Higher-order Logic Programming with λKanren

WEIXI MA, KUANG-CHEN LU, and DANIEL P. FRIEDMAN, Indiana University

We present λKanren, a new member of the Kanren family [2] that is inspired by λProlog [5]. With a shallow embedding implementation, the term language of λKanren is represented by the functions and macros of its host language. As a higher-order logic programming language, λKanren is extended with a subset of higher-order hereditary Harrop formulas [7].

CCS Concepts: • **Computer systems organization** → **Embedded systems**; *Redundancy*; Robotics; • **Networks** → Network reliability.

ACM Reference Format:
Weixi Ma, Kuang-chen Lu, and Daniel P. Friedman. 2020. Higher-order Logic Programming with λKanren. 1, 1 (July 2020), 8 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

λKanren introduces four new operators to μKanren [3]: `tie`, `app`, `assume-rel`, and `all`. The opertors `tie` and `app` create binding structures. In addition, the ≡ operator recognizes $\alpha\beta$-conversions between binding structures. The `assume-rel` and `all` operators enable more expressive reasoning with Hereditary Harrop formulas [6]. To demonstrate λKanren's increment to μKanren, we first review the two forms of logic, *fohc* and *hohh*, behind these two languages.

μKanren implements First-order Horn clause (fohc) [1]. The grammar of *Horn clause* is shown in Fig 1. We say μKanren is *first-order*, as its unification algorithm identifies only structural equivalence. As an example that illustrates the correspondence between μKanren definitions and fohc formulas, consider the relation `append`^{o}.

```
(defrel (appendᵒ xs ys zs)
  (condᵉ
    [(≡ nil xs) (≡ ys zs)]
    [(fresh (a d r)
       (≡ `(,a . ,d) xs)
       (appendᵒ d ys r)
       (≡ `(,a . ,r) zs))]))
```

Authors' address: Weixi Ma, mvc@iu.edu; Kuang-chen Lu, kl13@iu.edu; Daniel P. Friedman, dfried00@gmail.com, Indiana University, Bloomington, IN.

XXXX-XXXX/2020/7-ART $15.00
https://doi.org/10.1145/1122445.1122456

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

Fig. 1. Horn Clause Formulas

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

Fig. 2. Hereditary Harrop Formulas

D formulas of fohc. In μKanren, a `defrel` introduces a *D formula*. For example, the append^o definition corresponds to this *D formula*,

$$\begin{aligned} \forall xs\, \forall ys\, \forall zs \quad & (\equiv\ xs\ nil) \wedge (\equiv\ ys\ zs) \\ & \vee\ \exists a\, \exists d\, \exists r\, (\equiv\ xs\ `(, a., d)) \wedge (append^o\ d\ ys\ r) \wedge (\equiv\ `(, a., r)\ zs) \\ & \supset (append^o\ xs\ ys\ zs). \end{aligned}$$

Here append^o and ≡ both build atomic formulas. For example, $(append^o\ xs\ ys\ zs)$ and $(\equiv\ xs\ nil)$ are atomic formulas.

G formulas of fohc. In μKanren, a `run` query contains a *G formula*, e.g.,

```
(run 1
  (fresh (xs)
    (appendᵒ xs `(1 2) `(1 2))))
```

is formulated as

$$\exists xs\ (append^o\ xs\ `(1\,2)\ `(1\,2)).$$

Formulas of hohh. λProlog implements a more expressive logic, *higher-order hereditary Harrop formulas* (hohh) [6]. Shown in Figure 2, Hereditary Harrop formulas extend G formulas with implicational goals and forall-quantification. Also, with higher-order unification, the unification algorithm of λProlog identifies $\alpha\beta$-equivalence between binding structures (that are absent in μKanren).

This paper presents λKanren. λKanren implements implicational goals and forall-quantification with two new operators, `assume-rel` and `all`, respectively. Also, λKanren incorporates higher-order pattern unification [4] for the binding structures (that are created by another two new operators, `tie` and app).

The rest of this paper demonstrates the uses of these four operators and their implementation details when appropriate. Our implementation of λKanren is available at https://github.com/mvcccccc/MK2020.

2 HIGHER-ORDER UNIFICATION

This section shows the power of higher-order pattern unification. By adapting Miller [4]'s unification algorithm, λKanren is equipped with two new operators: `tie` and app. `tie` expressions are abstractions and app is the shorthand for application. The ≡ operator in λKanren identifies $\alpha\beta$-equivalence between terms that involve `tie` and app.

Consider the following example that demonstrates α-equivalence. This example, metaphorically, tests the equivalence between `(λ (a b) (a b))` and `(λ (x y) (x y))`.

```
> (run 1 q
    (== (tie (a b) (app a b))
        (tie (x y) (app x y))))
'(_0)
```

`tie` is implemented as the following macro. It takes a list of variable names and a term. It then creates a `Tie` structure that is internally used for curried binders. Hereafter, we call a variable that is introduced by `fresh` a *unification variable* and a variable that is introduced by `tie` a *binding variable*.

```
(define-syntax tie
  (syntax-rules ()
    [(_ () t) body]
    [(_ (x0 x ...) body)
     (let ([x0 (Var 'x0)])
       (Tie x0 (tie (x ...) body)))]))
```

app is implemented as the following macro that elaborates a list of terms to an App structure that is internally used for curried applications.

```
(define-syntax app
  (syntax-rules ()
    [(_ rator rand) (App rator rand)]
    [(_ rator rand0 rand ...)
     (app (App rator rand0) rand ...)]))
```

Next, consider the following example that queries for two instantiations of `f`. This example demonstrates (1) β-conversions during unification and (2) how binding structures are reified.

```
> (run 2 f
    (== (tie (a b) (app a b))
        (tie (x y) (app f x y))))
'((tie (_0) (tie (_1) (app _0 _1))))
```

There is only one instantiation: `f` is a function (a `Tie` structure) of two inputs and `f` outputs an application form (a App structure) that applies its first input on the second one.

The internal structures, `Tie` and App, are reified as tagged lists. These tagged lists reflect their corresponded user interfaces, `tie` and app. During reification, binding variables and unification variables are both converted to underscore-digit symbols.

The power of higher-order unification, however, comes in with limits. To ensure decidability, β-conversion in Miller [4]'s algorithm restricts application forms: when the operator of an app is a unification variable, its operands must be distinct binding variables, otherwise unification fails. For example, the following query has no solution because the operands, the two bs, of the unification variable `f` are not distinct. In this case, with `f` being a function of two input bs, we cannot decide which b takes control.

```
> (run 1 f
    (== (tie (a) a)
        (tie (b) (app f b b))))
'()
```

To enforce this restriction, Miller [4]'s algorithm imposes another restriction on variable scopes: the instantiation of a unification variable may only contain its *visible* binding variables. A binding variable `x` is visible to a unification variable `q` if the introduction of `x` lexically precedes that of `q`. Given the following example, it seems that `q` can be instantiated by `y`. Unfortunately, `y` is not visible to `q` and the query has no solution.

```
> (run 1 q
    (== (tie (a b) (app a b))
        (tie (x y) (app x q))))
'()
```

3 IMPLICATIONAL GOALS

This section introduces the `assume-rel` operator that implements implicational goals ($D \supset G$). An `assume-rel` operator takes two inputs: (1) the hypothesis in the form of a D formula and (2) the goal in the form of a G formula. The `assume-rel` operator then uses the hypothesis as a fact and moves on to the goal.

Implementing `assume-rel` is subtle with shallow embedding. Because the definitions of λKanren are kept in the run-time environment of its host language, extending these definitions requires updating the run-time environment. This problem is illustrated in the following example, liberally adapted from Miller and Nadathur [5, p. 80].

```
(defrel (taken name class)
  (conde
    [(== 'Josh name) (== 'B521 class)]
    [(== 'Josh name) (== 'B522 class)]))
(defrel (pl-major name)
  (taken name 'B521)
  (taken name 'B523)
  (taken name 'B522))
```

One may complete `pl-major` after taking three classes: `B521`, `B522`, and `B523`. And Josh currently has `taken` `B521` and `B522`. In the following query, the `assume-rel` operator extends the definition of `taken` with `(taken 'Josh q)` and then moves on to the goal `(pl-major 'Josh)`.

```
> (run 1 q
(assume-rel [(taken name class)
             (== 'Josh name)
             (== 'q class)]
  (pl-major 'Josh)))
'(B523)
```

From the implementation aspect, because the host language is lexically scoped, the definition of `pl-major` is fixed. This means that, the free variable in the definition of `pl-major`, namely `taken`, always uses the original definition of Josh taking `B521` and `B522`. To extend definitions on the fly, we need to create dynamic scope so that the free variables may use the latest, updated definitions.

Our approach is to add an extra layer between λKanren and the host language (Racket). This extra layer redirects function definitions.

We introduce two global maps, `name->idx` and `idx->def`. Each `defrel` extends these two maps by creating a new index, putting the name-idx pair and the idx-def pair in the two maps respectively. The `idx->def` map is global. And the `name->idx` map is threaded through during the execution of a query (an invocation of a `run`).

To invoke a definition, one follows `name->idx` and `idx->def`, i.e., first retrieving the index using `name->idx` and then getting the definition using `idx->def`. For example, the user interface

```
(pl-major 'Josh)
```

is macro-expanded to

```
((cdr (assv idx->def (cdr (assv name->idx 'pl-major))))
 'Josh).
```

When an `assume-rel` operator is invoked, the two maps are extended again: (1) a new index is created; (2) `idx->def` contains the pair of the new index and the extended function; and (3) `name->idx` now has a new pair of the definition name and the new index, this new pair shadows the previous one.

In the previous example, let's use t_1 for the `taken` definition that knows Josh has taken `B521` and `B522`, use t_2 for the extended `taken` (where we `assume-rel` Josh has taken `B523`), and use `p` for the definition of `pl-major`. With `taken` and `pl-major` first defined, `name->idx` is `((pl-major . 2) (taken . 1))` and `idx->body` is `((2 . p) (1 . t1))`. Then, after assuming `(taken 'Josh q)`, the query `(pl-major 'Josh)` runs in an updated environment where `name->idx` is `((taken . 3) (pl-major . 2) (taken . 1))` and `idx->body` is `((3 . t2) (2 . p) (1 . t1))`. The more recent pair in `name->idx` shadows the previous one. And therefore, when `taken` is invoked, we use t_2.

Many interesting examples only make hypothesis on atomic formulas. And thus we provide the `assume` operator that is a shorter version of the `assume-rel` operator. Instead of any D formula, the `assume` operator only takes an atomic hypothesis.

As an example, we define the `eq` relation to be reflexive, transitive, and symmetric as follows.

```
(defrel (eq x y)
  (cond^e
    [(== x y)]
    [(fresh (z)
       (eq x z)
       (eq z y))]
    [(eq y x)]))
```

Obviously `apple` and `orange` are by no means `eq`. In fact, the following query does not terminate in a naive μKanren implementation because the third cond^e line is very recursive.

```
> (run 1 q
    (eq 'apple 'orange))
```

Using `assume`, we may temporarily extend the definition of `eq` as follows.

```
> (run 5 q
    (assume (eq 'orange 'apple)
      (assume (eq 'orange 'dog)
        (eq 'orange q))))
'(dog apple orange orange dog)
```

Because λKanren runs backward, as in the following, the hypothesis can be inferred as well.

```
> (run 1 q
    (assume (eq 'orange q)
      (eq 'apple 'orange))))
'(apple)
```

4 FORALL-QUANTIFICATION

This section introduces the `all` operator ($\forall x G$) that takes a list of symbols and a goal. These symbols are used to create special variables that are virtually constants (eigenvariables).

Continuing with `taken` and `pl-major`, we create a random person `x` using the `all` operator.

```
> (run 1 q
    (all (x)
      (assume-rel (taken x 'B521)
        (assume-rel (taken x 'B522)
          (assume-rel (taken x 'B523)
            (pl-major x))))))
'(_0)
```

Like the `fresh` operator, the `all` operator creates a new variable in the scope. Unlike the `fresh` operator, the `all` operator effectively creates a constant. This semantics is similar to the proof technique of a for-all goal in first-order logic: to prove $\forall x.P$, we fix a constant x and then prove P.

Consider the next example that synthesizes the identity function using the `all` operator.

```
> (run 1 f
    (all (x)
      (== x (app f x))))
'((tie (_0) _0))
```

The implementation of the `all` operator follows that of the `fresh` operator, except that the created variable is a constant. In our implementation, we create an `all` variable as a free binding variable. Thus, the `all` variable cannot be unified with anything but itself.

5 λKANREN AS A THEOREM PROVER

λProlog is often regarded as a proof system. With hohh, λKanren suits a theorem prover as well. This section shows examples that use λKanren to prove intuitionistic style theorems.

We start with the definition of `proved`. At this moment, only `'trivial` is proved.

```
(defrel (proved x)
   (== x 'trivial))
```

Obviously, not everything is proved.

```
> (run 1 g
    (all (p)
      (proved p)))
'()
> (run 1 g
    (proved g))
'(trivial)
```

Next, we prove the commutativity of conjuction. I.e., $\forall p, q\ (p \wedge q) \supset (q \wedge p)$.

```
> (run 1 g
    (all (p q)
      (assume ((proved p) (proved q))
        (proved q)
        (proved p))))
'(_0)
```

The introduction rule of disjunction can be proved: $\forall p, q\ p \supset (p \vee q)$

```
  (run 1 goal
    (all (p q)
      (assume ((proved p))
        (conde
          [(proved p)]
          [(proved q)]))))
'(_0)
```

6 CONCLUSION

λKanren is based on higher-order hereditary Harrop formulas. It extends μKanren with four operators, `tie`, `app`, `assume-rel`, and `all`. In addition, unification (`==`) identifies $\alpha\beta$-equivalence between the binding operators.

Our implementation of λKanren is written in Racket by adding about 40 lines to μKanren. Overall, we appreciate the simplicity provided by the shallow embedding techniques.

REFERENCES

[1] Krzysztof R. Apt and M. H. van Emden. Contributions to the Theory of Logic Programming. *Journal of the ACM (JACM)*, 29(3):841–862, July 1982. ISSN 0004-5411. doi: 10.1145/322326.322339. URL https://doi.org/10.1145/322326.322339.

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

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

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

[5] Dale Miller and Gopalan Nadathur. *Programming with Higher-Order Logic*. Cambridge University Press, USA, 1st edition, 2012. ISBN 978-0-521-87940-8.

[6] Dale Miller, Gopalan Nadathur, and Andre Scedrov. HEREDITARY HARROP FORMULAS AND UNIFORM PROOF SYSTEMS. *Unknown Host Publication Title*, pages 98–105, January 1987. URL https://experts.umn.edu/en/publications/hereditary-harrop-formulas-and-uniform-proof-systems. Publisher: IEEE.

[7] Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51(1):125–157, March 1991. ISSN 0168-0072. doi: 10.1016/0168-0072(91)90068-W. URL http://www.sciencedirect.com/science/article/pii/016800729190068W.