : \mathsf{Type}.\mathsf{set}(\mathsf{tup}(\alpha_1, \alpha_2)) \to \mathsf{set}(\mathsf{tup}(\alpha_1, \alpha_2)) \to \mathsf{set}(\mathsf{tup}(\alpha_1, \alpha_2)) \\
\text{- } \otimes \text{ -} & : \Pi\alpha_1, \alpha_2, \alpha_3 : \mathsf{Type}.\mathsf{set}(\mathsf{tup}(\alpha_1, \alpha_2)) \to \mathsf{set}(\mathsf{tup}(\alpha_1, \alpha_3)) \to \\
\qquad & \quad \mathsf{set}(\mathsf{tup}(\alpha_1, \mathsf{tup}(\alpha_2, \alpha_3))) \\
\mathsf{prj}_1(\text{-}) & : \Pi\alpha_1, \alpha_2 : \mathsf{Type}.\mathsf{tup}(\mathsf{set}(\alpha_1), \mathsf{set}(\alpha_2)) \to \mathsf{set}(\mathsf{tup}(\mathsf{tup}(\alpha_1, \alpha_2), \alpha_1)) \\
\mathsf{prj}_2(\text{-}) & : \Pi\alpha_1, \alpha_2 : \mathsf{Type}.\mathsf{tup}(\mathsf{set}(\alpha_1), \mathsf{set}(\alpha_2)) \to \mathsf{set}(\mathsf{tup}(\mathsf{tup}(\alpha_1, \alpha_2), \alpha_2)) \\
\text{-}||\text{-} & : \Pi\alpha_1, \alpha_2, \alpha_3, \alpha_4 : \mathsf{Type}.\mathsf{set}(\mathsf{tup}(\alpha_1, \alpha_2)) \to \mathsf{set}(\mathsf{tup}(\alpha_3, \alpha_4)) \to \\
\qquad & \quad \mathsf{set}(\mathsf{tup}(\mathsf{tup}(\alpha_1, \alpha_3), \mathsf{tup}(\alpha_2, \alpha_4))) \\
\text{- } \nrightarrow \text{ -},\ \text{- } \to \text{ -},\ \text{- } \rightarrowtail \text{ -},\ \text{- } \rightarrowtail \text{ -},\ \text{- } \twoheadrightarrow \text{ -},\ \text{- } \twoheadrightarrow \text{ -},\ \text{- } \twoheadrightarrow\!\!\!\!\!\rightarrowtail \text{ -},\ \text{- } \twoheadrightarrow\!\!\!\!\!\rightarrowtail \text{ -} : & \\
\qquad & \Pi\alpha_1, \alpha_2 : \mathsf{Type}.\mathsf{set}(\alpha_1) \to \mathsf{set}(\alpha_2) \to \mathsf{set}(\mathsf{set}(\mathsf{tup}(\alpha_1, \alpha_2))) \\
\text{-(-)} & : \Pi\alpha_1, \alpha_2 : \mathsf{Type}.\mathsf{set}(\mathsf{tup}(\alpha_1, \alpha_2)) \to \alpha_1 \to \alpha_2
\end{array}
$$

Fig. 8. Type Constructors and Type Schemes of the Set Constructs

B Cumulative Time Graphs of the Experimental Results



T ≡ with types M ≡ with deduction modulo A ≡ with arithmetic

Fig. 9. Cumulative Times According to the Numbers of Proved POs