Lightweight Integration of the Ergo Theorem Prover inside a Proof Assistant [*]

Sylvain Conchon Evelyne Contejean
LRI, Univ Paris-Sud, CNRS
Orsay F-91405
INRIA Futurs, ProVal
Orsay, F-91893
{conchon,contejea}@lri.fr

Johannes Kanig Stéphane Lescuyer
INRIA Futurs, ProVal
Orsay, F-91893
LRI, Univ Paris-Sud, CNRS
Orsay F-91405
{kanig,lescuyer}@lri.fr

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 CC(X), a new combination scheme for the theory of uninterpreted symbols parameterized by a built-in theory X. In order to make a sound integration in a proof assistant possible, Ergo is capable of generating proof traces for CC(X). Alternatively, Ergo can also be called interactively as a simple oracle without further verification. It is currently used to prove correctness of C and Java programs as part of the Why platform.

1. INTRODUCTION

Critical software applications in a broad range of domains including transportation, telecommunication or electronic transactions are put on the market at an increasing rate. In order to guarantee the behavior of such programs, it is mandatory for a large part of the validation to be done in a mechanical way. In the ProVal project, we develop a platform [14] combining several tools of our own whose overall architecture is described in Figure 1. This toolkit enables the deductive verification of Java and C source code by generating *verification conditions* out of *annotations* in the source code. The annotations describe the logical specification of a program and the verification conditions are formulas whose validity ensures that the program meets its specification.

Much of the work of generating verification conditions for both Java and C programs is performed by Why, the tool which plays a central role in our toolkit and implements an approach designed by Filliâtre [13]. A main advantage of this architecture is that Why can output verification conditions to a large range of interactive higher-order provers (Coq, PVS, HOL, ...) and first-order automated provers such as CVC3 [3], Simplify[11], Yices [10], Z3 [9] or Ergo [4].

When using first-order automatic provers, a great number of formulas can be discharged in very little time in comparison to the tedious process of interactively proving these formulas in Coq or Isabelle.

[*] Work partially supported by A3PAT project of the French ANR (ANR-05-BLAN-0146-01).

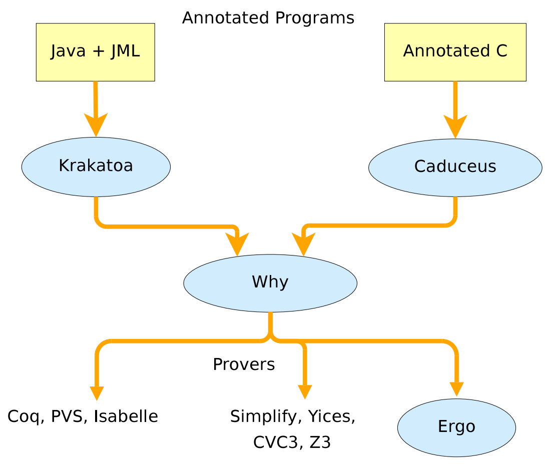

Figure 1: The Proval tool chain

The immediate downside of this method is that the soundness then depends on the soundness of the automated provers, which weakens the chain of trust. Depending on whether the automated provers shall be trusted or not, there are different ways of integrating them in the system : some may want to use automated provers as “black boxes” and invoke them from within the prover (like PVS does), while others will rather have the prover produce some *traces* of its proofs and typecheck these traces. The problem with the latter is that the production of a complete proof term (as done by the Omega or Zenon tactics in Coq) can be a slow and difficult process. Thus, the production of small, efficient traces is the cornerstone of the sound integration of an automated prover in an interactive prover. Another, more practical, issue raised by using automated theorem provers is that verification conditions generated by Why are expressed in a polymorphic first-order logic, while existing provers only handle untyped logic (such as Simplify and HaRVey) or monomorphic many-sorted logic (such as Yices, CVC3). It has been shown in [8] that finding encodings between these logics which are correct and do not deteriorate the performance of the provers is not a trivial issue.

Therefore, we have developed the Ergo theorem prover with these different limitations in mind; the main novelties in our system are the native support of polymorphism, a new modular congruence closure algorithm CC(X) for combining the theory of equality over uninterpreted symbols with a theory X, and a mechanism producing lightweight proof traces.

The remainder of this paper focuses on the design of Ergo and attempts of integration in Coq. Section 2 describes the different characteristics and central components in Ergo, whereas Section 3 details both a loose integration of Ergo in Coq, and a tighter integration based on the production of efficient traces for the *congruence closure* module of Ergo.

2. THE ERGO THEOREM PROVER

Ergo is an automatic theorem prover fully integrated in the program verification tool chain developed in our team. It solves goals that are directly written in the Why's annotation language [1]. This means that Ergo fully supports quantifiers and deals directly with polymorphism.

2.1 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 2 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 three 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.
- A module that handles the ground and monomorphic literals assumed by the SAT-solver. It is based on a new combination scheme,CC(X), for the theory of uninterpreted symbols and built-in theories such as linear arithmetic, the theory of lists etc.
- A matching module that builds monomorphic instances of the (possibly polymorphic) lemmas contained in the SAT-solver modulo the equivalence classes generated from the decision procedures.

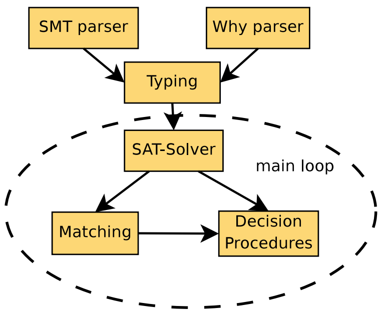

Figure 2: The modular architecture of Ergo.

The rest of this section explains the core decision procedures and how quantifiers are supported by Ergo[2] and their subtle interaction with polymorphism.

[1]Ergo also parses the standard [17] defined by the SMT-lib initiative.
[2]Ergo handles quantifiers in a very similar way to Simplify and Yices.

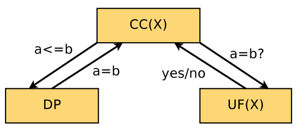

Figure 3: CC(X) architecture

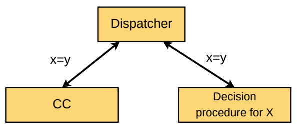

Figure 4: Nelson-Oppen architecture

2.2 Built-In Decision Procedures

The decision procedure module implements CC(X) [5], a new combination scheme *à la Shostak* [19, 18] 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 DP for its relational symbols except for the equality which is handled by a generic union-find algorithm, UF(X), parameterized by X. As shown in Figure 3, the combination relies on the following exchanges:

- CC(X) sends relations between *representatives* in UF(X) to DP. Using representatives automatically propagates the equalities implied by UF(X). In return, DP 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 [15] where, as shown in Figure 4, the combined modules have to discover and propagate all their new equalites.

Currently, CC(X) has been instantiated by linear arithmetic [3], the theory of lists, the general theory of constructors and a restricted theory of accessibility in graphs [7].

2.3 Quantifiers

The SAT-solver module takes as input CNF formulas, seen as sets of disjunctions where leaves are either ground literals or quantified formulas in prenex normal form.

A (standard) propositional SAT engine decides the satisfiability of a propositional formula by assuming (positively or negatively) each leaf of the associated CNF, and then by simplifying the formula accordingly to these choices. It stops whenever the CNF becomes equal to the empty set, proving that the input formula is satisfiable by providing a model. In Ergo, the general mechanism of the SAT-solver is quite similar, but there are two main differences.

The way the leaves are handled depends on their nature: assuming a ground literal amounts to sending it to the decision procedure module, while assuming positively a quantified formula simply means to store it in the current state of the SAT engine.

When the CNF is empty, the Ergo SAT-solver still has to handle the previously stored quantified formulas. In general, it is obviously not possible to decide whether these formulas are consistent

[3]Ergo is complete over rationals but uses heuristics for integers.

with the partial model already built. However, one may try to prove inconsistency by using ground instances of these formulas. The instantiation mechanism is provided by the matching module which builds a new CNF by instantiating the quantified formulas with some ground terms *occurring in the ground literals already handled*. A pattern mechanism (similar to Simplify's triggers) is used to guide quantifier instantiation. Patterns can either be defined by the user or automatically generated.

2.4 Polymorphism à la ML

In Ergo, the matching module also handles the polymorphism by instantiating type variables. Consider for instance the following example written in the Why syntax which 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)
```

First, the typing module checks that this input is well-typed, and when encountering a goal such as

```
goal g: ∀x:α. length(cons(x,nil))=1
```

it turns the term variable x into a constant a (usual transformation) as well as the implicitly universally quantified type variable α into a type constant τ. This implies in particular that the context contains the type constant τ and the term constant a of type τ and that the `goal g':length(cons(a,nil))=1` is monomorphic.

Now, in order to prove the goal `g'`, the matching module generates the instance of `a2` by the substitution

```
{α → τ, x → a, l → nil}
```

We are left to prove that `1+length(nil)=1` where `nil` has the type τ `list`. The only way to show this is by using `a1`.

At first glance, `a1` seems to be a monomorphic ground literal that could be sent to the decision procedure module and not a lemma, since no explicit quantified variable occurs in it. However if it is considered as such, the type of `nil` is fixed to an arbitrary constant which is *distinct* from τ. This prevents using `a1` to conclude that `1+length(nil)=1` holds when `nil` has type τ `list`. The actual lemma should be:

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

but in the Why syntax, the type variables such as α are only implicitly universally quantified. Some of these type variables 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 has to be internally translated by Ergo into `a1'`.

To sum up, there is an invariant in the main loop:

1. a goal is always monomorphic;
2. only monomorphic ground literals are sent by the SAT-solver to the decision procedures' module;
3. the matching module instantiates polymorphic lemmas using the monomorphic ground types and terms already handled, thus the generated instances are monomorphic.

2.5 General Benchmarks

Ergo is written in Ocaml and is very light (∼ 3000 lines of code). It is freely distributed under the Cecill-C licence at `http://ergo.lri.fr`.

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 in the SAT-solver [6].

Since the built-in decision procedures are tightly coupled to the top-level SAT-solver, the backtracking mechanism performed by the SAT module forces the decision procedure module to come back to its previous state. This is efficiently achieved by using functional data structures of Ocaml.

We benchmarked Ergo on a set of 1450 verification conditions that were automatically generated by the VCG Caduceus/Why from 69 C programs [16]. These goals make heavy use of quantifiers, polymorphic symbols and linear arithmetic. All these conditions are proved at least by one prover. Figure 5 shows the results of the comparison between Ergo and four other provers: Z3, Yices, Simplify and CVC3. As mentioned above, none of these provers can directly handle polymorhism; therefore we simply erased types for Simplify and we used an encoding for Yices, Z3 and CVC3. The five provers were run with a fixed timeout of 20s on a machine with Xeon processors (2.13 GHz) and 2 Gb of memory.

	valid	**timeout**	**unknown**	**avg. time**
Simplify $_{\text{v1.5.4}}$	98%	1%	1%	60ms
Yices $_{\text{v1.0}}$	95%	2%	3%	210ms
Ergo $_{\text{v0.7}}$	94%	5%	1%	150ms
Z3 $_{\text{v0.1}}$	87%	10%	3%	690ms
CVC3 $_{\text{v20070307}}$	71%	1%	28%	80ms

Figure 5: Comparison between Ergo, Simplify, Yices and CVC-Lite on 1450 verification conditions.

The column **valid** shows the percentage of the conditions proved valid by the provers. The column **timeout** gives the percentage of timeouts whereas **unknown** shows the amount of problems unsolved due to incompleteness. Finally, the column **avg. time** gives the average time for giving a valid answer.

As shown by the results in Figure 5, the current experimentations are very promising with respect to speed and to the number of goals automatically solved.

3. INTEGRATION OF ERGO IN COQ

Today, Coq still lacks good support of proof automation. There are two main reasons for that. On the one hand, Coq's rich higher-order logics is not well-adapted to decision procedures that were designed for first order logics. On the other hand, Coq is built following the de Bruijn principle: any proof is checked by a small and trusted part of the system. Making a complex decision procedure part of the trusted system would go against this principle.

Still, one would like to use automated provers such as Ergo in Coq. We briefly present two possible approaches to this problem, one by giving up the de Bruijn principle, the other one by maintaining it.

3.1 A loose integration

To be able to use Ergo in Coq, one has to translate goals in Coq higher order logic into first order logic, understood by Ergo. Ayache and Filliâtre have realized such a translation [2]. In their approach, the Coq goal is translated, sent to Ergo, and the answer of the automated prover is simply trusted. Here, by using the automated prover as an *oracle*, the de Bruijn principle is given up, but the resulting Coq tactics are quite fast.

This translation aims the polymorphic first order logic of the Why tool, which is the same logic as the one of Ergo. It not only translates terms and predicates of the Coq logic CIC, the Calculus of Inductive Constructions, but also includes several techniques to go beyond: abstractions of higher-order subterms, case analysis, mutually recursive functions and inductive types.

Using the Why syntax as target language has the advantage of being able to interface any automated prover supported by Why with the above translation. Simplify, Yices, CVC Lite and other provers may be called from Coq. The first order prover Zenon even returns a proof trace in form of a Coq term. The soundness of its answers can thus be checked by Coq.

3.2 A tight integration via traces

The oracle approach above has the disadvantage that it is very easy to introduce bugs in the translation or in the prover, which may compromise a whole proof development in Coq. Constructing a Coq proof term directly is sound, but may generate huge proof traces and is difficult if the problems contain interpreted function symbols of some theory, for example the theory of linear arithmetic (the tool Zenon mentioned above does not handle arithmetic). Another approach consists in modelling part of the prover in Coq and only communicating applications of inference rules or other facts that are relevant for soundness. In particular, any part of the execution that is concerned with proof search can be omitted. The size of the proof is expected to be considerably shorter, and thus the time to check this proof. Proofs that are guided by the execution of the decision procedure are called *traces*. This section describes the ongoing work of constructing proof traces for the core decision procedure CC(X).

As described in section 2.2, the core decision procedure CC(X) of Ergo uses a module UF(X) to handle the equality axioms, ie. reflexivity, symmetry and transitivity, as well as equality modulo the theory X. If the soundness of such a union-find module is established, and we are given a set E of initial equations, a sound concrete union-find structure can be constructed as follows: starting with the empty union-find structure (that only realizes equality modulo X), we only have the right to process (merge) equations that are either in E or are of the form $f(a_1,\cdots,a_n) = f(b_1,\cdots,b_n)$, where the representatives of a_i and b_i are the same, for all i. This translates directly into an inductive type definition in Coq:

```
Inductive Conf (e:list equation) : uf → Set :=
 | init : Conf e Uf.empty
 | in_p : ∀(u:uf) (t1 t2:term),
            Conf e u → In (t1,t2) e →
              Conf e (Uf.union u t1 t2)
 | congr : ∀(u:uf) (l1 l2: list term) (f: symbol),
        Conf e u → list_eq (Uf.equal u) l1 l2 →
        Conf e (Uf.union u (Term f l1) (Term f l2)).
```

The function `list_eq` takes a relation as first argument and returns the relation lifted to lists; otherwise this definition should be self explanatory.

Now, for any object of type `Conf e u`, it is not difficult to prove in Coq that the union-find structure `u` is indeed sound, by proving the following lemma:

```
Theorem correct_cc :
  ∀(u:uf) (e :list equation),
    Conf e u →
     (∀(t1 t2:term), Uf.equal u t1 t2 →
       Th.thEX e t1 t2).
```

where `X.thEX` is the target relation $=_{E,X}$, which means equality modulo the theory X and the set E of assumed ground equations. The Coq proof of theorem `correct_cc` follows the paper proof of soundness of CC(X) given in the appendix of [5] and consists of 150 lines of specification and 300 lines of proof script.

This enables us to construct a proof from a run of Ergo: by recording the processing of equations in the CC(X) module, establish a Coq object of type `Conf e u`, deliver a proof that this union-find module renders t_1 equal to t_2, and by the application of the theorem `correct_cc` we obtain a proof for $t_1 =_{E,X} t_2$. If the union-find module is actually implemented in Coq, we can even obtain the proof of t_1 and t_2 having the same representative automatically, by a technique called *reflection* that employs the calculating capabilities of the proof assistant.

Thus, all that is left is an implementation of a union-find modulo a theory X in Coq (UF_{Coq} in the following), which in turn requires the implementation of the theory X_{Coq}. To be independent of any particular theory, we use the same trick as CC(X) uses: we develop a parameterized module (a *functor*), that may be instantiated by *any* theory that provides the necessary constructs. With the strong type system of Coq, we can even express and require soundness properties of the theory that are necessary to prove the soundness of $\mathsf{UF}_{Coq}(\mathsf{X}_{Coq})$.

A subtlety in the implementation of proof traces is the handling of function symbols and constants. On the one hand, one would like to be as flexible as possible and not fix the set of used function symbols in advance (in general, every problem will use its own set of symbols). On the other hand, it is necessary to reason about some (fixed) function symbols, like $+$ in the case of arithmetic. Our solution to this dilemma is (again) the use of a functor: the theory X_{Coq} is parameterized by a signature S which provides uninterpreted function symbols. Internally, the theory completes this signature with its own function symbols and may then reason about the resulting signature. To obtain the union-find structure in Coq, we can now instantiate UF_{Coq} by $\mathsf{X}_{Coq}(S)$. In practice, the signature S is generated automatically by the proof traces generation mechanism. To summarize, we obtain the following instantiation chain:

$$S \rightarrow \mathsf{X}_{Coq}(S) \rightarrow \mathsf{UF}_{Coq}(\mathsf{X}_{Coq}(S)) \rightarrow \mathsf{CC}_{Coq}$$

To avoid reconstructing CC_{Coq} for identical signatures and theories, the instantiation may be part of a prelude file, as is already the case for type definitions, axioms, etc. in the VCGs generated by the Why tool.

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.

We also described two attempts at integrating Ergo in Coq: first as an oracle, which raises concerns about the soundness of the certification chain, and then we showed how to generate proof traces for the congruence closure algorithm of Ergo so as to let Coq verify the proof itself. Such traces are a very interesting way of mechanizing interactive proving without breaking the chain of trust [1]. Our first experiments with the traces generation are promising: the generic nature of $\mathsf{CC}(\mathsf{X})$ is truly captured and traces are concise. There is room for improvement in terms of efficiency and traces should ideally also cover the SAT-solver and matching components. For the moment, only traces for linear arithmetic are implemented. In total, the Coq development of the theory of linear arithmetic takes about 400 lines of specification and 900 lines of proof script.

Another direction, that we think is worth investigating, is to “prove the prover” in a proof assistant. Indeed, Ergo uses only purely functional data-structures (with the exception of the hash-consing modules), is highly modular and very concise ($\sim$ 3000 lines of code). All these features should make a formal certification feasible.

Also, since this prover is partly dedicated to the resolution of verification conditions generated by the Krakatoa/Caduceus/Why[14] toolkit, its future evolution is partly guided by the needs of these tools : designing efficient proof strategies to manage huge contexts and useless hypotheses and adding more built-in theories such as pointer arithmetic. We also plan to design parsers in order to run Ergo on other benchmarks such as ESC/Java, Boogie and NASA benchmarks.

Finally, we are currently working on a functor `Combine(X1,X2)` to effectively combine different built-in theories `X1` and `X2` under certain restrictions, and by taking advantage of the fact that our theories are typed.

5. REFERENCES

[1] The A3PAT Project. `http://www3.iie.cnam.fr/~urbain/a3pat/`.

[2] N. Ayache and J.-C. Filliâtre. Combining the Coq Proof Assistant with First-Order Decision Procedures. Unpublished, March 2006.

[3] C. Barrett and C. Tinelli. Cvc3. In W. Damm and H. Hermanns, editors, *Proceedings of the 19th International Conference on Computer Aided Verification (CAV’07), Berlin, Germany*, Lecture Notes in Computer Science. Springer, 2007.

[4] S. Conchon and E. Contejean. The Ergo automatic theorem prover. `http://ergo.lri.fr/`.

[5] S. Conchon, E. Contejean, and J. Kanig. CC(X): Efficiently Combining Equality and Solvable Theories without Canonizers. In S. Krstic and A. Oliveras, editors, *SMT 2007: 5th International Workshop on Satisfiability Modulo*, 2007.

[6] S. Conchon and J.-C. Filliâtre. Type-Safe Modular Hash-Consing. In *ACM SIGPLAN Workshop on ML*, Portland, Oregon, Sept. 2006.

[7] S. Conchon and J.-C. Filliâtre. Semi-Persistent Data Structures. Research Report 1474, LRI, Université Paris Sud, September 2007.

[8] J.-F. Couchot and S. Lescuyer. Handling polymorphism in automated deduction. In *21th International Conference on Automated Deduction (CADE-21)*, volume 4603 of *LNCS (LNAI)*, pages 263–278, Bremen, Germany, July 2007.

[9] L. de Moura and N. Bjørner. Z3, An Efficient SMT Solver. `http://research.microsoft.com/projects/z3/`.

[10] L. de Moura and B. Dutertre. Yices: An SMT Solver. `http://yices.csl.sri.com/`.

[11] D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.

[12] J.-C. Filliâtre. *Preuve de programmes impératifs en théorie des types*. PhD thesis, Université Paris-Sud, July 1999.

[13] J.-C. Filliâtre. Verification of non-functional programs using interpretations in type theory. 13(4):709–745, July 2003. [English translation of [12]].

[14] J.-C. Filliâtre and C. Marché. The Why/Krakatoa/Caduceus platform for deductive program verification.

[15] G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Trans. on Programming, Languages and Systems*, 1(2):245–257, Oct. 1979.

[16] ProVal Project. Why Benchmarks. `http://proval.lri.fr/why-benchmarks/`.

[17] S. Ranise and C. Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `http://www.SMT-LIB.org`, 2006.

[18] H. Rueß and N. Shankar. Deconstructing Shostak. In *LICS ’01: Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science*, page 19, Washington, DC, USA, 2001. IEEE Computer Society.

[19] R. E. Shostak. Deciding combinations of theories. *J. ACM*, 31:1–12, 1984.