Ergo : a theorem prover for polymorphic first-order logic modulo theories

Sylvain Conchon, Evelyne Contejean, and Johannes Kanig

Laboratoire de Recherche en Informatique, Univ Paris-Sud, CNRS, Orsay F-91405
INRIA Futurs, ProVal, Parc Orsay Université, F-91893

Abstract. Ergo is a little engine of proof dedicated to program verification. It fully supports quantifiers and directly handles polymorphic sorts. Its core component is $\mathsf{CC}(\mathsf{X})$, a new combination scheme for the theory of uninterpreted symbols and a built-in theory X. Its architecture is highly modular and it consists only of 3000 lines of Ocaml code. It is currently used to prove correctness of C and Java programs.

1 Introduction

An important application of automated theorem proving is deductive verification of programs. This is usually based on verification condition generators (VCG) which turn an annotated program into a set of formulas to be proved valid.

In this context, the goals to be proved are first-order multi-sorted formulas with quantifiers, built-in equalities and integer arithmetic. Sorts naturally arise from the usual data-types of programming languages (integers, arrays etc.) and from the user specifications. In our case, polymorphism is used as a convenience to write the generic parts of specifications used by the VCGs.

Unfortunately, as can be seen in the SMT-competition results, there are very few provers under active development that deal with quantifiers. Moreover, there are no available provers at all that can directly handle polymorphic logics. In order to use the existing provers (Simplify [3], Yices [4], CVC-Lite [1], etc.) which are either unsorted or multi-sorted but monomorphic, one solution is to guess the monomorphic instances which are needed, the other one is to use an encoding. However, none of these solutions is satisfactory and it is more convenient to handle directly these features in the theorem prover.

Ergo fully supports quantifiers and deals directly with polymorphism. It is based on $\mathsf{CC}(\mathsf{X})$, a new combination scheme for the theory of uninterpreted symbols and a built-in theory X. Currently X is instanciated by the linear arithmetic. Ergo accepts two input syntaxes, the polymorphic logic of the Why tool [8] and the standard [6] defined by the SMT-lib initiative. The tool is written in Ocaml and is very light ($\sim$ 3000 lines of code). It is freely distributed under the Cecill-C licence at `http://ergo.lri.fr`.

2 General architecture

The architecture of Ergo is highly modular: each part (except the parsers) of the code is described by a small set of inference rules and is implemented as a (possibly parameterized) module. Figure 1 describes the dependencies between the modules. Each

input syntax is handled by the corresponding parser. Both of them produce an abstract syntax tree in the same datatype. Hence, there is a single typing module for both input syntaxes. The main loop consists of 4 modules:

- A home-made efficient SAT-solver with backjumping that also keeps track of the lemmas of the input problem and those that are generated during the execution.
- CC(X) handles the ground atoms assumed by the SAT-solver. It combines uninterpreted symbols with a theory X via a congruence closure algorithm.
- The equivalence modulo X itself is implemented by the parameterized module UF(X) based on union-find techniques.
- A matching module builds instances of the lemmas contained in the SAT-solver modulo the equivalence classes of CC(X).

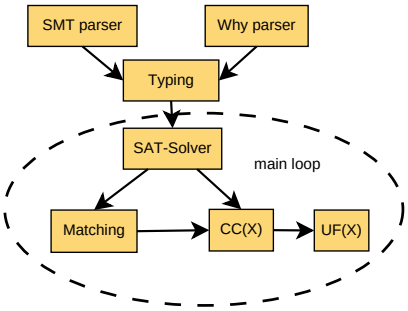

Fig. 1. The modular architecture of Ergo.

2.1 Matching and Polymorphism

This section illustrates the subtleties introduced by the polymorphism. The following example written in the Why syntax defines the sort of polymorphic lists (α `list`) and its constructors (`nil` and `cons`) as well as a function `length` with its properties (`a1` and `a2`).

```
type α list
logic nil: α list
logic cons: α, α list → α list
logic length: α list → int

axiom a1: length(nil) = 0
axiom a2: ∀x:α. ∀l:α list.  length(cons(x,l)) = 1 + length(l)
```

In this syntax, the type variables such as α are implicitly universally quantified. Some of them occur explicitly in the types of the quantified term variables (such as `x:`α in `a2`), but others are hidden in polymorphic constants. This is the case for the type variable α of the constant `nil` in the axiom `a1`, which is internally translated by Ergo to:

```
axiom a1' : ∀α. length(nil:α list) = 0
```

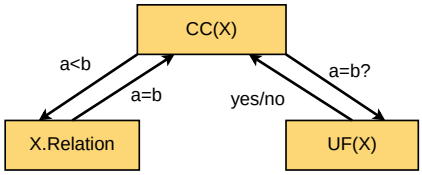

Fig. 2. CC(X) architecture

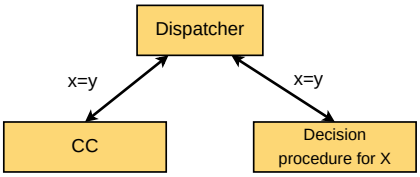

Fig. 3. Nelson-Oppen architecture

Now, in order to prove the goal `length(cons(1,nil))=1`, by `a2` we are left to prove that `1+length(nil)=1` where `nil` has the type `int list`. The only way to show this is by instantiating the type variable α by `int` in `a1'`. The matching module of Ergo thus has to instantiate both type and term variables.

2.2 CC(X)

This module implements a new combination scheme *à la Shostak* [7] between the theory of uninterpreted symbols and a theory X. CC(X) means "congruence closure parameterized by X". The module X should provide a decision procedure X.Relation for its relational symbols except for the equality which is handled by the module UF(X) (a union-find algorithm parameterized by X). As shown in Figure 2, the combination relies on the following exchanges:

- CC(X) sends relations between *representatives* in UF(X) to X.Relation. Using representatives automatically propagates the equalities implied by UF(X). In return, X.Relation sends its discovered equalities.
- CC(X) *asks* UF(X) *for relevant equalities* to propagate for congruence. Due to the union-find mechanism, asking for relevant equalities is much more efficient than letting UF(X) try to discover *all* new equalities.

This is different from the Nelson-Oppen combination [5] where, as shown in Figure 3, the combined modules have to discover and propagate all their new equalites.

2.3 Implementation Details

Ergo's efficiency mostly relies on the technique of hash-consing. Beyond the obvious advantage of saving memory blocks by sharing values that are structurally equal, hash-consing may also be used to speed up fundamental operations and data structures by several orders of magnitude when sharing is maximal. The hash-consing technique is also used to elegantly avoid the blow-up in size due to the CNF conversion [2] in the SAT-solver.

Since CC(X) is tightly coupled to the toplevel SAT-solver, the backtracking mechanism performed by the SAT module forces CC(X) to come back to its previous states. This is efficiently achieved by using functional data-structures of Ocaml.

3 Benchmarks

We benchmarked four provers on our test suite: 1349 verification conditions automatically generated by the VCG Caduceus/Why from 61 C programs. They were run with a fixed timeout of 20s on a machine with Xeon processors (3.20 GHz) and 2 Gb of memory.

	valid	timeout	unknown
Simplify	98%	1%	1%
Yices	95%	2%	3%
CVC-Lite	67%	30%	3%
Ergo	95%	4%	1%

For Yices and CVC-Lite, the polymorphism is handled by an encoding. For Simplify, sorts have just been erased (which is unsound).

4 Conclusion

We have presented Ergo, a new theorem prover for first-order polymorphic logic with built-in theories. The development started in January 2006 and the current experimentations are very promising with respect to speed and to the number of goals automatically solved.

Since this prover is mainly dedicated to the resolution of verification conditions generated by the toolkit Krakatoa/Caduceus/Why[8], its future evolution is partly guided by the needs of this tool: designing efficient proof strategies to manage huge contexts and useless hypotheses and adding more built-in theories such as pointer arithmetic.

Another direction is to "prove the prover" in a proof assistant. Indeed, Ergo uses only purely functional data-structures, is highly modular and very concise ($\sim$ 3000 lines of code). All these features should make a formal certification feasible.

References

1. C. Barrett. CVC Lite. `http://www.cs.nyu.edu/acsys/cvcl/`.
2. S. Conchon and J.-C. Filliâtre. Type-Safe Modular Hash-Consing. In *ACM SIGPLAN Workshop on ML*, Portland, Oregon, Sept. 2006.
3. D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.
4. B. Dutertre and L. de Moura. Yices. `http://yices.csl.sri.com/`.
5. G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Trans. on Programming, Languages and Systems*, 1(2):245–257, Oct. 1979.
6. S. Ranise and C. Tinelli. The SMT-LIB Standard: Version 1.2. Technical report, Department of Computer Science, The University of Iowa, 2006. Available at `www.SMT-LIB.org`.
7. R. E. Shostak. Deciding combinations of theories. *J. ACM*, 31:1–12, 1984.
8. The ProVal Project. Krakatoa/Caduceus/Why toolkit. `http://{krakatoa,caduceus,why}.lri.fr/`.