A New Higher-order Unification Algorithm for λKanren

WEIXI MA, Indiana University
DANIEL P. FRIEDMAN, Indiana University

λKanren, adapted from λProlog, extends miniKanren with higher-order logic programming. The existing higher-order unification algorithms, however, are neither sound nor complete w.r.t. an "intuitive" understanding of α-equivalence. This paper shows these gaps and proposes a new method to handle α-equivalence in higher-order unification.

ACM Reference Format:
Weixi Ma and Daniel P. Friedman. 2021. A New Higher-order Unification Algorithm for λKanren. 1, 1 (August 2021), 10 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 BACKGROUND

Is there a world where **true is equal to false**? Indeed, there is, if higher-order unification [13] and miniKanren [9] meet each other. Even if all components are already well-established, it may still end up with unnatural or even absurd results, if we combine them naively. This paper unveils the difficulties in handling binders with standard higher-order unification. To solve these problems, we propose a new unification formulation.

Higher-order unification [13] solves equations among λ-terms modulo $\alpha\beta\eta$-equivalence [3]. Miller [19] finds a decidable fragment of higher-order unification problems by restricting the application forms of the input terms. For an application form $F\,\overline{a}$, if F is a unification variable, then $\overline{a}$ must be a list of zero or more distinct bound variables. Such restricted application forms are called *patterns* and we hereafter refer to the problem identified by Miller as *pattern unification*.

Pattern unification then becomes the core of λProlog [20]. λProlog implements a more expressive logic [21] than Prolog does. Thanks to higher-order unification, λProlog also extends Prolog with binding structures and the ability to identify $\alpha\beta\eta$-equivalence among these binding structures. With such extensions, λProlog is a powerful tool for metaprogramming and theorem proving.

On the other hand, miniKanren [9] is a school of logic programming languages that focuses on simplicity, as well as the purity of functions and relations. It rebuilds and demystifies the essence of Prolog. 40 lines of code [11] are enough to build the core of miniKanren. With their emphasis on pedagogy, miniKanren is an ideal tool for beginners to learn logic programming and for experts to experiment using new ideas to enhance the language itself.

If a miniKanren language can rebuild and demystify λProlog, then, we believe, this joining of forces could yield a significant improvement to the logic programming community. Just as miniKanren is taught in undergraduate level courses, a "λKanren" may greatly open up the study of higher-order logic programming. Other than in a workshop paper [17], higher-order logic programming remains unexplored by the miniKanren community.

Authors' addresses: Weixi Ma, mvc@iu.edu, Indiana University; Daniel P. Friedman, dfried00@gmail.com, Indiana University.

XXXX-XXXX/2021/8-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

We show, in Sections 2 and 3, the obstacles to extend miniKanren with higher-order logic programming. To solve these difficulties, in section 4, we propose a new higher-order unification formulation and its algorithm. This algorithm is the heart of λKanren, a new member that brings higher-order logic programming to the Kanren family. Last but not least, section 6 concludes this paper with related work.

2 GRAFTING OR SUBSTITUTION?

miniKanren provides intuitive operators. In the following, `run` corresponds to the existential quantifier. [1] This query asks if there exists one variable `X` (hereafter capitalized letters are used for unification variables) that is equal to the constant `'cat`. miniKanren answers with a list of one element that is `'cat`.

```
> (run 1 X
    (== X 'cat))
'(cat)
```

This query succeeds. Internally, the answer is a *grafting*, a pure textual replacement, resulting from unifying `X` and `'cat`, written as $\{\texttt{X} \mapsto \texttt{'cat}\}$.

Next, binding structures are the first step towards higher-order logic programming. Let's introduce the λ operator to our language. Then λKanren should be able to handle the following query.

```
> (run 1 X
    (== (λ (a) a) (λ (b) b)))
'(_0)
```

This query succeeds. Here $\lambda a.a$ and $\lambda b.b$ can be unified because they are α-equivalent. And λKanren answers with a list with one element. This element is `_0`, a way to express that variable `X` is not restricted.

Out of curiosity, a novice in higher-order logic programming may try to explore more interesting queries, such as the following. [2]

```
> (run 1 X
    (== (λ (a) X) (λ (b) b)))
```

Does this query succeed? If the function body of the first term is a unification variable, what should `X` be to make the two terms equal? The variable that corresponds to the same binder, `a`, must be the answer for any newcomer to λ-calculus with a basic understanding of α-equivalence.

Surprisingly enough, with higher-order unification, there is no answer.

```
> (run 1 X
    (== (λ (a) X) (λ (b) b)))
'()
```

This query fails. Under any existing higher-order unification algorithm, `X` and `a` cannot be unified due to the *scope restrictions*.

[1] In miniKanren, `fresh` is the existential quantifier. `run` is an interface operator that starts computing a query and is implemented using `fresh`. To focus on our discussion on binders, we do not signify their difference.

[2] We use solid boxes for examples that are complete and dash boxes for examples that are incomplete or somewhat questionable.

Standard higher-order unification relies on *substitution*, but not grafting, to instantiate unification variables. Substitution is capture-avoiding, whereas grafting is not. If we take the following example into account, it makes some sense to disallow associating `X` and `a` (with either grafting or substitution).

```
> (run 1 X
    (== (λ (a) X) (λ (b) b))
    (== (λ (c) X) (λ (d) d)))
```

This example is an implicit conjunction between two equations. It asks for at most one answer, if there exists one, as `run 1` suggests. Associating `X` with `a` cannot work, neither does `c`. If we text-replace `X` with either, then the other one fails.

Higher-order unification implements this restriction with scopes. Unification variables are ∃-quantified and lambda variables are ∀-quantified. A scope is a list of those quantifiers. Further, a ∀-quantified variable may instantiate (or occurs in the instantiation of) an ∃-quantified variable, only if the ∀ variable precedes the ∃ variable in the scope.

In the above example that fails, the ∀-quantified `a` does not precede the ∃-quantified `X`. Its scope is `'((∃ X) (∀ a))`. So, `a` cannot instantiate `X`.

But, do we really want this? This failure is counter-intuitive. For beginners who are not familiar with higher-order unification, α-equivalence is only decided by the binder positions. But pattern unification imposes an appendage limitation by taking unification variables into account. It is fair to say that the existing pattern unification algorithms are not complete w.r.t. the natural sense of α-equivalence.

We first propose to treat unification variables and lambda variables differently. In particular, lambda variables are effectively constants if they are represented by *de Bruijn levels* [7] (hereafter *levels*). E.g., instead of `'()`, λKanren may return `'(d0)`, an answer that suggests "these two terms can be made α-equal, if `X` corresponds to de Bruijn level 0."

```
> (run 1 X
    (== (λ (a) X) (λ (b) b)))
'(d0)
```

By mapping unification variables to de Bruijn levels, grafting no longer imposes the problem of capturing. Consider the previously-failed example: `X` may correspond to de Bruijn level 0 in both $\lambda a.X \stackrel{?}{=} \lambda b.b$ and $\lambda c.X \stackrel{?}{=} \lambda d.d$. And λKanren would result in a success with `'(d0)`.

```
> (run 1 X
    (== (λ (a) X) (λ (b) b))
    (== (λ (c) X) (λ (d) d)))
'(d0)
```

If we instantiate `X` with `d0` and reify these levels w.r.t. their binders, we have the problem solved with no surprise.

```
    (== (λ (a) a) (λ (b) b))
    (== (λ (c) c) (λ (d) d))
```

After all, using grafting with de Bruijn levels captures the essence of α-equivalence: names don't matter. A unification algorithm based on de Bruijn levels would also be more straightforward

because it's practically first-order. But, we cannot yet lift the scope restriction on lambda variables and drop in de Bruijn levels, because sometimes names do matter and ...

3 ... WE CAN TURN FALSE TO TRUE

Here we are concerned about an implementation technique that has been used in all pattern unification algorithms, except for those with de Bruijn numbers. [3] We call this technique *binder-merging*.

When two terms are unified, one term's binders are replaced by the other's. Then α-equivalence is as simple as structural equivalence, e.g. unifying

$$\lambda x.\lambda y.Xxy \stackrel{?}{=} \lambda a.\lambda b.ab$$

is reduced to unifying

$$\lambda a.\lambda b.Xab \stackrel{?}{=} \lambda a.\lambda b.ab$$

and then solved by $\{X \mapsto \lambda a.\lambda b.ab\}$. Instantiating X gives

$$\lambda a.\lambda b.((\lambda a.\lambda b.ab)ab) \stackrel{?}{=} \lambda a.\lambda b.ab$$

that β-reduces to the following.

$$\lambda a.\lambda b.ab \stackrel{?}{=} \lambda a.\lambda b.ab$$

As convenient as it is, binder-merging does not handle the name shadowing problem. This is overlooked by all pattern unification algorithms. Algorithms that use what we call binder-merging state upfront that they don't consider name shadowing, e.g. Nipkow and Prehofer [25], Qian [32].

Take the following program.

```
> (run 1 X
    (== (λ (a) (λ (a) X)) (λ (c) (λ (d) c))))
```

With either binder-merging or de Bruijn syntax, the unification problem is effectively reduced to the following.

$$\lambda c.\lambda d.X \stackrel{?}{=} \lambda c.\lambda d.c$$

Then it appears to be solvable by $\{\texttt{X} \mapsto \texttt{c}\}$, or equivalently $\{\texttt{X} \mapsto \texttt{d0}\}$, that associates X with the outer-binder. If we instantiate the unification variables in the original problem and naively reify de Bruijn levels with concrete names, then we have the following,

```
(== (λ (a) (λ (a) a)) (λ (c) (λ (d) c)))
```

which claims that the λ-encoded false is true!

The name shadowing problem has wide impact. The next example shows that binder-merging could go wrong even without lifting the scope restrictions.

$$\lambda x.\lambda y.x\ y \stackrel{?}{=} \lambda a.\lambda a.a\ a$$

is converted to

$$\lambda a.\lambda a.a\ a \stackrel{?}{=} \lambda a.\lambda a.a\ a,$$

which then ends up with another false positive.

Current solutions use either a pre-processor that adds gensyms to lambda variables to make each individual unique or a smart reifier that detects name shadowing. One downside is that the randomly-generated symbols are not pleasantly-readable. Also, these extra processors demand extra complexity, as to both runtime speed and implementation technique.

[3] There are two ways to implement de Bruijn's nameless syntax, indices and levels. Both use numbers to represent variables.

$$
\begin{array}{rcll}
t & ::= & X & \text{unification variable} \\
 & | & t\ t & \text{function application} \\
 & | & \lambda a.t & \text{named-binder} \\
 & | & a & \text{name} \\
 & | & \lambda\, t & \text{nameless-binder} \\
 & | & \ell & \text{level} \\
\phi & ::= & \epsilon \mid a, \phi & \text{scope} \\
\kappa & ::= & \langle \phi;\, t \rangle & \text{static closure}
\end{array}
$$

Fig. 1. Terms, scopes, and static closures

4 STATIC CLOSURES, THE SOLUTION TO NAME SHADOWING

We propose a simple and efficient alternative, *static closures*. Static closures are integrated into the unification process and they decide α-equivalence while correctly handling name shadowing. We obtain a new algorithm by extending traditional pattern unification with static closures. This algorithm is formulated differently from the previous ones, since it uses a mixture of named and nameless syntax, as introduced in Fig. 1. The nameless syntax, i.e., nameless-binders and levels are derived during unification.

A *static closure* is a pair of a *scope* and a unification term. A *scope* is a list of binders. In essence, a static closure captures the surrounding binders of a term. E.g., the term $\lambda a.\lambda b.X$ reduces to the static closure $\langle a, b;\, X \rangle$.

Static closures support the following operations:

- `ext(`ϕ`, a)` adds the name `a` to the scope ϕ. If `a` has been previously added to ϕ, then this older one is shadowed and its corresponding level becomes unusable.
- `db(`ϕ`, a)` is `Just` ℓ where ℓ is the de Bruijn level of `a` w.r.t the scope ϕ, if `a` is not shadowed in ϕ; otherwise, the result is `Nothing`.
- `db(`ϕ`,` ℓ`)` is `Just` ℓ, if ℓ is available in ϕ; otherwise, the result is `Nothing`.

Example 1. Consider the following unification problem.

$$\lambda a.\lambda b.X \stackrel{?}{=} \lambda c.\lambda d.c$$

The problem is reduced to comparing two static closures.

$$\langle a, b;\, X \rangle \stackrel{?}{=} \langle c, d;\, c \rangle$$

We first apply `db` on the right-hand side.

$$\langle a, b;\, X \rangle \stackrel{?}{=} \texttt{Just 0}$$

Then we try to solve `X` by applying `db` on the scope of the left-hand side and 0.

$$\texttt{Just 0} \stackrel{?}{=} \texttt{Just 0}$$

It works. So, the solution is $\{\texttt{X} \mapsto \texttt{d0}\}$.

Example 2. Consider another example with name shadowing:

$$\lambda a.\lambda a.X \stackrel{?}{=} \lambda c.\lambda d.c$$

is reduced to

$$\langle a, a;\, X \rangle \stackrel{?}{=} \langle c, d;\, c \rangle.$$

$$\begin{array}{rrcl} \text{Definitions} & D & ::= & G \supset D \mid D \wedge D \mid \forall x\, D \\ \text{Goals} & G & ::= & G \wedge G \mid G \vee G \mid \exists x\, G \mid D \supset G \mid \forall x\, G \end{array}$$

Fig. 2. Horn Clauses and Hereditary Harrop Formulas

We first apply db on the right-hand side.

$$\langle a, a;\ X\rangle \stackrel{?}{=} \texttt{Just 0}$$

Next, try to solve X by applying db on the scope of the left-hand side and 0. This time, $\texttt{db}(\langle a, a;\ 0\rangle)$ is Nothing because the outer binder a is shadowed.

$$\texttt{Nothing} \stackrel{?}{=} \texttt{Just 0}$$

And we know the unification fails.

Example 3. Some solvability cannot be immediately decided, e.g.,

$$\lambda a.\lambda b.X \stackrel{?}{=} \lambda c.\lambda d.Y$$

becomes an equation between two static closures of unification variables.

$$\langle a, b;\ X\rangle \stackrel{?}{=} \langle c, d;\ Y\rangle$$

Here, the grafting itself is not able to describe the solution. Now our solution needs a *context*: the set of equations between static closures of unification variables. This problem has the solution: the grafting being empty and the context being $\{\langle a, b;\ X\rangle \stackrel{?}{=} \langle c, d;\ Y\rangle\}$. Practical problems, such as type checking, involve solving many unification problems. The accumulated equations between closures of one unification problem are usually solved and reduced to grafting, during the unification on other problems.

5 A SHALLOW EMBEDDING IMPLEMENTATION OF λKANREN

Besides the unification algorithm, the underlying logic of λProlog is more expressive than Prolog's. Prolog (and miniKanren) implements Horn Clauses [2] while the underlying logic of λProlog is Hereditary Harrop Formulas [21]. Fig. 2 illustrates these two logics. The dimmed part is their overlap and the non-dimmed part is from Hereditary Harrop Formulas only. This section reviews such language extensions of λKanren, introduced in Ma et al. [17].

miniKanren also implements first-order Horn clauses. As shown in the previous sections, the host language uses run to introduce *goal*s with some existentially quantified variables ($\exists x G$). Besides, conjunctions ($G \wedge G$) are implicit and disjunctions ($G \vee G$) use miniKanren's $\texttt{cond}^e$ operator.

For example, the formula

$$\exists X\, \exists Y\, ((X = Y \wedge X = \texttt{'cat}) \vee (X = \texttt{'cat} \wedge Y = \texttt{'dog}))$$

corresponds to the following program.

```
> (run* (X Y)
    (cond^e
      [(== X Y) (== X 'cat)]
      [(== X 'cat) (== Y 'dog)]))
'((cat cat) (cat dog))
```

On the other hand, miniKanren introduces *definitions* by defining a function in its host language. These definitions are universal quantifiers with implicit implications. E.g.,

$$\forall X\,(X = \texttt{'()}) \supset (\texttt{empty?}\ X)$$

can be translated to the following miniKanren program.

```
(define (empty? X)
  (== X '()))
```

λKanren relies on two new operators, *implicational goals* and *universally quantified goals*, to express Hereditary Harrop Formulas.

- Implicational goals are implemented with the operator `assume` that introduces hypotheses in its clause. As an example, we may use the `assume` operator to temporarily extend the == relation.

```
>(run 2 X
   (assume (== 'apple 'orange)
     (== 'apple X)))
'(apple orange)
```

- The `all` operator introduces universally quantified goals. E.g., we may synthesize a function that yields whatever it takes in, using the following.

```
>(run 1 F
   (all (A)
     (== A (F A))))
'((lambda (_0) _0))
```

- As in the previous example, the == operator recognizes β-equivalence. Indeed, it will be powered by the unification algorithm proposed in the last section and recognize $\alpha\beta$-equivalence.

6 RELATED WORK

The related work can be categorized using three tracks: variations of higher-order unification algorithms, logic programming languages that extend first-order logic, and nominal unification.

6.1 Higher-order unification algorithms

Higher-order unification emerged during the search of automated theorem proving of higher-order logic, i.e., logic with quantifiers over sets and functions. In the setting of λ-calculus, higher-order unification solves equations modulo $\alpha\beta\eta$-equivalence. Since β-conversion is a powerful and complicated reduction, it is not surprising that higher-order unification is not decidable. This is proved by Huet [12] and Lucchesi [16], independently.

Undecidability does not get in the way of practicality. Many applications [8, 28, 29] have been built on Huet [12]'s semi-decidable algorithm. This algorithm has introduced two core ideas of higher-order unification algorithms.

- *Rigid and flex terms.* A term is called rigid if its head symbol is a bound variable or a constant. A term is called flex if its head symbol is a unification variable. Being flexible is in the sense that the term can be removed through a substitution (or a grafting).

- *Elementary substitution.* Each unification variable should capture all its visible bound variables, e.g. in $\lambda a_0 ... \lambda a_i (h(X_0\ a_0 ... a_i) ... (X_j\ a_0 ... a_i))$, each X_n should take every a as input, so it doesn't lose any information.

The first decidable fragment, discovered by Miller [19], then becomes the foundation of the later research on higher-order unification. Qian [32] has claimed a result, modified from Miller's algorithm, that runs in linear time, although Qian's algorithm has not been implemented. Hardin et al. [10] have added explicit substitution with de Bruijn numbers to Miller's algorithm to improve its efficiency. The latter one results in practical implementations, such as in Pfenning and Schürmann [30].

Besides, Nipkow [23] introduces a functional version of Miller's algorithm. It is appreciated for its simplicity and shows its practicality in proof assistants [24].

Higher-order unification is also a critical component of building dependent typed languages for two reasons: (1) β-conversion is involved during type checking, and (2) unification is needed to elaborate a user-friendly surface language to a cumbersome core language. As examples, the elaborates of Agda [26] and Beluga [31] adapt Miller's algorithm for dependent types [1]. As Agda uses de Bruijn numbers in its core language [26], it suffers from the name shadowing problem mentioned above and must make all variable names unique in a "gensym" manner. Doing so bloats type checking with randomly generated symbols and makes the error messages painful to read.

6.2 Extending logic programming languages

λProlog is well-equipped for building theorem provers and programming languages, as it provides a natural integration of $\alpha\beta$-conversion. It is also a theorem prover with the built-in capability of proof search. Miller and Nadathur [20] provides a comprehensive introduction to λProlog, using an implementation called Teyjus[35]. Makam [33] is a refined version that lifts some restrictions in Teyjus. Stampoulis and Chlipala [34] demonstrate how Makam makes it easier to build complicated type systems.

Unlike the Prolog languages, which are implemented by writing interpreters or compilers, the Kanren languages feature shallow embedding. Instead of building everything from the ground up, language objects are represented by functions and macros of a host language. Shallow embedding enables fast prototyping and reuses the developers tools of the host language.

αKanren [4] shows the convenience of shallow embedding. It extends miniKanren with nominal unification [36] and the whole language is implemented within a few hundred lines of code. Its practicality is shown by Near et al. [22] that uses it to build a theorem prover.

6.3 Nominal Unification

Nominal unification [36] extends first-order unification with α-equivalence. It handles variable names with *swappings*, a list of name pairs.

Nominal unification and pattern unification are equally expressive. Cheney [6] shows that pattern unification problems can be reduced to nominal unification. Later, Levy and Villaret [14] introduces the reduction from nominal unification to pattern unification.

As pattern unification is decidable in linear time [32], many are seeking a linear time bound of nominal unification. Yet $O(n^2)$ is the state-of-the-art. Levy and Villaret [14]'s reduction to pattern unification is quadratic, which is the first proof of $O(n^2)$. Later, Levy and Villaret [15] and Calvès [5] independently find more quadratic results, building on advances of Paterson and Wegman [27] and Martelli and Montanari [18].

7 CONCLUSION AND FUTURE GOALS

Extending miniKanren with higher-order programming is a non-trivial endeavor. Our concern is that higher-order unification algorithms consist of many surprises in α-equivalence. In this regard, we have proposed static closures, a new means to handle binding structures. Static closures use a mixture between the ordinary λ-syntax and de Bruijn's nameless syntax.

This paper shows a preview of static closures. We anticipate extending the formulation of higher-order unification with static closures and derive a sound and complete algorithm for α-equivalence.

REFERENCES

[1] Andreas Abel and Brigitte Pientka. 2011. Higher-order Dynamic Pattern Unification for Dependent Types and Records. In *Proceedings of the 10th International Conference on Typed Lambda Calculi and Applications (TLCA'11)*. Springer-Verlag, Berlin, Heidelberg, 10–26.

[2] Krzysztof R. Apt and M. H. van Emden. 1982. Contributions to the Theory of Logic Programming. *Journal of the ACM (JACM)* 29, 3 (July 1982), 841–862.

[3] H. P. Barendregt. 1981. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland PubCo, Sole distributors for the USAand Canada Elsevier North-Holland.

[4] William E. Byrd and Daniel P. Friedman. 2007. αKanren: A Fresh Name In Nominal Logic Programming. *Université Laval Technical Report* DIUL-RT-0701, Scheme Workshop '07, editor Danny Dubé (2007), 79 – 90.

[5] Christophe Calvès. 2010. *Complexity and Implementation of Nominal Algorithms*. King's College of London.

[6] James Cheney. 2005. Relating Nominal and Higher-order Pattern Unification. In *Proceedings of UNIF 2005*. 104–119.

[7] Nicolaas G. de Bruijn. 1972. Lambda Calculus Notation with Nameless Dummies, a Tool for Automatic Formula Manipulation, with Application to the Church-Rosser Theorem. *Indagationes Mathematicae (Proceedings)* 75, 5 (Jan. 1972), 381–392.

[8] Conal Elliott. 1989. Higher-order unification with dependent types. *Rewriting Techniques and Applications* 355 (Jan. 1989).

[9] Daniel P. Friedman, William E. Byrd, Oleg Kiselyov, and Jason Hemann. 2018. The Reasoned Schemer, Second Edition.

[10] Thérèse Hardin, Gilles Dowek, Claude Kirchner, and Frank Pfenning. 1999. Unification via Explicit Substitutions: The Case of Higher-Order Patterns. (Jan. 1999).

[11] Jason Hemann and Daniel P. Friedman. 2013. μKanren: A Minimal Core for Relational Programming. In *Proceedings of the 2013 Workshop on Scheme and Functional Programming (Scheme'13)*, Vol. 6.

[12] Gerard P. Huet. 1973. The undecidability of unification in third order logic. *Information and Control* 22, 3 (April 1973), 257–267. https://doi.org/10.1016/S0019-9958(73)90301-X

[13] Gérard P.P. Huet. 2002. Higher Order Unification 30 Years Later. In *Proceedings of the 15th International Conference on Theorem Proving in Higher Order Logics (TPHOLs '02)*. Springer-Verlag, London, UK, UK, 3–12.

[14] Jordi Levy and Mateu Villaret. 2008. Nominal Unification from a Higher-Order Perspective. In *Rewriting Techniques and Applications (Lecture Notes in Computer Science)*, Andrei Voronkov (Ed.). Springer Berlin Heidelberg, 246–260.

[15] Jordi Levy and Mateu Villaret. 2010. An Efficient Nominal Unification Algorithm. In *Proceedings of the 21st International Conference on Rewriting Techniques and Applications (RTA '10)*. Edinburgh, Scottland, UK, 209–226.

[16] L Lucchesi. 1972. The undecidability of the unification problem for third order languages. https://www.semanticscholar.org/paper/The-undecidability-of-the-unification-problem-for-Lucchesi/218e43e101803d96031f56360e450fcb349b500b

[17] Weixi Ma, Kuang-chen Lu, and Daniel P. Friedman. 2020. Higher-order Logic Programming with λKanren. In *miniKanren 2020*.

[18] Alberto Martelli and Ugo Montanari. 1982. An Efficient Unification Algorithm. *ACM Trans. Program. Lang. Syst.* 4, 2 (April 1982), 258–282.

[19] Dale Miller. 1991. A Logic Programming Language with Lambda-abstraction, Function Variables, and Simple Unification. *Journal of Logic and Computation* 1, 4 (Sept. 1991), 497–536.

[20] Dale Miller and Gopalan Nadathur. 2012. *Programming with Higher-Order Logic*. Cambridge University Press, Cambridge.

[21] Dale Miller, Gopalan Nadathur, and Andre Scedrov. 1987. Hereditary Harrop Formulas and Uniform Proof Systems. (Jan. 1987), 98–105.

[22] Joseph P. Near, William E. Byrd, and Daniel P. Friedman. 2008. αleanTAP: A Declarative Theorem Prover for First-order Classical Logic. In *Logic Programming (LNCS 5366)*. Springer, Berlin, Heidelberg, 238–252.

[23] Tobias Nipkow. 1993. Functional Unification of Higher-Order Patterns. *Proc. 8th IEEE Symp. Logic in Computer Science* (1993).

[24] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. 2002. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Springer-Verlag, Berlin Heidelberg.
[25] T. Nipkow and Christian Prehofer. 1998. Higher-order rewriting and equational reasoning. *Automated Deduction - A Basis for Applications* 1 (Jan. 1998).
[26] Ulf Norell. 2007. *Towards A Practical Programming Language Based on Dependent Type Theory*. Chalmers University of Technology.
[27] M. S. Paterson and M. N. Wegman. 1978. Linear Unification. *J. Comput. System Sci.* 16, 2 (April 1978), 158–167.
[28] Lawrence C. Paulson. 1986. Natural deduction as higher-order resolution. *The Journal of Logic Programming* 3, 3 (Oct. 1986), 237–258.
[29] Frank Pfenning. 1989. Elf: a language for logic definition and verified metaprogramming. In *[1989] Proceedings. Fourth Annual Symposium on Logic in Computer Science*. 313–322. ISSN: null.
[30] Frank Pfenning and Carsten Schürmann. 1999. System Description: Twelf — A Meta-Logical Framework for Deductive Systems. In *CADE-16 (LNCS 1632)*. Springer, Berlin, Heidelberg, 202–206.
[31] Brigitte Pientka and Joshua Dunfield. 2010. Beluga: A Framework for Programming and Reasoning with Deductive Systems (System Description). In *Automated Reasoning (Lecture Notes in Computer Science)*, Jürgen Giesl and Reiner Hähnle (Eds.). Springer, Berlin, Heidelberg, 15–21.
[32] Zhenyu Qian. 1993. Linear Unification of Higher-Order Patterns. In *TAPSOFT'93: Theory and Practice of Software Development (Lecture Notes in Computer Science)*, M. C. Gaudel and J. P. Jouannaud (Eds.). Springer Berlin Heidelberg, 391–405.
[33] ANTONIS STAMPOULIS. 2021. Makam. http://astampoulis.github.io/makam/
[34] Antonis Stampoulis and Adam Chlipala. 2018. Prototyping a functional language using higher-order logic programming: a functional pearl on learning the ways of λProlog/Makam. *Proceedings of the ACM on Programming Languages* 2, ICFP (July 2018), 93:1–93:30. https://doi.org/10.1145/3236788
[35] Teyjus team. 2021. Teyjus. http://teyjus.cs.umn.edu/
[36] Christian Urban, Andrew M. Pitts, and Murdoch J. Gabbay. 2004. Nominal Unification. *Theoretical Computer Science* 323, 1-3 (Sept. 2004), 473–497.