Going for a miniKanren implementation.

JONATHAN FRECH, Universität Tübingen, Germany

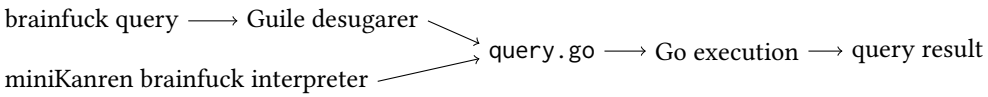
As a final project for the seminar "Relational Programming with miniKanren" held in the winter term of 2020/21, I wrote a miniKanren[2] interpreter in Go[5] and used it to explore the space of all brainfuck[4] programs. Jonathan Frech ⟨info@jfrech.com⟩, 2021-03-31

Additional Key Words and Phrases: relational programming, miniKanren, Go, concurrency, brainfuck

1 INTRODUCTION

In contrast to classical imperative thought and the uniform data flow associated with it, logic programming explores the space of all possible values, rejecting those that do not meet the desired properties. Amazingly, what at first glance might sound like a purely existencial axiomatically guaranteed comprehension is in certain settings computable on physically present hardware.

One particularly minimal specification of a logic programming language is miniKanren, as detailed in *The Reasoned Schemer*[2]. Following this book, I implemented a Go interpreter for the miniKanren language. Furthermore, I kneaded brainfuck programs through a brainfuck interpreter I wrote in miniKanren to solve various problems in $\mathbb{Z}_{2^8}$. Since the homoiconic beauty of Scheme is not present in Go, I wrote a GNU Guile desugarer to translate miniKanren programs represented as S-expressions to Go source code:



1.1 Installation

In the vein of embedding, the above mentioned desugarer produces a Go snippet; miniKanren programs are meant to be desugared, embedded and run as standalone Go programs:

```
% mkdir -p ~/src && cd ~/src
% curl -fsSL https://papers.jfrech.com/\
> 2021-03-31_jonathan-frech_going-for-a-minikanren-implementation/gomik.tar -o gomik.tar
% sha512sum gomik.tar | fold -w64 | sed 2q
ec40966528c616a8a2a47029ce98815cc973dc38b606a5479a5628af55956898
abedb08b50332ea4988eabaacb8e247a6ebed5690501d48b83568d3ffe4bf6b0
% tar -vxf gomik.tar && cd gomik
% ./test.sh
```

Instead of downloading the tar ball `gomik.tar` off the world-wide web and with access to this paper's digital form, one may also choose to *locally download* `gomik.tar` *through this very base64-encoded data URI* .

2 IMPLEMENTATION

My miniKanren implementation is a direct port of Friedman et al.'s Scheme implementation, with at times a more imperative approach. The major deviation is the choice of channels to implement goals and mutable maps to implement substitutions:

```
type Substitution = map[Variable]Term
type Stream = chan Substitution
type Goal = func(Substitution) Stream
```

Utilizing Go's higher-order capabilities, goals are combined as points:

```
var disj2 func(Goal, Goal) Goal
var conj2 func(Goal, Goal) Goal
```

This implementation neither supplies an embedding framework nor is meant to be executed as a standalone binary. By using a desugarer (`desugar.sh`, in turn running `desugarer.scm`), input miniKanren programs are transformed into Go snippets which are compiled together with the implementation to form an executable query. As a corollary, each query requires an invocation of the Go compiler. Source splicing is implemented in `run.sh`, which allows for a seemless invocation of miniKanren queries.

2.1 Symbolic purity

I chose not to support non-symbolic atoms found in many Scheme dialects (i. e. Booleans, integers, characters, strings, etc.) as I feel they distract from the profound and precisely meaningless beauty of S-expressions. Exacerbated by miniKanren's inability to reason about these other atom's structure – making them essentially equivalent in opaqueness to pure symbols – the only fundamental types I chose to implement are `Null`, `Symbol`, `Pair` and `Variable`.

2.2 Reification

Friedman et al. specify miniKanren's semantics only loosely, stating "[…] the order of the values does *not* matter."[2, p. 14] yet later reasoning about precisely how ordered query answers came to be. Due to the concurrent nature of channels and goroutines, I cannot guarantee for any order necessitating to sort the output for testing.

Reification semantics, however, are kept the same – with their implementation being imperative. All internal S-expressions are pretty-printed without shortening them.

2.3 A technicality regarding laziness

Querying the countably infinite space of all S-expressions requires special attention: Friedman et al. directly embed laziness into their stream data structure, defining it as a sum type of a finite list and a stream generating function [2, p. 153]. This way, postponing evaluation of the entire stream becomes possibly by piecewise trickling.

In my implementation, I opted to use Go's native concurrency model and thus chose to implement streams as channels of substitutions. However, since Go is as a C derivative fundamentally imperative, an unmodified goal combinator evaluates both of its arguments before analyzing its first, making implementing recursive relations impossible. To allow the declaration of recursive relations, special versions of `disj2` and `conj2` are crafted, called `disj2Lazy` and `conj2Lazy`: taking as their second argument a value of the type `func() Goal`, thus facilitating right-laziness.

A more fundamental problem is the intrinsically unidirectional nature of streams. Whereas laziness requires a communication between the data's user and its generator, no one listens when one yells that there is no need for a value at the receiving end of a channel. Thus, when spawning a goroutine to fill a channel, it will eagerly traverse the search space, looking for an answer even if there is no one who asks for such an answer. The net effect of this dissociation between data generation and usage is a buildup of unnecessarily working goroutines, wasting resources.

As a consequence, `conde`'s reordering property is even weaker than in Friedman et al.'s Scheme implementation: *a* `conde` *line may only possess one recursive relation which ought to appear as its last line.*

Unfortunately, the above culminates in an inefficient approach to laziness making my implementation inferior to *The Reasoned Schemer*'s Scheme implementation. See 3.4 where I detail how I ran the miniKanren queries presented in this paper.

2.4 Testing

I am no proponent of mindlessly and meticulously testing the smallest of implementation details; a reasonable encapsulation ethic paired with mindful program engineering leaves little room for local incorrect program behavior. A far more powerful approach is that of wholistic testing: when running the testing shell script `test.sh`, a set of fully-fledged miniKanren queries is invoked and compared to a statically reasoned about set of expected answers. Since this testing scenario is not trivialized by locally present assumptions, a high testing coverage is guaranteed, enstilling confidence in the implementation's correctness.

3 APPLICATION

One of the truly extraordinary possiblities logic programming offers is the ability to inquiry sub-implementations of other languages. Herein, I will use miniKanren to reason about brainfuck[4] – Urban Müller's minimalistic reimagination of a Turing machine.

3.1 Defining brainfuck

brainfuck programs consist of a finite string composed of eight instructions `+-<>[].,`, modify a writing head on a two-way infinitely long tape of bytes. Imperative in nature, a natural mapping from brainfuck source to C source exists, where `static char p[∞]; p += ∞/2`, i. e. `p` points somewhere in the middle of the infinitely long[1], initially zeroed tape:

```
+ ⟶ ++*p;
- ⟶ --*p;
> ⟶ ++p;
< ⟶ --p;
[ ⟶ while (*p) {
] ⟶ }
. ⟶ fputc(*p, stdout);
, ⟶ { int c = fgetc(stdin);
       *p = c == EOF ? 0 : c; }
```

As an example, `,[->+++<]>.` multiplies the inputted byte by three and outputs it.

[1]Note that `char[∞]` and `p += ∞/2` are an abuse of notation justified by the theoretic nature of the proposed C pseudo code; any real-world brainfuck implementation necessarily has to accept restrictions of finiteness.

3.2 Encoding brainfuck

To explore the space of all brainfuck programs, the space of all strings of eight characters $\coprod_{n\geq 0}\mathbb{Z}_8^n$ is too large, as a brainfuck program's brackets are always balanced. A more apt space to query is a subspace of the space of all S-expressions isomorphic to the space of all valid brainfuck programs in which `+-<>.,` are unique symbols {`INC`, `SUB`, `LFT`, `RGT`, `PUT`, `GET`} and any loop of the form `[`$\star$`]` is a pair `(LOP . `$\star$`)`.

A beneficial consequence of this encoding is the nonnecessity to store a program counter: a loop may or may not replicate itself based on the tape state, propagating itself until terminating. See 3.3 for further detail on the interpreter's implementation.

3.3 Interpreting brainfuck

I wrote my brainfuck interpreter in a very direct functional-to-relational translatory way. Since I am writing vanilla miniKanren without any extensions, e. g. support for finite domains, I enumerate the complete complement of `B00` in $\mathbb{Z}_{2^8}$ with a 255 cases long `disj` expression to facilitate disjoint branching encountered in brainfuck loops.

For `+` and `-`, a 256 elements large lookup table for `+` is hardcoded and `-` defined as its inverse.

The brainfuck program to be interpreted is represented using one list, with executed instructions getting popped off and loops potentially pushing new instructions to the front. Since the tape head may move in either direction, the tape is represented using two lists, each representing one side. Input and output are each represented using a single list.

See `miks/bf.mik` or equivalently 4.1 for the full implementation.

3.4 Running queries

Although I am confident in my implementation's correctness, albeit not its resourcefulness nor speed, querying benefits from an interactive execution approach; one for which my implementation is outclassed by a Scheme REPL. Thus, the following queries where performed using the implementation provided by the *The Reasoned Schemer*[3], evaluated using GNU Guile 2.2.6[1], called through `mik` which can be installed using the provided install script which fetches the miniKanren implementation from GitHub and installs `/usr/bin/mik` with auxillary files placed at `/usr/bin/.mik`:

```
# miks/install-the-the-reasoned-schemer-\
> second-edition-minikanren-implementation.sh
```

One may run the queries shown below using `./run.sh mik/bf.mik.q`, where one appends the query to the brainfuck interpreter implementation. However, as previously described in 2.3, some goroutines unfortunately may overflow the stack, rendering querying of large problems nearly impossible.

3.5 Querying brainfuck

3.5.1 brainfuck interpreter correctness. A basic query is forward-facing evaluation. The brainfuck program `++++[>+++<-]>.` computes $4 \cdot 3 = 12 =$ `0x0c` which is represented using the symbol `B0c`:

```
% mik miks/bf.mik
> (run* (o) (bf '() '(INC INC INC INC (LOP . (RGT INC INC INC LFT DEC)) RGT PUT) o))
$1 = (((B0c)))
```

Since an exhaustive `run*` has finished, this output is unique – as expected.

3.5.2 Six sixes. Unlike a classical interpreter, however, one is not restricted to only ask for the *result* of a computation and can query what inputs *may lead to* a given output. Since `bf` is a ternary relation with its center argument being brainfuck programs, one can generate brainfuck programs exhibiting a specified behavior by underspecifying the list of instructions, e. g. searching for six brainfuck programs which evaluate to 6:

```
% mik miks/bf.mik
> (run 6 (p) (bf '() p '(B06)))
$1 = (((INC INC INC INC INC INC PUT)) ((INC INC INC INC INC INC PUT INC))
((INC INC INC INC INC INC PUT DEC)) ((INC INC INC INC INC INC DEC PUT))
((INC INC INC INC INC INC DEC INC PUT)) ((INC INC INC INC INC DEC INC INC PUT)))
```

With an infinite number of brainfuck programs which evaluate to 6, `run*` is too ambitious a goal. Annotated with standard brainfuck notation, the above six programs are

```
(((INC INC INC INC INC INC PUT))          ; +++++.
 ((INC INC INC INC INC INC PUT INC))      ; +++++.+
 ((INC INC INC INC INC INC PUT DEC))      ; +++++.-
 ((INC INC INC INC INC INC INC DEC PUT))  ; +++++++-.
 ((INC INC INC INC INC INC DEC INC PUT))  ; ++++++-+.
 ((INC INC INC INC INC DEC INC INC PUT))) ; +++++-++.
```

and indeed all evaluate to 6.

3.5.3 Kneading subtraction. Querying for unrestricted programs is in most cases a vast task. More manageably, one can query for holes in a carefully crafted program specifying a problem. As an example, to calculate $8 - 5$, one may ask how to fill the brainfuck program `+++++++++`$\star$ to output 5. Of course, replacing $\star$ with `>+++++.` does not answer the original question, yet more immediate solutions do:

```
% mik miks/bf.mik
> (run 4 (p) (bf '() `(INC INC INC INC INC INC INC INC . ,p) '(B05)))
$1 = (((DEC DEC DEC PUT)) ((DEC DEC DEC PUT INC))
((RGT INC INC INC INC INC PUT)) ((DEC DEC DEC PUT DEC)))
```

From all but the third possible hole replacements one can count the leading `DEC` instructions and deduce the answer to the original question to be 3.

3.5.4 Solving multidimensional linear equations. Let $f : \mathbb{Z}_{2^8}^2 \to \mathbb{Z}_{2^8}, (x, y) \mapsto 2 \cdot x - 3 \cdot y$ and define $V_f := f^{-1}(\{0\})$ as its variety. To calculate V_f, one may implement f in brainfuck via `,[->++<],[->---<]>.` and query for all inputs which evaluate to zero:

```
% mik miks/bf.mik
> (run 5 (x y) (bf `(,x ,y) `(GET (LOP . (DEC RGT INC INC LFT))
... GET (LOP . (DEC RGT DEC DEC DEC LFT)) RGT PUT) '(B00)))
$1 = ((B00 B00) (B03 B02) (B06 B04) (B01 B56) (B09 B06))
```

Where the second to last solution $(\texttt{0x01}, \texttt{0x56}) = (1, 86)$ uncovers the modular nature of $\mathbb{Z}_{2^8}$, as

$$f(1, 86) = 2 \cdot 1 - 3 \cdot 86 = 2 - 258 = -256 \equiv 0.$$

3.5.5 Kneading division. In the same vein, one may also invert multiplication. That is, to calculate $^{12}/_{3}$ querying `,[>+++<-]>.`:

```
% mik miks/bf.mik
> (run* (i) (bf `(,i) '(GET (LOP . (RGT INC INC INC LFT DEC)) RGT PUT) '(B0c)))
$1 = ((B04))
```

Since it is a wholistic query, this *proves* that $3 \cdot x = 12$ has exactly one solution in $\mathbb{Z}_{2^8}$.
The same approach can be used to prove $\#\{x \in \mathbb{Z}_{2^8} : 4 \cdot x = 12\} = 4$:

```
% mik miks/bf.mik
> (run* (i) (bf `(,i) '(GET (LOP . (RGT INC INC INC INC LFT DEC)) RGT PUT) '(B0c)))
$1 = ((B03) (B43) (B83) (Bc3))
```

3.5.6 Boolean negation. With a starting fragment `+>,[`$\star_1$`]`$\star_2$ and the specification $\{0 \mapsto 1,\ 1 \mapsto 0\}$, miniKanren can fill both holes to satisfy the specification, implementing Boolean negation:

```
% mik miks/bf.mik
> (run 1 (p q1 q2) (== p `(INC RGT GET (LOP . ,q2) . ,q1))
... (bf '(B00) p '(B01)) (bf '(B01) p '(B00)))
$1 = (((INC RGT GET (LOP LFT) LFT PUT) (LFT PUT) (LFT)))
```

Replacing both holes $\star_1 :=$ `<.` and $\star_2 :=$ `<`, one arrives at `+>,[<]<.` which indeed implements a not gate.

Requesting four hole substitutions, further solutions `+>,[->]<.` as well as `+>,[+<]<.` and `+>,[>>]<.` are found which also implement not gates yet in a different manner.

```
% mik miks/bf.mik
> (run 4 (p q1 q2) (== p `(INC RGT GET (LOP . ,q2) . ,q1))
... (bf '(B00) p '(B01)) (bf '(B01) p '(B00)))
$1 = (((INC RGT GET (LOP LFT) LFT PUT) (LFT PUT) (LFT))
((INC RGT GET (LOP DEC RGT) LFT PUT) (LFT PUT) (DEC RGT))
((INC RGT GET (LOP INC LFT) LFT PUT) (LFT PUT) (INC LFT))
((INC RGT GET (LOP RGT RGT) LFT PUT) (LFT PUT) (RGT RGT)))
```

REFERENCES

[1] Free Software Foundation, Inc. *GNU Guile 2.2.6*. See https://www.gnu.org/software/guile/; accessed 2021-03-31. 1995–2019.

[2] Daniel P. Friedman et al. *The Reasoned Schemer*. 2nd ed. The MIT Press, 2018.

[3] Daniel P. Friedman et al. *trs2-impl.scm*. Hosted on GitHub: https://raw.githubusercontent.com/TheReasonedSchemer2ndEd/CodeFromTheReasonedSchemer2ndEd/master/trs2-impl.scm; accessed 2021-03-31. 2018.

[4] Urban Müller. *brainfuck*. See https://esolangs.org/wiki/Brainfuck; accessed 2021-03-31. 1993.

[5] The Go Project. *Go 1.16*. Accessed 2021-03-31. URL: https://golang.org/doc/go1.16.

Going for a miniKanren implementation.

4 LISTINGS

4.1 `miks/bf.mik`: a brainfuck interpreter in miniKanren

```
; Jonathan Frech, 2021-03-12, 2021-03-23, 2021-03-28, 2021-03-29 to 2021-03-31
; a brainfuck interpreter written in miniKanren

(defrel (bf old-in old-program new-out)
    (fresh (new-in new-tape-left new-tape-right)
        (bf-run-subnormal
            old-in '() '() old-program '()
            new-in new-tape-left new-tape-right '() new-out)))

(defrel (bf-run-subnormal
    old-in old-tape-left old-tape-right old-program old-out
    new-in new-tape-left new-tape-right new-program new-out)
    (fresh (normal-tape-left normal-tape-right)
        (normalize-tape old-tape-left normal-tape-left)
        (normalize-tape old-tape-right normal-tape-right)
        (bf-run
            old-in normal-tape-left normal-tape-right old-program old-out
            new-in new-tape-left new-tape-right new-program new-out)))

(defrel (bf-run
    old-in old-tape-left old-tape-right old-program old-out
    new-in new-tape-left new-tape-right new-program new-out)

    (fresh (in tape-left tape-right program out)
        (conde
            ((== old-program '())
                (== old-in new-in)
                (== old-tape-left new-tape-left)
                (== old-tape-right new-tape-right)
                (== '() new-program)
                (== old-out new-out))

            ((== old-program `(INC . ,program))
                (== old-in in)
                (tape-inc old-tape-left old-tape-right tape-left tape-right)
                (== old-out out)

                (bf-run-subnormal
                    in tape-left tape-right program out
                    new-in new-tape-left new-tape-right new-program new-out))

            ((== old-program `(DEC . ,program))
                (== old-in in)
                (tape-dec old-tape-left old-tape-right tape-left tape-right)
                (== old-out out)

                (bf-run-subnormal
                    in tape-left tape-right program out
                    new-in new-tape-left new-tape-right new-program new-out))

            ((== old-program `(RGT . ,program))
                (== old-in in)
                (tape-rgt old-tape-left old-tape-right tape-left tape-right)
                (== old-out out)

                (bf-run-subnormal
                    in tape-left tape-right program out
                    new-in new-tape-left new-tape-right new-program new-out))

            ((== old-program `(LFT . ,program))
                (== old-in in)
                (tape-lft old-tape-left old-tape-right tape-left tape-right)
                (== old-out out)

                (bf-run-subnormal
                    in tape-left tape-right program out
                    new-in new-tape-left new-tape-right new-program new-out))

            ((fresh (body further-program)
                (== old-in in)
                (== old-program `((LOP . ,body) . ,further-program))
                (tape-body body further-program
                    old-tape-left old-tape-right old-program
                    tape-left tape-right program)
                (== old-out out)

                (bf-run-subnormal
                    in tape-left tape-right program out
                    new-in new-tape-left new-tape-right new-program new-out)))

            ((== old-program `(PUT . ,program))
                (== old-in in)
                (fresh (t ts)
                    (== old-tape-left tape-left)
                    (== old-tape-right tape-right)
                    (== `(,t . ,ts) old-tape-right)
                    (== `(,t . ,old-out) out)

                    (bf-run-subnormal
                        in tape-left tape-right program out
                        new-in new-tape-left new-tape-right new-program new-out)))

            ((== old-program `(GET . ,program))
                (== old-out out)
                (fresh (i b bs)
                    (conde
                        ((== old-in `(,i . ,in)))
                        ((== old-in '())
                            (== 'B00 i)
                            (== '() in)))

                    (== old-tape-left tape-left)
                    (== old-tape-right `(,b . ,bs))
                    (== `(,i . ,bs) tape-right)

                    (bf-run-subnormal
                        in tape-left tape-right program out
                        new-in new-tape-left new-tape-right new-program new-out))))))

(defrel (inc b ++b)
    (conde
        ((== b 'B00) (== ++b 'B01)) ((== b 'B01) (== ++b 'B02))
        ((== b 'B02) (== ++b 'B03)) ((== b 'B03) (== ++b 'B04))
        ((== b 'B04) (== ++b 'B05)) ((== b 'B05) (== ++b 'B06))
        ((== b 'B06) (== ++b 'B07)) ((== b 'B07) (== ++b 'B08))
        ((== b 'B08) (== ++b 'B09)) ((== b 'B09) (== ++b 'B0a))
        ((== b 'B0a) (== ++b 'B0b)) ((== b 'B0b) (== ++b 'B0c))
        ((== b 'B0c) (== ++b 'B0d)) ((== b 'B0d) (== ++b 'B0e))
        ((== b 'B0e) (== ++b 'B0f)) ((== b 'B0f) (== ++b 'B10))
        ((== b 'B10) (== ++b 'B11)) ((== b 'B11) (== ++b 'B12))
        ((== b 'B12) (== ++b 'B13)) ((== b 'B13) (== ++b 'B14))
        ((== b 'B14) (== ++b 'B15)) ((== b 'B15) (== ++b 'B16))
        ((== b 'B16) (== ++b 'B17)) ((== b 'B17) (== ++b 'B18))
        ((== b 'B18) (== ++b 'B19)) ((== b 'B19) (== ++b 'B1a))
        ((== b 'B1a) (== ++b 'B1b)) ((== b 'B1b) (== ++b 'B1c))
        ((== b 'B1c) (== ++b 'B1d)) ((== b 'B1d) (== ++b 'B1e))
        ((== b 'B1e) (== ++b 'B1f)) ((== b 'B1f) (== ++b 'B20))
        ((== b 'B20) (== ++b 'B21)) ((== b 'B21) (== ++b 'B22))
        ((== b 'B22) (== ++b 'B23)) ((== b 'B23) (== ++b 'B24))
        ((== b 'B24) (== ++b 'B25)) ((== b 'B25) (== ++b 'B26))
        ((== b 'B26) (== ++b 'B27)) ((== b 'B27) (== ++b 'B28))
        ((== b 'B28) (== ++b 'B29)) ((== b 'B29) (== ++b 'B2a))
        ((== b 'B2a) (== ++b 'B2b)) ((== b 'B2b) (== ++b 'B2c))
        ((== b 'B2c) (== ++b 'B2d)) ((== b 'B2d) (== ++b 'B2e))
        ((== b 'B2e) (== ++b 'B2f)) ((== b 'B2f) (== ++b 'B30))
        ((== b 'B30) (== ++b 'B31)) ((== b 'B31) (== ++b 'B32))
        ((== b 'B32) (== ++b 'B33)) ((== b 'B33) (== ++b 'B34))
        ((== b 'B34) (== ++b 'B35)) ((== b 'B35) (== ++b 'B36))
        ((== b 'B36) (== ++b 'B37)) ((== b 'B37) (== ++b 'B38))
        ((== b 'B38) (== ++b 'B39)) ((== b 'B39) (== ++b 'B3a))
        ((== b 'B3a) (== ++b 'B3b)) ((== b 'B3b) (== ++b 'B3c))
        ((== b 'B3c) (== ++b 'B3d)) ((== b 'B3d) (== ++b 'B3e))
        ((== b 'B3e) (== ++b 'B3f)) ((== b 'B3f) (== ++b 'B40))
        ((== b 'B40) (== ++b 'B41)) ((== b 'B41) (== ++b 'B42))
        ((== b 'B42) (== ++b 'B43)) ((== b 'B43) (== ++b 'B44))
        ((== b 'B44) (== ++b 'B45)) ((== b 'B45) (== ++b 'B46))
        ((== b 'B46) (== ++b 'B47)) ((== b 'B47) (== ++b 'B48))
        ((== b 'B48) (== ++b 'B49)) ((== b 'B49) (== ++b 'B4a))
        ((== b 'B4a) (== ++b 'B4b)) ((== b 'B4b) (== ++b 'B4c))
        ((== b 'B4c) (== ++b 'B4d)) ((== b 'B4d) (== ++b 'B4e))
        ((== b 'B4e) (== ++b 'B4f)) ((== b 'B4f) (== ++b 'B50))
        ((== b 'B50) (== ++b 'B51)) ((== b 'B51) (== ++b 'B52))
        ((== b 'B52) (== ++b 'B53)) ((== b 'B53) (== ++b 'B54))
        ((== b 'B54) (== ++b 'B55)) ((== b 'B55) (== ++b 'B56))
        ((== b 'B56) (== ++b 'B57)) ((== b 'B57) (== ++b 'B58))
        ((== b 'B58) (== ++b 'B59)) ((== b 'B59) (== ++b 'B5a))
        ((== b 'B5a) (== ++b 'B5b)) ((== b 'B5b) (== ++b 'B5c))
        ((== b 'B5c) (== ++b 'B5d)) ((== b 'B5d) (== ++b 'B5e))
        ((== b 'B5e) (== ++b 'B5f)) ((== b 'B5f) (== ++b 'B60))
        ((== b 'B60) (== ++b 'B61)) ((== b 'B61) (== ++b 'B62))
        ((== b 'B62) (== ++b 'B63)) ((== b 'B63) (== ++b 'B64))
        ((== b 'B64) (== ++b 'B65)) ((== b 'B65) (== ++b 'B66))
        ((== b 'B66) (== ++b 'B67)) ((== b 'B67) (== ++b 'B68))
        ((== b 'B68) (== ++b 'B69)) ((== b 'B69) (== ++b 'B6a))
        ((== b 'B6a) (== ++b 'B6b)) ((== b 'B6b) (== ++b 'B6c))
        ((== b 'B6c) (== ++b 'B6d)) ((== b 'B6d) (== ++b 'B6e))
        ((== b 'B6e) (== ++b 'B6f)) ((== b 'B6f) (== ++b 'B70))
        ((== b 'B70) (== ++b 'B71)) ((== b 'B71) (== ++b 'B72))
        ((== b 'B72) (== ++b 'B73)) ((== b 'B73) (== ++b 'B74))
        ((== b 'B74) (== ++b 'B75)) ((== b 'B75) (== ++b 'B76))
        ((== b 'B76) (== ++b 'B77)) ((== b 'B77) (== ++b 'B78))
        ((== b 'B78) (== ++b 'B79)) ((== b 'B79) (== ++b 'B7a))
        ((== b 'B7a) (== ++b 'B7b)) ((== b 'B7b) (== ++b 'B7c))
        ((== b 'B7c) (== ++b 'B7d)) ((== b 'B7d) (== ++b 'B7e))
        ((== b 'B7e) (== ++b 'B7f)) ((== b 'B7f) (== ++b 'B80))
        ((== b 'B80) (== ++b 'B81)) ((== b 'B81) (== ++b 'B82))
        ((== b 'B82) (== ++b 'B83)) ((== b 'B83) (== ++b 'B84))
        ((== b 'B84) (== ++b 'B85)) ((== b 'B85) (== ++b 'B86))
        ((== b 'B86) (== ++b 'B87)) ((== b 'B87) (== ++b 'B88))
        ((== b 'B88) (== ++b 'B89)) ((== b 'B89) (== ++b 'B8a))
        ((== b 'B8a) (== ++b 'B8b)) ((== b 'B8b) (== ++b 'B8c))
        ((== b 'B8c) (== ++b 'B8d)) ((== b 'B8d) (== ++b 'B8e))
        ((== b 'B8e) (== ++b 'B8f)) ((== b 'B8f) (== ++b 'B90))
        ((== b 'B90) (== ++b 'B91)) ((== b 'B91) (== ++b 'B92))
        ((== b 'B92) (== ++b 'B93)) ((== b 'B93) (== ++b 'B94))
        ((== b 'B94) (== ++b 'B95)) ((== b 'B95) (== ++b 'B96))
        ((== b 'B96) (== ++b 'B97)) ((== b 'B97) (== ++b 'B98))
        ((== b 'B98) (== ++b 'B99)) ((== b 'B99) (== ++b 'B9a))
        ((== b 'B9a) (== ++b 'B9b)) ((== b 'B9b) (== ++b 'B9c))
        ((== b 'B9c) (== ++b 'B9d)) ((== b 'B9d) (== ++b 'B9e))
        ((== b 'B9e) (== ++b 'B9f)) ((== b 'B9f) (== ++b 'Ba0))
        ((== b 'Ba0) (== ++b 'Ba1)) ((== b 'Ba1) (== ++b 'Ba2))
        ((== b 'Ba2) (== ++b 'Ba3)) ((== b 'Ba3) (== ++b 'Ba4))
        ((== b 'Ba4) (== ++b 'Ba5)) ((== b 'Ba5) (== ++b 'Ba6))
        ((== b 'Ba6) (== ++b 'Ba7)) ((== b 'Ba7) (== ++b 'Ba8))
        ((== b 'Ba8) (== ++b 'Ba9)) ((== b 'Ba9) (== ++b 'Baa))
        ((== b 'Baa) (== ++b 'Bab)) ((== b 'Bab) (== ++b 'Bac))
        ((== b 'Bac) (== ++b 'Bad)) ((== b 'Bad) (== ++b 'Bae))
        ((== b 'Bae) (== ++b 'Baf)) ((== b 'Baf) (== ++b 'Bb0))
        ((== b 'Bb0) (== ++b 'Bb1)) ((== b 'Bb1) (== ++b 'Bb2))
        ((== b 'Bb2) (== ++b 'Bb3)) ((== b 'Bb3) (== ++b 'Bb4))
        ((== b 'Bb4) (== ++b 'Bb5)) ((== b 'Bb5) (== ++b 'Bb6))
        ((== b 'Bb6) (== ++b 'Bb7)) ((== b 'Bb7) (== ++b 'Bb8))
        ((== b 'Bb8) (== ++b 'Bb9)) ((== b 'Bb9) (== ++b 'Bba))
        ((== b 'Bba) (== ++b 'Bbb)) ((== b 'Bbb) (== ++b 'Bbc))
        ((== b 'Bbc) (== ++b 'Bbd)) ((== b 'Bbd) (== ++b 'Bbe))
        ((== b 'Bbe) (== ++b 'Bbf)) ((== b 'Bbf) (== ++b 'Bc0))
        ((== b 'Bc0) (== ++b 'Bc1)) ((== b 'Bc1) (== ++b 'Bc2))
        ((== b 'Bc2) (== ++b 'Bc3)) ((== b 'Bc3) (== ++b 'Bc4))
        ((== b 'Bc4) (== ++b 'Bc5)) ((== b 'Bc5) (== ++b 'Bc6))
        ((== b 'Bc6) (== ++b 'Bc7)) ((== b 'Bc7) (== ++b 'Bc8))
        ((== b 'Bc8) (== ++b 'Bc9)) ((== b 'Bc9) (== ++b 'Bca))
        ((== b 'Bca) (== ++b 'Bcb)) ((== b 'Bcb) (== ++b 'Bcc))
        ((== b 'Bcc) (== ++b 'Bcd)) ((== b 'Bcd) (== ++b 'Bce))
        ((== b 'Bce) (== ++b 'Bcf)) ((== b 'Bcf) (== ++b 'Bd0))
        ((== b 'Bd0) (== ++b 'Bd1)) ((== b 'Bd1) (== ++b 'Bd2))
        ((== b 'Bd2) (== ++b 'Bd3)) ((== b 'Bd3) (== ++b 'Bd4))
        ((== b 'Bd4) (== ++b 'Bd5)) ((== b 'Bd5) (== ++b 'Bd6))
        ((== b 'Bd6) (== ++b 'Bd7)) ((== b 'Bd7) (== ++b 'Bd8))
        ((== b 'Bd8) (== ++b 'Bd9)) ((== b 'Bd9) (== ++b 'Bda))
        ((== b 'Bda) (== ++b 'Bdb)) ((== b 'Bdb) (== ++b 'Bdc))
        ((== b 'Bdc) (== ++b 'Bdd)) ((== b 'Bdd) (== ++b 'Bde))
        ((== b 'Bde) (== ++b 'Bdf)) ((== b 'Bdf) (== ++b 'Be0))
        ((== b 'Be0) (== ++b 'Be1)) ((== b 'Be1) (== ++b 'Be2))
        ((== b 'Be2) (== ++b 'Be3)) ((== b 'Be3) (== ++b 'Be4))
        ((== b 'Be4) (== ++b 'Be5)) ((== b 'Be5) (== ++b 'Be6))
        ((== b 'Be6) (== ++b 'Be7)) ((== b 'Be7) (== ++b 'Be8))
        ((== b 'Be8) (== ++b 'Be9)) ((== b 'Be9) (== ++b 'Bea))
        ((== b 'Bea) (== ++b 'Beb)) ((== b 'Beb) (== ++b 'Bec))
        ((== b 'Bec) (== ++b 'Bed)) ((== b 'Bed) (== ++b 'Bee))
        ((== b 'Bee) (== ++b 'Bef)) ((== b 'Bef) (== ++b 'Bf0))
        ((== b 'Bf0) (== ++b 'Bf1)) ((== b 'Bf1) (== ++b 'Bf2))
        ((== b 'Bf2) (== ++b 'Bf3)) ((== b 'Bf3) (== ++b 'Bf4))
        ((== b 'Bf4) (== ++b 'Bf5)) ((== b 'Bf5) (== ++b 'Bf6))
        ((== b 'Bf6) (== ++b 'Bf7)) ((== b 'Bf7) (== ++b 'Bf8))
        ((== b 'Bf8) (== ++b 'Bf9)) ((== b 'Bf9) (== ++b 'Bfa))
        ((== b 'Bfa) (== ++b 'Bfb)) ((== b 'Bfb) (== ++b 'Bfc))
        ((== b 'Bfc) (== ++b 'Bfd)) ((== b 'Bfd) (== ++b 'Bfe))
        ((== b 'Bfe) (== ++b 'Bff)) ((== b 'Bff) (== ++b 'B00))))

(defrel (dec b --b)
    (inc --b b))

(defrel (nonzero b)
    (conde
        ((== b 'B01)) ((== b 'B02)) ((== b 'B03)) ((== b 'B04)) ((== b 'B05))
        ((== b 'B06)) ((== b 'B07)) ((== b 'B08)) ((== b 'B09)) ((== b 'B0a))
        ((== b 'B0b)) ((== b 'B0c)) ((== b 'B0d)) ((== b 'B0e)) ((== b 'B0f))
        ((== b 'B10)) ((== b 'B11)) ((== b 'B12)) ((== b 'B13)) ((== b 'B14))
        ((== b 'B15)) ((== b 'B16)) ((== b 'B17)) ((== b 'B18)) ((== b 'B19))
        ((== b 'B1a)) ((== b 'B1b)) ((== b 'B1c)) ((== b 'B1d)) ((== b 'B1e))
        ((== b 'B1f)) ((== b 'B20)) ((== b 'B21)) ((== b 'B22)) ((== b 'B23))
        ((== b 'B24)) ((== b 'B25)) ((== b 'B26)) ((== b 'B27)) ((== b 'B28))
        ((== b 'B29)) ((== b 'B2a)) ((== b 'B2b)) ((== b 'B2c)) ((== b 'B2d))
        ((== b 'B2e)) ((== b 'B2f)) ((== b 'B30)) ((== b 'B31)) ((== b 'B32))
        ((== b 'B33)) ((== b 'B34)) ((== b 'B35)) ((== b 'B36)) ((== b 'B37))
        ((== b 'B38)) ((== b 'B39)) ((== b 'B3a)) ((== b 'B3b)) ((== b 'B3c))
        ((== b 'B3d)) ((== b 'B3e)) ((== b 'B3f)) ((== b 'B40)) ((== b 'B41))
        ((== b 'B42)) ((== b 'B43)) ((== b 'B44)) ((== b 'B45)) ((== b 'B46))
        ((== b 'B47)) ((== b 'B48)) ((== b 'B49)) ((== b 'B4a)) ((== b 'B4b))
        ((== b 'B4c)) ((== b 'B4d)) ((== b 'B4e)) ((== b 'B4f)) ((== b 'B50))
        ((== b 'B51)) ((== b 'B52)) ((== b 'B53)) ((== b 'B54)) ((== b 'B55))
        ((== b 'B56)) ((== b 'B57)) ((== b 'B58)) ((== b 'B59)) ((== b 'B5a))
        ((== b 'B5b)) ((== b 'B5c)) ((== b 'B5d)) ((== b 'B5e)) ((== b 'B5f))
        ((== b 'B60)) ((== b 'B61)) ((== b 'B62)) ((== b 'B63)) ((== b 'B64))
        ((== b 'B65)) ((== b 'B66)) ((== b 'B67)) ((== b 'B68)) ((== b 'B69))
        ((== b 'B6a)) ((== b 'B6b)) ((== b 'B6c)) ((== b 'B6d)) ((== b 'B6e))
        ((== b 'B6f)) ((== b 'B70)) ((== b 'B71)) ((== b 'B72)) ((== b 'B73))
        ((== b 'B74)) ((== b 'B75)) ((== b 'B76)) ((== b 'B77)) ((== b 'B78))
        ((== b 'B79)) ((== b 'B7a)) ((== b 'B7b)) ((== b 'B7c)) ((== b 'B7d))
        ((== b 'B7e)) ((== b 'B7f)) ((== b 'B80)) ((== b 'B81)) ((== b 'B82))
        ((== b 'B83)) ((== b 'B84)) ((== b 'B85)) ((== b 'B86)) ((== b 'B87))
        ((== b 'B88)) ((== b 'B89)) ((== b 'B8a)) ((== b 'B8b)) ((== b 'B8c))
        ((== b 'B8d)) ((== b 'B8e)) ((== b 'B8f)) ((== b 'B90)) ((== b 'B91))
        ((== b 'B92)) ((== b 'B93)) ((== b 'B94)) ((== b 'B95)) ((== b 'B96))
        ((== b 'B97)) ((== b 'B98)) ((== b 'B99)) ((== b 'B9a)) ((== b 'B9b))
        ((== b 'B9c)) ((== b 'B9d)) ((== b 'B9e)) ((== b 'B9f)) ((== b 'Ba0))
        ((== b 'Ba1)) ((== b 'Ba2)) ((== b 'Ba3)) ((== b 'Ba4)) ((== b 'Ba5))
        ((== b 'Ba6)) ((== b 'Ba7)) ((== b 'Ba8)) ((== b 'Ba9)) ((== b 'Baa))
        ((== b 'Bab)) ((== b 'Bac)) ((== b 'Bad)) ((== b 'Bae)) ((== b 'Baf))
        ((== b 'Bb0)) ((== b 'Bb1)) ((== b 'Bb2)) ((== b 'Bb3)) ((== b 'Bb4))
        ((== b 'Bb5)) ((== b 'Bb6)) ((== b 'Bb7)) ((== b 'Bb8)) ((== b 'Bb9))
        ((== b 'Bba)) ((== b 'Bbb)) ((== b 'Bbc)) ((== b 'Bbd)) ((== b 'Bbe))
        ((== b 'Bbf)) ((== b 'Bc0)) ((== b 'Bc1)) ((== b 'Bc2)) ((== b 'Bc3))
        ((== b 'Bc4)) ((== b 'Bc5)) ((== b 'Bc6)) ((== b 'Bc7)) ((== b 'Bc8))
        ((== b 'Bc9)) ((== b 'Bca)) ((== b 'Bcb)) ((== b 'Bcc)) ((== b 'Bcd))
        ((== b 'Bce)) ((== b 'Bcf)) ((== b 'Bd0)) ((== b 'Bd1)) ((== b 'Bd2))
        ((== b 'Bd3)) ((== b 'Bd4)) ((== b 'Bd5)) ((== b 'Bd6)) ((== b 'Bd7))
        ((== b 'Bd8)) ((== b 'Bd9)) ((== b 'Bda)) ((== b 'Bdb)) ((== b 'Bdc))
        ((== b 'Bdd)) ((== b 'Bde)) ((== b 'Bdf)) ((== b 'Be0)) ((== b 'Be1))
        ((== b 'Be2)) ((== b 'Be3)) ((== b 'Be4)) ((== b 'Be5)) ((== b 'Be6))
        ((== b 'Be7)) ((== b 'Be8)) ((== b 'Be9)) ((== b 'Bea)) ((== b 'Beb))
        ((== b 'Bec)) ((== b 'Bed)) ((== b 'Bee)) ((== b 'Bef)) ((== b 'Bf0))
        ((== b 'Bf1)) ((== b 'Bf2)) ((== b 'Bf3)) ((== b 'Bf4)) ((== b 'Bf5))
        ((== b 'Bf6)) ((== b 'Bf7)) ((== b 'Bf8)) ((== b 'Bf9)) ((== b 'Bfa))
        ((== b 'Bfb)) ((== b 'Bfc)) ((== b 'Bfd)) ((== b 'Bfe)) ((== b 'Bff))))

(defrel (normalize-tape tape normal-tape)
    (fresh (t ts)
        (conde
            ((== tape '())
                (== '(B00) normal-tape))
            ((== tape `(,t . ,ts))
                (== tape normal-tape)))))
```

```
(defrel (tape-inc old-tape-left old-tape-right new-tape-left new-tape-right)
    (fresh (b bs)
        (conde
            ((== old-tape-right '())
                (== new-tape-left old-tape-left)
                (== new-tape-right '(B01)))
            ((== old-tape-right `(,b . ,bs))
                (fresh (++b)
                    (== new-tape-left old-tape-left)
                    (inc b ++b)
                    (== new-tape-right `(,++b . ,bs)))))))
(defrel (tape-dec old-tape-left old-tape-right new-tape-left new-tape-right)
    (fresh (b bs)
        (conde
            ((== old-tape-right '())
                (== new-tape-left old-tape-left)
                (== new-tape-right '(Bff)))
            ((== old-tape-right `(,b . ,bs))
                (fresh (++b)
                    (== new-tape-left old-tape-left)
                    (dec b ++b)
                    (== new-tape-right `(,++b . ,bs)))))))
(defrel (tape-rgt old-tape-left old-tape-right new-tape-left new-tape-right)
    (fresh (b bs)
        (conde
            ((== old-tape-right '())
                (== old-tape-left new-tape-left)
                (== '(B00) new-tape-right))
            ((== old-tape-right `(,b . ,bs))
                (== `(,b . ,old-tape-left) new-tape-left)
                (== bs new-tape-right)))))
(defrel (tape-lft old-tape-left old-tape-right new-tape-left new-tape-right)
    (fresh (b bs)
        (conde
            ((== old-tape-left '())
                (== '() new-tape-left)
                (== `(B00 . ,old-tape-right) new-tape-right))
            ((== old-tape-left `(,b . ,bs))
                (== bs new-tape-left)
                (== `(,b . ,old-tape-right) new-tape-right)))))
(defrel (tape-body body further-program
    old-tape-left old-tape-right old-program
    tape-left tape-right program)
    (fresh (t ts)
        (== old-tape-right `(,t . ,ts))
        (== old-tape-left tape-left)
        (== old-tape-right tape-right)
        (conde
            ((== t 'B00)
                (== further-program program))
            ((nonzero t)
                (appendo body old-program program)))))
(defrel (appendo l t out)
    (conde
        ((== '() l) (== t out))
        ((fresh (a d res)
            (== `(,a . ,d) l)
            (== `(,a . ,res) out)
            (appendo d t res)))))
```