Staged Relational Interpreters: Running with Holes, Faster

NADA AMIN, Harvard University, USA
MICHAEL BALLANTYNE, Northeastern University, USA
WILLIAM E. BYRD, University of Alabama at Birmingham, USA

We present a novel framework for staging interpreters written as relations, in which the programs under interpretation are allowed to contain holes representing unknown values. We apply this staging framework to a relational interpreter for a subset of Racket, and demonstrate significant performance gains across multiple synthesis problems.

Additional Key Words and Phrases: relational programming, functional programming, staging, interpreters, synthesis, miniKanren, Racket, Scheme

1 INTRODUCTION

Here is the well-known Fibonacci function in accumulator-passing style, using Peano numerals.

```
(define fib-aps
  (lambda (n a1 a2)
    (if (zero? n)
        a1
        (if (zero? (sub1 n))
            a2
            (fib-aps (- n '(s . z)) a2 (+ a1 a2))))))
(define fib (lambda (n) (fib-aps n 'z '(s . z))))
(fib 'z) => z
(fib '(s . z)) => (s . z)
(fib '(s s . z)) => (s . z)
(fib '(s s s . z)) => (s s . z)
(fib '(s s s s . z)) => (s s s . z)
(fib '(s s s s s . z)) => (s s s s s . z)
```

In a relational interpreter — an interpreter written in a relational language — we can leave holes in the interpreted program. For example, we can synthesize the initial values and the stepping of the accumulators (represented by unquoted variables), given the example calls to `fib`.

```
(define fib-aps
  (lambda (n a1 a2)
    (if (zero? n)
        a1
        (if (zero? (sub1 n))
            a2
            (fib-aps (- n '(s . z)) ,A ,B)))))
(define fib (lambda (n) (fib-aps n ,ACC1 ,ACC2)))
```

Relational interpreters can turn functions into relations. For example, Byrd et al. [2017] show a theorem checker turned theorem prover. Written as a Racket program, the theorem checker runs in a relational interpreter written in miniKanren. When querying the program, one can leave a hole for the proof of a statement to synthesize the proof.

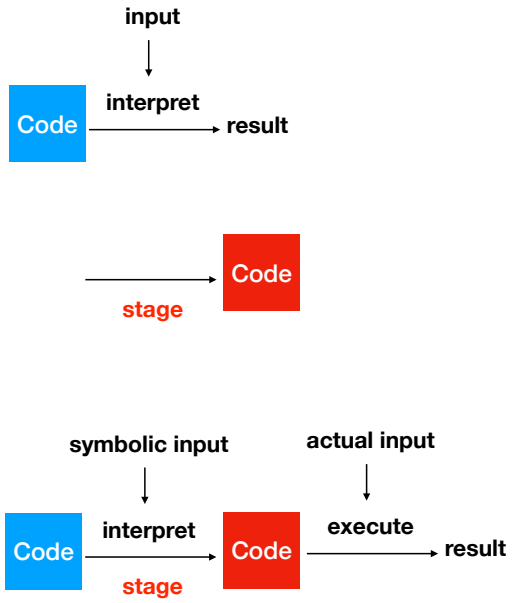

Fig. 1. Turning an interpreter into a compiler via staging. Interpretation is a process that takes code as input and produces a result. Staging is a process that produces code as output. Staged interpretation is compilation: it takes code as input and produces specialized code as output. In our setting, the blue code is functional code, the interpreter is the relational interpreter, and the red code is relational code.

Although relational interpreters are flexible, they suffer from overhead, the consequence of a layer of interpretation. It is well known that staging an interpreter removes interpretive overhead, yielding a compiler (Figure 1). Unfortunately, the standard techniques for staging do not work in the presence of holes. We show how to stage a relational interpreter while preserving the ability to place holes anywhere within the program under interpretation. This technique can also be used to remove interpretive overhead when using the staged interpreter to turn a function into a relation (Figure 2).

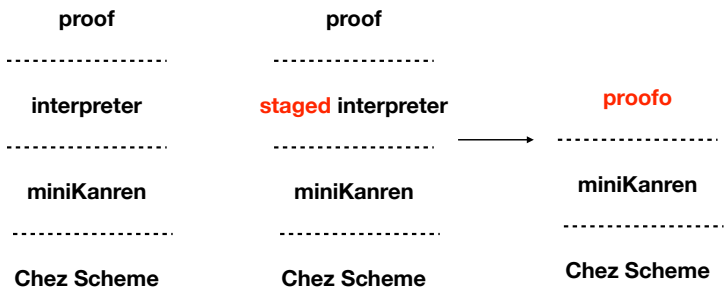

Fig. 2. Turning a function (theorem checker `proof`) into a relation (theorem prover `proofo`) without interpretive overhead via staging.

Our contributions:

- We add staging constructs to miniKanren, creating a multi-stage programming framework. The staging constructs syntactically save code fragments that are then assembled as part of an answer. We describe the interface in Section 3.1 and the implementation in Section 4.2.
- We stage a relational interpreter. We describe the interface in Section 3.2 and the implementation in Section 4.3. We explain the process of staging relational code.
- We exercise various programs that run with holes faster in the benchmarks of Section 6.1. The gains are quantitatively and qualitatively substantial as programs that stall when unstaged can complete within a second when staged. For the Fibonacci synthesis example with which we started, our staged relational interpreter outperforms Barliman [Byrd et al. 2017] by an order of magnitude, even though the latter includes hand-crafted low-level optimizations.
- We explore program synthesis by dealing with holes (unbound logic variables) in the generator in Section 5.

We assume passing familiarity with Scheme.
Our code is available at https://github.com/namin/staged-miniKanren.

2 BACKGROUND

2.1 miniKanren

miniKanren [Friedman et al. 2005, 2018] is a pure logic programming language, exemplifying relational programming: programming with pure relations with no distinction between inputs and output.

miniKanren is embedded in Scheme. The two languages interface through the `run` special form. The == binary function unifies its arguments.

```
(run* (q) (== q 5))
;; (5)
```

We get a list of answers. We get an empty list if the query fails.

```
(run* (q) (== 5 6))
;; ()
```

The query variable can also remain unbound, indicated by $\_.n$ where n is a number in order of reification.

```
(run* (q) (== 5 5))
;; (_.0)
```

The `fresh` form introduces new logic variables, and its body is a conjunction of clauses.

```
(run* (q) (fresh (x y) (== q `(,x ,y))))
;; ((_.0 _.1))
(run* (q) (fresh (x y) (== q `(,x ,y)) (== x 5)))
;; ((5 _.0))
```

The `conde` form introduces a disjunction of conjunctions.

```
(run* (q)
  (conde
    ((== q 5))
    ((== q 6))))
;; (5 6)
```

We get two answers, one for each successful branch of the conde form. By default, a query returns all answers and diverges if there are infinitely many. We can also explicitly ask for just *n* answers instead. The following query returns only one answer of all that satisfy the query.

```
(run 1 (q)
  (conde
    ((== q 5))
    ((== q 6))))
;; (5)
```

We can write recursive relations by reusing the functionality of Scheme, here the define form. Here is a well-known logic program to append two lists.

```
(define (appendo xs ys zs)
  (conde
    ((== xs '()) (== ys zs))
    ((fresh (xa xd zd)
       (== xs (cons xa xd))
       (== zs (cons xa zd))
       (appendo xd ys zd)))))
(run* (q) (appendo '(a b) '(c d e) q))
;; ((a b c d e))
(run* (q) (fresh (x y) (== q (list x y)) (appendo x y '(a b c d e))))
;; (
;;  (() (a b c d e))
;;  ((a) (b c d e))
;;  ((a b) (c d e))
;;  ((a b c) (d e))
;;  ((a b c d) (e))
;;  ((a b c d e) ())
;; )
(run* (x y z) (appendo `(a  . ,x) `(,y e) `(a b c d ,z)))
;; (((b c) d e))
```

Note that we can have multiple query variables like x, y and z in the last query. Then each answer is a list of those query variables: x is (b c), y is d and z is e.

miniKanren has constraints: type constraints numbero and symbolo, =/= (disequality) and absento constraints. Reified constraints appearing in answers are preceded by $$. The examples below hint at constraint simplification: the disequality constraint with foo is removed when the query variable q is a number, but left when it is a symbol.

```
(run* (q)
  (numbero q)
  (=/= 'foo q))
;; ((_.0 $$ (num _.0)))

(run* (q)
  (symbolo q)
  (=/= 'foo q))
;; ((_.0 $$ (=/= ((_.0 foo))) (sym _.0)))
```

2.2 Relational Interpreters

Previous work [Byrd et al. 2017, 2012] has explored writing and running interpreters for subsets of Racket or Scheme written in miniKanren.

The relation `evalo` relates an expression to the value it evaluates to.

```
(run* (q) (evalo 1 q))
;; (1)
(run* (q) (evalo '(car '(1 2)) q))
;; (1)
(run 4 (q) (evalo q q))
;; ((_.0 (num _.0))
;;   #t
;;   #f
;;   (((lambda (_.0) (list _.0 (list 'quote _.0)))
;;      '(lambda (_.0) (list _.0 (list 'quote _.0))))
;;    (=/= ((_.0 closure))
;;         ((_.0 list))
;;         ((_.0 prim))
;;         ((_.0 quote)))
;;    (sym _.0)))
```

The last query generates quines. The expression generated is indeed self-evaluating.

```
((lambda (x) (list x (list 'quote x)))
 '(lambda (x) (list x (list 'quote x))))
;; ((lambda (x) (list x (list 'quote x)))
;;  '(lambda (x) (list x (list 'quote x))))
```

Note that in general, increasing the number of clauses in the interpreter slows down synthesis.

We can turn functions into relations. For example, we give the functional definition of append, and we get the behavior of appendo.

```
(run* (x y)
  (evalo
   `(letrec ((append (lambda (xs ys)
                       (if (null? xs) ys
                           (cons (car xs) (append (cdr xs) ys))))))
      (append ',x ',y))
   '(a b c d e)))
;; (
;;  (() (a b c d e))
;;  ((a) (b c d e))
;;  ((a b) (c d e))
;;  ((a b c) (d e))
;;  ((a b c d) (e))
;;  ((a b c d e) ())
;; )
```

The dilemma: either we write the relation by hand, or we write the function in the `evalo` interpreter. The latter is easier, but incurs interpretation overhead. Staged relational interpreters show a way out of this dilemma.

2.3 Multi-Stage Programming in Functional Programming

We illustrate multi-stage programming in functional programming with the $\lambda_{\uparrow\downarrow}$ system due to Amin and Rompf [2017]. The $\lambda_{\uparrow\downarrow}$ language which is the target of Pink is unconventional in that it is stage-polymorphic.

The $\lambda_{\uparrow\downarrow}$ language has a primitive `lift` construct that lifts a bare value into a code value. The stage polymorphism comes into play for elimination forms: if the arguments are code, the result is code (deferred to the generated code); and if the arguments are not code, then the result is not code (evaluated right now).

Variable bindings are represented by de Bruijn levels. A λ function consumes two de Bruijn levels, the self (for easy recursion) and the parameter.

2.3.1 Lifting into Code.

```
(let (lambda (lambda (+ (var 1) (var 3))))
  (((var 0) 1) 2))
;; 3

(let (lambda (lambda (+ (var 1) (var 3))))
  (((var 0) (lift 1)) (lift 2)))
;; (code (let (+ 1 2) (var 0)))

(let (lambda (lambda ((var 1) (var 3))))
  (((var 0) (lambda (var 2))) 5))
;; 5

(let (lambda (lambda ((var 1) (var 3))))
  (((var 0) (lift (lambda (var 2)))) (lift 5)))
;; (code (let (lambda (var 1)) (let ((var 0) 5) (var 1))))
```

2.3.2 Turning Interpreters into Compilers. We can turn an interpreter into a compiler by making the program static (first-stage) and the inputs to the program dynamic (second-stage). Then, the program dispatching can happen during code generation. The result is a naive compiler which removes interpretation overhead. See Figure 1.

3 NEW GROUND

3.1 Staged miniKanren

3.1.1 Staging. The `l==` relation produces a staged, i.e. later, unification, returned as code marked after `!!` in a query answer.

```
(run* (q) (l== q 1))
;; ((_.0 !! ((== _.0 '1))))
```

Contrast the answer with the answer produced by a usual unification. The generated code after the `!!` above yields the same answer as the usual unification.

```
(run* (q) (== q 1))
;; (1)
```

Note that no optimizations are performed. The problem of optimizing the generated code is orthogonal. As an example of inefficiency, a later unification of similar structures (such as lists) does not cause the later unifications of parts.

```
(run* (q) (l== (list 1) (list q)))
```

```
;; ((_.0 !! ((== (cons '1 '()) (cons _.0 '()))))
;; not simplified to ((_.0 !! ((== '1 _.0))))
```

Staging of a unification that will certainly fail does not cause the first stage (code generation) to fail. Instead, it generates code that will certainly fail. This behavior is useful in ensuring that staging succeeds in all non-error cases.

```
(run* (q) (l== 1 2))
;; ((_.0 !! ((== '1 '2))))
```

The more general staging relation is called `later`; it produces a code fragment from its argument. The `l==` relation is implemented in terms of `later`. (We explain the `expand` annotations in Section 4.2.)

```
(run* (q) (later `(== ,(expand q) ,(expand 1))))
;; ((_.0 !! ((== _.0 '1))))
```

3.1.2 Running staged. The form `run-staged` runs a query in two stages: the first stage generates code, and the second stage immediately runs the generated code from the first stage. The form `run-staged` requires that the first stage results in one answer (no failure and no non-determinism). We forbid non-determinism during code generation in order to produce one later-stage program that encompasses all possibilities; staged relations are written to defer non-determinism to runtime (Section 4.3.4). The number parameter indicates the number of answers requested in the second stage.

Staging and running should produce the same result as an unstaged run:

```
(run-staged 1 (q)
  (l== q 1))
;; (1)
```

The following query causes an error because of non-determinism in the first stage.

```
(run-staged 2 (q)
  (conde
    ((l== q 1))
    ((l== q 2))))
;; running first stage
;; result 1: (_.0 !! ((== _.0 '1)))
;; result 2: (_.0 !! ((== _.0 '2)))
;; Exception: staging non-deterministic
```

We can still have non-determinism in the second stage by staging a conditional.

```
(run-staged 2 (q)
  (later `(conde
            ((== ,(expand q) 1))
            ((== ,(expand q) 2)))))
;; (1 2)
```

3.1.3 Relating Staged and Unstaged Evaluation. A `run` query is equivalent to a `run-staged` query where any subgoal can be annotated as `later`, modulo errors (since the staged interpreter might catch errors early) and modulo a unique answer in the staging run.

3.2 Staged Relational Interpreter

The relation `evalo-staged` provides access to the staged interpreter.

We can use the staged interpreter with the form `run-staged`. For example, we can synthesize the body of a `lambda` expression that, when mapped over the list `(a b c)`, produces the list `((a . a) (b . b) (c . c))`.

```
(run-staged 1 (q)
  (evalo-staged
   `(letrec ((map (lambda (f l)
                    (if (null? l)
                        '()
                        (cons (f (car l))
                              (map f (cdr l)))))))
      (map (lambda (x) ,q) '(a b c)))
    '((a . a) (b . b) (c . c))))
;; ((cons x x))
```

The first answer is the expression `(cons x x)`, corresponding to the `lambda` expression `(lambda (x) (cons x x))`.

This staged query is similar to the unstaged query:

```
(run 1 (q)
  (evalo-unstaged
   `(letrec ((map (lambda (f l)
                    (if (null? l)
                        '()
                        (cons (f (car l))
                              (map f (cdr l)))))))
      (map (lambda (x) ,q) '(a b c)))
    '((a . a) (b . b) (c . c))))
;; ((cons x x))
```

The form `define-staged-relation` runs a first-stage query and evaluates the generated code into a procedure. The defined staged relation behaves like a usual miniKanren relation; in particular, it can be used in a `run` query directly. Like for the form `run-staged`, the first-stage query must produce exactly one answer.

When we define a relation by staged evaluation of the fully-ground syntax of a function, the generated code is plain miniKanren code that does not require the relational interpreter at runtime.

3.2.1 appendo *as a staged relation.*

```
(define-staged-relation (appendo xs ys zs)
  (evalo-staged
   `(letrec ((append
              (lambda (xs ys)
                (if (null? xs)
                    ys
                    (cons (car xs)
                          (append (cdr xs) ys))))))
      (append ',xs ',ys))
   zs))

(run* (q) (appendo '(a b) '(c d e) q))
;; ((a b c d e))
```

```
(run* (x y) (appendo x y '(a b c d e)))
;; (
;;  (() (a b c d e))
;;  ((a) (b c d e))
;;  ((a b) (c d e))
;;  ((a b c) (d e))
;;  ((a b c d) (e))
;;  ((a b c d e) ())
;; )
```

Here is the code generated by the staged interpreter for the staged appendo relation modulo the constraints discussed in Section 4.3.3. The code has some inefficiencies, but no interpretive overhead. The main inefficiencies are that unified variables could be collapsed into one variable, and that some unifications are in unnecessary structures such as `(== (cons _.2 '()) (cons _.3 '()))` instead of `(== _.2 _.3)`.

```
'(lambda (xs ys out)
  (fresh (_.0)
    (== _.0 out)
    (letrec ([append
              (lambda (xs ys)
                (lambda (_.1)
                (fresh (_.2 _.3 _.4 _.5 _.6 _.9 _.7 _.10 _.11 _.13 _.12 _.14 _.8)
                    (== (cons _.2 '()) (cons _.3 '()))
                    (conde
                      ((== '() _.2) (== #t _.4))
                      ((=/= '() _.2) (== #f _.4)))
                    (== xs _.3)
                    (conde
                      ((=/= #f _.4) (== ys _.1))
                      ((== #f _.4)
                       (== (cons _.5 (cons _.6 '()))
                           (cons _.7 (cons _.8 '())))
                       (== (cons _.5 _.6) _.1)
                       (== (cons (cons _.7 _.9) '())
                           (cons _.10 '()))
                       (== xs _.10)
                       (== (cons (cons _.11 _.12) '())
                           (cons _.13 '()))
                       (== xs _.13) (== ys _.14)
                       (callo  append _.8 (cons _.12 (cons _.14 '()))))))))])
      (fresh (_.15 _.16)
        (== xs _.15)
        (== ys _.16)
        (callo append _.0 (cons _.15 (cons _.16 '()))))))))
```

With the original append function, queries could place logic variables in the argument positions to a call in order to generate expressions such that the evaluation produces an expected answer. With appendo we can only generate argument values because the interpreter has been staged away.

However, staged evaluation with `evalo-staged` can handle expression holes in the same manner as `eval-unstaged` by generating code that evaluates the hole using the unstaged interpreter.

```
(run-staged 2 (q)
  (evalo-staged
   `(letrec ((append
              (lambda (xs ys)
                (if (null? xs) ys
                    (cons (car xs) (append (cdr xs) ys))))))
      (append ,q '(3 4)))
   '(1 2 3 4)))
;; ('(1 2) (list 1 2))

(run 2 (q)
  (evalo-unstaged
   `(letrec ((append
              (lambda (xs ys)
                (if (null? xs) ys
                    (cons (car xs) (append (cdr xs) ys))))))
      (append ,q '(3 4)))
   '(1 2 3 4)))
;; ('(1 2) (list 1 2))
```

If we define the staged relation and query as below, we still benefit from the compiled lambda because the unstaged interpreter is aware of the closure values generated by the staged interpreter and can invoke them.

```
(define-staged-relation (context-appendo e res)
  (evalo-staged
   `(letrec ((append
              (lambda (xs ys)
                (if (null? xs)
                    ys
                    (cons (car xs)
                          (append (cdr xs) ys))))))
      ,e)
   res))

(run 2 (q) (context-appendo `(append ,q '(3 4))
                            '(1 2 3 4)))
;; ('(1 2) (list 1 2))
```

3.2.2 Quines with quasiquotes. By staging this example from Byrd et al. [2017], we are able to speed it up significantly. We generate quines with quasiquotes by implementing an interpreter (that understands quasiquotes) on top of the base interpreter.

```
(define-staged-relation (quasi-quine-evalo expr val)
  (evalo-staged
   `(letrec ([eval-quasi (lambda (q eval)
                           (match q
                             [(? symbol? x) x]
```

```
                              [`() '()]
                              [`(,`unquote ,exp) (eval exp)]
                              [`(quasiquote ,datum) 'error]
                              ;; ('error) in the 2017 ICFP Pearl, but
                              ;; the code generator rejects this erroneous code!
                              [`(,a . ,d)
                               (cons (eval-quasi a eval) (eval-quasi d eval))]))]
            [eval-expr
             (lambda (expr env)
               (match expr
                 [`(quote ,datum) datum]
                 [`(lambda (,(? symbol? x)) ,body)
                  (lambda (a)
                    (eval-expr body (lambda (y)
                                      (if (equal? x y)
                                          a
                                          (env y)))))]
                 [(? symbol? x) (env x)]
                 [`(quasiquote ,datum)
                  (eval-quasi datum (lambda (exp) (eval-expr exp env)))]
                 [`(,rator ,rand)
                  ((eval-expr rator env) (eval-expr rand env))]))])
      (eval-expr ',expr (lambda (y) 'error)))
   val))
(run 1 (q)
  (absento 'error q)
  (absento 'closure q)
  (quasi-quine-evalo q q))
;; ((((lambda (_.0) `(,_.0 ',_.0))
;;    '(lambda (_.0) `(,_.0 ',_.0)))
;;   $$
;;   (=/= ((_.0 call)) ((_.0 closure)) ((_.0 dynamic))
;;        ((_.0 error)) ((_.0 prim)))
;;   (sym _.0)))
((lambda (x) `(,x ',x)) '(lambda (x) `(,x ',x)))
;; ((lambda (x) `(,x ',x)) '(lambda (x) `(,x ',x)))
```

3.2.3 Relating Staged and Unstaged Interpreters. Staged execution of the staged evaluator is designed to produce the same answers as normal execution of the unstaged evaluator, modulo errors (since the staged interpreter might catch errors early), modulo answer order, and modulo the representation of closures (containing staged code and no staged code respectively).

```
(run-staged <n> (<q ...>)
  (evalo-staged <expr> <val>))
```

is equivalent to

```
(run <n> (<q ...>)
  (evalo-unstaged <expr> <val>))
```

These two queries both return `((1 2 3 4))`.

```
(run-staged 1 (q)
  (evalo-staged
   `(letrec ((append
              (lambda (xs ys)
                (if (null? xs) ys
                    (cons (car xs) (append (cdr xs) ys))))))
      (append '(1 2) '(3 4)))
   q))

(run 1 (q)
  (evalo-unstaged
   `(letrec ((append
              (lambda (xs ys)
                (if (null? xs) ys
                    (cons (car xs) (append (cdr xs) ys))))))
      (append '(1 2) '(3 4)))
   q))
```

4 IMPLEMENTATIONS

4.1 miniKanren

We use the original implementation of miniKanren [Friedman et al. 2005] as well as the optimized implementation called faster-miniKanren[1]. For our purpose, the difference is not material.

miniKanren threads a store for each potential answer. The store holds the substitution (from unification) as well as the constraint store (from disequality, type constraints, etc.).

4.2 Staged miniKanren

On top of miniKanren, we simply add another field to the store to represent the second-stage code in a potential answer. The `later` construct adds to this stored code. The stored code is reified in a `run` query. We also provide an additional construct `later-scope` to reify the stored code within a scope. The staged interpreter uses the `later-scope` construct to reify the branches of a conditional.

The second-stage code has to be quoted carefully: in unification, we want variables to be unquoted and other terms to be quoted. For example, when unifying a pair, say `(v '(a b))`, we generate the quasiquoted term `(cons v (cons 'a (cons 'b '())))`. This is the role of the `expand` annotations.

4.3 Staged Relational Interpreter

The signature of the recursive relation `eval-expo` for the staged interpreter `evalo-staged` has four parameters `(eval-expo stage? expr env val)`.

The first parameter `stage?` asks whether we are in a staging context, which should be usually true `#t`. In the application case, the parameter `stage?` is set to false `#f`, which in many cases allows the resulting closure value to be known at staging time.

The other parameters, `expr`, `env` and `val`, are the usual parameters of a relational interpreter: an expression `expr` in an environment `env` evaluates to a value `val`.

The guiding principle: everything that is part of the expression `expr` is unstaged (static) while everything that touches the value, the result of evaluation, is staged (dynamic). The environment has a static skeleton and in a variable binding, the variable is static while its value may be dynamic.

Some parts are intricate to stage, like the pattern matcher.

[1]The original repository for faster-miniKanren is https://github.com/michaelballantyne/faster-miniKanren.

The staging of λs and applications deserves mention. For λs, we keep both interpreted and compiled code. When applying a function during staging, we use the interpreted code and stage it, specializing to the argument. First-class λs can be part of the generated code. We use the compiled λ code when invoking it from generated code.

4.3.1 Branches. In the case of an `if` expression, we want to generate a `conde` form. We need to evaluate all the branches to stage their code. We don't want to make straight-line code; we want to make sure the generated code goes in the branch. We also don't want to keep extensions to the staging-time constraint state, because those are specific to each branch.

We can scope code generation with the goal constructor `later-scope` in order to generate code for non-deterministic branches. Execution within a scope separates later-stage code and constraint applications that belong in the branch so they are not included in the outer, unconditional context.

```
(run* (x y)
  (fresh (c1 c2)
    (later-scope (fresh () (l== 5 x) (l== 6 y)) c1)
    (later-scope (fresh () (l== 5 y) (l== 6 x)) c2)
    (later `(conde ,c1 ,c2))))
;; (((_.0 _.1)
;;  !!
;;  ((conde
;;     ((== '5 _.0) (== '6 _.1))
;;     ((== '5 _.1) (== '6 _.0))))))
```

We can now implement the staged interpretation of an `if` expression.

```
(fresh (e1 e2 e3 t c2 c3)
  (== `(if ,e1 ,e2 ,e3) expr)
  (not-in-envo 'if env)
  (eval-expo #t e1 env t)
  (later-scope (eval-expo #t e2 env val) c2)
  (later-scope (eval-expo #t e3 env val) c3)
  (later `(conde
           ((=/= #f ,(expand t)) . ,c2)
           ((== #f ,(expand t)) . ,c3))))
```

4.3.2 Unknowns. The staged interpreter uses the unstaged interpreter to evaluate holes with unknown expressions. Implementing this behavior requires two extensions of standard staging techniques.

(1) Normally the environment in a staged interpreter is static, and not present in the generated code. Here the unstaged interpreter needs the environment to generate expressions to fill holes. When generating calls to the unstaged interpreter, we use the environment as a cross-stage persistent value. We implement cross-stage persistence by generating code that reconstructs the environment at runtime when needed.
(2) When the unstaged evaluator synthesizes a call to a staged procedure defined in the context surrounding a hole, evaluation of the call should use the staged code to avoid interpretive overhead. We extend the unstaged evaluator to understand the representation of staged procedures and invoke their generated code.

A `run-staged` query may include multiple calls to the staged and unstaged evaluators. Depending on the ordering of the calls, an application of an unknown procedure may be evaluated before the λ expression that defines it. This situation may also arise if we introduce optimizations that

reorder clauses, as is done in Barliman[2]. Our current implementation only supports queries that use a single call to the evaluator, but we expect to handle these out-of-order calls using the parallel representations of closures as both data and code discussed above. When an application is reached before a λ, the evaluator can record constraints on the data representation and the evaluation of a staged λ can check compatibility with these constraints.

4.3.3 Constraints. Constraints require cross-stage persistence. In the example below, we need to remember the first-stage `symbolo` constraint on the query variable `q`. The second stage then runs with conflicting constraints. The entire query fails as it should.

```
(run-staged 1 (q)
  (symbolo q)
  (later `(numbero ,q)))
;; ()
```

Reifying constraints within `later-scope`s require effort. Our strategy: for every variable that has had a constraint store entry added or updated within the execution of the `later-scope`, we generate code that re-applies constraints equivalent to the variable's constraint store record. Supporting the intended relationship between staged and unstaged code requires reifying constraints, so that staged and unstaged interpreters behave the same way. Unfortunately, reifying constraints has significant performance and semantic cost. In particular, when turning functions into relations, the relation has the semantics *as if* it were interpreted, that is, the representation of closures as tagged lists leaks, and certain patterns are unnecessarily forbidden. But this behavior matches the unstaged interpreter, so it is the sound approach. An example of the constraints that the interpreter special tag `closure` does not occur in an argument (say `_.0`): `(absento 'closure _.0)`. This constraint comes about because we quote the arguments, say in the definition of `appendo` via `defined-staged-relation` shown in Section 3.2.1. The interpreter prevents quoting from constructing internal representations like closures. We could alternatively use environment extension to pass in the arguments directly, in which case the constraint would not occur. Figure 3 contrasts the performance of these two approaches to passing arguments. Note that this circumvention still matches the behavior of the unstaged interpreter and that other constraints might still occur.

We provide an abstraction for the alternative with environment extension. Manually, instead of writing

```
(define-staged-relation (appendo xs ys zs)
  (evalo-staged
   `(letrec ((append ...))
      (append ',xs ',ys))
   zs))
```

we would write

```
(define-staged-relation (appendo xs ys zs)
  (eval-expo
   #t
   `(letrec ((append ...))
      (append xs ys))
  `((xs . (val . ,xs)) (ys . (val . ,ys)) . ,initial-env)
   zs))
```

[2]Barliman is available at https://github.com/webyrd/Barliman.

appendo in miniKanren	0.047s
environment-passing append in evalo-staged	0.100s
quoted-argument append in evalo-staged	0.187s
quoted-argument append in evalo-unstaged	0.852s
environment-passing append in evalo-unstaged	0.919s

Fig. 3. Microbenchmarks contrasting environment-passing and quoted-argument relational interpreter definitions. The benchmarks runs append backwards with a list of 500 elements.

4.3.4 Determinism. We need to transform control flow of interpreter to make it deterministic. Conceptually, relational programs don't have just one evaluation. Depending on how one annotates a program, one might have infinitely many first-stage solutions. In the case of the interpreter, when evaluating queries without holes, the result is deterministic because the interpreter is syntax-directed. But with holes in the program input to the interpreter, the issue of non-determinism comes back. With logic variables in an expression position, we fall back to interpretation in the staged code, as we explain next.

5 PROGRAM SYNTHESIS BY DEALING WITH HOLES

5.1 Problem

The relation `evalo-staged` expects a ground (static, known) expression and specializes the interpreter with respect to this expression. But what if the expression is not ground? Can we handle holes in the generator? Is it useful?

5.2 Solution

If the expression is not ground, the staged interpreter generates code that falls back to a dynamic interpreter in the second stage. The staged interpreter generates this code in such a way that there is still only one answer and determinism.

The consequence: known parts of program do not incur overhead, and the second stage can focus on the unknown parts, composing the full program.

The known parts can call the unknown parts as needed, but the unknown parts can also call the known parts, thus creating a virtuous cycle of not incurring overhead on known parts. This is achieved by the unstaged interpreter being aware of compiled closures. In particular, this mechanism ensures that the unstaged interpreter calls `letrec`-bound variables using the code without overhead.

5.2.1 Code Walkthrough.

`(varo e)` succeeds if the term `e` is an unground logic variable.
`(non-varo e)` fails if the term `e` is an unground logic variable.

In the recursive relation `eval-expo`, we first check whether the expression `expr` is an unground logic variable `(varo expr)`. If it is, we fallback to the unstaged evaluator `u-eval-expo`.

```
(later `(u-eval-expo ,(expand expr) ,(expand env) ,(expand val)))
```

Calling the function `expand` ensures that the parameters are quoted properly. If the expression `expr` succeeds for the goal `(non-varo expr)`, we proceed with the case analysis `(conde ...)`. For example, the number case generates code to unify the expression `expr` with the value `val`.

```
((numbero expr) (l== expr val))
```

We group all cases where the expression `expr` is a pair, such as special forms `quote`, `lambda` and `if` as well as applications, so that we can ensure determinism in face of logic variables.

While the goals `varo` and `non-varo` are non-relational because they are sensitive to ordering, they essentially capture the idea that we stage when we know and defer when we don't.

5.3 Examples

This mechanism for dealing with holes enables synthesis.

5.3.1 Immediate delegation. These two queries both return Racket expressions that evaluate to (I love staged evaluation)[3]. Though instructive, this example is not so interesting because the staged interpreter delegates immediately to the unstaged one, and so has the same performance.

```
(run-staged 5 (q)
  (evalo-staged
   q
   '(I love staged evaluation)))

(run 5 (q)
  (evalo-unstaged
   q
   '(I love staged evaluation)))

;; ('(I love staged evaluation)
;;   ((car '((I love staged evaluation) . _.0))
;;   $$ (absento (call _.0) (call-code _.0) (closure _.0) (dynamic _.0) (prim _.0)))
;;   (cons 'I '(love staged evaluation))
;;   (((lambda _.0 '(I love staged evaluation)))
;;    $$ (=/= ((_.0 quote))) (sym _.0))
;;   ((letrec ([_.0 (lambda _.1 _.2)])
;;      '(I love staged evaluation))
;;    $$ (=/= ((_.0 quote))) (sym _.1)))
```

5.3.2 Ground context. This example features the Fibonacci function in accumulator-passing style. Numbers are encoded as Peano numerals, where `z` represents zero and `(s . n)` represents the successor of *n*. We give the skeleton of the Fibonacci function. Given the first five Fibonacci numbers, the query synthesizes the initial accumulators `ACC1` and `ACC2` as well as the holes `A`, `B` and `C` representing a base case and the accumulator arguments to the recursive case respectively.

```
(run-staged 1 (fib-acc ACC1 ACC2)
  (fresh (A B C)
    (== `(lambda (n a1 a2)
           (if (zero? n)
               ,A
               (if (zero? (sub1 n))
                   a2
                   (fib-aps (- n '(s . z)) ,B ,C))))
        fib-acc))
  (evalo-staged
   (peano-synth-fib-aps fib-acc ACC1 ACC2)
   '(z
     (s . z)
```

[3]This small synthesis exercise is based on a blog post by Matt Might:https://matt.might.net/articles/i-love-you-in-racket/.

```
     (s . z)
     (s s . z)
     (s s s . z)
     (s s s s s . z))))
;;  (((lambda (n a1 a2)
;;      (if (zero? n)
;;          a1
;;          (if (zero? (sub1 n))
;;              a2
;;              (fib-aps (- n '(s . z)) a2 (+ a1 a2)))))
;;    z
;;    (s . z)))
```

The staged interpreter generates staged code for the ground portions of the program, along with calls to the unstaged interpreter for the unknown holes. The holes themselves must be executed via unstaged relational interpretation in order to synthesize syntax. The staged execution of the ground syntax avoids interpretive overhead, meaning that information from the example calls propagates to the holes more efficiently.

6 EVALUATION

6.1 Benchmarks

We compare the staged interpreter with the unstaged interpreter in Figures 4, 5 and 6.

As part of the advantage of removing the interpretive overhead, the staged interpreter also is oblivious to the branching factor in the cases to consider during interpretation. Indeed the dispatch loop from the staged interpreter disappears, so the cost incurred by adding a new case is only paid at staging time. However, when deferring to the unstaged interpreter, the performance incurs the branching factor.

6.1.1 Parsing with derivatives. The example `parse` in Figure 4 is due to Might et al. [2011][4]. The function `d/dc` takes the Brzozozwksi derivative [Brzozowski 1964] of its first argument, which is a regular expression, with respect to its second argument, which is an atomic symbol. The output of this function is another regular expression, the empty set which represents a failure, or ϵ which represents the empty regular expression. The benchmark `parse (2)` takes the derivative of the regular expression `(foo bar)|(foo baz*)` with respect to `foo`, producing `bar|baz*`.

```
(run #f (parse-result)
  (d/dc-o '(alt (seq foo bar) (seq foo (rep baz))) 'foo parse-result))
;; ((alt bar (rep baz)))
```

The benchmark `parse-backwards (2)` finds a regular expression `regex` whose derivative with respect to `foo` is the regular expression `bar|baz*`; one such regular expression starts with `foo (bar|baz*)`.

```
(run 1 (regex)
  (d/dc-o regex 'foo '(alt bar (rep baz))))
;; (((seq foo (alt bar (rep baz)) . _.0)
;;    $$ ...)
```

[4]Our starting point is the code from Matt Might's blog post:https://matt.might.net/articles/implementation-of-regular-expression-matching-in-scheme-with-derivatives/.

6.1.2 Theorem checker turned prover. The example proofo in Figure 4 is mentioned in Section 1.

```
(define-staged-relation (proofo proof truth)
  (evalo-staged
   `(letrec ([member?
              (lambda (x ls)
                (if (null? ls) #f
                    (if (equal? (car ls) x) #t
                        (member? x (cdr ls)))))]
             [proof?
              (lambda (proof)
                (match proof
                  [`(,A ,assms assumption ()) (member? A assms)]
                  [`(,B ,assms modus-ponens
                        (((,A => ,B) ,assms ,r1 ,ants1)
                         (,A ,assms ,r2 ,ants2)))
                   (and (proof? (list (list A '=> B) assms r1 ants1))
                        (proof? (list A assms r2 ants2)))]
                  [`((,A => ,B) ,assms conditional
                     ((,B (,A . ,assms) ,rule ,ants)))
                   (proof? (list B (cons A assms) rule ants))]))])
      (proof? ',proof))
   truth))

(run 1 (prf)
  (fresh (body)
    (== prf `(C (A (A => B) (B => C)) . ,body))
    (proofo prf #t)))
;; ((C (A (A => B) (B => C))
;;     modus-ponens
;;     (((B => C) (A (A => B) (B => C)) assumption ())
;;       (B (A (A => B) (B => C))
;;          modus-ponens
;;          (((A => B) (A (A => B) (B => C)) assumption ())
;;            (A (A (A => B) (B => C)) assumption ()))))))

(run 1 (prf)
  (fresh (body)
   (== prf `(((A => B) =>
              ((B => C) =>
              ((C => D) =>
              ((D => E) =>
              (A => E))))) () . ,body))
    (proofo prf #t))))
;; ...
```

6.1.3 Negation Normal Form (NNF). This example is adapted from Haskell code by Szeredi et al. [2014].

6.1.4 Peano. This example is due to William E. Byrd and Kanae Tsushima[5]. We use Peano arithmetic to encode numbers and two different versions of Fibonacci, one in direct-style and one in accumulator-passing style (aps). The benchmarks `peano-fib` run `peano-fib-apso` forwards and backwards. The synthesis benchmarks `peano-synth-fib-aps (3)` and `(4)` correspond to the examples in Sections 1 and 5.3.2, respectively.

6.1.5 Synthesis. In Figure 5, the benchmarks `appendo-synth` exercise our synthesis capabilities.

The query `appendo-synth-1` synthesizes a part of the `append` function, the part given by the hole `,q` and the examples given by a list of inputs and a corresponding list of outputs.

The process completes in less than 10ms, while the equivalent unstaged query stalls.

```
(run-staged 1 (q)
  (evalo-staged
   `(letrec ((append
              (lambda (xs ys)
                (if (null? xs) ys
                    (cons ,q (append (cdr xs) ys))))))
      (list (append '() '())
            (append '(a) '(b))
            (append '(c d) '(e f))))
   '(()
     (a b)
     (c d e f))))
;; ((car xs))
```

6.1.6 Double evaluators. Figure 6 shows experiments for various `eval` interpreters within our `evalo` interpreters. The benchmarks mostly generate quines.

6.2 Shortcomings

We cannot run backwards from generated code, so we can't easily get a decompiler.

We only support two stages now, in terms of interface and use cases.

7 RELATED WORK

7.1 Relational Programming

Like here, Lozov et al. [2018] converts from functions to relations albeit with a different mechanism. Within OCanren [Kosarev and Boulytchev 2016], a typed dialect of miniKanren embedded in OCaml, Lozov et al. [2019]; Verbitskaia et al. [2020] have explored optimizations by partial deduction. We go further by having holes in programs and by supporting higher-order patterns.

7.2 Inverse Computation

Abramov and Glück [2001, 2002] study inverse computation. They achieve program inversion via an interpreter, and also remove overhead of one layer of interpretation. In our terminology, they are able to run "backwards" but not with arbitrary holes.

7.3 Multi-Stage Programming

Multi-stage programming systems include MetaML [Taha and Sheard 2000], MetaOCaml [Kiselyov 2014] and Lightweight Modular Staging (LMS) [Rompf and Odersky 2010, 2012] in Scala.

[5]Original code available at https://github.com/k-tsushima/Shin-Barliman/blob/master/transformations/peano.scm.

7.4 Strong Determinism

The Mercury [Somogyi et al. 1996] language sports strong determinism, where queries have bounded numbers of answers checked at compile time. We check whether first-stage queries have a unique answer at run time.

7.5 Partial Evaluation in Logic Programming (Partial Deduction)

The literature on partial evaluation in logic programming (partial deduction) is vast. *The Art of Prolog* [Sterling and Shapiro 1994] provides a good introduction in Chapter 18.

The most closely related works are by Leuschel et al. [2004a,b].

Leuschel et al. [2004a] specialize interpreters with offline partial deduction in the setting of Prolog. Offline partial evaluation acts on an intermediate representation in which annotations serve a similar purpose as in staging.

Leuschel et al. [2004b] adapt the "cogen" approach to Prolog. Previously successful in functional and imperative languages, the "cogen" approach consists in writing a compiler generator instead of a specializer. The compiler generator follows from the third Futamura projection [Futamura 1971, 1999] via a self-applicable specializer. The "cogen" approach stipulates that it is easier to write manually the compiler generator instead of an efficient self-applicable specializer, and that from the user point of view, a compiler generator is as useful regardless of whether it was manually created or derived.

Both these works rely on annotations rather than inference as is traditional in partial evaluation. They have a memoization annotation for handling recursive programs. They also have "binding types" annotations that specify which part of an argument to treat statically vs. dynamically, e.g. the spine of an environment and the variables should be static but the values dynamic.

Our work has a different character, as it is geared towards synthesis. This can be seen in two ways. First, we fallback to dynamic evaluation when they generate multiple answers. In our approach, this enables handling large compiled contexts around a small interpreted hole. Second, we convert from functions to relations.

7.6 Synthesis

Semantics-Guided Synthesis (SemGuS) [Kim et al. 2021] is a new framework for program synthesis with user-defined interpreters and encodings in Constraint Horn Clauses. Like here, a synthesis problem can sometimes be proved impossible — in our case, by failing.

We anticipate that the idea of staging on partially known clauses/expressions can be applied to other synthesis systems. It would be interesting to apply staging to horn clauses in the SemGuS system. The semantics in horn clauses play the role of the interpreter.

Solar-Lezama [2008] coined the term "sketching" for synthesis with holes in the program to be synthesized, which is similar in spirit to our work. A type-driven approach to sketching is also possible [Osera and Zdancewic 2015].

8 FUTURE WORK

8.1 Optimizations

We are naively generating code, we leave it as future orthogonal work to develop optimizers for miniKanren code.

8.2 Reconciling Semantics

Recall from Section 4.3.3. The unstaged interpreter has constraints to forbid the internals of the interpreters from being quoted. The staged interpreter reifies those constraints, generating code

that has different semantics from miniKanren's. For example, in the quoted-argument style, we cannot use an argument with the symbol `closure` because that conflicts with the internals of the interpreter.

Instead of using tagged list to represent interpreter values such as closures, we could use `structs` for internal representation to avoid this problem.

8.3 Polymorphic Staging

As of now, staging the interpreter requires an expert. We would like to support language features that would make the process easier.

We currently have to maintain two versions of the interpreter, staged and unstaged, and manually ensure that they cover the same language. With stage polymorphism, we would maintain only one version and get the same language in both interpreters by construction.

We would like to just write the interpreter once, as a stage polymorphic interpreter, from which we can extract an unstaged interpreter and a staged interpreter linked together. The staged interpreter knows when to fallback to the completely unstaged version, also generated from the same source. The process would not be as much manual work as staging an interpreter from scratch. To find out ways we could improve on the status quo, we reflect on the question: In what ways do the staged interpreter and unstaged interpreter differ?

8.3.1 Determinism. Currently, we have to manually adjust the control flow and add the "cuts" (using the `varo` relation). As future work, we would look for a more automatic approach to support determinism in code generation.

One approach is to figure out where to put the cuts, equivalent to the manually staged interpreter or even more conservative, staging a little bit less. When there are branches and they are not determinate, i.e. not enough information to know which way to go, then we generate later branches. This situation arises in practice in a `conde` form with not enough static information such that only one branch is left when doing top-level unifications. In that case, we would make the `conde` form later stage. Ideally, we would have one big `varo` cut at the top, and this would require coming up with the condition for when to do the cut. We would end up deterministically selecting the right branch in all branches.

Another approach is to design a staging system that allows a finite amount of non-determinism to be unrolled at compile time. We wouldn't be able to just pass the generated code to the state. We would need an alternate evaluation strategy of unfolding a tree. Partial deduction is relevant for unfolding and simplifying an and/or tree.

8.3.2 Abstraction. The way we handle λ and `letrec` abstractions differs in the staged and unstaged interpreters. Ideally, we would have a language feature that enables us to write the same handling in the stage-polymorphic interpreter. In order to not generate the code upfront, we need to use the same generated code for multiple calls. The scheme might involve staging-time memoization.

A key point: in the relational context, relations and code are not first-class objects. Whatever feature we consider should accommodate that fact. In functional staging, we can pass reference to generated (first-class) code. For running with holes, we need to have the data structural and generated code representations simultaneously.

8.4 Collapsing Mixed Towers

Currently, we only collapse one level of interpretation: that of an interpreter staged in miniKanren, such as the `evalo-staged` evaluator written as a miniKanren relation. However, when that evaluator evaluates another evaluator, such as in the quasi-quoted quine examples or the regular expression matcher example, we still have the interpretation overhead of the other evaluator, the user-most

one. In general, if we have n levels of evaluation, we would end up with $n-1$ levels of interpretation. Amin and Rompf [2017] have shown how to collapse towers of interpreters by staging the user-most interpreter in a stage-polymorphic language such as Pink.

We would like to hook our staging framework to Pink and extend Pink with holes. We imagine writing `evalo` for a small language that is enough for bootstrapping. Then we write the other languages that we want synthesis for in that small bootstrapped language. We could implement a JavaScript system on top of Pink, and get a system like that of JavaScript in miniKanren [Chirkov et al. 2020].

9 CONCLUSION

In this paper, we have shown how to stage a relational interpreter, while preserving the interpreter's ability to handle holes anywhere.

Extrapolating from the benchmarks, we can say that large context yields more significant speedups. In parsing with derivatives (Section 6.1.1), we see speedups of 1 or 2 orders of magnitude for a context of 17 helper functions. The ability to handle even larger contexts creates new possibilities for synthesis.

ACKNOWLEDGMENTS

We thank Jason Hemann for insightful discussions, for feedback on drafts and for suggesting Figure 2. We thank Anastasiya Kravchuk-Kirilyuk for feedback on drafts. We thank Laura Zharmukhametova for exploring this project. We thank Greg Rosenblatt for $\lambda_{\uparrow\downarrow}$ snippets. We thank Kaiwen He for pair-programming on the NNF example with William E. Byrd. William E. Byrd wrote the `peano-fib` code while visiting Kanae Tsushima at the National Institute for Informatics (NII) in Tokyo, and also received helpful advice from Youyou Cong of the Tokyo Institute of Technology and Kenichi Asai of Ochanomizu University. Tiark Rompf, along with Nada Amin, originally designed Figure 1 for a tutorial on Lightweight Modular Staging (LMS) at PLDI 2013.

William E. Byrd's work on this publication was supported by the National Center For Advancing Translational Sciences of the National Institutes of Health under Award Number OT2TR003435. The content is solely the responsibility of the authors and does not necessarily represent the official views of the National Institutes of Health. William E. Byrd thanks Matt Might for his leadership and support at the Hugh Kaul Precision Medicine Institute.

REFERENCES

Sergei Abramov and Robert Glück. 2001. From standard to non-standard semantics by semantics modifiers. *International Journal of Foundations of Computer Science* 12, 02 (apr 2001), 171–211. https://doi.org/10.1142/S0129054101000448

Sergei Abramov and Robert Glück. 2002. Principles of inverse computation and the universal resolving algorithm. In *The essence of computation*, Torben Æ. Mogensen, David A. Schmidt, I. Hal Sudborough, Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen (Eds.). Lecture notes in computer science, Vol. 2566. Springer Berlin Heidelberg, Berlin, Heidelberg, 269–295. https://doi.org/10.1007/3-540-36377-7_13

Nada Amin and Tiark Rompf. 2017. Collapsing Towers of Interpreters. *Proc. ACM Program. Lang.* 2, POPL, Article 52 (Dec. 2017), 33 pages. https://doi.org/10.1145/3158140

Janusz A. Brzozowski. 1964. Derivatives of Regular Expressions. *J. ACM* 11, 4 (Oct. 1964), 481–494. https://doi.org/10.1145/321239.321249

William E. Byrd, Michael Ballantyne, Gregory Rosenblatt, and Matthew Might. 2017. A unified approach to solving seven programming problems (functional pearl). *Proceedings of the ACM on Programming Languages* 1, ICFP (2017), 8. https://doi.org/10.1145/3110252

William E. Byrd, Eric Holk, and Daniel P. Friedman. 2012. miniKanren, Live and Untagged: Quine Generation via Relational Interpreters (Programming Pearl). In *Proceedings of the 2012 Annual Workshop on Scheme and Functional Programming* (Copenhagen, Denmark) *(Scheme '12)*. ACM, New York, NY, USA, 8–29. https://doi.org/10.1145/2661103.2661105

Artem Chirkov, Gregory Rosenblatt, Matthew Might, and Lisa Zhang. 2020. A Relational Interpreter for Synthesizing JavaScript. miniKanren Workshop.

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

Daniel P. Friedman, William E. Byrd, Oleg Kiselyov, and Jason Hemann. 2018. *The Reasoned Schemer* (2nd ed.). The MIT Press, Cambridge, MA, USA.

Yoshihiko Futamura. 1971. Partial Evaluation of Computation Process — An approach to a Compiler-Compiler. *Transactions of the Institute of Electronics and Communication Engineers of Japan* 54-C, 8 (1971), 721–728.

Yoshihiko Futamura. 1999. Partial Evaluation of Computation Process, Revisited. *Higher Order Symbol. Comput.* 12, 4 (Dec. 1999), 377–380. https://doi.org/10.1023/A:1010043619517

Jinwoo Kim, Qinheping Hu, Loris D'Antoni, and Thomas Reps. 2021. Semantics-guided synthesis. *Proceedings of the ACM on Programming Languages* 5, POPL (4 jan 2021), 1–32. https://doi.org/10.1145/3434311

Oleg Kiselyov. 2014. The Design and Implementation of BER MetaOCaml. In *Functional and Logic Programming*, Michael Codish and Eijiro Sumii (Eds.). Springer International Publishing, Cham, 86–102.

Dmitry Kosarev and Dmitry Boulytchev. 2016. Typed Embedding of a Relational Language in OCaml. *ACM SIGPLAN Workshop on ML* (2016).

Michael Leuschel, Stephen J. Craig, Maurice Bruynooghe, and Wim Vanhoof. 2004a. Specialising interpreters using offline partial deduction. In *Program development in computational logic*, Maurice Bruynooghe, Kung-Kiu Lau, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Dough Tygar, Moshe Y. Vardi, and Gerhard Weikum (Eds.). Lecture notes in computer science, Vol. 3049. Springer Berlin Heidelberg, Berlin, Heidelberg, 340–375. https://doi.org/10.1007/978-3-540-25951-0_11

Michael Leuschel, Jesper Jørgensen, Wim Vanhoof, and Maurice Bruynooghe. 2004b. Offline specialisation in Prolog using a hand-written compiler generator. *Theory and Practice of Logic Programming* 4, 1-2 (2004), 139–191. https://doi.org/10.1017/S1471068403001662

Petr Lozov, Ekaterina Verbitskaia, and Dmitry Boulytchev. 2019. Relational Interpreters for Search Problems. miniKanren Workshop.

Petr Lozov, Andrei Vyatkin, and Dmitry Boulytchev. 2018. Typed Relational Conversion. In *Trends in Functional Programming*, Meng Wang and Scott Owens (Eds.). Springer International Publishing, Cham, 39–58.

Matthew Might, David Darais, and Daniel Spiewak. 2011. Parsing with Derivatives: A Functional Pearl. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming* (Tokyo, Japan) *(ICFP '11)*. Association for Computing Machinery, New York, NY, USA, 189–195. https://doi.org/10.1145/2034773.2034801

Peter-Michael Osera and Steve Zdancewic. 2015. Type-and-Example-Directed Program Synthesis. *SIGPLAN Not.* 50, 6 (June 2015), 619–630. https://doi.org/10.1145/2813885.2738007

Tiark Rompf and Martin Odersky. 2010. Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs. In *Proceedings of the Ninth International Conference on Generative Programming and Component Engineering* (Eindhoven, The Netherlands) *(GPCE '10)*. Association for Computing Machinery, New York, NY, USA, 127–136. https://doi.org/10.1145/1868294.1868314

Tiark Rompf and Martin Odersky. 2012. Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs. *Commun. ACM* 55, 6 (June 2012), 121–130. https://doi.org/10.1145/2184319.2184345

Armando Solar-Lezama. 2008. *Program Synthesis by Sketching.* Ph.D. Dissertation. USA. Advisor(s) Bodik, Rastislav. AAI3353225.

Zoltan Somogyi, Fergus Henderson, and Thomas Conway. 1996. The execution algorithm of mercury, an efficient purely declarative logic programming language. *The Journal of Logic Programming* 29, 1 (1996), 17–64. https://doi.org/10.1016/S0743-1066(96)00068-4 High-Performance Implementations of Logic Programming Systems.

Leon Sterling and Ehud Shapiro. 1994. *The Art of Prolog (2nd Ed.): Advanced Programming Techniques.* MIT Press, Cambridge, MA, USA.

Péter Szeredi, Gergely Lukácsy, Tamás Benkő, and Zsolt Nagy. 2014. *The Semantic Web Explained: The Technology and Mathematics behind Web 3.0.* Cambridge University Press. https://doi.org/10.1017/CBO9781139194129

Walid Taha and Tim Sheard. 2000. MetaML and multi-stage programming with explicit annotations. *Theoretical Computer Science* 248, 1 (2000), 211–242. https://doi.org/10.1016/S0304-3975(00)00053-0 PEPM'97.

Ekaterina Verbitskaia, Danil Berezun, and Dmitry Boulytchev. 2020. An Empirical Study of Partial Deduction for miniKanren. miniKanren Workshop.

name	*staging*	*staged*	*run-staged*	*unstaged*	*gain*
parse	0.19002s				
parse (0)		0.00007s	0.20924s	0.00578s	77.731
parse (1)		0.00036s	0.18282s	0.03354s	94.128
parse (2)		0.00118s	0.19385s	0.24916s	211.839
parse (3)		0.00189s	0.19118s	0.16630s	87.797
parse (4)		0.00163s	0.18780s	0.14037s	86.003
parse (5)		0.00394s	0.21179s	0.45512s	115.535
parse-backwards (0)		0.03071s	0.22626s	3.50092s	114.014
parse-backwards (1)		0.01161s	0.21151s	0.27018s	23.272
parse-backwards (2)		11.60745s	12.75373s	[> 5 min]	[>25.845]
proofo	0.27211s				
proofo (1)		0.00930s	0.31692s	0.08064s	8.670
proofo (2)		0.25494s	0.54561s	1.40913s	5.527
proofo (3)		4.90357s	4.93577s	[> 5 min]	[>61.180]
nnf	0.26296s				
nnf (0)		0.25689s	0.44367s	6.17420s	24.034
nnf (1)		0.00026s	0.22781s	0.00173s	6.722
nnf (2)		0.00022s	0.22804s	0.00165s	7.487
nnf (3)		0.00027s	0.23962s	0.00217s	8.044
peano-synth-fib-direct (1)			0.01436s	0.09984s	6.951
peano-synth-fib-aps	0.03218s				
peano-synth-fib-aps (0)			0.01898s	0.06448s	3.397
peano-synth-fib-aps (1)			0.02445s	[> 5 min]	[>12268.943]
peano-synth-fib-aps (2)		0.00450s	0.03454s	1.40561s	312.062
peano-synth-fib-aps (3)		0.48442s	0.49497s	[> 5 min]	[>619.292]
peano-synth-fib-aps (4)			1.54791s	[> 5 min]	[>193.810]
peano-fib	0.01087s				
peano-fib (1)		0.00093s	0.00994s	0.03133s	33.829
peano-fib (2)		0.00230s	0.01215s	0.05963s	25.946
peano-fib (3)		0.00278s	0.01351s	0.07891s	28.360
peano-fib (4)			1.37853s	9.26185s	6.719

Fig. 4. Benchmarks contrasting staged and unstaged computation for turning functions into relations. The column *staging* represents the time to stage the program, comprising the time to generate it in miniKanren and evaluate it in Scheme. The column *staged* represents the time to run the staged program. The column *run-staged* is used when staging and running staged are combined. The column *unstaged* represents the time to run a version of the program without staging. The column *gain* calculates the ratio of *unstaged* over *staged* or *run-staged*, whichever is smaller when both are present. The *unstaged* numbers are marked [> 5 min] have timed out in which case the gain is over the 5 minutes.

name	*staging*	*staged*	*run-staged*	*unstaged*	*gain*
appendo-synth-0			0.40287s	1.04528s	2.595
appendo-synth-0b			7.12135s	[> 5 min]	[>42.127]
appendo-synth-0d			0.43379s	1.01420s	2.338
appendo-synth-1			0.00783s	[> 5 min]	[>38331.864]
appendo-synth-2			4.45163s	[> 5 min]	[>67.391]
appendo-synth-2b			0.05981s	[> 5 min]	[>5015.695]
appendo-tail			1.74004s	13.82629s	7.946
appendo-tail-quoted			0.00333s	0.00098s	0.295
map-hole (0)			0.00918s	0.00527s	0.574
map-hole (1)			2.47672s	1.99132s	0.804

Fig. 5. Benchmarks contrasting staged and unstaged computation for miscellaneous experiments. See Figure 4 for legend.

name	*staging*	*staged*	*run-staged*	*unstaged*	*gain*
eval-and-map-evalo	0.14968s	33.77272s		58.74909s	1.740
eval-and-map-and-list-evalo	0.19766s	31.91443s		23.99028s	0.752
quasi-quine-evalo	0.09353s	16.55327s		157.63767s	9.523
ho-quine-interp-cons	0.07264s	44.44917s		45.83638s	1.031
ho-double-evalo	0.08090s	0.24508s		2.83638s	11.573
map-in-double-eval	0.12170s	0.50420s		[> 5 min]	[>595.006]
double-evalo	0.06570s	0.15523s		3.31750s	21.371
double-evalo-variadic-list-fo	0.07264s	0.50668s		3.49524s	6.898
double-evalo-variadic-list-fo-better	0.06470s	0.35746s		2.82351s	7.899
double-evalo-variadic-list-ho	0.08495s	0.46386s		2.90831s	6.270
double-evalo-cons	0.06247s	2.60720s		18.74904s	7.191

Fig. 6. Benchmarks contrasting staged and unstaged computation for various `eval` interpreters within our `evalo` interpreter. The prefix ho stands for higher-order. See Figure 4 for legend.