Implementing MicroKanren in Haskell

CS2104 Term Paper

Mayank Keoliya
School of Computing
National University of Singapore
Singapore, Singapore
mayank@u.nus.edu

E-Liang Tan
School of Computing
National University of Singapore
Singapore, Singapore
tan.eliang@u.nus.edu

Noel Kwan
School of Computing
National University of Singapore
Singapore, Singapore
noelkwan@u.nus.edu

***Abstract*—This paper discusses the miniKanren family of languages, and in particular, the implementation of a desugared subset, microKanren, in the Haskell programming language. A comprehensive explanation of the implementation is undertaken, and possible explorations for quines, SAT solvers and procedural-logic programming are further elaborated.**

***Index Terms*—MiniKanren, logic programming, declarative, Haskell, delayed interleaving.**

I. Introduction

Conceptualized in 2010 by William E. Byrd, and published in *The Reasoned Schemer* miniKanren is a family of programming languages with a common interface - translating relational idioms in algebra purely to working code. miniKanren supports a variety of relational idioms and techniques, making it feasible and useful to write interesting programs as relations. This interface is designed to be flexible, and unlike patrons of conventional languages such as Java or Haskell, users of miniKanren are encouraged to adapt the miniKanren implementation according to their own needs.

miniKanren's core, called microKanren or μKanren, is a desugared subset of miniKanren, with Hemann et al [4] implementing it in 54 LOC. Concrete implementations of miniKanren in Scheme, Haskell, Racket and Clojure extend this microKanren core to include constraint-logic programming, probabilistic-logic programming, nominal programming and tabling. This paper will:

- cover salient differences between miniKanren and conventional logic programming languages such as Prolog, Mercury and Curry
- present a full implementation of the microKanren core in Haskell and discuss its assumptions and limitations
- explore quines, procedural-logic constructs, and a SAT solver in microKanren

II. Differences with Conventional Logic Languages

miniKanren uses different design choices compared to conventional logic programming languages, and it is instructive to consider these differences before implementing the language. The novel differences in miniKanren arise because of the motivation behind it - it was specifically designed to be used for relational declarative programming. miniKanren's emphasis on both pure relations and finite failure leads to complete (interleaving) search strategies (unlike Prolog), and full unification (unlike Mercury), and an absence of extra-logical constructs. [1].

A. Interleaving Search

miniKanren uses an interleaving DFS strategy (DFS_i) to unify disjunctions, which is an enhancement from Prolog's naive DFS, and has key implications for program termination [5]. This leads to interesting divergences in expected behaviour, when the same query is run in miniKanren versus Prolog, where *conde* in miniKanren is analogous to the disjunction *;* operator in Prolog:

```
(run 1 (q)
  (let non_terminating ()
    (conde
      ((non_terminating)))
      ((== 3 q))))
```

```
non_terminating_rule(Q) :-
        (rule(Y), non_terminating_rule(Y)) ;
        Q = 3.
```

miniKanren's use of a BFS-infused DFS or DFS_i results in a goal searching on both branches of the *conde* interleaved with each other, and terminates the unification when the latter clause (==3 q) matches. However, a naive implementation of DFS in Prolog would not terminate on account of left-recursion. This is a pitfall of Prolog since it **prevents commutativity of disjunctions**, which miniKanren fully supports.

However, note that miniKanren's interleaving isn't entirely **fair**. It assigns higher "weightage," or frequency in goal matching, to the branches that appear earlier in the disjunction. To understand the fuller implications of this, and to learn about the other strategies that miniKanren can potentially adopt, let us consider a Prolog-miniKanren analogue of the repeato rule:

```
% in Prolog
repeato(X, [X]).
repeato(X, [X|H]):- repeato(X, H).

% in miniKanren
(defrel (repeato x out)
    (conde
        ((== `(, x) out))
        ((fresh (res)
            (== `(,x ∘ , res) out)
            (repeato x res)))))
```

Next, consider the following query in miniKanren:

```
(run 12 q
    (conde
        ((repeato 'a q))
        ((repeato 'b q))
        ((repeato 'c q))
        ((repeato 'd q))))
```

A truly fair interleaving (called fair-DFS or DFS_f) would accord equal weightage in depth to all branches of a disjunction. Note that DFS_f will be implemented in Section III by manipulating lazy streams.

```
'((a) (b) (c) (d)
(a a) (b b) (c c) (d d)
(a a a) (b b b) (c c c) (d d d))
```

An almost-fair balanced interleaving, termed DFS_{bi} would be fair only when the number of goals is a power of 2, otherwise, it allocates some branch goals with twice the weightage as the others.

```
'((a) (c) (b)
(a a) (c c) (b b) (d)
(a a a) (c c c) (b b b) (e)
(a a a a) (c c c c) (b b b b) (d d)
(a a a a a) )
```

Finally, the purely interleaved vanilla version of DFS_i, which assigns priority weightage to branches that appear earlier would result in:

```
'((a ) (a a) (b) (a a a)
(a a a a) (b b)
(a a a a a) (c)
(a a a a a a) (b b b)
(a a a a a a a) (d))
```

TABLE I
FAIRNESS STRATEGIES W.R.T DISJUNCTION

Search Strategies	Typical disjunction	Disjunction with 2^n goals
DFS_f	fair	fair
DFS_{bi}	unfair	fair
DFS_i	unfair	unfair

The fairness of the search strategies are summarized above. This illustrates that miniKanren provides for not only a single strategy for terminating disjunctions, but multiple such search strategies that the programmer can use to weight clauses. This approach is particularly useful in building fuzzers for testing, where the programmer can place a more complicated use-case at the top to ensure that it has a heuristically higher coverage while testing [5].

B. Absence of Extra-Logical Constructs

In practice, more complex Prolog programs tend to use at least a few extra-logical features, such as cut, which inhibit the ability to treat the resulting program as a relation, and interfere with the process of unification. Consider the cut operator in Prolog:

```
max1(X, Y, X) :- X > Y, !.
max1(X, Y, Y).
```

In this case, the cut (!) operator prevents backtracking to the first rule to ensure the correctness of max1 in finding the maximum of two numbers. Without, the second rule would be matched as well, resulting in an incorrect answer. This is a **side-effect** in Prolog, because instead operating as a logical unification, the cut operator interferes with the solver itself by changing the way results are unified. The use of this procedural side-effect detracts from the aim of declarative programming.

Prolog also exposes a retract keyword to enable the user to explicitly remove rules during the execution of the program. For example, consider the following session in Prolog:

```
?- likes(mary, pizza).
true.
?- retract(likes(mary, pizza)).
true.
?- likes(mary, pizza).
false.
```

In contrast, miniKanren is explicitly designed to support the purely declarative (and thus non-procedural, non-side-effect-prone) style of relational programming. There are no procedural side-effects such as cut, and no global logic database (and thus the problem of using the retract construct does not arise). More recent versions of miniKanren have support for symbolic constraint solving to make it easier to write non-trivial programs such as quines (as shown in Section 5) as relations.

III. Basic Syntax of microKanren

The microKanren API is rather minimal, designed to be pure and simple, to highlight the essentials of relational programming [2].

```
module MicroKanren (
                    -- | Types
                      Stream (..)
                    , Goal
                    , State
                    , Term (..)
                    , Var
                    , VariableCounter
                    -- | Relational operators
                    , (===)
                    , unify
                    , fresh
                    , disj
                    , conj
                    -- | Utilities
                    , delay)
```

IV. Implementation

A microKanren program has several core parts. In our microKanren implementation [7], we have a goal, which is constructed with relational operators and provides relational constraints for our program. We have State which contains variable bindings and a means to construct new variable bindings. Lastly we have the results of program, which is modelled as a stream of states.

The goal is applied to some *state*, returning all states satisfying the declared constraints. The possible states are represented as a *stream* of *states*.

In the following sections, we provide a detailed explanation on how these are implemented. Before we proceed, we demonstrate a simple program to complement our explanation above.

A. Sample program - generating natural numbers

In the following program, we implement a goal which gives us all natural numbers.

```
import          MicroKanren
import          MiniKanren  (runEval)

nil :: Term
nil = Atom mempty

suc :: Term → Term
suc = Pair (Atom mempty)

-- | Constraint a variable to be a natural number
nat :: Term → Goal
nat x = disj
    (x === nil)
    (fresh (λd →
        conj (x === suc d)
             (nat d)))

main :: IO ()
main = putStrLn
     $ prettyPrintResults
     $ takeS 4
     $ runEval nat initialState
```

We do so in a declarative way, by binding a variable to constraints. We assert that the variable is either:

- *nil* (Zero)
- the successor of some natural number.

The declaration extracted and shown below.

```
-- | Constrain a variable to be a natural number
nat :: Term → Goal
nat x = disj
    (x === nil)
    (fresh (λd →
        conj (x === suc d)
             (nat d)))
```

This results in the following output:

```
-- Zero
Var 0 := Atom ""

-- One
Var 0 := Pair (Atom "") (Atom "")

-- Two
Var 0 := Pair (Atom "") (Pair (Atom "") (Atom ""))

-- Three
Var 0 := Pair (Atom "")
              (Pair (Atom "")
                    (Pair (Atom "")
                          (Atom "")))
```

In the above output, *Var 0* is the *reified* form of variable *x* in the declaration. As shown, x is natural number and can be 0, 1, 2, 3 and so on. Since we selected the first four elements of the result stream, we only obtained the first four elements in the output.

B. Initial State

```
type State = (Subst, VariableCounter)
```

State is represented as the product of an association list and a variable counter as shown above. The association list maintains all variable bindings. We then perform unification to ensure constraints as well as bindings are satisfied.

The variable counter provides us the means to extend our state with unique variables. Further explanation is given in the section on the *fresh* operator.

Hence, we usually start from an empty state, and add new variables and their constraints through the *fresh* operator.

We shall declare empty state as follows, to be used through out the rest of this paper:

```
emptyState :: State
emptyState = ([], 0)
```

C. Results

We represent our results as a Stream of states. This is because a set of constraints can have multiple satisfactory states.

We shall illustrate this with an example. We declare a variable x, with the constraint that x is either 1 or 2. In which case, we have two possible states which satisfy the constraint, one where **variable x is bound to 1** and another where **variable x is bound to 2**.

D. goals

```
type Goal = State → Stream State
```

As you may infer from the type signature, goals allow us to extend states and apply constraints the variable bindings.

The following example extends a state by creating a new variable, and binding it to the literal "1".

```
newState :: State → Stream State
newState (s, c) = return (s', c + 1)
    where s' = (c, Atom "1") : s
```

In the above example, we use the variable counter to instantiate our variable and increment it after, ensuring the next variable is uniquely tagged.

A goal either *succeeds*, when it returns a non-empty stream of states, or *fails*, when it returns an empty stream.

E. operators

MicroKanren has 4 operators which we use as building blocks: *fresh*, *disj*, *conj* and === . These are used to construct *relational constraints* in our program.

F. fresh

The *fresh* operator takes in a function which binds a single variable to a *goal*, and returns a *goal*, as you can see below.

```
fresh :: (Term → Goal) → Goal
fresh f = λ(s, c) → f (Var c) (s, c + 1)
```

This is used to create new variable bindings with constraints.

```
varIsOne :: Goal
varIsOne = fresh $
    λx → x ≡≡ Atom "1"
```

In the above example, we extend the state with a new variable, x, with the constraint that it has to be one.

Applying it to an empty *state* gives us the following results:

```
varIsOne emptyState

—— Results in

Var 0 := Atom "1"
```

G. disj

The *disj* operator takes in two *goals* and performs a logical disjunction / union on them.

```
disj :: Goal → Goal → Goal
disj g1 g2 = λs → g1 s `mplus` g2 s
```

Sample goal with disjunction

```
oneOrTwo :: Goal
oneOrTwo = λfresh $ x →
    disj (x ≡≡ 1)
         (x ≡≡ 2)

——— Results

Var 0 := 1

Var 0 := 2
```

As earlier mentioned in the *introduction*, we interleave our results to provide a *complete search*. We declare a *MonadPlus* instance which implements the *mplus* method above. *mplus* in turn performs the interleaving and handles *switching* via *delays* as well.

MonadPlus and Monad instances

```
instance Monad Stream where
    return = pure
    Nil >>= _           = Nil
    x `Cons` xs >>= f = f x `mplus` (xs >>= f)
    Delayed s >>= f   = Delayed (s >>= f)

— | We can reuse Alternative instances,
— due to it being a class constraint for MonadPlus
instance MonadPlus Stream where
    mzero = empty
    mplus = (<|>)

— | Interleaving rather than appending 2 streams
instance Alternative Stream where
    empty = Nil
    Nil <|> xs            = xs
    (x `Cons` xs) <|> ys = x `Cons` (ys <|> xs)
    Delayed xs <|> ys     = Delayed (ys <|> xs)
```

The first question usually asked is why we need a *Stream* type to represent all possibilities of results, when we could use the list datatype in Haskell (*[]*).

In strict languages, a new datatype has to be constructed as both arguments of the cons operator would be fully evaluated.

In Haskell however, our cons operator, *(:)* is lazy in both the head and tail as well. This means that even if a goal returns an infinite list of possible states, we can just force evaluation on what we need and ignore the rest.

Hence it seems strange to create a new *Stream* type.

Laziness of cons

```
*MicroKanren> results = ["a", "b"]
*MicroKanren> :sprint results
```

```
results = _
*MicroKanren> head x
"a"
*MicroKanren> :sprint x
x = "a" : _ -- Tail is ignored
```

```
*MicroKanren> results = ["a", "b"]
*MicroKanren> :sprint results
results = _
*MicroKanren> tail x
["b"]
*MicroKanren> :sprint x
x = _ : ["b"] -- head is ignored
```

To understand why, we look at the following goal.

```
goalR :: Goal
goalR = goalR `disj` goalT

goalT :: Goal
goalT = return
```

Inutitively, this should work fine for us, as `disj` should behave as follows:

- If one of its goals never terminates, and the other one does, it should return the results from the terminal goal.
- `disj` should not care about the order of its arguments, relations are commutative.

When we evaluate the above goal however,

```
goalR s = (goalR `disj` goalT) s
        = goalR s `mplus` goalT s
        -- mplus behavior is the same as interleave
        -- We evaluate the first argument,
        -- in order to deconstruct it
        = (goalR `disj` goalT) s `mplus` goalT s
        = (goalR s `mplus` goalT s) `mplus` goalT s
        = ((goalR s `mplus` goalT s)
            `mplus` goalT s)
                `mplus` goalT s
        =◦ ..
```

We realize that we are never able to extract the head of `goalR s` due to its recursive definition.

However, if we swapped the arguments around:

```
goalR s = (goalT `disj` goalR) s
        = goalT s `mplus` goalR s
```

We would be able to ignore the recursive part, as it is pushed to the back, and *mplus* deconstructs its first argument.

Hence we need a way to encode this into the results. We do so with Streams, by extending the definition of the list.

H. streams

```
data Stream a = Nil
              | Cons a (Stream a)
              | Delayed (Stream a) deriving (Eq, Show)
```

In the definition of our Stream, apart from normal list primitives like *Nil* and *Cons*, we also have a *Delayed* constructor.

If we go back to our declaration of *MonadPlus*, where we defined *mplus*:

```
-- | We can reuse Alternative instances,
-- due to it being a class constraint for MonadPlus
instance MonadPlus Stream where
    mzero = empty
    mplus = (<|>)

-- | Interleaving rather than appending 2 streams
instance Alternative Stream where
    empty = Nil
    Nil <|> xs           = xs
    (x `Cons` xs) <|> ys = x `Cons` (ys <|> xs)
    Delayed xs <|> ys    = Delayed (ys <|> xs)
```

We can observe that we pattern match on the *Delayed* constructor, swapping around streams and prioritizing the other whenever a *Stream* is *Delayed*.

This manual annotation is still rather error prone however, as we can easily forget to annotate, or incorrectly annotate our goals.

To solve this, we need to encode the *Delay* constructor inside recursive definitions.

We can wrap the resulting Stream of all our operators with *Delay* to do so.

This results in the following expansion

```
goalR s = Delayed (goalR s `mplus` goalT s)
        = Delayed (
            (Delayed (goalR s
                `mplus` goalT s))
                    `mplus` goalT s)
        = Delayed
            (Delayed (goalT s
                `mplus` goalR s
                    `mplus` goalT s))
```

Since goalT s is not recursively defined, so we will always be able to deconstruct it.

More generally, as long as one of the goals are not recursively defined, we will always be able to evaluate `disj` to give us results.

I. conj

The *conj* operator takes in 2 *goals* and performs a logical conjunction on them.

```
conj :: Goal → Goal → Goal
conj g1 g2 = λs → g1 s `bind` g2
```

Sample goal with conjunction

```
x_And_y_eq_one :: Goal
x_And_y_eq_one = fresh $ λx →
    fresh $ λy →
        conj (x === Atom "1")
             (y === Atom "1")

——— Results

Var 0 := Atom "1"
Var 1 := Atom "1"
```

J. equality (===)

The === operator takes in 2 *terms* and performs a equality check on them.

```
(===) :: Term → Term → Goal
(===) t1 t2 = λ(s, vc) → case unify t1 t2 s of
    Nothing → Nil
    Just s' → return (s', vc)
```

V. Usage

Begin by defining a goal that we want to satisfy. For example, this is how we can define the following boolean expression:

$$(X = \text{“1”} \vee X = \text{“2”}) \wedge (Y = \text{“}a\text{”} \vee Y = \text{“}b\text{”})$$

```
goal :: Goal
goal = fresh $
    λx → fresh $
        λy → ((x === Atom "1") `disj` (x === Atom "2"))
            `conj` ((y === Atom "a") `disj` (y ===
Atom "b"))
```

Goals are executed against state. We can use the *initialGoal* for this, defined as a state with an empty set of bindings and a variable counter set to 0. *initialGoal* is already defined within our microKanren implementation.

```
initialState :: State
initialState = ([], 0)

results :: Stream State
results = goal initialState
```

prettyPrintResults can be used to reduce the stream of goals into a string for display. All associations with satisfy the state constraints in the *goal* will be output.

```
displayResults :: IO ()
displayResults = putStrLn $ prettyPrintResults results

— Output
Var 1 := Atom "a"
Var 0 := Atom "1"

Var 1 := Atom "a"
Var 0 := Atom "2"

Var 1 := Atom "b"
Var 0 := Atom "1"

Var 1 := Atom "b"
Var 0 := Atom "2"
```

VI. Extensions

A. Quines

A quine is a program that evaluates to itself [3], more formally, “A quine is an expression q in L such that uqq.” A trivial such example would be a literal in Haskell, or Scheme - numbers, booleans, or characters. Typically, quines in procedural languages have tended to be complicated, such as those submitted by Broukhis et al for the International Obfuscated C Code Contest [6]. However, miniKanren, due to its declarative nature, offers a way to translate the computational meta notation (CSM) directly to code via the following:

```
(run 1 (e) (evalo e e))
```

Here, evalo binds the value of the first argument to the second, i.e the value of e to itself [2]. Infinity such quines can be generated using miniKanren's run* command. Interestingly, we can also generate the more notationally-complicated twines (twin quines), which are distinct programs p and q that evaluate to each other, and thrines, which are distinct programs p, q, and r such that p evaluates to q, q evaluates to r , and r evaluates to p. For example, twin quines can be generated with the following:

```
(run 1 (p q)
    (fresh (p q)
        (=/= p q)
        (evalo q p)
        (evalo p q)))
```

B. Procedure in Logic Programming

Another benefit of using miniKanren is the ability to write forward and backward queries for procedural functions, without having to define the logical rules for the function. For example, in when written in Scheme, miniKanren allows the user to query the append function backwards (i.e with its ”output”) as well:

```
> (run 1 (q)
    (evalo `(letrec ((append
                (λ (l s)
                  (if (null? l) s
                  (cons (car l) (append (cdr l) s))))))
            (append ,q `(d e))
            `(a b c d e))

% results in `(a b c)
```

Here, we could define a procedural function using the comfortable idiom of the host language (Scheme), pass in the ”output” of the function, and proceed in relationally backward direction to bind the free variable q to the desired input. This is paradigmatically different from the Prolog approach, where to *use* the append function relationally, it would have to be *defined* relationally as well:

```
append([], L, L).
append([H|T], L, [H|R]) :-
  append(T, L, R).
```

C. SAT Solver

SAT solvers check if a boolean expression is solvable. Although the boolean satisfiability problem is NP-hard in the most general case, it can be solved efficiently if we only consider boolean expressions in the conjunctive normal form, (colloqiually, ”ands of ors”) or the disjunctive normal form

("ors of ands"). We will demonstrate how a 2-SAT solver can be implemented in microKanren, which solves the 2-SAT problem.

First, we begin by defining utility constants and functions, so that the final code will be more readable.

```
true :: Term
true = Atom "True"

false :: Term
false = Atom "False"

and :: Term → Term → Term
and (Atom "True") (Atom "True") = true
and _ _                         = false

or :: Term → Term → Term
or (Atom "False") (Atom "False") = false
or _ _                           = true

not :: Term → Term
not (Atom "true") = Atom "False"
not _             = Atom "true"
```

Next, we define our SAT solver as a goal. In the main body, we define *twoSat* to some arbitrary 2-SAT boolean expression and *bindings* to all possible pairs of x and y.

```
satSolver :: Goal
satSolver =
  fresh $ λq →
    fresh $ λx →
      fresh $ λy →
          let twoSat = (x `or` y) `and` (x `or` y)
              bindings = conj
                  ((x == true) `disj` (x == false))
                  ((y == true) `disj` (y == false))
              -- | Our expression we want to
              -- solve has to evaluate to true
          in (twoSat == q) `conj` (q == true)
               `conj`
                 bindings
               -- | Bind True, False to x and y
```

Finally, we can evaluate the goal against the initial state and print the results.

```
printRes :: Stream State → String
printRes Nil = "False"
printRes _   = "True"

main :: IO ()
main = putStrLn $ printRes $ satSolver initialState

-- Output
True
```

VII. Conclusion

In sum, this paper has shown how miniKanren's pure and compact design allows deeper exploration into sophisticated constructs - whereas microKanren's implementation in Haskell combines a lazy approach with logic programming, quines and SAT solvers offer a purely declarative way to express algebraic formulations. A further extension to the current Haskell implementation that we aim to include is relational type inference (which can aid in type-checking when used in the backward direction).

References

[1] William E Byrd. Relational programming in minikanren: techniques, applications, and implementations. 2010.

[2] William E. Byrd, Eric Holk, and Daniel P. Friedman. minikanren, Live and Untagged: Quine Generation via Relational Interpreters. In Olivier Danvy, editor, *Proceedings of the 2012 Annual Workshop on Scheme and Functional Programming, Scheme 2012, Copenhagen, Denmark, September 9-15, 2012*, pages 8–29. ACM, 2012.

[3] Douglas. Godel, Escher, Bach : An Eternal Golden Braid. 1979.

[4] Jason Hemann, Daniel P. Friedman, William E. Byrd, and Matthew Might. A simple complete search for logic programming. In Ricardo Rocha, Tran Cao Son, Christopher Mears, and Neda Saeedloei, editors, *Technical Communications of the 33rd International Conference on Logic Programming, ICLP 2017, August 28 to September 1, 2017, Melbourne, Australia*, volume 58 of *OASICS*, pages 14:1–14:8. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017.

[5] Weixi Ma Kuang-Chen Lu and Danniel P. Friedman. Towards a minikanren with Fair Search Strategies. 2019.

[6] Simon Cooper Leo Broukhis and Landon Curt Noll. International Obfuscated C Code Contest, http://www.ioccc.org/.

[7] Self. microkanren Implementation in Haskell. https://github.com/kwannoel/uKanren.

VIII. Appendix

```
module MicroKanren ( Stream (..)
                   , Goal
                   , State
                   , Subst
                   , Term (..)
                   , Var
                   , VariableCounter
                   , walk
                   , extS
                   , (===)
                   , unify
                   , fresh
                   , disj
                   , conj
                   , delay
                   -- | Utilities
                   , failure
                   , initialState
                   , takeS
                   , prettyPrintResult
                   , prettyPrintResults
                   ) where

import           Control.Applicative (Alternative (..))
import           Control.Monad       (MonadPlus (..))

type Goal = State → Stream State
type State = (Subst, VariableCounter)
type Subst = [(Var, Term)]
type Var = Integer
type VariableCounter = Integer

data Term = Atom String
          | Var Var
          | Pair Term Term
          deriving (Eq, Show)


walk :: Term → Subst → Term
walk (Var v) s = case lookup v s of Nothing → Var v
                                    Just a  → walk a s
walk (Pair t1 t2) s = Pair (walk t1 s) (walk t2 s)
walk t _ = t

extS :: Var → Term → Subst → Maybe Subst
extS v t s | occurs v t s = Nothing
           | otherwise = Just $ (v, t) : s

(===) :: Term → Term → Goal
(===) t1 t2 = λ(s, vc) → case unify t1 t2 s of
    Nothing → Nil
    Just s' → return (s', vc)

unify :: Term → Term → Subst → Maybe Subst
unify t1 t2 s = go (walk t1 s) (walk t2 s)
    where
        go (Atom a1) (Atom a2) | a1 == a2 = Just s
        go (Var v1) (Var v2) | v1 == v2 = Just s
        go (Var v1) t2' = extS v1 t2' s
        go t1' (Var v2) = extS v2 t1' s
        go (Pair u1 u2) (Pair v1 v2) = unify u1 v1 s >>= unify u2 v2
        go _ _ = Nothing -- Short circuit if we fail to unify
```

```
occurs :: Var → Term → Subst → Bool
occurs v t s = case walk t s of Var tv → tv == v
                                Pair s1 s2 → occurs v s1 s
                                           || occurs v s2 s
                                _ → False

fresh :: (Term → Goal) → Goal
fresh f = λ(s, c) → f (Var c) (s, c + 1)

disj :: Goal → Goal → Goal
disj g1 g2 = λs → g1 s `mplus` g2 s

conj :: Goal → Goal → Goal
conj g1 g2 = λs → g1 s >>= g2

———— Auxiliary functions and types

data Stream a = Nil
              | Cons a (Stream a)
              | Delayed (Stream a) deriving (Eq, Show)

instance Monad Stream where
    return = pure
    Nil >>= _         = Nil
    x `Cons` xs >>= f = f x `mplus` (xs >>= f)
    Delayed s >>= f   = Delayed (s >>= f)

— We can reuse Alternative instances, due to it being a class constraint for MonadPlus
instance MonadPlus Stream where
    mzero = empty
    mplus = (<|>)

— Interleaving rather than appending 2 streams
instance Alternative Stream where
    empty = Nil
    Nil <|> xs           = xs
    (x `Cons` xs) <|> ys = x `Cons` (ys <|> xs)
    Delayed xs <|> ys    = Delayed (ys <|> xs)

— Unused, just to satisfy class constraints for Applicative
instance Functor Stream where
    fmap _ Nil          = Nil
    fmap f (a `Cons` s) = f a `Cons` fmap f s
    fmap f (Delayed s)  = Delayed (fmap f s)

— Unused, just to satisfy class constraints for Monad
instance Applicative Stream where
    pure a = a `Cons` empty
    Nil <*> _            = Nil
    _ <*> Nil            = Nil
    (f `Cons` fs) <*> as = fmap f as <|> (fs <*> as)
    Delayed fs <*> as    = Delayed (fs <*> as)

— | A goal which always fails
failure :: Goal
failure _ = Nil

— | Annotate recursive goals with this
delay :: Goal → Goal
delay = fmap Delayed

———— Pretty printing

prettyPrintResults :: Stream State → String
prettyPrintResults Nil = ""
```

```
prettyPrintResults (Delayed s) = prettyPrintResults s
prettyPrintResults (Cons s ss) = prettyPrintResult (fst s) <> "λn" <> prettyPrintResults ss

prettyPrintResult :: Subst → String
prettyPrintResult = unlines ∘ fmap showBinding
  where showBinding (v, t) = unwords ["Var", show v, ":=", show t]

--------- Tests

initialState :: State
initialState = ([], 0)

tests :: [Goal]
tests = [ atomTest
        , conjTest1
        , conjTest2
        , disjTest1
        , disjTest2
        , fail1
        ]
    where
        atomTest = fresh (λx → x === Atom "5")
        conjTest1 = conj (fresh (=== Atom "7"))
                         (fresh (=== Atom "8"))
        conjTest2 = conj (fresh (=== Atom "7"))
                         (fresh (=== Atom "8"))
        disjTest1 = disj (fresh (=== Atom "7"))
                         (fresh (=== Atom "8"))
        disjTest2 = disj (fresh (=== Atom "8"))
                         (fresh (=== Atom "8"))
        fail1 = fresh (λa → (a === Atom "8") `conj` (a === Atom "9"))

recursiveTests :: [Goal]
recursiveTests = [ takeS 5 <$> recurseDelayedFstTest -- This test terminates since recursive parts are delayed
                 , takeS 5 <$> recurseDelayedSndTest -- This test terminates since recursive parts are delayed
                 , takeS 5 <$> recurseDelayedBothFstTest
                 , takeS 5 <$> recurseDelayedBothSndTest
                 , takeS 5 <$> recurseDelayedEverythingFst
                 , takeS 5 <$> recurseDelayedEverythingSnd
                 -- , takeS 2 <$> recurseFstTest -- This test never terminates
                 -- , takeS 2 <$> recurseSndTest -- This test never terminates
                 ]
    where
        -- These tests fail because we did not make recursion explicit
        recurseFstTest = disj recurseFstTest (fresh (=== Atom "7"))
        recurseSndTest = disj (fresh (=== Atom "7")) recurseSndTest

        recurseDelayedFstTest = disj (delay recurseDelayedFstTest) (fresh (=== Atom "7"))
        recurseDelayedSndTest = disj (fresh (=== Atom "7")) (delay recurseDelayedSndTest)

        -- Delay both
        recurseDelayedBothFstTest = disj (delay recurseDelayedBothFstTest) (delay $ fresh (=== Atom "7"))
        recurseDelayedBothSndTest = disj (delay $ fresh (=== Atom "7")) (delay recurseDelayedBothSndTest)

        -- Delay everything
        recurseDelayedEverythingFst = delay $ disj recurseDelayedEverythingFst (fresh (=== Atom "7"))
        recurseDelayedEverythingSnd = delay $ disj (fresh (=== Atom "7")) recurseDelayedEverythingSnd

takeS :: Int → Stream a → Stream a
takeS 0 _             = Nil
takeS n Nil           = Nil
takeS n (Delayed s)   = Delayed (takeS n s)
takeS n (a `Cons` as) = a `Cons` takeS (n - 1) as

runTests :: State → [Goal] → IO ()
runTests _ [] = return ()
runTests s (g:gs) = do
```

```
    print $ g s
    runTests s gs

main :: IO ()
main = do
    runTests initialState tests
    runTests initialState recursiveTests
```