1 Programação Relacional em Scheme

Uma pessoa só tem certeza daquilo que constrói

Giambattista Vico - Historiador italiano, século XVIII

Neste capítulo, veremos uma linguagem de programação lógica com um sabor diferente do Prolog que vimos no início, chamada *miniKanren*, e veremos como implementá-la, por meio da linguagem *Scheme*.

Usaremos *Scheme* porque é uma linguagem pequena, o que significa que sua apresentação será curta, e que é, ao mesmo tempo, poderosa, o que significa que não precisaremos de muito código para fazer o que nos propomos.

Esse capítulo tem um sabor diferente dos demais. A diferença pode ser vista rapidamente, pela cara do código* para , mas também de várias outras formas. Redefiniremos alguns termos usados anteriormente. Essas redefinições terão semelhanças e diferenças às definições originais, mas escolhemos não explicitar essas diferenças aqui, por acreditarmos serem claras o suficiente sem um comentário a mais. Outro ponto que vale nota é que, aqui, buscamos apenas um maior entendimento e, para tanto, tentamos deixar o código como "implementado em primeiros princípios", isto é, de forma mais simples. Isso significa evitar construções que poderiam deixá-lo mais eficiente, mas que requeririam uma discussão maior, o que seria às custas de uma divergência de atenção aos pontos principais que queremos passar. Para uma implementação que tenta ser eficiente, veja, por exemplo [1].

1.1 Introdução ao Scheme

Scheme é um *Lisp*†. O termo *Lisp* é às vezes usado para se referir a uma linguagem de programação, mas o mais correto seria ser usado para se referir a uma família de linguagens (de fato, dezenas de linguagens), todas com algumas características em comum, em particular:

- São linguagens multi-paradigma, mas como um foco no paradigma de programação funcional, o que significa, entre outras coisas, que funções são "cidadãos de primeira classe";
- Todo código *Lisp* (que não tenha erro de sintaxe) é avaliado para algum valor no momento de execução;

*Em particular, nota-se que usamos aqui, para os códigos, fontes e cores diferentes das usadas nos capítulos anteriores. Isso é, principalmente, porque entendemos que, aqui, a utilização de cores e artifícios como o **negrito** tenderiam a atribuir atenção especial para elementos que não precisam de atenção especial.

†De *LISt Processing*.

- Programas são expressos em "notação polonesa" (notação prefixada), em formato de listas[*](listas em *Lisps* são delimitadas por parenteses)[†], por exemplo:
  `(+ 1 2)` ⇒ 3;

- O que nos leva a outro ponto: não existe diferença sintática entre a forma como programas *Lisp* são escritos e a forma como suas estruturas de dados são representadas. Diz-se, assim, que *Lisps* são homoicônicas (vale dizer, Prolog também é uma língua homoicônica), o que significa que a diferença entre dados e programa é "borrada", e programas podem, e frequentemente são, manipulados livremente.

Agora, vamos rapidamente introduzir a sintaxe principal de *Scheme*, com alguns exemplos[‡]:

- Listas são representadas como `(`el_1 el_2 ... el_n`)`, em que el_i é o i-ésimo elemento da lista. No entanto, se escrevêssemos uma lista assim, ela seria confundida com uma aplicação de função (aplicação da função el_1 aos argumentos el_2 a el_n), então, para fins de desambiguação, é usada uma aspa simples, e a lista é escrita como '$(el_1 el_2 ... el_n)$[§], que é equivalente a `(list '`el_1 ... `'`el_n`)` . Essa aspa simples também pode ser usada para "evitar que um objeto seja avaliado"[¶]:
  A linha
  `a`
  resulta em erro se `a` não for uma variável, já que o executor do código tentará avaliá-la (gerar um valor a partir dela), mas `a` não tem valor associado. Mas a linha
  `'a` ⇒ `'a`
  é avaliada, "como ela mesma".

- Uma estrutura de dados mais geral do que lista em *Scheme* é o que é chamado *cons pair* (que nós chamaremos daqui para frente simplesmente de "par"). A lista `'(a b c d)` é equivalente a `(cons a (cons b (cons c (cons d '()))))`, em que `'()` é a lista vazia Assim *cons* constrói uma estrutura de dados formada por um par. Para obter o primeiro elemento do par, usa-se *car* e, para obter o segundo, *cdr*[‖]. Temos, por exemplo,

[*] Também conhecidas como *S-Expressions*, ou *Sexps*.

[†] Usaremos daqui em diante a notação **codigo** ⇒ **valor** para denotar que o código **código** avalia para o valor **valor**.

[‡] Só introduziremos a parte da linguagem que nos será relevante, o que não é a linguagem inteira.

[§] Note que, por ser fechada por parênteses, apenas uma aspa é o suficiente.

[¶] Uma colocação mais correta seria "tornar objetos auto-avaliantes", mas não precisamos entrar em muitos detalhes de como isso funciona. Para nós é suficiente dizer que 'a é um símbolo.

[‖] Esses nomes têm uma origem histórica: eram nomes de registradores quando os primeiros

(car (cons 1 (cons 2 3))) $\Rightarrow$ 1
(cdr (cons 1 (cons 2 3))) $\Rightarrow$ (cons 2 3) = '(2 . 3)
(car '(1 2 3 4)) $\Rightarrow$ 1
(cdr '(1 2 3 4)) $\Rightarrow$ '(2 3 4)
Com isso, podemos definir um lista indutivamente como sendo ou a lista vazia, '(), ou um par, cujo *cdr* é uma lista*.

- Para definir funções, use **lambda**, ou λ[†]:
  ((λ (a b) (/ a b)) 1 3) $\Rightarrow$ 1/3n

- Para definir constantes, use **define**:
  (define divide (λ (a b) (/ a b)))[‡]
  (divide 1 3) $\Rightarrow$ 1/3
  (define c (divide 1 3))
  (divide (divide 1 9) c) $\Rightarrow$ 1/3
  (define (divide3 (divide3 a) (divide a 3)))
  (divide3 9) $\Rightarrow$3[§],;

- Um ponto importante, que usaremos muito logo mais é que, se quisermos criar listas com os valores das variáveis, no lugar de nomes simbólicos, podemos usar, no lugar da aspa simples, a crase e preceder o nome da variável com uma vírgula:
  (define x 10)
  `(1 2 'x ,x) $\Rightarrow$ '(1 2 'x 10)

- Para realizar execuções condicionais, use **cond**:

```
(cond
  ((< 1 0) (+ 3 4))
  ((< 0 1) (- 3 4))
  (else 0))
```

Lisps estavam sendo criados (vale notar que o primeiro Lisp foi também uma das primeiras linguagens de alto-nível ainda em uso, tendo surgido pouco depois do Fortran).

*Note que (cons 2 3), por exemplo, não é uma lista. Esse tipo de estrutura é chamada *dotted list*, porque, para distingui-la de uma lista, é costumeiramente impressa como '(2 . 3), mas, assim como com Prolog, é uma estrutura de dados diferente, que tem o nome *dotted **list*** devido a uma aparência como que acidental.

†Editores de texto atuais podem aceitar os dois tipos de entrada, mas optamos por usar λ. Este uso do símbolo tem a seguinte origem: Bertrand Russel e Alfred Whitehead buscaram, no início do século XX, lançar as bases lógicas da matemática em seu trabalho *Principia Mathematica*. Lá, para denotar que uma variável é livre, ela recebia um chapéu, como em $\hat{a}(a + y)$. Mais tarde, ainda trabalhando nos fundamentos da matemática, Alonzo Church achou que seria mais conveniente ter fórmulas crescendo linearmente na horizontal (note que o "chapéu" faz com que a fórmula cresça para cima), então decidiu mover o chapéu para o lado, obtendo $\hat{}a(a + y)$. Mas o chapéu flutuando parece engraçado, então Church o trocou pelo o símbolo não usado mais próximo que tinha, um Λ, como em $\Lambda a(a + y)$. Mas Λ tem uma grafia muito parecida com outra letra comum, o que foi percebido como um incoveniente, então ele acabou eventualmente trocando para λ em sua teoria, que acabou se chamando *cálculo* λ [6].

‡Um *açucar sintático* para essa construção é (define (divide a b) (/ a b)).

§Esse tipo de uso é chamado *currying*, e é possível porque *Scheme* tem fecho léxico.

⇒ -1

Podem ser adicionadas quantas cláusulas do tipo `((condicao)(efeito))` se quiser (vale notar, elas são avaliadas sequencialmente), sendo que a última pode opcionalmente ser como `(else (efeito))`, ou `(#t (efeito))`.

- Para adicionar variáveis locais, use *let*:

```
(let ((a (+ 3 4))
      (b (cons 1 2)))
  (+ a (car b)))
```

⇒ 8

O *let* tem duas partes, a de definições, da forma `((variavel valor)(variavel valor) ... (variavel valor))`* e, em seguida, a parte de valor, que nos dá o valor que *let* assume.

Dada essa introdução, faremos uso dessas e outras construções da linguagem sem maiores comentários (exceto quando uma construção especialmente difícil ou complexa o justificar). Para uma introdução mais compreensiva à linguagem, veja [5].

1.2 A linguagem miniKanren

Nosso objetivo aqui é implementar miniKanren, uma linguagem de programação relacional. No lugar de descrever toda a linguagem e depois implementá-la, seguimos pelo caminho de mostrar um pequeno exemplo do que esperamos conseguir e, então, implementamos a linguagem. A esperança é que essa abordagem ofereça maior entendimento dos conceitos explorados.

O tipo de coisa que queremos poder fazer com miniKanren é como o seguinte†:

```
(defrel (teacupᵒ t)
  (disj₂ (≡ 'tea t) (≡ 'cup t)))
(run* x
  (teacupᵒ x))
```

*Podem ser adicionadas quantas variáveis se quiser. As atribuições são feitas “paralelamente” (o que significa que a atribuição de valor a uma variável não influencia no da outra, o que pode ser feito de forma paralela, no sentido usual, ou não).

†Usaremos as convenções da literatura de usar subescritos e sobrescritos e símbolos matemáticos, como ≡, para representar as relações, na esperança de que isso clarifique a notação e deixe o texto menos pesado. Em particular, para diferenciar um objeto relacional de um funcional, o relacional terá um “o” sobrescrito (como em $relacao^o$). Ao escrever o programa para o computador ler, os sobrescritos e subescritos que forem alfa-numéricos ou “*” podem ser escritos normalmente, na frente do termo (como em *relacaoo*, ou *run**). O símbolo ≡ é escrito “==” e, termos como $termo^\infty$, “$termo-inf$”. Ademais, #u e #s devem ser trocados por *fail* e *succeed*, respectivamente.

⇒ `'(tea cup)`

Veremos mais exemplos quando o construirmos. A construção que se segue é em grande parte baseada em [2]. Para conferir detalhes de implementações completas, veja [4].

Como visto no primeiro exemplo, não seguimos, como em Prolog, uma convenção de nomeação de variáveis (em Prolog, a convenção era de que variáveis são capitalizadas). Assim, precisamos de algo para discerni-las e, para tal fim, o que usamos é o *fresh*, informando que a variável é "fresca".

Lembre-se que uma variável relacional não é a mesma coisa que uma variável em uma linguagem de programação tradicional (não relacional). Para definirmos uma variável única, vamos precisar de*

```
(define (var name)(vector name))
```

Usaremos também a seguinte definição:†

```
(define (var? name)(vector? name))
```

Para evitar problemas como os de colisão de variáveis, as variáveis são locais, assim como em *Scheme*. Precisamos, então, de uma forma de modelar isso (note que a definição acima não reflete isso) e, o que usamos é o seguinte:

```
(define (call/fresh name f)
  (f (var name)))
```

Essa função‡, espera, como segundo argumento, uma expressão λ, que recebe como argumento uma variável e produz como valor um goal, o qual tem acesso à variável criada.

Precisamos, agora, saber como associar um valor a uma variável. Diremos que o par `` `(,z . a) `` é uma associação de a à variável z. Mais em geral, um par é uma associação quando o seu *car* for uma variável.

Uma lista de associações será chamada *substituição*.

Na substituição `` `((,x . ,z)) ``, a variável x é "fundida" (ou, na nossa linguagem anterior, unificada) à variável z. A substituição vazia é simplesmente uma lista vazia: `(define empty-s '())`. Nem toda lista de associações é uma substituição, no entanto. Isso porque, não aceitamos, em nossas substituições, associações com o mesmo *car*. Então, o seguinte não é uma substituição: `` `((,z . a)(,x . c)(,z . b)) ``.

Precisamos de dois procedimentos importantes para lidar com substituições: um para estendê-las e um para obter o valor de uma variável presente nela.

*Usamos *vector* para que a unicidade da variável seja definida por sua posição de memória. Outra opção seria distingui-las por valor, se nos assegurássemos de que seu valor é único.

†Símbolos como "?" podem ser usados no meio do código da mesma forma que outros, tais como "a" ou "b".

‡"Método", ou "*procedure*", como é mais popularmente conhecido na comunidade Scheme.

Para obter o valor associado a *x*, usamos *walk*, que deve se comportar como o seguinte:

```
(walk y
  `((,z . a)(,x . ,w)(,y . ,z)))
```

⇒ *a*

porque *y* está fundido a *z*, que está associado a *a*. O *walk* é como se segue*:

```
(define (walk v s)
  (let ((a (and (var? v)(assv v s))))
    (cond
      ((pair? a)(walk (cdr a) s))
      (else v))))
```

Esse código faz uso de *assv*, que ou produz *#f*, se não há associação cujo *car* seja *v* na substituição *s*, ou produz o *cdr* de tal associação. Perceba que, se *walk* produz uma variável como valor, ela é necessariamente fresca (isto é, que não foi associada).

Para estender uma substituição, usamos *ext-s*, que faz uso de *occurs?*:

```
(define (ext-s x v s)
  (cond
    ((occurs? x v s) #f)
    (else (cons '(,x . ,v) s))))

(define (occurs? x v s)
  (let ((v (walk v s)))
    (cond
      ((var? v) (eqv? v x))
      ((pair? v)
        (or (occurs? x (car v) s)
            (occurs? x (cdr v) s)))
      (else #f))))
```

Esse *occurs?* realiza o "teste de ocorrência" (aquele que, como mencionamos anteriormente, não é feito por padrão no Prolog por razões de eficiência, e que faz com que substituições do tipo `` `((,y . (,x))(,x . (a . ,y)) `` sejam inválidas).

Com isso em mãos, podemos definir nosso unificador:

*Lembre-se que `(and a b)` ⇒ b, se $a \neq \#f$.

```
(define (unify u v s)
  (let ((u (walk u s))(v (walk v s)))
    (cond
      ((eqv? u v) s)
      ((var? u) (ext-s u v s))
      ((var? v) (ext-s v u s))
      ((and (pair? u) (pair? v))
       (let ((s (unify (car u)(car v) s)))
         (and s
           (unify (cdr u)(cdr v) s))))
      (else #f))))
```

1.3 Streams

Antes de continuarmos, precisamos tocar no modelo de avaliação de Scheme. Scheme é "uma linguagem de ordem aplicativa", o que significa que, quando uma função é avaliada, seus argumentos são todos avaliados no momento de aplicação. Por esse motivo, os *and* e *or* usuais, por exemplo, não podem ser funções em Scheme*. Uma alternativa à ordem aplicativa é a "ordem normal", que outras linguagens de programação funcional adotaram. Linguagens de ordem normal só avaliam o argumento de uma função quando esse argumento for usado, "atrasando" a avaliação do mesmo (no que é chamado "avaliação tardia", ou "avaliação preguiçosa").

Avaliação preguiçosa é conveniente em diversas ocasiões e pode ser emulada em linguagens de programação funcional de ordem aplicativa† pelo uso de **streams**. *Streams* são definidos indutivamente como sendo ou a lista vazia, ou um par cujo *cdr* é um *stream*, ou uma suspenção. Uma **suspenção** é uma função do tipo `(λ () corpo)`, em que `((λ () corpo))` é uma *stream*. Agora, se fizermos

streams

suspenção

```
(define (≡ u v)
  (λ (s)
    (let ((s (unify u v s)))
      (if s `(,s) '()))))
```

temos que ≡ produz um goal. Dois outros goals, *sucesso* e *falha*, são denotados #s e #u:

```
(define #s
  (λ (s)
    `(,s)))
```

*Se fossem, `(or #t a)`, por exemplo, poderia gerar erro quando `a` não for uma variável. Como *or* não é uma função, o *a* nessa linha não chega a ser avaliado, e temos `(or #t a)` ⇒#t.

†Com fecho léxico.

```
(define #u
  (λ (s)
    '()))
```

Definimos, neste contexto, um goal como uma função que recebe uma substituição e que, se retorna, retorna uma *stream* de substituições.

Como um exemplo, temos que `((≡ x y) empty-s)` ⇒ `‘(((,x . ,y)))`, uma lista com uma substituição (com uma associação).

Ao lidar com *Streams*, precisamos de funções especiais, já que não são “simples listas”. *Streams* são uteis (entre outras coisas) para a representação de estruturas de dados infinitas, então, por isso, funções e variáveis para lidar com elas terão um ∞ sobrescrito, para diferenciá-las das funções para listas comuns. Podemos, então, definir $append^{\infty}$:

```
(define (append∞ s∞ t∞)
  (cond
    ((null? s∞) t∞)
    ((pair? s∞)
      (cons (car s∞
        (append∞ (cdr s∞) t∞))))
    (else (λ ()
            (append∞ t∞ (s∞))))))
```

Note que, na suspensção, a ordem dos argumentos é trocada* Com essa função, podemos fazer

```
(define (disj₂ g₁ g₂)
  (λ (s)
    (append∞ (g₁ s) (g₂ s))))
```

em que $disj_2$ é uma disjunção (como um *ou* lógico).

Veja agora a seguinte definição:

```
(define (never°)
  (λ (s)
    (λ ()
      ((never°) s))))
```

Esse é um goal que não sucede nem falha. Para entender porque a ordem dos argumentos é trocada na suspenção de $append^{\infty}$, compare o valor de s^{∞} em

*No que é chamado de *trampolim binário* [3].

```
(let ((s∞ ((disj₂
              (neverº)
              (≡ 'olive x))
              empty-s)))
  s∞)
```

com o valor de s^∞ em

```
(let ((s∞ ((disj₂
              (≡ 'olive x))
              (neverº)
              empty-s)))
  s∞)
```

Em contraste com *never*^{o}, aqui está *always*^{o}, que sempre sucede:

```
(define (alwaysº)
  (λ (s)
    (λ ()
      ((disj₂ #s (alwaysº)) s))))
```

Antes de continuar, será útil conhecer a função *map*:
`(map f '(`el_1 ... el_n`))` $\Rightarrow$ `'((f` e_1`) ... (f` el_n`))`

A lista construída por *map* é constrúida por *cons*. Mas existe também *map-append*, análoga a *map*, mas em que a lista resultante é construída por *append*. Usaremos um *append-map*, mas para *streams*, isto é, um *append-map*$^\infty$:

```
(define (append-map∞ g s∞)
  (cond
    ((null? s∞) '())
    ((pair? s∞)
      (append∞ (g (car s∞))
        (append-map∞ g (cdr s∞))))
    (else (λ ()
            (append-map∞ g (s∞))))))
```

Assim como definimos a disjunção de dois goals, com isso podemos definir também a conjunção:

```
(define (conj₂ g₁ g₂)
  (λ (s)
    (append-map∞ g₂ (g₁ s))))
```

1.3.1 Voltando ao problema das variáveis

No *miniKanren*, assim como em linguagens de programação relacional no geral, as variáveis são lógicas. Mas, como a implementação está sendo feita em *Scheme*, precisamos, eventualmente, representar variáveis em termos de *Scheme*, num processo de reificação (lembre-se que reificação tem a ver com "concretização"). Em particular, um termo *miniKanren* reificado não pode conter variáveis lógicas. Fazemos isso associando variáveis lógicas a símbolos do tipo $\_{i}$. Para realizar essa operação, precisamos primeiro do *reify-name*[*]:

```
(define (reify-name n)
  (string→symbol
    (string-append ''_''
      (number→string n))))
```

Com *reify-name*, podemos criar o *reify-s*, que recebe uma variável e uma substituição, inicialmente vazia, r:

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

Vale notar, aqui *length* produz um número único em cada uso de *reify-name*.

Para continuar com nosso esquema de reificação, vamos precisar de uma versão ligeiramente diferente do *walk*, o qual chamaremos *walk**[†]. Veja a definição:

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

[*] *string → symbol* é escrito *string− > symbol*.
[†] Leia *walk star*.

```
        (else v))))
```

Note que *walk* e *walk** só diferem se *walk** *caminhar* a um par com alguma variável com associação na substituição s (algo como `(,z . (1 . ,x))`). Além disso, note que, se *walk** produz um valor v ao *caminhar* por uma substituição s, temos garantia de que as variáveis em v (se existirem) são frescas.

Com isso em mãos, podemos substituir cada variável fresca por sua reificação, com

```
(define (reify s)
  (λ (s)
    (let ((v (walk* v s)))
      (let ((r (reify-s v empty-s)))
        (walk* v r)))))
```

1.4 Finalizando

Na seção anterior, definimos a "coluna dorsal" do *miniKanren*. Para terminarmos[*] precisaremos usar as macros do *Scheme*. A palavra "macro", no geral, é usada (neste contexto de programação) para se referir a código que "escreve código", isto é, que realiza transformações no código a ser compilado ou interpretado. Várias linguagens têm macros de tipos diferentes, mas poucas são tão poderosas como as macros que (geralmente) estão presentes em linguagens Lisp[†]. *Scheme* não tem na própria linguagem mecanismos de iteração, por exemplo (como um laço *for*, ou *while*), mas esses (assim como vários outros mecanismos de iteração) podem ser facilmente implementados por meio de macros.

Para começar, notamos que temos a disjunção e a conjunção, mas para apenas dois argumentos, na forma de $disj_2$ e $conj_2$. Gostaríamos de realizar disjunções e conjunções com n goals, em que n pode ser diferente de 2. A disjunção de n termos é definida indutivamente (a conjunção é análoga):

$$(disj) \Rightarrow \#u$$
$$(disj\ g) \Rightarrow g$$
$$(disj\ g_0\ g\ ...) \Rightarrow (disj_2\ g_0\ (disj\ g\ ...))$$

que se traduz em código como

[*] Mais precisamente, terminarmos o *início*, já que *miniKanren* vai além do que vemos aqui.

[†] Provavelmente isso é devido ao aspecto homoicônico da linguagem, que torna transformações desse tipo mais simples de se realizar do ponto de vista do compilador ou interpretador em relação a outras linguagens.

```
(define-syntax disj
  (syntax-rules ()
    ((disj) #u)
    ((disj g) g)
    ((disj g0 g ...) (disj2 g0 (disj g ...)))))
```

Cada *defrel* vai definir uma nova função:

```
(define-syntax defrel
  (syntax-rules ()
    ((defrel (name x  ...) g ...)
     (define (name x ...)
       (λ (s)
         (λ ()
           ((conj g ...) s)))))))
```

Váriaveis frescas são criadas com *fresh*:

```
(define-syntax fresh
  (syntax-rules ()
    ((fresh () g ...)(conj g ...))
    ((fresh (x0 x ...) g ...)
      (call/fresh 'x0
        (λ (x0)
          (fresh (x ...) g ...))))))
```

Para executar o goal, definimos run^*, que recebe uma lista de variáveis e um goal e, se terminar de executar, assume como valor uma lista com os valores de associção a tais variáveis de modo que o goal tenha sucesso (vale lembrar, tal valor pode ser uma variável reificada). Definimos também *run*, que recebe um número natural n, uma lista de variáveis e um goal e, se terminar de executar, assume como valor os n primeiros elementos de run^*.

Para termos essas definições, usaremos $take^\infty$, que, quando dado um número n e uma stream s^∞, se algo, produz uma lista de, no máximo, n valores:

```
(define (take∞ n s∞)
  (cond
    ((and n (zero? n)) '())
    ((null? s∞) '())
    ((pair? s∞)
     (cons (car s∞)
       (take∞ (and n (sub1 n))
         (cdr s∞))))
    (else (take∞ n (s∞)))))
```

note que, se $n = false$, $take^{\infty}$, se retornar, produz uma lista de *todos* os valores (pergunta: valores de que?).

Agora podemos definir

```
(define (run-goal n g)
  (take∞ n (g empty-s)))
```

e

```
(define-syntax run
  (syntax-rules ()
    ((run n (x0 x ...) g ...)
     (run n q (fresh (x0 x ...)
                (≡ `(,x0 ,x ...) q) g ...)))
    ((run n q g ...)
     (let ((q (var 'q)))
       (map (reify q)
         (run-goal n (conj g ...)))))))

 (define-syntax run*
   (syntax-rules ()
     ((run* q g ...)(run #f q g ...))))
```

Com isso, temos um implementação mínima de *miniKanren*. Algumas construções importantes foram deixadas de lado em favor da brevidade, como os $cond^e$, $cond^a$ e $cond^u$, A leitora interessada é convidada a checar [2] ou [4] para mais detalhes.

Referências

[1] Ballantyne Michael, “A fast implementation of miniKanren with disequality and absento, compatible with Racket and Chez.” https://github.com/michaelballantyne/faster-miniKanren

[2] Daniel P. Friedman, William E. Byrd, Oleg Kiselyov, Jason Hemann, “The Reasoned Schemer - Second Edition”, The MIT Press, 2018.

[3] Ganz, Steven E., Daniel P. Friedman, and Mitchell Wand. “Trampolined style.”, In ACM SIGPLAN Notices, vol. 34, no. 9, pp. 18-27. ACM, 1999.

[4] Site do miniKanren http://minikanren.org/.

[5] R. Kent Dybvyg, “The Scheme Programming Language - Fourth Edition”, disponível em https://www.scheme.com/tspl4/, acesso em Setembro de 2018.

[6] Peter Norvig, “Paradigms of Artificial Inteligence Programming - Case Studies in Common Lisp”, Morgan Kauffman Publishers, 1992.