The Nanopass Framework as a Nanopass Compiler

Andy Keep

Background

Background

The Nanopass Framework is an embedded domain-specific language for creating compilers that focuses on creating single purpose passes and precise intermediate representations. The DSL aims to minimize boilerplate and the resulting compilers are easier to understand and maintain.

Background

- Two language forms: **`define-language`** and **`define-pass`**
- **`define-language`** specifies the grammar of an intermediate language
    - A language can extend an existing language
- **`define-pass`** specifies a pass operating over an input to produce an output
    - A pass can operate over two languages, which might be the same;
    - Only an input language or output language; or
    - Even over non-language inputs and outputs

Example

```
(define-language L1
  (terminals
    (symbol (x))
    (datum (d))
    (primitive (pr)))
  (Expr (e body)
    x
    pr
    (quote d)
    (if e0 e1)
    (if e0 e1 e2)
    (begin e* ... e)
    (let ([x* e*] ...) body* ... body)
    (letrec ([x* e*] ...) body* ... body)
    (lambda (x* ...) body* ... body)
    (e e* ...)))
```

Example

```
(define-language L2
  (extends L1)
  (Expr (e body)
    (- (if e0 e1)
       (let ([x* e*] ...) body* ... body)
       (letrec ([x* e*] ...) body* ... body)
       (lambda (x* ...) body* ... body))
    (+ (letrec ([x* e*] ...) body)
       (lambda (x* ...) body))))
```

Example

```
(define-pass simplify : L1 (ir) -> L2 ()
  (Expr : Expr (e) -> Expr ()
    [(if ,[e0] ,[e1]) `(if ,e0 ,e1 (void))]
    [(lambda (,x* ...) ,[body*] ... ,[body])
     `(lambda (,x* ...) (begin ,body* ... ,body))]
    [(letrec ([,x* ,[e*]] ...) ,[body*] ... ,[body])
     `(letrec ([,x* ,e*] ...) (begin ,body* ... ,body))]
    [(let ([,x* ,[e*]] ...) ,[body*] ... ,[body])
     `((lambda (,x* ...) (begin ,body* ... ,body)) ,e* ...)]))
```

Evolution

- `define-language`
  - `language->s-expression`
  - `diff-languages`
  - `prune-language`
  - `define-language-node-counter`
  - `define-parser`
  - `define-unparser`
  - *etc.*

- `define-pass`
  - `with-output-language`
  - `nanopass-case`
  - `echo-define-pass`
  - `trace-define-pass`
  - `pass-input-parser`
  - `pass-output-unparser`
  - *etc.*

What do I want?

- A language for nanopass languages
    - Many extensions naturally flow from this: **`language->s-expression`**, **`diff-languages`**, **`prune-language`**, **`define-parser`**, and **`define-language-node-counter`**
- A language for nanopass passes
    - Extensions like **`echo-define-pass`** could be improved
- Why not write even more of the nanopass framework using this?

An API for languages

The language of languages

- **`define-language`** already provides a syntax, why not just use it?
  - Grammar is messy
    - Language clauses are unordered
    - Pretty syntax for unparsers can use non-s-expression syntax
      `(call e e* ...) => (e e* ...)`
    - Language extensions are part of the grammar
  - Meta-variables need to be mapped to terminal and nonterminal clauses

Aside: nanopass internals

Aside: current internal structure

- Languages are represented as a collection of records:
  - **`language`** - describes fixed parts and contains terminals and nonterminals
  - **`tspec`** - describes a terminal: predicate, meta-vars, etc.
  - **`ntspec`** - describes a nonterminal: predicates, meta-vars, productions, etc.
  - **`alt`** - describes a production: syntax, etc. with three derived records:
    - **`pair-alt`** - pattern production: pattern, fields, etc.
    - **`terminal-alt`** - bare terminal production
    - **`nonterminal-alt`** - bare nonterminal production (essentially a subterminal)

Aside: current internal structure

- Language description records contain source syntax and internal information
- Description can be used to generate record definitions, constructors, etc.
- The internal information is not needed for **`language->s-expression`**, etc.
- Perhaps our language API should provide both views:
    - A language for describing something closer to the source structure
    - An annotated language for describing the internal details

Aside: patterns

- Patterns are composed of the following forms:
  - **`id`** - a bare identifier, always a reference to a terminal or nonterminal
  - **`(maybe id)`** - represents an optional field, will have a value or **`#f`**
  - **`()`** - matches null
  - **`(x . y)`** - matches a pair of patterns: **`x`** and **`y`**
  - **`(x dots)`** - matches a list of pattern **`x`** (**`dots`** is the syntax **`...`**)
  - **`(x dots y ... . z)`** - matches a list of **`x`**, followed by zero or more patterns **`y`**, terminated by a final pattern **`z`**

Aside: patterns

- Patterns are composed of the following forms:
  - **`id`** - a bare identifier, always a reference to a terminal or nonterminal
  - **`(maybe id)`** - represents an optional field, will have a value or **`#f`**
  - **`()`** - matches null
  - **`(x . y)`** - matches a pair of patterns: **`x`** and **`y`**
  - **`(x dots)`** - matches a list of pattern **`x`** (**`dots`** is the syntax **`...`**)
  - **`(x dots y ... . z)`** - matches a list of **`x`**, followed by zero or more patterns **`y`**, terminated by a final pattern **`z`**

Too complicated!

Aside: patterns

- Patterns are composed of the following forms:
  - **`id`** - a bare identifier, always a reference to a terminal or nonterminal
  - **`(ma`**
  - **`()`** -
  - **`(x`**

`(x dots)` is really the same as **`(x dots y ... . z)`**

where **`(y ...)`** is zero length and **`z`** is **`null`**

  - **`(x dots)`** - matches a list of pattern **`x`** (**`dots`** is the syntax **`...`**)
  - **`(x dots y ... . z)`** - matches a list of **`x`**, followed by zero or more patterns **`y`**, terminated by a final pattern **`z`**

Aside: patterns

- Patterns are composed of the following forms:
  - **`id`** - a bare identifier, always a reference to a terminal or nonterminal
  - **`(maybe id)`** - represents an optional field, will have a value or **`#f`**
  - **`()`** - matches null
  - **`(x . y)`** - matches a pair of patterns: **`x`** and **`y`**
  - **`(x dots y ... . z)`** - matches a list of **`x`**, followed by zero or more patterns **`y`**, terminated by a final pattern **`z`**

Aside: patterns

- Patterns are composed of the following forms:
  - **`id`** - a bare identifier, always a reference to a terminal or nonterminal
  - **`(maybe id)`** - represents an optional field, will have a value or **`#f`**
  - **`()`** - matches null
  - **`(x . y)`** - matches a pair of patterns: **`x`** and **`y`**
  - **`(x dots y ... . z)`** - matches a list of **`x`**, followed by zero or more patterns **`y`**, terminated by a final pattern **`z`**

Still!!
Too complicated!

Aside: patterns

- Patterns are composed of the following forms:
  - **`id`** - a bare identifier, always a reference to a terminal or nonterminal
  - **`(may`**
  - **`()`** -
  - **`(x`**

`(x dots y ... . z)` is **`x dots`** followed by an improper list, but we can represent an improper list with **`(x . y)`**, so we really just need **`(x dots . y)`**

  - **`(x dots y ... . z)`** - matches a list of **`x`**, followed by zero or more patterns **`y`**, terminated by a final pattern **`z`**

Aside: patterns

- Patterns are composed of the following forms:
    - **`id`** - a bare identifier, always a reference to a terminal or nonterminal
    - **`(maybe id)`** - represents an optional field, will have a value or **`#f`**
    - **`()`** - matches null
    - **`(x . y)`** - matches a pair of patterns: **`x`** and **`y`**
    - **`(x dots . y)`** - matches a list of pattern **`x`** followed by a pattern **`y`** where **`dots`** is the syntax **`...`**

Language API

The simple language

```
(define-language Llanguage
  (terminals
    (identifier (id))
    (datum (handler))
    (box (b))
    (dots (dots))
    (null (null)))
  (Defn (def)
    (define-language id cl* ...))
  (Clause (cl)
    (entry ref)
    (terminals term* ...)
    (nongenerative-id id)
    (id (id* ...) b prod* ...))
  (Terminal (term)
    simple-term
    (=> simple-term handler))
```

```
  (SimpleTerminal (simple-term)
    (id (id* ...) b))
  (Production (prod)
    pattern
    (=> pattern0 pattern1)
    (-> pattern handler))
  (Pattern (pattern)
    id
    null
    ref
    (maybe ref)
    (pattern0 . pattern1)
    (pattern0 dots . pattern1))
  (Reference (ref)
    (term-ref id0 id1 b)
    (nt-ref id0 id1 b)))
```

The simple language

```
(define-language Llanguage
  (terminals
    (identifier (id))
    (datum (handler))
    (box (b))
    (dots (dots))
    (null (null)))
  (Defn (def)
    (define-language id
  (Clause (cl)
    (entry ref)
    (terminals term* ..
    (nongenerative-id i
    (id (id* ...) b prod* ...))
  (Terminal (term)
    simple-term
    (=> simple-term handler))
```

```
  (SimpleTerminal (simple-term)
    (id (id* ...) b))
  (Production (prod)
    pattern
          attern0 pattern1)
          attern handler))
          (pattern)



          ref)
          ern0 . pattern1)
    (pattern0 dots . pattern1))
  (Reference (ref)
    (term-ref id0 id1 b)
    (nt-ref id0 id1 b)))
```

```
(terminals
  (identifier (id))
  (datum (handler))
  (box (b))
  (dots (dots))
  (null (null)))
```

The simple language

```
(define-language Llanguage
  (terminals
    (identifier (id))
    (datum (handler))
    (box (b))
    (dots (dots))
    (null (null)))
  (Defn (def)
    (define-langu
  (Clause (cl)
    (entry ref)
    (terminals term* ...)
    (nongenerative-id id)
    (id (id* ...) b prod* ...))
  (Terminal (term)
    simple-term
    (=> simple-term handler))
```

```
  (SimpleTerminal (simple-term)
    (id (id* ...) b))
  (Production (prod)
    pattern
    (=> pattern0 pattern1)
    (=> pattern handler))
               tern)

    ref
    (maybe ref)
    (pattern0 . pattern1)
    (pattern0 dots . pattern1))
  (Reference (ref)
    (term-ref id0 id1 b)
    (nt-ref id0 id1 b)))
```

```
(Defn (def)
  (define-language id cl* ...))
```

The simple language

```
(define-language Llanguage
  (terminals
    (identifier (id))
    (datum (handler))
    (box (b))
    (dots (dots))
    (null (null)))
  (Defn (def)
    (define-langua
  (Clause (cl)
    (entry ref)
    (terminals ter
    (nongenerative-id id)
    (id (id* ...) b prod* ...))
  (Terminal (term)
   simple-term
    (=> simple-term handler))
```

```
(SimpleTerminal (simple-term)
  (id (id* ...) b))
(Production (prod)
  pattern
           n0 pattern1)
           n handler))
           ttern)

          )
  (pattern0 . pattern1)
  (pattern0 dots . pattern1))
(Reference (ref)
  (term-ref id0 id1 b)
  (nt-ref id0 id1 b)))
```

```
(Clause (cl)
  (entry ref)
  (terminals term* ...)
  (nongenerative-id id)
  (id (id* ...) b prod* ...))
```

The simple language

```
(define-language Llanguage
  (terminals
    (identifier (id))
    (datum (handler))
    (box (b))
    (dots (dots))
    (null (null)))
  (Defn (def)
    (define-langua
  (Clause (cl)
    (entry ref)
    (terminals ter
    (nongenerative-id id)
    (id (id* ...) b prod* ...))
  (Terminal (term)
    simple-term
    (=> simple-term handler))
```

```
  (SimpleTerminal (simple-term)
    (id (id* ...) b))
  (Production (prod)
    pattern
               n0 pattern1)
               n handler))
               ttern)

                    )
    (pattern0 . pattern1)
    (pattern0 dots . pattern1))
  (Reference (ref)
    (term-ref id0 id1 b)
    (nt-ref id0 id1 b)))
```

```
(Terminal (term)
  simple-term
  (=> simple-term handler))
(SimpleTerminal (simple-term)
  (id (id* ...) b))
```

The simple language

```
(define-language Llanguage
  (terminals                                          nal (simple-term)
    (identifier (i                                    ..) b))
    (datum (handle                                    (prod)
    (box (b))
    (dots (dots))                                     n0 pattern1)
    (null (null)))                                    n handler))
  (Defn (def)                                         ttern)
    (define-langua
  (Clause (cl)
    (entry ref)
    (terminals ter                                    )
    (nongenerative                                    . pattern1)
    (id (id* ...)                                     dots . pattern1))
  (Terminal (term)                                    ref)
    simple-term                                       id0 id1 b)
    (=> simple-term handler))                         (nt-ref id0 id1 b)))
```

```
(Production (prod)
  pattern
  (=> pattern0 pattern1)
  (-> pattern handler))
(Pattern (pattern)
  id
  null
  ref
  (maybe ref)
  (pattern0 . pattern1)
  (pattern0 dots . pattern1))
(Reference (ref)
  (term-ref id0 id1 b)
  (nt-ref id0 id1 b)))
```

The simple language

```
(define-language Llanguage
  (terminals
    (identifier (id))
    (datum (handler))
    (box (b))
    (dots (dots))
    (null (null)))
  (Defn (def)
    (define-language id cl* ...))
  (Clause (cl)
    (entry ref)
    (terminals term* ...)
    (nongenerative-id id)
    (id (id* ...) b prod* ...))
  (Terminal (term)
    simple-term
    (=> simple-term handler))
  (SimpleTerminal (simple-term)
    (id (id* ...) b))
  (Production (prod)
    pattern
    (=> pattern0 pattern1)
    (-> pattern handler))
  (Pattern (pattern)
    id
    null
    ref
    (maybe ref)
    (pattern0 . pattern1)
    (pattern0 dots . pattern1))
  (Reference (ref)
    (term-ref id0 id1 b)
    (nt-ref id0 id1 b)))
```

The annotated language

```
(define-language Lannotated
  (terminals
    (record-constructor-descriptor (rcd))
    (record-type-descriptor (rtd))
    (exact-integer (tag level tag-mask))
    (datum (handler pred all-pred all-term-pred accessor maker))
    (box (b))
    (identifier (id))
    (dots (dots))
    (null (null)))
  (Defn (def)
    (define-language id ref (maybe id0)
      rtd rcd tag-mask
      (term* ...)
      nt* ...))
  (Terminal (term)
    (id (id* ...) b (maybe handler) pred))
  (Nonterminal (nt)
    (id (id* ...) b rtd rcd tag pred all-pred all-term-pred
      prod* ...))
  (Production (prod)
    (production pattern (maybe pretty-prod) rtd tag pred maker
      field* ...)
    (terminal ref (maybe pretty-prod))
    (nonterminal ref (maybe pretty-prod)))
```

```
(PrettyProduction (pretty-prod)
  (procedure handler)
  (pretty pattern))
(Field (field)
  (ref level accessor)
  (optional ref level accessor))
(Pattern (pattern)
  id
  null
  ref
  (maybe ref)
  (pattern0 . pattern1)
  (pattern0 dots . pattern1))
(Reference (ref)
  (term-ref id0 id1 b)
  (nt-ref id0 id1 b)))
```

The annotated language

```
(define-language Lannotated
  (terminals
    (record-constructor-descriptor (rcd))
    (record-type-descriptor (rtd))
    (exact-integer (tag level tag-mask))
```

```
(terminals
  (record-constructor-descriptor (rcd))
  (record-type-descriptor (rtd))
  (exact-integer (tag level tag-mask))
  (datum (handler pred all-pred all-term-pred accessor maker))
  (box (b))
  (identifier (id))
  (dots (dots))
  (null (null)))
```

```
  (D
  (T
  (N
```

```
  (Production (prod)
    (production pattern (maybe pretty-prod) rtd tag pred maker
      field* ...)
    (terminal ref (maybe pretty-prod))
    (nonterminal ref (maybe pretty-prod)))
```

```
  (Reference (ref)
    (term-ref id0 id1 b)
    (nt-ref id0 id1 b)))
```

The annotated language

```
(define-language Lannotated
  (terminals
    (record-constructor-descriptor (rcd))
    (record-type-descriptor (rtd))
    (exact-integer (tag level tag-mask))
    (datum (handler pred all-pred all-term-pred accessor maker))
    (box (b))
    (identifier (id))
    (dots (dots))
    (null (null)))
  (Defn (def)
    (define-language i
      rtd rcd tag-mask
      (term* ...)
      nt* ...))
  (Terminal (term)
    (id (id* ...) b (r
  (Nonterminal (nt)
    (id (id* ...) b rtd rcd tag pred all-pred all-term-pred
      prod* ...))
  (Production (prod)
    (production pattern (maybe pretty-prod) rtd tag pred maker
      field* ...)
    (terminal ref (maybe pretty-prod))
    (nonterminal ref (maybe pretty-prod)))
```

```
tion (pretty-prod)
handler)
tern))
l)
accessor)
ef level accessor))
tern)
```

```
    (maybe ref)
    (pattern0 . pattern1)
    (pattern0 dots . pattern1))
  (Reference (ref)
    (term-ref id0 id1 b)
    (nt-ref id0 id1 b)))
```

```
(Defn (def)
  (define-language id ref (maybe id0)
    rtd rcd tag-mask
    (term* ...)
    nt* ...))
```

The annotated language

```
(define-language Lannotated
  (terminals
    (record-constructor-descriptor (rcd))
    (record-type-descriptor (rtd))
    (exact-integer (tag level tag-mask))
    (datum (handler pred all-pred all-term-pred accessor maker))
    (box (b))
    (identifier (id))
    (dots (dots))
    (null (null)))
  (Defn (def)
    (define-language
      rtd rcd tag-ma
      (term* ...)
      nt* ...))
  (Terminal (term)
    (id (id* ...) b (maybe handler) pred))
  (Nonterminal (nt)
    (id (id* ...) b rtd rcd tag pred all-pred all-term-pred
      prod* ...))
  (Production (prod)
    (production pattern (maybe pretty-prod) rtd tag pred maker
      field* ...)
    (terminal ref (maybe pretty-prod))
    (nonterminal ref (maybe pretty-prod)))
```

```
(PrettyProduction (pretty-prod)
  (procedure handler)
             ern))

            ccessor)
            f level accessor))
            ern)

  null
  ref
  (maybe ref)
  (pattern0 . pattern1)
  (pattern0 dots . pattern1))
(Reference (ref)
  (term-ref id0 id1 b)
  (nt-ref id0 id1 b)))
```

```
(Terminal (term)
  (id (id* ...) b (maybe handler) pred))
```

The annotated language

```
(define-language Lannotated
  (terminals
    (record-constructor-descriptor (rcd))
    (record-type-descriptor (rtd))
    (exact-integer (tag level tag-mask))
    (datum (handler pred all-pred all-term-pred accessor maker))
    (box (b))
    (identifier (id))
    (dots (dots))
    (null (null))
  (Defn (def)
    (define-langu
      rtd rcd tag
      (term* ...)
      nt* ...))
  (Terminal (term
    (id (id* ...) b (maybe handler) pred))
  (Nonterminal (nt)
    (id (id* ...) b rtd rcd tag pred all-pred all-term-pred
      prod* ...))
  (Production (prod)
    (production pattern (maybe pretty-prod) rtd tag pred maker
      field* ...)
    (terminal ref (maybe pretty-prod))
    (nonterminal ref (maybe pretty-prod)))
```

```
  (PrettyProduction (pretty-prod)
                    ler)
                    ))

                    ssor)
                    evel accessor))
                    )

    ref
    (maybe ref)
    (pattern0 . pattern1)
    (pattern0 dots . pattern1))
  (Reference (ref)
    (term-ref id0 id1 b)
    (nt-ref id0 id1 b)))
```

```
(Nonterminal (nt)
  (id (id* ...) b
    rtd rcd tag pred all-pred all-term-pred
    prod* ...))
```

The annotated language

```
(define-language Lannotated
  (terminals
    (record-constructor-descriptor (rcd))
    (record-type-d
    (exact-integer
    (datum (handle
    (box (b))
    (identifier (i
    (dots (dots))
    (null (null)))
  (Defn (def)
    (define-langua
      rtd rcd tag-
      (term* ...)
      nt* ...))
  (Terminal (term)
    (id (id* ...)
  (Nonterminal (nt
    (id (id* ...)
      prod* ...))
  (Production (pro
    (production pa
      field* ...)
    (terminal ref (maybe pretty-prod))
    (nonterminal ref (maybe pretty-prod)))
```

```
n (pretty-prod)
dler)
n))

essor)
level accessor))
n)

ttern1)
. pattern1))

id1 b)
(nt-ref id0 id1 b)))
```

```
(Production (prod)
  (production pattern (maybe pretty-prod)
    rtd tag pred maker
    field* ...)
  (terminal ref (maybe pretty-prod))
  (nonterminal ref (maybe pretty-prod)))
(PrettyProduction (pretty-prod)
  (procedure handler)
  (pretty pattern))
(Field (field)
  (ref level accessor)
  (optional ref level accessor))
```

The annotated language

```
(define-language Lannotated
  (terminals
    (record-constructor-descriptor (rcd))
    (record-type-descriptor (rtd))
    (exact-integer (tag lev
    (datum (handler pred al
    (box (b))
    (identifier (id))
    (dots (dots))
    (null (null)))
  (Defn (def)
    (define-language id ref
      rtd rcd tag-mask
      (term* ...)
      nt* ...))
  (Terminal (term)
    (id (id* ...) b (maybe
  (Nonterminal (nt)
    (id (id* ...) b rtd rcd
      prod* ...))
  (Production (prod)
    (production pattern (maybe pretty-prod) rtd tag pred maker
      field* ...)
    (terminal ref (maybe pretty-prod))
    (nonterminal ref (maybe pretty-prod)))
```

```
Production (pretty-prod)
edure handler)
ty pattern))
(field)
level accessor)
onal ref level accessor))
n (pattern)

e ref)
ern0 . pattern1)
ern0 dots . pattern1))
(Reference (ref)
  (term-ref id0 id1 b)
  (nt-ref id0 id1 b)))
```

```
(Pattern (pattern)
  id
  null
  ref
  (maybe ref)
  (pattern0 . pattern1)
  (pattern0 dots . pattern1))
(Reference (ref)
  (term-ref id0 id1 b)
  (nt-ref id0 id1 b)))
```

Using the Language API

The language experiment

- Two libraries **`(nanopass experimental)`** and **`(nanopass exp-syntax)`**
- **`(nanopass experimental)`** contains languages and passes
  - **`lookup-language`** retrieves language forms from syntactic environment
  - **`language-information-language`** returns **`Llanguage`**
  - **`language-information-annotated-language`** returns **`Lannotated`**
- **`(nanopass exp-syntax)`** contains new syntactic forms
  - **`define-language-exp`**, **`language->s-expression-exp`**, **`prune-language-exp`**, **`diff-languages-exp`**, *etc.*

The language experiment

```
(define-syntax define-language-exp
  (lambda (x)
    (lambda (rho)
      (syntax-case x ()
        [(_ . rest)
         (let* ([lang (parse-np-source x 'define-language-exp)]
                [lang (handle-language-extension
                        lang 'define-language-exp rho)]
                [lang (check-and-finish-language lang)]
                [lang-annotated (annotate-language lang)])
           (nanopass-case (Llanguage Defn) lang
             [(define-language ,id ,cl* ...)
              #`(begin
                  (define-language . rest)
                  (define-property #,id experimental-language
                    (make-language-information '#,lang '#,lang-annotated))
                  (define-language-records #,id)
                  #;(define-language-predicates #,id))]))]))))
```

The language experiment

How has it turned out?

- Rewrote all of the language extensions as passes over languages
    - Often used annotated language to avoid unordered clauses
    - **`Llanguage`** might be better with this structure.
    - Patterns instead of syntax made some things a little more complicated
- Producing syntax is relatively easy with a couple caveats
    - Sometimes need to use **`datum->syntax`** to "repaint" identifiers
    - Might want to expand into a pass language with a helper to produce syntax

An API for passes

The language of passes

- Well... I didn't quite get to this yet.

The language of passes

- Well... I didn't quite get to this yet.

- So, instead lets talk about plans...

Future direction

Future direction

- Next step is to add a language of passes
  - Implementing the language is not too difficult
  - Implementing **`define-pass-exp`** is a little more involved
    - Need to implement meta-parser for matching and construction
    - Need to implement boilerplate generation code as a pass
    - Provides an opportunity to improve things

Future direction

- The experimental language and pass are only a start
- They still rely on the original nanopass framework to work
- We need a way to partially evaluate these to produce a language core
- This is possibly the most challenge part

Wrapping up

Wrapping up

- The language experiment seems promising
- The pass experiment seems relatively straightforward
- I'm hopeful the core can be generated from this source
- You can try it out (currently in Chez Scheme only):
  https://github.com/nanopass/nanopass-framework-scheme/

Thanks!

https://github.com/nanopass/nanopass-framework-scheme/

Questions?

https://github.com/nanopass/nanopass-framework-scheme/