Logtk: A Logic ToolKit for Automated Reasoning and its Implementation

Simon Cruanes

École polytechnique and INRIA, 23 Avenue d’Italie, 75013 Paris, France
https://who.rocq.inria.fr/Simon.Cruanes/

July 23rd, 2014
1. Overview

2. Basics: Terms, Types and Substitutions

3. Algorithms

4. Applications and Discussion
Automated Theorem Proving is hard.

- Find a calculus
- Theory: need to prove correctness and (semi)-completeness
- Implementation: requires a lot of work
- Works in theory $\not\Rightarrow$ works in practice
- Efficiency and correctness concerns
Resolution-based Theorem Proving, in Theory

Example: first-order (typed) resolution & co (Superposition).

Inferences

\[
\frac{C_1 \lor l_1 \quad C_2 \lor \neg l_2}{(C_1 \lor C_2)\sigma} \quad \text{Resolution}
\]

\[
\frac{C \lor l_1 \lor l_2}{(C \lor l_1)\sigma} \quad \text{Factoring}
\]

With \(l_1\) and \(l_2\) literals, \(C, C_1, C_2\) clauses, and \(l_1\sigma = l_2\sigma\) (mgu)

Sound and complete! We’re done!
Easy to state, not to implement.
  - Many refinements necessary for efficiency
  - Subsumption, Equality (Superposition), Rewriting...
  - Real provers: up to 200,000 LOC of C

Real provers hard to modify

Prototyping: still a lot of work (several kLOCs)

Hence, Logtk.
Summary

1. Overview

2. Basics: Terms, Types and Substitutions

3. Algorithms

4. Applications and Discussion
Basic Design Choices

In a nutshell, our goals:

- OCaml $\rightarrow$ high expressiveness and decent performance
- Typed logic
- Proper handling of free and bound variables
- Many types and algorithms
- Free software (permissive BSD license)
- Decent overall performance (won’t beat C)
Polymorphic First-Order

Example: polymorphic lists

\[ \land \alpha. \forall l : list(\alpha). (l = \text{nil}(\alpha) \lor (\exists x : \alpha, l' : list(\alpha). l = \text{cons}(\alpha)(x, l'))) \]

\[ \land \alpha. \forall x : \alpha, l : list(\alpha). \text{nil}(\alpha) \neq \text{cons}(\alpha)(x, l) \]

Why?

- Typed logic increases expressiveness
  - More complex models (one domain per type)
  - Example: \( \forall x : \text{bool}. (x = \text{true} \lor x = \text{false}) \)
  - Bounded quantification
- Many real problems are typed (program verification, arith, etc.)

Still, few provers support typed logic.
Lot of duplicated code (bound variables, hashconsing, substitutions, etc.)

$\Rightarrow$ Use a common representation, named `scoped_term`.

```haskell
type scoped_term = {
  ty : scoped_term option;
  term : term_cell;
  kind : term_kind;
}

and term_cell =
  | Const of symbol
  | 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 =
  | Term
  | Type
  | Formula
```
Term representation, cont’d

Terms, types, formulas: views of scoped_term

module Type : sig  
  type t = private scoped_term

  type view = private
  | Var of int (* free var *)
  | BVar of int (* bound var *)
  | App of symbol * t list
  | Fun of t list * t
  | Forall of t

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

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
Use *private aliases*

- Type.t subtype of scoped_term
- upcast always possible
- downcast requires Type.of_term : scoped_term → Type.t option

*smart constructors* enforce invariants

specific operations (printing, etc.)

unification, substitution, etc.: trivial

Terms, Formulas: idem.
Resolution requires premises not to share variables
⇒ renaming of free variables
⇒ However, renaming is error-prone and expensive.

\[ J_x^0 \neq J_x^1 \]

substitutions bind scoped variables to scoped terms
resolve collisions when substitution is applied.

Note: can bind both term and type variables.

\(^1\)similar to what iProver does.
Resolution requires premises not to share variables
⇒ renaming of free variables
⇒ However, renaming is error-prone and expensive.
Solution: use a notion of scope\textsuperscript{1}.

**Scope**

- $[x]_0$: variable $x$ in scope 0
- $[x]_1$: variable $x$ in scope 1
- $[x]_0 \neq [x]_1$
- substitutions bind scoped variables to scoped terms
- resolve collisions when substitution is applied.

Note: can bind both term and type variables.

\textsuperscript{1} similar to what iProver does.
Substitutions: example

Example

1. Assume $\sigma = \{[x]_0 \mapsto [f(x)]_1, [x]_1 \mapsto [g(y)]_1\}$
2. Goal: evaluate clause $[p(x,y)]_0 \sigma \lor [q(x,y)]_1 \sigma$
3. Use a renaming, an injection (variable, scope) $\rightarrow$ variable
4. For instance $[y]_1 \mapsto u, [y]_0 \mapsto v$
5. Solution: $p(f(g(u)), v) \lor q(g(u), u)$

Useful e.g. for resolution.
In practice

type scope = int

type subst = (variable * scope * term * scope) list

type renaming = ((variable * scope), variable) Hashtbl.t

val unify : term -> scope -> term -> scope -> subst option

val rename : renaming -> variable -> scope -> variable

val apply : renaming -> subst -> term -> scope -> term

(with term = scoped_term)

- Efficient and flexible
- Escape hatch when renaming not necessary (term rewriting)
Summary

1. Overview

2. Basics: Terms, Types and Substitutions

3. Algorithms

4. Applications and Discussion
Many fundamental operations needed for a prover
- CNF, unification, indexing, type-checking, parsing, etc.
- Not always easy to write.

In Logtk: all the above.

⇒ this section: highlight a few algorithms
Classic (naive) Robinson unification
- performs well in practice
- \textit{n-ary} versions using \textit{iterators}
- \textit{n-ary} also useful for subsumption, etc.

- works on \texttt{scoped\_term}
- also unifies types

Also, alpha-equivalence and matching.
**Term Indexing**

**idea:** multimap from terms to elements\(^2\), indexed by unifiability

**Simplified Signature**

\[
\begin{align*}
type & \quad element \\
type & \quad index \\
val & \quad add : \ index \rightarrow \ term \rightarrow \ element \rightarrow \ index \\
val & \quad unify : \ index \rightarrow \ scope \rightarrow \ term \rightarrow \ scope \rightarrow \\
& \quad (\ term \ast \ element \ast \ Subst.t ) \ iterator
\end{align*}
\]

`unify idx 1 t 0` returns an iterator over tuples \((t', v, \sigma)\)
where \([t]_0\sigma = [t']_1\sigma\) and `add idx t' v` was called before

\(^2\text{often pairs of (clause, position).}\)
In Logtk, index = functor over the element type.

Non-perfect Discrimination Trees (default implementation)
- Roughly, a prefix tree over "flat" terms (prefix traversal)
- Variables replaced by "*
- Unification performed at leaves
- Implementation: "lazy" flattening of terms (iterator)
  (flattening can be costly)

Fingerprint Trees

Feature Vector Indexing (for subsumption)

Perfect Discrimination Trees for rewriting
Picture >> Words: a Discrimination Tree

Diagram of a discrimination tree with labeled nodes and branches.
And also...

- Reduction to Clausal Normal Form (CNF) (with formula renaming)
- Type inference and checking
- TPTP parser
- Term orderings (LPO, KBO), precedences over symbols
- Term rewriting
Summary

1. Overview

2. Basics: Terms, Types and Substitutions

3. Algorithms

4. Applications and Discussion
**Tools**

Logtk ships with small tools:

- `cnf_of_tptp`: parses a TPTP file, prints its CNF
- `type_check_tptp`: parses a TPTP (TFF) file, infers and prints types
- `proof_check_tptp`: parses a TPTP derivation, checks every step using external provers

→ Provide small examples of how to use the library.
Our experimental theorem prover, **Zipperposition**, is based on **Logtk**.

- **Logtk** actually forked from Zipperposition
- Logtk provides most data structures and types
- What remains prover-side:
  - literal and clause types (too specific)
  - Inference rules
  - Saturation algorithm and main loop
  - Proof objects
- Types: handled by Logtk
- High-level design allows to modify the code easily.
  - in particular, for *new inference rules* (arithmetic... )
Today

- **Logtk**: library for typed first-order logic
- **Ocaml**: expressiveness and safety
- High-level design: iterators, functors, views
- Used in a non-trivial prover (Zipperposition)
- Free software! Use it, contributions are welcome.

Tomorrow (or later)

- Extensions of Terms (HO...)
- More term index algorithms
- Use in more projects
- ...
Thank you for your attention!
Questions?³

³other than this one.