Logtk : A Logic ToolKit for Automated Reasoning and its Implementation

Simon Cruanes

École polytechnique and Deducteam, Inria,
23 Avenue d'Italie, 75013 Paris, France
simon.cruanes@inria.fr
https://who.rocq.inria.fr/Simon.Cruanes/

Abstract

We describe the design and implementation of Logtk, an OCaml library for writing automated reasoning tools that deal with (possibly typed) first-order logic. The library provides data structures to represent terms, formulas and substitutions, and algorithms to index terms, unify them. It also contains parsers and a few tools to demonstrate its use. It is the basis of a full-fledged superposition prover.

1 Introduction

Writing automated reasoning tools, such as theorem provers, is a difficult engineering task that requires solving many difficult problems in addition to the actual deduction rules. Efficient provers for first-order logic, such as E[13], SPASS[17] or Vampire[11] are usually developed in a low-level language, over many years with great effort. On the other hand it is often useful to be able to prototype an idea in a higher-level language, only requiring decent performance; for instance Saturate[3] was written in Prolog. Our goal with Logtk is to make such prototyping easier by providing solid foundations that most systems need, including typing (and type inference), term representations, formulas, indexing, substitutions, unification algorithms, parsers for standard formats (e.g., TPTP) and various transformations (in particular, reduction to CNF of a set of formulas).

The OCaml language is a representative of the ML family, and as such is well-suited to symbolic manipulations and theorem proving. It was therefore a natural choice for such a library, as a trade-off between safety, expressiveness and performance. Logtk is actively developed and is free software, available at `https://www.rocq.inria.fr/deducteam/Logtk/index.html`.

2 Basic Building Blocks

In this section we will present the fundamentals of an automated reasoning tool: how to represent terms, formulas, substitutions, and how to manipulate them. We target polymorphic first-order logic, as described in [1], because it encompasses the usual untyped logic but brings more safety and expressiveness for many problems involving data structures, arithmetic, set theory, etc. Our library can also be used, in a lesser extent, for higher-order logic, and other term representations are relatively easy to implement from the existing ones.

2.1 Definitions

We use the notation $t : \tau$ to state that a term t is of type τ. In general a *substitution* will be a finite mapping from variables to objects (terms, types...). Types are inductively defined as

(i) atoms ι, o (propositions), `int`, a, b, f, etc. or (ii) variables α_1, α_2, etc. or (iii) application of type constructor $c(\tau_1, \ldots, \tau_n)$ where c is a type constructor and the τ_i are types. A type constructor must be used with the same arity consistently, or (iv) function types of the form $(\tau_1 * \tau_2 * \cdots * \tau_n) > \tau$ where $n \geq 1$, all τ_i and τ are types, or (v) quantified types $\Lambda\alpha_1, \ldots, \alpha_n.\ \tau$ where the α_i are type variables and τ is a type.

Terms are inductively defined by (i) v is a term if $v : \tau \in X$ (an infinite, countable set of typed variables), (ii) defined constants (integers, rationals, propositions, etc.) such as $1 : \text{int}$, $4/3 : \text{rat}$, $\top : o$ and $\bot : o$ (true and false) are terms, (iii) $f\langle\tau_1, \ldots, \tau_k\rangle(t_1, \ldots, t_n)$ $(k, n \in \mathbb{N}^2)$ is a term if $\forall i \in \{1, \ldots, k\}, \tau_i$ is a type, f is a symbol of type $\Lambda\alpha_1, \ldots, \alpha_k.\ (\kappa_1 * \cdots * \kappa_n) > \kappa$ and $\forall i \in \{1 \ldots k\},\ t_i : \kappa_i\sigma$ where $\sigma = \{\alpha_1 \mapsto \tau_1, \ldots, \alpha_k \mapsto \tau_k\}$ is a type substitution. The resulting term is of type $\kappa\sigma$.

For instance, formal verification of a program that uses polymorphic lists might need the symbols $\text{cons} : \Lambda\alpha.\ (\alpha * \text{list}(\alpha)) > \text{list}(\alpha)$ and $\text{nil} : \Lambda\alpha.\ \text{list}(\alpha)$. Note that the polymorphism is explicit, all variables occurring in the type of f must be bound to arguments of f (the $\tau_1, \ldots, \tau_k$ arguments). The list of integers $[1, 2]$ (of type list(int)) would then be represented by the term $\text{cons}\langle\text{int}\rangle(1, \text{cons}\langle\text{int}\rangle(2, \text{nil}\langle\text{int}\rangle))$.

From there we can build formulas from atoms (propositions and equations), conjunction, disjunctions, negation, existential and universal quantification.

2.2 Terms and Types in OCaml

Interactions between terms, types and formulas are non-trivial. For instance, unifying terms also requires unifying their types, and substituting a type variable deep inside a formula should deal with all formula, term and type binders in between. In general, we make a distinction between bound variables, represented as De Bruijn indices, and free variables — allowed to participate in unification, and therefore useful for resolution procedures, type inference, etc. — that have meaningless numbers.

We could represent types, terms, and formulas with different OCaml types, but that leads to some repetitions and duplication for dealing with substitutions, unification and bound variables (especially type variables). Instead, we take a different path and define a single underlying type, named *scoped term*, roughly as shown in Figure 1. More variants, including extensible records[1], are not shown here for the sake of brevity, but are used in other parts of Logtk (for instance records are used in higher-order terms and in the meta-prover).

The type scoped_term can be used to represent many term-like structures, which will then define more specific *views* and constructors that use scoped_term underneath. The type term_kind is a dynamic tag[2] that is used to efficiently discriminate between terms, types, formulas, etc. For instance, a fragment of the Type module, in Figure 2, displays a type-centric view and dedicated constructors. Other types (such as higher-order terms) can be built on top of scoped_term[3] by providing such constructors and views, and adding a variant to term_kind[4]. Also note the field ty, which points to another term representing the type, (or maybe another term for dependently-typed calculi). It is wrapped in an option so that the inductive type is actually well-founded[5].

Let us detail more precisely the code in Figure 2. First, the type Type.t (representing rank-1

[1] Extensible records are an interesting case, because they can appear both in terms and in their types. Since they are useful and make unification relatively subtle, we included them.

[2] Similar tags are very common in dynamic programming languages such as Python.

[3] because the term is responsible for manipulating properly scoped De Bruijn indices.

[4] OCaml's next version will include *open types*, similar to exceptions, that can be extended anywhere. That would be a good fit for our tags.

[5] In TPTP, the pseudo-type `$tType` is used as the top of the type hierarchy.

```
type scoped_term = {
    ty : scoped_term option;
    term : term_cell
    kind : term_kind
}
and term_cell =
    | At of scoped_term * scoped_term
    | App of scoped_term * scoped_term list
    | Var of int
    | BoundVar of int
    | Bind of symbol * scoped_term * scoped_term
and term_kind =
    | FOTerm
    | HOTerm
    | Type
    | Formula
```

Figure 1: Declaration of `scoped_term`

types) is defined as a *private alias* of scoped_term, which means every Type.t can be safely coerced into the generic representation (e.g. for substitutions, unifications, etc.) but not conversely; down-casting must be done with the function Type.of_term that checks the dynamic tag. The type Type.view is used for pattern-matching against types, using the eponymous function. Finally, some constructors that always return valid types (without down-casting) are then defined.

Unification, substitutions, equality, hashconsing, handling of De Bruijn indices are all defined only once to operate on scoped_terms. It is also easier to mix term and type arguments, to quantify over types in formula-level binders, etc. because the underlying common structure will ensure that substitutions and unification remain correct.

FOTerm is the module of (typed) first-order terms. All constructors for leaf terms require a type argument (variables and constants are typed); other constructors just check the types of their arguments and deduce the type of their result. Every term is annotated with its type; the reason is that unifying terms also requires unifying their types, which must be easy to obtain. As is done for the Type module, FOTerm provides a *view* of terms into the following variant:

Var: free variable, whose name is an integer;
BVar: bound variable (De Bruijn index);
TyApp: apply a term to a type (for instance nil(int));
Const: constant term, parametrized by a symbol (and its type);
App: apply a term to a list of other terms. The first term should be composed only of TyApp and Const so that the term remains in the first-order fragment.

2.3 Substitutions

We distinguish here *substitutions*, that is, say mapping from free variables to terms (or types), from *environments* that are used in conjunction with bound variables and the De Bruijn indexing system. Let us examine substitutions more closely. In many cases (rewriting, resolution...), unification works on free variables, but often requires renaming:

- for term rewriting, a subterm $t|_p$ is matched against the left-hand side of a rule $l \to r$ so is it necessary for t and l not to share any variable;

```
module Type : sig
    type t = private scoped_term

    type view = private
      | Var of int                (* Type variable *)
      | BVar of int               (* Bound variable (De Bruijn) *)
      | App of symbol * t list    (* parametrized type *)
      | Fun of t list * t         (* Function type (arg list -> ret) *)
      | Forall of t               (* explicit quantification *)

    val view : t -> view          (* open the type's root *)
    val of_term : scoped_term -> t option (* dynamic cast *)

    val var : int -> t
    val app : symbol -> t list -> t
    val const : symbol -> t
    val arrow : t -> t -> t
    val forall : t list -> t -> t
end

module FOTerm : sig
    type t = private scoped_term

    type view = private
      | Var of int            (* Term variable *)
      | BVar of int           (* Bound variable (De Bruijn) *)
      | Const of Symbol.t     (* Typed constant *)
      | TyApp of t * Type.t   (* Type parameter *)
      | App of t  * t list    (* List of parameters *)

    val view : t -> view
    val of_term : scoped_term -> t option

    val var : ty:Type.t -> int -> t
    val bvar : ty:Type.t -> int -> t
    val const : ty:Type.t -> symbol -> t
    val tyapp : t -> Type.t -> t
    val app : t -> t list -> t
end
```

Figure 2: View and Constructor for Types and FOTerms

- for resolution (or superposition), binary inferences such as
  $\frac{C \vee l_1 \qquad C' \vee \neg l_2}{(C \vee C')\sigma}$ if $l_1\sigma = l_2\sigma$
  will require the two clauses to share no variable prior to unification.

To avoid renaming, which can be costly, some techniques have been used by provers such as SPASS[17] or Otter[7], involving so-called *variable banks*. Assuming variables are indexed by natural numbers, a variable bank is an array that maps each index $0 \leq i <$ MAXVAR (where MAXVAR is a bound on the total number of distinct variables) to either: (i) itself (variable not bound), or (ii) to a tuple `(term, varbank)` where `varbank` provides bindings to free variables of `term`, that can be recursively look up the same way. Variable banks can therefore point to one another in a cyclic way, for instance after unifying the terms $f(X, g(Z))$ and $f(g(Y), Y)$ where X and Z live in one bank and Y in another one. This technique works fine but suffers

from two limitations:

- it requires substitutions to be mutable arrays (rather than functional-friendly immutable structure that can safely be kept for generating proofs, or stored in data structures);
- it requires allocating big arrays (as big as the maximal authorized variable index), which limits the number of substitutions that are allowed to live simultaneously.

To overcome those limitations we use a persistent representation and a notion of *scope*, inspired from the code[6] of iProver[5].

A scope is a value that represents one interpretation for free variables, which means that the same variable can have distinct bindings in distinct scopes. In our implementation a scope is simply an integer. Substitutions and unification therefore map pairs (variable,scope) to pairs (term,scope), rather than directly variable to term. A substitution is a finite mapping from pairs to pairs (currently a persistent hash table, but balanced trees or mere linked lists could do too). Figure 3 shows the type signatures of some operations on substitutions[7]. Note that if one does not wish to rename variables (e.g. for type inference), one can use only one scope and essentially fall back to the usual representation of substitutions. We write $[\![t]\!]_i$ for the term t interpreted in the scope i, and trivially extend the notation to literals and clauses.

When a substitution σ has been computed by unification or matching (see Section 3.1), for instance after a resolution step between two clauses $[\![C \vee l_1]\!]_0$ and $[\![C' \vee \neg l_2]\!]_1$, we need to *apply* it to build a new clause $(C \vee C')\sigma$. Here we need be careful because, in $C \vee C'$, some variables are bound in scope 0, some other in scope 1; we need to evaluate $([\![C]\!]_0 \vee [\![C']\!]_1)\sigma$ instead. Now the question is: how shall we deal with free variables that are not bound in the substitution?

For instance, say we have a substitution $\sigma = \{[\![X]\!]_0 \mapsto [\![f(X)]\!]_1, [\![X]\!]_1 \mapsto [\![Y]\!]_1\}$ (remember that $[\![X]\!]_0$ and $[\![X]\!]_1$ are distinct variables because they live in different scopes). To evaluate the clause $[\![p(X,Y)]\!]_0\sigma \vee [\![p(X,Y)]\!]_1\sigma$ we must rename one of $[\![Y]\!]_0$ and $[\![Y]\!]_1$ because *they are distinct variables*. To do so, applying a substitution requires an object called *renaming*, that builds an injection from (variable, scope) to variable; the result, as expected, will be alpha-equivalent to $p(f(X),Y) \vee p(X,X)$ (renaming $[\![Y]\!]_1$ to X, and $[\![Y]\!]_0$ to Y).

```
type scope = int
type subst = (scoped_term * scope * scoped_term * scope) list
type renaming = (variable * scope) -> variable

val unify : scoped_term -> scope -> scoped_term -> scope -> subst option
val rename : renaming -> variable -> scope -> variable
val apply : renaming -> subst -> scoped_term -> scope -> scoped_term
```

Figure 3: Operations on Substitutions

3 Algorithms

Many algorithms are very often useful for processing logic formulas. Some of the particularly useful ones are implemented in Logtk.

[6] it is, to our knowledge, the first occurrence of this technique.
[7] The type renaming is abstracted into a function for clarity.

3.1 Unification and Matching

The usual first-order unification and matching algorithms are implemented only once, on the scoped_term shared structure. Their type signature is shown in Figure 3. The algorithm can be used with any view of scoped_term, including FOTerm.t and Type.t. We need to recursively unify subterms pairwise, but also types. Indeed, term-level variables can have polymorphic types, as is shown in the few clauses of Figure 4 that encode polymorphic lists. Note the presence of Skolem symbols `head` and `tail` in the inversion axiom, that encode the fact that any non-nil list is necessarily an application of `cons`. Which such axioms, we may need to unify both terms and types (the type variable α) when working with concrete lists such as $\text{cons}\langle\text{int}\rangle(1, \text{nil}\langle\text{int}\rangle)$; if some variables are *unshielded* (i.e. they appear under some equation, but under no function symbol) then unifying types becomes crucial for soundness, for otherwise we would try to unify boolean variables with list-sorted terms.

$$\forall x : \alpha,\ l : \text{list}(\alpha), \text{cons}\langle\alpha\rangle(x, l) \neq \text{nil}\langle\alpha\rangle \qquad \text{(injection)}$$

$$\forall x : \text{list}(\alpha),\ x = \text{nil}\langle\alpha\rangle \vee x = \text{cons}\langle\alpha\rangle(\text{head}\langle\alpha\rangle(x), \text{tail}\langle\alpha\rangle(x)) \qquad \text{(inversion)}$$

Figure 4: A Polymorphic Theory with Unshielded Variables

3.2 Reduction to Clausal Normal Form

It is often practical to transform a given problem into a clausal form (CNF). Resolution provers, for instance, require it. However, in many cases they prefer to rely on an external prover (for instance SPASS[17]). Here, we can't do that, first because Logtk is intended to be self-contained, and because our terms may be more general, for they are typed and may contain additional constructs such as records or curried application. Naive CNF is quite easy to implement; however, many real problems cause naive CNF to blow up because of the exponential number of clauses; many others will yield suboptimal skolemization. Therefore, we implemented CNF reduction with miniscoping and formula renaming[8], following[9]. This is enough to avoid the exponential blowup.

3.3 Indexing

Saturation provers rely heavily on unification. When the clause set grows, *term indices* become necessarily to keep a good inference rate. In Logtk we define several such indices for first-order (typed) terms, parametrized by the data stored at the leaves of the index. Conceptually, a term index maps each term to a set of values of some type (for instance, a pair (clause, position) can be used for superposition provers), and allows to retrieve values by unification or matching with a query term. We provide several indexing schemes for theorem provers, rewriting systems, etc.

- *fingerprint indexing*[15] as a general purpose index;
- *feature vector indexing*[14] for subsumption checking;
- *perfect discrimination trees*[10] for rewriting, and *non-perfect discrimination trees* as a general purpose index.

[8] although the criterion for triggering the renaming of a formula is simpler than the optimal one presented in [9].

The index implementations are all purely functional, which is facilitated by their tree-like structure (most often a prefix tree). This can be useful in contexts where duplicating an index might be necessary, for instance in Tableaux provers or for other splitting-like inference rules.

Let us focus on the implementation of the discrimination trees. The classic way to implement them is based on the use of *flatterms*, i.e., terms represented as a flat array of symbols (including $*$ that represents variables in their imperfect variant, or regular variables in the perfect indexing version). However this representation isn't convenient for many other operations, and it is incompatible with any kind of subterm sharing.

Conversion between tree-like terms and flatterms can be very costly. A pathologic example would be, in the context of term rewriting, the application of the rule $s(X) + Y \mapsto s(X + Y)$ that describes the addition in Peano arithmetic to the term $\overline{500000} + \overline{500000}$ (where $\overline{n}$ is the encoding of $n \in \mathbb{N}$ into a Peano term $s^n(0)$). We would build a flatterm of size 1000000 only to see it matched against a small rule, then the term $s(\overline{499999} + \overline{500000})$ would be converted to a flatterm, matched, and so on. This series of conversions would be very expensive.

Our solution here is to perform a *lazy* conversion to flatterms, by using a specialized iterator type that provides the required `next` and `skip` operations. The type of the iterator is shown in Figure 5 and is discussed further. At any point in the traversal of a term (we traverse the term and the corresponding branches of the discrimination tree) we remember its siblings and the siblings of its superterms. When the term has been fully traversed, calling next or skip will return None. This iterator type is persistent, which makes backtracking (exploring several branches of a discrimination tree) trivial.

Let us explain the code in Figure 5. The function open_term is used to flatten its term argument's root (given a stack of parent terms and their siblings) into a new iterator; flatten starts the flattening of a whole term (meaning the surrounding stack is empty). The function next and skip both use the stack; the only difference is that the latter ignores the current term's siblings (if any).

4 Applications

4.1 Use Case: the Theorem Prover Zipperposition

Logtk is currently used to implement the experimental theorem prover Zipperposition. It is based on the superposition calculus and has been modified to experiment on arithmetic, polymorphism, and other extensions. Zipperposition uses many components of Logtk, including the typing system, type inference, TPTP parser, term indexes, unification algorithms, subterm positions, reduction to CNF, etc. One benefit is that would first-order terms be extended with new variants (records, algebraic variants, curried application...), few changes would be required at all in Zipperposition to support the extension.

4.2 Tools

We believe Logtk is well-suited for writing tools that process logic objects. Several such tools are provided in the library, both for their usefulness and as examples of how to use it. Let us describe those tools:

proof_check_tstp calls external provers to check traces a theorem prover can print upon success. For instance if E[13] proves a theorem, it can print the DAG of the inferences it had to perform. `proof_check_tstp` can then parse this DAG (in the TSTP[16] format),

Listing 1: Interface

```
type iterator

val skip : iterator -> iterator option
val next : iterator -> iterator option
val flatten : FOTerm.t -> iterator
```

Listing 2: Implementation

```
module T = FOTerm

type iterator = {
    cur_term : FOTerm.t;  (* current sub-term *)
    stack : FOTerm.t list list;
}

let open_term stack t = match T.view t with
  | T.Var _
  | T.BVar _
  | T.TyApp _
  | T.Const _ ->
      Some {cur_term=t; stack=[]::stack;}
  | T.App (_, l) ->
      Some {cur_term=t; stack=l::stack;}

let rec next_rec stack = match stack with
  | [] -> None
  | []::stack' -> next_rec stack'
  | (t::next')::stack' ->
      open_term (next'::stack') t

let skip iter = match iter.stack with
  | [] -> None
  | _::stack' -> next_rec stack'

let next iter = next_rec iter.stack
let flatten t = open_term [] t
```

Figure 5: Lazy Conversion to Flatterms

and check the validity of every deductive inference by calling one or more trusted provers. Steps that only preserve satisfiability, such as skolemization, are not checked.

cnf_of_tptp parses TPTP files, infers types, and prints the clausal normal form (CNF) of the parsed formulas.

type_check_tptp : a simple type-checker for TFF0 and TFF1 problems, including some type inference for wildcards `$_`.

detect_theories can use the implementation of a *meta prover*[2] to detect instances of axiomatic theories in a problem. For instance it will detect the presence of an abelian group in `RNG008-4.p` (a ring theory problem).

orient : reads a term rewriting system from a file, and looks for a LPO precedence that orients all rules left-to-right (thus proving the termination of the system in this case. Our tool can then print the witness precedence if required). The part that attempts to orient rewrite

rules using a LPO is one of the modules provided in Logtk.

hysteresis is a more sophisticated tool that currently serves as a pre-processor for E. It detects theories using the aforementioned meta-prover, collects associated rewrite systems (if any), attempts to orient them (see previous tool) using a LPO and sends the modified problem to E.

5 Discussion

Many provers ship with some internal library that is designed to cope with the same problems as Logtk, for instance E[13] comes with CLIB, Prover9[6] with LADR, some other the Dedam[8] system, etc. However, there are several significant differences with most of those libraries, and ours.

First, Logtk is written in OCaml. While the choice of a programming language is important for such a performance-sensitive area as Automated Theorem Proving, we made this trade-off to make prototyping much faster than in all the aforementioned C libraries. OCaml, as a dialect of ML, has a long record track of usage for symbolic reasoning, including the implementation of Coq[4]. We clearly cannot hope to beat optimized C in terms of performance, but our goal with Logtk is to make prototyping and writing decent theorem provers much easier. Similarly, abstractions like iterators (on subterms, subformulas, the types in a term, etc.) are pervasively used and exposed to make the code simpler and avoid repeating the same recursive functions everywhere. This kind of abstraction again brings more expressiveness to the user (and implementer of the library)[9]. Stronger typing (absence of `NULL`, polymorphism, modules) and the presence of recursive algebraic types and pattern-matching also improve readability and safety. For instance the formula representation is an algebraic type with 14 cases; checking the exhaustivity of pattern-matching helps ensuring every case is dealt with.

Providing functional structures for types such as substitutions, term indices, and signatures is also a significant difference. More allocations are needed (although OCaml's GC is very good at allocating short-lived structures) but reasoning about one's program becomes easier; again, less time spent debugging improves the programmer's productivity.

The library comes with small tools that illustrate the use of some of its core features – type-checking, reduction to CNF, ...– but is separated from Zipperposition. We deliberately kept the superposition-specific structures outside of Logtk (in particular, the representation of clauses which is very specific) so as not to constrain users to follow the same design choices. It is possible, however, that some structures we use in Zipperposition for linear arithmetic migrate back to Logtk (e.g. linear expressions)[10].

Since Logtk is still very young, we can't evaluate yet how easy (or difficult) it is for someone to use it without any assistance for the authors. Good documentation and openness to contributions will be necessary to make it as easy as possible. The choice of the very permissive BSD2 license should make Logtk easy to contribute to and use.

[9] The performance impact is hard to evaluate but shouldn't be high, especially outside of critical paths.

[10] Some changes needed for Zipperposition have been made, when useful in general. For instance, multisets in which elements can have very large multiplicities are often useful for linear arithmetic, when $n \cdot t$ actually means $\sum_{i=1}^{n} t$, a sum of n elements.

Conclusion

We presented Logtk, a generic OCaml library to represent, process and reason with polymorphic, first-order logic formulas and terms. Several classic algorithms such as unification, reduction to clausal form, term indexing, etc. in addition to parsers and command-line tools are also provided. The library also features some insights about its implementation, especially regarding the handling of bound and free variables and the term representation. We only presented a part of the library, but other modules such as term orderings (LPO, KBO) and term rewriting are also provided.

The code, released under a permissive free license, is usable for experimental automated theorem proving, as demonstrated by our tool Zipperposition[11]. We believe OCaml occupies a sweet spot regarding the trade-off between efficiency and expressiveness in the area of symbolic computing and hope this work will be useful to other practitioners.

References

[1] Blanchette, Jasmin Christian and Paskevich, Andrei. TFF1: The TPTP typed first-order form with rank-1 polymorphism. In *Automated Deduction–CADE-24*, pages 414–420. Springer, 2013.

[2] Guillaume Burel and Simon Cruanes. Detection of First Order Axiomatic Theories. In Fontaine, Pascal and Ringeissen, Christophe and Schmidt, RenateA., editor, *Frontiers of Combining Systems*, volume 8152 of *Lecture Notes in Computer Science*, pages 229–244. Springer Berlin Heidelberg, 2013.

[3] Ganzinger, Harald and Nieuwenhuis, Robert and Nivela, Pilar. The saturate system. 1998.

[4] Gérard Huet, Gilles Kahn, and Christine Paulin-Mohring. The coq proof assistant. *A Tutorial. Version*, 5, 1997.

[5] Konstantin Korovin. iProver — An Instantiation-Based Theorem Prover for First-Order Logic (System Description). In *Proceedings of the 4th international joint conference on Automated Reasoning*, IJCAR '08, pages 292–298, Berlin, Heidelberg, 2008. Springer-Verlag.

[6] W. McCune. Prover9 and mace4. `http://www.cs.unm.edu/~mccune/prover9/`, 2005–2010.

[7] William W. McCune. OTTER 3.0 Reference Manual and Guide, 1995.

[8] Robert Nieuwenhuis, Jos Rivero, and Miguel Vallejo. Dedam: A kernel of data structures and algorithms for automated deduction with equality clauses. In William McCune, editor, *Automated DeductionCADE-14*, volume 1249 of *Lecture Notes in Computer Science*, pages 49–52. Springer, 1997. 10.1007/3-540-63104-6_5.

[9] Andreas Nonnengart and Christoph Weidenbach. Computing small clause normal forms. In Robinson and Voronkov [12], pages 335–367.

[10] I. V. Ramakrishnan, R. C. Sekar, and Andrei Voronkov. Term indexing. In Robinson and Voronkov [12], pages 1853–1964.

[11] Alexandre Riazanov and Andrei Voronkov. Vampire 1.1 (system description). In *Proceedings of the First International Joint Conference on Automated Reasoning*, IJCAR '01, pages 376–380, London, UK, UK, 2001. Springer-Verlag.

[12] John Alan Robinson and Andrei Voronkov, editors. *Handbook of Automated Reasoning (in 2 volumes)*. Elsevier and MIT Press, 2001.

[13] Stephan Schulz. E - a brainiac theorem prover. *AI Commun.*, 15(2,3):111–126, August 2002.

[14] Stephan Schulz. Simple and Efficient Clause Subsumption with Feature Vector Indexing. In *Proc. of the IJCAR-2004 Workshop on Empirically Successful First-Order Theorem Proving*. Elsevier Science, 2004.

[11] Zipperposition, while still a young prover, had decent performance at CASC 2013.

[15] Schulz, Stephan. Fingerprint indexing for paramodulation and rewriting. In *Automated Reasoning*, pages 477–483. Springer, 2012.

[16] G. Sutcliffe. The TPTP Problem Library and Associated Infrastructure: The FOF and CNF Parts, v3.5.0. *Journal of Automated Reasoning*, 43(4):337–362, 2009.

[17] Weidenbach, Christoph and Schmidt, Renate and Hillenbrand, Thomas and Rusev, Rostislav and Topic, Dalibor. System Description: SPASS Version 3.0. In Frank Pfenning, editor, *Automated Deduction CADE-21*, volume 4603 of *Lecture Notes in Computer Science*, pages 514–520. Springer, 2007.