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

Summary

State your Intent!

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

$$\frac{C_1 \vee l_1 \qquad C_2 \vee \neg l_2}{(C_1 \vee C_2)\sigma} \text{ Resolution}$$

$$\frac{C \vee l_1 \vee l_2}{(C \vee l_1)\sigma} \text{ Factoring}$$

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!

. . . And in Practice

- 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

Basic Design Choices

In a nutshell, our goals:

- OCaml → 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

$$\Lambda\alpha.\forall l : \mathit{list}(\alpha).(l = \mathit{nil}\langle\alpha\rangle \vee (\exists x : \alpha, l' : \mathit{list}(\alpha).l = \mathit{cons}\langle\alpha\rangle(x, l')))$$
$$\Lambda\alpha.\forall x : \alpha, l : \mathit{list}(\alpha).\mathit{nil}\langle\alpha\rangle \neq \mathit{cons}\langle\alpha\rangle(x, l)$$

Why?

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

Still, few provers support typed logic.

Representing Terms, Types and Formulas

Lot of duplicated code (bound variables, hashconsing, substitutions, etc.)

⇒ Use a common representation, named scoped_term.

```
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
```

Term representation, cont'd

- Use *private aliases*
  - Type.t subtype of scoped_term
  - upcast always possible
  - downcase requires Type.of_term : scoped_term −> Type.t **option**
- *smart constructors* enforce invariants
- specific operations (printing, etc.)
- unification, substitution, etc.: trivial

Terms, Formulas: idem.

Substitutions

Resolution requires premises not to share variables
$\Rightarrow$ renaming of free variables
$\Rightarrow$ However, renaming is error-prone and expensive.

[1]similar to what iProver does.

Substitutions

Resolution requires premises not to share variables
$\Rightarrow$ renaming of free variables
$\Rightarrow$ However, renaming is error-prone and expensive.
Solution: use a notion of scope[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.

[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 $[\![p(x,y)]\!]_0\sigma \vee [\![q(x,y)]\!]_1\sigma$
3. Use a renaming, an injection $(\text{variable}, \text{scope}) \rightarrow \text{variable}$
4. For instance $[\![y]\!]_1 \mapsto u, [\![y]\!]_0 \mapsto v$
5. Solution: $p(f(g(u)), v) \vee q(g(u), u)$

Useful e.g. for resolution.

Substitutions: Implementation

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

Algorithms

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

In LOGTK: all the above.

$\Rightarrow$ this section: highlight a few algorithms

Unification

- Classic (naive) Robinson unification
  - performs well in practice
  - *n*-ary versions using iterators
  - *n*-ary also useful for subsumption, etc.
- works on `scoped_term`
- also unifies types

Also, alpha-equivalence and matching.

Term Indexing

idea: multimap from terms to elements[2], indexed by unifiability

Simplified Signature

```
type element
type index
val add : index -> term -> element -> index
val unify : index -> scope -> term -> scope ->
            (term * element * Subst.t) iterator
```

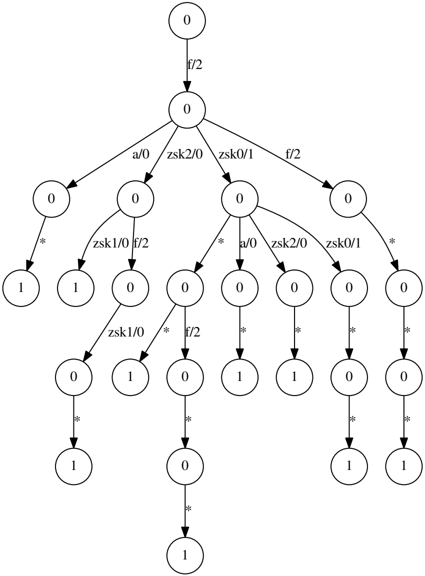
`unify idx 1 t 0` returns an iterator over tuples (t', v, σ) where $[\![t]\!]_0\sigma = [\![t']\!]_1\sigma$ and `add idx t' v` was called before

[2] often pairs of (clause, position).

Term Indexing (cont'd)

- 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



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

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.

Zipperposition

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.
  $\rightarrow$ in particular, for new inference rules (arithmetic. . .)

Conclusion

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]

[3] other than this one.