Implementing Polymorphism in Zenon [*]

Guillaume Bury, Raphaël Cauderlier and Pierre Halmagrand

Cedric/Cnam/Inria, Paris, France,
Guillaume.Bury@inria.fr Raphael.Cauderlier@inria.fr Pierre.Halmagrand@inria.fr

Abstract

Extending first-order logic with ML-style polymorphism allows to define generic axioms dealing with several sorts. Until recently, most automated theorem provers relied on preprocess encodings into mono/many-sorted logic to reason within such theories. In this paper, we discuss the implementation of polymorphism into the first-order tableau-based automated theorem prover Zenon. This implementation led us to modify some basic parts of the code, from the representation of expressions to the proof-search algorithm.

1 Introduction

Formal verification tends to be a common milestone in the development of software for safety-critical systems. Among the family of verification techniques, those using automated deductive tools raised confidence to a high level during last decades and are now used in a wide range of fields. These achievements were made possible thanks to the ability of such automated deductive tools to reason on specific theories, like set theory or arithmetic for instance, helped by an efficient decision procedure for each theory. When reasoning in several theories combining different sorts, it is often necessary to express some general axioms regarding all different sorts. One solution is to postulate one axiom per sort, leading to a multiplication of axioms. Another solution is to express axioms in a generic way. First-order logic extended with ML-style polymorphism (FOL-ML) is a good candidate to address this issue.

Until recently, most automated deductive tools, like automated theorem provers (ATP) or SMT solvers, were not handling polymorphism. Whenever someone wanted to use such an ATP or SMT solver to prove statements coming from a FOL-ML theory, he relied on a preprocessing phase to encode into a mono/many-sorted logic [1]. Such encodings generally modify theories by deconstructing the shape of formulas and adding some new axioms, leading to less efficient proof search. A solution to keep the original form of the input theory and the statement is to develop some deductive tools which natively understand polymorphism. As far as the authors know, implementation of polymorphism into an automated deductive tool began with the development of the SMT solver Alt-Ergo [3]. Two other projects have since been released, both based on superposition. The first one is a prototype based on the prover SPASS [8], and the other one is the new ATP Zipperposition [7].

Zenon [4] is a first-order monosorted tableau-based ATP. We present in this paper some insights about the implementation of polymorphism into Zenon. This extension has required to properly adapt a large part of the existing code, from the representation of expressions to the proof-search algorithm.

This paper is organized as follows: in Sec. 2, we describe the new syntax for typed expressions, in Sec. 3, we give the type-checking algorithm, and finally in Sec. 4, we discuss modifications in the proof-search algorithm and give the results of a benchmark.

[*] This work has received funding from the BWare project (ANR-12-INSE-0010) funded by the INS programme of the French National Research Agency (ANR).

Expressions

$a, b, \ldots$	$::=$	v	*(typed variable)*
	$\mid$	A	*(metavariable)*
	$\mid$	$\epsilon(e)$	*(Hilbert's operator)*
	$\mid$	$f(a_1, \ldots, a_n)$	*(function/predicate application)*
	$\mid$	$a_1 \rightarrow \ldots \rightarrow a_n \rightarrow b$	*(arrow type)*
	$\mid$	$\top \mid \bot$	*(true and false)*
	$\mid$	$a = b$	*(equality)*
	$\mid$	$\neg a \mid a \wedge b \mid a \vee b \mid a \Rightarrow b \mid a \Leftrightarrow b$	*(logical connectives)*
	$\mid$	$\forall v : a.\ b$	*(universal quantification on variables)*
	$\mid$	$\exists v : a.\ b$	*(existential quantification on variables)*

Figure 1: AST for types, terms and formulas

2 Syntax of typed expressions

In Zenon, both terms and formulas are represented using a single abstract syntax tree, presented in Figure 1. This decision follows from the use of Hilbert's operator to handle existentially quantified formulas, which introduces terms that depend on formulas. When we implemented typed expressions in Zenon, we chose to extend that single abstract syntax tree (AST) with the arrow type constructor, so that it could also represent types, rather than introduce another AST for types. This allowed us to minimize modifications to the code base, as well as reuse existing code and benefit from features already implemented such as hashconsing and substitutions.

In our implementation, each function/predicate and node of the AST is tagged with an optional type (itself built using the same AST). We then have four distinct classes of expressions built using the AST :

- A constant Type, with an empty tag
- Types are built using variables, meta-variables, and ϵ-terms with tag Type, universal quantification[1], the arrow type constructor, and application of type constructors, i.e functions whose type is of the form: Type $\rightarrow \ldots \rightarrow$ Type $\rightarrow$ Type. Types are tagged with Type.
- Terms are built using variables, meta-variables, ϵ-terms and application of functions. Terms are tagged with a type.
- Formulas are built using $\top$, $\bot$, equality of terms, application of predicates, logical connectives, universal quantification and existential quantification of formulas. Formulas are tagged with a type constant Prop.

We then have access to the type of expressions through the get_type function, which returns either Type, or a type.

3 Type checking

For efficiency reasons, the type-checking phase in Zenon occurs before the beginning of proof search so that expressions are checked once and for all. Since the equality relation uses implicit

[1] Universal quantification in types is used to represent the type of polymorphic functions and predicates.

$$\frac{(v:\tau)\in\Gamma}{\Sigma,\Gamma\vdash v\Rightarrow\tau}\qquad\frac{(f:\forall\alpha_1.\ \ldots\forall\alpha_n.\ \tau'_1\to\ldots\to\tau'_m\to\tau)\in\Sigma}{\Sigma,\Gamma\vdash f(\tau_1,\ldots,\tau_n,a_1,\ldots,a_m)\Rightarrow\tau\{\alpha_1:=\tau_1,\ldots,\alpha_n:=\tau_n\}}$$

$$\frac{(v:\tau)\in\Gamma}{\Sigma,\Gamma\vdash v\Leftarrow\tau}\qquad\frac{\Sigma,\Gamma\vdash a\Rightarrow\tau\qquad\Sigma,\Gamma\vdash a\Leftarrow\tau\qquad\Sigma,\Gamma\vdash b\Leftarrow\tau}{\Sigma,\Gamma\vdash a=b\Leftarrow\mathsf{Prop}}$$

$$\frac{c\in\{\top,\bot\}}{\Sigma,\Gamma\vdash c\Leftarrow\mathsf{Prop}}\qquad\frac{\Sigma,\Gamma\vdash a\Leftarrow\mathsf{Prop}}{\Sigma,\Gamma\vdash\neg a\Leftarrow\mathsf{Prop}}\qquad\frac{\Box\in\{\wedge,\vee,\Rightarrow,\Leftrightarrow\}\ \Sigma,\Gamma\vdash a\Leftarrow\mathsf{Prop}\ \Sigma,\Gamma\vdash b\Leftarrow\mathsf{Prop}}{\Sigma,\Gamma\vdash a\Box b\Leftarrow\mathsf{Prop}}$$

$$\frac{Q\in\{\forall,\exists\}\qquad\tau=\mathsf{Type}\text{ or }\Sigma,\Gamma\vdash\tau\Leftarrow\mathsf{Type}\qquad\Sigma,\Gamma,v:\tau\vdash a\Leftarrow\mathsf{Prop}}{\Sigma,\Gamma\vdash Qv:\tau.\ a\Leftarrow\mathsf{Prop}}$$

$$\frac{\Sigma,\Gamma,v:\mathsf{Type}\vdash a\Leftarrow\mathsf{Type}}{\Sigma,\Gamma\vdash\forall v:\mathsf{Type}.\ a\Leftarrow\mathsf{Type}}\qquad\frac{\Sigma,\Gamma\vdash\tau_i\Leftarrow\mathsf{Type}\qquad\text{for }i\in[0,n]}{\Sigma,\Gamma\vdash\tau_1\to\ldots\to\tau_n\to\tau_0\Leftarrow\mathsf{Type}}$$

$$\frac{\begin{array}{c}(f:\forall\alpha_1.\ \ldots\forall\alpha_n.\ \tau'_1\to\ldots\to\tau'_m\to\tau'_0)\in\Sigma\\ \sigma:=\{\alpha_1:=\tau_1,\ldots,\alpha_n:=\tau_n\}\\ \sigma(\tau'_0)=\tau_0\end{array}\qquad\begin{array}{c}\Sigma,\Gamma\vdash\tau_i\Leftarrow\mathsf{Type}\\ \Sigma,\Gamma\vdash a_i\Leftarrow\sigma(\tau'_i)\end{array}\qquad\text{for all }i\in[1,n]}{\Sigma,\Gamma\vdash f(\tau_1,\ldots,\tau_n,a_1,\ldots,a_m)\Leftarrow\tau_0}$$

Figure 2: Zenon's type-checking algorithm

polymorphic typing, we require each quantifier in the input problem to specify the type of the variables it binds (otherwise, formulas such as $\forall x.\ x=x$ would be ambiguous) and each function, type constructor, and predicate symbol to be declared with its type (this is required to type formulas such as $f(0)=f(1)$). Since equality is the only implicitly polymorphic symbol, we do not really need to infer types for all expressions but only for terms. We denote by $\Sigma,\Gamma\vdash t\Rightarrow\tau$ the functional relation mapping a term to its inferred type and by $\Sigma,\Gamma\vdash a\Leftarrow\tau$ the type-checking relation. These relations are defined in Figure 2 using a syntax-directed set of typing rules.

We do not need to give rules for expressions which are not present in the input problem such as meta variables and Hilbert's epsilons. Moreover, we only need to define inference for terms. Therefore it can be done by a single look-up in Σ or Γ. However, inference can return a type for an ill-typed term because it does not take subterms into account; this is the reason why in the rule for checking equality, we check back that a has the type returned by the inference machinery.

3.1 Typing substitutions

During proof search, the only way to generate ill-typed expressions is by applying substitutions, for example in the rules instantiating quantifiers and unfolding definitions. To avoid this issue, we check that substitutions are well-typed. However, there are at least two ways to define the notion of well-typed substitution:

- Strongly well-typed substitution:
  The substitution $\sigma=\{x_1:=t_1,\ldots,x_n:=t_n\}$ is strongly well-typed if each x_i has the same type as t_i

- Weakly well-typed substitution:
  The substitution $\sigma=\{x_1:=t_1,\ldots,x_n:=t_n\}$ is weakly well-typed if each $x_i\{x_1:=t_1,\ldots,x_{i-1}:=t_{i-1}\}$ has the same type as $t_i\{x_1:=t_1,\ldots,x_{i-1}:=t_{i-1}\}$

The function performing substitutions in Zenon does not preserve strong well-typedness in its recursive calls but only weak well-typedness which is, fortunately, enough to guarantee that applying the substitution on a well-typed expression will result in a well-typed expression.

Strong well-typedness is faster to check because we only need to traverse the list $[(x_1, t_1); \dots ; (x_n, t_n)]$ once, leading to linear complexity. Checking that two terms have the same type is performed in essentially constant time thanks to the type annotations of all expression nodes (hence obtaining the types is fast) and hashconsing (hence comparing expressions is most of the time as fast as comparing their hashes). On the other hand, checking weak well-typedness is of quadratic complexity.

As a compromise between safety and efficiency we distinguish two substitution functions: the old one, `substitute_unsafe`, preserving weak well-typedness in its recursive calls but not performing any typing check; and a wrapper function `substitute_safe` checking[2] that the substitution it gets as argument is strongly well-typed and then calling `substitute_unsafe`.

In new version of Zenon extended with typing, only `substitute_safe` is used during proof search and other parts of the new Zenon which need to substitute in well-typed expressions.

4 Proof Search

4.1 Dealing with Type Metavariables

Extension of Zenon to polymorphism slightly modifies the implementation of the proof-search algorithm. The main modification deals with universal quantification over type variables. When Zenon encounters a universally quantified formula φ, it generates a so-called metavariable linking to φ by applying the $\delta_{\forall M}$ rule [5]. The original formula φ is kept in the context for later instantiation.

For metavariables linked to quantified formulas over terms, the behavior of Zenon has not changed: such metavariables are only used as tricks to find, by unification, some possible instantiations for the original formulas that allow to close the local branches. After finding a relevant value, Zenon instantiates the original formula by applying the $\delta_{\forall_{\text{inst}}}$ rule and continues its proof search.

For type metavariables, we have to instantiate original formulas as soon as possible, when it makes possible the application of a further rule. Actually, only the relational rules of Zenon (those dealing with the equality symbol) are concerned, because the possibility to apply them depend on side conditions over the type of their parameters [5]. So, if we have some type metavariables in a formula and if there are some possible instantiations that allow to apply one of these rules, we instantiate the original formulas linked to the metavariables. In such a way, we ensure to capture all the possible instantiations needed for the proof search.

4.2 Experimental Results

To assess our extension of Zenon to polymorphism, we performed an experiment using a benchmark made of all the 337 problems with a theorem status coming from the TFF1 [2] category of the TPTP library, run on an Intel Xeon E5-2660 v2 2.20 GHz computer, with a timeout of 30 s and a memory limit of 1 GiB. We compare the new typed version of Zenon[3] presented in this paper with the previous monosorted one (using the encoding of types into pure first-order logic [1] implemented into the Why3 platform) and the other automated deductive tools dealing with

[2] these checks can also be disabled

[3] Available at: `https://www.rocq.inria.fr/deducteam/ZenonModulo/`.

337 Problems	Zenon Old	Zenon Typed	Zipperposition	Alt-Ergo
Proved	96	106	150	221
Mean Time (sec.)	1.9	0.95	3.3	0.64

Table 1: Experimental Results over the TPTP TFF1 Benchmark

polymorphism, Zipperposition v0.6.1 and Alt-Ergo v0.99.1, except the prototype based on SPASS which does not yet read TFF1 syntax. The results are summarized in Tab. 1 and, for each prover, they give the number of proved problems and the mean time needed to prove a problem. This experiment shows that the polymorphic version of Zenon proves 10 more problems than the monosorted one while being twice as fast. On the other side, the superposition-based ATP Zipperposition proves 44 more problems than Zenon with a larger mean time and the SMT solver Alt-Ergo proves 115 more problems with a lower mean time.

5 Conclusion

We have extended the automated theorem prover Zenon to polymorphism. Since we were adapting an existing code, we chose to minimize the impact of this extension to the original structure of Zenon. The experimental results, presented in this paper, show that the polymorphic version of Zenon is more efficient than the monosorted one on polymorphic problems.

Considering the significance of polymorphism in program verification, we hope that more provers will integrate expressive typing systems in the future, especially since it does not affect the generation of proof certificates [6] because proof checkers usually provide rich typing systems.

References

[1] 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*. Springer, 2013.

[2] J. C. Blanchette and A. Paskevich. TFF1: The TPTP Typed First-Order Form with Rank-1 Polymorphism. In *Conference on Automated Deduction (CADE)*. Springer, 2013.

[3] F. Bobot, S. Conchon, E. Contejean, and S. Lescuyer. Implementing Polymorphism in SMT solvers. In *SMT 2008: 6th International Workshop on Satisfiability Modulo*, 2008.

[4] 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)*. Springer, 2007.

[5] G. Bury, D. Delahaye, D. Doligez, P. Halmagrand, and O. Hermant. Automated Deduction in the B Set Theory using Typed Proof Search and Deduction Modulo. In *Logic for Programming Artificial Intelligence and Reasoning (LPAR)*, 2015.

[6] R. Cauderlier and P. Halmagrand. Checking Zenon Modulo proofs in Dedukti. In *Proof Exchange for Theorem Proving (PxTP)*, EPTCS, Aug. 2015.

[7] S. Cruanes. *Extending Superposition with Integer Arithmetic, Structural Induction, and Beyond.* PhD thesis, Ecole Polytechnique, 2015.

[8] D. Wand. Polymorphic+Typeclass Superposition. In B. Konev, L. de Moura, and S. Schulz, editors, *4th Workshop on Practical Aspects of Automated Reasoning (PAAR 2014)*, Vienna, Austria, 2014.