microkanrenpy Documentation

Release 1.0

Massimo Nocentini

May 02, 2017

Contents

CHAPTER 1

Introduction

Welcome to documentation of *ηkanrenpy*, a *pythonic* implementation of a *relational interpreter* and an effort to port the original code written in Scheme by Jason Hemann and Daniel P. Friedman. We report their words to describe what this work is about:

> This paper presents μKanren, a minimalist language in the miniKanren family of relational (logic) programming languages. Its implementation comprises fewer than 40 lines of Scheme. We motivate the need for a minimalist miniKanren language, and iteratively develop a complete search strategy. Finally, we demonstrate that through sufficient user-level features one regains much of the expressiveness of other miniKanren languages. In our opinion its brevity and simple semantics make μKanren uniquely elegant.

CHAPTER 2

Contents

We organize our thoughts according to the following outline:

Intro

I love *Lisp*, *Python* too. The work I'm going to describe has born *for fun*, for my education, to bang my head against a taugh, beautiful, not so simple to grasp yet elegant piece of software which is *ηkanren [HF13]*, in the *miniKanren [RS05]* family of logic languages.

To be honest, this documentations has many targets:

- it (should) describes what I've done
- it is used to get credits for my PhD course
- to clear my thoughts about the system I'm keeping alive
- to understand and play with documentation tools
- to write down abstract machines that I believe *charming* and *elegant*
- to keep one foot in Lisp and the other in Python, however keep coding

I would like to structure my exposition following advices from an interesting group of people much more experienced that me on writing this kind of stuff; consequently, in what follows you will find the same sections as you find in the referenced page, hoping to provide answers and to taylor paragraphs to this particular project. Quoting their Sidebar on open source:

> There is a magical feeling that happens when you release your code. It comes in a variety of ways, but it always hits you the same. Someone is using my code?! A mix of terror and excitement.
>
> > I made something of value! What if it breaks?! I am a real open source developer! Oh god, someone else is using my code...
>
> Writing good documentation will help alleviate some of these fears. A lot of this fear comes from putting something into the world. My favorite quote about this is something along these lines:

> Fear is what happens when you're doing something important. If you are doing work that isn't scary, it isn't improving you or the world.
>
> Congrats on being afraid! It means you're doing something important.

from my little and humble position, I'm proud to have been afraid as I was doing all this stuff...

Why write docs

You will be using your code in 6 months (aka, *the future of this project*)

The reality:

> I find it best to start off with a selfish appeal. The best reason to write docs is because you will be using your code in 6 months. Code that you wrote 6 months ago is often indistinguishable from code that someone else has written. You will look upon a file with a fond sense of remembrance. Then a sneaking feeling of foreboding, knowing that someone less experienced, less wise, had written it.

I started to document what I've done because I would like to keep this code alive, after the core has been proven mature enough, I would like to add *constraints [HF15]*, *guided search [CF15]* and *uncertainty [GS17]*.

You want people to use your code

If people don't know why your project exists, they won't use it. This project exists mainly *for fun*, to code something beautiful and understand core and elegant ideas underlying *logic programming*. In parallel, instead of providing a clone implementation again in Scheme as the original authors do, we prefere to target the *Python* language which allows us to characterize our version with respect to:

- we stick to the original language as possible, so we use the same names for functions and objects; moreover, we adopt the same functional style, without messing up with different paradigms
- we use native *generator* objects provided by Python to handle relations that can be satisfied by both a finite or a *countably* infinite set of substitutions for *logic variables*
- we rewrite the fundamental function that explores the space of substitutions that satisfy a relation as an *iterative* process instead of *recursive*, in this way we lose the elegance of recursion (inspite of limit on recursive calls imposed by the language), gaining a *fair* exploration of the solutions space even if it infinite respect to both the number of streams and the content in each one of them (for more details on the topic, have a look at docstring of function `muk.core.mplus()`)
- try to write concise, powerful and idiomatic python code for a non-trivial problem, we would like to keep functions and definitions as small and simple as possible; remember the mantra

```
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
```

```
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

If people can't figure out how to install your code, they won't use it. All that is required is to have a Git client available, so type the following in a console:

```
git clone https://github.com/massimo-nocentini/microkanrenpy.git # get stuff
cd on-python # go into there
git checkout microkanren # switch to the right branch
cd microkanren # go into the correct dir
python3 # start the Python interpreter
```

Now in the python interpreter it is possible to load our core module:

```
>>> from muk.core import *
```

that's it!

If people can't figure out how to use your code, they won't use it. We provide a simple tutorial in a dedicated page *A Primer*.

You want people to help out

You only get contributions after you have put in a lot of work. We work our ηkanren with:

- the little book *[RS05]* or https://mitpress.mit.edu/books/reasoned-schemer, reproducing **all relations** and answering **all questions**: definitions are literally included in the page *The Reasoned Schemer*, while we record with in a unit test file about **350 asserts** to cover questions in the book;
- the puzzle **The Mistery of the Monte Carlo lock** from the book *[RS82]* by Raymond Smullyan, where we implement a generic machine based on *inference rules*, coding and plugging in those rule necessary to build McCulloch's machines to find **he key of the lock**, finally. All defs and some comments are recorded in *Monte Carlo Lock puzzle* and in the tests suite there are about **60 asserts**.

You only get contributions after you have users (aka, to whet your appetite). Although I've played with ηkanren mostly by writing down definitions in the context of sexp, namely `cons` cells manipulations, I believe of interest the definition of a generic machine to be configured with *arbitrary inference rules*: this allows us to implement a relational interpreter for some *process algebras*, one of them is Milner's CCS calculus.

You only get contributions after you have documentation. I'm building it! Moreover, this is an *executable documentation*: every snippet of Python code appearing in these pages, in particular in *A Primer*, is required to pass in the sense of doctests, in other words a line that fails to produce the expected output does preevent the generation of the *whole* documentation. This is possible by running Sphinx with the corresponding doctest extension; moreover, this extension runs and checks doctests written in docstrings of **any** referenced Python definition stored in source files.

It makes your code better

I would like to enhance each definition in the `muk.core` module with a solid and consistent docstring, mainly to clarify my thoughts as done for function `muk.core.mplus()` in my humble opinion, where we try to explain

different enumeration strategies to achieve *interleaving* of satisfying substitutions.

Moreover, looking for some chunk of code that needs better doc allows us to find new ideas and possible refactorings and/or enhancements, the following is a list of ideas:

- introduce a `goal` class, in order to implement magic methods `__and__`, `__or__` to provide syntactic sugar for goal composition;
- override method `__radd__` in class `var` in order to write something like `[1,2,3]+a`, where `a` is a logic variable; this should construct an object that represent an *extension list*, currently implemented using `cons` objects;
- override method `__add__` in class `var` in order to write something like `a+[1,2,3]`, where `a` is a logic variable; this should construct an object that represent an *append list* , currently not;
- from the previous two points would be possible to write `([1,2,3]+a)-a`, namely a *difference list* as provided by standard Prolog.

You want to be a better writer (aka, *looking at the past*)

I use this section to summarize references and existing works.

First of all, thank you William E. Byrd for your hard work. Will wrote his PhD thesis *[WB09]* and shared a series of hangout video lectures known as miniKanran uncourse; moreover, he is a coauthor of *[RS05]* and maintains a tremendous archive at http://minikanren.org/, where it is possible to find references to scientific publications, existing implementation in lots of different programming languages and, finally, links to talks presented in many confs, all of this stuff related to the field of *relational programming*.

Python implementations

According to Will, we just review the following existing projects:

pykanren which implements the very basic definitions of the core system of both miniKanren and ηkanren: variables, unification only for *list* objects, reification and a complete enumeration strategy in the style of *[RS05]* without *interleaving* of satisfying substitutions. Tests are present to catch the essentials, no application to particular problems.

pythological which implements a *domain specific language* in the style of Prolog clauses on top of Python. The core interpreter uses primitive *generators* as we do, implements all the basic definitions of the logic system with *interleaving* and *occur-check* to detect circular substitutions. In my opinion it is an interesting project due to the parsing of the dsl which we do not provide; for what concerns testing, there are no test suites but simple and nice examples written in the defined language.

logpy which implements the core system, supports relations satisfiable by infinite substitutions using primitive Python generator objects and the style is close to the canonical version, although not closer; in in our humble opinion, the code seems a bit complex to read and understand because of the presence of many auxiliary functions to handle streams of satisfying substitutions and to combine goals. Moreover, this project aims to extend unification over objects of arbitrary types using two dependencies to perform *double dispatching* and *structural unification*, which we aim to reach just by sending messages in a more pure and fundamental way keeping the *Smalltalk way* as tenet.

microkanren which implements the very very basic implementation in about 100 lines of code: due to its brevity, it supplies variables, primitive goals, disjs and conjs as combinators, without interleaving and tests. It is released as a GitHub gist.

In spite of these existing projects, our main principle is to remain as close as possible to ideas presented in *[HF13]*: we use the same names for functions as they use in the paper and we write definitions in the same *purely functional style*. This allows us to preserve compact and concise defs, most of them do not exceed 5 lines of code. A drawback

is some boilerplate and redundancy, which can be seen immediately in the unit tests, due to the presence of many `lambda` expressions and to the repeated use of `run`, `fresh` and `conj`. Finally, our main contribution comprises two test suites, the first reproduces all examples (including those about the *discrete logarithm*) of *[RS05]*, the second solves a logic puzzle by building an abstract machine composing inference rules.

Lisp implementations

There are many implementation in the Lisp family:

- for vanilla Scheme there are the canonical one, with symbolic constraints, with probabilistic inference and with guided search; finally, I'm particular interested in a recent impl which uses the last r7rs Scheme standard.
- for Clojure there is the outstanding core.logic maintained by David Nolen; otherwise, two plain ports for Racket and for CommonLisp, respectively.

Other languages and links

I would like to list some resources that I've discovered in the meanwhile:

ηkanren in Haskell a blog post by *Michael J. Sullivan*, also a recorded talk

ηkanren in Ruby a Ruby implementation by *Tom Stuart*

Hakaru an embedded probabilistic programming language in Haskell

Logic Programming 15-819K Logic Programming course, Fall 2006, taught by prof. *Frank Pfenning*

Prolog course Prolog course, 2008–09, Principal lecturer: Dr *David Eyers*

What technology

Information for people who want to contribute back

I think that GitHub is a very strong platform where we can keep alive this project. Moreover, I think also that the decentralized model and the workflow proposed by GitHub itself, which is based on pull requests, is a clean and healthy methodology to do development. Therefore, I would like to accept contributions according to this settings, using facilities provided by the platform to do discussions and to track issues. Finally, here is the address of the repository:

> https://github.com/massimo-nocentini/microkanrenpy

Together with simplicity and elegance, we believe that *automated testing* is a vital principle to keep the code base healthy. Therefore, we try to capture almost every idea with unit tests as much as we can and documentation is no exception, namely every code snippet in the documentation and every doctest in docstrings should pass against their asserts in order to produce this documentation itself.

README.rd first

As raccomanded by this article, have a look to our README.md first.

How to get support

If you are having issues, please let us know and feel free to drop me an email at massimo.nocentini@unifi.it for any info you would like to known.

Your project's license

A Primer

This short document aims to show the basic concepts underlying *μkanren*, namely relations as goals, and how to use primitive goals as building blocks to build and define relations. We start from two primitive relations toward recursive relations, through constructors, fresh logic variables, unification and η-inversion.

First of all we start importing objects and definitions of the logic system:

```
>>> from muk.core import *
>>> from muk.ext import *
```

We use the interface function `run` to ask for substitutions that satisfy the relation under study; moreover, nouns *relation*, *goal* and *predicate* are equivalent notions and denotes the same concept in the rest of this presentation.

Primitive goals

The two primitive goals `succeed` and `fail` denote *truth* and *falsehood* in classic logic, respectively.

succeed

Asking for all substitutions that satisfy the `succeed` relation yields a tautology, namely any substitution works:

```
>>> run(succeed)
[Tautology]
```

fail

On the other hand, there is no substitution that satisfies the `fail` relation:

```
>>> run(fail)
[]
```

Goal ctors

Before introducing goal constructors we show how a logic variable is represented when it doesn't get any association:

```
>>> rvar(0)
0
```

where class `rvar` builds a *reified variable* providing an integer, namely `0`, to track the variable's birthdate.

fresh

It is a goal ctor that introduces *new* logic variables, namely variables without associations, according to the plugged-in lambda expression, which receives vars and returns a goal. Consider the following query where the variable `q` is introduced fresh:

```
>>> run(fresh(lambda q: succeed))
[_0]
```

since the returned goal is `succeed` and var `q` doesn't get any association, it remains fresh in the list of values, result of the whole `run` invocation, that satisfy the goal `fresh(lambda q: succeed)`.

As particular case, it is possible to plugin a *thunk*, namely a lambda expression without argument:

```
>>> run(fresh(lambda: succeed))
[Tautology]
```

at a first this could look useless but it is of great help for the definition of *recursive relations* as we will see in later examples (it is an instance of η-inversion, formally).

unify

It is a goal ctor that attempts to make two arbitrary objects equal, recording associations when fresh variables appears in the nested structures under unification. Here we show two simple examples of unification, the first succeeds while the second doesn't:

```
>>> run(unify(3, 3))
[Tautology]
>>> run(unify([1, 2, 3], [[1]]))
[]
```

On the other hand, things get interesting when fresh variables are mixed in:

```
>>> run(fresh(lambda q: unify(3, q)))
[3]
>>> run(fresh(lambda q: unify([1, 2, 3], [1] + q)))
[[2, 3]]
>>> run(fresh(lambda q: unify([[2, 3], 1, 2, 3], [q, 1] + q)))
[[2, 3]]
```

When two fresh vars are unified it is said that they *share* or *co-refer*:

```
>>> run(fresh(lambda q, z: unify(q, z)))
[_0]
>>> run(fresh(lambda q, z: conj(unify(q, z), unify(z, 3))),
...     var_selector=lambda q, z: q)
[3]
```

disj

It is a goal ctor that consumes two goals and returns a new goal that can be satisfied when *either* the former *or* the latter goal can be satisfied:

```
>>> run(disj(succeed, fail))
[Tautology]
>>> run(disj(fail, fresh(lambda q: unify(q, True))))
[Tautology]
>>> run(fresh(lambda q: disj(fail, fail)))
[]
>>> run(fresh(lambda q: disj(unify(q, False), unify(q, True))))
[False, True]
```

conj

It is a goal ctor that consumes two goals and returns a new goal that can be satisfied when *both* the former *and* the latter goal can be satisfied:

```
>>> run(conj(succeed, fail))
[]
>>> run(conj(fail, fresh(lambda q: unify(q, True))))
[]
>>> run(fresh(lambda q: conj(unify(q, 3), succeed)))
[3]
>>> run(fresh(lambda q: conj(unify(q, False), unify(q, True))))
[]
>>> run(fresh(lambda q: conj(fresh(lambda q: unify(q, False)),
...                          unify(q, True))))
[True]
```

Facts and recursive relations

In order to represent *facts* we introduce the `conde` goal ctor, which is defined as a combination of conjs and disjs and we show how to write recursive relation, possibly satisfied by a countably infinite number of values.

conde

The following simple example resembles facts declaration in Prolog:

```
>>> run(fresh(lambda q: conde([unify(q, 'orange'), succeed],
...                           [unify(q, 'lemon'), fail],
...                           [unify(q, 'pear'), succeed],
...                           [unify(q, 'apple'), succeed])))
['orange', 'pear', 'apple']
```

η-inversion

Let us define a relation that yields countably many 5 objects; in order to do that, the usual solution is to write a recursive definition. However, we proceed step by step, adjusting and learning from the Python semantic of argument evaluation at *function-call time*. Consider the following as initial definition:

```
>>> def fives(x):
...     return disj(unify(5, x), fives(x))
...
>>> run(fresh(lambda x: fives(x)))
Traceback (most recent call last):
...
RecursionError: maximum recursion depth exceeded while calling a Python object
```

Exception `RecursionError` is raised because in the body of function `fives` it is required to evaluate `fives(x)` in order to return a `disj` object, but this is the point from were we started, hence no progress for recursion.

Keeping in mind the previous argument, why not wrapping the recursion on `fives(x)` inside a `fresh` ctor in order to refresh the var `x` at each invocation?

```
>>> def fives(x):
...     return disj(unify(5, x), fresh(lambda x: fives(x)))
...
>>> run(fresh(lambda x: fives(x)))
Traceback (most recent call last):
...
RecursionError: maximum recursion depth exceeded while calling a Python object
```

Again the same exception as before, this time for a different reason, however: since we ask for all associations that satisfy the *countably infinite* relation `fives`, function `run` continue to look for such values which are infinite, of course. So, select only the first 10 objects:

```
>>> def fives(x):
...     return disj(unify(5, x), fresh(lambda x: fives(x)))
...
>>> run(fresh(lambda x: fives(x)), n=10)
[5, _0, _0, _0, _0, _0, _0, _0, _0, _0]
```

Although not a list of 10 fives objects, it makes sense: the very first 5 gets associated to the var `x` introduced by the goal provided to `run` by the `fresh` ctor, and this association is only one way to satisfy the `disj` in the definition of relation `fives`. Looking for other associations that work, we attempt to satisfy the second goal in the `disj`, namely `fresh(lambda x: fives(x))`: it introduces a new var `x`, different from the previous one, and then recurs, leaving the original var without association. Since associations shown in the output list refer to the very first var `x`, we get many `_0` symbols which represent the absence of association, therefore `x` remains fresh.

fives

One way to actually get a list of fives is to unify inside the inner `fresh`, as follows:

```
>>> def fives(x):
...     return disj(unify(5, x),
...                 fresh(lambda y: conj(fives(y), unify(y, x))))
...
>>> run(fresh(lambda x: fives(x)), n=10)
[5, 5, 5, 5, 5, 5, 5, 5, 5, 5]
```

or to use `fresh` as *η-inversion* rule, as follows:

```
>>> def fives(x):
...     return disj(unify(5, x), fresh(lambda: fives(x)))
...
```

```
>>> run(fresh(lambda x: fives(x)), n=10)
[5, 5, 5, 5, 5, 5, 5, 5, 5, 5]
```

nats

Just for fun, using the previous trick and abstracting out the 5s, we can generate the naturals, taking only the first 10 as follows:

```
>>> def nats(x, n=0):
...     return disj(unify(n, x), fresh(lambda: nats(x, n+1)))
...
>>> run(fresh(lambda x: nats(x)), n=10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

append

Here we want to define the relation `append(r, s, o)` which holds if `r + s == o` where `r, s, o` are both lists. First of all we need an helper relation `nullo` which holds if `l == []`:

```
>>> def nullo(l):
...     return unify([], l)
```

so it follows the recursive definition, as usual:

```
>>> def append(r, s, out):
...     def A(r, out):
...         return conde([nullo(r), unify(s, out)],
...                      else_clause=[fresh(lambda a, d, res:
...                                         conj(unify([a]+d, r),
...                                              unify([a]+res, out),
...                                              fresh(lambda: A(d, res)),))])
...     return A(r, out)
```

Some examples follow:

```
>>> run(fresh(lambda q: append([1,2,3], [4,5,6], q)))
[[1, 2, 3, 4, 5, 6]]
>>> run(fresh(lambda l, q: append([1,2,3]+q, [4,5,6], l)), n=4)
[[1, 2, 3, 4, 5, 6],
 [1, 2, 3, _0, 4, 5, 6],
 [1, 2, 3, _0, _1, 4, 5, 6],
 [1, 2, 3, _0, _1, _2, 4, 5, 6]]
>>> run(fresh(lambda r, x, y:
...             conj(append(x, y, ['cake', 'with', 'ice', 'd', 't']),
...                  unify([x, y], r))))
[[[], ['cake', 'with', 'ice', 'd', 't']],
 [['cake'], ['with', 'ice', 'd', 't']],
 [['cake', 'with'], ['ice', 'd', 't']],
 [['cake', 'with', 'ice'], ['d', 't']],
 [['cake', 'with', 'ice', 'd'], ['t']],
 [['cake', 'with', 'ice', 'd', 't'], []]]
```

The Reasoned Schemer

```
from muk.sexp import *
from muk.core import *
from muk.ext import *

@adapt_iterables_to_conses(all_arguments)
def nullo(l):
    return unify([], l)

@adapt_iterables_to_conses(lambda a, d, p: {d, p})
def conso(a, d, p):
    return unify(cons(a, d), p)

@adapt_iterables_to_conses(all_arguments)
def pairo(p):
    return fresh(lambda a, d: conso(a, d, p))

@adapt_iterables_to_conses(lambda p, a: {p})
def caro(p, a):
    return fresh(lambda d: conso(a, d, p))

@adapt_iterables_to_conses(all_arguments)
def cdro(p, d):
    return fresh(lambda a: conso(a, d, p))

@adapt_iterables_to_conses(all_arguments)
def listo(l):
    return conde([nullo(l), succeed],
                 #[pairo(l), fresh(lambda d: conj(cdro(l, d), listo(d)))])
                 [pairo(l), fresh(lambda a, d: conj(unify([a] + d, l), listo(d)))])

@adapt_iterables_to_conses(all_arguments)
def lolo(l):
    return conde([nullo(l), succeed],
                 #[fresh(lambda a: conj(caro(l, a), listo(a))), fresh(lambda d:␣
↪conj(cdro(l, d), lolo(d)))])
                 [fresh(lambda a, d: conj(unify([a] + d, l), listo(a), lolo(d))),␣
↪succeed])

@adapt_iterables_to_conses(all_arguments)
def twinso(s):
    #return fresh(lambda a, d: conj(unify([a] + d, s), unify([a], d))) # → unify([a␣
↪a], s)
    return fresh(lambda x: unify([x, x], s))

@adapt_iterables_to_conses(all_arguments)
def loto(l):
    return conde([nullo(l), succeed],
                 #[fresh(lambda a: conj(caro(l, a), twinso(a))), fresh(lambda d:␣
↪conj(cdro(l, d), loto(d)))])
                 [fresh(lambda a, d: conj(unify([a] + d, l), twinso(a), loto(d))),␣
↪succeed])

@adapt_iterables_to_conses(lambda predo, l: {l})
def listofo(predo, l):
    return conde([nullo(l), succeed],
```

```
                [fresh(lambda a, d: conj(unify([a] + d, l), predo(a), listofo(predo,␣
↪d))), succeed])

@adapt_iterables_to_conses(lambda x, l: {l})
def membero(x, l):
    return conde([nullo(l), fail],
                 [caro(l, x), succeed],
                 [fresh(lambda d: conj(cdro(l, d), membero(x, d))), succeed])

@adapt_iterables_to_conses(lambda x, l: {l})
def pmembero(x, l):
    return conde([nullo(l), fail],
                 [unify([x], l), succeed], # [caro(l, x), cdro(l, [])], in other words
                 [fresh(lambda d: conj(unify([x] + d, l), pairo(d))), succeed], #␣
↪[caro(l, x), fresh(lambda a, d: cdro(l, (a, d)))], in other words
                 [fresh(lambda d: conj(cdro(l, d), pmembero(x, d))), succeed])

@adapt_iterables_to_conses(lambda x, l: {l})
def pmemberso(x, l):
    return conde([nullo(l), fail],
                 [fresh(lambda d: conj(unify([x] + d, l), pairo(d))), succeed],
                 [unify([x], l), succeed],
                 [fresh(lambda d: conj(cdro(l, d), pmemberso(x, d))), succeed])

def first_value(l):
    return run(fresh(lambda y: membero(y, l)), n=1)

@adapt_iterables_to_conses(lambda x, l: {l})
def memberrevo(x, l):
    return conde([nullo(l), fail],
                 [fresh(lambda d: conj(cdro(l, d), memberrevo(x, d))), succeed],
                 else_clause=[caro(l, x)])

def reverse_list(l):
    return run(fresh(lambda y: memberrevo(y, l)))

@adapt_iterables_to_conses(lambda x, l, out: {l, out})
def memo(x, l, out):
    return fresh(lambda a, d: conj(unify([a] + d, l),
                                   conde([unify(a, x), unify(l, out)],
                                         else_clause=[memo(x, d, out)])))


@adapt_iterables_to_conses(lambda x, l, out: {l, out})
def rembero(x, l, out):
    return conde([nullo(l), nullo(out)],
                 #[unify([x] + out, l), succeed], # in order to write this one we␣
↪should promote `cons` to a class and implement __add__ and __radd__
                 [caro(l, x), cdro(l, out)],
                 else_clause=[fresh(lambda a, d, res: conj(unify([a] + d, l),
                                                           rembero(x, d, res),
                                                           unify([a] + res, out)))])

@adapt_iterables_to_conses(lambda s, l: {l})
def surpriseo(s, l):
    return rembero(s, l, l)

@adapt_iterables_to_conses(all_arguments)
```

```
def appendo(l, s, out):
    return conde([nullo(l), unify(s, out)],
                 else_clause=[fresh(lambda a, d, res: conj(unify([a] + d, l),
                                                           appendo(d, s, res),
                                                           unify([a] + res, out)))])

@adapt_iterables_to_conses(all_arguments)
def appendso(l, s, out):
    return conde([nullo(l), unify(s, out)],
                 else_clause=[fresh(lambda a, d, res: conj(unify([a] + d, l),
                                                           unify([a] + res, out),
                                                           appendso(d, s, res)))])

@adapt_iterables_to_conses(all_arguments)
def swappendso(l, s, out):
    return conde([succeed, fresh(lambda a, d, res: conj(unify([a] + d, l),
                                                        unify([a] + res, out),
                                                        swappendso(d, s, res)))],
                 else_clause=[nullo(l), unify(s, out)])

def bswappendso(bound):

    with delimited(bound) as D:
        @adapt_iterables_to_conses(all_arguments)
        def R(l, s, out):
            return D(conde([succeed, fresh(lambda a, d, res: conj(unify([a] + d, l),
                                                                  unify([a] + res,␣
↪out),
                                                                  R(d, s, res)))],
                           else_clause=[nullo(l), unify(s, out)]))
    return R

@adapt_iterables_to_conses(lambda x, out: {x})
def unwrapo(x, out):
    return conde([fresh(lambda a, d: conj(unify([a] + d, x), unwrapo(a, out))),␣
↪succeed],
                 else_clause=[unify(x, out)])

@adapt_iterables_to_conses(all_arguments)
def unwrapso(x, out):
    return conde([succeed, unify(x, out)],
                 else_clause=[fresh(lambda a, d: conj(unify([a] + d, x), unwrapso(a,␣
↪out)))])

@adapt_iterables_to_conses(all_arguments)
def flatteno(s, out):
    def F(a, d, out_a, out_d):
        return conj(unify([a] + d, s),
                    flatteno(a, out_a),
                    flatteno(d, out_d),
                    appendo(out_a, out_d, out))
    return conde([nullo(s), nullo(out)],
                 [fresh(F), succeed],
                 else_clause=[unify([s], out)])


@adapt_iterables_to_conses(all_arguments)
def flattenrevo(s, out):
```

```
    def F(a, d, out_a, out_d):
        return conj(unify([a] + d, s),
                    flattenrevo(a, out_a),
                    flattenrevo(d, out_d),
                    appendo(out_a, out_d, out))
    return conde([succeed, unify([s], out)],
                 [nullo(s), nullo(out)],
                 else_clause=[fresh(F)])

def anyo(g):
    return conde([g, succeed],
                 else_clause=[fresh(lambda: anyo(g))]) # by η-inversion

nevero = anyo(fail) # `nevero` ever succeeds because although the question of the␣
↪first `conde` line within `anyo` fails,
                    # the answer of the second `conde` line, namely `anyo(fail)` is␣
↪where we started.
alwayso = anyo(succeed) # `alwayso` always succeeds any number of times, whereas␣
↪`succeed` can succeed only once

def succeed_at_least(g, times=1):
    return conde(*[[succeed, succeed] for _ in range(times)],
                 else_clause=[g])

def bit_xoro(α, β, γ):
    return conde([unify(0, α), unify(0, β), unify(0, γ)],
                 [unify(1, α), unify(0, β), unify(1, γ)],
                 [unify(0, α), unify(1, β), unify(1, γ)],
                 [unify(1, α), unify(1, β), unify(0, γ)],)

def bit_ando(α, β, γ):
    return conde([unify(0, α), unify(0, β), unify(0, γ)],
                 [unify(1, α), unify(0, β), unify(0, γ)],
                 [unify(0, α), unify(1, β), unify(0, γ)],
                 [unify(1, α), unify(1, β), unify(1, γ)],)

def half_addero(α, β, γ, δ):
    return conj(bit_xoro(α, β, γ), bit_ando(α, β, δ))

def full_addero(ϵ, α, β, γ, δ):
    return fresh(lambda w, αβ, wϵ: conj(half_addero(α, β, w, αβ),
                                        half_addero(ϵ, w, γ, wϵ),
                                        bit_xoro(αβ, wϵ, δ)))

@adapt_iterables_to_conses(all_arguments, ctor=num.build)
def zeroo(n):
    return unify([], n)

@adapt_iterables_to_conses(all_arguments, ctor=num.build)
def oneo(n):
    return unify([1], n)

@adapt_iterables_to_conses(all_arguments, ctor=num.build)
def twoo(n):
    return unify([0, 1], n)

@adapt_iterables_to_conses(all_arguments, ctor=num.build)
def threeo(n):
```

```
    return unify([1, 1], n)

@adapt_iterables_to_conses(all_arguments, ctor=num.build)
def poso(n):
    return pairo(n)

@adapt_iterables_to_conses(all_arguments, ctor=num.build)
def greater_than_oneo(n):
    return fresh(lambda a, ad, dd: unify([a, ad] + dd, n))

@adapt_iterables_to_conses(all_arguments, ctor=num.build)
def rightmost_representative(n):
    raise NotImplemented

@adapt_iterables_to_conses(lambda δ, n, m, r: {n: num.build, m: num.build, r: num.
↪build,})
def addero(δ, n, m, r):
    return condi([unify(0, δ), zeroo(m), unify(n, r)],
                 [unify(0, δ), poso(m), zeroo(n), unify(m, r)],
                 [unify(1, δ), zeroo(m), fresh(lambda: addero(0, n, [1], r))],
                 [unify(1, δ), poso(m), zeroo(n), fresh(lambda: addero(0, [1], m, r))],
                 [oneo(n), oneo(m), fresh(lambda α, β: conj(full_addero(δ, 1, 1, α,␣
↪β), unify([α, β], r)))],
                 [oneo(n), _addero(δ, [1], m, r)],
                 [greater_than_oneo(n), oneo(m), _addero(δ, [1], n, r)], # we delete␣
↪`greater_than_oneo(r)` respect to The Reasoned Schemer
                 [greater_than_oneo(n), _addero(δ, n, m, r)])


@adapt_iterables_to_conses(lambda δ, n, m, r: {n: num.build, m: num.build, r: num.
↪build,})
def _addero(δ, n, m, r): # alias for `gen_adder` as it appears in The Reasoned Schemer
    return fresh(lambda α, β, γ, ϵ, x, y, z:
                    conj(unify((α, x), n),
                         unify((β, y), m), poso(y),
                         unify((γ, z), r), poso(z),
                         conji(full_addero(δ, α, β, γ, ϵ),
                               addero(ϵ, x, y, z))))

@adapt_iterables_to_conses(all_arguments, ctor=num.build)
def pluso(n, m, k):
    return addero(0, n, m, k)

@adapt_iterables_to_conses(all_arguments, ctor=num.build)
def minuso(n, m, k):
    return pluso(k, m, n)

def not_thingo(x, *, what):
    return conda([unify(what, x), fail],
                 else_clause=[succeed])

def onceo(g):
    return condu([g, succeed])

@adapt_iterables_to_conses(all_arguments, ctor=num.build)
def bumpo(n, x):
    return conde([unify(n, x), succeed],
                 else_clause=[fresh(lambda m: conj(minuso(n, [1], m), bumpo(m, x)))])
```

```
def gentesto(op, *args):
    return onceo(fresh(lambda *lvars: conj(op(*lvars), *[unify(a, l) for a, l in␣
↪zip(args, lvars)]),
                       arity=len(args)))

def enumerateo(op, r, n):
    return fresh(lambda i, j, k: conj(bumpo(n, i),
                                      bumpo(n, j),
                                      op(i, j, k),
                                      gentesto(op, i, j, k),
                                      unify([i, j, k], r)))


@adapt_iterables_to_conses(lambda δ, n, m, r: {n: num.build, m: num.build, r: num.
↪build,})
def _addereo(δ, n, m, r): # alias for `gen_adder` as it appears in The Reasoned Schemer
    return fresh(lambda α, β, γ, ϵ, x, y, z:
                     conj(unify((α, x), n),
                          unify((β, y), m), poso(y),
                          unify((γ, z), r), poso(z),
                          conj(full_addero(δ, α, β, γ, ϵ),
                               addereo(ϵ, x, y, z))))

@adapt_iterables_to_conses(lambda δ, n, m, r: {n: num.build, m: num.build, r: num.
↪build,})
def addereo(δ, n, m, r):
    return condi([unify(0, δ), zeroo(m), unify(n, r)],
                 [unify(0, δ), poso(m), zeroo(n), unify(m, r)],
                 [unify(1, δ), zeroo(m), fresh(lambda: addereo(0, n, [1], r))],
                 [unify(1, δ), poso(m), zeroo(n), fresh(lambda: addereo(0, [1], m,␣
↪r))],
                 [oneo(n), oneo(m), fresh(lambda α, β: conj(full_addero(δ, 1, 1, α,␣
↪β),
                                                            unify([α, β], r)))],
                 [oneo(n), _addereo(δ, [1], m, r)],
                 [greater_than_oneo(n), oneo(m), _addereo(δ, [1], n, r)], # we delete␣
↪`greater_than_oneo(r)` respect to The Reasoned Schemer
                 [greater_than_oneo(n), _addereo(δ, n, m, r)])

@adapt_iterables_to_conses(all_arguments, ctor=num.build)
def pluseo(n, m, k):
    return addereo(0, n, m, k)


@adapt_iterables_to_conses(all_arguments, ctor=num.build)
def multiplyo(n, m, p):
    return condi([zeroo(n), zeroo(p)],
                 [poso(n), zeroo(m), zeroo(p)],
                 [oneo(n), poso(m), unify(m, p)],
                 [poso(n), oneo(m), unify(n, p)],
                 [fresh(lambda x, z: conj(unify((0, x), n), poso(x),
                                          unify((0, z), p), poso(z),
                                          greater_than_oneo(m),
                                          fresh(lambda: multiplyo(x, m, z)))),␣
↪succeed],
                 [fresh(lambda x, y: conj(unify((1, x), n), poso(x),
                                          unify((0, y), m), poso(y),
```

```
                                        fresh(lambda: multiplyo(m, n, p)))),␣
↪succeed],
                [fresh(lambda x, y: conj(unify((1, x), n), poso(x),
                                         unify((1, y), m), poso(y),
                                         multiply_oddo(x, n, m, p))), succeed])

@adapt_iterables_to_conses(all_arguments, ctor=num.build)
def multiply_oddo(x, n, m, p):
    return fresh(lambda q: conj(multiply_boundo(q, p, n, m),
                                multiplyo(x, m, q),
                                pluso((0, q), m, p)))

@adapt_iterables_to_conses(all_arguments, ctor=num.build)
def multiply_boundo(q, p, n, m):
    return conde([nullo(q), pairo(p)],
                 else_clause=[fresh(lambda x, y, z:
                                    conj(cdro(q, x),
                                         cdro(p, y),
                                         condi([nullo(n), cdro(m, z)),␣
↪fresh(lambda: multiply_boundo(x, y, z, []))],
                                                 else_clause=[cdro(n, z),␣
↪fresh(lambda: multiply_boundo(x, y, z, m))]))])])

@adapt_iterables_to_conses(all_arguments, ctor=num.build)
def length_equalo(n, m):
    return conde([zeroo(n), zeroo(m)],
                 [oneo(n), oneo(m)],
                 else_clause=[fresh(lambda α, β, x, y:
                                    conj(unify((α, x), n), poso(x),
                                         unify((β, y), m), poso(y),
                                         fresh(lambda: length_equalo(x, y))))])

@adapt_iterables_to_conses(all_arguments, ctor=num.build)
def length_lto(n, m):
    return conde([zeroo(n), poso(m)],
                 [oneo(n), greater_than_oneo(m)],
                 else_clause=[fresh(lambda α, β, x, y:
                                    conj(unify((α, x), n), poso(x),
                                         unify((β, y), m), poso(y),
                                         fresh(lambda: length_lto(x, y))))])

def _length_leqo(n, m, condo):
    return condo([length_equalo(n, m), succeed],
                 [length_lto(n, m), succeed])

@adapt_iterables_to_conses(all_arguments, ctor=num.build)
def lengthe_leqo(n, m):
    return _length_leqo(n, m, conde)

@adapt_iterables_to_conses(all_arguments, ctor=num.build)
def lengthi_leqo(n, m):
    return _length_leqo(n, m, condi)

@adapt_iterables_to_conses(all_arguments, ctor=num.build)
def lto(n, m):
    return condi([length_lto(n, m), succeed],
                 [length_equalo(n, m), fresh(lambda x: conj(poso(x), pluso(n, x,␣
↪m)))])
```

```
@adapt_iterables_to_conses(all_arguments, ctor=num.build)
def leq(n, m):
    return disj(unify(n, m), lto(n, m))
    return condi([unify(n, m), succeed],
                 [lto(n, m), succeed])

@adapt_iterables_to_conses(all_arguments, ctor=num.build)
def divmodo(n, m, q, r):
    return condi([zeroo(q), unify(n, r), lto(r, m)],
                 [oneo(q), zeroo(r), equalo(n, m), lto(r, m)],
                 [lto(m, n), lto(r, m), fresh(lambda mq:
                                              conj(length_equalo(mq, n),
                                                   multiplyo(m, q, mq),
                                                   pluso(mq, r, n)))])

@adapt_iterables_to_conses(all_arguments, ctor=num.build)
def divmod_proo(n, m, q, r):
    return condi([zeroo(q), lto(n, m), unify(r, n)],
                 [oneo(q), length_equalo(n, m), pluso(r, m, n), lto(r, m)],
                 else_clause=[conji(length_lto(m, n), lto(r, m), poso(q),
                                    fresh(lambda n_h, n_l, q_h, q_l, qlm, qlmr, rr, r_
↪h:
                                                conji(splito(n, r, n_l, n_h),
                                                      splito(q, r, q_l, q_h),
                                                      conde([zeroo(n_h), zeroo(q_h),␣
↪minuso(n_l, r, qlm), multiplyo(q_l, m, qlm)],
                                                            else_clause=[conji(poso(n_h),
                                                                               multiplyo(q_
↪l, m, qlm),
                                                                               pluso(qlm,␣
↪r, qlmr),
                                                                               minuso(qlmr,
↪ n_l, rr),
                                                                               splito(rr,␣
↪r, [], r_h),
                                                                               ␣
↪fresh(lambda: divmod_proo(n_h, m, q_h, r_h)))])))))])

def splito(n, r, l, h):
    return condi([zeroo(n), zeroo(l), zeroo(h)],
                 [fresh(lambda β, n_hat:
                           conj(unify((0, β, n_hat), n),
                                zeroo(r),
                                unify((β, n_hat), h),
                                zeroo(l))), succeed],
                 [fresh(lambda n_hat:
                           conj(unify((1, n_hat), n),
                                zeroo(r),
                                unify(n_hat, h),
                                oneo(l))), succeed],
                 [fresh(lambda α, β, n_hat, r_hat:
                           conj(unify((0, β, n_hat), n),
                                unify((α, r_hat), r),
                                zeroo(l),
                                fresh(lambda: splito((β, n_hat), r_hat, [], h)))),␣
↪succeed],
                 [fresh(lambda α, n_hat, r_hat:
```

```
                            conj(unify((1, n_hat), n),
                                 unify((α, r_hat), r),
                                 oneo(l),
                                 fresh(lambda: splito(n_hat, r_hat, [], h)))),␣
↪succeed],
                [fresh(lambda α, β, n_hat, r_hat, l_hat:
                            conj(unify((β, n_hat), n),
                                 unify((α, r_hat), r),
                                 unify((β, l_hat), l),
                                 poso(l),
                                 fresh(lambda: splito(n_hat, r_hat, l_hat, h)))),␣
↪succeed],)


@adapt_iterables_to_conses(all_arguments, ctor=num.build)
def logo(n, b, q, r):
    return condi([oneo(n), poso(b), zeroo(q), zeroo(r)],
                 [zeroo(q), lto(n, b), pluso(r, [1], n)],
                 [oneo(q), greater_than_oneo(b), length_equalo(n, b), pluso(r, b, n)],
                 [oneo(b), poso(q), pluso(r, [1], n)],
                 [zeroo(b), poso(q), unify(r, n)],
                 [twoo(b), fresh(lambda a, ad, dd, s: conj(poso(dd),
                                                           unify((a, ad, dd), n),
                                                           expo(n, [], q, base=2),
                                                           splito(n, dd, r, s)))],
                 [fresh(lambda a, ad, add, ddd: conde([threeo(b), succeed],
                                                      else_clause=[unify((a, ad, add,␣
↪ddd), b)])),
                  length_lto(b, n),
                  fresh(lambda bw1, bw, nw, nw1, ql1, q_l, s:
                            conj(expo(b, [], bw1, base=2),
                                 pluso(bw1, [1], bw),
                                 length_lto(q, n),
                                 fresh(lambda q_1, bwq1:
                                           conj(pluso(q, [1], q_1),
                                                multiplyo(bw, q_1, bwq1),
                                                lto(nw1, bwq1),
                                                expo(n, [], nw1, base=2),
                                                pluso(nw1, [1], nw),
                                                divmodo(nw, bw, ql1, s),
                                                pluso(q_l, [1], ql1))),
                                 conde([unify(q, q_l), succeed],
                                       else_clause=[length_lto(q_l, q)]),
                                 fresh(lambda bql, q_h, s, qdh, qd:
                                           conj(multiply_repeatedo(b, q_l, bql),
                                                divmodo(nw, bw1, q_h, s),
                                                pluso(q_l, qdh, q_h),
                                                pluso(q_l, qd, q),
                                                conde([unify(qd, qdh), succeed],
                                                      else_clause=[lto(qd, qdh)]),
                                                fresh(lambda bqd, bq1, bq:
                                                          conj(multiply_repeatedo(b,
↪ qd, bqd),
                                                               multiplyo(bql, bqd,␣
↪bq),
                                                               multiplyo(b, bq,␣
↪bq1),
                                                               pluso(bq, r, n),
```

```
                                                                    lto(n, bq1))))))),␣
↪succeed])

@adapt_iterables_to_conses(all_arguments, ctor=num.build)
def expo(n, b, q, *, base):
    return condi([oneo(n), zeroo(q)],
                 [greater_than_oneo(n), oneo(q), fresh(lambda s: splito(n, b, s,␣
↪[1]))],
                 [fresh(lambda q_1, b_2:
                           conji(unify((0, q_1), q), poso(q_1),
                                 length_lto(b, n),
                                 appendo(b, (1, b), b_2),
                                 fresh(lambda: expo(n, b_2, q_1, base=base)))),␣
↪succeed],
                 [fresh(lambda q_1, n_h, b_2, s:
                           conji(unify((1, q_1), q), poso(q_1),
                                 poso(n_h),
                                 splito(n, b, s, n_h),
                                 appendo(b, (1, b), b_2),
                                 fresh(lambda: expo(n_h, b_2, q_1, base=base)))),␣
↪succeed])

@adapt_iterables_to_conses(all_arguments, ctor=num.build)
def multiply_repeatedo(n, q, nq):
    return conde([poso(n), zeroo(q), oneo(nq)],
                 [oneo(q), unify(n, nq)],
                 [greater_than_oneo(q), fresh(lambda q_1, nq1:
                                              conj(pluso(q_1, [1], q),
                                                   fresh(lambda: multiply_
↪repeatedo(n, q_1, nq1)),
                                                   multiplyo(nq1, n, nq)))])


@adapt_iterables_to_conses(all_arguments, ctor=num.build)
def powo(b, q, n):
    return logo(n, b, q, [])
```

Monte Carlo Lock puzzle

```
from muk.sexp import *
from muk.core import *
from muk.ext import *

def machine(*, rules):
    @adapt_iterables_to_conses(all_arguments)
    def M(α, β):
        return condi(*[[r(α, β, machine=M), succeed] for r in rules])
    return M

@adapt_iterables_to_conses(all_arguments)
def nullo(l):
    return unify([], l)
```

```
@adapt_iterables_to_conses(all_arguments)
def appendo(r, s, out):

    def A(r, out):
        return conde([nullo(r), unify(s, out)],
                     else_clause=[fresh(lambda a, d, res:
                                        conj(unify([a]+d, r),
                                             unify([a]+res, out),
                                             fresh(lambda: A(d, res)),))])
    return A(r, out)

@adapt_iterables_to_conses(all_arguments)
def reverseo(a, l):
    return conde([nullo(a), nullo(l)],
                 else_clause=[fresh(lambda car, cdr, res:
                                    conji(unify([car]+cdr, a),
                                          appendo(res, [car], l),
                                          fresh(lambda: reverseo(cdr, res)),))])

@adapt_iterables_to_conses(all_arguments)
def associateo(γ, γ2γ):
    return appendo(γ, [2]+γ, γ2γ)

@adapt_iterables_to_conses(all_arguments)
def symmetrico(α):
    return reverseo(α, α)

@adapt_iterables_to_conses(all_arguments)
def repeato(γ, γγ):
    return appendo(γ, γ, γγ)

def mcculloch_first_ruleo(α, β, *, machine):
    return fresh(lambda η: conj(unify([2]+η, α), unify(η, β)))

def mcculloch_second_ruleo(α, δ2δ, *, machine):
    return fresh(lambda η, δ: conj(unify([3]+η, α), associateo(δ, δ2δ), machine(η, δ)))

def mcculloch_third_ruleo(α, δ_reversed, *, machine):
    return fresh(lambda η, δ: conj(unify([4]+η, α), reverseo(δ, δ_reversed), machine(η,
↪ δ)))

def mcculloch_fourth_ruleo(α, δδ, *, machine):
    return fresh(lambda η, δ: conj(unify([5]+η, α), repeato(δ, δδ), machine(η, δ)))

def mcculloch_fifth_ruleo(α, β, *, machine):
    return appendo([2]+β, [2], α)
    #return conji(appendo([2]+β, [2], α), complement(nullo(β)))
    #return fresh(lambda η: conji(appendo([2]+η, [2], α), unify(η, β)))

def mcculloch_sixth_ruleo(α, _2δ, *, machine):
    return fresh(lambda η, δ: conji(unify([6]+η, α), unify([2]+δ, _2δ), machine(η, δ)))

mccullocho = machine(rules=[ mcculloch_second_ruleo, mcculloch_first_ruleo, ])

def mcculloch_lawo(γ, αγ, *, machine=mccullocho):
    return fresh(lambda α: conji(appendo([3, 2]+α, [3], γ),
                                 appendo(α, γ, αγ),
                                 complement(nullo(α)),
```

```
                                    complement(nullo(γ)),
                                    machine(γ, αγ)))

@adapt_iterables_to_conses(lambda α, l: {α})
def lengtho(α, l):

    def L(α, *, acc):
        return conde([nullo(α), unify(l, acc)],
                     else_clause=[fresh(lambda a, β: conj(unify([a]+β, α),
                                                          fresh(lambda: L(β,␣
↪acc=acc+1)))))])

    return L(α, acc=0)

def leqo(a, b):
    return project(a, b, into=lambda a, b: unify(True, a <= b))


mcculloch_o = machine(rules=[ mcculloch_second_ruleo, mcculloch_third_ruleo,␣
↪mcculloch_first_ruleo, ])
mcculloch__o = machine(rules=[ mcculloch_second_ruleo, mcculloch_fourth_ruleo,␣
↪mcculloch_third_ruleo, mcculloch_first_ruleo, ])

@adapt_iterables_to_conses(all_arguments)
def opnumbero(α):
     return disj(nullo(α), fresh(lambda a, β: conj(appendo(β, [a], α),
                                      condi([unify(a, 3), succeed],
                                            [unify(a, 4), succeed],
                                            [unify(a, 5), succeed]),
                                      fresh(lambda: opnumbero(β)))))


def operationo(M, χ, M_of_χ, *, machine=mcculloch__o):
    return fresh(lambda M2χ: conji(appendo(M, [2]+χ, M2χ), machine(M2χ, M_of_χ)))

def craig_lawo(Mγ, M_of_Mγ, *, machine=mcculloch__o):
    return fresh(lambda γ: conji(mcculloch_lawo(γ, Mγ), machine(Mγ, M_of_Mγ)))

def craig_second_lawo(Mγ, M_of_Mγ, *, machine=mcculloch__o):
    return fresh(lambda γ, M, A: conji(mcculloch_lawo(γ, Mγ), appendo(A, M, Mγ),␣
↪machine(Mγ, M_of_Mγ)))

mcculloch___o = machine(rules=[ mcculloch_fourth_ruleo, mcculloch_third_ruleo,␣
↪mcculloch_sixth_ruleo,mcculloch_fifth_ruleo ])
```

`muk.core` module

Primitive goals and ctors

States streams and enumerations

Solver and interface

This documentation is automatically generated using Sphinx, which provides content search facilities, so a dedicated page is provided for free for open searches.

CHAPTER 3

About

My name is Massimo Nocentini, I'm:

- a PhD student @ University of Florence
- coding @ GitHub

Bibliography

[HF13] Jason Hemann and Daniel P. Friedman, *microKanren: A Minimal Functional Core for Relational Programming*, In Proceedings of the 2013 Workshop on Scheme and Functional Programming (Scheme '13), Alexandria, VA, 2013.

[HF15] Jason Hemann and Daniel P. Friedman, *A Framework for Extending microKanren with Constraints*, In Proceedings of the 2015 Workshop on Scheme and Functional Programming (Scheme '15), Vancouver, British Columbia, 2015.

[CF15] Cameron Swords and Daniel P. Friedman, *rKanren: Guided Search in miniKanren*, In Proceedings of the 2013 Workshop on Scheme and Functional Programming (Scheme '13), Alexandria, VA, 2013.

[GS17] N. D. Goodman and A. Stuhlmüller (electronic), *The Design and Implementation of Probabilistic Programming Languages*, Retrieved 2017-4-27 from http://dippl.org

[RS05] Daniel P. Friedman, William E. Byrd and Oleg Kiselyov, *The Reasoned Schemer*, The MIT Press, Cambridge, MA, 2005

[RS82] Raymond Eric Smullyan, *The Lady or the Tiger*, Knopf; 1st edition, 1982

[WB09] William E. Byrd, *Relational Programming in miniKanren: Techniques, Applications, and Implementations*, Ph.D. thesis, Indiana University, Bloomington, IN, 2009.