Automated Deduction in the B Set Theory using Typed Proof Search and 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, PSL Research University, Fontainebleau, France,
Olivier.Hermant@mines-paristech.fr

Abstract

We introduce an encoding of the set theory of the B method using polymorphic types and deduction modulo, which is used for the automated verification of proof obligations in the framework of the BWare project. Deduction modulo is an extension of predicate calculus with rewriting both on terms and propositions. It is well suited for proof search in theories because it turns many axioms into rewrite rules. We also present the associated automated theorem prover Zenon Modulo, an extension of Zenon to polymorphic types and deduction modulo, along with its backend to the Dedukti universal proof checker, which also relies on types and deduction modulo, and which allows us to verify the proofs produced by Zenon Modulo. Finally, we assess our approach over the proof obligation benchmark provided by the BWare project.

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 tool sets, such as Atelier B [15], 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. He/she must then discharge POs 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 in particular in industry, the BWare project [17, 21] 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.

*This work is supported by the BWare project [17, 21] (ANR-12-INSE-0010) funded by the INS programme of the French National Research Agency (ANR).

Leaving the axioms and definitions of set theory at the same level as the hypotheses is not a reasonable option: first, it induces a combinatorial explosion in the search space and second, axioms do not bear any specific meaning that an ATP can take advantage of. To avoid these drawbacks, we replace axioms by rewrite rules, along the lines of deduction modulo [18], a framework combining first order proof systems with a congruence generated by rewrite rules on terms and propositions. This last distinctive feature allows us to go beyond pure first order reasoning. However, we must take care of preserving desirable properties for proof search, such as consistency, cut elimination, or completeness.

In this paper, we define an encoding of the set theory of the B method 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, 18], we also provide this encoding with an extension of the Zenon tableau-based ATP [7] to deduction modulo, getting a tool called Zenon Modulo [16]. In addition, this tool features a backend [16, 13] to Dedukti [5], a universal proof checker that also relies on deduction modulo, in order to verify the produced proofs [13].

To cope properly with the POs provided by the industrial partners of the BWare project, we extend, in this paper, Zenon to typed proof search. We therefore provide Zenon with a polymorphic type system, which offers more flexibility, and in particular which allows us to deal with theories that rely on elaborate type systems, like the B set theory. If integrating types to tableaux has already been proposed in [22, 19, 9], the novelty consists in considering polymorphic types à la ML, through a type system in the spirit of [3]. In addition and compared to those previous approaches, we also provide one of the very few concrete implementations of ATPs with polymorphic types.

The paper is organized as follows: in Sec. 2, we describe the extension of Zenon to typed proof search using a polymorphic type system; 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 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 allows the user to feed the tool with high-level theories as expressed in a flexible way thanks to polymorphism.

2.1 Polymorphic Type System

Before introducing the proof search 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 [3]. 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 symbols, 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 proof search. This explains why there are metavariables (capitalized here, often named free variables in the tableau-related literature), which are used to find instantiations by unification, as well as Hilbert's 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. If the use of ϵ-terms at the level of types may appear as non-standard, it allows us to make the handling of quantification over terms and types uniform.

Term-level Types

$$
\begin{array}{rcll}
\tau & ::= & \alpha & \textit{(type variable)} \\
 & | & A_{\mathsf{Type}} & \textit{(type metavariable)} \\
 & | & \epsilon(\alpha : \mathsf{Type}).e(\alpha) & \textit{(type } \epsilon\textit{-term)} \\
 & | & f(\tau_1, \ldots, \tau_m) & \textit{(type constructor application)}
\end{array}
$$

Type Schemes

$$
\begin{array}{rcll}
\sigma & ::= & \Pi\alpha_1 : \mathsf{Type} \ldots \alpha_m : \mathsf{Type}.\tau_1 \to \ldots \to \tau_n \to \tau & \textit{(polymorphic terms)} \\
 & | & \Pi\alpha_1 : \mathsf{Type} \ldots \alpha_m : \mathsf{Type}.\tau_1 \to \ldots \to \tau_n \to o & \textit{(polymorphic formulas)} \\
 & | & \Pi\alpha_1 : \mathsf{Type} \ldots \alpha_m : \mathsf{Type}.\mathsf{Type} & \textit{(polymorphic types)}
\end{array}
$$

Expressions

$$
\begin{array}{rcll}
e & ::= & x & \textit{(variable)} \\
 & | & X_\tau & \textit{(metavariable)} \\
 & | & \epsilon(x : \tau).e(x) & \textit{(}\epsilon\textit{-term)} \\
 & | & e_1 =_\tau e_2 & \textit{(equality)} \\
 & | & f(\tau_1, \ldots, \tau_m; e_1, \ldots, e_n) & \textit{(application)} \\
 & | & \top \mid \bot & \textit{(true and false)} \\
 & | & \neg e \mid e_1 \wedge e_2 \mid e_1 \vee e_2 \mid e_1 \Rightarrow e_2 \mid \ e_1 \Leftrightarrow e_2 & \textit{(logical connectives)} \\
 & | & \forall x : \tau.e(x) \mid \exists x : \tau.e(x) & \textit{(quantifiers over terms)} \\
 & | & \forall \alpha : \mathsf{Type}.e(\alpha) \mid \exists \alpha : \mathsf{Type}.e(\alpha) & \textit{(quantifiers over types)}
\end{array}
$$

Figure 1: Types, Type Schemes, and Expressions

In presence of empty types, these type ϵ-terms may introduce inconsistencies, but there is no empty type in our context (each type is inhabited by an infinity of variables of this type). 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 $\Gamma_G; \Gamma_L$.

Fig. 2 presents the rules for a single (as customary for pure type systems) 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, $\kappa : \mathsf{Kind}$ is either a term-level type, or Type. Similarly, κ_o is either κ, or o (the type of formulas). It should be noted that o is not a type, just as Kind, but a pseudo-type, which does not appear in the syntax of types, and therefore does not have any corresponding typing rule. The usual freshness condition 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 rules summarized in Figs. 3 and 4 are an adaptation of the rules of Zenon [7] to typed formulas. For the sake of simplicity, we have omitted the unfolding and extension rules. The

Well-Formedness Rules

$$\frac{}{(\emptyset;\emptyset)\ \text{wf}}\ \text{WF}_0 \qquad \frac{\Gamma_G;\Gamma_L \vdash \kappa : \mathsf{Kind}}{(\Gamma_G;\Gamma_L, x:\kappa)\ \text{wf}}\ \text{WF}_1 \qquad \frac{\Gamma_G;\emptyset \vdash f : \sigma}{(\Gamma_G, f:\sigma;\emptyset)\ \text{wf}}\ \text{WF}_2$$

$$\frac{\begin{array}{c}\Gamma_G;\alpha_1 : \mathsf{Type},\ldots,\alpha_m : \mathsf{Type} \vdash \tau_i : \mathsf{Type}, i = 1\ldots n \\ \Gamma_G;\alpha_1 : \mathsf{Type},\ldots,\alpha_m : \mathsf{Type} \vdash \kappa_o : \mathsf{Kind}\ (\text{or } \kappa_o = o)\end{array}}{\Gamma_G;\emptyset \vdash f : \Pi\alpha_1 : \mathsf{Type}\ldots\alpha_m : \mathsf{Type}.\tau_1 \to \ldots \to \tau_n \to \kappa_o}\ \text{Sym}$$

Typing Rules

$$\frac{\Gamma, x:\kappa\ \text{wf}}{\Gamma, x:\kappa \vdash x : \kappa}\ \text{Var} \qquad \frac{\Gamma_G;\emptyset \vdash \kappa : \mathsf{Kind}}{\Gamma_G;\Gamma_L \vdash X_\kappa : \kappa}\ \text{Meta} \qquad \frac{\Gamma\ \text{wf}}{\Gamma \vdash \mathsf{Type} : \mathsf{Kind}}\ \mathsf{Type} \qquad \frac{\Gamma \vdash \tau : \mathsf{Type}}{\Gamma \vdash \tau : \mathsf{Kind}}\ \text{Sub}$$

$$\frac{\Gamma \vdash \kappa : \mathsf{Kind} \qquad \Gamma, x:\kappa \vdash P(x) : o}{\Gamma \vdash \epsilon(x:\kappa).P(x) : \kappa}\ \epsilon \qquad \frac{\Gamma \vdash \tau : \mathsf{Type} \qquad \Gamma \vdash a : \tau \qquad \Gamma \vdash b : \tau}{\Gamma \vdash a =_\tau b : o}\ =$$

$$\frac{\begin{array}{c}(f : \Pi\alpha_1 : \mathsf{Type}\ldots\alpha_m : \mathsf{Type}.\tau_1 \to \ldots \to \tau_n \to \kappa_o) \in \Gamma \\ \Gamma \vdash \tau'_i : \mathsf{Type}, i = 1\ldots m \qquad \Gamma \vdash e_i : \tau_i[\alpha_1/\tau'_1,\ldots,\alpha_m/\tau'_m], i = 1\ldots n\end{array}}{\Gamma \vdash f(\tau'_1,\ldots,\tau'_m;e_1,\ldots,e_n) : \kappa_o[\alpha_1/\tau'_1,\ldots,\alpha_m/\tau'_m]}\ \text{App}$$

$$\frac{\Gamma\ \text{wf}}{\Gamma \vdash \top : o}\ \top \qquad \frac{\Gamma\ \text{wf}}{\Gamma \vdash \bot : o}\ \bot \qquad \frac{\Gamma \vdash P : o}{\Gamma \vdash \neg P : o}\ \neg$$

$$\frac{\Gamma \vdash P : o \qquad \Gamma \vdash Q : o}{\Gamma \vdash P \wedge Q : o}\ \wedge \qquad \frac{\Gamma \vdash P : o \qquad \Gamma \vdash Q : o}{\Gamma \vdash P \vee Q : o}\ \vee$$

$$\frac{\Gamma \vdash P : o \qquad \Gamma \vdash Q : o}{\Gamma \vdash P \Rightarrow Q : o}\ \Rightarrow \qquad \frac{\Gamma \vdash P : o \qquad \Gamma \vdash Q : o}{\Gamma \vdash P \Leftrightarrow Q : o}\ \Leftrightarrow$$

$$\frac{\Gamma \vdash \kappa : \mathsf{Kind} \qquad \Gamma, x:\kappa \vdash P(x) : o}{\Gamma \vdash \forall x:\kappa.P(x) : o}\ \forall \qquad \frac{\Gamma \vdash \kappa : \mathsf{Kind} \qquad \Gamma, x:\kappa \vdash P(x) : o}{\Gamma \vdash \exists x:\kappa.P(x) : o}\ \exists$$

Figure 2: Typing Rules

"$\odot$" symbol is used for closure, while "|" separates two distinct nodes to be created, and R_r, R_s, R_t, and R_{ts} respectively denote reflexive, symmetric, transitive, and transitive-symmetric relations, including equality in particular. It should be noted that the δ and γ rules also deal with quantification over types.

The proof-search algorithm is 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 end with a closure rule, the tree is closed, and it is a proof of the goal. Search 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 never changes the formulas elsewhere.

Given an initially well-typed formula, it should be noted that the proof search rules generate only well-typed formulas. All these generated formulas 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.

Within polymorphic theories, the typed version of Zenon allows us to narrow the search space and produce smaller proofs, since the typing constraints are handled at the metalevel,

Closure Rules

$$\frac{\bot}{\odot}\ \odot_{\bot} \qquad \frac{P, \neg P}{\odot}\ \odot \qquad \frac{\neg R_r(\tau_1, \ldots, \tau_m; a, a)}{\odot}\ \odot_r$$

$$\frac{\neg\top}{\odot}\ \odot_{\neg\top} \qquad \frac{R_s(\tau_1, \ldots, \tau_m; a, b), \neg R_s(\tau_1, \ldots, \tau_m; b, a)}{\odot}\ \odot_s$$

Analytic Rules

$$\frac{\neg\neg P}{P}\ \alpha_{\neg\neg} \qquad \frac{P \wedge Q}{P, Q}\ \alpha_{\wedge} \qquad \frac{\neg(P \vee Q)}{\neg P, \neg Q}\ \alpha_{\neg\vee}$$

$$\frac{\neg(P \Rightarrow Q)}{P, \neg Q}\ \alpha_{\neg\Rightarrow} \qquad \frac{P \vee Q}{P \mid Q}\ \beta_{\vee} \qquad \frac{\neg(P \wedge Q)}{\neg P \mid \neg Q}\ \beta_{\neg\wedge}$$

$$\frac{P \Rightarrow Q}{\neg P \mid Q}\ \beta_{\Rightarrow} \qquad \frac{P \Leftrightarrow Q}{\neg P, \neg Q \mid P, Q}\ \beta_{\Leftrightarrow} \qquad \frac{\neg(P \Leftrightarrow Q)}{\neg P, Q \mid P, \neg Q}\ \beta_{\neg\Leftrightarrow}$$

$$\frac{\exists x : \kappa.P(x)}{P(\epsilon(x : \kappa).P(x))}\ \delta_{\exists} \qquad \frac{\neg\forall x : \kappa.P(x)}{\neg P(\epsilon(x : \kappa).\neg P(x))}\ \delta_{\neg\forall}$$

γ-Rules

$$\frac{\forall x : \kappa.P(x)}{P(X_\kappa)}\ \gamma_{\forall M} \qquad \frac{\neg\exists x : \kappa.P(x)}{\neg P(X_\kappa)}\ \gamma_{\neg\exists M}$$

$$\frac{\forall x : \kappa.P(x)}{P(t)}\ \begin{array}{l}\gamma_{\forall\text{inst}}\\ t : \kappa\end{array} \qquad \frac{\neg\exists x : \kappa.P(x)}{\neg P(t)}\ \begin{array}{l}\gamma_{\neg\exists\text{inst}}\\ t : \kappa\end{array}$$

Figure 3: Proof Search Rules (Part 1)

while the untyped version of Zenon needs an encoding of the polymorphic layer, as in [2]. This will become clear in Sec. 4 with a comparative benchmark in the B set theory, which is typed.

2.3 Handling Metavariables

In Zenon, metavariables, introduced by the rules $\gamma_{\forall M}$ and $\gamma_{\neg\exists M}$, play a special role. They serve to simulate a closure rule to determine a substitution by unification. But we do not substitute them everywhere in the tableau, instead we instantiate the formulas that introduced the metavariables. A single metavariable may therefore generate several instances.

In presence of polymorphism, type metavariables may also be introduced (by the same rules). In contrast with terms, we can afford more simplicity. Figs. 3 and 4 have rules with non-linearity constraints (e.g., the type arguments of P 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 formulas (in contrast with the a_i, b_i in pred), so unification is not postponed to closure. Instead, when trying to apply such a rule, we directly look for a (type metavariable) substitution that satisfies the constraints. In case of success, we instantiate the initial formulas. This shortcut minimizes both the search space and the size of proof trees.

As an example, consider a relation $P : \Pi\alpha : \mathsf{Type}.\alpha \to \alpha \to o$, and constants $\tau : \mathsf{Type}$ and $a, b : \tau$. Assuming $\forall\alpha : \mathsf{Type}.\forall x, y : \alpha.P(\alpha; x, y)$, we prove $P(\tau; a, b)$. The proof is given in Fig. 5

Relational Rules

$$\frac{P(\tau_1,\ldots,\tau_m;a_1,\ldots,a_n),\neg P(\tau_1,\ldots,\tau_m;b_1,\ldots,b_n)}{a_1\neq_{\tau'_1} b_1 \mid \ldots \mid a_n\neq_{\tau'_n} b_n}\ \text{pred}\quad a_i,b_i:\tau'_i, i=1\ldots n$$

$$\frac{f(\tau_1,\ldots,\tau_m;a_1,\ldots,a_n)\neq f(\tau_1,\ldots,\tau_m;b_1,\ldots,b_n)}{a_1\neq_{\tau'_1} b_1 \mid \ldots \mid a_n\neq_{\tau'_n} b_n}\ \text{fun}\quad a_i,b_i:\tau'_i, i=1\ldots n$$

$$\frac{R_s(\tau_1,\ldots,\tau_m;a,b),\neg R_s(\tau_1,\ldots,\tau_m;c,d)}{a\neq_\tau d \mid b\neq_\tau c}\ \text{sym}\quad a,b,c,d:\tau$$

$$\frac{\neg R_r(\tau_1,\ldots,\tau_m;a,b)}{a\neq_\tau b}\ \neg\text{refl}\quad a,b:\tau$$

$$\frac{R_t(\tau_1,\ldots,\tau_m;a,b),\neg R_t(\tau_1,\ldots,\tau_m;c,d)}{\begin{array}{c}c\neq_\tau a,\neg R_t(\tau_1,\ldots,\tau_m;c,a)\\ \mid b\neq_\tau d,\neg R_t(\tau_1,\ldots,\tau_m;b,d)\end{array}}\ \text{trans}\quad a,b,c,d:\tau$$

$$\frac{R_{ts}(\tau_1,\ldots,\tau_m;a,b),\neg R_{ts}(\tau_1,\ldots,\tau_m;c,d)}{\begin{array}{c}d\neq_\tau a,\neg R_t(\tau_1,\ldots,\tau_m;d,a)\\ \mid b\neq_\tau c,\neg R_{ts}(\tau_1,\ldots,\tau_m;b,c)\end{array}}\ \text{transsym}\quad a,b,c,d:\tau$$

$$\frac{a=_\tau b,\neg R_t(\tau_1,\ldots,\tau_m;c,d)}{\begin{array}{l}c\neq_\tau a,\neg R_t(\tau_1,\ldots,\tau_m;c,a)\\ \mid \neg R_t(\tau_1,\ldots,\tau_m;c,a),\neg R_t(\tau_1,\ldots,\tau_m;b,d)\\ \mid b\neq_\tau d,\neg R_t(\tau_1,\ldots,\tau_m;b,d)\end{array}}\ \text{transeq}\quad a,b,c,d:\tau$$

$$\frac{a=_\tau b,\neg R_{ts}(\tau_1,\ldots,\tau_m;c,d)}{\begin{array}{l}d\neq_\tau a,\neg R_{ts}(\tau_1,\ldots,\tau_m;d,a)\\ \mid \neg R_{ts}(\tau_1,\ldots,\tau_m;a,d),\neg R_{ts}(\tau_1,\ldots,\tau_m;b,c)\\ \mid b\neq_\tau c,\neg R_{ts}(\tau_1,\ldots,\tau_m;b,c)\end{array}}\ \text{transeqsym}\quad a,b,c,d:\tau$$

Figure 4: Proof Search Rules (Part 2)

(before pruning of useless formulas; see [7] for more details 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.

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 Deduction Modulo

Deduction modulo [18] reasons over equivalence classes of formulas under a congruence generated by rewrite rules. Compared to [18], we extend deduction modulo to first order logic with polymorphic types, so as to properly extend the typed proof search method of Zenon of Sec. 2.

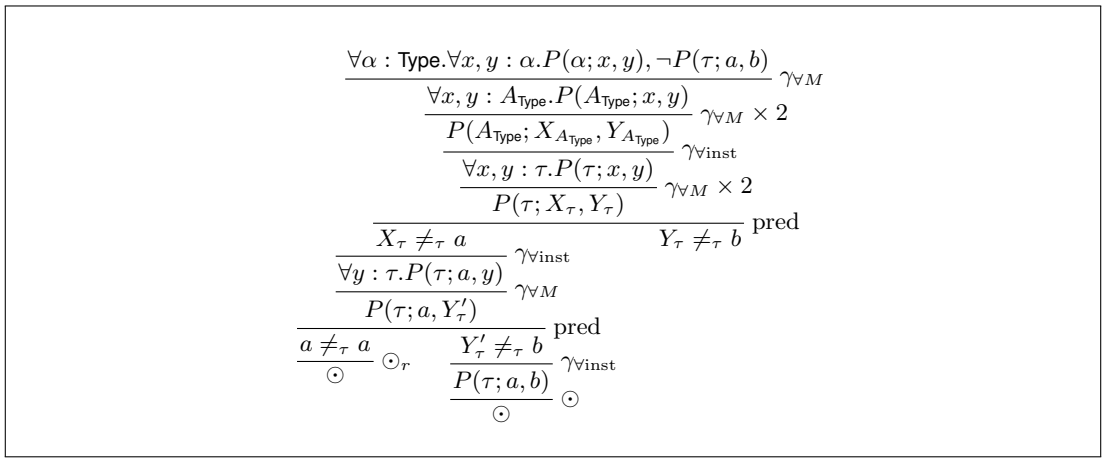


Figure 5: Proof Search Example with Metavariables

The language is that of Fig. 1, without metavariables and ϵ-terms. A term is an expression that is either a variable or an application, a proposition or formula is an expression that is not a term. In the following, given an expression e, $\mathrm{FV}(e)$ and $\mathrm{FV}_\tau(e)$ respectively stand for the set of (type and term) free variables and the set of free type variables of e.

Definition (Class Rewrite System)**.** *A term (resp. proposition) rewrite rule is a pair of terms (resp. formulas) l and r together with a local typing context Γ_L, denoted $l \longrightarrow_{\Gamma_L} r$, verifying $\mathrm{FV}(r) \subseteq \mathrm{FV}(l) \subseteq \Gamma_L$, and s.t. if l is a term, then it is not a variable, and if l is a formula, then it is atomic. An equational axiom is a pair of terms l and r together with a local typing context Γ_L, denoted $l =_{\Gamma_L} r$.*
Rewrite rules and equational axioms are said to be well-formed in a global typing context Γ_G if l and r have the same type in $\Gamma_G; \Gamma_L$.
A class rewrite system, denoted $\mathcal{RE}$, consists of a set of proposition rewrite rules $\mathcal{R}$ and a set of term rewrite rules and equational axioms $\mathcal{E}$. It is well-formed in a global typing context Γ_G if all the rewrite rules are well-formed in Γ_G.

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

Definition ($\mathcal{RE}$-Rewriting)**.** *Given a global typing context Γ_G and a class rewrite system $\mathcal{RE}$ that is well-formed in Γ_G, a formula φ is said to $\mathcal{RE}$-rewrite to φ', denoted $\varphi \longrightarrow_\mathcal{RE} \varphi'$, if $\varphi =_\mathcal{E} \psi$, $\psi_{|\omega} = \sigma(\rho(l))$, and $\varphi' = \psi[\sigma(\rho(r))]_\omega$, for some rule $l \longrightarrow_{\Gamma_L} r \in \mathcal{R}$, some formula ψ, some occurrence ω in φ, some type substitution ρ, some term substitution σ, and where $\psi_{|\omega}$ is the expression at occurrence ω in ψ, and $\psi[\sigma(\rho(r))]_\omega$ is the expression ψ where $\psi_{|\omega}$ has been replaced by $\sigma(\rho(r))$.*

If the formula φ is well-typed in Γ_G, then the type and term substitutions ρ and σ are well-formed in the sense that they replace types or terms with types or terms of the same type, and φ' has the same type than φ in Γ_G.

The relation $=_\mathcal{RE}$ is not decidable in general, but this is in particular the case when $\longrightarrow_\mathcal{RE}$ is confluent and (weakly) terminating, and $=_\mathcal{E}$ is decidable.

3.2 Extension of Zenon to Deduction Modulo

Given a global typing context Γ_G and a class rewrite system $\mathcal{RE}$ well-formed in Γ_G, extending Zenon to deduction modulo then consists in adding to the proof search rules of Figs. 3 and 4 the following conversion rule:

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

This presentation is more modular than the one described in [16], where the congruence $=_{\mathcal{RE}}$ is part of every proof search rule.

The metavariable instantiation mechanism of Sec. 2.3 needs adaptation: we look for formulas P and Q s.t. $P =_{\mathcal{RE}} P'$, $Q =_{\mathcal{RE}} \neg Q'$, and there exist a type substitution ρ and a term substitution σ s.t. $\sigma(\rho(P')) =_{\mathcal{E}} \sigma(\rho(Q'))$. To have a complete rewriting algorithm, we also extend metavariable instantiation to propositional narrowing: we look for a formula P, a type substitution ρ, and a term substitution σ s.t. $P =_{\mathcal{RE}} P'$, and there exist $P'_{|\omega}$ and a rule $l \longrightarrow_{\Gamma_L} r$ of $\mathcal{RE}$ s.t. $\sigma(\rho(P'_{|\omega})) =_{\mathcal{E}} \sigma(\rho(l))$.

3.3 Rules for a B Set Theory Modulo

Expressing the B set theory as a theory modulo amounts to building an adequate class rewrite system. To do so, we transform whenever possible the axioms and definitions of Chap. 2 of the B-Book [1] into rewrite rules (equational axiom are not needed), reusing the infix notation of the B-Book. The resulting theory is summarized in Figs. 6 and 7, where we omit the set BIG (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). It should be noted that the constructs and notations are, for a large part of them, specific to the B method, as they are used for the modeling of industrial projects, and are not necessarily standard in set theory.

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. 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 remove the type annotations in tuples when they are redundant with the membership construct, 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 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:

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

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

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

It should be noted that expressing the B set theory in a polymorphic way is necessary to transform it into a theory modulo, as it allows us to remove the parts of the formulas that are pure type information, and therefore obtain formulas that can be transformed into rewrite

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

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

Figure 6: Rules of the B Set Theory Modulo (Part 1)

rules. For example, in pure first order logic (without types), as done in the B-Book, the axiom for intersection is expressed as follows:

$$\forall x.s \subseteq u \Rightarrow t \subseteq u \Rightarrow (x \in s \cap t \Leftrightarrow x \in s \wedge x \in t)$$

The rule corresponding to this axiom requires conditional rewriting (namely the guards $s \subseteq u$ and $t \subseteq u$, that force s and t to belong to a same superset u: this is how the B-Book gives types to s and t). This has not yet been studied in the framework of deduction modulo (a start in this direction is [10]), and guard conditions must be therefore avoided in the axioms as much as possible.

To sum up, our formulation of the BIG- and comprehension-free B set theory sticks closely to the B-Book. Still, we need conservativity results, which boils 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 correctness of our "inlined" rewrite rules follows from immediate applications of the comprehension axiom itself, but without further considerations, the completeness fails. Sometimes, it is necessary to introduce a set defined by comprehension, for example to prove that "a set is not in bijection with its power set". However, this kind of smart trick is seldom met in practice. This is also related to completeness of the proof search algorithm, which performs in a cut-free calculus. In fact, elaborating a theory

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

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

Figure 7: Rules of the B Set Theory Modulo (Part 2)

modulo does not only consist in turning axioms into rewrite rules, but we must be careful to keep cut-free completeness, especially when this theory is the heart of a proof search method. The problem of cut elimination is known to be very difficult in deduction modulo [8], and we therefore leave these questions 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 [17, 21]. This benchmark comes from four anonymized industrial projects that were selected by the industrial partners for the representativity of their POs. The POs are not necessarily tricky mathematical properties, their difficulty comes from their size, the large context provided, or the number of quantified variables (the mean size of the statements of these POs in TFF1 format is 515 KiB, with a maximum of 2,690 KiB). To run the tests, we rely on the BWare verification platform, which we outline briefly. The POs are initially produced by Atelier B.

They are then translated into Why3 files [4], using a Why3 encoding of the B set theory [20]. Next, from these files, the Why3 platform produces (through appropriate drivers) the POs for 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 [3] of the TPTP community for the first order polymorphic ATPs (this translation is straightforward as there are very few differences between the polymorphic input language of Why3 and TFF1), or the regular FOF format with an encoding of the polymorphic layer [2] 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 and polymorphism.

The benchmark of BWare consists in 12,876 POs[1], and the experiment was run on an Intel Xeon E5-2660 v2 2.20 GHz computer, with a timeout of 120 s and a memory limit of 1 GiB. 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 and compared to the main prover (mp) of Atelier B 4.0. The second table compares 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 with deduction modulo and iProver Modulo implement deduction modulo, while only Zenon with types and Alt-Ergo have polymorphic types. Further information regarding this experiment can be found in Appx. C (in particular, the input format of the benchmark and the command line used for each tool are provided).

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. 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 the regular version of Zenon Modulo[2]. The arithmetic extension [12] handles linear arithmetic formulas, and relies on the simplex algorithm to compute solutions for systems over rationals, as well as on the branch and bound method to deal with integer systems [14]. As can be observed, the more extensions we plug, the more POs we prove. The most significant gain is provided by the type extension, where we get an increase of about 1755% compared to Zenon. Plugging deduction modulo gives an additional increase of 65%. Finally, connecting arithmetic on top allows us to prove 20% more POs, and to improve by 10 percentage points on mp.

Looking at 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 w.r.t. the number of proved POs. Vampire remains close to Zenon (10,154 proofs compared to 10,340 proofs for Zenon), but Zenon appears to be about 4 times faster than Vampire over all the proved POs (with a cumulative time of 31,665 s compared to 118,541 s for Vampire). Similarly, Zenon with all the extensions proves more POs than the SMT solvers CVC4 and Z3, except Alt-Ergo. However, it should be noted that CVC4 is close to Zenon (12,173 proofs compared to 12,281 proofs for

[1] This benchmark is publicly available at: `http://bware.lri.fr/`.
[2] Available at: `https://www.rocq.inria.fr/deducteam/ZenonModulo/`.

	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	7,919	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	36,969	8,378	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

Zenon), and has a significant lower cumulative time (8,378 s compared to 31,689 s for Zenon). 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 [16], 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 natively deals with rewriting and is well-suited to verify the proofs of Zenon Modulo. In particular, we do not 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 [16]). This translation has been replaced by a more syntactical one using excluded middle explicitly (see [13]), which is more efficient in practice. This backend deals with all the proofs not involving arithmetic, and all the 10,340 relevant proofs of Tab. 1 have been approved by Dedukti.

5 Conclusion

In this paper, we have observed the benefits of typing and rewriting for automated deduction, implemented in the Zenon Modulo tool, and applied to the B set theory. We have assessed this claim on thousands of POs of the BWare project. Our tool competes with state-of-the-art first order ATPs (without arithmetic) and SMT solvers (with arithmetic), and obtains better results than a large part of them. This tends to show that our tool scales up, and is ready to be applied

in any industrial project using the B method. More generally, this work could also be adapted to any framework of formal method based on set theory modeling, as the B set theory includes few specificities compared to Zermelo-Fraenkel set theory for example.

As future work, we aim to integrate conditional rewrite rules to Zenon Modulo along the lines of [10], so as to turn more axioms of the B set theory into rewrite rules (currently, some axioms remain, for instance function application). We also plan to study theoretical properties, in particular cut-free completeness. As there is no general way to ensure this in deduction modulo [11], we will probably use specific techniques to our case. We also plan 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 yet more compact certificates. Finally, within the BWare project, we will increase the variety of available domains and applications of the benchmark, by integrating a new large project. The final benchmark will contain 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 usable in 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 certificates checkable by Dedukti.

References

[1] J.-R. Abrial. *The B-Book, Assigning Programs to Meanings.* Cambridge University Press, Cambridge (UK), 1996. ISBN 0521496195.

[2] J. C. Blanchette, S. Böhme, A. Popescu, and N. Smallbone. Encoding Monomorphic and Polymorphic Types. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 7795 of *LNCS*, pages 493–507, Rome (Italy), Mar. 2013. Springer.

[3] 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.

[4] F. Bobot, J.-C. Filliâtre, C. Marché, and A. Paskevich. Why3: Shepherd Your Herd of Provers. In *International Workshop on Intermediate Verification Languages (Boogie)*, Wrocław (Poland), Aug. 2011.

[5] M. Boespflug, Q. Carbonneaux, and O. Hermant. The λΠ-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] R. Bonichon and O. Hermant. A Semantic Completeness Proof for Tableaux Modulo. In *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, volume 4246 of *LNCS*, pages 167–181, Phnom Penh (Cambodia), Nov. 2006. Springer.

[9] C. E. Brown and G. Smolka. Analytic Tableaux for Simple Type Theory and its First-Order Fragment. *Logical Methods in Computer Science (LMCS)*, 6(2), June 2010.

[10] 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.

[11] G. Burel and C. Kirchner. Regaining Cut Admissibility in Deduction Modulo using Abstract Completion. *Information and Computation*, 208(2):140–164, Feb. 2010.

[12] G. Bury and D. Delahaye. Integrating Simplex with Tableaux. In *Automated Reasoning with Analytic Tableaux and Related Methods (TABLEAUX)*, volume 9323 of *LNCS*, pages 86–101, Wrocław (Poland), Sept. 2015. Springer.

[13] R. Cauderlier and P. Halmagrand. Checking Zenon Modulo Proofs in Dedukti. In *Proof Exchange for Theorem Proving (PxTP)*, volume 186 of *EPTCS*, pages 57–73, Berlin (Germany), Aug. 2015.

[14] V. Chvátal. *Linear Programming*. Series of Books in the Mathematical Sciences. W. H. Freeman and Company, New York (USA), 1983. ISBN 0716715872.

[15] ClearSy. *Atelier B 4.2.1*, Mar. 2015. `http://www.atelierb.eu/`.

[16] 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.

[17] 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)*, volume 8477 of *LNCS*, pages 126–127, Toulouse (France), June 2014. Springer.

[18] G. Dowek, T. Hardin, and C. Kirchner. Theorem Proving Modulo. *Journal of Automated Reasoning (JAR)*, 31(1):33–72, Sept. 2003.

[19] M. Giese. A Calculus for Type Predicates and Type Coercion. In *Automated Reasoning with Analytic Tableaux and Related Methods (TABLEAUX)*, volume 3702 of *LNCS*, pages 123–137, Koblenz (Germany), Sept. 2005. Springer.

[20] 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*, pages 238–251, Pisa (Italy), June 2012. Springer.

[21] The BWare Project, 2012. `http://bware.lri.fr/`.

[22] C. Weidenbach. First-Order Tableaux with Sorts. *Logic Journal of the IGPL*, 3(6):887–906, Oct. 1995.

A Typing of the Theory Modulo for the B Set Theory

Type Constructors

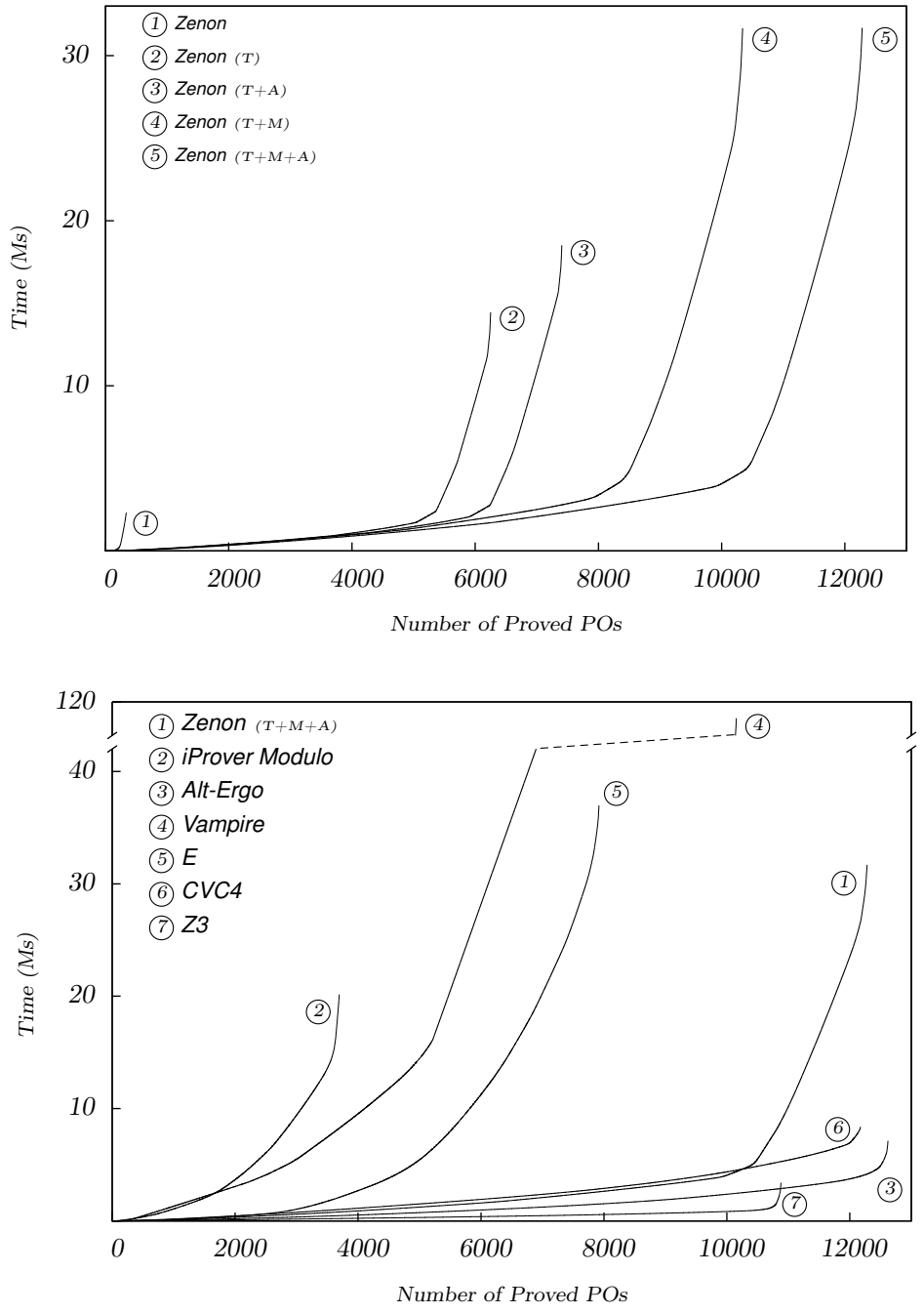
$\mathsf{tup} : \Pi\alpha_1, \alpha_2 : \mathsf{Type}.\mathsf{Type}$ $\mathsf{set} : \Pi\alpha : \mathsf{Type}.\mathsf{Type}$

Type Schemes of the Set Constructs

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

Figure 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

Figure 9: Cumulative Times According to the Numbers of Proved POs

C Detailed Information regarding the Experiment

The experiment results presented in Sec. 4 were performed over a benchmark of 12,876 POs, which has been provided by two industrial partners of the BWare project. The benchmark is publicly available (under the CeCILL-B license[3]) at: `http://bware.lri.fr/`. Several formats are proposed and divided into several archives. The considered formats are the following:

- TPTP FOF (regular TPTP format for mono-sorted first order logic);
- TPTP TFF1 (TPTP format for first order logic with polymorphic types);
- SMT-LIB v2 (regular SMT format for many-sorted first order logic);
- Alt-Ergo (input native format of Alt-Ergo).

The experiment was run on an Intel Xeon E5-2660 v2 2.20 GHz computer, with a timeout of 120 s and a memory limit of 1 GiB. For each tool (except mp, which was tested directly over the native format of POs coming from Atelier B), the following input formats and command lines (where `%t` is the timeout, `%m` the memory limit, and `%f` the file name) were used:

- Zenon Modulo 0.4.1 (Zenon with types, deduction modulo, and arithmetic):
  Input format: TPTP TFF1;
  Command line: "`zenon_modulo -p0 -itptp -b-rwrt -rwrt -x arith -max-size %mM -max-time %ts %f`".
- iProver Modulo v0.7+0.2:
  Input format: TPTP FOF;
  Command line: "`iprover_modulo_launcher.sh %f %t --strategies 'Id;Equiv(ClausalAll)' --normalization_type dtree --omit_eq false --dedukti_out_proof false`".
- Alt-Ergo 0.99.1:
  Input format: Alt-Ergo;
  Command line: "`alt-ergo -timelimit %t %f`".
- Vampire 2.6:
  Input format: TPTP FOF;
  Command line: "`vampire --proof tptp --mode casc -t %t %f`".
- E 1.8:
  Input format: TPTP FOF;
  Command line: "`eprover --auto --tptp3-format %f`".
- CVC4 1.4:
  Input format: SMT-LIB v2;
  Command line: "`cvc4 --lang=smt2 --rlimit %t000 %f`".
- Z3 4.3.2:
  Input format: SMT-LIB v2;
  Command line: "`z3 -smt2 -rs:42 %f`".

[3]CeCILL is a French free software license, compatible with the GNU GPL. For more information, see: `http://www.cecill.info/licences.en.html`.