Constraint microKanren in the CLP Scheme

Jason Hemann

Chair: Daniel Friedman

Committee: Amr Sabry

Sam Tobin-Hochstadt

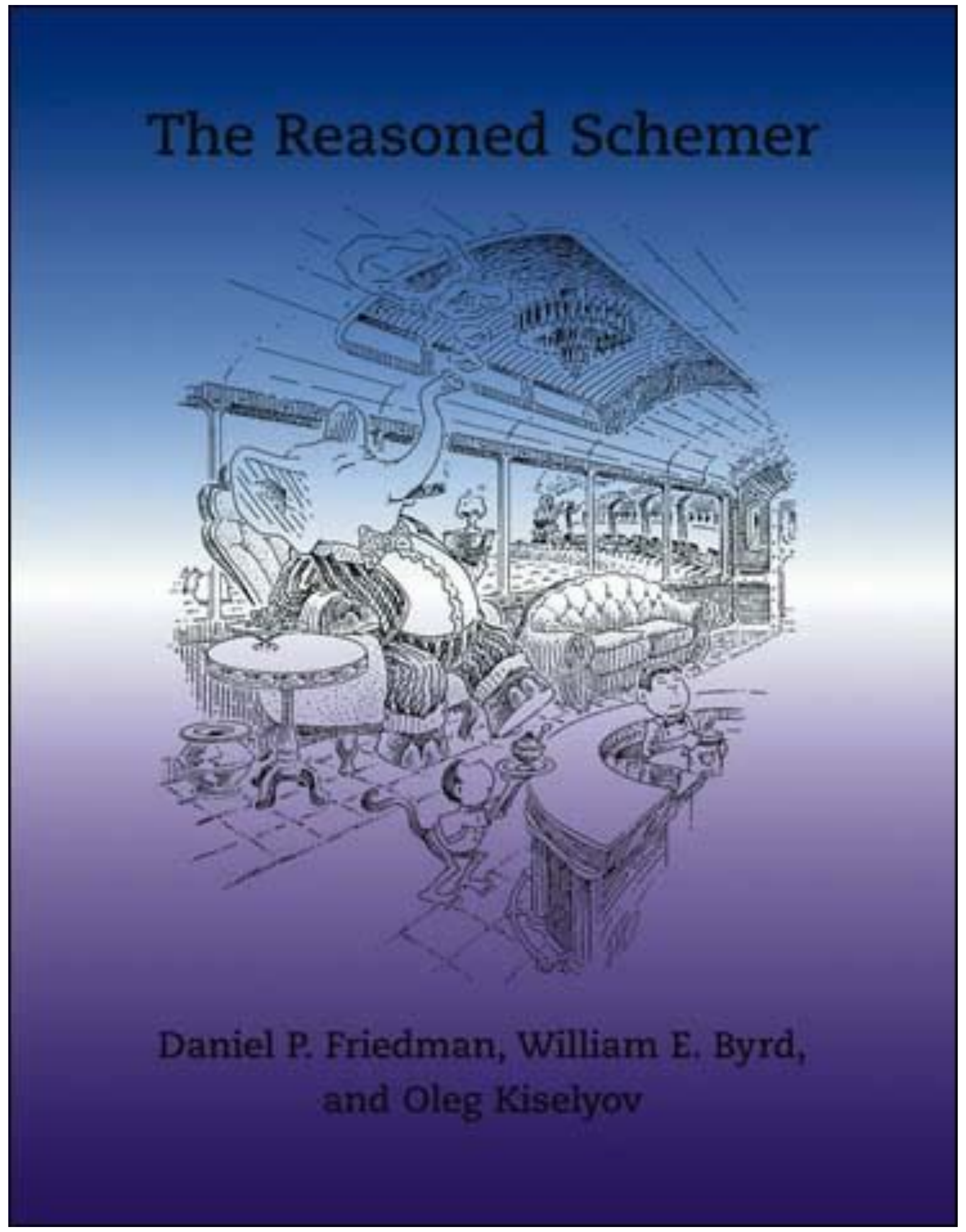
Larry Moss

12-20-2019


I want Prolog to help solve this problem!

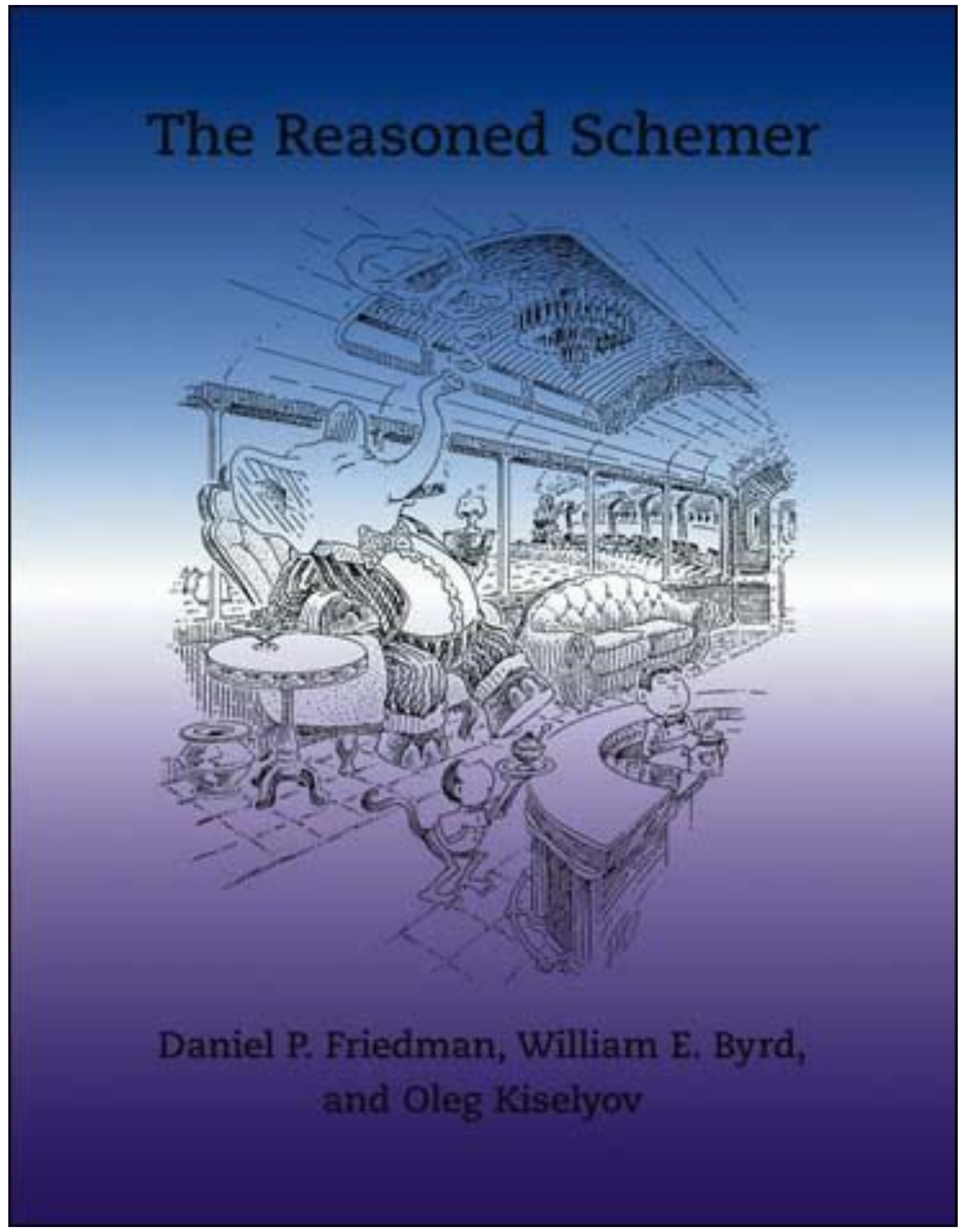

What’s a Prolog?

“Kanren Approach”



“Kanren Approach”

“little” LP DSL

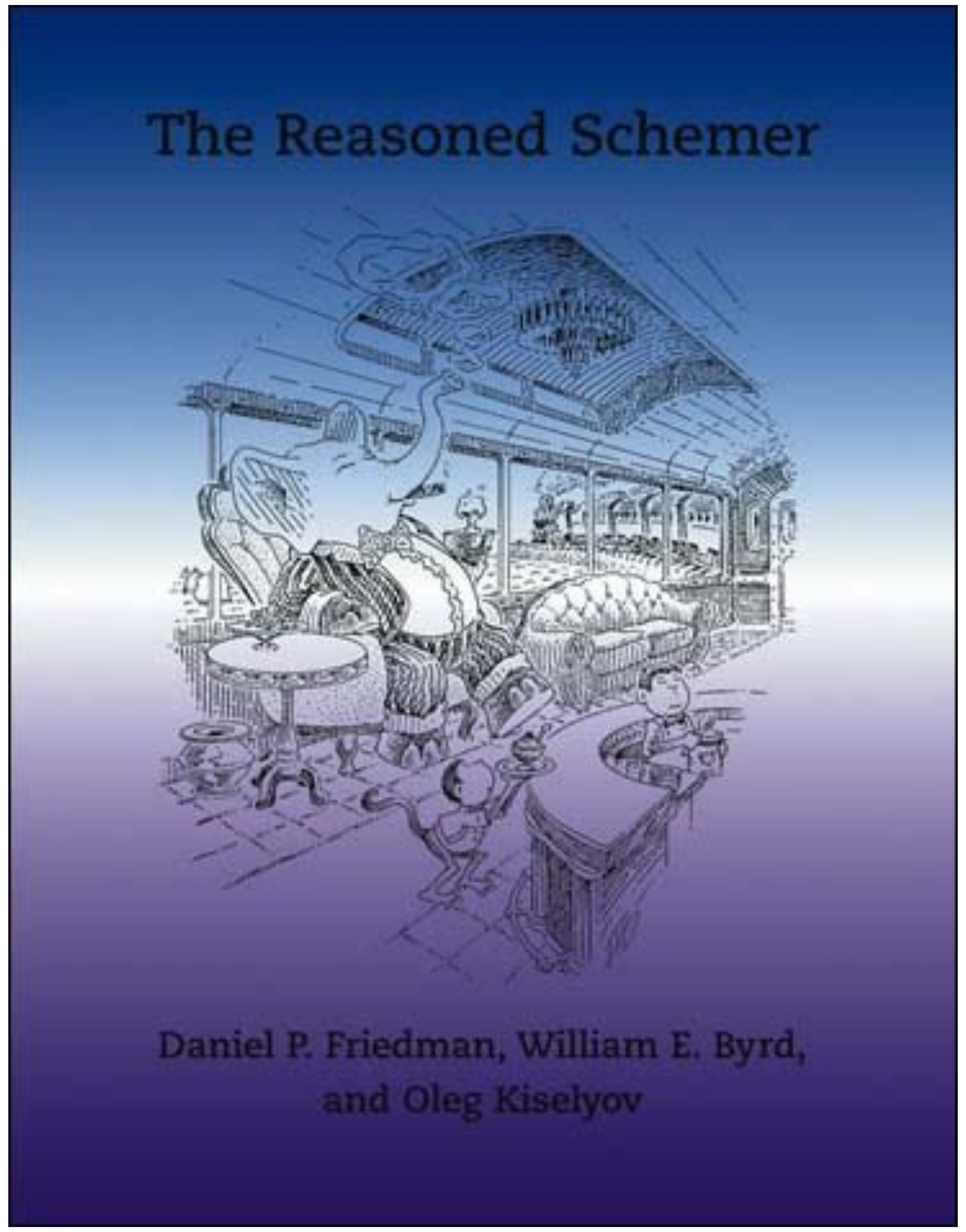


“Kanren Approach”

pure

“little” LP DSL

negation free



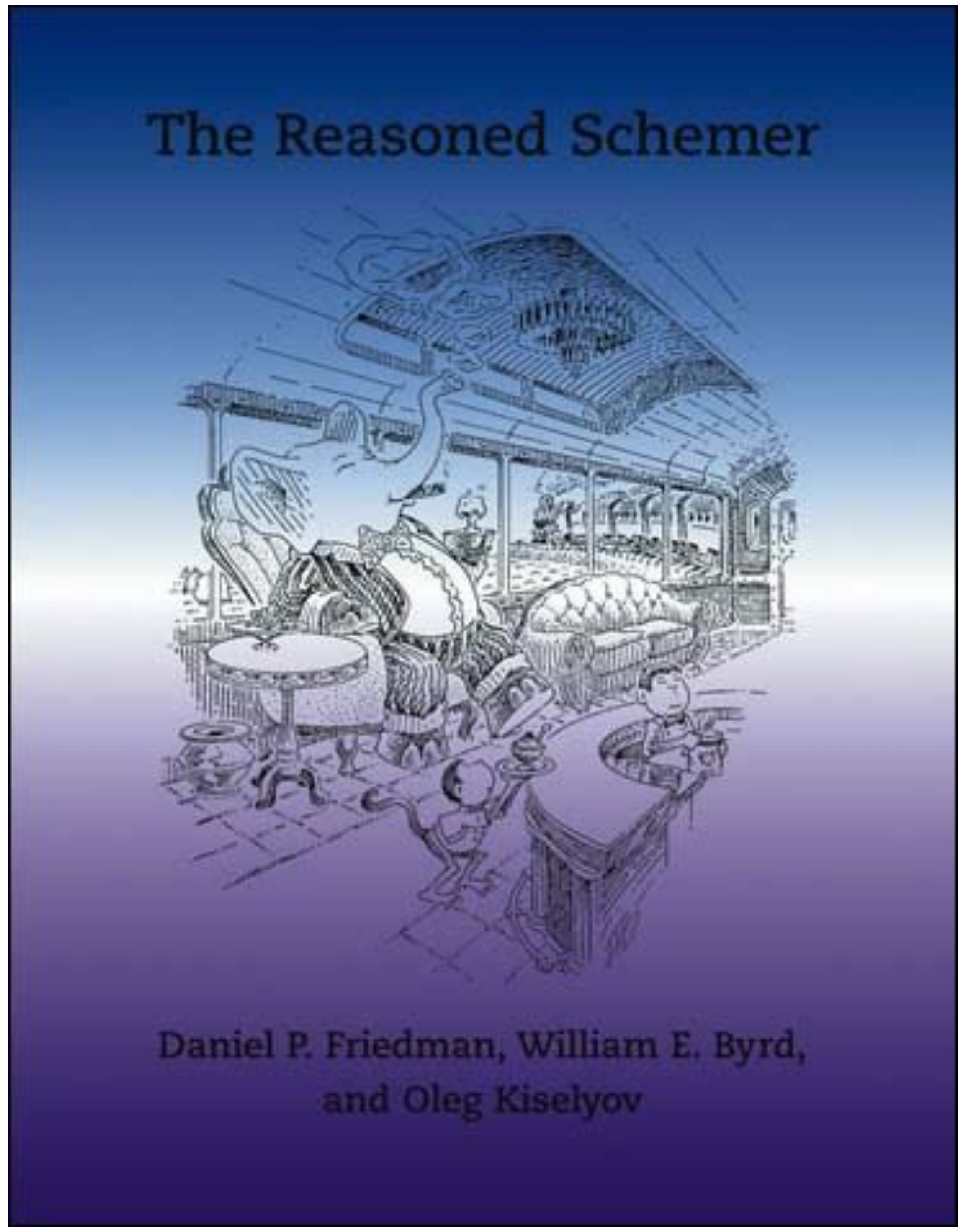
“Kanren Approach”

pure

“little” LP DSL

negation free

programmed in the completion



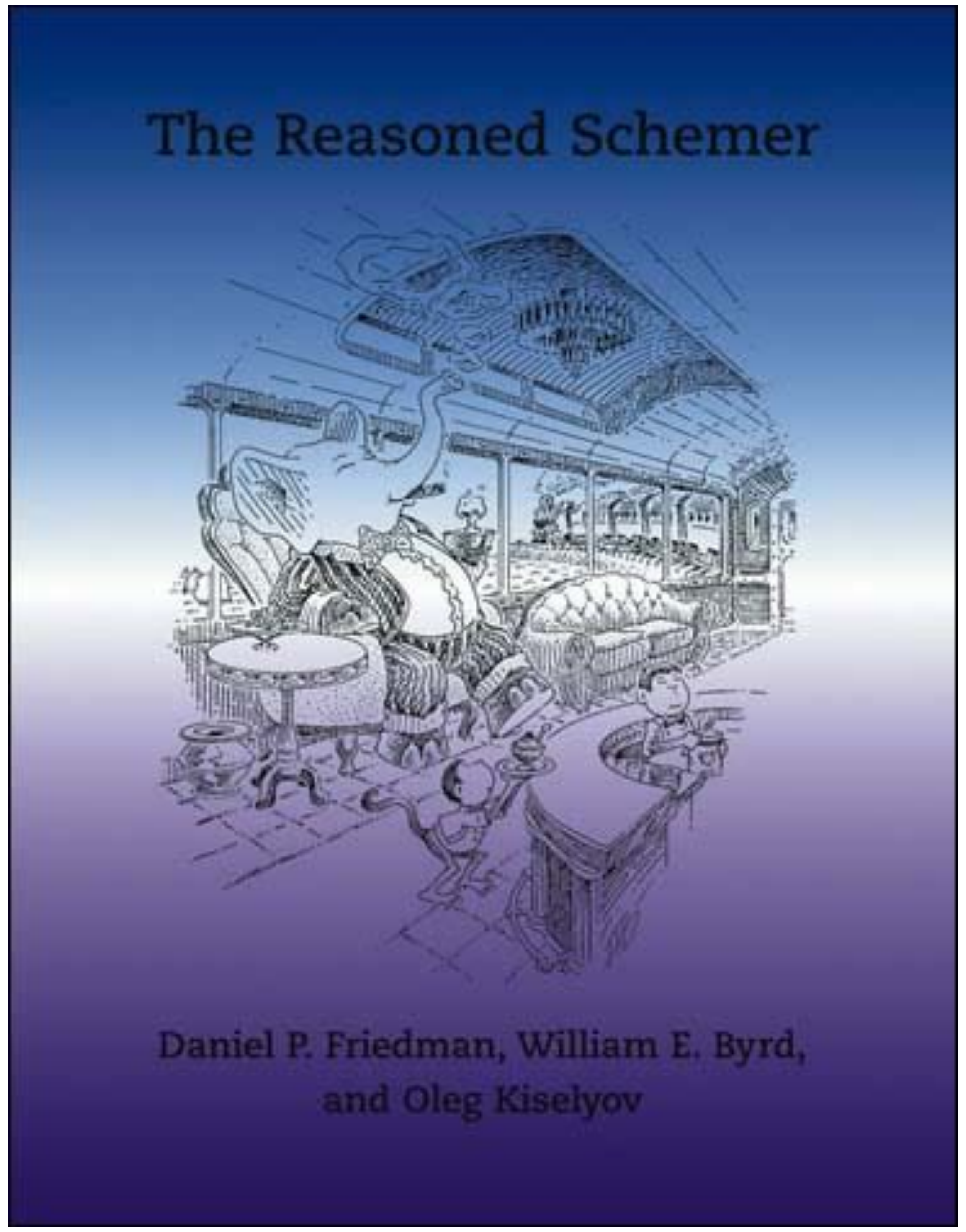
“Kanren Approach”

pure

“little” LP DSL

negation free

programmed in the completion

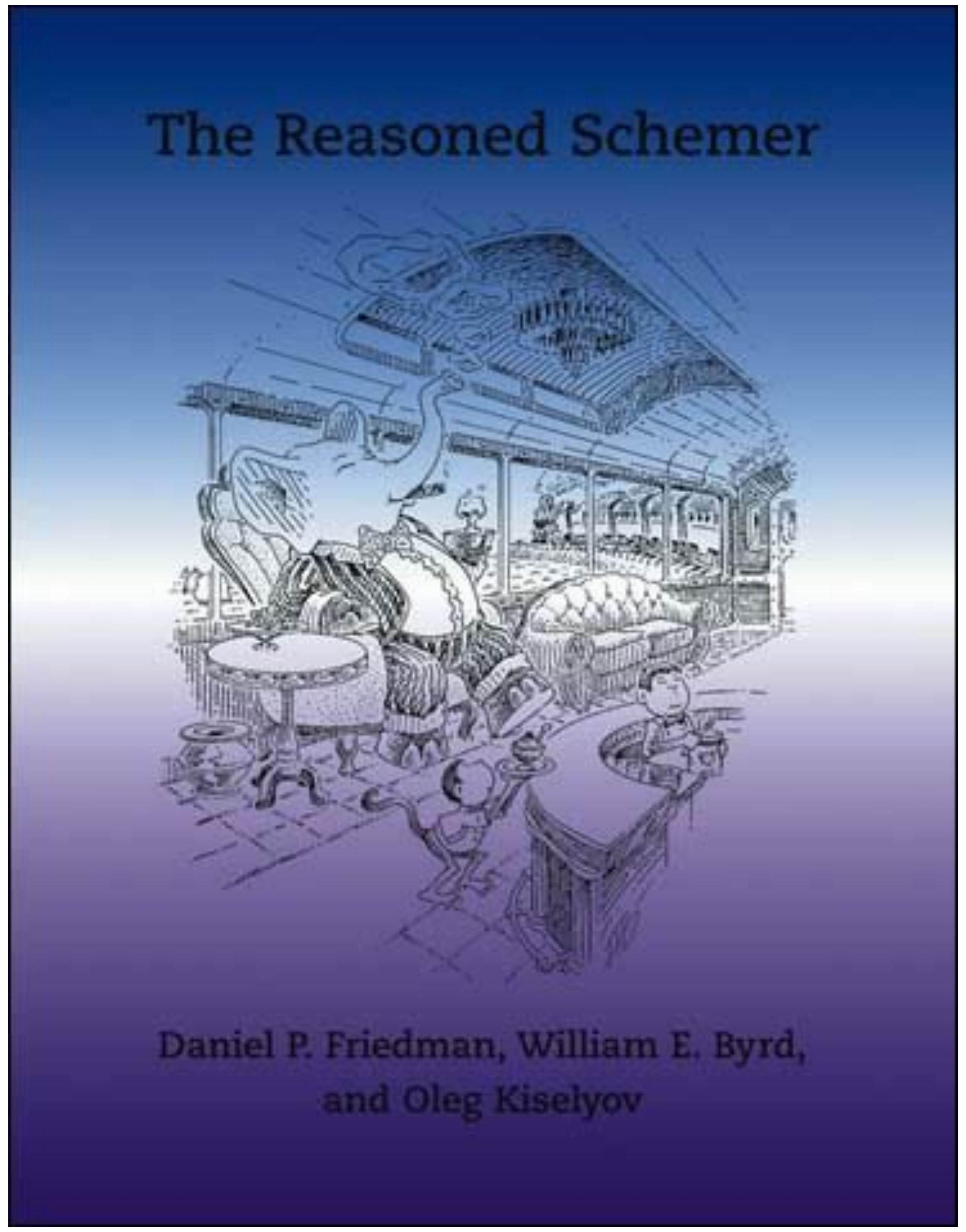


embedded implementation

shallowly embedded

via pure FP

“Kanren Approach”



pure

“little” LP DSL

negation free

programmed in the completion

embedded implementation

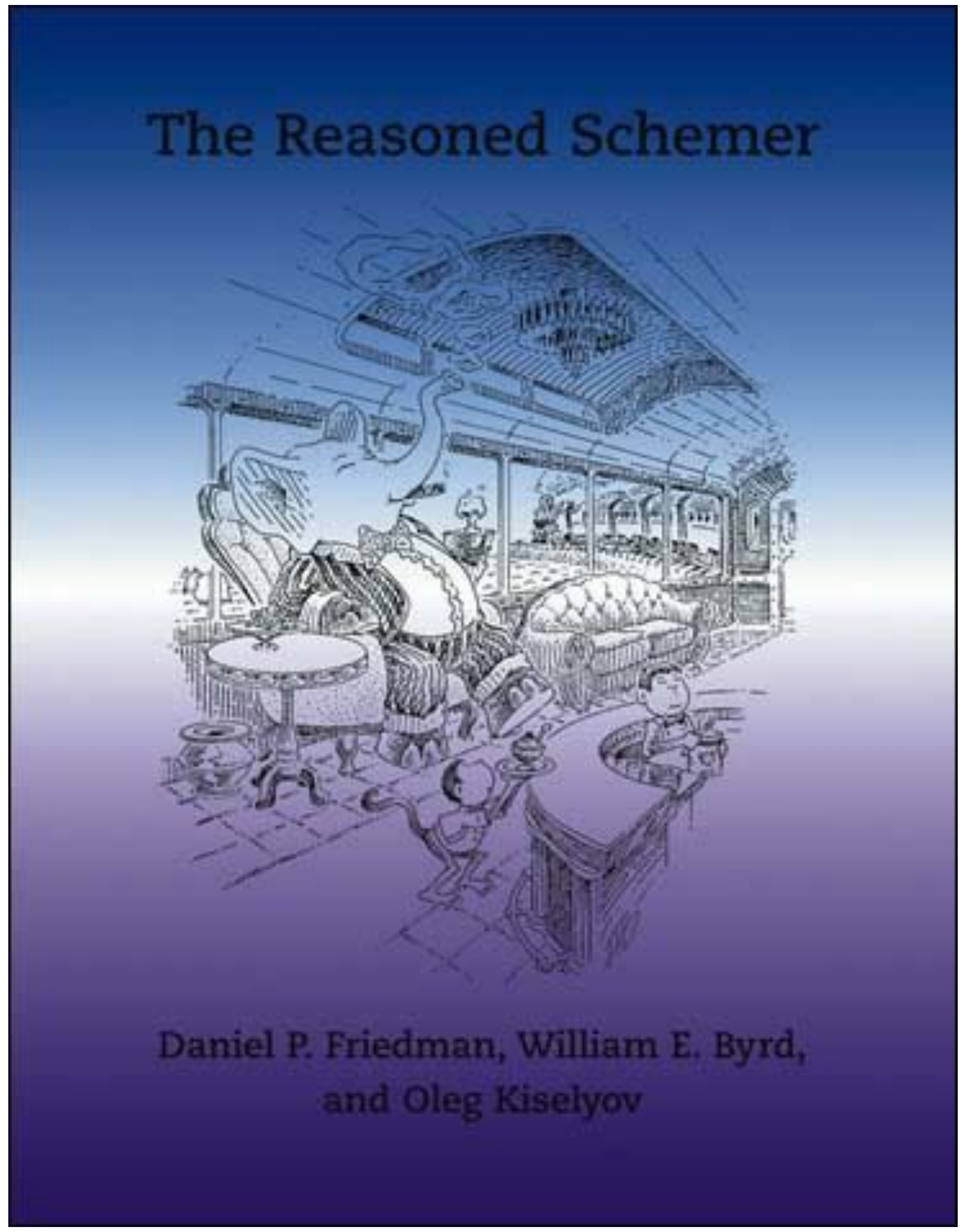
shallowly embedded

via pure FP

with an interleaving search

and additional constraints

“Kanren Approach”



(Robinson 1981)
pure

(Elcock 1989)
“little” LP DSL

(Van Emden & Kowalski 1976)
negation free

(Clark 1972)
programmed in the completion

(Robinson 1981, Seres & Spivey 2001)
with an interleaving search

(Robinson 1981)
embedded implementation

(Felleisen 1985)
shallowly embedded

(Hinze 1998, Seres & Spivey 2001)
via pure FP

(Roussel 1972, Colmerauer 1982)
and additional constraints

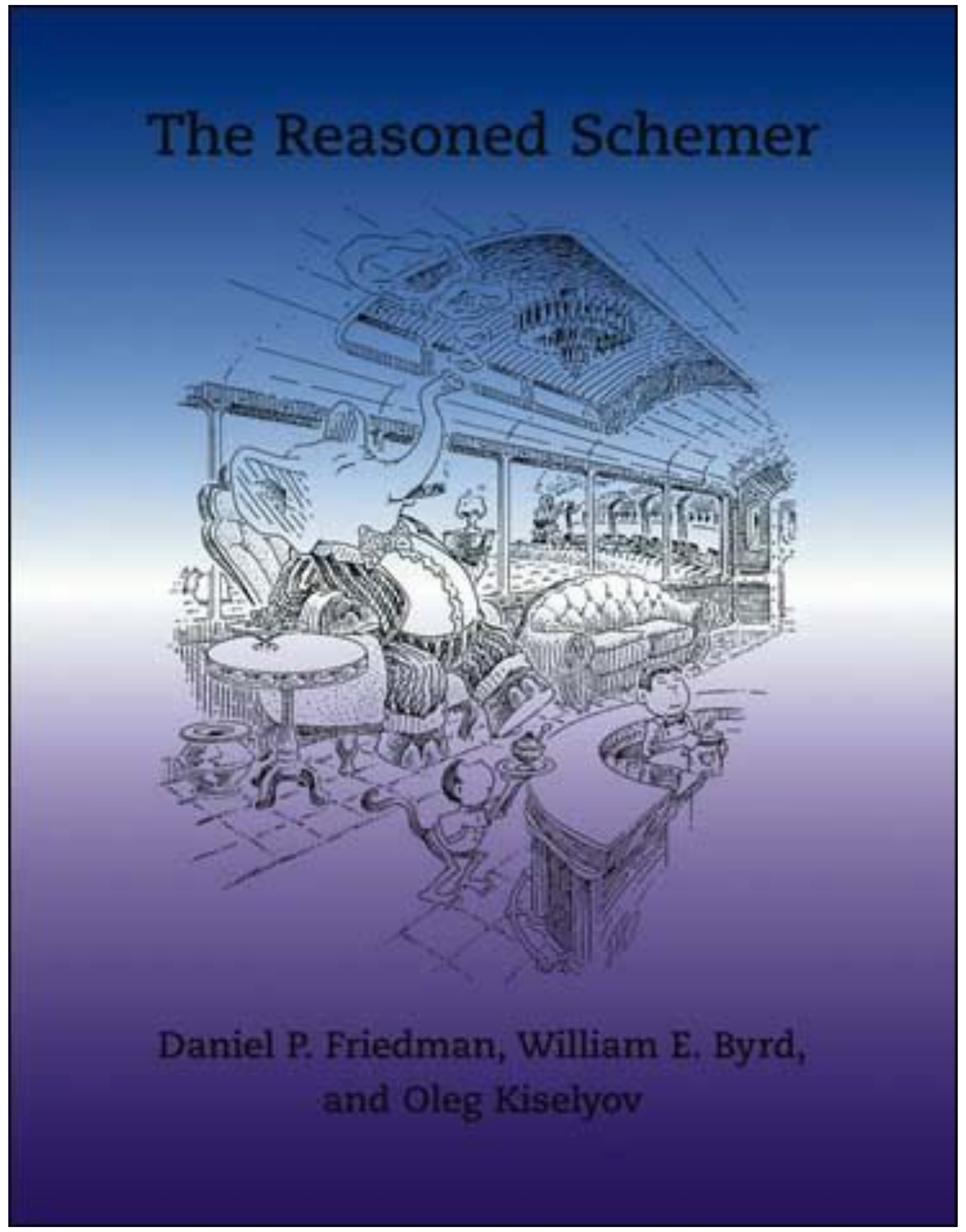
“Kanren Approach”

(Robinson 1981)
pure

(Elcock 1989)
“little” LP DSL

(Van Emden & Kowalski 1976)
negation free

(Clark 1972)
programmed in the completion



(Robinson 1981)
embedded implementation

(Felleisen 1985)
shallowly embedded

(Hinze 1998, Seres & Spivey 2001)
via pure FP

(Robinson 1981, Seres & Spivey 2001)
with an interleaving search

(Roussel 1972, Colmerauer 1982)
and additional constraints


CAUTION
AHEAD


DANGER
FALLING
ROCKS

Commingled Syntax and Control

- limits uptake to host languages with macros
- obscures simpler, intended interleaving behavior

Compounded by “Constraints”

- Describes **maybe** mK($\mathcal{X}$)—"Don't be so open minded …"
- Large implementations, unwieldy semantics
- No leveraging of scale or repetition
- Whither negation?

Compounded by “Constraints”

- Describes **maybe** mK($\mathcal{X}$)—"Don't be so open minded …"
- Large implementations, unwieldy semantics
- No leveraging of scale or repetition
- Whither negation?

microKanren + constraints

Inbox x

William Byrd

to Jason, Daniel

Mar 25 (5 days ago)

Hey Jason!

What is the state of microKanren + constraints? Is the implementation able to handle =/=, absento, symbolo, numbero? Can it handle CLP(FD)? Might it integrate with SMT?

How fast is the impl? Does it use attributed variables?

Could we build evalo/Barliman on top of it?

faster-miniKanren + Barliman is way too unwieldy (5K lines or more), and the complexity and fragility are starting to seriously slow down our research, and makes it much harder to teach the ideas.

Thanks!

--Will

unwieldy

unwieldy
complex

unwieldy
complex
fragile

unwieldy
complex
fragile
seriously slowing research

unwieldy
complex
fragile
seriously slowing research
obscures the basic ideas

My Thesis

A wide class of miniKanren languages are syntactic extensions over a small kernel logic programming language with interrelated semantics parameterized by their constraint systems, and this characterization bolsters the development of useful tools and aids in solving important tasks with pure relational programming

Roadmap

- miniKanren, briefly
- a small kernel logic programming language
- miniKanren languages are syntactic extensions
- generalizing to constraints
- interrelated semantics
- parameterized by their constraint systems
- constraint system framework
- bolsters the development of useful tools and aids in solving important tasks

Roadmap

- miniKanren, briefly
- a small kernel logic programming language
- miniKanren languages are syntactic extensions
- generalizing to constraints
- interrelated semantics
- parameterized by their constraint systems
- constraint system framework
- bolsters the development of useful tools and aids in solving important tasks

Language Examples

```
Welcome to Racket v7.4.
>
```

Language Examples

```
Welcome to Racket v7.4.
> (define-relation (member x ls o)
    (fresh (a d)
      (== ls `(,a . ,d))
      (conde
        ((== x a) (== ls o))
        ((member x d o)))))
>
```

Language Examples

```
Welcome to Racket v7.4.
> (define-relation (member x ls o)
    (fresh (a d)
      (== ls `(,a . ,d))
      (conde
        ((== x a) (== ls o))
        ((member x d o)))))
> (run* (q) (member 'x '(a x c) q))
'((x c))
>
```

Language Examples

```
Welcome to Racket v7.4.
> (define-relation (member x ls o)
    (fresh (a d)
      (== ls `(,a . ,d))
      (conde
        ((== x a) (== ls o))
        ((member x d o)))))
> (run* (q) (member 'x '(a x c) q))
'((x c))
> (run* (q) (member q '(a x c) '(x c)))
'(x)
>
```

HOST

HOST

EDSL

HOST

EDSL

INTERPRETER

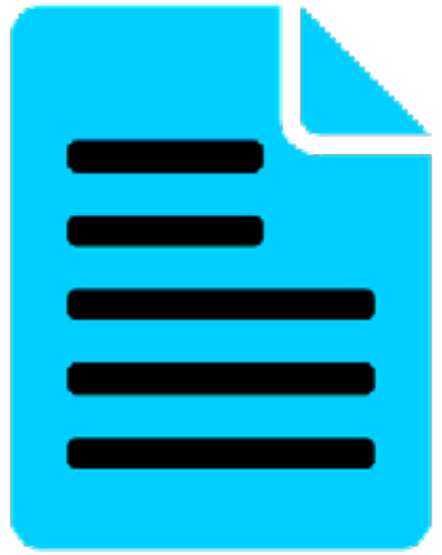
HOST

EDSL

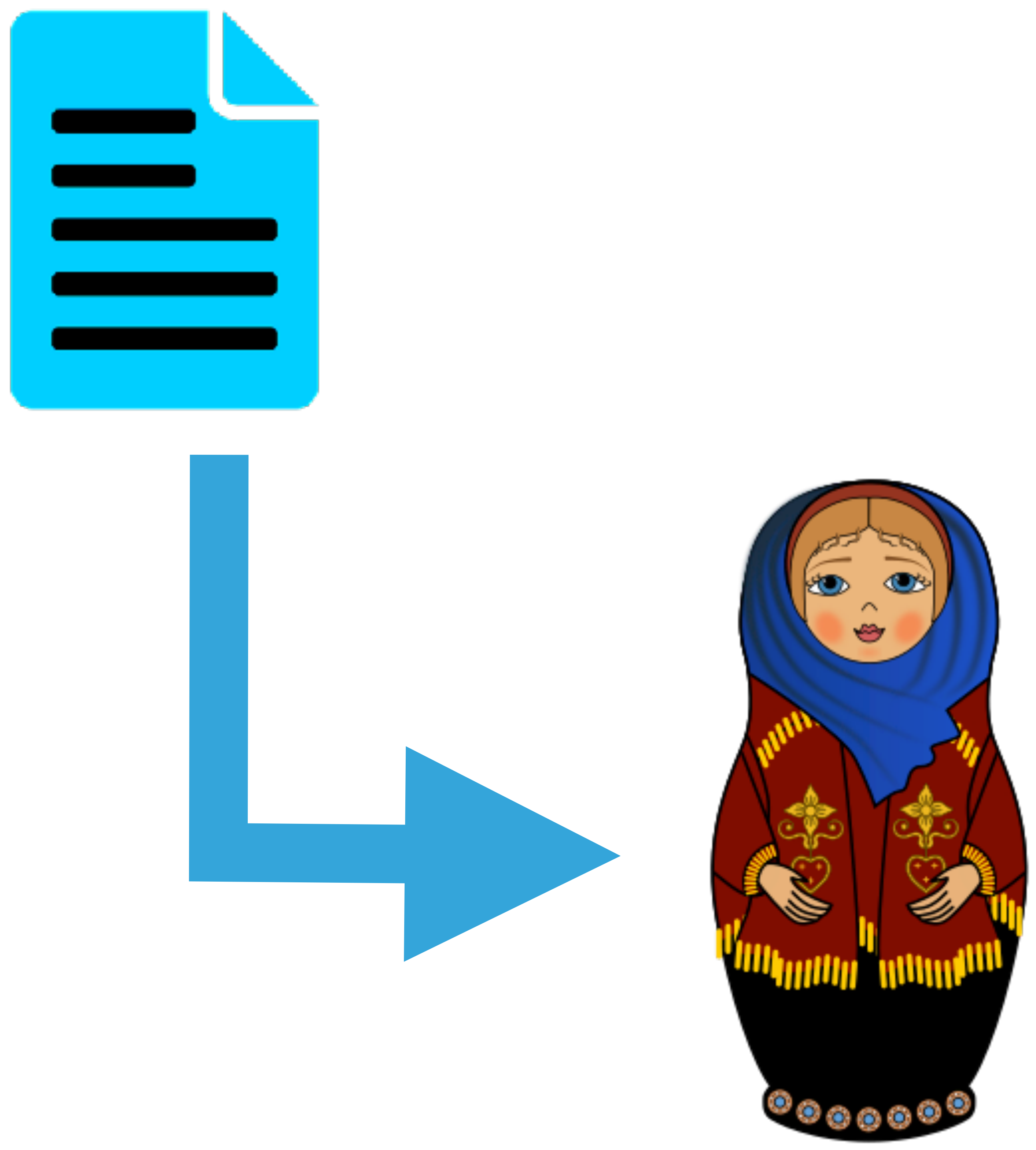
INTERPRETER

INTERPRETER

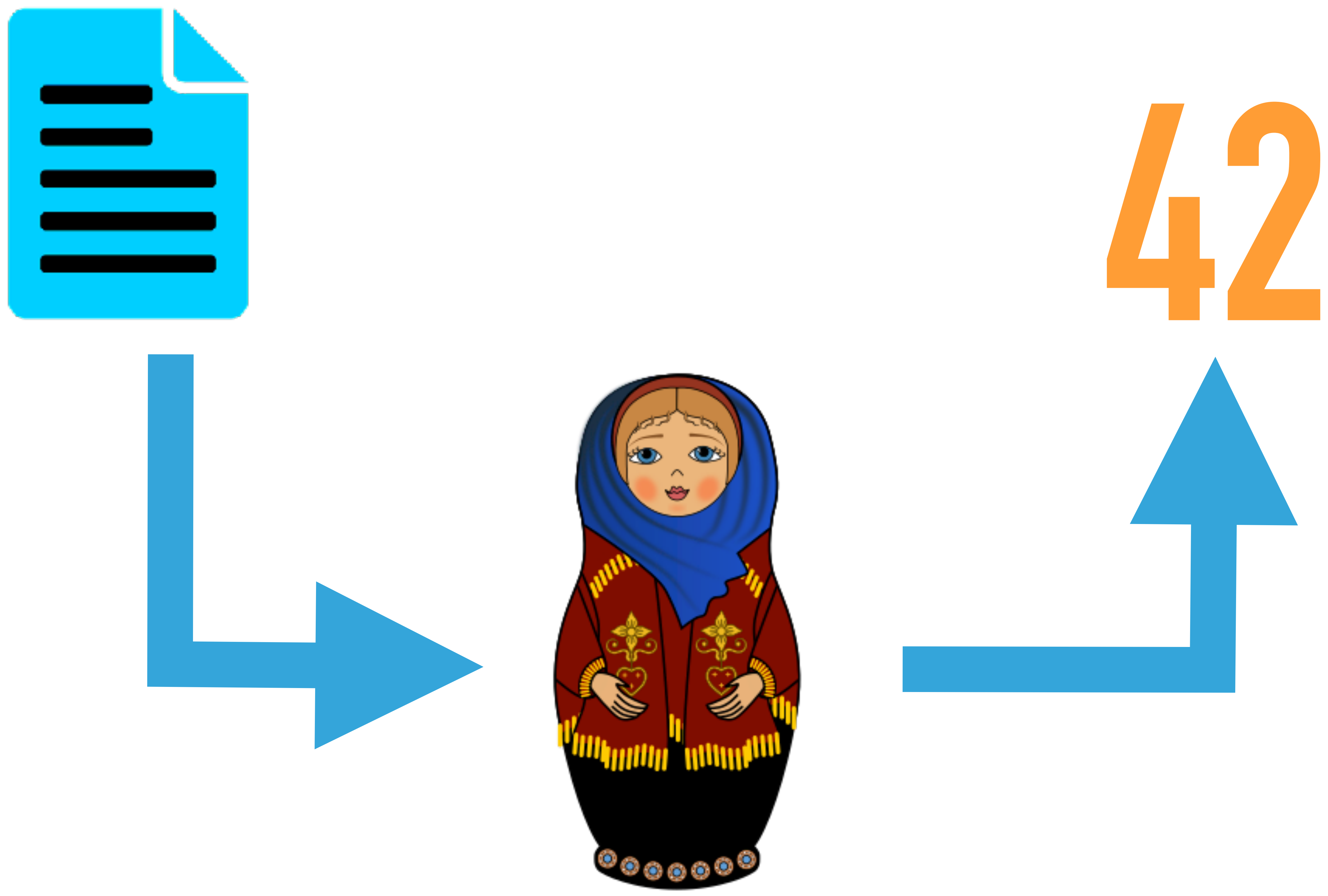
INTERPRETER (RELATIONAL)



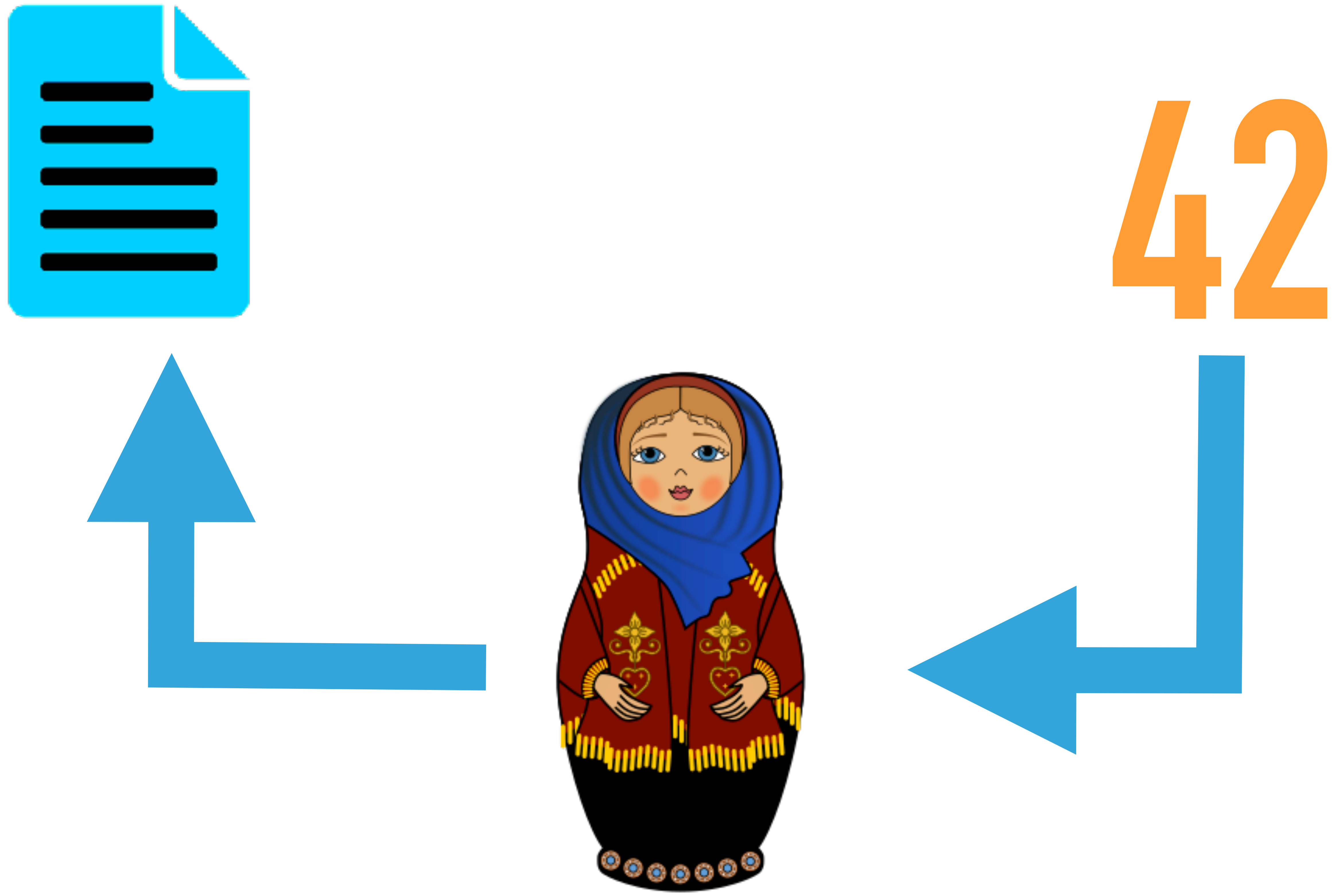
INTERPRETER
(RELATIONAL)



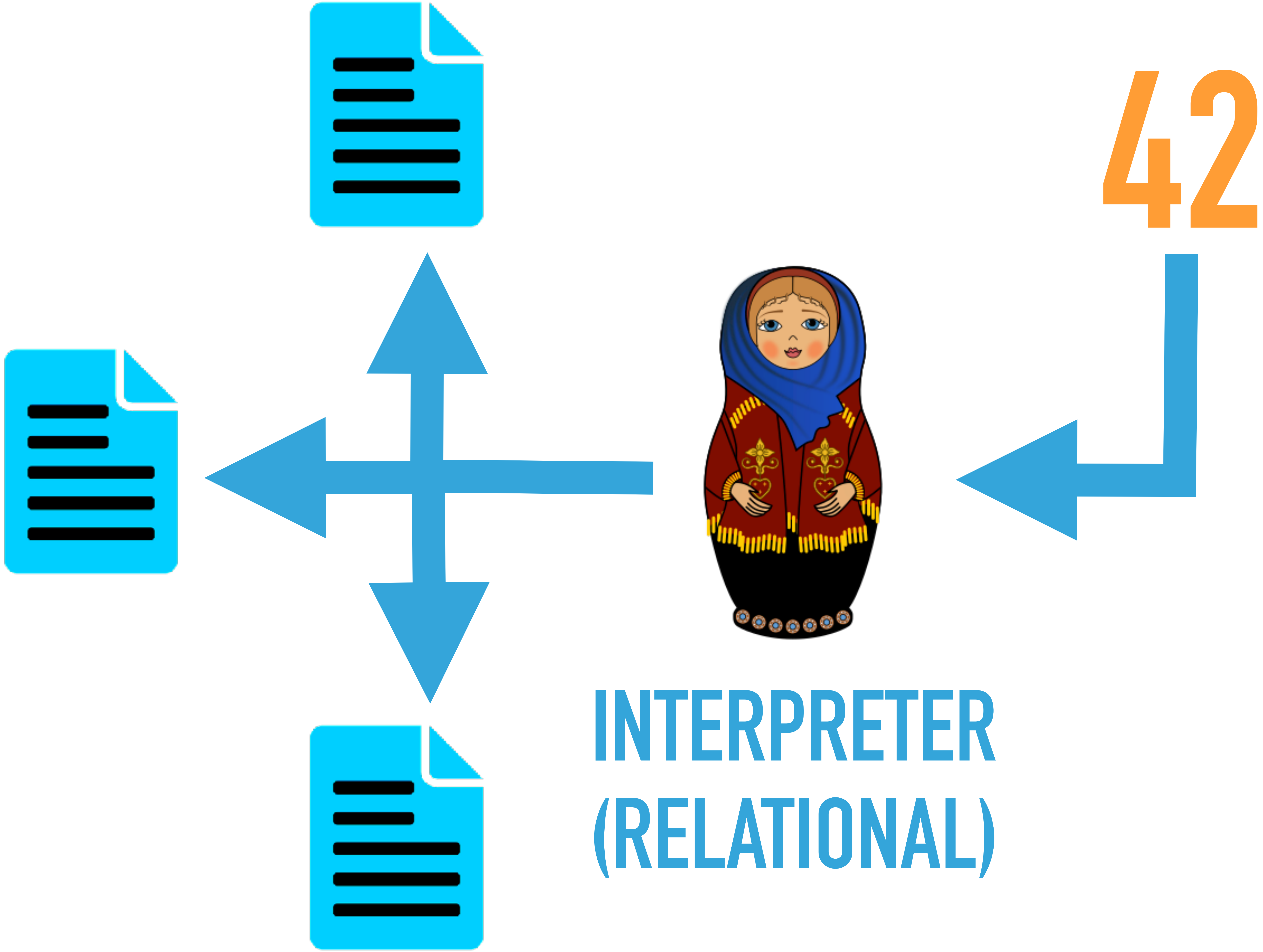
INTERPRETER
(RELATIONAL)



INTERPRETER
(RELATIONAL)



INTERPRETER
(RELATIONAL)


42
INTERPRETER
(RELATIONAL)

Relational Programming in miniKanren: Techniques, Applications, and Implementations

William E. Byrd

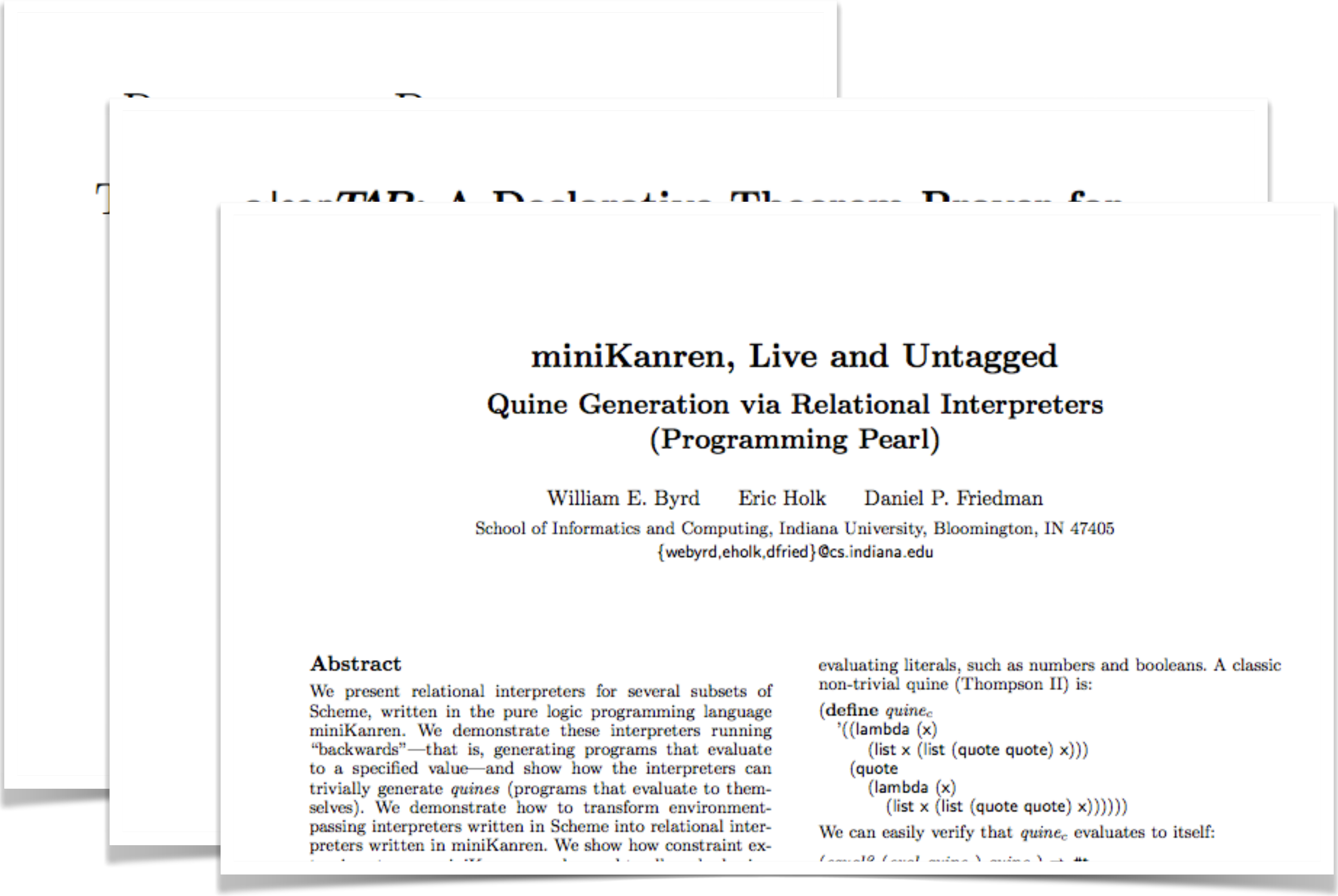
Submitted to the faculty of the
University Graduate School
in partial fulfillment of the requirements

αleanTAP: A Declarative Theorem Prover for First-Order Classical Logic

Joseph P. Near**, William E. Byrd, and Daniel P. Friedman

Indiana University, Bloomington, IN 47405
{jnear,webyrd,dfried}@cs.indiana.edu

Abstract. We present αleanTAP, a declarative tableau-based theorem prover written as a pure relation. Like leanTAP, on which it is based, αleanTAP can prove ground theorems in first-order classical logic. Since it is declarative, αleanTAP *generates* theorems and accepts non-ground theorems and proofs. The lack of mode restrictions also allows the user to provide guidance in proving complex theorems and to ask the prover to instantiate non-ground parts of theorems. We present a complete implementation of αleanTAP, beginning with a translation of leanTAP into αKanren, an embedding of nominal logic programming in Scheme. We



miniKanren, Live and Untagged

Quine Generation via Relational Interpreters (Programming Pearl)

William E. Byrd Eric Holk Daniel P. Friedman

School of Informatics and Computing, Indiana University, Bloomington, IN 47405

{webyrd,eholk,dfried}@cs.indiana.edu

Abstract

We present relational interpreters for several subsets of Scheme, written in the pure logic programming language miniKanren. We demonstrate these interpreters running "backwards"—that is, generating programs that evaluate to a specified value—and show how the interpreters can trivially generate *quines* (programs that evaluate to themselves). We demonstrate how to transform environment-passing interpreters written in Scheme into relational interpreters written in miniKanren. We show how constraint ex-

evaluating literals, such as numbers and booleans. A classic non-trivial quine (Thompson II) is:

```
(define quine_c
  '((lambda (x)
      (list x (list (quote quote) x)))
    (quote
      (lambda (x)
        (list x (list (quote quote) x))))))
```

We can easily verify that $quine_c$ evaluates to itself:

A Unified Approach to Solving Seven Programming Problems (Functional Pearl)

WILLIAM E. BYRD, University of Utah, USA
MICHAEL BALLANTYNE, University of Utah, USA
GREGORY ROSENBLATT, Toronto, Ontario, Canada
MATTHEW MIGHT, University of Utah, USA

We present seven programming challenges in Racket, and an elegant, unified approach to solving them using constraint logic programming in miniKanren.

CCS Concepts: • **Software and its engineering** → **Functional languages**; **Constraint and logic languages**; **Automatic programming**;

Additional Key Words and Phrases: relational programming, program synthesis, miniKanren, Racket, Scheme

ACM Reference format:
William E. Byrd, Michael Ballantyne, Gregory Rosenblatt, and Matthew Might. 2017. A Unified Approach to Solving Seven Programming Problems (Functional Pearl). *Proc. ACM Program. Lang.* 1, ICFP, Article 8 (September 2017), 26 pages.
https://doi.org/10.1145/3110252

A Unified Approach to Solv
Problems (Functional Pearl

WILLIAM E. BYRD, University of Utah, US
MICHAEL BALLANTYNE, University of
GREGORY ROSENBLATT, Toronto, Ontario, Canada
MATTHEW MIGHT, University of Utah, USA

We present seven programming challenges in Racket, and an elegant, unified approach to solving them using constraint logic programming in miniKanren.

CCS Concepts: • **Software and its engineering** → **Functional languages**; **Constraint and logic languages**; **Automatic programming**;

Additional Key Words and Phrases: r eme

ACM Reference format:
William E. Byrd, Michael Ballantyne ach to Solving Seven Programming Prol le 8 (September 2017), 26 pages.
https://doi.org/10.1145/3110252

Abst
We pr
Scheme
miniKa
"backw
to a sp
triviall
selves).
passing
preters

Roadmap

- miniKanren, briefly
- a small kernel logic programming language
- miniKanren languages are syntactic extensions
- generalizing to constraints
- interrelated semantics
- parameterized by their constraint systems
- constraint system framework
- bolsters the development of useful tools and aids in solving important tasks

```scheme
(define (var x) x)
(define (var? x) (number? x))

(define (find u s)
  (let ((pr (and (var? u) (assv u s))))
    (if pr (find (cdr pr) s) u)))

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

(define (occurs? x u s)
  (cond
    ((var? u) (eqv? x u))
    ((pair? u) (or (occurs? x (find (car u) s) s)
                   (occurs? x (find (cdr u) s) s)))
    (else #f)))

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

(define ((== u v) s/c)
  (let ((s (car s/c)))
    (let ((s (unify (find u s) (find v s) s)))
      (if s (list `(,s . ,(cdr s/c))) `()))))
```

```scheme
(define ((call/fresh f) s/c)
  (let ((c (cdr s/c)))
    ((f (var c)) `(,(car s/c) . ,(+ c 1)))))

(define-syntax-rule (define-relation (defname . args) g)
  (define ((defname . args) s/c) (delay/name (g s/c))))

(define (pull $) (if (promise? $) (pull (force $)) $))

(define (take n $)
  (cond
    ((null? $) '())
    ((and n (zero? (- n 1))) (list (car $)))
    (else (cons (car $)
            (take (and n (- n 1)) (pull (cdr $)))))))

(define (call/initial-state n g)
  (take n (pull (g '(() . 0)))))

(define ((disj g1 g2) s/c) ($append (g1 s/c) (g2 s/c)))
(define ((conj g1 g2) s/c) ($append-map (g1 s/c) g2))

(define ($append $1 $2)
  (cond
    ((null? $1) $2)
    ((promise? $1) (delay/name ($append $2 (force $1))))
    (else (cons (car $1) ($append (cdr $1) $2)))))

(define ($append-map $ g)
  (cond
    ((null? $) `())
    ((promise? $) (delay/name ($append-map (force $) g)))
    (else ($append (g (car $)) ($append-map (cdr $) g)))))
```

```
(define ((disj g1 g2) s/c) ($append (g1 s/c) (g2 s/c)))
(define ((conj g1 g2) s/c) ($append-map (g1 s/c) g2))

(define ($append $1 $2)
  (cond
    ((null? $1) $2)
    ((promise? $1) (delay/name ($append $2 (force $1))))
    (else (cons (car $1) ($append (cdr $1) $2)))))

(define ($append-map $ g)
  (cond
    ((null? $) `())
    ((promise? $) (delay/name ($append-map (force $) g)))
    (else ($append (g (car $)) ($append-map (cdr $) g)))))
```

```scheme
(define ((disj g1 g2) s/c) ($append (g1 s/c) (g2 s/c)))
(define ((conj g1 g2) s/c) ($append-map (g1 s/c) g2))

(define ($append $1 $2)
  (cond
    ((null? $1) $2)
    ((promise? $1) (delay/name ($append $2 (force $1))))
    (else (cons (car $1) ($append (cdr $1) $2)))))

(define ($append-map $ g)
  (cond
    ((null? $) `())
    ((promise? $) (delay/name ($append-map (force $) g)))
    (else ($append (g (car $)) ($append-map (cdr $) g)))))
```

```
(define ((disj g1 g2) s/c) ($append (g1 s/c) (g2 s/c)))
(define ((conj g1 g2) s/c) ($append-map (g1 s/c) g2))

(define ($append $1 $2)
  (cond
    ((null? $1) $2)
    (else (cons (car $1) ($append (cdr $1) $2)))))


(define ($append-map $ g)
  (cond
    ((null? $) `())
    (else ($append (g (car $)) ($append-map (cdr $) g)))))
```

```scheme
(define ((disj g1 g2) s/c) (append (g1 s/c) (g2 s/c)))
(define ((conj g1 g2) s/c) (append-map (g1 s/c) g2))

(define (append l1 l2)
  (cond
    ((null? l1) l2)
    (else (cons (car l1) (append (cdr l1) l2)))))


(define (append-map l f)
  (cond
    ((null? l) `())
    (else (append (f (car l)) (append-map (cdr l) f)))))
```

```scheme
(define ((disj g1 g2) s/c) ($append (g1 s/c) (g2 s/c)))
(define ((conj g1 g2) s/c) ($append-map (g1 s/c) g2))

(define ($append $1 $2)
  (cond
    ((null? $1) $2)
    (else (cons (car $1) ($append (cdr $1) $2)))))


(define ($append-map $ g)
  (cond
    ((null? $) `())
    (else ($append (g (car $)) ($append-map (cdr $) g)))))
```

```scheme
(define ((disj g1 g2) s/c) ($append (g1 s/c) (g2 s/c)))
(define ((conj g1 g2) s/c) ($append-map (g1 s/c) g2))

(define ($append $1 $2)
  (cond
    ((null? $1) $2)
    ((promise? $1) (delay/name ($append (force $1) $2)))
    (else (cons (car $1) ($append (cdr $1) $2)))))

(define ($append-map $ g)
  (cond
    ((null? $) `())
    ((promise? $) (delay/name ($append-map (force $) g)))
    (else ($append (g (car $)) ($append-map (cdr $) g)))))
```

Unproductive Relation

```
(define-relation (nevero x)
  (nevero x))
```

```
(define-relation (nevero x)
  (nevero x))
```

Unproductive Relation

```
(disj (nevero ‘cat) (== ‘cat ‘cat))
```

Disjunctive Query

```
(define-relation (nevero x)
  (nevero x))
```

Unproductive Relation

```
>(disj (nevero ‘cat) (== ‘cat ‘cat))
```

Disjunctive Query

```
(define-relation (nevero x)
  (nevero x))
```

Unproductive Relation

```
>(disj (nevero ‘cat) (== ‘cat ‘cat))
>      (nevero ‘cat)
```

Disjunctive Query

```
(define-relation (nevero x)
  (nevero x))
```

Unproductive Relation

```
>(disj (nevero ‘cat) (== ‘cat ‘cat))
>      (nevero ‘cat)
>      (nevero ‘cat)
```

Disjunctive Query

```
(define-relation (nevero x)
  (nevero x))
```

Unproductive Relation

```
>(disj (nevero ‘cat) (== ‘cat ‘cat))
>       (nevero ‘cat)
>       (nevero ‘cat)
        ...
```

Disjunctive Query

```
(define ((disj g1 g2) s/c) ($append (g1 s/c) (g2 s/c)))
(define ((conj g1 g2) s/c) ($append-map (g1 s/c) g2))

(define ($append $1 $2)
  (cond
    ((null? $1) $2)
    ((promise? $1) (delay/name ($append (force $1) $2)))
    (else (cons (car $1) ($append (cdr $1) $2)))))

(define ($append-map $ g)
  (cond
    ((null? $) `())
    ((promise? $) (delay/name ($append-map (force $) g)))
    (else ($append (g (car $)) ($append-map (cdr $) g)))))
```

```scheme
(define ((disj g1 g2) s/c) ($append (g1 s/c) (g2 s/c)))
(define ((conj g1 g2) s/c) ($append-map (g1 s/c) g2))

(define ($append $1 $2)
  (cond
    ((null? $1) $2)
    ((promise? $1) (delay/name ($append $2 (force $1))))
    (else (cons (car $1) ($append (cdr $1) $2)))))

(define ($append-map $ g)
  (cond
    ((null? $) `())
    ((promise? $) (delay/name ($append-map (force $) g)))
    (else ($append (g (car $)) ($append-map (cdr $) g)))))
```

Unproductive Relation

```
(define-relation (nevero x)
  (nevero x))
```

```
(define-relation (nevero x)
  (nevero x))
```

Unproductive Relation

```
(disj (nevero ‘cat) (== ‘cat ‘cat))
```

Disjunctive Query

```
> (disj (nevero ‘cat) (== ‘cat ‘cat))
```

```
> (disj (nevero ‘cat) (== ‘cat ‘cat))
```

```
>(disj (nevero ‘cat) (== ‘cat ‘cat))
```

```
>(disj (nevero 'cat) (== 'cat 'cat))
```

```
(== 'cat 'cat)  (nevero 'cat)
```

```
>(disj (nevero 'cat) (== 'cat 'cat))
```

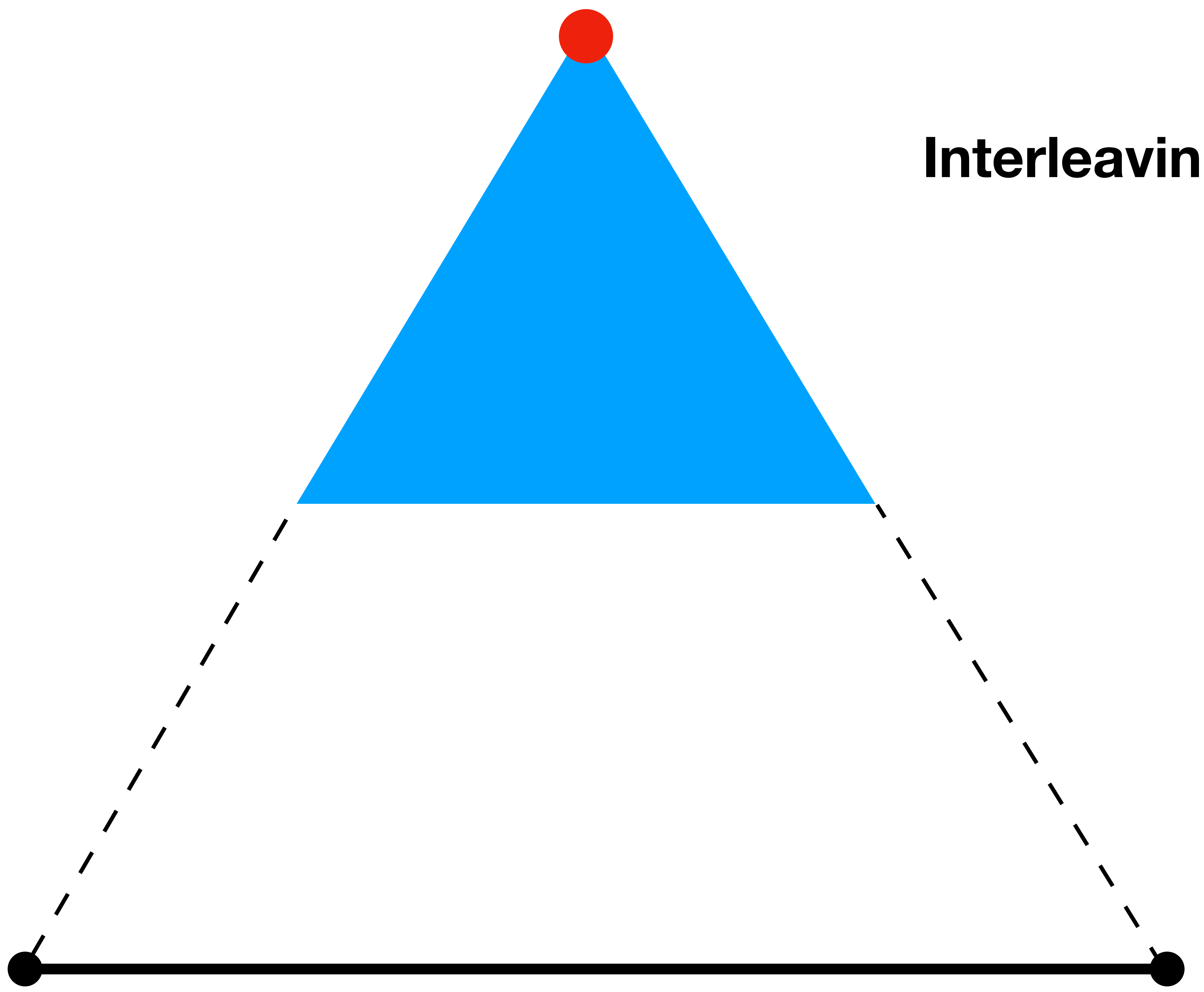
```
> (== 'cat 'cat)  (nevero 'cat)
```

```
>(disj (nevero ‘cat) (== ‘cat ‘cat))

>        (== ‘cat ‘cat)  (nevero ‘cat)
```

```
>(disj (nevero ‘cat) (== ‘cat ‘cat))

>         (== ‘cat ‘cat)  (nevero ‘cat)
```

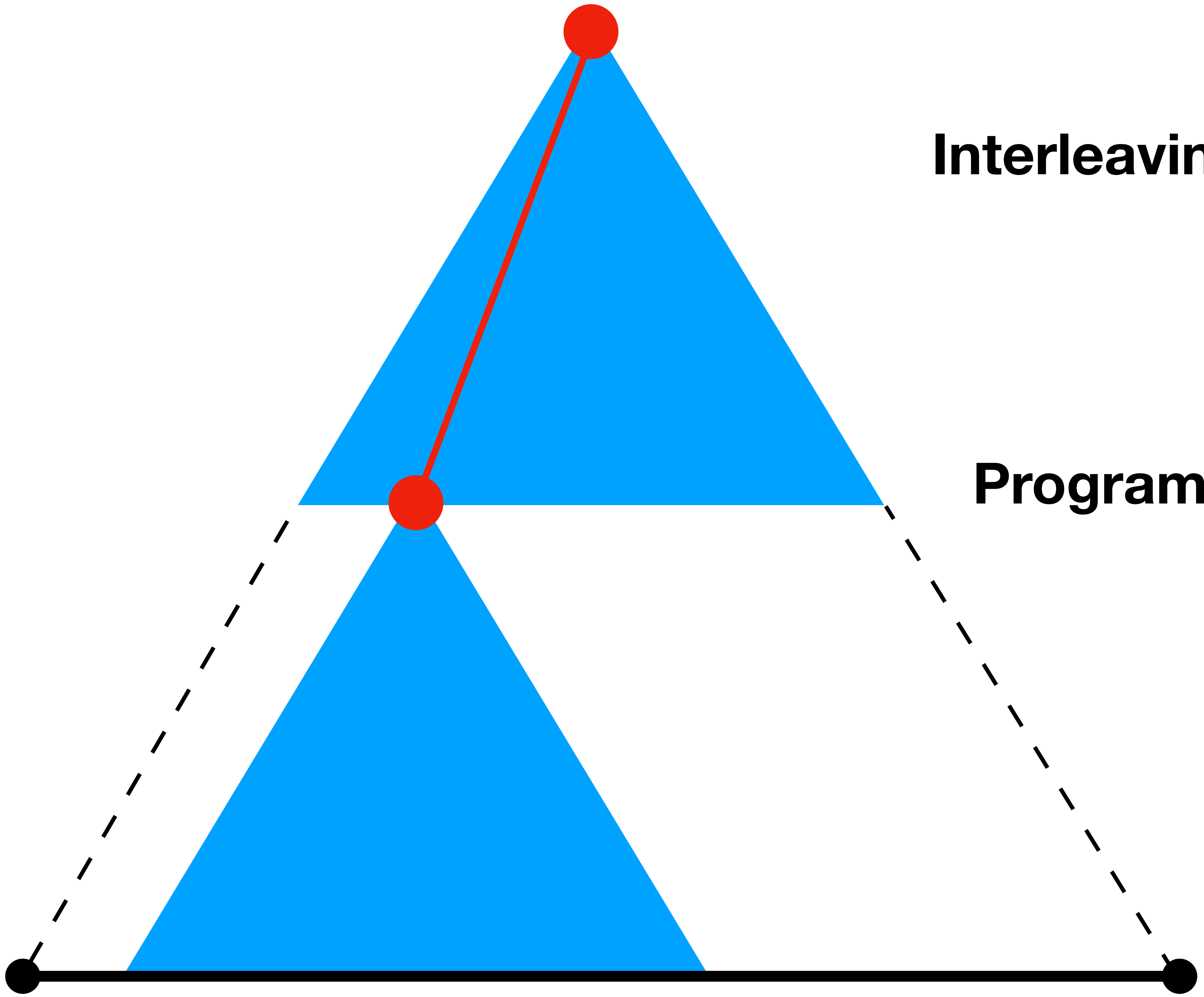
(Rozplokhas et al. 2019)

DOESN'T ALONE FIX A PARTICULAR SEARCH

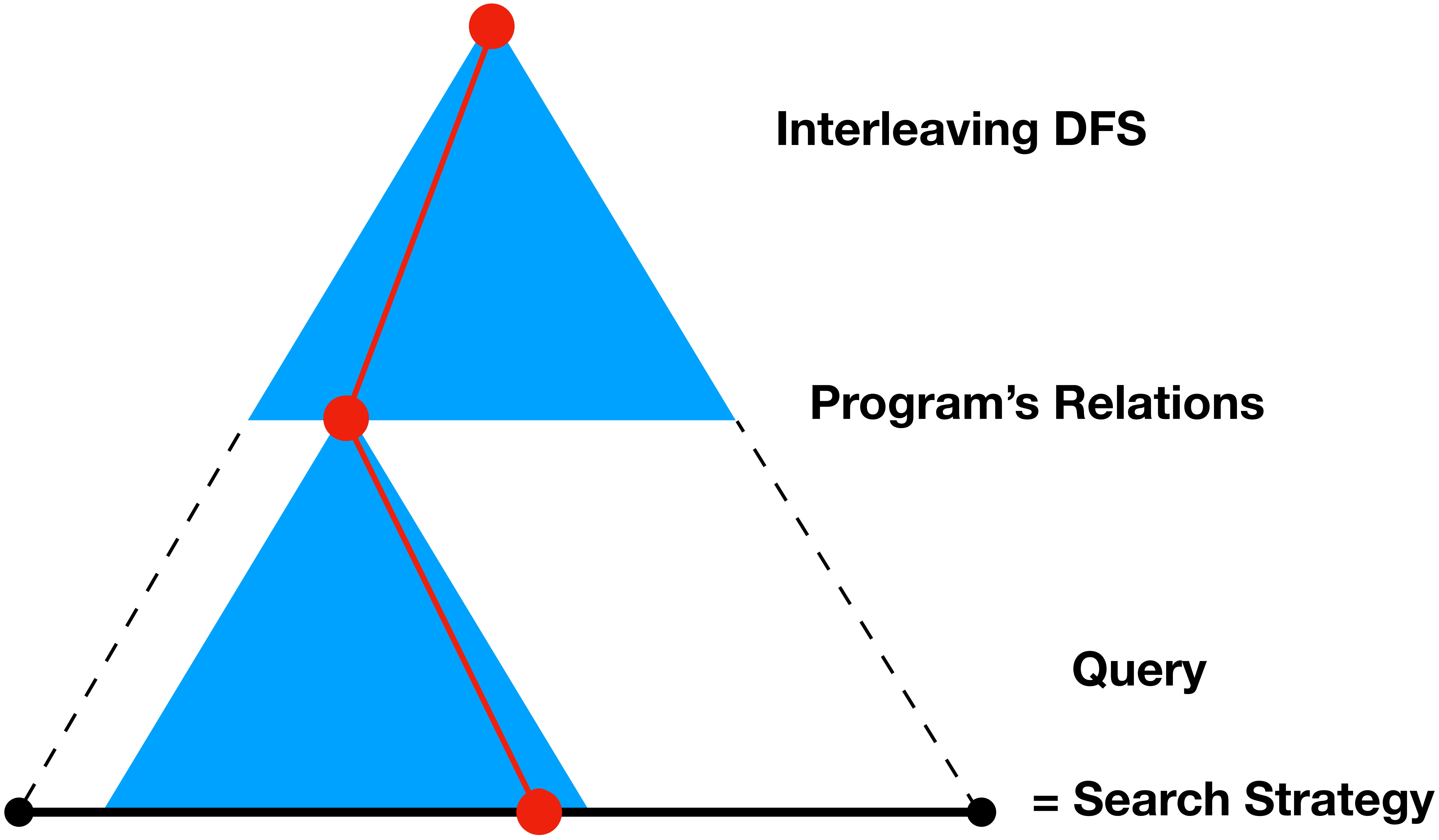


Interleaving DFS

DLS ’16, ICLP-DS ‘17



DLS ’16, ICLP-DS ‘17



DLS '16, ICLP-DS '17

miniKanren

Macros + Functions

microKanren Proliferates

Macros | Functions

Functions

Javascript

Functions

Javascript

Python

Functions

Javascript

Python

Functions

Ruby

Javascript

Python

Functions

Ruby

PHP

Java

Javascript

Python

Functions

Ruby

PHP

Java

Javascript

Python

Functions

Ruby

PHP

Erlang

Java

Javascript

Python

Prolog

Functions

Ruby

PHP

Erlang

Java

Javascript

Python

Prolog

Functions

Lua

Ruby

PHP

Erlang

Smalltalk Nu Haskell

Clojure LFE

Java miniKanren

Shen Javascript

Python

Rust

F# Prolog Scheme

Dylan

Lua

Scala

Ruby

Moxie PHP

Erlang

Purescript C# Extempore Elixir ML

Smalltalk Nu Julia Haskell

Idris Elm Clojure LFE Coffeescript

Java miniKanren

Shen Javascript

OCaML Python

Rust

F# Prolog Scheme

Dylan

Hy Lua

Scala

Groovy Ruby Pony

Moxie PHP

Erlang

Purescript C# Extempore Elixir ML

Smalltalk Nu Julia Haskell

Idris Elm Clojure LFE Coffeescript

Java miniKanren

Shen

F# Rust

Over **150** implementations
in **50** languages
(see miniKanren.org)

Groovy Moxie PHP Pony

Purescript C# Erlang Extempore Elixir ML

```
(define (var x) x)
(define (var? x) (number? x))

(define (find u s)
  (let ((pr (and (var? u) (assv u s))))
    (if pr (find (cdr pr) s) u)))

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

(define (occurs? x u s)
  (cond
    ((var? u)
    ((pair? u)

    (else #f)

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

(define ((== u v) s/c)
  (let ((s (car s/c)))
    (let ((s (unify (find u s) (find v s) s)))
      (if s (list `(,s . ,(cdr s/c))) `()))))
```

Equality

```
(define ((call/fresh f) s/c)
  (let ((c (cdr s/c)))
    ((f (var c)) `(,(car s/c) . ,(+ c 1)))))

(define-syntax-rule (define-relation (defname . args) g)
  (define ((defname . args) s/c) (delay/name (g s/c))))

(define (pull $) (if (promise? $) (pull (force $)) $))

(define (take n $)
  (cond
    ((null? $) '())
    ((and n (zero? (- n 1))) (list (car $)))
    (else (cons (car $)
                                                  ))

(define
  (take

(define ((disj g1 g2) s/c) ($append (g1 s/c) (g2 s/c)))
(define ((conj g1 g2) s/c) ($append-map (g1 s/c) g2))

(define ($append $1 $2)
  (cond
    ((null? $1) $2)
    ((promise? $1) (delay/name ($append $2 (force $1))))
    (else (cons (car $1) ($append (cdr $1) $2)))))

(define ($append-map $ g)
  (cond
    ((null? $) `())
    ((promise? $) (delay/name ($append-map (force $) g)))
    (else ($append (g (car $)) ($append-map (cdr $) g)))))
```

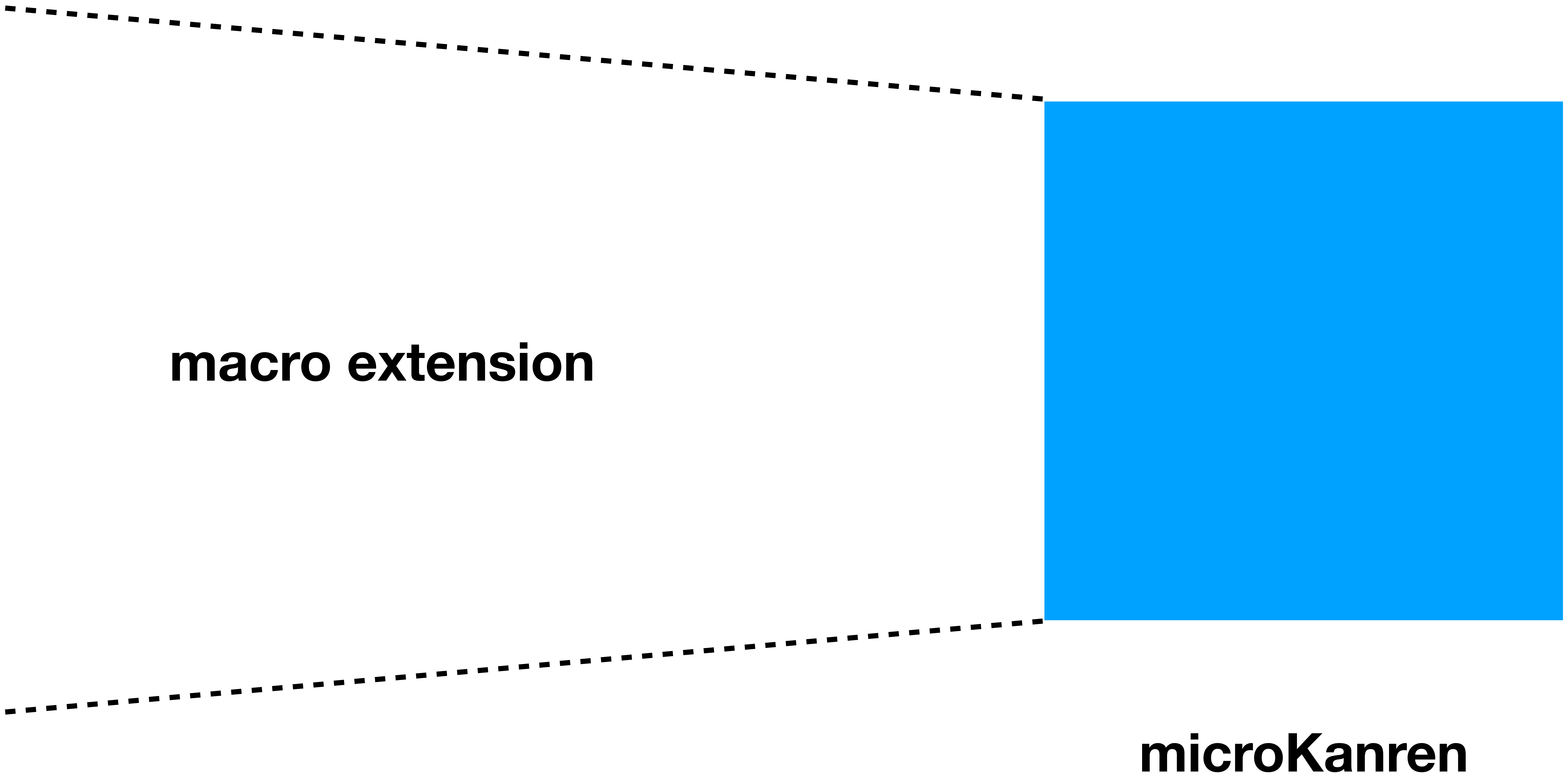
Control

```
(define ((call/fresh f) s/c)
  (let ((c (cdr s/c)))
    ((f (var c)) `(,(car s/c) . ,(+ c 1)))))

(define-syntax-rule (define-relation (defname . args) g)
  (define ((defname . args) s/c) (delay/name (g s/c))))

(define (pull $) (if (promise? $) (pull (force $)) $))

(define (take n $)
  (cond
    ((null? $) '())
    ((and n (zero? (- n 1))) (list (car $)))
    (else (cons (car $)
            (take (and n (- n 1)) (pull (cdr $)))))))

(define (call/initial-state n g)
  (take n (pull (g '(() . 0)))))

(define ((disj g1 g2) s/c) ($append (g1 s/c) (g2 s/c)))
(define ((conj g1 g2) s/c) ($append-map (g1 s/c) g2))

(define ($append $1 $2)
  (cond
    ((null? $1) $2)
    ((promise? $1) (delay/name ($append $2 (force $1))))
    (else (cons (car $1) ($append (cdr $1) $2)))))

(define ($append-map $ g)
  (cond
    ((null? $) `())
    ((promise? $) (delay/name ($append-map (force $) g)))
    (else ($append (g (car $)) ($append-map (cdr $) g)))))
```

Roadmap

- miniKanren, briefly
- a small kernel logic programming language
- miniKanren languages are syntactic extensions
- generalizing to constraints
- interrelated semantics
- parameterized by their constraint systems
- constraint system framework
- bolsters the development of useful tools and aids in solving important tasks

miniKanrens as Syntactic Extensions



```scheme
(define-syntax disj+
  (syntax-rules ()
    ((_ g) g)
    ((_ g0 g ...) (disj g0 (disj+ g ...)))))
```

```scheme
(define-syntax conj+
  (syntax-rules ()
    ((_ g) g)
    ((_ g0 g ...) (conj g0 (conj+ g ...)))))
```

```scheme
(define-syntax-rule (conde (g0 g ...) (g0* g* ...) ...)
  (disj+ (conj+ g0 g ...) (conj+ g0* g* ...) ...))
```

Roadmap

- miniKanren, briefly
- a small kernel logic programming language
- miniKanren languages are syntactic extensions
- generalizing to constraints
- interrelated semantics
- parameterized by their constraint systems
- constraint system framework
- bolsters the development of useful tools and aids in solving important tasks

```scheme
(define (var x) x)
(define (var? x) (number? x))

(define (find u s)
  (let ((pr (and (var? u) (assv u s))))
    (if pr (find (cdr pr) s) u)))

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

(define (occurs? x u s)
  (cond
    ((var? u) (eqv? x u))
    ((pair? u) (or (occurs? x (find (car u) s) s)
                   (occurs? x (find (cdr u) s) s)))
    (else #f)))

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

(define ((== u v) s/c)
  (let ((s (car s/c)))
    (let ((s (unify (find u s) (find v s) s)))
      (if s (list `(,s . ,(cdr s/c))) `()))))
```

```scheme
(define ((call/fresh f) s/c)
  (let ((c (cdr s/c)))
    ((f (var c)) `(,(car s/c) . ,(+ c 1)))))

(define-syntax-rule (define-relation (defname . args) g)
  (define ((defname . args) s/c) (delay/name (g s/c))))

(define (pull $) (if (promise? $) (pull (force $)) $))

(define (take n $)
  (cond
    ((null? $) '())
    ((and n (zero? (- n 1))) (list (car $)))
    (else (cons (car $)
            (take (and n (- n 1)) (pull (cdr $)))))))

(define (call/initial-state n g)
  (take n (pull (g '(() . 0)))))

(define ((disj g1 g2) s/c) ($append (g1 s/c) (g2 s/c)))
(define ((conj g1 g2) s/c) ($append-map (g1 s/c) g2))

(define ($append $1 $2)
  (cond
    ((null? $1) $2)
    ((promise? $1) (delay/name ($append $2 (force $1))))
    (else (cons (car $1) ($append (cdr $1) $2)))))

(define ($append-map $ g)
  (cond
    ((null? $) `())
    ((promise? $) (delay/name ($append-map (force $) g)))
    (else ($append (g (car $)) ($append-map (cdr $) g)))))
```

```
(define (var x) x)
(define (var? x) (number? x))

(define (find u s)
  (let ((pr (and (var? u) (assv u s))))
    (if pr (find (cdr pr) s) u)))

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

(define (occurs? x u s)
  (cond
    ((var? u)
    ((pair? u)

    (else #f))

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

(define ((== u v) s/c)
  (let ((s (car s/c)))
    (let ((s (unify (find u s) (find v s) s)))
      (if s (list `(,s . ,(cdr s/c))) `()))))
```

Equality

```
(define ((call/fresh f) s/c)
  (let ((c (cdr s/c)))
    ((f (var c)) `(,(car s/c) . ,(+ c 1)))))

(define-syntax-rule (define-relation (defname . args) g)
  (define ((defname . args) s/c) (delay/name (g s/c))))

(define (pull $) (if (promise? $) (pull (force $)) $))

(define (take n $)
  (cond
    ((null? $) '())
    ((and n (zero? (- n 1))) (list (car $)))
    (else (cons (car $)
                                                ))

(define
  (take

(define ((disj g1 g2) s/c) ($append (g1 s/c) (g2 s/c)))
(define ((conj g1 g2) s/c) ($append-map (g1 s/c) g2))

(define ($append $1 $2)
  (cond
    ((null? $1) $2)
    ((promise? $1) (delay/name ($append $2 (force $1))))
    (else (cons (car $1) ($append (cdr $1) $2)))))

(define ($append-map $ g)
  (cond
    ((null? $) `())
    ((promise? $) (delay/name ($append-map (force $) g)))
    (else ($append (g (car $)) ($append-map (cdr $) g)))))
```

Control

χ

```
(define ((call/fresh f) s/c)
  (let ((c (cdr s/c)))
    ((f (var c)) `(,(car s/c) . ,(+ c 1)))))

(define-syntax-rule (define-relation (defname . args) g)
  (define ((defname . args) s/c) (delay/name (g s/c))))

(define (pull $) (if (promise? $) (pull (force $)) $))

(define (take n $)
  (cond
    ((null? $) '())
    ((and n (zero? (- n 1))) (list (car $)))
    (else (cons (car $)
                                                       ))

(define
  (take

(define ((disj g1 g2) s/c) ($append (g1 s/c) (g2 s/c)))
(define ((conj g1 g2) s/c) ($append-map (g1 s/c) g2))

(define ($append $1 $2)
  (cond
    ((null? $1) $2)
    ((promise? $1) (delay/name ($append $2 (force $1))))
    (else (cons (car $1) ($append (cdr $1) $2)))))

(define ($append-map $ g)
  (cond
    ((null? $) `())
    ((promise? $) (delay/name ($append-map (force $) g)))
    (else ($append (g (car $)) ($append-map (cdr $) g)))))
```

Control

Constraints Add More Problems

Constraints, generically

```
(define (== u v)
  (λ (S/c)
    (let ((S (ext-S (car S/c) '== '(u v))))
      (if (invalid? S)
          '()
          (list `(,S . ,(cdr S/c)))))))
```

Constraints, generically

```
(define (== u v)
  (λ (S/c)
    (let ((S (ext-S (car S/c) '== '(u v))))
      (if (invalid? S)
          '()
          (list `(,S . ,(cdr S/c)))))))
```

Constraints, generically

```
(define (listo u)
  (λ (S/c)
    (let ((S (ext-S (car S/c) 'listo '(u))))
      (if (invalid? S)
          '()
          (list `(,S . ,(cdr S/c)))))))

(define (== u v)
  (λ (S/c)
    (let ((S (ext-S (car S/c) '== '(u v))))
      (if (invalid? S)
          '()
          (list `(,S . ,(cdr S/c)))))))

(define (=/= u v)
  (λ (S/c)
    (let ((S (ext-S (car S/c) '=/= '(u v))))
      (if (invalid? S)
          '()
          (list `(,S . ,(cdr S/c)))))))
```

```
(define (listo u  )
  (λ (S/c)
    (let ((S (ext-S (car S/c) 'listo '(u  ))))
      (if (invalid? S)
          '()
          (list `(,S . ,(cdr S/c)))))))

(define (==    u v)
  (λ (S/c)
    (let ((S (ext-S (car S/c) '==    '(u v))))
      (if (invalid? S)
          '()
          (list `(,S . ,(cdr S/c)))))))

(define (=/=   u v)
  (λ (S/c)
    (let ((S (ext-S (car S/c) '=/=   '(u v))))
      (if (invalid? S)
          '()
          (list `(,S . ,(cdr S/c)))))))
```

```
(define (listo u v)
  (λ (S/c)
    (let ((S (ext-S (car S/c) 'listo '(u v))))
      (if (invalid? S)
          '()
          (list `(,S . ,(cdr S/c)))))))
```

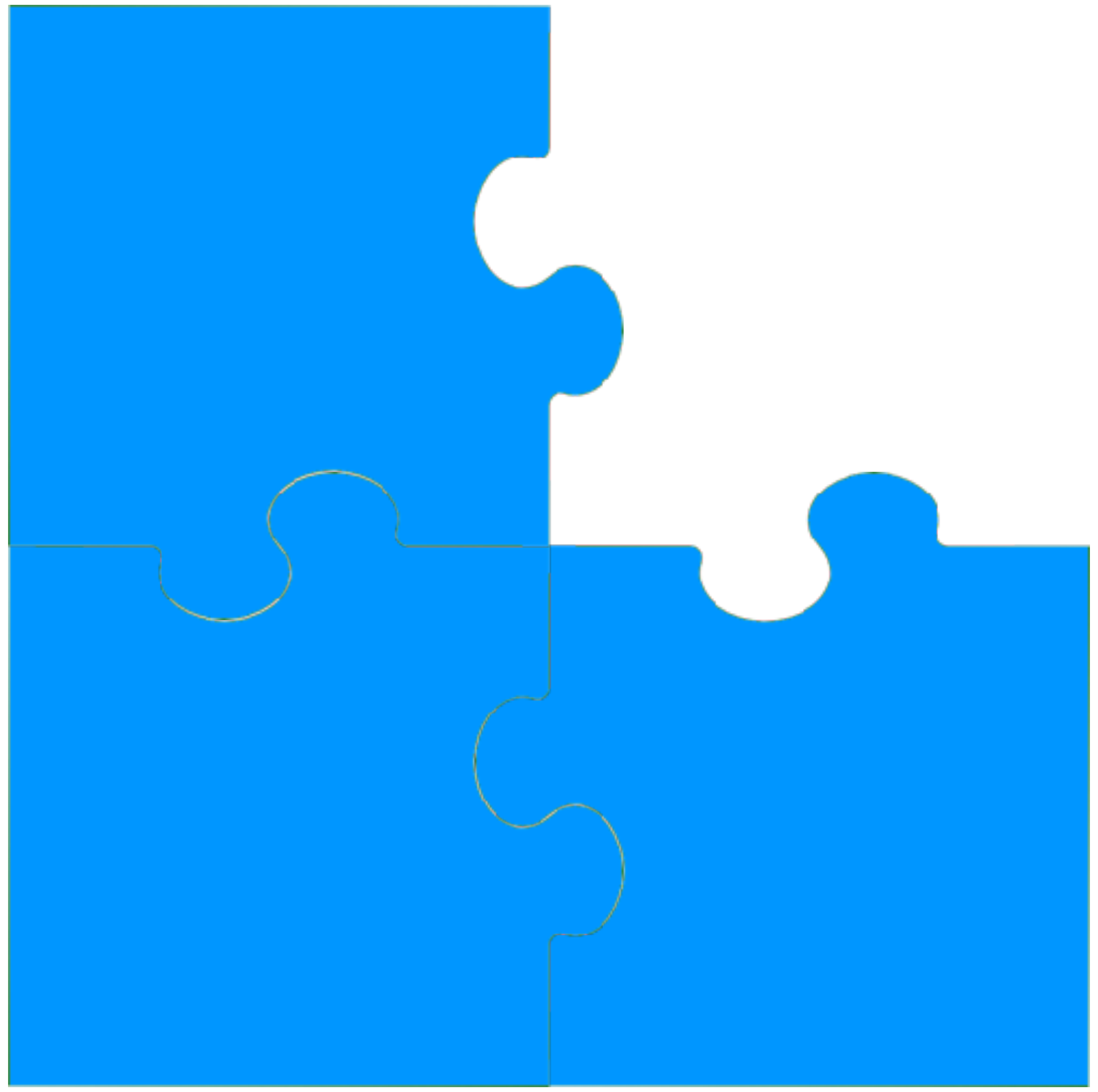
```
(define (listo u v)
  (λ (S/c)
    (let ((S (ext-S (car S/c) 'listo '(u v))))
      (if (invalid? S)
          '()
          (list `(,S . ,(cdr S/c)))))))
```

Factor out common portions of carefully-considered implementations

Roadmap

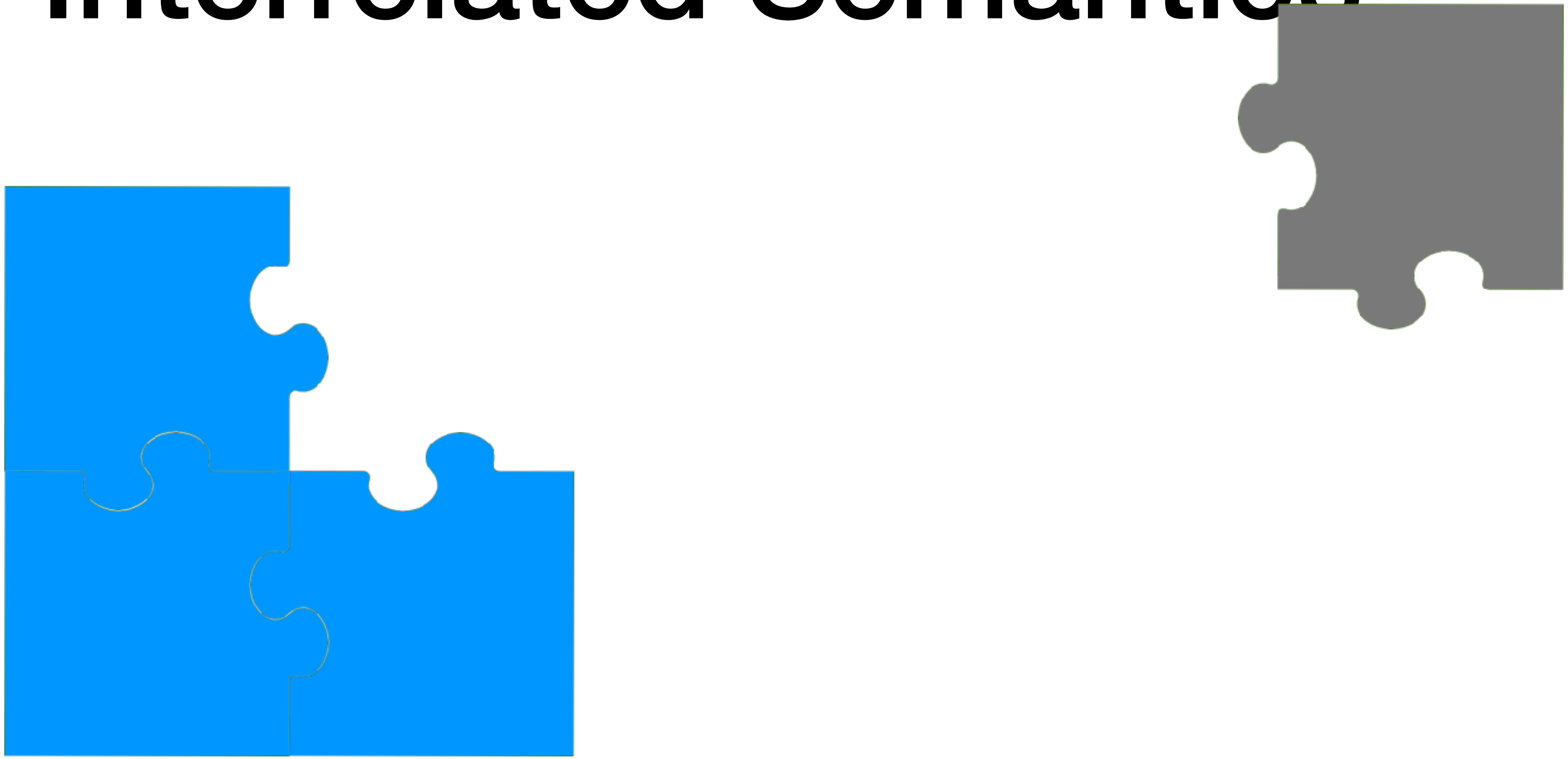
- miniKanren, briefly
- a small kernel logic programming language
- miniKanren languages are syntactic extensions
- generalizing to constraints
- interrelated semantics
- parameterized by their constraint systems
- constraint system framework
- bolsters the development of useful tools and aids in solving important tasks

Interrelated Semantics



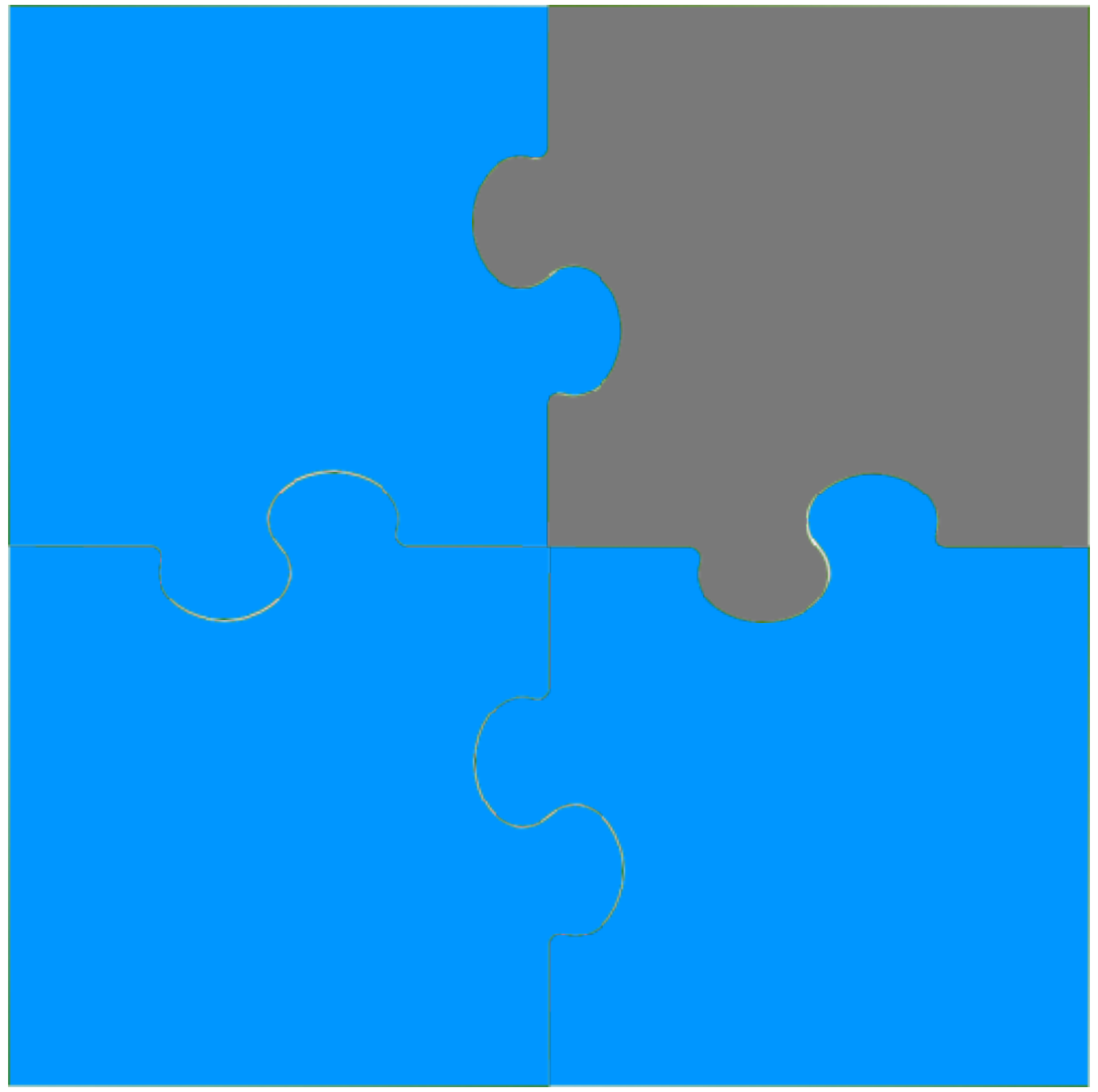
Parameterized Lang Class CLP($\cdot$)

Interrelated Semantics



Instantiated with a Constraint Domain $\mathcal{X}$

Interrelated Semantics



A Language CLP($\mathcal{X}$)

Roadmap

- miniKanren, briefly
- a small kernel logic programming language
- miniKanren languages are syntactic extensions
- generalizing to constraints
- interrelated semantics
- parameterized by their constraint systems
- constraint system framework
- bolsters the development of useful tools and aids in solving important tasks

CLP Scheme & Constraint Systems $\mathcal{X}$

CLP Scheme & Constraint Systems $\mathcal{X}$

- A **signature Σ** for the constraint domain
- The **constraints**, a class of FO Σ-formulas
- A **Σ-theory**, the constraints' logical semantics
- A **Σ-structure**, the constraints' intended interpretation — algebraic semantics
- A function **solve** from constraints to {T,F,?} — operational semantics
- The theory, structure, and solver **agree**
- **A bit more** =, ρ, etc.

How Did We Fit In? (Restrictions)

- total solver
- symbolic constraints
- negative constraints
- n-independent

“miniKanren Constraint” Systems

- disequalities
- sort constraints
- subterm “discontainment”
- “shape” constraints

SW ’16, WFLP '15

“miniKanren Constraint” Systems

- disequalities
- sort constraints
- subterm “**dis**containment”
- “shape” constraints

SW ’16, WFLP '15

w

$$w \sqsubseteq x$$

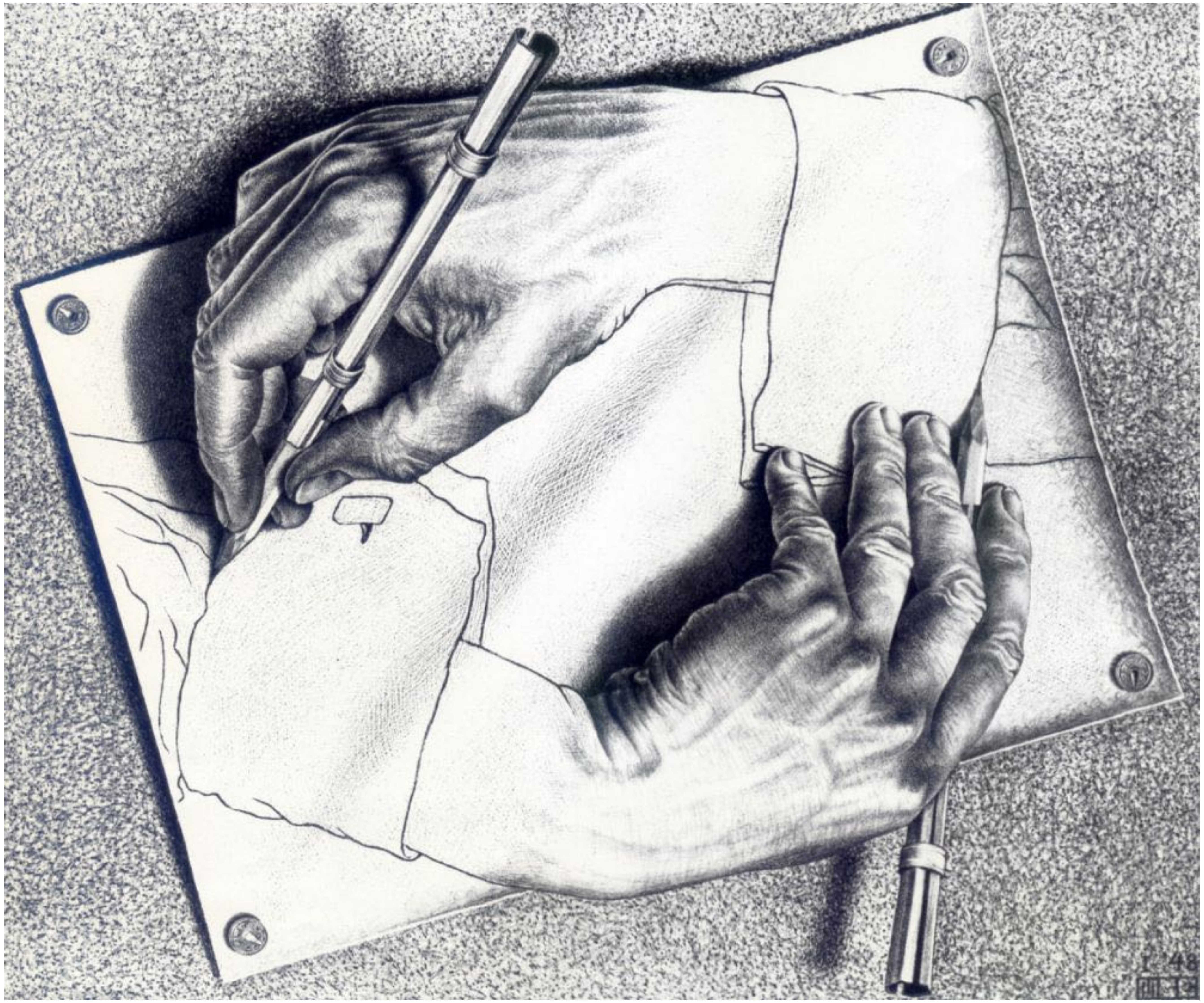
$$w \sqsubseteq x \sqsubseteq y$$

$$w \sqsubseteq x \sqsubseteq y \sqsubseteq z$$

$$\cdots w \sqsubseteq x \sqsubseteq y \sqsubseteq z \sqsubseteq \cdots$$

$$\begin{tikzcd}
 & x & \\
w \arrow[ur, phantom, "\sqsubseteq" sloped] & & y \arrow[ul, phantom, "\sqsupseteq" sloped] \\
 & z \arrow[ul, phantom, "\sqsubseteq" sloped] \arrow[ur, phantom, "\sqsubseteq" sloped] &
\end{tikzcd}$$

$$
\begin{tikzcd}
 & x \arrow[dl, equal] \arrow[dr, equal] & \\
w \arrow[dr, equal] & & y \arrow[dl, equal] \\
 & z &
\end{tikzcd}
$$

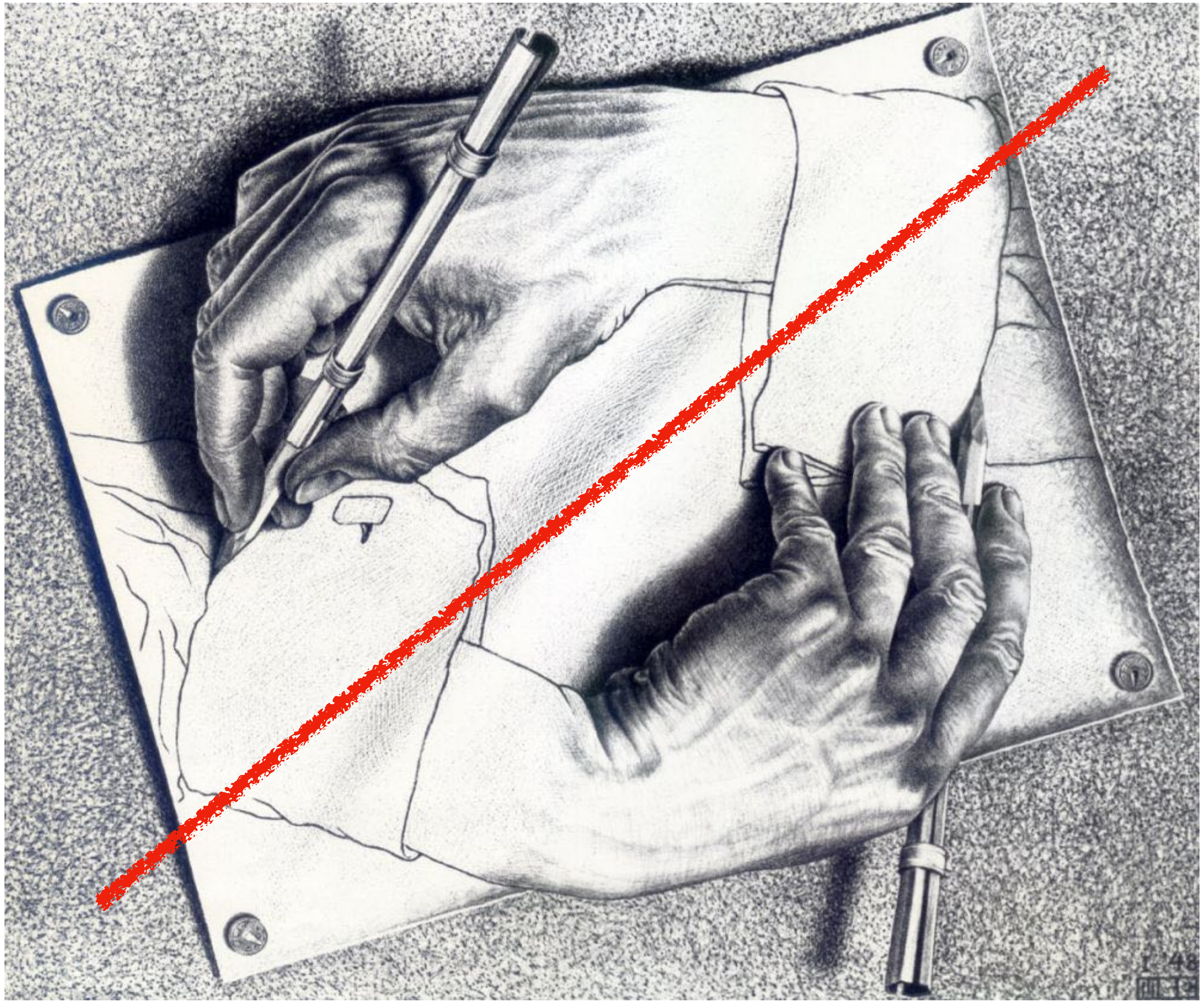


Independence of Negative Constraints

P constraints
Q *negatable* constraints

$\{p_1,\ldots,p_n\} \vDash \{q_1,\ldots,q_m\}$ implies $\{p_1,\ldots,p_n\} \vDash q_i$ for some $1 < i \leq m$

Independence of Negative Constraints



Heterogenous Collections of *equations*

$$P \times Q \times R \times S \times \vdots$$

Roadmap

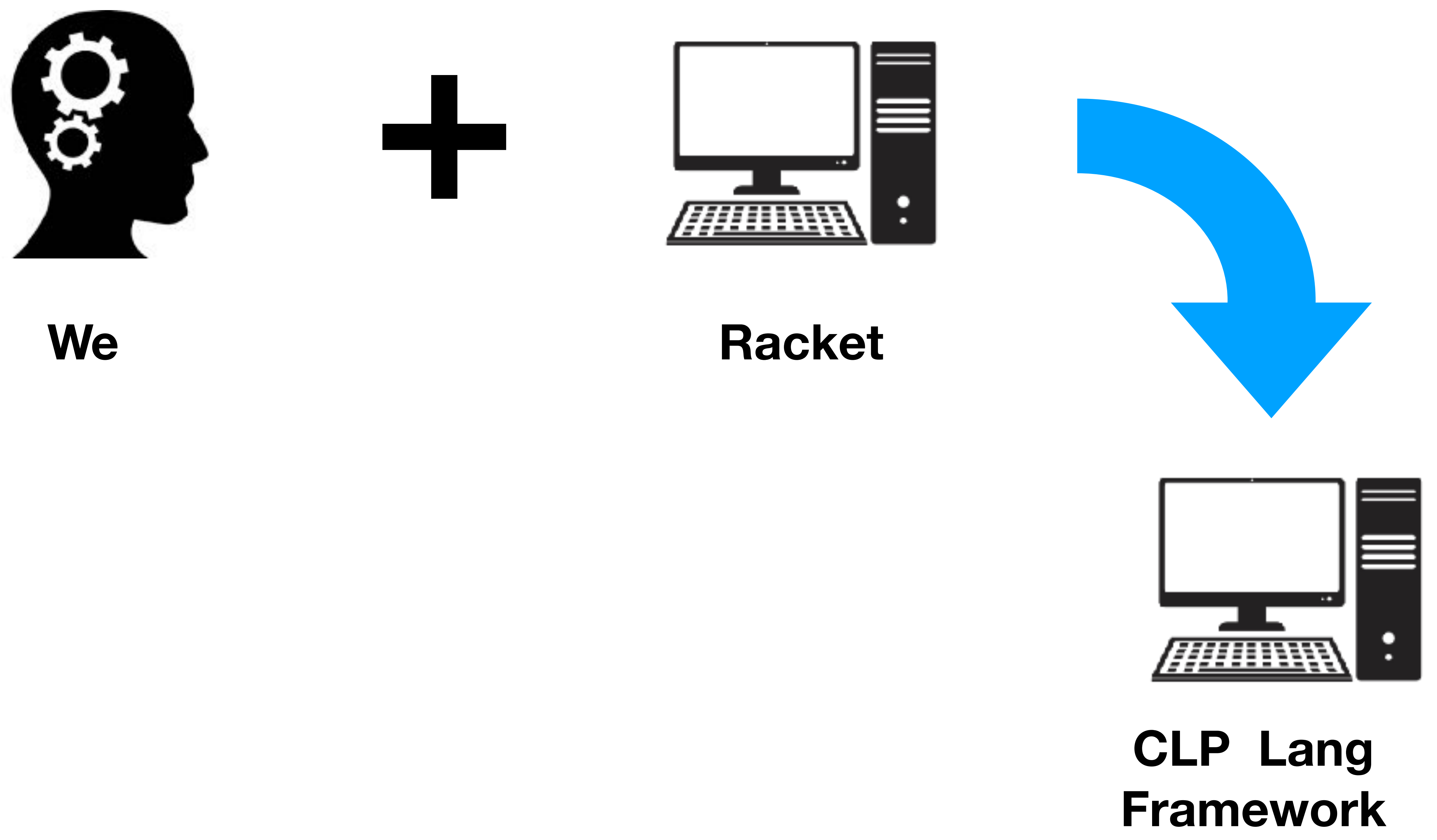
- miniKanren, briefly
- a small kernel logic programming language
- miniKanren languages are syntactic extensions
- generalizing to constraints
- interrelated semantics
- parameterized by their constraint systems
- constraint system framework
- bolsters the development of useful tools and aids in solving important tasks

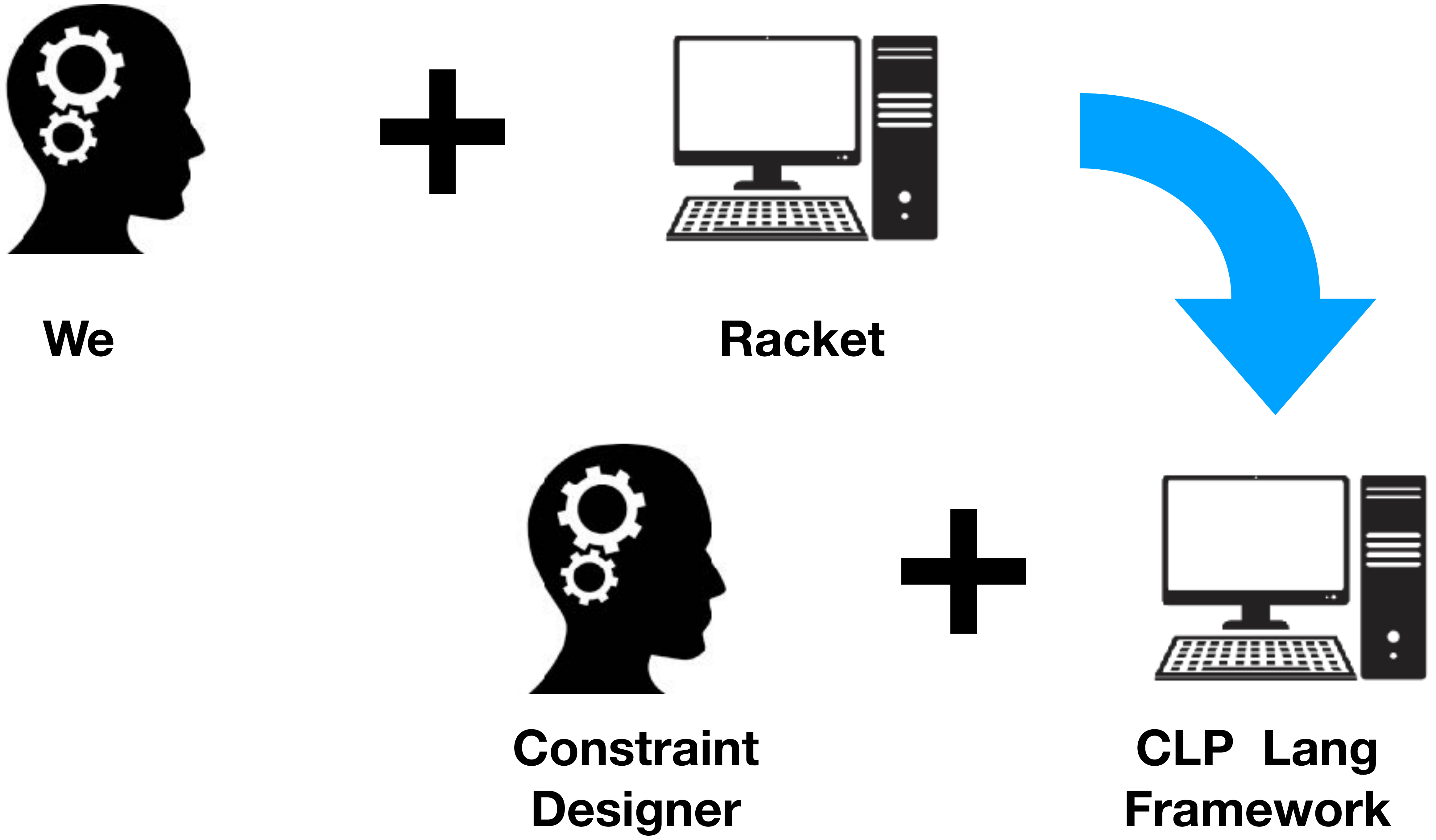
We

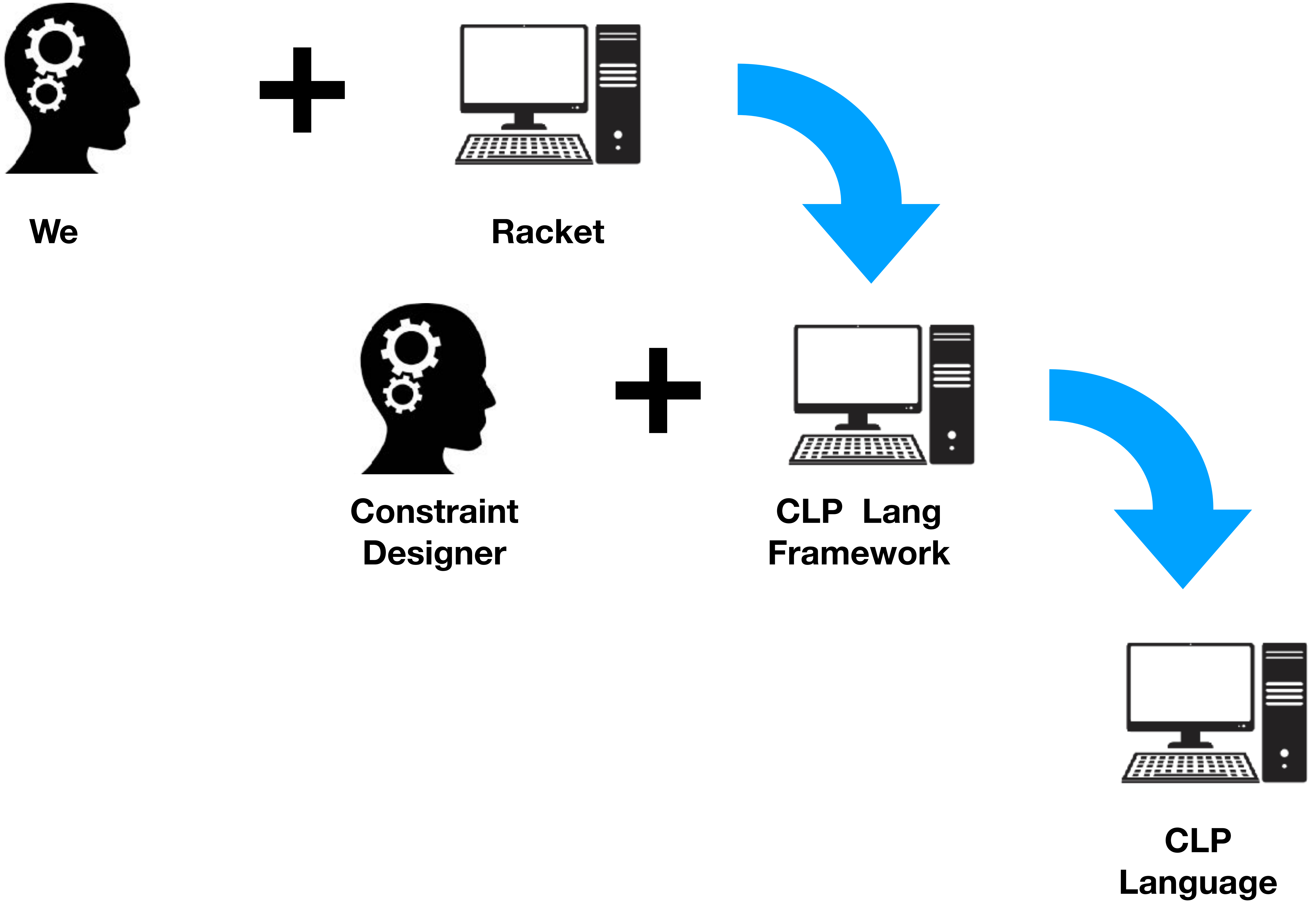
+

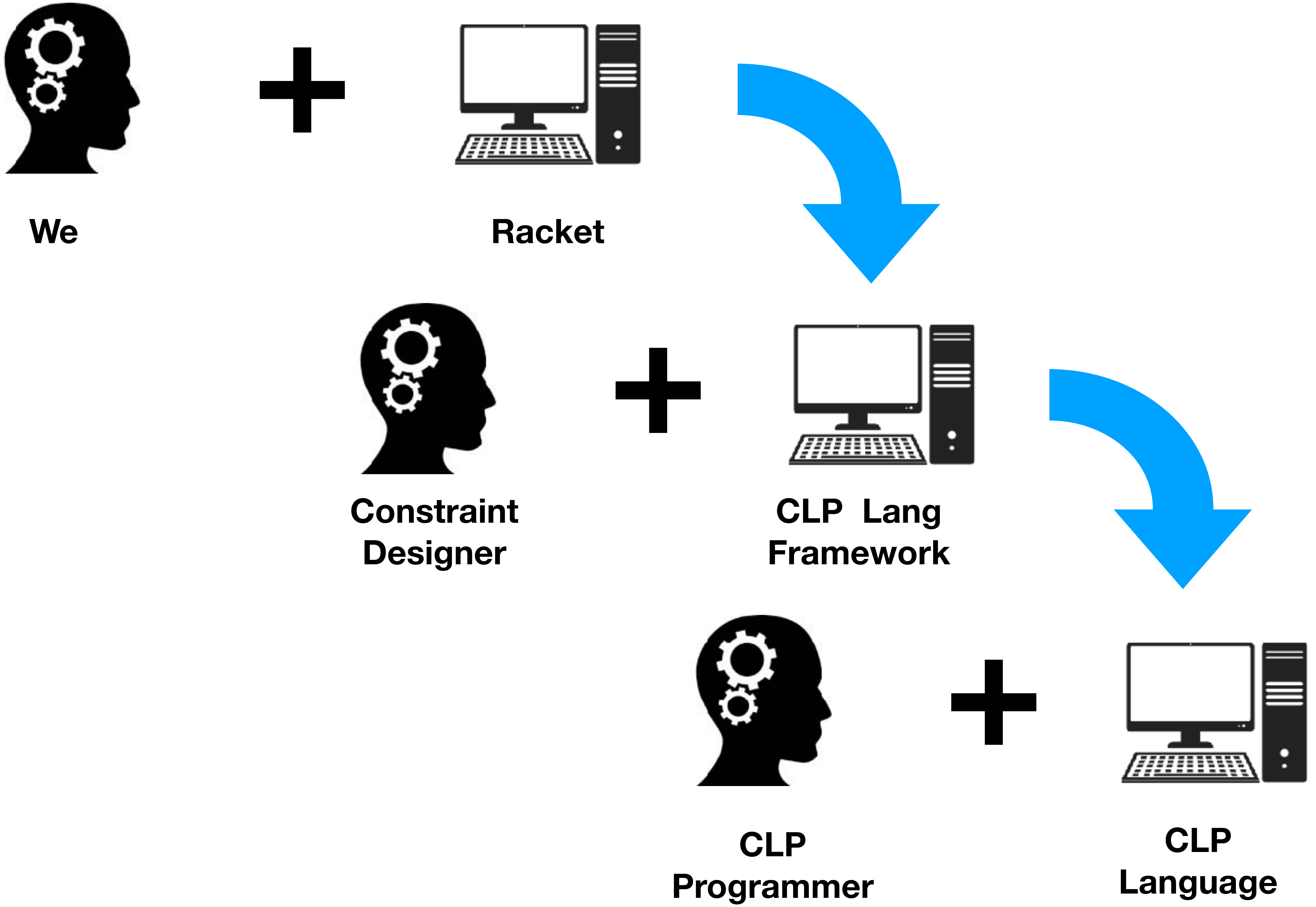
We

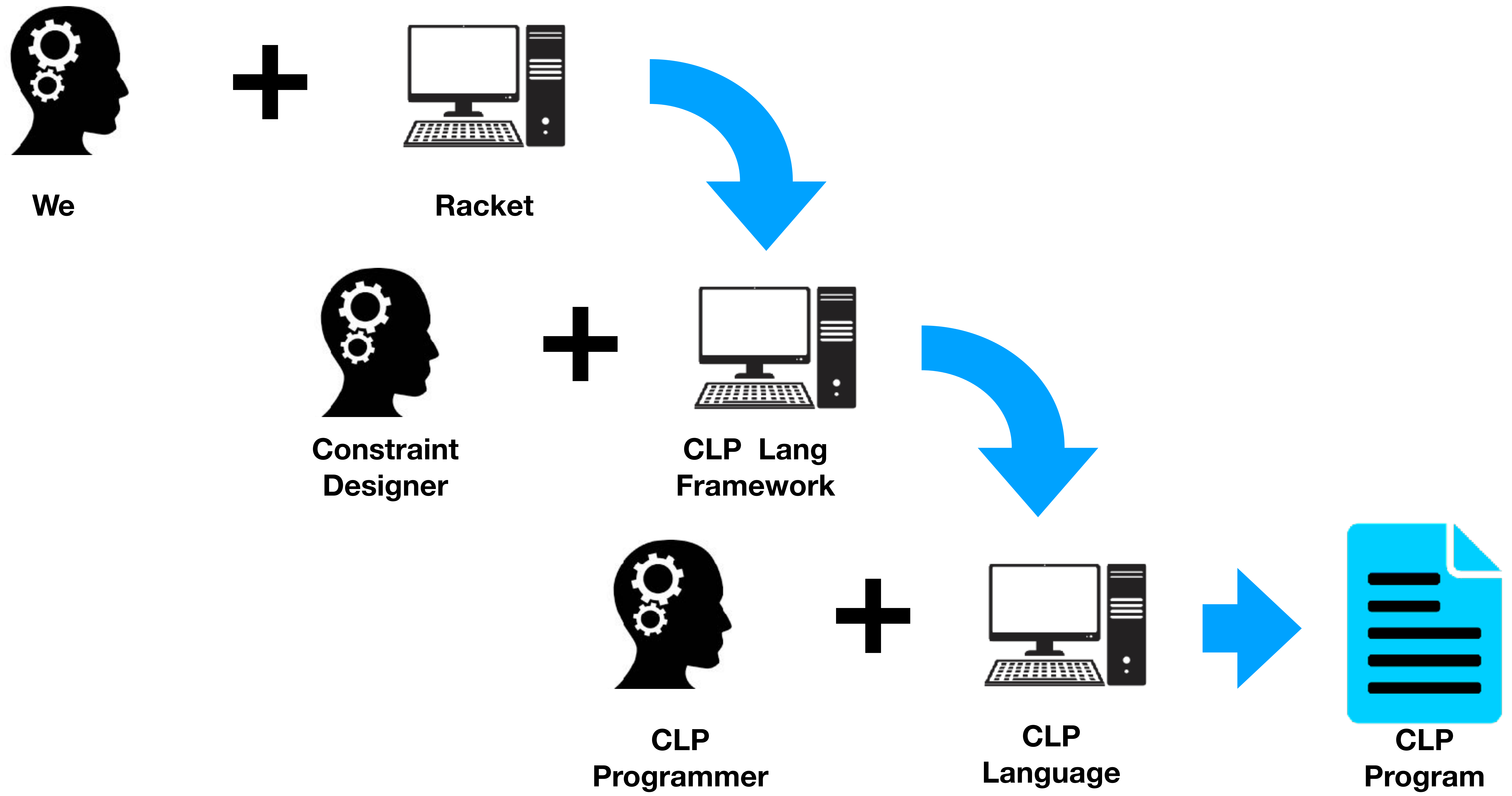
Racket


We
Racket
CLP Lang
Framework


We
Racket
Constraint
Designer
CLP Lang
Framework


We
Racket
Constraint
Designer
CLP Lang
Framework
CLP
Language


We
Racket
Constraint
Designer
CLP Lang
Framework
CLP
Programmer
CLP
Language


We
Racket
Constraint
Designer
CLP Lang
Framework
CLP
Programmer
CLP
Language
CLP
Program

Framework Design

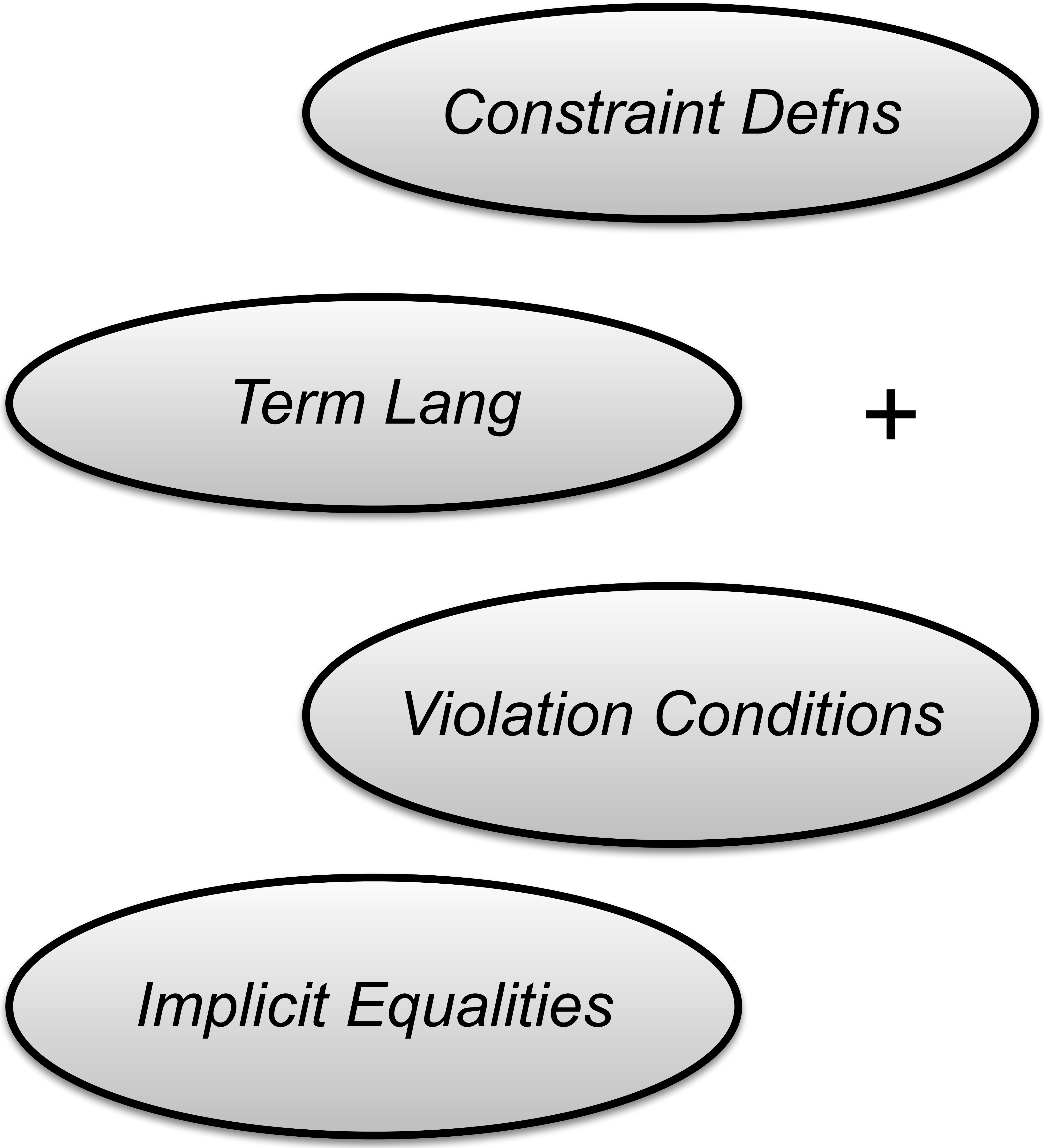
Constraint Defns

Framework Design

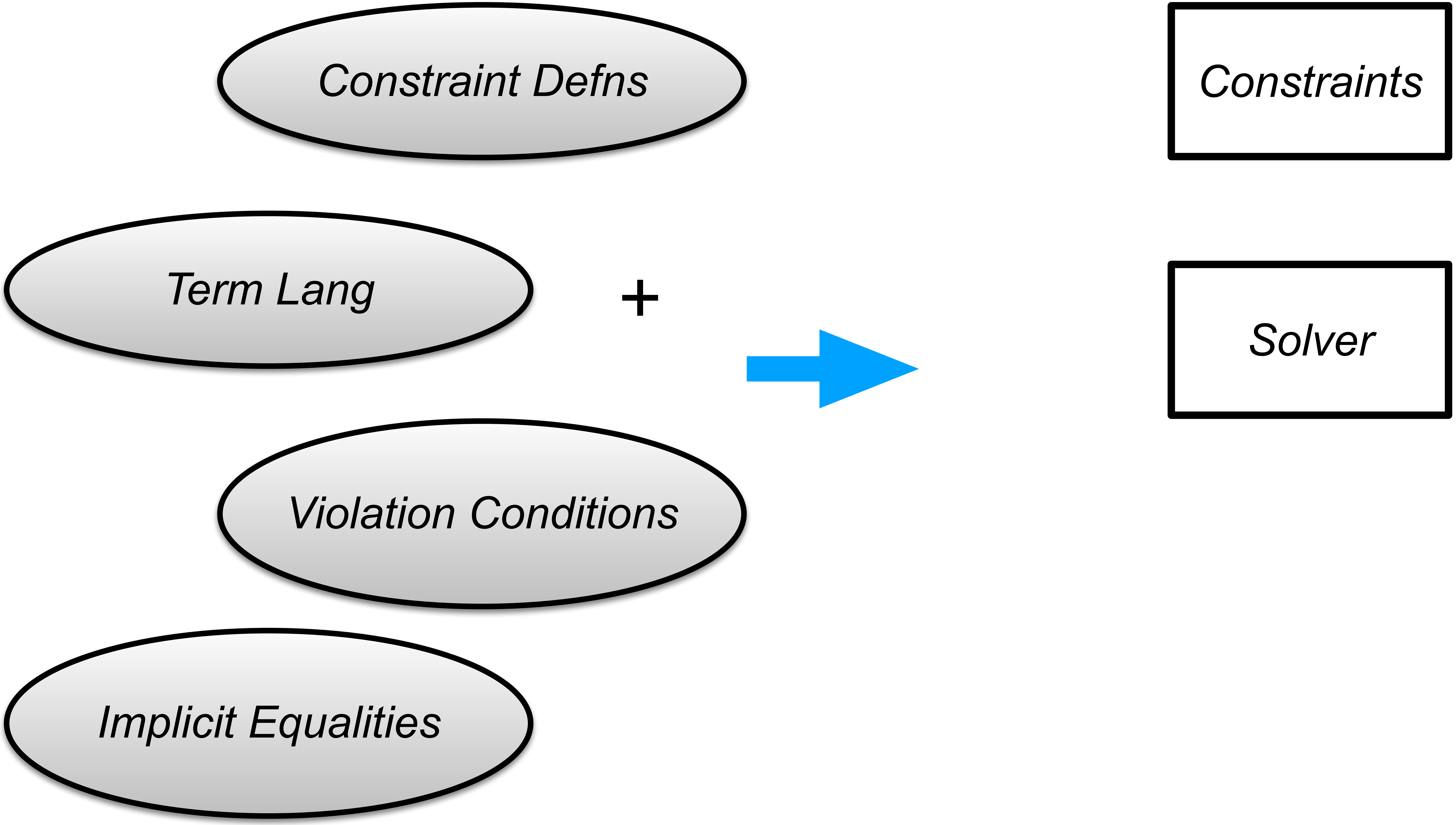
Constraint Defns

+

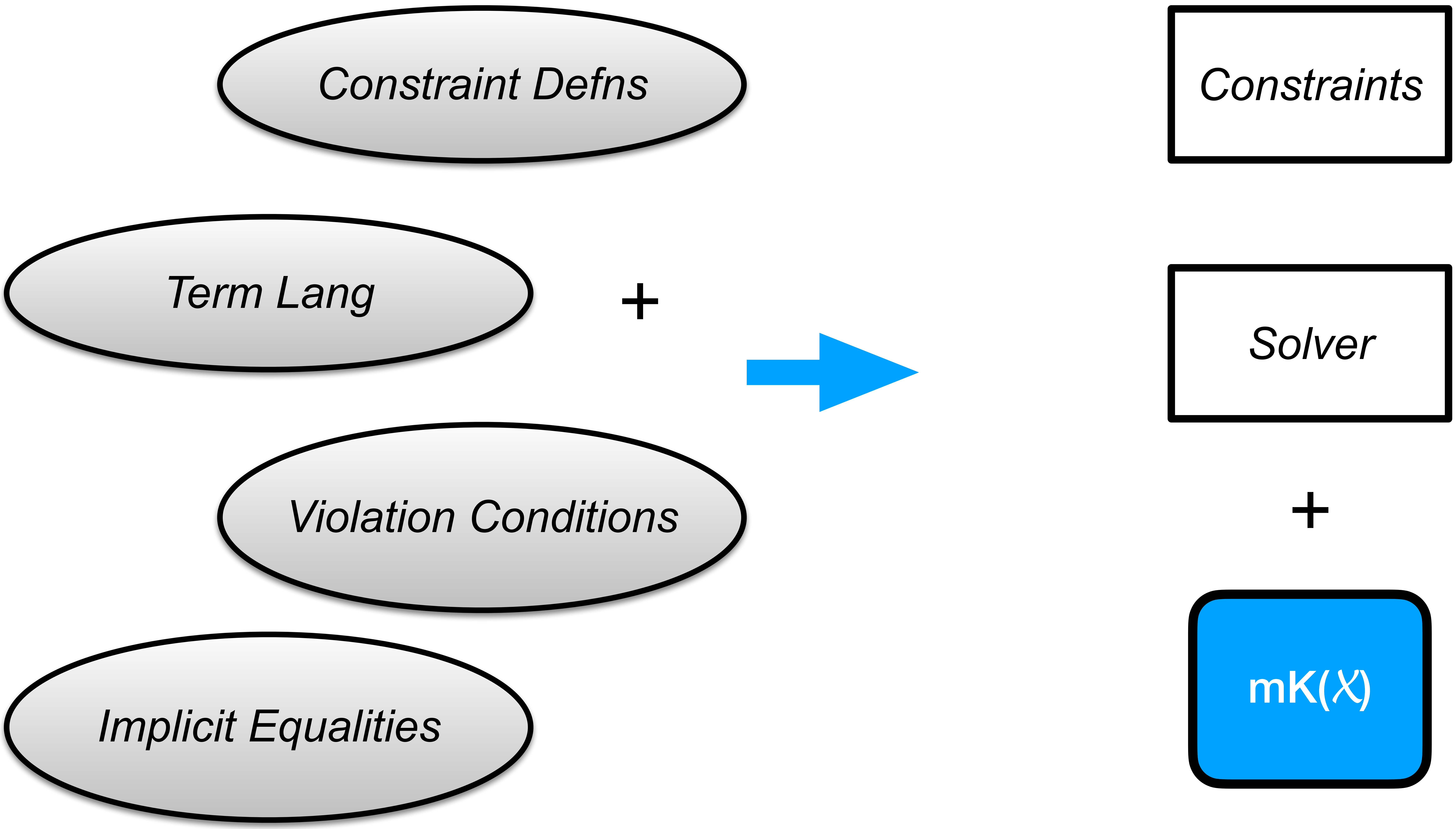
Framework Design



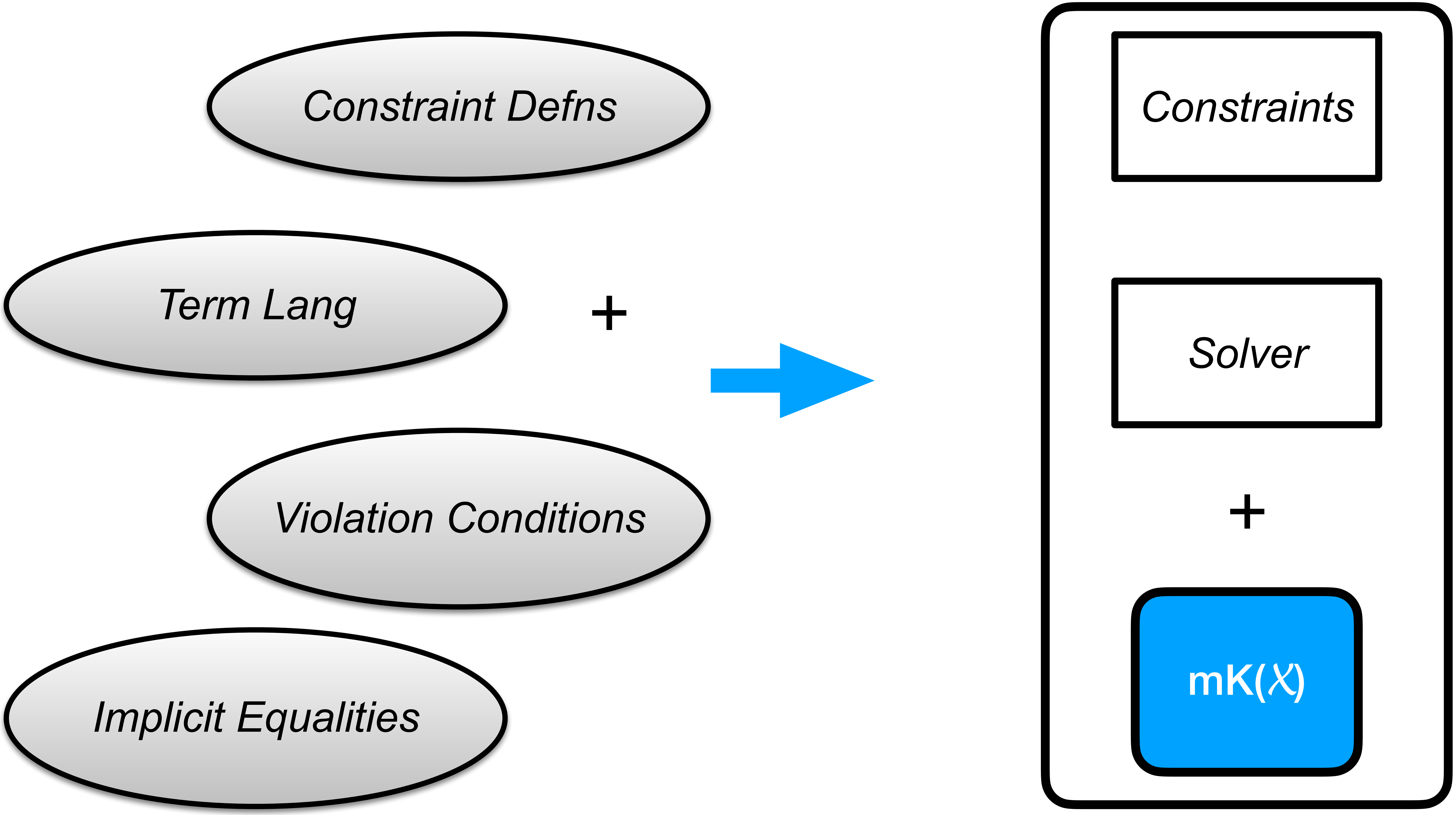
Framework Design



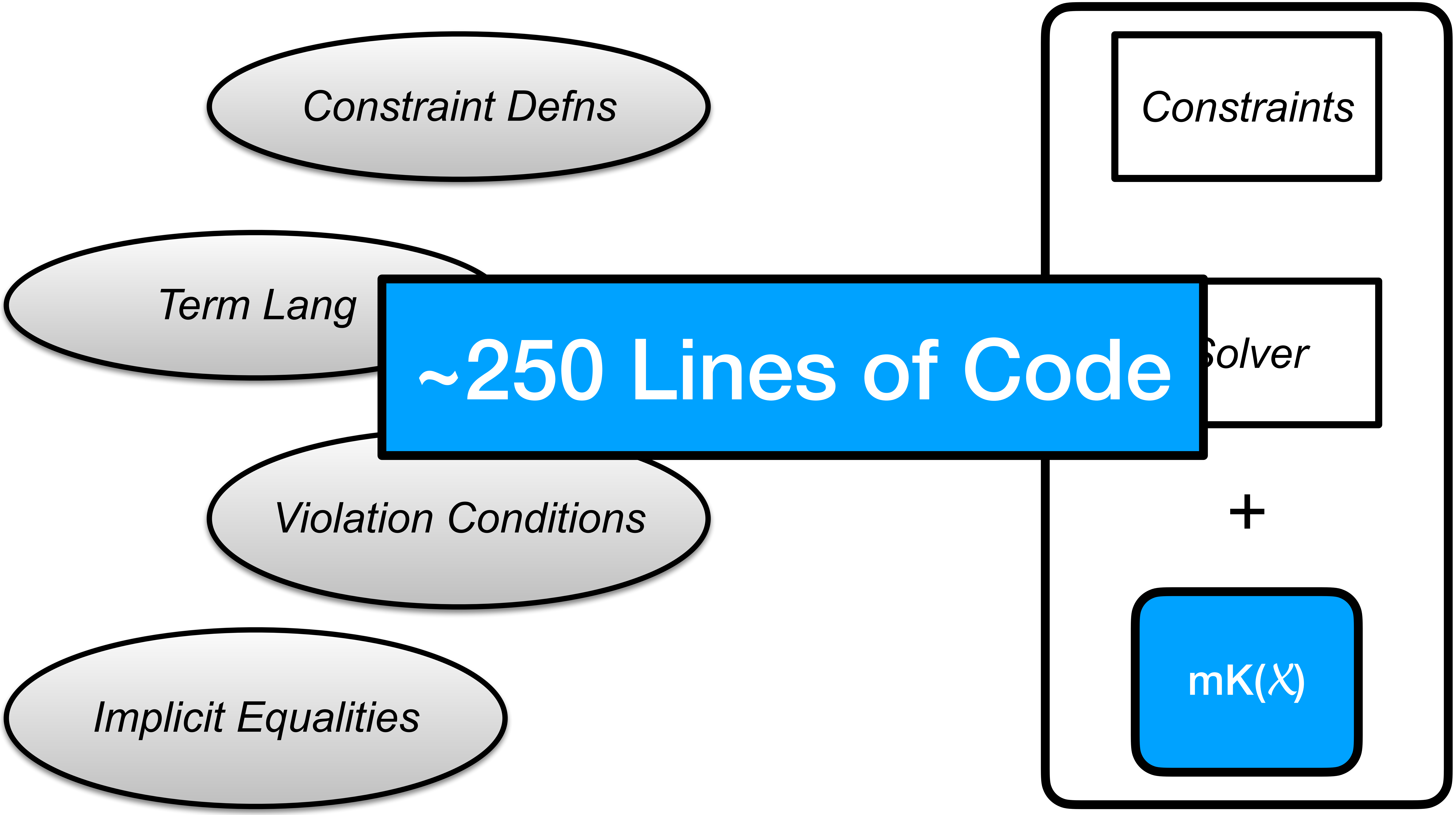
Framework Design



Framework Design



Framework Design



Solving, Generally

Solving, Generally

1. Solve explicit equality constraints

Solving, Generally

1. Solve explicit equality constraints
2. Sequentially solve any implicit equalities

Solving, Generally

1. Solve explicit equality constraints
2. Sequentially solve any implicit equalities
3. Check *n*-wise constraint violation conditions

Roadmap

- miniKanren, briefly
- a small kernel logic programming language
- miniKanren languages are syntactic extensions
- generalizing to constraints
- interrelated semantics
- parameterized by their constraint systems
- constraint system framework
- bolsters the development of useful tools and aids in solving important tasks

INTERPRETER
(RELATIONAL)

$$\frac{\lambda \notin_{\mathbf{env}} \rho}{\rho \vdash \lambda x.e \Downarrow \langle \lambda x.e\ \underline{\mathtt{in}}\ \rho \rangle}$$

$$\frac{\lambda \notin_{\textbf{env}} \rho}{\rho \vdash \lambda x.e \Downarrow \langle \lambda x.e\ \underline{\mathtt{in}}\ \rho \rangle}$$

```
(define-relation (not-in-envo x env)
  (conde
    [(== '() env)]
    [(fresh (y _ rest)
       (== `((,y ,_) . ,rest) env)
       (=/= y x)
       (not-in-envo x rest))]))
```

$$\frac{\lambda \notin_{\textbf{env}} \rho}{\rho \vdash \lambda x.e \Downarrow \langle \lambda x.e\ \underline{\mathtt{in}}\ \rho \rangle}$$

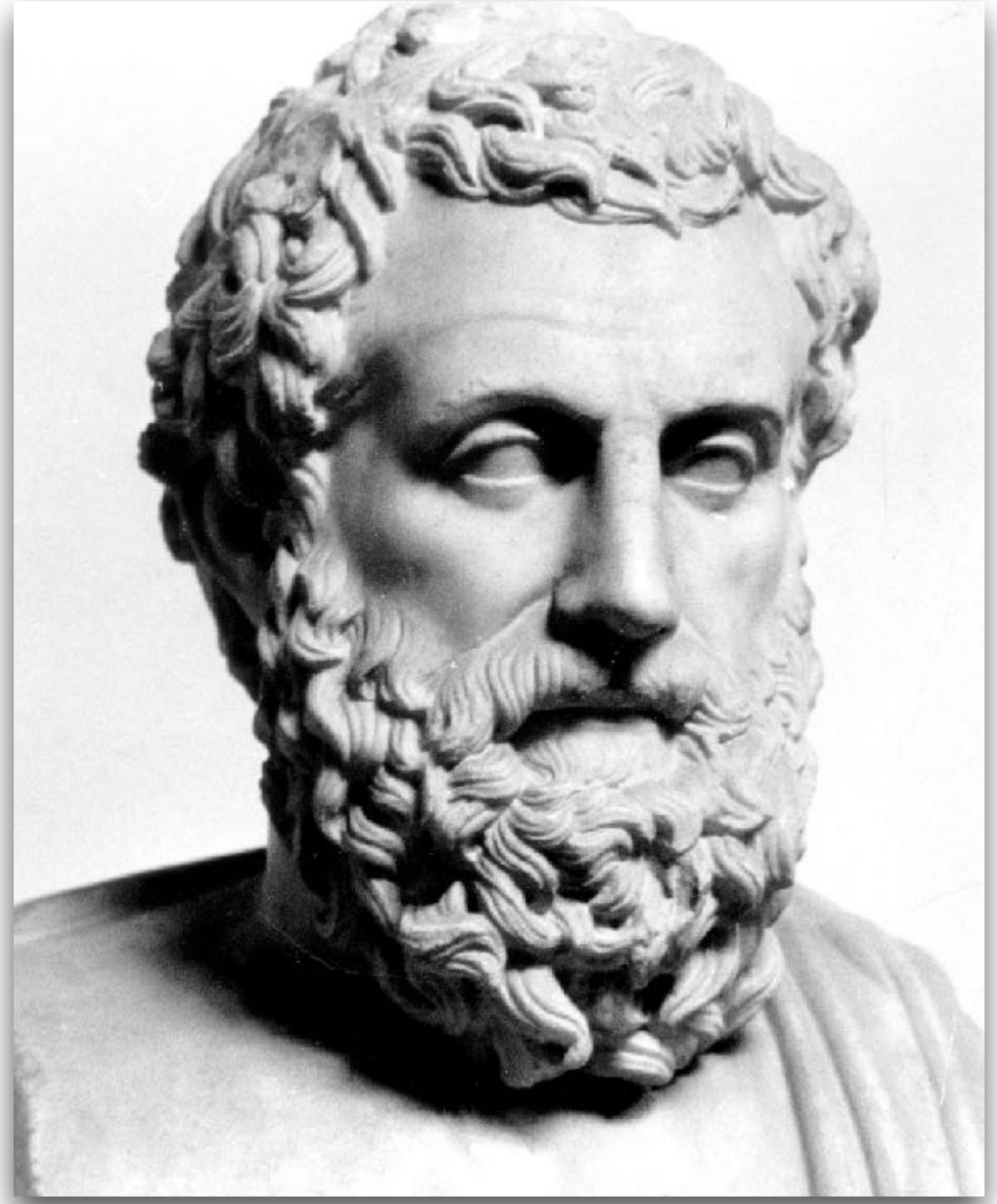
```
(define-relation (not-in-envo x env)
  (conde
    [(== '() env)]
    [(fresh (y _ rest)
       (== `((,y ,_) . ,rest) env)
       (=/= y x)
       (not-in-envo x rest))]))
```

```
(define-relation (=/= n₁ n₂)
  (conde
    [(fresh (pn₂)
       (== n₂ `(s . ,pn₂))
       (== n₁ '()))]
    [(fresh (pn₁)
       (== n₁ `(s . ,pn₁))
       (== n₂ '()))]
    [(fresh (pn₁ pn₂)
       (== n₁ `(s . ,pn₁))
       (== n₂ `(s . ,pn₂))
       (=/= pn₁ pn₂))]))
```

```
(define-relation (not-in-envo x env)
  (conde
    [(== '() env)]
    [(fresh (y _ rest)
       (== `((,y ,_) . ,rest) env)
       (=/= y x)
       (not-in-envo x rest))]))
```

RELATIONAL SYLLOGISTIC LOGIC PROGRAMS

```
(define-relation (A φ Γ prf)
  (matche φ
    [(∀ ,a ,a) (== φ prf)]
    [,x (membero x Γ)
     (== prf `(,x in-Γ))]
    [(∀ ,n ,q)
     (fresh (p prf1 prf2)
       (== `((,prf1 ,prf2) => ,φ) prf)
       (A `(∀ ,n ,p) Γ prf1)
       (A `(∀ ,p ,q) Γ prf2))]))
```



RELATIONAL SYLLOGISTIC LOGIC PROGRAMS

```
(define-relation (A φ Γ prf)
  (matche φ
    [(∀ ,a ,a) (== φ prf)]
    [,x (membero x Γ)
     (== prf `(,x in-Γ))]
    [(∀ ,n ,q)
     (fresh (p prf1 prf2)
       (== `((,prf1 ,prf2) => ,φ) prf)
       (A `(∀ ,n ,p) Γ prf1)
       (A `(∀ ,p ,q) Γ prf2))]))
```

Axiom

Lookup

“Barbara” inference

```
(define-relation (un-atomo a)
  (fresh (sym)
    (symbolo sym)
    (== a `(-2 . ,sym))))
```

Still relying on primitives!

Adding tags!

Results

Results

1. Characterize classes of "miniKanren constraints"

Results

1. Characterize classes of "miniKanren constraints"
2. Connect our constraints to prior work in negation in logic programming

Results

1. Characterize classes of "miniKanren constraints"
2. Connect our constraints to prior work in negation in logic programming
3. A parameterized family of constraint miniKanren languages

Results

1. Characterize classes of "miniKanren constraints"
2. Connect our constraints to prior work in negation in logic programming
3. A parameterized family of constraint miniKanren languages
4. Racket macro-generate these constraint mK language implementations

Results

1. Characterize classes of "miniKanren constraints"
2. Connect our constraints to prior work in negation in logic programming
3. A parameterized family of constraint miniKanren languages
4. Racket macro-generate these constraint mK language implementations
5. Implementing constraint systems via macros

Results

1. Characterize classes of "miniKanren constraints"
2. Connect our constraints to prior work in negation in logic programming
3. A parameterized family of constraint miniKanren languages
4. Racket macro-generate these constraint mK language implementations
5. Implementing constraint systems via macros
6. Introduce new constraints to miniKanren languages

Results

1. Characterize classes of "miniKanren constraints"
2. Connect our constraints to prior work in negation in logic programming
3. A parameterized family of constraint miniKanren languages
4. Racket macro-generate these constraint mK language implementations
5. Implementing constraint systems via macros
6. Introduce new constraints to miniKanren languages
7. miniKanren over microKanren

Thanks!