Kanren Light

A Dynamically Semi-Certified Interactive Logic Programming System

ANONYMOUS AUTHOR(S)

We present an experimental system strongly inspired by miniKanren, implemented on top of the tactics mechanism of the HOL Light theorem prover. Our tool is at the same time a mechanism for enabling the logic programming style for reasoning and computing in a theorem prover, and a framework for writing logic programs that produce solutions endowed with a formal proof of correctness.

ACM Reference Format:
Anonymous Author(s). 2020. Kanren Light: A Dynamically Semi-Certified Interactive Logic Programming System. In *Proceedings of 25th ACM SIGPLAN International Conference on Functional Programming (miniKanren 2020).* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

In straightforward terms, the computation of a logic program evolves by refining a substitution seeking for solutions of a unification problem. This has been made explicit in the Kanren approach [Byrd 2009; Friedman et al. 2005; Hemann and Friedman 2013], where programs are described by composing (higher-order) operators that act on streams of substitutions. Such a methodology allows for a streamlined approach to logic programming; however, the intended semantics and the correctness of a Kanren program rest entirely on the meta-theoretic level.

We propose a framework that extends the Kanren approach to a system that computes both candidate substitutions and corresponding certificates of correctness with respect to a given specification. Such certificates will be formally verified logical truths synthesized using a theorem prover.

Our setup is based on the HOL Light theorem prover [Harrison 1996], in which we extend the currently available tactic mechanism with three basic features: (i) the explicit use of meta-variables, (ii) the ability to backtrack during the proof search, (iii) a layer of tools and facilities for interfacing with the underlying proof mechanism.

The basic building block of our framework are ML procedures that we call *solvers*, which are a generalization of HOL tactics and are –as well as tactics– meant to be used compositionally in order to define arbitrarily complex proof search strategies.

We say that our approach is *semi-certified* because

- on the one hand, the synthesized solutions are formally proved theorems, hence their validity is guaranteed by construction;
- on the other hand, the completeness of the search procedure cannot be enforced in our framework and consequently has to be ensured by a meta-reasoning.

miniKanren 2020, August 23–28, 2020, Jersey City, New Jersey

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Moreover, we say that our system is *dynamically* semi-certified, because the proof certificate is built at run-time.

At the present stage, our implementation is intended to be a testbed for experiments and further investigation on this reasoning paradigm. Section 6 gives some further information on our code.

2 A WORD ABOUT THE HOL LIGHT THEOREM PROVER

In the HOL system, there are two fundamental datatypes called *term* and *theorem*. Terms model fragments of (well-formed) mathematical expressions. Theorems are Boolean terms that are proved correct according to a fixed set of logical rules. Examples of both a term and a theorem in the concrete syntax of HOL Light are

`` `2 + 2` `` and `|- 2 + 2 = 4`.

Notice that terms are written enclosed in backquotes while theorems use the entailment symbol `|-`. The Boolean connectives '$\wedge$', '$\vee$', '$\Longrightarrow$' are represented in ASCII encoding `/\`, `\/` and `==>`, respectively. Universal and existential quantifier $\forall x.\, Px$ and $\exists x.\, Px$ are denoted with exclamation and interrogation marks: `!x. P x` and `?x. P x`. Other syntactic elements are borrowed from the ML world, such us the notation for concrete lists `[x1;...]`.

As the name suggests, HOL (*Higher-Order Logic*) implements a higher-order language based on a variant of the typed lambda calculus. Hence, in a rough comparison with classical logic programming languages, our system is closer to λProlog [Miller 1991] than the usual (first-order) Prolog.

Interactive proofs in HOL Light are performed by running *tactics* that operate on a context called *goal*, which represents the intermediate status of the current logical reasoning. There are simple tactics that model basic logical inference steps as well as sophisticated tactics that implement powerful decision procedures.

From the theorem proving perspective, our work consists in extending the tactic mechanism of HOL Light by introducing specific ideas coming from the miniKanren methodology.

3 A SIMPLE EXAMPLE

To give the flavor of our framework, we show an example of how to perform simple computations on lists. Let us consider the problem of computing the concatenation of two lists `[1; 2]` and `[3]`. One idiomatic way to approach this problem in HOL is by using *conversions* [Paulson 1983]. Conversions are ML procedures that receive as input a term t and output a theorem of the form `|-` t `=` t'. The term t' is the *result* of the computation, and the theorem itself is the *certificate* that guarantees its correctness. Let us show first how conversions are used before describing how one can perform the same task using our framework.

In HOL Light, one has the constant `APPEND` and the equational theorem (of the same name) that characterize it

```
|- (!l. APPEND [] l = l) /\
   (!h t l. APPEND (h :: t) l = h :: APPEND t l)
```

We can then use the conversion `REWRITE_CONV` which performs the rewriting. The ML command is

```
# REWRITE_CONV [APPEND] `APPEND [1;2] [3]`;;
```

which produces the theorem

```
|- APPEND [1; 2] [3] = [1; 2; 3]
```

Our implementation allows us to address the same problem from a logical point of view. We start by proving two theorems, namely

```
# APPEND_NIL;;
val it : thm = |- !l. APPEND [] l = l
```

and

```
# APPEND_CONS;;
val it : thm =
```

```
|- !x xs ys zs. APPEND xs ys = zs
               ==> APPEND (x :: xs) ys = x :: zs
```

to give the logical rules, in form of Horn clauses, that characterize the `APPEND` operator. Then we define a *solver*

```
let APPEND_SLV : solver =
  REPEAT_SLV (CONCAT_SLV (ACCEPT_SLV APPEND_NIL)
                         (RULE_SLV APPEND_CONS));;
```

which implements the most obvious strategy for proving a relation of the form `` `APPEND x y = z` `` by structural analysis on the list `` `x` ``. The precise meaning of the above code will be clear later; however, this can be seen as the direct translation of the Prolog program

```
append([],X,X).
append([X|Xs],Ys,[X|Zs]) :- append(Xs,Ys,Zs).
```

Then, the problem of concatenating the two lists is described by the term

```
`??x. APPEND [1;2] [3] = x`
```

where the binder `` `(??)` `` is a syntactic variant of the usual existential quantifier `` `(?)` ``, which introduces the *meta-variables* of the *query*.

The following command

```
list_of_stream
  (solve APPEND_SLV
     `??x. APPEND [1; 2] [3] = x`);;
```

runs the search process where (i) the `solve` function starts the proof search and produces a stream (i.e., a lazy list) of *solutions* and (ii) the outermost `list_of_stream` transforms the stream into a list.

The output of the previous command is a single solution which is represented by a pair where the first element is the instantiation for the meta-variable `` `x` `` and the second element is a HOL theorem

```
val it : (term list * thm) list =
  [([`x = [1; 2; 3]`], |- APPEND [1; 2] [3] = [1; 2; 3])]
```

Since the theorem is the instantiation of the original query term, it certifies the correctness of the solution.

Now comes the interesting part: as in logic programs, our search strategy (i.e., the `APPEND_SLV` solver) can be used for backward reasoning. Consider the variation of the above problem where we want to enumerate all possible splits of the list `[1; 2; 3]`. This can be done by simply changing the goal term in the previous query:

```
# list_of_stream
    (solve APPEND_SLV
       `??x y. APPEND x y = [1;2;3]`);;

val it : (term list * thm) list =
  [([`x = []`; `y = [1; 2; 3]`],
    |- APPEND [] [1; 2; 3] = [1; 2; 3]);
   ([`x = [1]`; `y = [2; 3]`],
    |- APPEND [1] [2; 3] = [1; 2; 3]);
   ([`x = [1; 2]`; `y = [3]`],
    |- APPEND [1; 2] [3] = [1; 2; 3]);
   ([`x = [1; 2; 3]`; `y = []`],
    |- APPEND [1; 2; 3] [] = [1; 2; 3])]
```

The system finds the above solutions by filtering and refining a stream of substitutions, precisely in the same way it is done in any typical miniKanren implementation; eventually, the interesting part is the associated theorems that are synthesized.

4 A LIBRARY OF SOLVERS

Our framework is based on ML procedures called *solvers* which generalize classical HOL tactics in two ways: (i) they facilitate the manipulation of meta-variables (and their associated substitutions) in the goal[1] and (ii) they allow the proof search to backtrack. Before digging into the description of what a solver is, we warn the reader that the word *goal* has a different meaning in Kanren and HOL. For the former, a goal is a function that consumes a substitution and produces a stream of substitutions; for the latter, a goal is a pair of (already proved) assumptions and a term that still has to be proved. From now on, we will use the word *goal* in the sense of HOL.

For the sake of completeness, it is worth to describe the differences among goals, tactics, and solvers.

On the one hand, the refinements that a Kanren goal does on substitutions are performed by a HOL tactic which takes a HOL goal apart into a tuple $(\mathcal{M}, \mathcal{S}, f)$, where $\mathcal{M}$ is a set collecting the introduced meta-variables so far, $\mathcal{S}$ is a list of (sub)goals, and f is a function that certifies the performed refinement. The usual HOL routine is to push and pop those tuples in a stack that represents the steps left to prove the claimed term – whenever the stack gets empty, the proof is completed.

On the other hand, a *solver* is a function that consumes a HOL goal as well as a tactic does, and produces a *stream* of such tuples that actually allows us to equip HOL Light with backtracking. To tie the knot, solvers extend tactics in the sense that every HOL tactic can be “promoted” into a solver using the ML function

```
TACTIC_SLV : tactic -> solver
```

We provide a library of basic solvers, usually having a name that ends in `_SLV`. For the rest of the paper, the following elementary solvers

- `RULE_SLV : thm -> solver`, that implements the backward chaining rule;
- `ACCEPT_SLV : thm -> solver`, that solves a goal by unifying with the supplied theorem;
- `CONJ_SLV : solver`, that splits a goal using the introduction rule of the conjunction;
- `REFL_SLV : solver`, that solves a goal which is an equation by unifying of the left- and right-hand sides;
- `ALL_SLV : solver`, that leaves the goal unmodified.

Please note that, as in Kanren systems, the unification procedure employed is not hard-wired by our framework, and each solver can implement its own unification strategy. We see two main interesting variants that one would have at disposal. The first one is to use pattern matching instead of unification; this would allow for a mechanism of input/output modes as in certain Prolog implementations. The second one would be to use a higher-order unification algorithm to unleash the full expressivity of the underlying higher-order language.

Solvers are highly compositional, as tactics in HOL and goals in Kanren are, and complex solvers can be built from simpler ones using high-order functions. For instance, given two solvers s_1 and s_2 the solver combinator `CONCAT_SLV` make a new solver that collect sequentially all solutions of s_1 followed by all solutions of s_2. This is the most basic construction for introducing backtracking into the proof strategy. The solver `COLLECT_SLV` iterates `CONCAT_SLV` over a list of solvers. Two other high-order solvers are (i) `THEN_SLV : solver -> solver -> solver` which combines sequentially two solvers and (ii) `REPEAT_SLV : solver -> solver` that keeps applying a given solver.

Solvers can be used interactively, as we can start a new goal with the command `gg` and execute solvers with `ee`; moreover, the command `bb` restore the previous proof state and the stream of results is produced by a call to

[1]The tactic mechanism currently implemented in HOL Light already provides basic support for meta-variables in goals. However, it seems to be used only internally in the implementation of the intuitionistic tautology prover `ITAUT_TAC`.

`top_thms()`. Here is an example of interaction. We first introduce the goal, notice the use of the binder `(??)` for the meta-variable `x`:

```
# gg `??x. 2 + 2 = x`;;
val it : mgoalstack =
`2 + 2 = x`
```

one possible solution is by using reflexivity that closes the proof

```
# ee REFL_SLV;;
val it : mgoalstack =
```

and allows us to form the resulting theorem

```
# list_of_stream(top_thms());;
val it : thm list = [|- 2 + 2 = 2 + 2]
```

Now, if one want to find a different solution, we can restore the initial state

```
# bb();;
val it : mgoalstack =
`2 + 2 = x`
```

then use a different solver, for instance by unifying with the equational theorem `|- 2 + 2 = 4`, which can be automatically proved using the HOL procedure `ARITH_RULE`,

```
# ee (ACCEPT_SLV(ARITH_RULE `2 + 2 = 4`));;
val it : mgoalstack =
```

and, again, take the resulting theorem

```
# list_of_stream(top_thms());;
val it : thm list = [|- 2 + 2 = 4]
```

Finally, we can change the proof strategy to find both solutions by using backtracking

```
# bb();;
val it : mgoalstack =
`2 + 2 = x`

# ee (CONCAT_SLV REFL_SLV (ACCEPT_SLV(ARITH_RULE `2 + 2 = 4`)));;
val it : mgoalstack =
# list_of_stream(top_thms());;
val it : thm list = [|- 2 + 2 = 2 + 2; |- 2 + 2 = 4]
```

The function `solve : solver -> term -> (term list * thm) stream` runs the proof search non interactively and produces a list of solutions as already shown in Section 3. In this last case it would be

```
# list_of_stream
    (solve (CONCAT_SLV REFL_SLV (ACCEPT_SLV(ARITH_RULE `2 + 2 = 4`)))
           `??x. 2 + 2 = x`);;
val it : (term list * thm) list =
  [([`x = 2 + 2`], |- 2 + 2 = 2 + 2);
   ([`x = 4`], |- 2 + 2 = 4)]
```

5 CASE STUDY: EVALUATION FOR A LISP-LIKE LANGUAGE

The material in this section is strongly inspired by the ingenious work of Byrd, Holk, and Friedman about the miniKanren system [Byrd et al. 2012], where the authors work with the semantics of the Scheme language. Here we target a dynamically scoped variant of the LISP language –not unlike it is done in [Byrd et al. 2017]– formalized as an object language inside the HOL prover. The HOL prover could be a powerful tool for a formal study of the meta-theory of a programming language such as LISP. In this perspective, this section may have a scientific interest beyond the entertaining nature of the example it is going to present.

First, we need to extend our HOL Light environment with an object datatype `sexp` for encoding S-expressions according to the following BNF grammar

```
sexp ::= Symbol string
       | List (sexp list)
```

For instance, the sexp `(list a (quote b))` is represented as HOL term with

```
`List [Symbol "list";
       Symbol "a";
       List [Symbol "quote";
             Symbol "b"]]`
```

This syntactic representation can be hard to read and gets quickly cumbersome as the size of the terms grows. Hence, we also introduce a notation for concrete sexp terms, which is activated by the syntactic pattern `'(...)`. For instance, the above example is written in the HOL concrete syntax for terms as

```
`'(list a (quote b))`
```

With this setup, we can easily specify the evaluation rules for our minimal LISP-like language. This is an inductive predicate with rules for (i) quoted expressions, (ii) variables, (iii) lambda abstractions, (iv) lists and (v) unary applications. We define a ternary predicate `` `EVAL $e$ $x$ $y$` ``, where e is a variable environment expressed as associative list, x is the input program and y is the result of the evaluation.

```
let EVAL_RULES,EVAL_INDUCT,EVAL_CASES = new_inductive_definition
  `(!e q. EVAL e (List [Symbol "quote"; q]) q) /\
   (!e a x. RELASSOC a e x ==> EVAL e (Symbol a) x) /\
   (!e l. EVAL e (List (CONS (Symbol "lambda") l))
              (List (CONS (Symbol "lambda") l))) /\
   (!e l l'. ALL2 (EVAL e) l l'
          ==> EVAL e (List (CONS (Symbol "list") l)) (List l')) /\
   (!e f x x' v b y.
      EVAL e f (List [Symbol "lambda"; List[Symbol v]; b]) /\
      EVAL e x x' /\ EVAL (CONS (x',v) e) b y
      ==> EVAL e (List [f; x]) y)`;;
```

We now use our framework for running a certified evaluation process for this language. First, we define a solver for a single step of computation

```
let STEP_SLV : solver =
  COLLECT_SLV
    [CONJ_SLV;
     ACCEPT_SLV EVAL_QUOTED;
     THEN_SLV (RULE_SLV EVAL_SYMB) RELASSOC_SLV;
     ACCEPT_SLV EVAL_LAMBDA;
     RULE_SLV EVAL_LIST;
     RULE_SLV EVAL_APP;
     ACCEPT_SLV ALL2_NIL;
     RULE_SLV ALL2_CONS];;
```

In the above code, we collect the solutions of several different solvers. Other than the five rules of the EVAL predicate, we include specific solvers for conjunctions and the two predicates REL_ASSOC and ALL2.

The top-level recursive solver for the whole evaluation predicate is now easy to define:

```
let rec EVAL_SLV : solver =
   fun g -> CONCAT_SLV ALL_SLV (THEN_SLV STEP_SLV EVAL_SLV) g;;
```

Let us make a simple test. The evaluation of the expression

```
((lambda (x) (list x x x)) (list))
```

can be obtained as follows:

```
# get (solve EVAL_SLV
             `??ret. EVAL []
                          '((lambda (x) (list x x x)) (list))
                          ret`);;

val it : term list * thm =
  ([`ret = '(() () ())`],
   |- EVAL [] '((lambda (x) (list x x x)) (list)) '(() () ()))
```

Again, we can use the declarative nature of logic programs to run the computation backwards. For instance, one intriguing exercise is the generation of quine programs, that is, programs that evaluate to themselves. In our formalization, they are those terms q satisfying the relation `EVAL [] $q$ $q$`. The following command computes the first two quines found by our solver.

```
# let sols = solve EVAL_SLV `??q. EVAL [] q q`);;
# take 2 sols;;

val it : (term list * thm) list =
  [([`q = List (Symbol "lambda" :: _3149670)`],
    |- EVAL [] (List (Symbol "lambda" :: _3149670))
       (List (Symbol "lambda" :: _3149670)));
   ([`q =
      List
      [List
       [Symbol "lambda"; List [Symbol _3220800];
        List [Symbol "list"; Symbol _3220800; Symbol _3220800]];
       List
       [Symbol "lambda"; List [Symbol _3220800];
        List [Symbol "list"; Symbol _3220800; Symbol _3220800]]]`],
    |- EVAL []
       (List
       [List
        [Symbol "lambda"; List [Symbol _3220800];
         List [Symbol "list"; Symbol _3220800; Symbol _3220800]];
        List
        [Symbol "lambda"; List [Symbol _3220800];
         List [Symbol "list"; Symbol _3220800; Symbol _3220800]]])
       (List
```

```
      [List
       [Symbol "lambda"; List [Symbol _3220800];
        List [Symbol "list"; Symbol _3220800; Symbol _3220800]];
       List
       [Symbol "lambda"; List [Symbol _3220800];
        List [Symbol "list"; Symbol _3220800; Symbol _3220800]]]))]
```

One can easily observe that any lambda expression is trivially a quine for our language. This is indeed the first solution found by our search:

```
([`q = List (Symbol "lambda" :: _3149670)`],
 |- EVAL []
         (List (Symbol "lambda" :: _3149670))
         (List (Symbol "lambda" :: _3149670)))
```

The second solution is more interesting. Unfortunately, it is presented in a form that is hard to decipher. A simple trick can help us to present this term as a concrete sexp term: it is enough to replace the HOL generated variable (`` `_3149670` ``) with a concrete string. This can be done by an ad-hoc substitution:

```
# let [_; i2,s2] = take 2 sols;;
# vsubst [`"x"`,hd (frees (rand (hd i2)))] (hd i2);;

val it : term =
  `q = '((lambda (x) (list x x)) (lambda (x) (list x x)))`
```

If we take one more solution from `sols` stream, we get a new quine which, interestingly enough, is precisely the one obtained in [Byrd et al. 2012]:

```
val it : term =
  `q =
   '((quote (lambda (x) (list x (list (quote quote) x))))
     (quote (quote (lambda (x) (list x (list (quote quote) x))))))`
```

6 DESCRIPTION OF OUR CODE

The HOL Light theorem prover and our extension are written in OCaml and, more precisely, in a rather minimal and conservative subdialect of it, which should be understandable to everyone that has some familiarity with any of the languages of the ML family. Our code is available from a public repository, in particular, a release has been created at https://github.com/massimo-nocentini/kanren-light/releases/tag/miniKanren2020 .

Besides the code presented in this article, the above repository contains some other experiments of various nature, including the following:

- An implementation of the Quicksort algorithm. The procedure outputs the sorted list together with a formal proof that such list is indeed sorted and in bijection with the input lists.
- A solver for the *Monte Carlo Lock*, a brain teaser by Smullyan [Smullyan 2009], where one has to unlock a *safe* whose *key* is the fixed point of an abstract machine. The interesting thing is that the solver is essentially derived from the formal specification in HOL of the puzzle.
- An intuitionistic first-order tautology prover `ITAUT_SLV`. This is inspired by a similar tactic `ITAUT_TAC` already available in HOL Light.[2] However, HOL tactics cannot backtrack, which implies that `ITAUT_TAC` is

[2]The tactic `ITAUT_TAC` has a peculiar role in the HOL system. It is used during the *bootstrap* of the system to prove several basic logical lemmas. After the intial stages, a much more powerful and faster procedure for (classical) first-order logic `MESON_TAC` is installed in the system, and the `ITAUT_TAC` becomes superfluous.

incomplete. Our solver `ITAUT_SLV` is coded in pretty much the same way as `ITAUT_TAC`, but it is complete (although this latter fact can be claimed only via a meta-theoretical analysis).

Our code is conceived for experimenting, and very little or no attention has been paid to optimizations. Despite this, the OCaml runtime and the HOL Light implementation have an established reputation of being time- and memory-efficient systems (compared with similar tools). From our informal tests, it seems that this efficiency is, at least partially, inherited by our implementation.

7 FUTURE AND RELATED WORK

We presented a rudimentary framework inspired by Kanren systems implemented on top of the HOL Light theorem prover that enables a logic programming paradigm for proof searching. More specifically, it facilitates the use of meta-variables in HOL goals and permits backtracking during the proof construction. Despite the simplicity of the present implementation, we have already shown the implementation of some paradigmatic examples of logic-oriented proof strategies.

It would be interesting to enhance our framework with more features:

- Implement higher-order unification as Miller's higher-order patterns, so that our system can enable higher-order logic programming in the style of λProlog [Felty et al. 1988].
- Support constraint logic programming [Hemann and Friedman 2017], e.g., by adapting the data structure that represents goals.

Besides extending our system with new features, we plan to test it on further examples. One natural domain of applications would be the development of decision procedures. While HOL Light already offers some remarkable tools for automatic theorem proving, our system could offer new alternatives leaning to simplicity and compositionality. For instance, we could try to translate in our system the approach of αlean*TAP* [Near et al. 2008] for implementing an automatic procedure for first-order classical logic in HOL Light analogous to the `blast` tactic of Paulson [Paulson 1999] in Isabelle.

REFERENCES

William E. Byrd. 2009. *Relational Programming in Minikanren: Techniques, Applications, and Implementations.* Ph.D. Dissertation. USA. Advisor(s) Friedman, Daniel P.

William E. Byrd, Michael Ballantyne, Gregory Rosenblatt, and Matthew Might. 2017. A Unified Approach to Solving Seven Programming Problems (Functional Pearl). *Proc. ACM Program. Lang.* 1, ICFP, Article 8 (Aug. 2017), 26 pages. https://doi.org/10.1145/3110252

William E. Byrd, Eric Holk, and Daniel P. Friedman. 2012. miniKanren, Live and Untagged: Quine Generation via Relational Interpreters (Programming Pearl). In *Proceedings of the 2012 Annual Workshop on Scheme and Functional Programming* (Copenhagen, Denmark) *(Scheme '12)*. ACM, New York, NY, USA, 8–29. https://doi.org/10.1145/2661103.2661105

Amy P. Felty, Elsa L. Gunter, John Hannan, Dale Miller, Gopalan Nadathur, and Andre Scedrov. 1988. Lambda-Prolog: An Extended Logic Programming Language. In *Proceedings of the 9th International Conference on Automated Deduction*. Springer-Verlag, Berlin, Heidelberg, 754–755.

Daniel P. Friedman, William E. Byrd, and Oleg Kiselyov. 2005. *The Reasoned Schemer.* The MIT Press.

John Harrison. 1996. HOL Light: A Tutorial Introduction. In *Proceedings of the First International Conference on Formal Methods in Computer-Aided Design (FMCAD '96)*. Springer-Verlag, Berlin, Heidelberg, 265–269.

Jason Hemann and Daniel P. Friedman. 2013. microKanren: A Minimal Functional Core for Relational Programming. In *Proceedings of the 2013 Workshop on Scheme and Functional Programming* (Alexandria, VA) *(Scheme '13)*.

Jason Hemann and Daniel P. Friedman. 2017. A Framework for Extending microKanren with Constraints. *Electronic Proceedings in Theoretical Computer Science* 234 (Jan 2017), 135–149. https://doi.org/10.4204/eptcs.234.10

Dale Miller. 1991. A Logic Programming Language with Lambda-Abstraction, Function Variables, and Simple Unification. *Journal of Logic and Computation* 1 (1991), 253–281.

Joseph P. Near, William E. Byrd, and Daniel P. Friedman. 2008. αleanTAP: A Declarative Theorem Prover for First-Order Classical Logic. In *Logic Programming*, Maria Garcia de la Banda and Enrico Pontelli (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 238–252.

Lawrence Paulson. 1983. A higher-order implementation of rewriting. *Science of Computer Programming* 3, 2 (1983), 119 – 149. https://doi.org/10.1016/0167-6423(83)90008-4

Lawrence C. Paulson. 1999. A Generic Tableau Prover and its Integration with Isabelle. *Journal of Universal Computer Science* 5, 3 (mar 1999), 73–87. http://www.jucs.org/jucs_5_3/a_generic_tableau_prover

Raymond M. Smullyan. 2009. *The Lady Or the Tiger?: And Other Logic Puzzles.* Dover Publications.