microKanren with Delayed Goals

MICROKANREN WITH DELAYED GOALS

Laura Zharmukhametova

A Thesis

Submitted to the
Department of Computer Science and the Department of Mathematics
In partial fulfillment of the requirement
For the degree of
Bachelor of Arts
at
Harvard University
March 26, 2021

Acknowledgements

Professor Nada Amin of Harvard University and William Byrd, PhD, of Hugh Kaul Precision Medicine Institute at the University of Alabama, Birmingham guided me through this project. They gave me a rigorous introduction to relational programming, and their insightful ideas and feedback were essential for the completion of this thesis.

Abstract

miniKanren is a relational programming language embedded in Scheme. This paper studies the problem of supporting delayed goals in the context of μKanren, the core of miniKanren. It explores different representations of delayed goals and tradeoffs between them. In the presented approach, delayed goals are both accumulated and dispatched throughout the execution of a program, and the ones that remain at the end are returned to the programmer as a continuation. A novel interface, inspired by the concept of engines, allows continuously supplying μKanren code to partially computed program outputs.

Table of Contents

Abstract ..... **iii**
Chapter 1: Introduction **1**
Chapter 2: μKanren and Delayed Goals **2**
2.1 Motivation 2
2.2 Challenges and Tradeoffs 6
2.3 Delayed Goals and Streams 13
2.4 Partial Computation and Continuations 16
Chapter 3: Future work **24**
Chapter 4: Conclusion **25**
References **26**
Appendix **28**

Chapter 1
Introduction

μKanren is a relational programming language which is a basic version of miniKanren [1, 2]. It has the core functionality of miniKanren, but its implementation is significantly simpler, making it a great starting point for introducing a new feature to the miniKanren family of languages.

We explore the problem of extending μKanren with delayed goals. A delayed goal is, roughly, a normal μKanren goal that requires more ground data to be executed. Certain useful operations such as arithmetic and trigonometric functions can be easily computed forward using the built-in Scheme procedures. A common problem for a miniKanren programmer is to define the relational versions of these operations. Defining such relations can be a meticulous task, as it is normally not enough to have access to the built-in procedures. The issue is that we no longer have a fixed amount of available data, so, for example, inverting multiplication with division could still leave some arguments unknown. In the existing implementations of delayed goals, this lack of concrete arguments leads to an error, unless enough variables become ground eventually. We designed a delayed goals mechanism that eliminates this kind of error via continuations. The implementation presented in this work can potentially be adopted for the full-featured miniKanren.

The contribution of this paper is an implementation of delayed goals that does not treat inexecutable delayed goals as errors. The concepts of continuations and partial evaluation were used to create an interface for continuously providing more data to a stream of results that was only partially computed due to a lack of ground variables.

Chapter 2 begins with a use case for delayed goals in μKanren and motivation for modifying the interface. Example programs involving delayed goals are discussed to illustrate the different models developed and assess design choices. The design and implementation of delayed goals are presented in Section 2.2. Section 2.4 provides several interface variants and explores the tradeoffs between them. Chapter 3 is a discussion of future work, with a focus on how delayed goals and the presented interface could be used for staged programming. The appendix provides an overview of μKanren and its features.

Chapter 2
μKanren and Delayed Goals

2.1 Motivation

To demonstrate a use case of supporting delayed goals in miniKanren, consider the `pluso` relation. The terms `x, y, z` satisfy the goal `(pluso x y z)` if `x + y = z`. This relation can be implemented in μKanren without delayed goals, but it involves several non-trivial definitions. First, numbers are built as lists of bits [3]:

```
(define build-num
  (lambda (n)
    (cond
      ((odd? n)
       (cons 1
         (build-num (quotient (- n 1) 2))))
      ((and (not (zero? n)) (even? n))
       (cons 0
         (build-num (quotient n 2))))
      ((zero? n) '()))))
```

Then we need `zeroo`, `poso` and `>1o` which are relations for checking that a number represented this way is zero, positive and non-negative respectively. Then, we need the procedure `addero` [3]:

```
(define addero
  (lambda (d n m r)
    (conde
      ((== 0 d) (== '() m) (== n r))
      ((== 0 d) (== '() n) (== m r)
       (poso m))
      ((== 1 d) (== '() m)
       (addero 0 n '(1) r))
      ((== 1 d) (== '() n) (poso m)
       (addero 0 '(1) m r))
      ((== '(1) n) (== '(1) m)
       (fresh (a c)
```

```
    (== `(,a ,c) r)
    (full-addero d 1 1 a c)))
  ((== '(1) n) (gen-addero d n m r))
  ((== '(1) m) (>1o n) (>1o r)
   (addero d '(1) n r))
  ((>1o n) (gen-addero d n m r)))))
```

Further, `addero` itself relies on two other procedures `full-addero` and `gen-addero`. Our relation `pluso` is then defined as a special case of `addero`:

```
(define pluso
  (lambda (n m k)
    (addero 0 n m k)))
```

Defined this way, the goal `pluso` is purely relational. It does not use the built-in + operator in Scheme, and we can safely run it if two or three variables `x`, `y` and `z` are still fresh. However, if a number is computed using this `pluso`, it is either the arithmetic sum or difference of the other two. We would like to take advantage of this fact and try to make the `pluso` implementation more concise. In particular, we wish to define `pluso` in terms of the non-relational Scheme + and - procedures. An outcome of the present project is an alternative, shorter definition of `pluso` that, despite not being purely relational, allows a safe use of the one-way addition and subtraction procedures.

What we gain from being able to use the built-in Scheme procedures within μKanren is not only a simpler implementation of arithmetic relations:

- In the purely relational arithmetic, we have to use the special list-like numerals, not the conventional Arabic numerals.
- Running the operations forward is less efficient than calling the built-in procedures due to the vast number of unifications involved.
- This arithmetic is not applicable to probabilistic programming, since operations like sine, logarithm, and floating-point point arithmetic are too complex for it.
- Constraint logic programming makes use of external solvers such as Z3 to outsource these operations. However, if we need to compute $e^{1/2}$, why not simply call the built-in Scheme procedure? The challenge is to stay relational and be able to place logic variables in arbitrary parts of the expression, so that e^x does not lead to an error.

- The miniKanren interpreter can be extended with the purely relational arithmetic, but the expressions need to be tagged. With delayed goals, we can avoid tagging, thus improving usability.
- Delayed goals could be used to stage the relational interpreter, by treating the environment as unavailable data.
- Being able to use such built-in operations as `string->symbol`, `print`, `length`, etc, code gives us power to write more kinds of programs, e.g. one can build a reifier inside miniKanren.

In the present extension of μKanren, arbitrary built-in Scheme procedures can be injected to μKanren code, without introducing a new type of error. This is different from the existing approaches, where we an error is raised unless all delayed goals are run. The interface and behavior of μKanren were modified so that the following definition is allowed due to the delayed goals mechanism:

```
(define (pluso-2 x y z)
  (lambda (c: S D C)
    (let ((a (walk* x S))
          (b (walk* y S))
          (c (walk* z S)))
    ((cond
       ((and (number? a) (number? b)) (== (+ a b) c))
       ((and (number? a) (number? c)) (== (- c a) b))
       ((and (number? b) (number? c)) (== (- c b) a)))
     c))))
```

Here we check if `x`, `y` or `z` are ground and project to numbers, and if at least two of them are, we unify the third with their sum or difference. While it is shorter than the first definition, since we are not using `addero`, such a goal is not safe to run in the original μKanren. A least two variables need to be ground and bound to a number. Furthermore, whether or not it is safe to run it depends on both the previous and the future goals. It is possible that it is unsafe to run `pluso-2` now, but a future unification makes enough variables ground. One solution is to wrap such an unsafe goal in a "delayed goal" and postpone it until it becomes safe.

With the delayed goal mechanism developed in this project, we can safely feed the `pluso-2` goal to `run*` and place it in an arbitrary position with the other goals. When the other goals are computed, we either have enough information for `pluso-2`, or more

variables need to be grounded. In the former case, we run `pluso-2` and get a stream, as we would in the original μKanren. Otherwise, the result is a continuation represented by a list of the variables that are involved in the `pluso-2` goal and a procedure. This procedure takes more μKanren code and runs it on the partially computed answer.

Here is a program that will delay `pluso-2` and return a continuation:

```
(run* (q)
   (fresh (x y)
     (== q `(,x . ,y))
     (maybe-delay ground-2? `(,x ,y 2) pluso-2)))
```

The result is a unit of one continuation: the pair of fresh variables `(y!2 x!1)` and a procedure, which we call `f` below. We can then feed the goal `(== x!1 1)` to `f`, making the `x` variable ground.

```
 (let* ((res (run* (q)
   (fresh (x y)
     (== q `(,x . ,y))
     (maybe-delay ground-2? `(,x ,y 2) pluso-2))))
     (res0 (car res))
     (vars (K->vars res0))
     (f (K->f res0)))
   (cons vars (f '(== x!1 1))))
```

Two of the arguments to `pluso-2` are now ground, so the third one `y` is inferred to be 1. The result of the above program is then `'((y!2 x!1) (1 . 1))`.

It is possible that the continuation returns, among the stream of answers, another continuation, in which case we can provide more μKanren code. For example, the program

```
 (let* ((res (car (run* (q)
    (fresh (x y z)
      (== q `(,x . ,y))
      (maybe-delay ground-2? `(,x ,y ,z) pluso-2)))))
   (f (K->f res))
   (res^ (car (krun (== x!1 1) f)))
   (f^ (K->f res^)))
   (list (K->vars res) (K->vars res^) (krun (== z!3 5) k^)))
```

yields the list `'((z!3 y!2 x!1) (z!3 y!2) ((1 . 4)))`. After the first `run` there are three variables that have a delayed goal associated with them, namely `x`, `y` and `z`. Hence all three of them are listed in the `vars` field of `res`. After the second run, `x` becomes ground, and unifying it with a value will not facilitate the progress of the `pluso-2` goal. (Every time we need `x` we walk it in the substitution map, so it is effectively 1.) Since there are still two fresh variables, namely `y` and `z`, the second result `res^` is still a continuation. Finally, adding the goal `(== z!3 5)` triggers the delayed `pluso-2` goal and the result is a stream with one element, `(1 . 4)`.

2.2 Challenges and Tradeoffs

One issue with returning a continuation instead of an answer is that the programmer needs to know which variables are involved in the delayed goals and hence need to be grounded to complete the computation. We cannot simply refer to the lexical variables in the original program. Firstly, there is variable shadowing and the possibility of using the same variable name in different parts of the program, such as the branches of a `conde`. This can be addressed with the help of α-renaming. That is, the program that is given to `run` is transformed to an equivalent program where every variable has a unique name. Thus, the program

```
(fresh (x y z)
  (fresh (x)
    (== x y)))
```

is renamed to

```
(fresh (x y z)
  (fresh (x.0)
    (== x.0 y)))
```

However, this does not entirely solve the problem, as we might have fresh variables created in recursive goals. Consider the definition of `appendo`:

```
(define appendo
  (lambda (l s out)
    (conde
      [(== '() l) (== s out)]
      [(fresh (a d res)
         (== `(,a . ,d) l)
```

```
        (== `(,a . ,res) out)
        (appendo d s res))])))
```

Note that α-renaming will not change the program as there are no repeating identifiers for variables. However, calling `appendo` with a list `l` of 5 elements will create 5 different variables all corresponding to the identifier `a`. These 5 variables might have distinct delayed goals associated with them, and we would like the programmer to be able to differentiate between them.

To remedy this, in addition to α-renaming we append an index to the reified variable. This index is acquired through tracing and indicates the "depth" of the variable. It gives one an idea of where in the execution of the μKanren program this variable was created. For example, an identifier `x` can be first α-renamed to `x.0`, and a corresponding variable can then be reified to `x.0!3`.

The process of going from a variable back to a symbol is done through a modified reification function. In the original implementation, the identifier is forgotten once the variable, i.e. a vector with a unique index, is created. Now in addition to the index the vector holds the identifier. To reify the variable, if it is still fresh, we simply retrieve this symbol and append `!i` to it, where `i` is the index.

In the reification process, an important step is creating a map from variables to their reified names. We take advantage of this map in the subsequent `run`'s. In particular, the subsequent `run`'s take goals involving reified variables from previous `run`'s, and they are not automatically bound. However, swapping the pairs in the reification map gives a map from symbols to variables, which we then use for variable binding. Note that if the original map was not surjective, we may pick any variable as the inverse, because all variables in the pre-image of a symbol are unified.

The following example demonstrates how the variables are reified in recursive programs and can be referred to in the subsequent `run`'s. Consider the `sum-o` relation that unifies `out` with the sum of the elements of the list `l`:

```
(define sum-o
  (lambda (l out)
    (conde
      [(== '() l) (== 0 out)]
      [(fresh (a d res)
         (== `(,a . ,d) l)
         (maybe-delay ground-2? `(,a ,res ,out) pluso-2)
         (sum-o d res))])))
```

Suppose we wish to compute

```
(res (run* (q)
     (fresh (x y z w u)
       (== q `(,x ,y ,z ,w ,u))
       (sum-o `(,x ,y ,z ,w) u))))
```

Parts of the code, such as the creation of multiple sets of `a, d` and `res` variables and the `(== `(,a . ,d) l)` goal, run immediately. However, since not enough variables are ground, the `pluso-2` goal is delayed. We get a continuation `k` and a list of reified variables `(res!14 w!4 res!11 z!3 res!8 y!2 u!5 x!1)`. We can then provide more code, for example:

```
(krun
  (fresh ()
    (== y!2 3)
    (== z!3 2))
  k)
```

yields another continuation `k-1`, this time with a shorter list of variables `(res!14 w!4 res!11 res!8 u!5 x!1)`. Lastly, running

```
(krun (fresh ()
       (== x!1 1)
       (== u!5 10))
     k-1)
```

We obtain the result, `(1 3 2 4 10)`. Note that the variables continue to have a single `!` index after each `krun`. This is so because they continue to be the same variable and they do not acquire a new index when we bind the new goals to the old. There is an aspect of this approach that needs to be clarified. Suppose we have a `conde` of two branches, both of which are identical. For instance, suppose they both call appendo on the same set of fresh variables:

```
(fresh (x y l)
  (conde
    ((appendo (list x) (list y) l))
    ((appendo (list x) (list y) l))))
```

The `x, y` and `l` variables will have unique reified names, however, the "hidden" fresh

variables such as `res` in the definition of `appendo` are no longer unique, as they will have the same index and symbol, but belong to two different states. A goal like `(== res!3 11)` will be applied to both states, and there is no way to tell `krun` to apply it to one `conde` branch but not the other. Even if we somehow give the two variables distinct indices during tracing, telling the branches apart without α-renaming the fresh variables inside `appendo` would be difficult. The problem is the hidden fresh variables do not undergo α-renaming like the rest of the program. One solution is to "deep" α-rename `appendo` to differentiate the fresh `a, d` and `res` variables on different branches. Alternatively, we could make the restriction that the programmer cannot touch the hidden variables in the subsequent `krun`'s.

To implement delayed goals, we modify the state in μKanren as follows. Originally, a state consisted of a substitution map `S` and a counter `C`, and we add a third store `D`, holding the delayed goals.

A delayed goal is a triple consisting of a predicate `pr`, a list of arguments `args`, and a goal constructor `gc` (a relation). We can create a delayed goal via the `maybe-delay` mechanism, which can be thought of as taking an unsafe goal and making it a safe, delayed goal. Thus, if `(pluso-2 x y z)` is an unsafe goal, `(maybe-delay ground-? ‘(,x ,y ,z) pluso-2)` is a safe equivalent of it. It first checks if the predicate applied to the arguments is true. If so, the goal contructor is applied to the arguments to yield a goal, which we then apply to the state. Otherwise, the `pluso-2` goal is pushed to the `D` store of the state, and it will only be dispatched when it is safe to do so (i.e. when at least two of the variables are ground). This "safeness" check is performed by applying the predicate `ground-2?` to the list of arguments `‘(,x ,y ,z)` after we `walk*` it in the current substitution map.

Once we have a list of delayed goals in the `D` store, we need to be able to selectively check which of them become safe as the execution of the goals progresses. A good place to make these checks is during the unifications via `==`. During an `==` procedure, we know precisely which variables are involved. Conversely, every time a term becomes more ground, it must happen via the `==` operator. One approach would be to perform the safeness checks for every goal in the `D` store every time there is a unification. However, this would not be very efficient because most goals do not even involve the variable being unified. Instead, the `D` store could keep a map from variables to lists of delayed goals. Every time `x` and `y` are unified, we look up the associated delayed goals and only run the predicates of those goals. But now we have another issue: different variables can be involved in different delayed goals. When two variables are unified, they effectively become the same variable, so their delayed goals ought to be merged. To achieve a well-

defined behavior, we make the following observation. In a substitution map, represented in μKanren as a list of pairs, every key has a unique `walk` value, which is guaranteed by the unification algorithm. This means that if we have a set A of variables where each variable is unified with some other variable in the set, there exists a unique variable which is a representative of this set and can be reached from any other variable in A. Namely, it is the result of `walk` applied to any element of A. For example, the result of applying the goal

```
(fresh (x y z u w)
  (== x y)
  (== u w)
  (== u z)
  (== y z))
```

to the empty state is the state

```
((((#(1 y) . #(2 z))
   (#(4 w) . #(2 z))
   (#(3 u) . #(4 w))
   (#(0 x) . #(1 y)))
  (((#(2 z))) () . 0)
  .
  5))
```

The unique representative reachable from any variable here is `z`. This is used in the delayed goals mechanism as follows. Of all variables in A, only `z` holds the delayed goals. Whenever a new variable `v` is added to the set A through ==, it either becomes the new unique representative, or we can reach the old representative by calling `walk* v s`, where `s` is the substitution map. In either case, we merge the delayed goals of the old representative and those of `v` and update the map in the `D` store. The delayed goals of A are now all attached to the new representative. We can then perform the safeness checks by accessing these merged goals.

An important question to be addressed at this stage is how to deal with delayed goals that rely on each other. We may have a list of goals such that, in a given state, only one of them is safe to run. However, as a result of running that goal, some of the remaining ones might become safe, which themselves can make more goals ready to run. Moreover, a delayed goal can yield more delayed goals.

To handle the first challenge, we partition the list of goals associated with a vari-

able being unified. The safe goals are wrapped in a single goal `g` using `conj+`, and the unsafe goals are put back into the state before `g` is applied to it. For example, suppose we wish to apply the goal `(== x y)` to the state `c`. In the original μKanren, the result of a successful unification is a stream with one element `c'`, where `c'` is `c` with an updated substitution map. We modify it so that the result is the goal `g` applied to `c''`, where `c''` is `c'` with an updated delayed goals store. The state `c''` holds the unsafe goals of our partition, so these are passed to the safe goals, as necessary. The case of safe goals yielding more delayed goals is automatically handled because the safe delayed goals are treated as normal goals. Note that `(== x y)` does have the usual behavior if there are no safe goals to be run, except the returned stream of one state potentially has unfulfilled delayed goals.

There are multiple options regarding what can be done with the delayed goals that are still unfulfilled by the end of the program. One option is to return an error, but this is not very interesting. Another approach is to return a continuation of the following form. We list the variables that need to be grounded and provide a procedure `k`. The programmer needs to give the procedure a map which binds some of the listed variables to values. The function `k` then unifies these variables with the provided values. Specifically, `k` holds a stream it has built so far, wraps the new unifications in a single goal `g`, and applies it to the stream by invoking `(bind g $ )`. The result can be another continuation or a stream. We can keep feeding more values until a stream is returned. We have implemented this approach, however, it is not much more interesting than the first one. We would like to be able to add goals involving the variables with unfulfilled delayed goals. A third option, which has been discussed earlier and which we adopted, provides this flexibility. Specifically, the result is a list of answers and continuations. A continuation is represented by a list of the variables that are involved in the unfulfilled delayed goals along a certain path, and a procedure that holds a delayed state. This procedure takes more μKanren code and applies it to the state, resulting in another stream which can contain more fully computed answers.

Another challenge is representing the delayed goals in a given state. There are two design choices to be made: representing the goal itself and listing the goals in the μKanren state.

We distinguished the following three approaches to representing a single goal: procedural, textual, and combined. A textural representation is somewhat similar to staging. A delayed goal could be represented as an expression that is to be evaluated later, along with a predicate for testing whether it is safe to evaluate the goal. If the result of a `run` program contains unfulfilled delayed goals, they can be returned to the user as an explicit expression. This approach has an advantage that one can inspect the delayed goals

and understand exactly what kind of information is missing and which part of the program has not been computed. However, implementing it would require addressing some lexical scope issues. For example, in the `pluso` example, either we would need to pass the definitions of `pluso` and `ground-2?` to the `run` command, or these functions themselves would have to hold their own textual representations. In either case, we would need to have a mechanism for capturing the environment and lexical scope. Then it is not clear how such an environment would fit in as a relational programming construct.

The procedural approach is less invasive in the sense that we do not need to re-engineer the interface and implementation of μKanren too much. However, the result of a `run` expression, if it is a continuation, is less informative. This was mitigated by providing α-renamed and traced variables along with the continuation. Note that this solution requires more mental work from the programmer to figure out the contents of the continuation itself. A delayed goal is represented as a triple consisting of a predicate, an argument list, and a goal constructor. The goal constructor is a procedure that does not remember the expression it was derived from. Thus, on the one hand, we lose this convenient textual information. On the other hand, the lexical scope issue is automatically resolved.

Our implementation follows the second approach, but one could also try to combine the two to have the benefits of both. A continuation could be a procedure accompanied with a textual representation for inspection purposes.

Let us now address the second question, namely, how to keep track of the delayed goals. The first step was to extend the state from holding a substitution map `S` and a counter `C` to a triple `S, D` and `C`, where `D` is the delayed goals store. Now structuring the store `D` is tricky. A straightforward representation would be simply listing the delayed goals in the designated store. However, this introduces a great inefficiency as every goal needs to be checked every time there is a unification. Most of these checks are likely trivial as they do not involve the variables being unified.

Another way to represent the delayed goals store is by storing a map `m` from sets of variables to sets of delayed goals. If the triple `{x, y, z}` is associated with the set `{dg0, dg2}` where `dg0, dg1` are some delayed goals, then grounding any of the three variables can trigger the goals `dg0` and `dg1`. The delayed goal would be a pair of two procedures, the predicate `pr` and the goal constructor `gc`. This would ensure that we do not have duplicate delayed goals, hence when a goal is executed, we do not need to worry about removing all occurences of the goal in the store `D`. There is a tradeoff: suppose a fresh variable, say `x`, enters a unification. Then we need to find every occurrence of its old and new representatives in a key of the map `m` and check the corresponding sets of

goals. Moreover, it is necessary to carry over the remaining unsafe delayed goals of `x` to the new representative `r` of the variables `x` is unified with. For the sake of consistency, all occurrences of `x` in the keys of `m` ought to be replaced with `r`. As a result, some keys might become identical, so their values, i.e. sets of delayed goals, ought to be merged. Even though this approach appears simple at first glance, ensuring consistency and correctness would require some meticulous work.

A third approach to representing the store `D` is to make `m` map individual variables to sets of goals. We let `dg = (pr, args, gc)` be a delayed goal depending on the fresh variables `x, y, z`. Then each of these variables would map to a set containing `dg`. This certainly makes looking up the delayed goals associated with a given variable easier. However, we now need to duplicate the delayed goal `dg` and put copies of it in three different sets. Whenever this goal is dispatched as a result of a unification of one of the three variables, the remaining duplicates must be removed. To remedy this, we enumerate all the delayed goals in a given state using a counter, so that the map `m` sends variables to sets of indices. Whenever a delayed goal is executed, we remove it from the store `D` entirely (from the enumeration and the sets of indices in `m`). Whenever `x` undergoes a unification, we simply remove the corresponding binding in `m` and merge the indices of delayed goals of `x` with those of the new representative variable, if there is any. Otherwise, if `x` becomes ground, we simply remove its binding in `m` and run the safe goals of `x`. We can discard the indices of the unsafe goals of `x` in this case because `x` no longer plays a role in whether the predicate checks succeed. If it is still possible to fulfil these goals, then their indices appear somewhere else in the map `m`. Otherwise, an unfulfilled unsafe goal remains in the store with no variables mapping to it. This would indicate the delayed goal was ill-defined, as the predicate should always succeed if enough entries of `args` are ground.

2.3 Delayed Goals and Streams

When a goal is applied to a state, the result is a stream of states. A stream can be the empty list, a thunk, or a pair whose `car` is a state and whose `cdr` is a stream. The delayed goals mechanism is implemented by modifying the definition of the goal constructor ==. We wish to keep the input and the output format of == intact. In particular, == takes two terms `u` and `v` and returns a goal, i.e. a function from states to streams. The following was the baseline definition of `(== u v)`:

```
(define (== u v)
  (lambda (c)
```

```
    (let ((s (unify u v (car c))))
      (if s (unit `(,s . ,(cdr c))) mzero))))
```

Let `c` be some state, represented by a map `S` and a counter `C`. The goal `(== u v)` takes this state and unifies `u` and `v` in the substitution map `S` using the standard unification algorithm. The unification algorithm returns either a new substitution map `S1` or the boolean `# f`, indicating failure. In the former case the result of `((== u v) c)` is the stream `(unit `(,S1, ,C))`, i.e. a list with a single element, the pair `(S1, C)`. In the latter case, the result is `mzero`, which is an alias for the empty stream `()`. In our extension with delayed goals, this definition is modified as follows.

```
(define (== u v)
  (lambda-c (c : S D C)
    (pmatch (pull-delayed u S D)
      (`(,goals-u . ,D-)
       (pmatch (pull-delayed v S D-)
    (`(,goals-v . ,D--)
     (let ((S (unify u v S)))
       (if S
      (let ((c `(,S ,D-- . ,C)))
        (dispatch-safe-goals (walk* u S) goals-u goals-v c))
      mzero))))))))
```

Here `(lambda-c (c : S D C) ...)` is a macro for disassembling the state `c`, so `S, D` and `C` are the substitution map, the delayed goals store, and the counter respectively. The `pmatch` is a pattern matcher. The `pull-delayed` line retrieves the binding of `(walk* u S)` in `D`. The result is the pair `goals-u` and `D-`, where `goals-u` is the set of indices corresponding to the delayed goals of `u`, and `D-` is `D` minus the goals of `u`. Then we do the same for `v`. Note that most of these steps will be effectively skipped if `u` or `v` is not a variable. (If they are lists then this process will apply to the variables inside them.) Lastly, the `(dispatch-safe-goals (walk* u S) goals-u goals-v c)` line is responsible for the handling the delayed goals accumulated so far:

```
(define (dispatch-safe-goals u u-dg v-dg c)
  ((lambda-c (c : S D C)
     ((lambda-D (D : i G C1)
       (let* ((u-safe/unsafe (partition (is-safe S G) u-dg))
         (v-safe/unsafe (partition (is-safe S G) v-dg))
```

```
        (safe (append (car u-safe/unsafe) (car v-safe/unsafe)))
        (unsafe (append (cdr u-safe/unsafe) (cdr v-safe/unsafe)))
        (i (if (var? u) `((,u . ,unsafe) . ,i) i))
        (D `(,i ,G . ,C1))
        (c `(,S ,D . ,C)))
    (apply* safe c))) D)) c))
```

The `(lambda-D (D : i G C1) ...)` macro is used for disassembling the `D` store of the state `c`. Specifically, `i` is the map sending variables to sets of indices, `G` is the enumeration of all delayed goals, and `C1` is the counter for the delayed goals.

Next, we partition the delayed goals of `u` and the delayed goals of `v`, namely `u-dg` and `v-dg`, into safe and unsafe goals. This is done by running a check on every delayed goal via the function `(is-safe S G)`:

```
(define (is-safe S G)
  (lambda (id)
    (pmatch (find-g id G)
   (`(,id ,pr ,args . ,gc)
    (pr (walk* args S)))
   (`#f #f))))
```

Here `id` is the index of the current delayed goal, which we then look up in the enumeration `G`. The predicate of this goal is then applied to `(walk* args S)` to determine if the goal is safe to run.

Returning to our definition of `dispatch-safe-goals`, we can now assemble the `safe` and `unsafe` lists. The map `i` is then updated to reflect that we have merged the goals of `u` and `v` into a single list, now attached to the representative of `u` (which also the representative of `v`). (Again, this only makes sense for variables.) The lines

```
        (D `(,i ,G . ,C1))
        (c `(,S ,D . ,C)))
```

are responsible for assembling a new state `c` where only the unsafe goals of `u` and `v` remain. Then, `(apply* safe c)` wraps the `safe` delayed goals into a single usual goal. This goal is in turn applied to the updated state `c`. The resulting stream is the value returned by `((== u v) c)`.

2.4 Partial Computation and Continuations

A major conceptual change introduced with the delayed goals is that the output of a `run` command can be viewed as a partially computed program. If it is fully computed, then all goals were eventually safe, and the result is the stream that would be computed in the usual μKanren. Running programs of the form `(run n (q) ...)` or `(run* (q) ...)` yields two types of outputs: a reified answer or a continuation. If an answer is returned, it means all delayed goals created during the `run` were fulfilled, and the reified answer provides a computed value for `q`. The second type of output is a continuation. It means there is not enough information to run certain delayed goals. The continuation provides a list of variables on which the delayed goals rely. One can feed more μKanren code to the continuation, and this additional code is allowed to have the listed variables as identifiers. The result of the continuation can itself be a stream or another continuation. If it is a stream, then it must be the stream resulting from an injection of the goals that were added later to the original goals. If we get a continuation again, it means more information is necessary. This process continues and can possibly never stop, but if a stream is returned at some point, it must be a result of combining all goals we have incrementally provided so far. There is no obvious way to combine the old and the new goals, mostly due to scoping issues, and we have explored several models for formatting the continuation in a way that would make this extension sound.

Firstly, the assumptions about what kind of goals can be applied to a stream of states need to be revisited. In μKanren, we have the `bind` operator that takes a goal `g` and a stream `$`, and makes a new stream by applying `g` to the elements of `$`:

```
(define (bind $ g)
  (cond
   ((null? $) mzero)
   ((procedure? $) (lambda () (bind ($) g)))
   (else (mplus (g (car $)) (bind (cdr $) g)))))
```

In the baseline μKanren, a goal is self-contained in the sense that any variable must be declared as part of a `(fresh (...) ...)` expression. There is no dependence on the existing variables of the state a goal is applied to, because the state is assumed to be arbitrary. A variable cannot be on its own: it is a vector with a single entry holding its unique index in a particular state. The `(fresh (...) ...)` macro is responsible for allocating the correct index for the variable in each specified state.

This assumption on the goals changes when we transition to μKanren with de-

layed goals. We want to give the programmer the ability to provide more information to the result of a `run` expression (provided that it is a continuation), and to manipulate the variables that have already been created. How much flexibility we wish to grant to the programmer determines the extent to which the language needs to change. Let us first consider the following simple and safe solution, which is also the least permissible and flexible. Our rule is that the programmer is only allowed to use the variables that have been declared at the very top of the `run` program, that is, the variables `q0 q1 ...` in `run n (q0 q1 ...) ...` or `run* (q0 q1 ...) ...`. Let us call them the outer-most variables. This approach is easy to implement because every state in the stream is guaranteed to have these variables. Combined with α-renaming, we can eliminate any ambiguity as to which variable a given identifier refers to.

Even though this provides some leverage to fulfil the residual delayed goals, it would not be very useful. It is possible that some or all of the delayed goals in the `run` expression involve variables that are buried deeper in the program. In that case, if a continuation is returned, i.e. if some delayed goals are not fulfilled, then there is no way one could provide more code to trigger those goals. This problem is not resolved even if the programmer writes a delayed goal that only depends on the outer-most fresh variables. The very purpose of a fresh variable is to be unified with something later. As soon as an outer-most fresh variable x enters a unification with another fresh variable y, its delayed goals are carried over to y. Hence a delayed goal of x will be treated as a delayed goal of y, the inner variable that is not allowed to be used in the subsequent `run`. We could attempt to transform the input program and pull all fresh variables on top, so an expression of the form

```
(fresh (x)
  (conde
    ((fresh (y)
       (== x y) ...)))
  ...)
```

becomes

```
(fresh (x y)
  (conde
    ((== x y) ...))
  ...)
```

However, this would certainly not yield an equivalent program, since, for example, the

scope of variables would have to change.

An alternative approach is to allow the programmer to use any fresh variable that appears in the `run` expression. This means we are further changing the assumption about what variables a given goal is allowed to involve. In the previous approach, we could only have fresh variables and the variables that are guaranteed to exist after the previous `run`. Adding more goals was equivalent to simply appending it as another clause at the end of the `run` expression. In other words, if a program

```
(run (x y z) A B C)
```

yields a continuation `k`, then we are allowed to write a goal `D` using the variables `x, y, z`. Then the result of `(k D)` is the result of `(run (x y z) A B C D)`.

Now consider a program of the form

```
(run (x y z)
   (conde A B C))
```

Suppose we allow `D` to contain a variable that was declared inside one of the goals `A, B, C`. For example, suppose `C` is the goal

```
(fresh (w) (== w '(1 2 3)))
```

and the other goals do not contain a `w`. (If they do, α-renaming will give them a unique name.)

We would like to change our specification and make it legal for `D` to involve the variable `w`. Now the question is, what is the meaning of such a goal? If `D` is `(== w '(1 2))`, do we return the empty stream, or should only the `C` branch fail? Let us try to formalize both variations.

(a) The first option is to interpret `D` as part of the goal `C`. That is, the result of `(k D)` is the result of

```
(run n (x y z)
   (conde
      A
      B
      (fresh (w)
         (== w '(1 2 3))
         (== w '(1 2)))))
```

In general, for an arbitrary goal `D` involving existing variables `w0, w1, ...`, we would place it in the first `fresh` expression of the original `run` program that makes the use of `w0, w1, ...` legal. We can immediately see that such an intersection of the `fresh`-es might not exist. But in this case the entire program would still be allowed to succeed, because the other clauses such as `A` and `B` would not involve the goal `D` at all. This seems counter-intuitive, and it would be difficult to reason about the language if we define it this way.

(b) As an alternative approach, we consider treating the goal `D` as a separate piece of code that we conjugate with the original goals `A B C` in `(run n (x y z) A B C)`. In other words, we further expand our definition of permissible goals, so that `(k D)` could yield the same stream as

```
(run n (x y z)
   (conde
      A
      B
      C)
   D)
```

We modify our interface as follows. The goal `D` is allowed to use any variable declared in the original `run` expression. However, unifying a variable in a state where it does not exist produces the empty stream. Thus, if the result of

```
(run n (x y z)
   (conde
      A
      B
      (fresh (w)
         (== w '(1 2 3)))))
```

is a continuation `k`, then `(k (== w '(1 2)))` will evaluate to the empty stream. This is so because

- Applying `(== w '(1 2))` to the states in the stream `(A empty-state)` yields the empty stream. The variable `w` does not exist in these states. The same is true for the `B` clause.
- Applying `(== w '(1 2))` to the single state in `((fresh (w) (== w '(1 2`

3))) empty-state) yields the empty stream as well. The unification algorithm will fail to unify '(1 2 3) and '(1 2).

Such an approach is relatively straightforward to implement, since we are only modifying the definition of ==.

One might wonder how we can keep the streams resulting from the clauses A and B. With this approach, would it be possible to add goals to all three clauses at the same time? It would be possible to do so by using a conde. Suppose A has a local variable u and B has a local variable v. Consider the expression

```
(k
 (conde
   ((== u 1))
   ((== v 2))
   ((== w 3))))
```

Let A', A' and C' be A, B and C with the additional goal (== u 1) respectively. Then the above expression will yield the same stream of results as

```
(run n (x y z)
   (conde A' B' C'))
```

The second model requires a lot of restrictions on the format of the code being passed to the continuation, and it is quite invasive. The scoping issue persists if we allow arbitrary variables to be used in the same expression. We therefore need a compromise between the first and the second approaches. Returning a single continuation as the output of a run program can thus be achieved if we:

- Explicitly transform the expression in the body of the run so that all variables are uniquely α-renamed. The programmer will be able to see the transformed expression and use the renamed variables in the subsequent run's.
- Implicitly hoist all inner variables and to the outermost level, making them arguments of the run command.

The explicit α-renaming part is quite straightforward. The implicit hoisting can be done by pulling the variables from inner paths. We begin by modifying the function take. This is a procedure that, given a number n and a stream $, returns the first n elements of that stream.

The baseline definition of take is as follows:

```
(define (take n $)
  (if (zero? n) '()
    (let (($ (pull $)))
      (if (null? $) '() (cons (car $) (take (- n 1) (cdr $)))))))
```

Where `pull` is a procedure for forcing an immature stream.

These *n* elements are states, i.e. pairs `(S, C)` where `S` is a substitution map and `C` is the counter. With delayed goals, the *n* elements are triples of the form `(S, D, C)` where `S` and `C` are as before, and `D` is the store with delayed goals. Now we cannot simply return these *n* states as the final result, because the stores `D` might still contain some delayed goals. If each store `D` is empty, we return the states as usual. Otherwise, we could keep pulling the stream until we find the first *n* states that do not have delayed goals. However, this would produce rather uninteresting results. It would also mean that we are treating the states with unfulfilled goals as "failed", which does not seem right. Instead, we wrap the result in a continuation that takes more goals and binds them with the stream. To this end, we define a helper procedure `take^` that accumulates the delayed goals as it pulls the stream.

```
(define (take^ n $ res vars)
  (if (zero? n) (cons res vars)
      (let (($ (pull $)))
   (cond
     ((null? $) (cons res vars))
     (else
      (let ((vars
        (pmatch (car $)
          (`(,s (,vars^ () . ,c^) . ,c) vars)
          (`(,s (,vars^ ,G . ,c^) . ,c) (append-1 vars^ vars))))
       (res
        (append res (list (car $)))))
        (take^ (- n 1) (cdr $) res vars)))))))
```

Here `append-1` accumulates the variables that are associated with delayed goals in the non-empty `D` stores. These are necessary because, if there are unfulfilled goals, we need to know which variables are involved in them.

```
(define (append-1 list1 list2)
       (if (or (null? list1) (null? (cdar list1))) list2
```

```
          (cons (caar list1) (append-1 (cdr list1) list2))))
```

The first entry in `D` is a map from variables to its delayed goals, and `append-1` uses the keys of that map to assemble the list of variables `vars`.
Next, the `take` function is modified as follows.

```
(define (take n $)
  (pmatch (take^ n $ '() '())
    (`(,res . ()) res)
    (`(,res . ,vars)
     (cons vars (lambda (goal)
        (take n (bind $ goal)))))))
```

The function `take^` returns a pair `(res, vars)` where `res` is a list of the first n elements of the stream and `vars` is a list of the variables that have delayed goals associated with them. If this list is empty, we return `res`. Otherwise we return `vars` along with the continuation

```
(lambda (goal)
   (take n (bind $ goal)))
```

One way to define the continuation is that it is a procedure that takes a list of values `vals` and unifies its elements with the variables in `vars`, one by one:

```
(define (k vars vals)
  (cond
    ((or (null? vars) (null? vals)) (lambda (s/d/c) (unit s/d/c)))
    (else
     (conj+ (== (car vars) (car vals)) (k (cdr vars) (cdr vals))))))
```

This is the most primitive way to pass values: here `k` takes a list of variable length and unifies the first l elements of `vars` with the first l elements of `vars`, where l is the minimum of of their lengths. We could also allow skipping some arguments by introducing a special symbol that represents a missing value. If `!` represents a missing value, `vars` is `x, y, z`, and the result of a `run` program is `((x, y, z), k)`, then `(k 1 ! 2)` unifies `x` with 1 and `z` with 2, and leaves `y` fresh. Alternatively, we can pass a map from variables to values to the continuation `k`.

However, there is not much one can do with such a continuation. In the above example, we are required to pass concrete values to be unified with the variables `x, y`

and z; we cannot introduce arbitrary new goals. We therefore modified the format of the continuation to achieve greater flexibility. The type of information we are allowed to feed in to k is a strict superset of that in the previous approach. Instead of passing ready values one provides information in the form of additional μKanren code that involves the symbols x, y and z. The continuation then binds these identifiers to the μKanren variables corresponding to x, y and z and evaluates the provided additional code, which yields a goal. The goal is applied to the stream, which should trigger some of the delayed goals, thus making progress in the overall computation. In fact, the predicate of every delayed goal involving x, y or z is tested whenever one of these variables is unified in the augmented goal.

As noted before, hoisting the variables to the top does not yield the same program, due to the changed scope. The last version of the interface solves the problem by replacing the one continuation holding the entire stream with many small ones.

The list of μKanren states returned by a call to take or take-all can contain fully computed states, i.e. with no unfulfilled delayed goals left. So, instead of wrapping the entire stream in a single continuation we can consider individual states and turn the ones with a non-empty D store into continuations. The run program will then always return a list where some entries are reified answers that have been fully computed, while others are continuations. The programmer can then feed additional code to specific continuations. Choosing a continuation corresponds to selecting a specific path along the program and injecting the code at the end of that path. The variables attached to a given continuation are automatically all in the same scope and visible in the part of the program where the goal is injected.

To summarize, both the choice of internal representation of delayed goals and interface design are subject to various tradeoffs. In terms of internal representation, avoiding information redundancy comes at the cost of additional levels of abstraction. On the other hand, the usability of the interface is affected when we try to achieve greater flexibility, e.g. by returning continuations associated with specific paths in the program.

Chapter 3
Future work

This project was originally aimed at exploring the relation between delayed goals and staged programming. One future direction for delayed goals is staging the miniKanren interpeter. The idea is to combine delayed goals and staging by using the former to implement the latter.

Staged programming is manual partial evaluation, where offline manual binding time analysis is used to select operations for lifting. In the case of staged miniKanren, we typically choose to lift some unifications and some conditionals.

In partial evaluation, maximizing the amount of computation depends on the correct annotation of arguments according to their availability. This is done via binding time analysis. The power of staging is manual control of binding time analysis, where one can manually decide which operations are to be lifted. The manual design stems from the observation that automatic binding-time analysis is hard, and that manual control is a powerful compromise that allows the staging user to specify what they want exactly.

In staged miniKanren, some unifications are done in the first stage, while others are quoted out and get deferred to the second stage. The second stage represents code that is "kept for later", while the first stage is for executing now. Deferring a unification is similar to deferring a command in functional programming.

Thus, lifting goals by manual annotation has a similar feel to delayed goals. It therefore might be potentially useful to integrate the two concepts in a single miniKanren extension. Specifically, an interesting application would be to use delayed goals to stage the relational interpreter. This would involve adopting the μKanren implementation and integrating the delayed goals mechanism in miniKanren. The environment could then be treated as unavailable data. Consequently, when evaluating a lambda expression, we could evaluate its body at once, preventing the need to traverse it during every application.

Chapter 4
Conclusion

This paper presents an extension of μKanren for supporting delayed goals. It sets up a framework for treating program outputs with unfulfilled delayed goals as continuations. We designed delayed goals as data accumulated for deferred computation throughout the execution of a miniKanren program. Multiple extensions of μKanren were developed and tested in Scheme. The main contribution of this project is an exploration of the tradeoffs between different models, as well as several variants of an interface for programming delayed goals in μKanren.

References

[1] J. Hemann and D. P. Friedman, "Microkanren: A minimal functional core for relational programming,"

[2] W. E. Byrd, "Relational programming in minikanren: Techniques, applications, and implementations," Ph.D. dissertation, Citeseer, 2009.

[3] D. P. Friedman, W. E. Byrd, and O. Kiselyov, *The Reasoned Schemer*. The MIT Press, 2005, ISBN: 0262562146.

[4] N. Amin and T. Rompf, "Collapsing towers of interpreters," *Proceedings of the ACM on Programming Languages*, vol. 2, no. POPL, pp. 1–33, 2017.

[5] N. H. Christensen and R. Glück, "Offline partial evaluation can be as accurate as online partial evaluation," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 26, no. 1, pp. 191–220, 2004.

[6] A. Ershov, "Mixed computation: Potential applications and problems for study," *Theoretical Computer Science*, vol. 18, no. 1, pp. 41–67, 1982, ISSN: 0304-3975. DOI: https://doi.org/10.1016/0304-3975(82)90111-6. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0304397582901116.

[7] M. Bugliesi, E. Lamma, and P. Mello, "Partial deduction for structured logic programming," *The Journal of Logic Programming*, vol. 16, no. 1-2, pp. 89–122, 1993.

[8] A. M. Pitts, "Nominal logic, a first order theory of names and binding," *Information and computation*, vol. 186, no. 2, pp. 165–193, 2003.

[9] T. Rompf, "Lightweight modular staging and embedded compilers: Abstraction without regret for high-level high-performance programming," Ph.D. dissertation, Citeseer, 2012.

[10] H. Rogers Jr, "Theory of recursive functions and effective computability," 1987.

[11] Y. Futamura, "Partial computation of programs," in *RIMS Symposia on Software Science and Engineering*, Springer, 1983, pp. 1–35.

[12] M. Kuklina and E. Verbitskaia, "Supercompilation strategies for minikanren,"

[13] V. F. Turchin, "The concept of a supercompiler," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 8, no. 3, pp. 292–325, 1986.

[14] M. A. Bulyonkov, “Extracting polyvariant binding time analysis from polyvariant specializer,” in *Proceedings of the 1993 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, 1993, pp. 59–65.

[15] D. P. Friedman and M. Wand, “Reification: Reflection without metaphysics,” in *Proceedings of the 1984 ACM Symposium on LISP and functional programming*, 1984, pp. 348–355.

[16] D. Rozplokhas, A. Vyatkin, and D. Boulytchev, “Certified semantics for minikanren,” in *Proceedings of the 2019 miniKanren and Relational Programming Workshop*, 2019, pp. 80–98.

[17] C. E. Alvis, J. J. Willcock, K. M. Carter, W. E. Byrd, and D. P. Friedman, “Ckanren minikanren with constraints,” 2011.

[18] W. E. Byrd and D. P. Friedman, “Alphakanren a fresh name in nominal logic programming,” 2007.

Appendix

An overview of μKanren

The basic constructs of μKanren are variables and goals. A μKanren variable is an object that can be in a relation with other variables and values. In [1], variables are represented by Scheme vectors, where the first element is the unique index of that variable. A goal is a function that takes a state and returns a stream of states. In μKanren, goals are used to unify variables with terms such as lists, symbols and numbers. Different versions of miniKanren have additional types of constraints. For example, the goal `(absento x y)` is the relation that `x` is not part of `y`. If `x` is a fresh variable satisfying the goal `(numbero x)`, then unifying `x` with any term that is not a number will fail.

The == operator and the macros `fresh` and `conde` are used for creating and combining goals. Some of the simplest goals are of the form `(== x y)`. We write `(fresh (x ...) g ...)` to create fresh variables `x ...` and run the goals `g ...`, which can contain these variables. The procedures `run` and `run*` serve as the main interface for computing the variables satisfying a given set of relations. An expression of the form `(run n (q) g0 ...)` means "compute the first `n` values for `q` that satisfy the goals `g0 ...`". The `run` command starts from an empty state and applies the goals `g0 ...` one by one. The program returns the first `n` elements accessed from the resulting stream of states.

Scheme implementation of μKanren

We used a Scheme version of μKanren by Hemman and Friedman [1] as a baseline implementation.

Streams and states

A μKanren *state* holds information about the progress of a goal along a certain path. It accumulates relations between variables, as well as their properties and constraints. In the canonical miniKanren, there are constraint stores that associate variables with types, such as "x is a symbol", disequality constraints, such as "x is not y", and `absento` constraints, such as "x is not in y". In μKanren, a state `s/c` consists of a substitution map `s` and a counter `c`. The substitution map is a list of pairs, where the `car` of each pair is a variable (i.e. a vector), and the `cdr` is an arbitrary term such as a number or

another variable.

A stream is a sequence of elements that has been only partially computed. The empty stream is denoted by `mzero`, while a stream of consisting of one element `s/c` is a `unit`:

```
(define (unit s/c) (cons s/c mzero))
(define mzero '())
```

One can readily access the first element of the stream, and the rest of it is wrapped in a thunk. Such a stream is called a *mature* stream, while the thunk is called an *immature* stream. Forcing the thunk yields a mature stream, i.e. the next computed element of the stream along with a thunk representing the rest of the elements.

In addition to the usual lexical variables in Scheme which can be created using `let`, λ, etc, μKanren has its own logic variables, which can be associated with values via a substitution mapping. These can be *ground*, i.e. associated with a value, or *fresh*. If we unify two fresh variables `x` and `y`, they both remain fresh. Logic variables can be created using the `fresh` operator, which creates a fresh logic variable and locally binds it to a lexical variable. Initially, the logic variable does not have a value associated with it, but it can become ground through unification. In the full-featured miniKanren, in addition to associating logic variables with terms, there can be constraint stores mapping variables to type, disequality, and other kinds of constraints.

The variables are represented by vectors holding a single non-negative integer.

```
(define (var c) (vector c))
(define (var? x) (vector? x))
```

When a variable is created, the integer `c` depends on a counter carried in the state along with the substitution map. Variable equality is defined as equality of the indices.

```
(define (var=? x1 x2) (= (vector-ref x1 0) (vector-ref x2 0)))
```

This definition is preferrable to pointer equality because it is more pure.

The == operator

The main relation in μKanren is ==, which is used for creating unification goals. These have the form `(== x y)`. If `x` and `y` are variables, then they are said to be *fused* through this relation [3]. This means that whatever happens to `x` also happens to `y`. If only `x` (or only `y`) is a variable, then a unification procedure binds `x` to `y`.

```
(define (== u v)
  (lambda (s/c)
    (let ((s (unify u v (car s/c))))
      (if s (unit `(,s . ,(cdr s/c))) mzero))))
```

The unification algorithm `(unify u v s)` is implemented as follows. First, both terms `u` and `v` are `walk`-ed in the substitution map `s`. The recursive function `walk` looks up the binding of a given variable in `s`, and applies itself to the result, if it exists. If `u` (or `v`) is still a variable after we `walk` them in `s`, it is said to be fresh. Suppose `u` is fresh. Then `(unify u v s)` extends the map `s` with a binding associating `u` with `v`:

```
(define (unify u v s)
  (let ((u (walk u s)) (v (walk v s)))
    (cond
      ((and (var? u) (var? v) (var=? u v)) s)
      ((var? u) (ext-s u v s))
      ((var? v) (ext-s v u s))
      ...
```

If `u` is not fresh but `v` is, the subsitution map is extended with a pairing `(v, y)`. If neither of `u` or `v` is a variable after they are `walk`-ed in the substitution map, then we check whether they are both lists. If so, every element of `u` must be unified with every element of `v`. Finally, if `u` and `v` fail to be variables or lists, they must be equal for the unification to succeed:

```
      ...
      ((and (pair? u) (pair? v))
       (let ((s (unify (car u) (car v) s)))
         (and s (unify (cdr u) (cdr v) s))))
      (else (and (eqv? u v) s)))))
```

In this most basic version of miniKanren, the counter and the substitution map are the only characteristics of a state, but the language retains the essential features such as `fresh` and `conde`. Fresh variables are implemented using `call/fresh`. Calling `(call/fresh f)` yields a procedure that takes a state `s/c` and creates a fresh variable `(var c)`, where `c` is the current counter value of the state. Note that `f` must be a function taking a variable and returning a goal. It is applied to the new variable and the resulting goal is applied to the state `` `(,s . ,(+ c 1)) ``. The substitution map remains the same,

while the counter is incremented by one, so that the next fresh variable will be a vector holding a higher index. This ensures that all fresh variables are unique.

```
(define (call/fresh f)
  (lambda (s/c)
    (let ((c (cdr s/c)))
      ((f (var c)) `(,(car s/c) . ,(+ c 1))))))
```

The result of applying a goal `g` to a state `s` is a stream of states all of which satisfy this goal. Given two such streams, we can combine them into one using the `mplus` operator. It is useful for splitting paths and creating disjunctions of goals.

```
(define (mplus $1 $2)
  (cond
    ((null? $1) $2)
    ((procedure? $1) (lambda () (mplus $2 ($1))))
    (else (cons (car $1) (mplus (cdr $1) $2)))))
```

This ensures interleaving of the two streams, as we are pulling the first element of the first stream, and the next element will be the first element of the second stream, etc.

If the stream `$1` is mature, then `$1` is a pair, and its `car` becomes the `car` of the resulting stream. On the other hand, the `cdr` of `$1` is an immature stream. So, calling `(mplus (cdr $1) $2)` will create another immature stream that swaps the streams (`$1`) and `$2`, where (`$1`) is the result of forcing the thunk `$1`. This means the next element of the resulting stream will be either

(a) The `car` of `$2`, if `$2` is mature, or

(b) the `car` of `$1`, if `$2` is immature. However, in this case the third element of the resulting stream is guaranteed to be the first element of `$2`.

In either case, we eventually achieve a stream of alternating elements.

Another important mechanism is `bind`, which is used for applying a goal to a stream of states.

```
(define (bind $ g)
  (cond
    ((null? $) mzero)
    ((procedure? $) (lambda () (bind ($) g)))
    (else (mplus (g (car $)) (bind (cdr $) g)))))
```

The `bind` operator is a dual of `mplus`. It takes a stream `$` and a goal `g` and returns the the resulting from applying `g` to each state in `$`.

Now that we have operators for merging streams and binding goals with them, the conjunction and disjunction of goals can be defined.

```
(define (disj g1 g2) (lambda (s/c) (mplus (g1 s/c) (g2 s/c))))
(define (conj g1 g2) (lambda (s/c) (bind (g1 s/c) g2)))
```

The disjunction of the goals `g1` and `g2` is a new goal `g` such that, when applied to the state `s/c`, it returns the merged streams `g1 s/c` and `g2 s/c`. This is used when `g1` and `g2` are goals independent of each other and represents the OR of the two goals.

On the other hand, `(conj g1 g2)` is the AND of the goals `g1` and `g2`, so they are applied sequentially. The conjunction of `g1` and `g2` is a goal `g` such that, given a state `s/c`, it applies `g1` to it, and then applies `g2` to the resulting stream.

The `fresh` and `conde` macros

The `fresh` macro creates a fresh variable in a given state. It is defined as

```
(define-syntax fresh
  (syntax-rules ()
    ((_ () g0 g ...) (conj+ g0 g ...))
    ((_ (x0 x ...) g0 g ...)
     (call/fresh
      (lambda (x0)
        (fresh (x ...) g0 g ...))))))
```

The goals `g0, g ...` are expressions that can contain the variables `x0, x ...`. Given a state `s/c`, the `call/fresh` mechanism allocates a new index for the variable `x0`, then for `x`, and so on. As a result, we have a new state with an increased counter, and the Scheme variables `x0, x ...` are bound to μKanren variables, i.e. vectors holding indices. Next, the goals `g0 g ...` are all conjugated and applied to this new state, in an environment that now knows about `x0, x ...`.

The `conde` macro creates the disjunction of a set of goals.

```
(define-syntax conde
  (syntax-rules ()
    ((_ (g0 g ...) ...) (disj+ (conj+ g0 g ...) ...))))
```

A principal difference between `conde` and Scheme's `cond` is that the latter checks the

condition of every clause and only executes the first clause whose test passed. On the other hand, `conde` runs every clause, yielding multiple streams, all of which are then merged using `mplus`.

To run a μKanren program, the `run` and `run*` macros are used. These constitute an interface allowing to compute a stream of values for a variable satisfying a set of μKanren goals. Given a set of fresh variables `x ...` and goals `g0 g ...`, the `run` and `run*` procedures apply the conjunction of these goals to the empty state `'(() . 0)`, i.e. the empty substitution map and the counter `0`.

```
(define-syntax run
  (syntax-rules ()
    ((_ n (x ...) g0 g ...)
     (map reify-1st (take n (call/goal (fresh (x ...) g0 g ...)))))))
```

Here `(call/goal g)` simply returns `(g empty-state)`. The difference between `run n` and `run*` is that `run n` computes and returns the first n elements of the stream, while `run*` computes the entire stream. The definition only differs in that `take n` is replaced with `take-all`. The two procedures comprise the main interface for programming in the miniKanren family of languages. It is a wrapper for the lower level procedures that manipulate streams, states, binding and interleaving.

An arbitrary stream is made mature via the `pull` procedure that forces immature streams:

```
(define (pull $)
  (cond
   ((null? $) mzero)
   ((procedure? $) (pull ($)))
   ((pair? $) $)))
```

The result of `pull` is thus either the empty stream `mzero` or a pair (a mature stream with a computed first element). The `take` and `take-all` procedures used by the `run` interface invoke `pull` to compute some or all elements of the stream:

```
(define (take-all $)
  (let (($ (pull $)))
    (if (null? $) '() (cons (car $) (take-all (cdr $))))))
```

The `take-all` procedure continues to build a list of the elements of `$` until the stream is exhausted. The function `take` is defined similarly, except it only computes the first `n`

elements of `$`. The `run` and `run*` procedures are reified versions of `take` and `take-all`, where we are only interested in the first variable, `(var 0)`.

Reification is the process of giving an abstract state in a computation an explicit representation [15]. In our case, the state of a computation is a stream, and we use reification to be able to inspect the contents of that stream. Simply returning the substitution map does not convey much information to the programmer. The Scheme vectors used to represent μKanren variables are internal constructs that the μKanren programmer is not concerned with. The function `reify-st` converts μKanren terms to a human-readable form and is defined as

```
(define (reify-1st s/c)
  (let ((v (walk* (var 0) (car s/c))))
    (walk* v (reify-s v '()))))
```

Thus, a term is walked in the substitution map, i.e. all variables are recursively replaced with their `walk*` values. The variables that still appear in the term by the end of this process are guaranteed to be fresh. We then have a term containing concrete values such as numbers and symbols, as well as vectors representing the fresh variables. Lastly, each of these vectors acquires a unique special symbol, e.g. one starting with the prefix "`_.`". It might appear that seeing only the first variable is not very informative. However, an arbitrary set of variables can be viewed this way. One just needs to unify the first variable with a list of variables we are interested in. Suppose we wish to find all `x` and `y` such that appending `y` to `x` yields the list `'(1 2 3 4 5)`. Then we can query the values for `q` in the following goal:

```
(run* (q)
  (fresh (x y)
    (== `(,x ,y) q)
    (appendo x y '(1 2 3 4 5))))
```

This program returns the list of all possible combinations of `x` and `y` satisfying the relation `(appendo x y '(1 2 3 4 5))`, which holds if and only if `(append x y)` returns `'(1 2 3 4 5)`:

```
'((() (1 2 3 4 5))
  ((1) (2 3 4 5))
  ((1 2) (3 4 5))
  ((1 2 3) (4 5))
  ((1 2 3 4) (5))
```

```
   ((1 2 3 4 5) ()))
```

Each value in the list is the result of reifying `q` in an element of the computed stream, i.e. calling `(reify-1st s/c)` where `s/c` runs through the states in the list

```
(take-all (call/goal (fresh (q) (fresh (x y)
    (== `(,x ,y) q)
    (appendo x y '(1 2 3 4 5))))))
```

However, we are actually reifying the `walk*` values of `` `(,x ,y) ``, because that is what the variable `q` maps to in the substitution map. The goal `(== `(,x ,y) q)` invokes the unification algorithm, which `walk*`-s both `` `(,x ,y) `` and `q` in the empty substitution map. The `walk*` procedure is defined as follows:

```
(define (walk* v s)
  (let ((v (walk v s)))
    (cond
      ((var? v) v)
      ((pair? v) (cons (walk* (car v) s)
                       (walk* (cdr v) s)))
      (else v))))
```

This procedure, when applied to `` `(,x ,y) `` and `q` in the empty substitution map, returns `` `(,x ,y) `` and `q` back. Then, the unification algorithm checks whether one of these is a variable. Since `q` is a variable and `` `(,x ,y) `` is not, we conclude that `q` must map to `` `(,x ,y) `` as a result of this unification.

Reification is performed by incrementally building a map from fresh variables to symbols. First, a new symbol is created for every fresh variable in a given list:

```
(define (reify-s v s)
  (let ((v (walk v s)))
    (cond
      ((var? v)
       (let ((n (reify-name (length s))))
         (cons `(,v . ,n) s)))
      ((pair? v) (reify-s (cdr v) (reify-s (car v) s)))
      (else s))))
```

A variable is given a symbol based on the length of the map `s` constructed so far. If the length of `s` is 4, then the index in the reified name for the next fresh variable `x` is 4, and

the map is extended with a new entry mapping `x` to its reified name. The final map `s` is then used to reify the first variable. In the earlier example, we walk the vector `#(0)`, i.e. the first variable `q`, in a list of states. In each of the states, the substitution map associates `q` with the pair of vectors `#(1)` and `#(2)` corresponding to the expression `` `(,x ,y) ``. The result of `walk*`-ing this pair is different in each answer, since `appendo` contains a `conde`, which splits the program into multiple paths.

These are the essential features of miniKanren upon which implementing other canonical constraints such as `absento`, `=/=`, `symbolo`, `numbero`, etc is straightforward. For a more detailed exposition of the language and its implementation, see [1]. The techniques and theory behind the design of miniKanren are covered in [2]. *The Reasoned Schemer* is a thorough guide to programming in miniKanren and implementing it [3]. This project used μKanren as a core implementation of miniKanren to build the extension for supporting delayed goals. Other variations of miniKanren include cKanren for constraint logic programming, αKanren for nominal logic, Probabilistic Kanren, Staged miniKanren, and more [17, 18].