rKanren

Guided Search in miniKanren

Cameron Swords
Indiana University, USA
cswords@indiana.edu

Daniel P. Friedman
Indiana University, USA
dfried@indiana.edu

Abstract

Traditional relational programming languages often provide a fixed, simple search strategy: Prolog implementations natively provide depth-first search to find answers (though other strategies are often supported) while most miniKanren implementations perform search using a breadth-first approach. In the field of Artificial Intelligence, both of these strategies are often discarded for best-first strategies, including A* Search, Beam Search, Dijkstra's Algorithm, and Uniform Cost Search.

We focus on uniform-cost search, presenting a framework for guided search in the miniKanren relational language embedded in Scheme. This paper revises the traditional miniKanren distributed trampoline implementation, imposing a partial ordering on the search space exploration in order to find answers in a programmer-specified order.

Keywords relational programming, search, minikanren, logic programming

1. Introduction

Most traditional relational programming languages adopt a fixed search strategy to use when exploring solution *search spaces*—the entire domain of possible solutions that must be searched through. These search strategies are taken from the early days of Artificial Intelligence research, and different search strategies have been adopted for languages including Prolog and miniKanren.

The Prolog programming language allows programmers to perform relational programming for theorem proving and artificial intelligence [4, 8, 9]. Prolog has traditionally provided users with a *depth-first search* [4, 20], wherein each search branch is explored until termination before any other branch is explored [19]. This approach works most effectively in finite search spaces with deep answers, as infinite search spaces have search branches that never terminate.

Conversely, miniKanren, a relational programming language based on monadic research into nodeterminism and embedded in Scheme [5, 6, 11, 15], uses a search that most closely resembles *breadth-first search*, wherein a single step is taken down each search branch in turn [19]. This approach will find shallow answers quickly, even in infinite search spaces, but may take more time to find answers that live deeper in the space.

In the field of Artificial Intelligence, both of these strategies are often discarded for either Iterative-Deepening Depth-First Search [16] or Best-First Search [10, 12, 18, 19]. Iterative-Deepening Depth-First Search performs a depth-first search with a limit on the depth—any branch that exceeds that limit is abandoned. The search is run repeatedly, increasing the depth limit with each iteration until the required number of answers is found.

Best-first search associates each path in the tree with a cost and the search always explores the lowest-ranked search path [13, 17, 19]. The path cost is the sum of the distance from the starting node and a cost function or heuristic estimate of the distance from the current state to the goal, determined by examining the current state with a programmer-provided evaluation function. This search has the dual advantages of finding answers earlier in large search spaces and finding more "ideal" answers in problems where multiple answers exist.

In this paper, we present rKanren, an extension of miniKanren for performing guided search using uniform-cost search operation that allows programmers to inform the search space exploration. This implementation revisits the traditional miniKanren stream-merging procedures [11], imposing a partial ordering on the miniKanren search space exploration in order to find answers in a different order.

By allowing users to rank particular search branches, the miniKanren search mechanism can be guided toward answers deeper in the search space without risking the common non-termination problems associated with depth-first search.

MiniKanren's basic usage and implementation are presented in *The Reasoned Schemer* and a number of publications in the literature [2, 5–7, 11]. We assume that the reader is familiar with the language and its basic implementation.

The rest of the paper proceeds as follows:

- Section 2 introduces the `cond`^{r} language form in miniKanren, demonstrating the basic search guidance system in the context of small examples and full recursion.
- Section 3 discusses our changes to miniKanren's internal implementation (including the introduction of stream priority ranks and rank-sensitivity modifications to `mplus`, `bind`, and `case-inf`) to facilitate this new search strategy while preserving the old search forms (including `cond`^{e}).
- Section 4 explores advanced examples and applications of `cond`^{r}, using the new form to address undesirable behavior when synthesizing complex answers.
- Section 5 wraps up and describes potential future works.

2. Deeper Results in miniKanren

In traditional AI research, the logical progression of breadth-first and depth-first search are the techniques of iterated-deepening depth-first search and best-first search [19], which utilizes per-node weights in the search space to guide exploration in directions that the programmer deems more likely to yield an answer.

The most-utilized search technique in classic miniKanren is `cond`^{e}, which performs search space exploration in a breadth-first style.

```
> (run* (q)
    (conde ((≡ q #t)) ((≡ q #f))))

(#t #f)
```

Using `cond`^{e} causes the goals listed to be explored in a first-in, first-out style, exploring the first goal of the first clause, then the first goal of the second clause, and so on. While miniKanren produces a *set* of answers in theory, there are many situations where programmers would like to change the order they receive their answers in practice:

```
> (run* (q)
    (conde ((≡ q #f)) ((≡ q #t))))

(#f #t)
```

Depending on clause ordering to change the answer order is less than ideal: programmers must carefully modify programs to affect the order of the answer stream, and the results may be unexpected. One alternative is to provide a new form, `cond`^{r}, which associates a number value with each clause in order to guide the search:

```
> (run* (q)
    (condr
      (2 (≡ q #f))
      (1 (≡ q #t))))

(#t #f)
```

2.1 Weighted Search

While the searches are associated with a rank, there are a number of other considerations when designing guided searches. The most important is that more goals mean more work, and the work done during a computation contributes to the cost, and thus the computation's rank [13]. Thus if a search path is ranked lower but does more work, the cost increase may cause the answer to occur later.

```
> (run* (q)
    (fresh (a b)
      (condr
        (2 (≡ q #f))
        (1 (≡ q `(,a ,b))
           (≡ a #t)
           (≡ b #t)))))

(#f (#t #t))
```

This additional computation cost may be dealt with by increasing the rank of other branches: each goal contributes one more point of work, so we must increase the first goal to rank 4 in order to produce the expected behavior once again.

```
> (run* (q)
    (fresh (a b)
      (condr
        (4 (≡ q #f))
        (1 (≡ q `(,a ,b))
           (≡ a #t)
           (≡ b #t)))))

((#t #t) #f)
```

The `cond`^{r} form, like `cond`^{e}, is itself a proper goal in the miniKanren system, so it may be used inside of `cond`^{e}. Here, `cond`^{e} interleaves exploration of the two `cond`^{r} clauses, and thus the lower-ranked branches of each are performed before the higher-ranked branches.

```
> (run* (q)
    (fresh (a b)
      (≡ q `(,a ,b))
      (conde
        ((condr
           (2 (≡ a 'a) (≡ b 'b))
           (1 (≡ a 'b) (≡ b 'a))))
        ((condr
           (2 (≡ a 'a) (≡ b 'a))
           (1 (≡ a 'b) (≡ b 'b)))))))

((b a) (b b) (a b) (a a))
```

2.2 Recursive Weights

The original motivation for developing `cond`^{r} was to find specific answers earlier. Large search trees may be generated using recursive functions, but previous miniKanren implementations have been unable to return answers found recursively earlier than *ground answers* listed in the function's body.

Ground answers are answers found at the end of search branches, and *grounding out* indicates arriving at the end of a search path and finding an answer. Consider the following function, written with cond^e, and associated call. It has two recursive options and a single clause (the first one) that grounds the search path.

```
(define recur-e
  (λ (e)
    (fresh (a b)
      (cond^e
        ((≡ e '(x)))
        ((≡ e `(b . ,a)) (recur-e a))
        ((≡ e `(a . ,b)) (recur-e b))))))

> (run 5 (q) (recur-e q))
((x) (b x) (a x) (b b x) (a b x))
```

The cond^e clause that grounds out first—in this case, the first clause—is the first answer, and each subsequent answer is produced in order of the cond^e clauses. If we would like these answers in a different order, we may use cond^r to rank our desired answer order.

```
(define recur-r
  (λ (e)
    (fresh (a b)
      (cond^r
        (10 (≡ e '(x)))
        (4 (≡ e `(b . ,a)) (recur-r a))
        (2 (≡ e `(a . ,b)) (recur-r b))))))

> (run 5 (q) (recur-r q))
((x) (a x) (b x) (a a x) (b a x))
```

This result may be somewhat unexpected: though the first clause is ranked higher, and thus should occur later than the other clauses, it is still produced as the first answer. Luckily, the intuition here is straight-forward: the other two clauses must eventually ground out in a recursive call (via the first clause), and ranks are cumulative. As a result, the answer `(x)` has rank 10 while the answer `(a x)` has rank 12 and the answer `(b x)` has rank 14[1].

If we would like complex answers earlier, we must change the *cost* of grounding during later recursive calls. This is a simple fix: each rank is a full Scheme expression that evaluates to a natural number, so we may use an extra parameter to keep track of our recursive depth and use the information to change the grounding cost:

```
(define recur-r-n
  (λ (e n)
    (fresh (a b)
      (cond^r
        ((if (< n 1) 10 1)
          (≡ e '(x)))
        (4 (≡ e `(b . ,a))
           (recur-r-n a (add1 n)))
        (2 (≡ e `(a . ,b))
           (recur-r-n b (add1 n)))))))

> (run 5 (q) (recur-r-n q 0))
((a x) (b x) (a a x) (x) (a b x))
```

[1] The actual ranks are larger because each step contributes to the rank (as previously discussed), but the intuition here should be sufficient for program reasoning.

$(\mathbf{cond^r}$
  $(\langle exp \rangle \langle exp \rangle \langle exp \rangle^*)$
  $(\langle exp \rangle \langle exp \rangle \langle exp \rangle^*)^*)$

Figure 1. Formal syntax for **cond**^{r}

Our recursion tracks the depth, and after the first two steps the grounding cost is reduced from 10 to 1, causing the answers `(a x)`, `(b x)`, and `(a a x)` each to have a total cost lower than 10. This technique allows users to explore deeper branches of the search space before considering the shallow answers.

2.3 Formal Syntax and Behavior

In the cond^r form, each clause of goals is associated with a Scheme expression that evaluates to some kind of natural number[2]. These numbers, referred to as *priorities* or *ranks*, are then used to guide the search space exploration by always exploring the numerically lowest-ranked path.

The formal syntax is given in Fig. 1. Here, the first $\langle exp \rangle$ of each set of clauses is any Scheme expression that evaluates to a natural number and every subsequent $\langle exp \rangle$ is any valid miniKanren goal (as with cond^e).

The behavior of cond^r is modeled after the best-first search technique, where the traditional heuristic approach of A* search has been discarded in favor of simple Scheme expressions that evaluate to numeric values, producing classic uniform-cost search. Uniform-cost search still requires terminating search trees: consider revising `recur-p` by ranking the first clause with one hundred: 100 recursive steps would need to be taken before any ground answers were produced, and yet the answer order would be identical to the one presented.

In this way, the cond^r form still performs a *complete search*: unlike the cond^a and cond^u forms that provide users some control over search paths but ultimately discards certain answers, cond^r will find every answer in finite search spaces. Furthermore, cond^r strictly subsumes cond^e: cond^e's behavior may be reproduced by replacing each rank in cond^r with the number 0.

3. Implementation

Changing the implicit search strategy of miniKanren requires modifying a number of internal functions that are often kept far from the user, including the stream composition structures and goal exploration mechanisms. We perform these changes in several stages:

- We convert streams and substitutions to records which have associated "ranks" in order to evaluate which should be dealt with next. We also introduce additional operators to perform rank operations.

[2] It is possible to use negative numbers, but this may lead to difficulty when reasoning about result order.

- We rewrite the `mplus` operators to always invoke the branch with the lowest rank.
- We modify the internal `case-inf` macro to systematically count each step a search path takes to ensure control is properly handed off.
- We rewrite the custom lambda forms, `cond`^{e}, `run`, and `fresh`, and implement `cond`^{r}.

A thorough explanation of the initial miniKanren implementation may be found in Alvis, et. al. [2].

3.1 Recording the Ranks

The first step toward guided search is to associate each stream and substitution (which is treated as a stream of length one) with a rank by converting each into a record that contains the original contents and an additional field for rank:

```
(define-record stream (rank proc))
(define-record subst (rank alist diseq))
```

There are also a number of helpers provided in the appendix to make subsequent changes easier (including `invoke`, `get-rank`, `incr-rank`, and more). We primarily use these helpers for inspecting or modifying the records' ranks, but `invoke` extracts and invokes a stream's procedure.

3.2 Rebalancing the Trampoline

In miniKanren, the search is ordered by suspending each search path as a *thunk*, a function of no arguments [14], which is internally treated as a *stream* of answers [1]. These streams are combined by a *binary trampoline*—a trampoline [3] that alternates between invoking two thunks.

By nesting these trampolines, miniKanren produces a lazy, binary tree that represents streams and substitutions as nodes. The classic miniKanren search traverses this tree a breadth-first search strategy. In the original miniKanren implementation, `mplus` performs this search, culling up answers returned during invocation. In order to guide the search, we must reform this tree structure to build a lazy, binary min heap. Each time we invoke a node of the heap, the trampoline mechanism examines the rank associated with the two sub-nodes and invokes the node with the lowest rank at each step. In the case of ties, the last-invoked stream relinquishes control.

The implementation presented in Fig. 2 is straight forward: any time streams are involved, their ranks are compared and the lower one is chosen as the next one to invoke. The entire stream is given the lower rank when constructing the stream to be returned (because that is the rank of the path to be expanded when the outer thunk is invoked). The expanded form, `mplus`*, performs a similar operation for an entire list of possible streams, choosing the appropriate rank and building the appropriate answer.

```
(define mplus
  (λ (a-inf f)
    (case-inf a-inf
      (() (invoke f))
      ((f')
       (let ((f'-rank (get-rank f'))
             (f-rank (get-rank f)))
         (if (< f'-rank f-rank)
           (thunk f'-rank (mplus (invoke f') f))
           (thunk f-rank (mplus (invoke f) f')))))
      ((a) (choice a f))
      ((a f')
       (choice
         a
         (let ((f'-rank (get-rank f'))
               (f-rank (get-rank f)))
           (if (< f'-rank f-rank)
             (λf () f'-rank (mplus (invoke f') f))
             (λf () f-rank
                    (mplus (invoke f) f'))))))))))

(define-syntax mplus*
  (syntax-rules ()
    ((_ e) e)
    ((_ e0 e ...)
      (let ((min-rank
              (apply min (map get-rank `(,e ...))))
            (e0 -rank (get-rank e0 )))
        (if (< e0 -rank min-rank)
          (mplus e0
                 (λf () min-rank (mplus* e ...)))
          (mplus (λf () min-rank (mplus* e ...))
                 (λf () e0 -rank e0 )))))))
```

Figure 2. A rank-sensitive implementation of mplus.

We must also update `bind` to make use of `get-rank` and `invoke`:

```
(define bind
  (λ (a-inf g)
    (case-inf a-inf
      (() (mzero))
      ((f) (thunk (get-rank f) (bind (invoke f) g)))
      ((a) (g a))
      ((a f)
       (mplus (g a)
              (λf () (get-rank f)
                     (bind (invoke f) g)))))))
```

3.3 Rigid Rank Increase

Next, we must ensure that every time a step is taken in the system, the appropriate rank is increased. In this instance, we are in luck: every goal step in the system is performed using the `case-inf` macro, so we need only modify it to ensure that every step will increase the rank of the appropriate stream. In each case, we increment the rank of whatever stream (or answer) would be returned via `incr-rank` (defined in the appendix), ensuring that the next time it is seen by `mplus` it will indicate that it has done additional work.

```
(define-syntax case-inf
  (syntax-rules ()
    ((_ e (() e0 ) ((f') e1 ) ((a') e2 ) ((a f) e3 ))
     (let ((a-inf e))
       (cond
         ((not a-inf) (incr-rank e0 ))
         ((stream? a-inf)
          (let ((f' a-inf)) (incr-rank e1 )))
         ((not (and (pair? a-inf)
                    (stream? (cdr a-inf))))
          (let ((a' a-inf)) (incr-rank e2 )))
         (else (let ((a (car a-inf)) (f (cdr a-inf)))
                 (incr-rank e3 ))))))))
```

3.4 Ranked Revelry

After performing these modifications, we must revisit the λ_f and `thunk` forms of the initial implementation: streams must now be associated with ranks, and these macros build the streams. We revise them as follows:

```
(define-syntax λf
  (syntax-rules ()
    ((_ () r e) (make-stream r (λ () e)))))

(define-syntax thunk
  (syntax-rules ()
    ((_ r e) (make-stream r (λ () e)))))
```

Now we have incorporated ranks through the miniKanren implementation, and we may express `cond`^{r}:

```
(define-syntax cond^r
  (syntax-rules ()
    ((_ (r0  g0 g ...) (r1  g1 g' ...) ...)
     (λg (s)
       (thunk (subst-rank s)
         (let ((s (subst-incr-rank s)))
           (mplus*
             (bind*
               (g0  (subst-add-rank s r0 )) g ...)
             (bind*
               (g1  (subst-add-rank s r1 )) g' ...)
             ...)))))))
```

The `cond`^{r} macro culls up each new rank (`r0` , `r1` , and so forth) and adds each individually to the rank of the substitution before passing the now-properly-ranked version into the appropriate `bind`* form. The `thunk` being built will dispatch to `mplus`* when invoked, returning a stream of the lowest rank.

3.5 Wrapping Up

There are a few last pieces of bookkeeping: we must update the `run` macro to reflect this change, seeding the initial λ_f with rank 0. We also need to modify the `fresh` macro to produce a rank for its body, which we take from the substitution passed in:

```
(define-syntax fresh
  (syntax-rules ()
    ((_ (x ...) g0 g ...)
     (λg (s)
       (thunk (subst-rank s)
         (let ((x (var 'x)) ...)
           (bind* (g0  s) g ...)))))))
```

Finally, we must rewrite `cond`^{e} to correctly interact with these threaded rank values. The implementation is almost identical to `cond`^{r}, but we treat each clause as if it had an implicit rank of 0:

```
(define-syntax cond^e
  (syntax-rules ()
    ((_ (g0  g ...) (g1  g' ...) ...)
     (λg (s)
       (thunk (subst-rank s)
         (let
           ((s (subst-incr-rank s)))
           (mplus*
             (bind* (g0  s) g ...)
             (bind* (g1  s) g' ...) ...)))))))
```

4. Delving Deeper

The original motivation for controlling miniKanren's search order was to find "interesting" answers quickly: when open-ended miniKanren programs are run, they easily produce simplistic answers and must perform much larger searches to find complex ones.

One such example is a *type inferencer*—a program that takes a program as input and produces the program's type as output, ensuring it is well-typed in the process. In miniKanren, it is straightforward to encode a basic type inferencer for the lambda calculus; such an inferencer is presented below. This inferencer uses tagged application (along with tagged variables and integers) to derive the type of expressions.

```
(define ⊢o
  (λ (Γ  e t)
    (fresh (e1  e2 e3 t1 t2 )
      (conde
        ((≡ e `(intc ,e1 )) (≡ t 'int))
        ((≡ e `(+ ,e1  ,e2 )) (≡ t 'int)
         (⊢o Γ e1 'int)
         (⊢o Γ e2 'int))
        ((≡ e `(var ,e1 )) (lookupo Γ e1 t))
        ((≡ e `(λ (,e1 ) ,e2 ))
         (≡ t `(→ ,t1  ,t2 ))
         (⊢o `((,e1  . ,t1 ) . ,Γ ) e2 t2 ))
        ((≡ e `(app ,e1  ,e2 ))
         (⊢o Γ e1 `(→ ,t1  ,t))
         (⊢o Γ e2 t1 ))))))

(define lookupo
  (λ (Γ  x t)
    (fresh (rest type y)
      (conde
        ((≡ `((,x . ,t) . ,rest) Γ ))
        ((≡ `((,y . ,type) . ,rest) Γ )
         (≢ x y)
         (lookupo rest x t))))))
```

The type inferencer encodes the basic lambda calculus typing rules, using a type environment Γ (represented as an association list) to derive expression types. The `lookup`^{o} procedure performs look-ups in the association list, and we use the disequality constraint $\not\equiv$ to ensure we recur only when `x` may not be unified with `y`.

Using this inferencer and a pair of fresh variables, we may ask miniKanren to generate 4 well-typed expressions:

```
> (run 4 (q)
    (fresh (e t) (≡ q `(,e : ,t)) (⊢ᵒ '() e t)))
(((intc _.0) : int)
 ((λ (_.0) (intc _.1)) : (→ _.2 int))
 ((λ (_.0) (var _.0)) : (→ _.1 _.1))
 ((+ (intc _.0) (intc _.1)) : int))
```

These expressions are incredibly basic, however: the breadth-first approach of $\texttt{cond}^e$ finds the shallow answers first, thus producing the simplest expressions first. Using our new $\texttt{cond}^r$, we may change which answers are found first, producing more complex answers first.

A revised implementation, given below, uses a recursive depth counter, as described in §2.2. In our implementation, the application and lambda expressions are given low cost while variables and integers decrease in cost during deeper recursive calls.

```
(define ⊢ʳ
  (λ (Γ e t n)
    (let ((n (add1 n)))
      (fresh (e₁ e₂ e₃ t₁ t₂)
        (condʳ
          ((if (< n 3) 30 1)
           (≡ e `(intc ,e₁))
           (≡ t 'int))
          ((if (< n 3) 30 1)
           (≡ e `(+ ,e₁ ,e₂))
           (≡ t 'int)
           (⊢ʳ Γ e₁ 'int n)
           (⊢ʳ Γ e₂ 'int n))
          ((if (< n 4) 30 15)
           (≡ e `(var ,e₁))
           (lookupᵒ Γ e₁ t))
          (4 (≡ e `(λ (,e₁) ,e₂))
           (≡ t `(→ ,t₁ ,t₂))
           (⊢ʳ `((,e₁ . ,t₁) . ,Γ) e₂ t₂ n))
          (2 (≡ e `(app ,e₁ ,e₂))
           (⊢ʳ Γ e₁ `(→ ,t₁ ,t) n)
           (⊢ʳ Γ e₂ t₁ n)))))))
```

We once again ask miniKanren to generate 4 well-typed expressions:

```
> (run 4 (q)
    (fresh (g e t) (≡ q `(,e : ,t))
                   (⊢ʳ '() e t 0)))
(((λ (_.0) (λ (_.1) (intc _.2)))
  : (→ _.3 (→ _.4 int)))
 ((λ (_.0) (λ (_.1) (λ (_.2) (intc _.3))))
  : (→ _.4 (→ _.5 (→ _.6 int))))
 ((λ (_.0) (app (λ (_.1) (intc _.2)) (intc _.3)))
  : (→ _.4 int))
 ((app (λ (_.0) (intc _.1)) (λ (_.2) (intc _.3)))
  : int))
```

The cost-based search guidance leads miniKanren to delve deeper into the search space before grounding out, and the result is a list of answers with increased complexity. By actively seeking out a certain shape of answers, we are able to explore the more interesting areas of the search space.

5. Conclusion and Future Work

The field of search strategies is well developed, and Prolog has long had many ways to encode various search strategies [8, 20]. Growing miniKanren's language features to provide a wider range of applications is important to make it an effective, competitive relational language, and providing a new search strategy that encompasses the current, widely-used search mechanism should prove invaluable in future development. Using this new search strategy, it is possible to concretely predict the order of clause evaluation in miniKanren and, as a result, better manage the expected outcome of programs.

Future work includes:

- Encoding depth-first search using $\texttt{cond}^r$. Each clause will have to relate to a global counter, ensuring that branches are assigned lowering priorities in the order they are discovered, meaning that the most recently-discovered search path is always the one explored next.
- Developing straight-forward syntax that enables programmers to define full heuristic functions in place of numeric ranks, providing the heuristic function with sufficient information (such as the current substitution and recursion rank) to be admissible, recovering full A* search.
- Fine-tuning clause rank choices, producing a short guide to help users understand what numeric values are ideal in various situations.

References

[1] Harold Abelson and Gerald Jay Sussman. *Structure and Interpretation of Computer Programs - 2nd Edition (MIT Electrical Engineering and Computer Science)*. The MIT Press, 1996.

[2] Claire E. Alvis, Jeremiah J. Willcock, Kyle M. Carter, William E. Byrd, and Daniel P. Friedman. ckanren: minikanren with constraints. In *Proceedings of the 2011 Scheme and Functional Programming Workshop*, 2011.

[3] Henry G. Baker. Cons should not cons its arguments, part ii: Cheney on the m.t.a., 1994.

[4] Ivan Bratko. *Prolog Programming for Artificial Intelligence*. Addison Wesley, 2000.

[5] William E. Byrd. *Relational programming in miniKanren: techniques, applications, and implementations*. PhD thesis, Indiana University, Indianapolis, IN, USA, 2009. AAI3380156.

[6] William E. Byrd and Daniel P. Friedman. From variadic functions to variadic relations: A minikanren perspective. In *Proceedings of the 2006 Scheme and Functional Programming Workshop*, pages 105–117, 2006.

[7] William E. Byrd, Eric Holk, and Daniel P. Friedman. minikanren, live and untagged - quine generation via relational interpreters. In *Proceedings of the 2012 Scheme and Functional Programming Workshop*, 2012.

[8] William F. Clocksin and Christopher S. Mellish. *Programming in Prolog: Using the ISO Standard*. Springer, 2003.

[9] Alain Colmerauer and Philippe Roussel. The birth of prolog. In *The second ACM SIGPLAN conference on History of programming languages*, HOPL-II, pages 37–52, New York, NY, USA, 1993. ACM.

[10] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, December 1959.

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

[12] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems, Science, and Cybernetics*, SSC-4(2):100–107, 1968.

[13] P.E. Hart, N.J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *Systems Science and Cybernetics, IEEE Transactions on*, 4(2):100 – 107, july 1968.

[14] P. Z. Ingerman. Thunks: a way of compiling procedure statements with some comments on procedure declarations. *Commun. ACM*, 4(1):55–58, January 1961.

[15] Oleg Kiselyov, Chung-chieh Shan, Daniel P. Friedman, and Amr Sabry. Backtracking, interleaving, and terminating monad transformers: (functional pearl). *SIGPLAN Not.*, 40(9):192–203, September 2005.

[16] Richard E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27:97–109, 1985.

[17] Allen Newell and Herbert A. Simon. Computer science as empirical inquiry: symbols and search. *Commun. ACM*, 19(3):113–126, March 1976.

[18] Judea Pearl. *Heuristics: Intelligent Search Strategies for Computer Problem Solving (The Addison-Wesley series in artificial intelligence)*. Addison-Wesley Pub (Sd), 1984.

[19] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach (3rd Edition)*. Prentice Hall, 2009.

[20] Ehud Sterling and Leon Shapiro. *The Art of Prolog, Second Edition: Advanced Programming Techniques (Logic Programming)*. The MIT Press, 1994.

Helper Functions

There are a number of simple helper functions used throughout the implementation and provided here. Most of the definitions are wrappers around record constructors and getters, and we use them for syntactic sugar.

The definitions `get-rank` and `incr-rank` both deal with their input in a style similar to `case-inf`: the former simply extracts the rank (or returns `-1` in the case of failure, ensuring immediate failure) and the latter increments the rank on whatever it is handed (if such an operation is possible).

The function `lookup-s` and definition `empty-a` have been updated to reflect the record type of substitutions, and the `invoke` function has been provided to ease the loss of raw thunks in the implementation.

```
(define get-rank
  (λ (a-inf)
    (cond
      ((stream? a-inf) (stream-rank a-inf))
      ((subst? a-inf) (subst-rank a-inf))
      (else -1))))

(define incr-rank
  (λ (a-inf)
    (cond
      ((stream? a-inf) (stream-incr-rank a-inf))
      ((subst? a-inf) (subst-incr-rank a-inf))
      ((pair? a-inf)
       (cons
         (incr-rank (car a-inf))
         (incr-rank (cdr a-inf))))
      (else a-inf))))

(define subst-add-rank
  (λ (s r)
    (make-subst
      (+ (subst-rank s) r)
      (subst-alist s)
      (subst-diseq s))))

(define subst-incr-rank
  (λ (s)
    (make-subst
      (add1 (subst-rank s))
      (subst-alist s)
      (subst-diseq s))))

(define stream-incr-rank
  (λ (stream)
    (make-stream
      (add1 (stream-rank stream))
      (stream-proc stream))))

(define empty-a (make-subst 0 '() '()))

(define lookup-s
  (λ (u S)
    (assq u (subst-alist S))))

(define invoke
  (λ (stream)
    ((stream-proc stream))))
```