Multi-stage Relational Programming

MICHAEL BALLANTYNE, Northeastern University, USA
RAFAELLO SANNA, Harvard University, USA
JASON HEMANN, Seton Hall University, USA
WILLIAM E. BYRD, University of Alabama at Birmingham, USA
NADA AMIN, Harvard University, USA

We transport multi-stage programming from functional to relational programming, with novel constructs to give programmers control over staging and non-determinism. We stage interpreters written as relations, in which the programs under interpretation can contain holes representing unknown expressions or values. By compiling the known parts without interpretive overhead and deferring interpretation to run time only for the unknown parts, we compound the benefits of staging (e.g., turning interpreters into compilers) and relational interpretation (e.g., turning functions into relations and synthesizing from sketches). We extend miniKanren with staging constructs and apply the resulting multi-stage language to relational interpreters for subsets of Racket and miniKanren as well as a relational recognizer for context-free grammars. We demonstrate significant performance gains across multiple synthesis problems, systematically comparing unstaged and staged computation, as well as indicatively comparing with an existing hand-tuned relational interpreter.

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

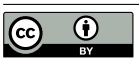
Additional Key Words and Phrases: relational programming, staging, synthesis, miniKanren, Racket, Scheme

ACM Reference Format:
Michael Ballantyne, Rafaello Sanna, Jason Hemann, William E. Byrd, and Nada Amin. 2025. Multi-stage Relational Programming. *Proc. ACM Program. Lang.* 9, PLDI, Article 211 (June 2025), 25 pages. https://doi.org/10.1145/3729314

1 Introduction

This paper concerns extending the techniques of staged functional programming to the relational programming context. Relational programming—pure, all-modes constraint logic programming—provides concise uniform solutions to problems across a wide number of areas. Figure 1 shows two illustrative examples. In the first example, we synthesize part of the append function from a sketch [Solar-Lezama 2009] and some examples of its behavior. In the second, we invert [Harrison and Khoshnevisan 1992] the behavior of append and derive the set of ways to split the given list. The uniform solution for these two example tasks and a host of others [Byrd et al. 2017] hinges on using an interpreter implemented as a relation [Byrd et al. 2012] in a relational programming language. The price of interpretation is interpretive overhead, and presently, the performance penalty of interpretation makes the relational interpreter approach impractical for most applications.

Authors' Contact Information: Michael Ballantyne, Northeastern University, Boston, USA, ballantyne.m@northeastern.edu; Rafaello Sanna, Harvard University, Cambridge, USA, rsanna@g.harvard.edu; Jason Hemann, Seton Hall University, South Orange, USA, hemannja@shu.edu; William E. Byrd, University of Alabama at Birmingham, Birmingham, USA, webyrd@uab.edu; Nada Amin, Harvard University, Cambridge, USA, namin@seas.harvard.edu.



ACM 2475-1421/2025/6-ART211
https://doi.org/10.1145/3729314

```
(synth/sketch (e)
  ([append
    (lambda (xs ys)
      (if (null? xs)
          ys
          (cons
           ,e
           (append (cdr xs) ys))))])
 [(append '() '()) -> ()]
 [(append '(a) '(b)) -> (a b)]
 [(append '(c d) '(e f)) -> (c d e f)])
↪
(car xs)
```

(a) Example-based synthesis from a program sketch. The first part sketches the append body with `,e` creating a hole. The remainder provides example invocations and expected outputs.

```
(invert-execute
  ([append
    (lambda (xs ys)
      (if (null? xs)
          ys
          (cons
           (car xs)
           (append (cdr xs) ys))))])
  (a b c))
↪
(() (a b c))
((a) (b c))
((a b) (c))
((a b c) ())
```

(b) Function inversion generates the set of inputs that lead to an output. Here, we compute values of `xs` and `ys` so that `(append xs ys)` yields the list `(a b c)`

Fig. 1. Two examples of tasks the relational interpreter technique enables.

In the functional-programming world, *staging* [Jørring and Scherlis 1986; Taha 1999] is an effective technique for eliminating interpretive overhead. Stratifying the computation into a sequence of stages permits pre-computing earlier stages and thereby generating a residual program that may run faster than the original. A multi-stage program is a conventional program plus staging annotations that declare which program fragments should wait to be computed in a later stage.

What's more, a multi-stage program can be instead understood as an ordinary single-stage program by ignoring the staging annotations. This erasure property [Inoue and Taha 2016] distinguishes multi-stage programming from other program generation techniques, and offers a unique advantage for programmers. With staging, program generators can often be derived simply by starting with a correct but slow unstaged program and adding annotations. A programmer can subsequently reason about a multi-stage program more easily by ignoring the staging annotations and thinking about it as a single-stage program.

Combining staging with the relational programming model, however, requires grappling with three challenges—non-determinism, pervasive partially-unknown data, and lazily checked computations of constraints—that do not arise in the simpler functional programming context. Those three facets are integral to the relational computation model: constraint-logic programming languages compute through accumulating information about unknowns (logic variables) and rely on non-deterministic search to compute solutions to a query according to equations specified in the program. All three features interact with staging and require new functionality in a staged relational programming language.

This paper generalizes functional program staging to the context of all-modes relational programming, in order to speed up relational interpreter-based programming tasks. In doing so we solve the non-trivial task of reconciling staging with those three challenging features. Concretely, we make the following contributions:

- We present a design (Section 3) and operational semantics (Section 4) for *multi-stage miniKanren*, a staged relational programming language. Its design extends staging to accommodate the unique features of relational programming. New kinds of staging annotations allow programmers to control the effect of staging-time nondeterminism. Logic variables may

appear anywhere, so data that a staged program expects to know at staging time may in fact only be available at run time. Execution falls back to run-time in the case that the data are in fact dynamic. Logic variables in terms may have associated constraints. Multi-stage miniKanren properly residualizes any such constraints attached to cross-stage persistent values [Hanada and Igarashi 2014; Taha and Sheard 2000].
- We state a multi-stage programming erasure property for the relational programming context (Section 4.1)
- We use multi-stage miniKanren to stage a relational interpreter and other similar programs such as a relational recognizer for context-free grammars. We apply the staged relational interpreter to accelerate program inversion and synthesis from sketches (Section 5).
- We demonstrate that our technique achieves orders-of-magnitude performance improvements as compared to unstaged relational programs (Section 6).

We assume passing familiarity with Scheme and the concept of functional program staging.

2 Background

Our work lies at the intersection of multi-stage programming and relational programming. The following subsections review the necessary essentials of both areas through some concrete examples.

2.1 Relational Programming in miniKanren

Relational programming uses equations and constraints to compute relations. An *n*-ary miniKanren relation computes a subset of the *n*-fold product of the set of miniKanren *terms*—finite `cons`-based binary trees over an infinite set of atomic terms. The miniKanren [Friedman et al. 2005, 2018] language is a common substrate for relational programming. A miniKanren program consists of a set of relations defined with `defrel` and a query written with `run`. This simple relation and example queries shows the basics of miniKanren programming and its all-modes behavior:

```
(defrel (same a b)
  (== a b))
(run 1 (p q) (same 'dog p))
↪ ((dog _.0))
(run 1 (q) (== 5 6))
↪ ()
```

```
(run 1 (p q) (same (cons 'dog p) (cons p q)))
↪ ((dog dog))
(run 1 (p q) (same (cons p (cons p 'dog)) q))
↪ ((_.0 (_.0 _.0 . dog)))
```

The `defrel` form introduces a relation named `same` with parameters `a` and `b`, and its body is a program that computes this relation. Here, that computation consists of a single *goal* that asserts `a` and `b` describe the same tree. Equations are written with the `==` operator. Operationally, the `==` binary goal constructor unifies its arguments. Disequalities are written using the `=/=` operator, and the language has several other basic constraints as well.

Having defined a set of relations, the programmer uses the `run` operator to query against this set of relation definitions for satisfying terms. The first `run` asks for at most one solution of assignments to variables `p` and `q` so that `p` is the same as the quoted symbol constant `'dog`. The user can write queries with respect to one or more variables (in this example the variables `p` and `q`) and miniKanren expresses the answers with respect to these variables. Here we see miniKanren produces a list that contains the single solution, `(dog _.0)`. This solution is itself a list, containing a value for each queried variable. In this case the variable `p` is assigned the term `dog`, and `q`'s value is unknown. In general, something that looks like `_.`*n* represents an unknown value. Rather than one single concrete answer with a value for `p` and a value for `q`, miniKanren returns a solution that describes an infinite set of answers; every value we could assign `q` would also yield a distinct answer. The

$$
\begin{array}{lll}
e & \in \mathit{Expr} ::= b \mid x \mid (\lambda(x)\ e) \mid (\texttt{or}\ e_1\ e_2) \mid (e_1\ e_2) \\
v & \in \mathit{Value} ::= b \mid \texttt{closure}(x, e, \rho) \\
x & \in \mathit{Var} \\
b & \in \mathit{Boolean} ::= \texttt{true} \mid \texttt{false}
\end{array}
$$

$$\mathit{make\text{-}clos}(x, e, \rho) \quad = \texttt{closure}(x, e, \rho)$$

$$\frac{\rho, x : v_1 \vdash e \Rightarrow v}{\mathit{apply\text{-}clos}(\texttt{closure}(x, e, \rho), v_1) \Rightarrow v}\ \textsc{Closure}$$

$$\frac{}{\rho \vdash b \Rightarrow b}\ \textsc{const}$$

$$\frac{\rho \vdash e_1 \Rightarrow v_1 \qquad v_1 \equiv \texttt{false} \qquad \rho \vdash e_2 \Rightarrow v}{\rho \vdash (\texttt{or}\ e_1\ e_2) \Rightarrow v}\ \textsc{or-1}$$

$$\frac{\rho \vdash e_1 \Rightarrow v \qquad v \not\equiv \texttt{false}}{\rho \vdash (\texttt{or}\ e_1\ e_2) \Rightarrow v}\ \textsc{or-2}$$

$$\frac{\rho(x) = v}{\rho \vdash x \Rightarrow v}\ \textsc{ref}$$

$$\frac{}{\rho \vdash (\lambda(x)\ e) \Rightarrow \mathit{make\text{-}clos}(x, e, \rho)}\ \textsc{abs}$$

$$\frac{\rho \vdash e_1 \Rightarrow v_1 \qquad \rho \vdash e_2 \Rightarrow v_2 \qquad \mathit{apply\text{-}clos}(v_1, v_2) \Rightarrow v}{\rho \vdash (e_1\ e_2) \Rightarrow v}\ \textsc{app}$$

(a) Big-step semantics.

```
(defrel (make-closo x e env clos)
  (== clos `(clos ,x ,e ,env)))

(defrel (apply-closo clos v1 v)
  (fresh (x e env)
    (== clos `(clos ,x ,e ,env))
    (evalo-or e `((,x . ,v1) . ,env) v)))

(defrel (evalo-or e env v)
  (conde
    [(booleano e) (== e v)]
    [(fresh (e1 e2 v1)
       (== e `(or ,e1 ,e2))
       (evalo-or e1 env v1)
         (conde
           [(== v1 #f) (evalo-or e2 env v)]
           [(=/= v1 #f) (== v1 v)]))]
    [(symbolo e) (lookupo e env v)]
    [(fresh (x e1)
       (== e `(lambda (,x) ,e1))
       (symbolo x)
       (make-closo x e1 env v))]
    [(fresh (e1 e2 v1 v2)
       (== e `(,e1 ,e2))
       (evalo-or e1 env v1)
       (evalo-or e2 env v2)
       (apply-closo v1 v2 v))]))
```

(b) A relational interpreter for a small language. It relates expressions, environments, and values

Fig. 2. Big-step semantics for λ-or and a corresponding miniKanren relation.

result of a `run` query is always a list of answers; the list is empty if the query fails (e.g., in the second query terms `5` and `6` will fail to unify).

Even short queries can lead to some non-trivial constraints with partially ground data. In the third query, we unify two *partially ground terms*. These represent two trees where some of the tree structure is known and fixed, but other portions are unknown, indicated by these *logic variables*. The first term is a pair (built with `cons`) where the left-branch is an atom `dog` and the right branch is the unknown `p`. The structure of the second term is also only partially known; it must be at least a pair, but its left child `p` and its right child `q` are unknown. Satisfying that equality constraint on these trees forces assignments of variables that `q` and `p` must both be `dog`.

Unification produces a most general solution by only assigning variables as concrete a value as is required. The last query dictates that the variable `q` must be the same as some term `(cons p (cons p 'dog))`. Since there are no further restrictions on `p`, however, `q`'s value has no more known structure than that. The two uses of `p` in the first tree of that query lead to the repetition of `_.0` in the solution—although unconstrained to what term it is, all valid assignments for `q` demand the same tree occur in both positions.

Computing more complex relations and asking more complex queries require using a few additional syntactic forms—namely `fresh` and `conde`—and several additional constraints. Figure 2b

```
(run 1 (e)
  (evalo
   `(letrec
      ([append
        (lambda (xs ys)
          (if (null? xs) ys
              (cons ,e
                    (append (cdr xs) ys))))])
      (list
       (append '() '())
       (append '(a) '(b))
       (append '(c d) '(e f))))
   initial-env
   '(() (a b) (c d e f))))
↪
((car xs))
```

```
(defrel (appendo xs ys zs)
  (evalo
   `(letrec
      ([append
        (lambda (xs ys)
          (if (null? xs) ys
              (cons (car xs)
                    (append (cdr xs) ys))))])
      (append ',xs ',ys))
   initial-env
   zs))

(run* (xs ys) (appendo xs ys '(a b c))
↪
(() (a b c))
((a) (b c))
((a b) (c))
((a b c) ())
```

Fig. 3. Relational interpreter based implementations of the tasks in Figure 1.

introduces `evalo-or`, a relational interpreter for a small lambda-calculus based language with booleans and `or`-expressions. We use the code in this figure both to introduce new miniKanren forms and to illustrate a relational interpreter. It relies on two helper relations, `make-closo` and `apply-closo`, and an elided relation `lookupo`. On the left-hand side of the figure, we show a big-step semantics for the same language.

The `quasiquote` (`` ` ``) and `unquote` (`,`) syntax is merely convenient shorthand for building term structures containing logic variables at certain positions. The second relation, `apply-closo` demonstrates the syntactic form `fresh`. The `fresh` form introduces new auxiliary logic variables scoped over the goals in the body of the form. A goal created with `fresh` succeeds whenever the conjunction of its body goals succeed. The `apply-closo` relation introduces `x`, `e`, and `env`, and succeeds when both the variable `clos` can take the form of a four-element list beginning with the symbol `clos` and also when the second goal succeeds. Relation definitions can contain calls to themselves or to other relations. In the final line of the `apply-closo` definition, the second argument in the call to `evalo` is an extended environment, represented in a first-order fashion as a list of pairs. The call to `evalo-or` takes place against a pair of variables `x` and `a` added to the front of `env`.

The `evalo-or` relation connects three terms when the first has the shape of an expression and the second has the shape of an environment in which the expression evaluates to the third element, the value. The body of `evalo-or` is a `conde` expression. The `conde` form expresses disjunction of its clauses. Each clause expresses the conjunction of its goals. The first clause relies on a `booleano` constraint. The miniKanren language implements a small variety of type constraints, which are lazily enforced as logic variables are unified with values.

2.1.1 Synthesis with Relational Interpreters. The `evalo-or` interpreter only supports a small language, but it is sufficient to show how relational evaluation can accomplish synthesis. This query simultaneously infers that the only sensible completion for the `p` query variable is `lambda` and evaluates the application to the value `#t`:

```
(run 1 (p q) (evalo-or `((,p (x) x) #t) '() q)) ↪ ((lambda #t))
```

Previous work by Byrd et al. [2017, 2012] explores the use of relational interpreters for somewhat larger functional languages to accomplish a surprising variety of synthesis tasks. Figure 3 shows how the example-based synthesis and program inversion examples from Figure 1 can be accomplished

```
type orval =
  | Bool of bool
  | Closure of (orval -> orval);;

type orexp =
  | Lit of bool
  | Or of orexp * orexp
  | Sym of string
  | Lam of string * orexp
  | App of orexp * orexp ;;

type env = (string * orval code) list
```

```
let rec eval_or (e : orexp) (env : env) : orval code =
  match e with
  | Lit b -> .< Bool b >.
  | Or (e1, e2) ->
    .< (match .~(eval_or e1 env) with
        | Bool false -> .~(eval_or e2 env)
        | v1 -> v1) >.
  | Sym s -> List.assoc s env
  | Lam (s, e) ->
    .< Closure
       (fun v -> .~(eval_or e ((s, .<v>.) :: env))) >.
  | App (e1, e2) ->
    .< (match .~(eval_or e1 env) with
        | Closure f -> f .~(eval_or e2 env)) >.
```

Fig. 4. A MetaOCaml implementation of a staged interpreter for the language from Figure 2.

using the larger `evalo` relation provided by Byrd et al. From the perspective of miniKanren, the only difference between these tasks is the position of the variables in the argument to `eval`. For the synthesis task, we place the query variable `e` in the text of the `append` body; for program inversion, the query variables `xs` and `ys` stand for arguments of a call to `append`.

The `appendo` relation holds between `xs`, `ys`, and `zs` when the interpreter relates the `letrec` expression, including the call to the `append` function on values `xs` and `ys` in the body, to a value `zs`. As the query result indicates, this indeed defines the relationship between lists to append and their result. The binding expression in the `letrec` argument to `evalo` is the function `append` written in the interpreter's subset of Racket. Instead of writing the `appendo` miniKanren relation on terms in a direct style, however, the example in Figure 3 defines the relationship at a meta-level. Byrd et al. [2017] introduce this trick of using a function to define the behavior of a relation through a relational interpreter; this `append`/`appendo` is due to them. The relational interpreter-based synthesis technique is expressive, but in the absence of staging, using the relational interpreter to query Racket functions in this way suffers from interpretive overhead. Let us turn to staging, which offers a strategy to reduce this cost.

2.2 Program Staging

We use MetaOCaml [Calcagno et al. 2003; Kiselyov 2014] to re-introduce traditional multi-stage functional programming as background material. Figure 4 contains an interpreter for the same language as in Figure 2, implemented as a staged function in MetaOCaml.

The type `orexp` of the argument `e` shows that the expression is staging-time. The return value is wrapped in `code`, which means that this program generates code that will, at run time, produce an `orval`. The environment argument is mixed-stage data: its keys come at staging time but its values at run time. The matching on `eval_or`'s input expression happens at staging time, outside of any quotation (`.< ... >.`). Any matching or construction of values happens inside the quotations, because we are generating code to execute at run time.

Recursive calls to the evaluator are unquoted (`.~`). The upshot of this is that we will splice the code generated by the recursion into the surrounding quoted code. Environment lookup happens at staging time; the result of the environment lookup will be the `orval code` stored in the environment, which will splice into the generated code in the context where the lookup occurs.

3 Introduction to Multi-stage miniKanren

As with traditional two-stage functional programming, staged relational programming separates goal execution into a *staging-time stage* that generates goal code for run time, and a *run-time stage* in

which those generated goals, together with goals not executed at staging time, are subsequently run in a query. In this model the two stages are separate and disjoint, with the staging-time component wholly preceding the run-time component. Portions of code for each stage can be inter-nested, and code executed at run time can contain fragments generated at staging time within it.

The goal annotation `staged` promotes a goal from run time to staging time, and the annotation `later` correspondingly demotes a goal at staging time to a run-time goal, deferring its computation. The query in Figure 5 demonstrates both of these annotations in context. By itself the query is not

```
(run 1 (q)
  (staged
    (fresh (p r)
      (== q (list p r))
      (== p 'dog)
      (later (== r 'fish)))))
```

```
⇓ generated code (simplified)
(run 1 (q)
  (fresh (r)
    (== q (list 'dog r))
    (== r 'fish)))
↪ ((dog fish))  generated query's result
```

Fig. 5. Staging time and deferring to run time.

particularly interesting; the lone answer is the list `(dog fish)`. Worth noticing however is the stage at which each subgoal executes. Computation of the `fresh` goal will be performed during staging because the goal is within a `staged` annotation. The `(== r 'fish)` unification executes at run time because, while the `staged` annotation promotes the `fresh` goal to staging time, the `later` annotation demotes its contents from staging time to run time.

A staging time computation produces code as its value. The run-time state is by definition not yet available during staging time—in Figure 5 this includes the values of `q` and `r`. Therefore, sometimes even some of the computation within a `staged` block must be deferred to run time. Other portions of the computation, however, can be pre-computed during staging time and doing so simplifies the residual task. Staging time computation simplifies the `fresh` goal, for example, by integrating the value of `p` into `(== q (list 'dog r))`. This residual goal expresses solutions to the original equations, and makes no reference to variable `p`. From the perspective of logic programming, the staging simplifies this constraint problem and residualizes the remainder. From the implementer's perspective as a staging-aware functional programmer, this is persisting values across staging—specifically constraints, which can be equational or otherwise. In this case multi-stage miniKanren can optimize away the introduction of variable `p`. Any of the goals in the body of the `fresh` can be re-ordered, and the staging time computation would produce the same result, because miniKanren is a relational programming language.

Like any other goal, a relation call can be executed at staging time—provided the relation definition is itself defined and available at staging time. A relation is defined for staging time with `defrel/staged`, for example:

```
(defrel/staged (pet q)
  (== q 'dog))
(run 1 (q) (staged (pet q)))
```

```
;; using pet in both stages
(run 1 (q) (staged (pet q)) (pet q))
```

The form `defrel/staged` defines a multi-stage relation. A use of this form creates at once both a staged and a dynamic variant of the relation. The body of the static variant of a multi-stage relation definition is a staging-time goal. The dynamic variant of the relation is created by erasing the staging annotations from this body goal. The goal in the query resembles one of those from Figure 5, except that the constraint is now indirectly stated through the relation `pet`. The second query demonstrates this cross-stage persistence of the relation definition with a run-time relation call to `pet`. A relation defined for staging time will persist into the run-time stage.

Termination. A key requirement of staging, carried over from traditional functional multi-stage programming is that the staging-time computation should terminate. The task of manually annotating the program puts the programmer in charge of determining where staging-time computation begins and ends, and is what separates staging from partial evaluation more generally. Multi-stage miniKanren programmers use the `later` annotation as needed to defer computations to runtime in order to ensure termination at staging time.

Nondeterminism. Relational computations are in general non-deterministic, and the novel annotations in the multi-stage miniKanren system let the programmer specify how staging should behave in the face of non-determinism. What's more, a relation's computation can be deterministic or non-deterministic, depending on the values of the arguments in the call (see example in Figure 6).

```
(defrel/staged (noto p q)
  (conde
    [(== p #t) (== q #f)]
    [(== p #f) (== q #t)]))
```

```
(run* (p q) (fresh (x) (noto p x) (noto #f q)))
↪ ((#t #t) (#f #t))
(run 1 (p q) (staged (noto p q)))
;; error due to non-determinism
```

Fig. 6. Staging-time evaluation must produce a single result, unlike a run-time query.

The aim of staging is to produce a single residual program, and thus staging-time evaluation must only produce a single result. The programmer staging the `noto` relation needs to resolve staging-time non-determinism. Multi-stage miniKanren provides two ways to defer staging-time non-determinism to run-time: the `gather` and `fallback` annotations. Both forms should syntactically contain goals, and they indicate two different approaches as to how staging should handle the nondeterminism of their contents. If there are multiple successful branches at staging time, do we want to generate specialized code for each branch (`gather`), or do we want to fall back to un-specialized, dynamic code (`fallback`)? The queries in Figure 7 demonstrate these two different annotations' consequences on the generated code.

```
(defrel/staged (noto/gather p q)
  (gather
    (conde
      [(== p #t) (== q #f)]
      [(== p #f) (== q #t)])))

(defrel/staged (noto/fallback p q)
  (fallback
    (conde
      [(== p #t) (== q #f)]
      [(== p #f) (== q #t)])))
```

```
(run* (p q)
  (staged
    (fresh (x)
      (noto/gather #f q)
      (noto/gather p x))))
⇓ generated code (simplified)
(run* (p q)
  (fresh (x)
    (== #t q)
    (conde
      [(== p #t) (== x #f)]
      [(== p #f) (== x #t)])))
```

```
(run* (p q)
  (staged
    (fresh (x)
      (noto/fallback #f q)
      (noto/fallback p x))))
⇓ generated code (simplified)
(run* (p q)
  (fresh (x)
    (== #t q)
    (invoke-fallback
      noto/fallback/1 p x)))
```

Fig. 7. Two approaches to staging non-determinism in the `noto` relation.

The `noto/gather` and `noto/fallback` are exactly the `noto` of Figure 6, except with annotations around the `conde` in their bodies. The annotations only affect code generation when non-determinism arises at staging time. In each example, the second call to `noto` is deterministic because the statically-known `#f` determines which branch to follow. As such, the generated code in both queries contains an unconditional unification of the variable with `#t`.

When a staging-time use of the relation produces non-deterministic behavior, however, the two annotations produce significantly different generated code. The first call to `noto` in the two queries of Figure 7 illustrate the two possibilities. In such cases, annotating with `gather` causes

staging to generate non-deterministic code. That non-deterministic code can however, benefit from improvements made during the staging process. Annotating with `fallback`, however, will defer the computation of the entire annotated goal to run-time. Unlike with `later`, `fallback` only has this effect when a staging-time use of the relation produces non-deterministic behavior. The `noto/fallback-1` referred to in the generated code is a run-time relation equivalent to `noto` of Figure 6. Our multi-stage miniKanren generates the relation when `noto/fallback` is defined, by simply erasing the staging annotations from the body of the `fallback` form. Each `fallback` form in a given relation produces a different entry point into run-time code. The `1` in `noto/fallback-1` indicates this is the first (but in this example the only) fallback form.

The `gather` and `fallback` annotations each offer a way to make deterministic what would otherwise be a non-deterministic staging-time use of a relation. When we expect to get non-determinism at run-time we use `gather`. However, in other situations we may not have non-determinism in the dispatch. It might just be that some variable whose value we had expected to know at staging time is still unavailable. In these cases we want to generate a branch in the residual program that accounts for these possibilities. A staged interpreter demonstrates these two possibilities in a more substantial and less contrived context.

3.1 Staged Interpreters in Multi-stage miniKanren

The `evalo-or-staged` relation is a staged version of the `evalo-or` interpreter, specialized with respect to the expression `e` (the first argument of the interpreter) in the first stage. The result of specialization is miniKanren code that interprets the specific expression without the overhead of the interpreter's dispatch.

Before we consider its implementation, let's look at the code generated in a small example query:

```
(run* (x-val v) (staged (evalo-or-staged '(or #f x) `((x . ,x-val)) v)))
⇓ generated code (simplified)
(run* (x-val v) (== v x-val))
↪ ((_.0 _.0))
```

Notice that there are no unifications matching the syntax of the `or`-expression in the generated code. In that sense, staging removes the overhead of the interpreter dispatch.

For all examples in this paper, the actual generated code is more verbose than shown. For clarity, we present cleaned up code with friendly variable names and some basic optimizations applied, including constant folding and dead code elimination. In practice, multi-stage miniKanren will be used in conjunction with a compiler that performs such optimizations, e.g., [Ballantyne et al. 2024].

Figure 8 shows the implementation of `evalo-or-staged`. This staged interpreter is much like Figure 2b with the addition of staging annotations, typeset with underlines. This small language

```
(defrel/staged (evalo-or-staged e env v)
  (fallback
    (conde
      [(booleano e) (== e v)]
      [(fresh (e1 e2 v1)
         (== e `(or ,e1 ,e2))
         (evalo-or-staged e1 env v1)
         (gather
           (conde
             [(== v1 #f)
              (evalo-or-staged e2 env v)]
             [(=/= v1 #f) (== v1 v)])))]
      [(symbolo e) (lookupo e env v)]
      [(fresh (x e0)
         (== e `(lambda (,x) ,e0))
         (symbolo x)
         (make-closo x e0 env v))]
      [(fresh (e1 e2 v1 v2)
         (== e `(,e1 ,e2))
         (evalo-or-staged e1 env v1)
         (evalo-or-staged e2 env v2)
         (later (apply-closo v1 v2 v)))])))
```

Fig. 8. A staged interpreter for the λ-or language of Figure 2. Staging annotations are underlined.

showcases the features and forms of multi-stage miniKanren. It uses `later` to delay a goal from staging time to runtime and `fallback` and `gather` to control the staging of nondeterminism.

This interpreter uses the `later` form to delay the execution of function bodies via `apply-closo` in applications until runtime. This delay ensures that staging terminates even for non-terminating lambda calculus expressions. Constraints, including `==`, are automatically residualized cross-stage as needed, so `later` annotations are not needed most of the time.

3.1.1 Handling Nondeterminism. Nondeterminism that arises in a relational interpreter sometimes indicates an incomplete expression, and sometimes represents a decision to be made based on runtime values. The `evalo-or-staged` interpreter has two `conde` expressions: one to dispatch on the syntax of the expression, and one to handle the branch in `or`-expressions.

Holes in Staging-Time Terms. For dispatch on the expression, since the expression is conceptually static, nondeterminism means that there is an unknown hole in the program. In this case, we should fall back to run-time evaluation—so we use `fallback`. This query demonstrates the scenario:

```
(run 3 (e v) (staged (evalo-or-staged `(or (or #f ,e) #f) '() v)))
⇓ generated code (simplified)
(run 3 (e v)
  (fresh (e-val)
    (invoke-fallback evalo/1 e '() e-val)
    (conde
      [(=/= v #f) (== e-val v)]
      [(== e-val #f) (== v #f)])))
↪ ((#t #t) (#f #f) ((lambda (_.0) _.1) #<apply-rep>))
```

Runtime Nondeterminism. For the semantics of `or`, we want to specialize code for each option and generate a run-time branch—so we use `gather`. This query demonstrates the result:

```
(run* (x-val v) (staged (evalo-or-staged `(or x #t) `((x . ,x-val)) v)))
⇓ generated code (simplified)
(run* (x-val v)
  (conde
    [(=/= x-val #f) (== v x-val)]
    [(== x-val #f) (== v #t)]))
↪ (((_.0 _.0) $$ (=/= ((_.0 #f)))) (#f #t))
```

When the interpreter's first argument is fully ground, the dispatch on the parameter `e` is deterministic. However, when evaluating `(or x #t)` we don't know the value of `x`, mapped to the query variable `x-val` in the environment. Therefore, we want to generate code in the residual program that accounts for both possibilities, with each residual branch specialized to the expression. Notice that in the second branch, `v` is unified with the concrete value `#t` from the expression being evaluated, rather than being discovered through some recursive runtime call to the evaluator.

3.1.2 Sharing Code for Abstractions. When staging the interpretation of a program that applies a lambda expression several times, we want to generate code corresponding to the lambda body once, together with an invocation of that code for each application. A staged interpreter in a functional language such as MetaOCaml uses host-language lambdas in the interpretation of object-language lambdas. These host-language lambdas evaluate to first-class functions. This strategy is illustrated in Figure 4. The case for `Lam` returns a quotation containing an ML function.

For miniKanren, the equivalent to a first-class function would be a first-class relation. Supporting first-class relations in general requires higher-order unification. Multi-stage miniKanren instead includes a restricted form of first-class relation, employed in Figure 9. The `defrel-partial/staged` form defines a relation that can be partially applied. Such a definition has two lists of formal

```
(defrel-partial/staged (applyo rep [x e env] [v1 v])
  (evalo-or-staged e `((,x . ,v1) . ,env) v))

(defrel/staged (make-closo x e env clos)
  (specialize-partial-apply clos applyo x e env))

(defrel (apply-closo clos v1 v)
  (finish-apply clos applyo v1 v))
```

Fig. 9. Helpers for the small staged interpreter of Figure 8.

parameters: one for an initial partial application, and one for the final application. The goal forms `specialize-partial-apply` and `finish-apply` execute these application steps.

In Figure 9, the `applyo` relation handles a function application: it evaluates the function body `e` in an environment `env` extended with a binding of the function parameter `x` to a value `v1`. The staged relational interpreter represents closures as a partial application of this relation via `make-closo`. A function application is evaluated by finishing the `applyo` relation application, providing the value for the function argument and the term to unify the evaluation result with via `apply-closo`.

To avoid the complexities of higher-order unification, we require that both the `partial-apply` and `finish-apply` forms specify the name of the relation that is partially applied. That way, when the partially applied relation value (here, `clos`) is not known, miniKanren can nonetheless continue evaluation by falling back to a runtime version of the original relation, using fresh logic variables to represent the values from the initial partial application (here, `x`, `v`, and `env`).

Figure 10 shows the code generated by the relational interpreter for a small example using a first-class function: the identity function passed to a lambda, binding the function to the name `f` and applying it twice. The generated code contains two `apply-rep` terms, one for each lambda in the source program. An `apply-rep` term represents the result of a partial application and includes the name of the partially applied relation, the arguments given in the partial application, and a function representing specialized code for the partial application. Notice that the specialized code for the first lambda contains `finish-apply` forms that invoke the partially applied relation used to represent the second function, passed as `f`.

```
(run* (v)
  (staged (evalo-or-staged '((lambda (f) (or (f #t) (f #f)))
                             (lambda (x) x))
                           '() v)))
⇓ generated code (simplified)
(run* (v)
  (fresh (proc1 proc2)
    (== proc1 (apply-rep 'applyo '(f (or (f #t) (f #f)) ())
                         (λ (f out)
                           (fresh (f-res)
                             (finish-apply f applyo #t f-res)
                             (conde [(=/= f-res #f) (== f-res out)]
                                    [(== f-res #f) (finish-apply f applyo #f out)])))))
    (== proc2 (apply-rep 'applyo '(x x ()) (λ (x out) (== x out))))
    (finish-apply proc1 applyo proc2 v)))
↪ ((_.0 #t))
```

Fig. 10. Generated code with first-class values for partially applied relations, used to represent the staged evaluation of object-language lambda expressions.

```
term var tv,  term t
goal c_{p∈{r,s,l}}  ::=  (== t t) | (=/= t t) | etc.
                    |    (fresh (tv ...) g_p ...) | (conde (g_p ...) ...)
                    |    (partial-apply t rname t ...) | (finish-apply t rname t ...)
runtime goal g_r    ::=  c_r | (staged g_s)
staging-time goal g_s ::= c_s | (later g_l) | (gather g_s) | (fallback g_s)
                    |    (specialize-partial-apply t r t ...)
later goal g_l      ::=  c_l
definition d        ::=  (defrel (rname param ...) g_r)
                    |    (defrel/staged (rname param ...) g_s)
                    |    (defrel-partial (rname tv [tv ...] [tv ...]) g_r)
                    |    (defrel-partial/staged (rname tv [tv ...] [tv ...]) g_s)
expression e        ::=  (run* (tv ...) g_r) | (run n (tv ...) g_r)
```

Fig. 11. The syntax of multi-stage miniKanren.

Name	Description
$empty	The unique empty stream.
*$singleton*($x$)	Returns a stream of length 1 whose only element is x.
*$append*($s_1$, s_2)	Returns a stream whose elements include those of s_1 and s_2. This operation interleaves, taking elements from each stream as they become available.
*$append-map*($f$, s)	Given a function f which returns a stream, returns the result of mapping f over s, producing a stream of streams, then *$append*-ing all of the streams together. Appears elsewhere as *»=* or *flatMap*.
*$take*($n$, s)	Returns the first n elements of the stream s as a list.
*$takeAll*($s$)	Returns all the elements of the stream s as a list.

Fig. 12. Stream operations

4 Semantics

This section presents the syntax of multi-stage miniKanren together with a semantics for staging-time evaluation of a desugared core language, focusing on the treatment of nondeterminism. Due to space limitations the presentation does not address relation definitions and applications or the disequality, type, and absence constraints [Byrd et al. 2012] provided by the implementation.

The hallmark of relational programming is a pure, nondeterministic mental model. Like many miniKanren implementations, our semantics is based on interleaving streams [Hemann et al. 2016; Kiselyov et al. 2005]. One way to understand this model is in terms of "many worlds", where nondeterministic choice splits the world into two, one for each option. The central definition of our semantics, presented in Figure 13, is the denotation of a staged goal, $G[\![g]\!]_{\rho\sigma}$. Each world is represented as a state σ, and the denotation of a staged goal g is a function $G[\![g]\!]_{\rho\sigma}$ from an initial state σ and environment ρ to a stream of states in which the goal holds. These streams are lazy and potentially infinite. The metafunctions for manipulating streams are outlined in Figure 12 and correspond to the implementation from Hemann et al. [2016].

The denotation is defined for a desugared core syntax; Figure 11 shows the surface syntax. Desugaring transforms a `conde` into a disjunction (`disj`) of conjunctions (`conj`). Desugaring also replaces the implicit conjunction in the body of a `fresh` with an explicit `conj`.

$\sigma = (subst, code, determinacy\text{-}check)$

$vars(\rho)$, $vars(\sigma)$	Find the set of variables appearing anywhere in the environment or state.
$add\text{-}code(g_l, \rho, \sigma)$	Conjoin the code accumulated for runtime evaluation in σ with a new goal, where term variable references in the goal are replaced by their values from ρ.
$walk^*(t, sb)$, $walk^*(stx, sb)$	Apply a substitution to all logic variables in a term or syntax object containing terms until a fixed point is reached.
$T[\![t]\!]_\rho$	Replace variable references in the term with their value from ρ.

$$
\begin{aligned}
G[\![\texttt{(== } t_1 \ t_2\texttt{)}]\!]_{\rho\sigma} = \ & sb = unify(T[\![t_1]\!]_\rho, T[\![t_2]\!]_\rho, subst(\sigma)) \\
& \textbf{if } sb \textbf{ then } \$singleton(set\text{-}subst(sb, \sigma)) \\
& \textbf{else } \$empty \\
G[\![\texttt{(fresh (}tv\texttt{) } g_s\texttt{)}]\!]_{\rho\sigma} = \ & lv \notin (vars(\rho) \cup vars(\sigma)) \\
& \rho' = \rho[tv \mapsto lv] \\
& G[\![g_s]\!]_{\rho'\sigma} \\
G[\![\texttt{(conj } g_{s1} \ g_{s2}\texttt{)}]\!]_{\rho\sigma} = \ & \$append\text{-}map(\lambda\sigma'.G[\![g_{s2}]\!]_{\rho\sigma'}, G[\![g_{s1}]\!]_{\rho\sigma}) \\
G[\![\texttt{(disj } g_{s1} \ g_{s2}\texttt{)}]\!]_{\rho\sigma} = \ & \$append(G[\![g_{s1}]\!]_{\rho\sigma}, G[\![g_{s2}]\!]_{\rho\sigma}) \\
G[\![\texttt{(later } g_l\texttt{)}]\!]_{\rho\sigma} = \ & \$singleton(add\text{-}code(g_l, \rho, \sigma)) \\
G[\![\texttt{(fallback } g_s\texttt{)}]\!]_{\rho\sigma} = \ & \textbf{if } determinacy\text{-}check(\sigma) \textbf{ then } \$singleton(\sigma) \\
& \textbf{else} \begin{cases} \sigma' = set\text{-}determinacy\text{-}check(\sigma, T) \\ \textbf{case } \$take(2, G[\![g_s]\!]_{\rho\sigma'}) \\ \quad [] \ \Rightarrow \ \$empty \\ \quad [x] \ \Rightarrow \ G[\![g_s]\!]_{\rho\sigma} \\ \quad \_ \ \Rightarrow \ \$singleton(add\text{-}code(erase(g_s), \rho, \sigma)) \end{cases} \\
G[\![\texttt{(gather } g_s\texttt{)}]\!]_{\rho\sigma} = \ & \textbf{if } determinacy\text{-}check(\sigma) \textbf{ then } \$singleton(\sigma) \\
& \textbf{else } \$singleton(add\text{-}code(\texttt{(disj . } capture\text{-}syntax(g_s, \rho, \sigma)\texttt{)}, \rho, \sigma))
\end{aligned}
$$

$$
\begin{aligned}
& capture\text{-}syntax(g_s, \rho, \sigma) = \\
& \quad \sigma' = set\text{-}code(\sigma, \texttt{succeed}) \\
& \quad states = \$takeAll(G[\![g_s]\!]_{\rho\sigma'}) \\
& \quad map(generate\text{-}syntax(\rho, \sigma), states)
\end{aligned}
\qquad
\begin{aligned}
& generate\text{-}syntax(\rho, \sigma)(\sigma') = \\
& \quad sb = subst(\sigma') \\
& \quad L_{subst} = \textbf{for } (x, t) \in s : \texttt{(==} \ x \ \ walk^*(t, sb)\texttt{)} \\
& \quad L_{st} = walk^*(code(\sigma'), sb) \\
& \quad v = vars(\sigma') \setminus (vars(\sigma) \cup vars(\rho)) \\
& \quad \texttt{(fresh (}v \ldots\texttt{) } L_{subst} \ldots L_{st} \ldots\texttt{)}
\end{aligned}
$$

$$
\begin{aligned}
& \text{erase} : g_s \to g_l \\
& erase(\texttt{later } g_l) = g_l \\
& erase(\texttt{gather } g_s) = erase(g_s) \\
& erase(\texttt{fallback } g_s) = erase(g_s) \\
& erase(\texttt{specialize-partial-apply } t_0 \ r \ t_1 \ \ldots) = \texttt{partial-apply(}t_0 \ r \ t_1 \ \ldots\texttt{)}
\end{aligned}
$$

Fig. 13. The semantics of multi-stage miniKanren.

The staging-time state σ includes three pieces of information: a substitution, which stores all the information we know about the program's logic variables; a code store, which accumulates code to be evaluated at run time; and a flag used during `fallback`'s determinism check, explained below. The environment ρ is a map from the set of syntactic variables to terms. We require that multi-stage miniKanren programmers write their programs such that staging-time evaluation produces only a single state. The code stored in this state is then evaluated at runtime.

The semantics of unification (==), existential variable binding (`fresh`), conjunction (`conj`), disjunction (`disj`) are standard, following the implementation of Hemann et al. [2016]. The denotations for the staging annotations `later`, `gather`, and `fallback` are the novel portion.

```
(defrel/staged (listso x ls)
  (fallback (disj (== ls '())
                  (fresh (rest) (conj (== ls (cons x rest)) (listso x rest))))))
(run 1 (q) (staged (listso 7 q)))
```

(a) The `listso` relation builds lists `ls` of every length that contain only the value `x`. The query provides a fresh logic variable `q` for the `ls` argument, leading to nondeterminism that triggers fallback to runtime evaluation. If instead a ground value were to be provided for `ls`, the relation evaluation would complete at staging time.

$G[\![\texttt{(listso 7 q)}]\!]_{\{\texttt{q} \mapsto \texttt{q'}\}(\emptyset,\text{succeed},F)}$

$\Rightarrow$ (Evaluate the relation call by inlining the body of `listso`.)

$G[\![\texttt{(fallback (disj (== ls '())}$
$\texttt{(fresh (rest) (conj (== ls (cons 7 rest)) (listso 7 rest)))))}]\!]_{\{\texttt{ls} \mapsto \texttt{q'}\}(\emptyset,\text{succeed},F)}$

Evaluation of the `fallback` form checks whether the argument may produce more than one result:

$\$take(2, G[\![\texttt{(disj (== ls '())}$
$\texttt{(fresh (rest) (conj (== ls (cons 7 rest)) (listso 7 rest))))}]\!]_{\{\texttt{ls} \mapsto \texttt{q'}\}(\emptyset,\text{succeed},T)})$

$\Rightarrow$ (Evaluate the denotation of the `fallback` argument; the recursive call to `listso` yields an inner `fallback`.)

$\$take(2, \$append(\$singleton((\{\texttt{q'} \mapsto []\}, \text{succeed}, T)),$
$G[\![\texttt{(fallback ...)}]\!]_{\{\texttt{ls} \mapsto \texttt{rest'}\}(\{\texttt{q'} \mapsto \texttt{[7 | rest']}\},\text{succeed},T)}))$

$\Rightarrow$ (Because the determinacy-check flag is set, the inner `(fallback ...)` trivially succeeds.)

$\$take(2, \$append(\$singleton((\{\texttt{q'} \mapsto []\}, \text{succeed}, T)),$
$\$singleton((\{\texttt{q'} \mapsto \texttt{[7 | rest']}\}, \text{succeed}, T))))$

$\Rightarrow$ (Fully evaluating the stream *$take* yields a list of two result states with different substitutions.)

$[(\{\texttt{q'} \mapsto []\}, \text{succeed}, T), (\{\texttt{q'} \mapsto \texttt{[7 | rest']}\}, \text{succeed}, T)]$

$\Rightarrow$ (As there are multiple possibilities, the evaluation of `fallback` residualizes the erasure of its argument.)

$\$singleton(add\text{-}code(erase(\texttt{(disj ...)}), \{\texttt{ls} \mapsto \texttt{q'}\}, (\emptyset, \text{succeed}, F)))$

(b) Steps of staging-time evaluation. In this case, evaluation of `fallback` discovers that its argument goal is not determinate, so it residualizes a version of the argument goal with all staging annotations erased.

Fig. 14. A program and staging-time evaluation trace illustrating the semantics of `fallback`.

The denotation of a later goal $G[\![\texttt{(later } g_l\texttt{)}]\!]_{\rho\sigma}$ is a stream containing a single state which defers the evaluation of the goal g_l to run time by including it in the code field.

The denotation of a fallback goal $G[\![\texttt{(fallback } g_s\texttt{)}]\!]_{\rho\sigma}$ is either the result of evaluating g_s at staging time, or is a stream containing a state in which evaluation of g_s has been deferred to run-time. The goal g_s evaluates at staging time if it is proved to be determinate—that is, if it is certain to produce only a single answer at staging time. Otherwise, the goal is deferred to run time. The check to decide whether a goal is determinate is approximate. Precisely deciding determinacy could require evaluating the goal fully, which might lead to nontermination.[1] Instead, our design assumes that programmers use `fallback` and `gather` goals such that recursive evaluation within a `fallback` eventually reaches one of these goals. The determinacy check conservatively assumes that such nested `fallback` or `gather` goals always succeed, without evaluating them.

The `fallback` determinacy check is implemented by evaluating the goal g_s with the "determinacy-check" flag set. With the flag set, any `fallback` or `gather` goal within g_s trivially succeeds. The check uses the *$take* metafunction to evaluate the stream just far enough to find out whether or not it contains at least two answers. Figure 14 demonstrates this process with an evaluation trace.

[1] Our semantics does not include relation application, so all goals terminate. In the full system, goals may diverge.

$G[\![$`(gather (disj (conj (== p #t) (== q #f))`
`(conj (== p #f) (== q #t))))`$]\!]_{\{\mathtt{p} \mapsto \mathtt{p'}, \mathtt{q} \mapsto \mathtt{q'}\}(\emptyset, \text{succeed}, F)}$

To evaluate the `gather`, we first evaluate all possible results of the form's argument goal:

capture-syntax(`(disj (conj (== p #t) (== q #f)) (conj (== p #f) (== q #t)))`,
$\{\mathtt{p} \mapsto \mathtt{p'}, \mathtt{q} \mapsto \mathtt{q'}\}$, $(\emptyset, \text{succeed}, F)$)

$\Rightarrow$ (Evaluate the call to the `capture` metafunction and the denotation of the disjunction.)

map(*generate-syntax*($\{\mathtt{p} \mapsto \mathtt{p'}, \mathtt{q} \mapsto \mathtt{q'}\}$, $(\emptyset, \text{succeed}, F)$),
$[(\{\mathtt{p'} \mapsto T, \mathtt{q'} \mapsto F\}, \text{succeed}, F), (\{\mathtt{p'} \mapsto F, \mathtt{q'} \mapsto T\}, \text{succeed}, F)]$)

$\Rightarrow$ (Generate the syntax for each disjunct.)

`[(fresh () (== p' #t) (== q' #f)),(fresh () (== p' #f) (== q' #t))]`

$\Rightarrow$ (Assemble the syntaxes from each result into a residualized `disj`.)

$singleton(*add-code*(`(disj (fresh () (== p' #t) (== q' #f))`
`(fresh () (== p' #f) (== q' #t)))`, $\{\mathtt{p} \mapsto \mathtt{p'}, \mathtt{q} \mapsto \mathtt{q'}\}$, $(\emptyset, \text{succeed}, F)$))

Fig. 15. An evaluation trace illustrating the semantics of `gather`. The argument goal is evaluated to produce all of its result states, which are reified as syntax and residualized as a disjunction.

The denotation of a gather goal $G[\![$`(gather` g_s`)`$]\!]_{\rho\sigma}$ is a singleton stream that residualizes a disjunction with reified code for each state produced by the evaluation of g_s. Producing code for all the alternatives requires fully evaluating the stream produced by g_s, so staging will only terminate if g_s terminates with a finite stream. Figure 15 shows an example evaluation trace.

Code generated by `gather` should not include code residualized outside the `gather` form. The *capture-syntax(g_s, ρ, σ)* metafunction thus evaluates g_s using a state with an empty *code* field. The *generate-syntax(ρ, σ)(σ')* metafunction is applied to each state σ' in the stream produced by g_s. This metafunction constructs a conjunction that includes the goals explicitly residualized in σ' as well as unification goals that reconstruct the substitution extension in σ'. Logic variables allocated during the evaluation of g_s are `fresh`-bound in the generated code, whereas others refer to the surrounding context. These local variables are identified by comparing σ' with the the ρ and σ provided as input to g_s. Generating code as described in the semantics may residualize unification goals for substitution elements left over from staging-time evaluation that are irrelevant at run time. In the real system, a simple dead-code analysis removes these extra unifications.

Semantics for logic programs are often defined in terms of sets such as in Rozplokhas et al. [2020], but our semantics use streams. In the context of staging, programmers often need precise control over the code that is generated. For `gather`, the generated code depends on answer order, which is preserved by streams. Streams also allow the `fallback` determinacy check to discover whether the goal produces at least two answers without evaluating the entire stream.

4.1 Erasure

An important property of many multi-stage programming languages is *erasure*. The erasure of a staged program is the unstaged program resulting from removing all staging annotations. Assuming staging terminates, a staged program should have the same extensional behavior as its erasure though different performance characteristics [Inoue and Taha 2016]. Figure 13 defines the erasure of multi-stage miniKanren goals. Because of miniKanren's nondeterministic semantics, the equivalence between staged evaluation and erasure works modulo answer order. That is, staged evaluation should produce the same answer set as erasure followed by unstaged evaluation.

5 Removing Interpretive Overhead from Relational Queries with Staging

As discussed in Section 2.1.1, relational interpreters make it surprisingly easy to specify a variety of synthesis tasks. However, this flexibility comes at the cost of interpretive overhead. Staging can alleviate that cost by eliminating interpretive overhead for statically-known program text. This section discusses a number of concrete applications that benefit from staging. Section 6 evaluates the performance improvements in detail.

5.1 Interpreting Functions Relationally

Figure 3 illustrates how functions such as append yield behavior equivalent to relations such as appendo when executed within a relational interpreter, at the cost of interpretive overhead.

We created a staged relational interpreter, `evalo-staged`, for a similar subset of Racket as Byrd et al. [2017]'s `evalo`. Similar to our simplified illustration in Figure 8, the `evalo-staged` interpreter is staged with respect to the first argument—the program text. When program text is known in advance, staging-time computation specializes the interpreter to the specific program.

Figure 16 revisits the earlier example from Figure 3 using staging. The definition in Figure 16a mirrors the original appendo definition but includes a `staged` annotation and calls `evalo-staged`. Because the function definition is fully known, staging generates miniKanren code that executes the function without the overhead of the interpreter's dispatch. The code, shown in Figure 16b, constructs a term representing a specialized partial application of the `apply-letrec` relation from the `staged-evalo` interpreter. The `appendo-rep` definition contains the specialized relation body.

The generated code in `appendo-rep` differs from what a miniKanren programmer would write by hand because it is derived from the structure of the staged interpreter. For example, the interpretation of the `if` expression and the `null?` check from the append definition yield separate `conde` goals in the generated code, whereas a hand-written implementation would contain a single `conde` goal.

(a). With staging-time computation

```
(defrel (appendo xs ys zs)
  (staged
    (evalo-staged
      `(letrec
         ([append #| elided defn |# ])
         (append ',xs ',ys))
      initial-env
      zs)))
```

⇓

(b). Generated appendo relation

```
(defrel (appendo xs ys zs)
  (fresh (rep)
    (== rep (apply-rep
              #| elided partial app args |#
              'apply-letrec
              appendo-rep))
    (finish-apply rep
      (apply-letrec (list xs ys) zs))))
```

(c). Generated appendo rep, simplified

```
(define appendo-rep
  (λ (xs-and-ys zs)
    (fresh (b xs ys)
      (absento 'struct xs) (absento 'struct ys)
      (== (list xs ys) xs-and-ys)
      (conde
        [(== xs '()) (== b #t)]
        [(=/= xs '()) (== b #f)])
      (conde
        [(=/= b #f) (== ys zs)]
        [(fresh (a r d)
           (=/= a 'struct)
           (== b '#f)
           (== xs (cons a d))
           (== zs (cons a r))
           (finish-apply rep
             (apply-letrec
               (list d ys) r)))]))))
```

Fig. 16. Rather than write miniKanren relations such as appendo directly, a programmer can write the append function in Racket and use the staged relational interpreter to work with it as a relation, without interpretive overhead. This definition of appendo stages the one in Figure 3. The generated code performs similarly to a hand-written relation despite including some additional code due to the structure of the interpreter.

The generated code also includes constraints to ensure that the list values do not overlap with the interpreter's other data types, which are represented by lists beginning with the tag `'struct`.

5.2 Accelerating Program Synthesis By Sketch

When a relational interpreter evaluates a program where some of the program text is unknown, it performs program synthesis. Staging can eliminate interpretive overhead for evaluations of known parts of the program that occur during synthesis of the unknown parts. The query in Figure 17 synthesizes parts of an accumulator-passing style Fibonacci program that uses Peano numerals.[2]

```
(run 1 (e1 e2 e3 acc1 acc2)
  (staged
    (evalo-staged
      `(letrec (#| definitions of zero?, +, and - elided |#
                [fib (lambda (n a1 a2)
                       (if (zero? n)
                           ,e1
                           (if (zero? (sub1 n))
                               a2
                               (fib (- n '(s . z)) ,e2 ,e3))))])
         (list (fib 'z ',acc1 ',acc2) #| six more examples elided |# ))
      initial-env
      '(z #| six more outputs elided |# ))))
↪ ((a1 a2 (+ a1 a2) z (s . z)))
```

Fig. 17. Synthesizing parts of a Fibonacci function that uses a library of Peano arithmetic functions.

The query includes partial text for the Fibonacci function (a sketch) and complete definitions of the arithmetic helper functions `zero?`, `+`, and `-` (elided for space). Staging the query generates specialized code for the known portions of the program. When staged interpretation encounters a hole in the program text (e.g., `e1`, `e2`, or `e3`), the `fallback` feature (Section 3.1.1) automatically generates a call to the run-time interpreter. The run-time interpretation required for synthesizing expressions in the holes incurs interpretive overhead. However, when runtime evaluation synthesizes a call to one of the statically known helper functions such as the Peano `+` function, evaluation of the call uses the specialized code that was generated for the definition.

5.3 Other Staged Interpreters

We have also created staged relational interpreters for two other object languages: miniKanren, and context-free grammars. As with `staged-evalo`, synthesis using these interpreters avoids interpretive overhead when parts of the object-language program text are statically known.

miniKanren-in-miniKanren. Joshi and Byrd [2021] introduce metaKanren, a relational interpreter for a minimalist miniKanren. The following query uses our staged version of this interpreter to complete a definition of the appendo relation:

```
(run 1 (relcall)
  (fresh (w x y)
    (symbolo w) (symbolo x) (symbolo y)
    (synth-appendo-recursive-call `(call-rel appendo ,w ,x ,y))))
↪ ((call-rel appendo d ys res))
```

Specifically, it synthesizes variable references to fill three holes in the recursive call to appendo.

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

Relational Recognizer. We also implemented a staged relational recognizer for strings of a provided context-free grammar. The three-place relation `recognizeo` holds between a grammar, its start symbol, and a string derivable from the start symbol using the grammar's production rules. Staging specializes the recognizer to the grammar provided, eliminating overhead from the grammar data structures. Such a relation can be used either to validate that a given string matches the grammar, or to generate matching strings by leaving a logic variable in place of the string argument. The query below generates 200 strings in the language of a small arithmetic expression grammar.

```
(define E-grammar
  '((E . (or 'S (seq 'S * 'S))) (S . (or 'T (seq 'T + 'T))) (T . (or 0 (seq < 'E >)))))
(run 200 (str) (recognizeo E-grammar 'E str))
```

5.4 Interpreting Interpreters

Writing a relational interpreter in miniKanren is one way of implementing a synthesizer for a language. However, there is a second, easier way: write an interpreter in Racket, and run it within the `evalo-staged` relational interpreter. A programmer can write an `eval` function for an object language L in `evalo-staged`'s subset of Racket and then query the function like a relation. Staging eliminates the interpretive overhead of `evalo-staged`, yielding performance akin to writing a relational interpreter for L directly. The following applications follow this pattern.

Quines with Quasiquote. Byrd et al. [2017] propose generating a quine expressed with `quasiquote` using a nested interpreter. The language of their relational interpreter `evalo` (like our `evalo-staged`) does not support quasiquotation. To work around this, Byrd et al. [2017] use an interpreter for a subset of Racket that does include `quasiquote`, written in the language of their `evalo`. The interpreter consists of two mutually recursive evaluation functions—one for standard evaluation and one for evaluation under a quasiquote [Bawden 1999]. The query searches for a value `q` such that interpreting it with the nested interpreter yields `q` itself:

```
(run 1 (q)
  (absento 'error q) (absento 'struct q)
  (staged
    (evalo-staged
      `(letrec ([eval-expr  #| elided |#]
                [eval-quasi #| elided |#])
         (eval-expr ',q (lambda (x) 'error)))
      initial-env
      q)))
```

The query yields the quine `((lambda (x) `(,x ',x)) '(lambda (x) `(,x ',x)))`.

Regular Expressions. This example is adapted from Might et al. [2011]. The `regex-match` function decides if a string is in the language defined by a regular expression, using repeated application of the Brzozozwksi derivative [Brzozowski 1964]. The function can be viewed as an interpreter for regular expressions. With `evalo-staged`, we can use this function to synthesize regular expressions:

```
(run 1 (regex) (regex-matcho regex '(foo bar foo bar foo bar) #t))
```

This query generates the regular expression $(\texttt{foo bar})^*$, which indeed matches the given string.

Theorem Checker Turned Prover. Figure 18 presents a quoted lambda expression for a `prf?` function that checks proofs for a fragment of propositional logic. Aside from being quoted, it is an ordinary program written in a subset of Racket. Just as in Figure 16 we turn `append` into `appendo`, in Figure 18 we use staged interpretation to turn `prf?` into a relation `proofo`. The `run` query calls the newly compiled relation with a partially instantiated term `prf`. The term includes a proposition but leaves a hole, `body`, for the actual proof. The result of evaluation is constrained to be true (`#t`), so the query searches for a proof that satisfies the `prf?` predicate.

```
(define prf?-fn
  '(lambda (prf)
     (match prf
       [`(,A ,prems Assm ()) (mem? A prems)]
       [`(,B ,prems ModusPonens
          (((,A => ,B) ,prems ,r1 ,prf1)
           (,A ,prems ,r2 ,prf2)))
        (and (prf? `((,A => ,B) ,prems ,r1 ,prf1))
             (prf? `(,A ,prems ,r2 ,prf2)))]
       [`((,A => ,B) ,prems =>Intro
          ((,B (,A . ,prems) ,r ,tr1)))
        (prf? `(,B (,A . ,prems) ,r ,tr1))])))
```

```
(defrel (proofo prf is-valid)
  (staged
    (evalo-staged
      `(letrec
         ( #| mem? binding elided |#
          [prf? ,prf?-fn])
         (prf? ',prf))
      initial-env
      is-valid)))
```

```
(run 1 (prf)
  (fresh (body)
    ;; Prove C, assuming A, A ⇒ B, and B ⇒ C
    (== prf `(C (A (A => B) (B => C)) . ,body))
    (proofo prf #t)))
```

```
((C (A (A => B) (B => C)) ModusPonens
 (((B => C) (A (A => B) (B => C)) Assm ())
  (B (A (A => B) (B => C)) ModusPonens
   (((A => B) (A (A => B) (B => C)) Assm ())
    (A (A (A => B) (B => C)) Assm ()))))))
```

Fig. 18. A propositional logic theorem checker turned automatic prover. The `proofo` relation can both check and generate proofs. It is defined by staged relational interpretation of the Racket `prf?` predicate, which in its usual functional meaning can only check a proof's validity.

6 Evaluation

In empirically evaluating multi-stage miniKanren, we consider the following research questions:

- **Q1** Is it feasible to stage a variety of relational programs, and in particular relational interpreters?
- **Q2** Does staging substantially reduce the runtime cost of solving synthesis problems and other queries that take advantage of the flexibility of relational interpreters?
- **Q3** Is the time cost of staging reasonable?

To address these questions, we stage a collection of relational programs and evaluate the performance of staged and unstaged benchmark queries. Table 1 presents the results.

Relational programs from the literature inspire many of our benchmarks. Byrd et al. [2017] introduce the possibility of querying functions via a relational interpreter (Section 5.1) and lifting relational behavior through additional layers of interpretation (Section 5.4). The specific tasks of using the append function as a relation (the "invert-append" benchmark), lifting synthesis to proofs (the "proofo" benchmarks), and lifting synthesis to a language with quasiquote via a metacircular evaluator ("quasi-quine") all come from Byrd et al. [2017]. The "pow-8-backward" benchmark exercises staging in the context of the relational arithmetic library from Kiselyov et al. [2008]. The task of synthesizing portions of the function defining the Fibonacci series ("synth-fib", "synth-fib-larger") is inspired by Chirkov et al. [2020], which uses this example in the context of a relational interpreter for JavaScript. Finally, Joshi and Byrd [2021] presents a relational interpreter for miniKanren, written in miniKanren, which makes it possible to synthesize fragments of miniKanren programs. Our "metaKanren" benchmark is one such synthesis task. The remaining benchmarks are programs we created to explore the capabilities of multi-stage miniKanren.

The benchmarks in Table 1 are organized into four categories. The first category comprises the simplest benchmarks, which are straightforward uses of relations staged with respect to one argument. The benchmarks in the second category apply the `evalo-staged` interpreter to Racket programs that include fully-ground function definitions. As described in Section 5.1, staging has the effect of compiling such functions to miniKanren code. The benchmarks in the third

Table 1. Times are in milliseconds. [> 5m] indicates an execution that failed to terminate within 5 minutes. Each benchmark represents a single execution on a 2021 Apple MacBook Pro with an M1 Max CPU and 64GB of RAM running macOS 14.2 and Racket v8.15. For benchmarks labeled with a multiplier (e.g., ×1000), timings reflect repeatedly running the query that number of times to yield measurable results.

Name	Staging	Staged	Unstaged	Speedup	Description
relations: Simple relations staged with respect to one staging-time argument.					
replicate-unknown	6	288	237	0.82	Find a Peano numeral n and a list of values l such that replicating n times each value in l gives a given output (x1000)
replicate-partial	4	266	552	2.08	Given a partially-instantiated Peano numeral and fixed input list, replicate n times each value in l (x10000)
pow8-backward	0	1650	2079	1.26	Solve $n^8 = 6561$ via relational arithmetic [Kiselyov et al. 2008] (x1000)
grammar-synthesis	12	12	51	4.25	Find 200 strings that match a given grammar as in Section 5.3
functions: Functions written in Racket executing within the `evalo-staged` interpreter, exhibiting relational behavior.					
invert-append	6	26	480	18.46	Use append to split a list as in Figure 3 (x1000)
fib	29	59	2800	47.46	Synthesize an input value for which fib returns 13 (x100)
fib-exprs	29	121	10490	86.69	Synthesize 5 input expressions for which `fib` returns 13
nnf	400	65	453	6.97	Transformation to negation normal form [Szeredi et al. 2014] (x1000)
interpreters: Queries that leverage interpreters within the `evalo-staged` interpreter to lift relational behavior to new languages.					
eval-or	105				An interpreter written in Racket for the language of Figure 2b
eval-or 1		223	1869	8.38	Synthesize inputs and outputs to $\lambda x.\text{or } x\ x$ (x1000)
eval-or 2		108	1898	17.57	Synthesize 5 programs that evaluate to `#t` (x1000)
proofo	325				Checks the validity of proofs for implicational propositional calculus
proofo 1		1	22	22.00	Synthesize a proof of C from assumptions as in Figure 18
proofo 2		59	554	9.39	Synthesize a proof of $(A \Rightarrow B) \Rightarrow ((B \Rightarrow C) \Rightarrow (A \Rightarrow C))$
proofo 3		1097	[> 5m]	[>273.47]	Synthesize a proof of a longer chain of inferences
regex-match	186				Check that a string matches a regex
regex-match 1		214	12723	59.45	Check that a string matches a regex (x100)
regex-match 2		431	77047	178.76	Synthesize a regex that matches a given string as in Section 5.4 (x10)
list-eval	218	1647	9991	6.07	Synthesis within a metacircular evaluator with list functions
quasi-quine	117	1737	25636	14.76	Synthesize a quine that employs `quasiquote` as in Section 5.4
ground context: Synthesis queries in which a provided sketch of the program or library of helper functions is compiled by staging.					
synth-fib	38	945	46440	49.14	Fib function base case from examples (x1000)
synth-fib-larger	43	1797	[> 5m]	[>166.94]	Fib function accumulators and three holes as in Figure 17
map-eval	251	61	518	8.49	Body of function mapped via anonymous recursion in an evaluator
metaKanren	20	6812	23175	3.40	The recursive call arguments in appendo as in Section 5.3
evalo-map	13	20	58	2.90	The body of a function mapped over several examples
synth-append	9	84	201	2.39	Portion of append from examples as in Figure 3 (x100)

category similarly use fully-ground functions running in `evalo-staged`, but in these benchmarks the functions are themselves interpreters for another language, as discussed in Section 5.4. The benchmarks of the fourth category are synthesis problems with a provided sketch or library of helper functions. Staging generates code for the statically-known portions, as described in Section 5.2.

One benchmark shows a slowdown: "replicate-unknown". The `replicate` relation is staged with the expectation that the first argument will be known at staging time. However, this query provides a fresh logic variable for this argument, which precludes any specialization. The benchmark shows a slowdown relative to the original, unstaged program because the staged version of the replicate relation uses `partial-apply` in the recursion to enable staging. Our partial-application mechanism adds some overhead, and because in this example there is no benefit to staging, the overhead leads to a slowdown. All the rest of our benchmarks test situations where staging has some benefit.

Comparison to a Hand-Tuned Relational Interpreter. Table 2 reports benchmarks comparing our (un)staged relational interpreter to an interpreter that integrates specialized search heuristics. Byrd et al. [2017, p. 15–16] describe the heuristics in detail. The most important heuristic reorders parts of the evaluation of a function application. Normally the interpreter evaluates the argument

Table 2. Our (un)staged interpreter vs. Byrd et al. [2017]'s unstaged interpreter with hand-tuned heuristics. Barliman tests were conducted under Chez Scheme version 10.1.0.

Name	Unstaged	Staged	Hand-tuned	Speedup (Staged vs. Hand-tuned)
proofo 1	0.022s	0.001s	0.20s	200x
proofo 2	0.554s	0.059s	9.12s	155x
synth-fib-larger	[> 5m]	1.797s	11.11s	6x
quasi-quine	25.636s	1.737s	119.84s	69x

expressions first, and then the body of the function. The heuristic reverses this order when the argument expressions are unknown but the body is ground. Other heuristics include deferring certain non-deterministic goal evaluations, a depth-limited search, and manual weighting of the interpreter's main branch. The benefits of these search heuristics are orthogonal to the benefits of removing interpretive overhead that multi-stage miniKanren provides. The subset of Racket supported by our relational interpreter differs somewhat from that of Byrd et al. [2017] and the systems use different host Scheme implementations and versions of miniKanren, so the comparison is imprecise. Nonetheless the results suggest that staging a straightforward interpreter produces results competitive to an unstaged interpreter that uses carefully-tuned search heuristics.

Summary. With these results in hand we can summarize the answers to our research questions.

- **Q1** Our examples demonstrate that staged versions of a variety of relational programs are expressible in multi-stage miniKanren, including interpreters for a subset of Racket, grammars, and for miniKanren itself (metaKanren). Staging these programs requires adding staging annotations as well as small refactorings to use partially applied relations.
- **Q2** The results in Table 1 indicate that staging is effective at removing overhead introduced by layers of interpretation. Queries that evaluate ground functions within the `evalo-staged` interpreter exhibit speedups between 7x and 87x (the "functions" category in the benchmark table). Similarly, queries that leverage additional interpreters executing within `evalo-staged` all benefit from staging, with speedups between 6x and 178x (the "interpreters" category). Queries where a sketch or library of helpers can be compiled via staging exhibit speedups ranging from 2x to 49x (the "ground context" category).
- **Q3** Improved run-time performance easily pays for the time cost of staging in the case of difficult synthesis queries such as "proofo 3", "regex-match", "quasi-quine", "synth-fib-larger", and "metaKanren". For smaller queries that return within a few hundred milliseconds, staging may not be beneficial unless many queries are made using the same generated code.

7 Related Work

Staged relational programming continues a long line of investigation into multi-stage programming and related efforts in partial evaluation and metaprogramming [Lilis and Savidis 2019].

Relational Programming. Lozov et al. [2019] and Verbitskaia et al. [2020] explore partial deduction in the context of a typed miniKanren dialect in OCaml [Kosarev and Boulytchev 2016], We go further by allowing holes in programs and by supporting higher-order patterns. Lozov et al. [2018] translate functions to relational programs, but not using staging. Abramov and Glück [2001, 2002] achieve program inversion via non-standard interpretations and also remove overhead of one layer of interpretation. In our terminology, they are able to run "backwards" but not with arbitrary holes.

Multi-Stage Programming. Multi-stage programming systems for functional programming languages include MetaML [Taha and Sheard 2000], MetaOCaml [Kiselyov 2014], Lightweight Modular Staging [Rompf and Odersky 2010, 2012], and Terra [DeVito et al. 2013]. Our work adapts multi-stage

programming to the context of relational programming. We require that staging-time evaluation produces a unique answer via a dynamic check. Systems such as Mercury [Somogyi et al. 1996] and Ciao Prolog [Bueno et al. 1996] can statically check the determinism of relations. Such a static property could make correct staged programs easier, but it is unclear how to combine such reasoning with our fallback feature.

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. Offline partial deduction relies on an intermediate representation in which binding-time annotations serve a similar role to staging annotations, but are automatically inferred by binding-time analysis [Bruynooghe et al. 1998; Craig et al. 2005]. The tradeoff between multi-stage miniKanren and partial deduction mirrors the tradeoff between multi-stage functional languages and partial evaluation: manual staging is less automatic but more predictable.

Leuschel et al. [2004a,b] specialize interpreters in Prolog using offline partial deduction. They rely on manual annotations rather than the inference that is typical in partial evaluation. A memoization annotation handles recursion while "binding types" 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. We can fall back to dynamic evaluation, which supports specializing large compiled contexts around a small interpreted hole. We also leverage interpretation to convert from functions to relations.

Gallagher [1986] applies partial deduction to specialize Prolog meta-interpreters that realize alternate control strategies such as co-routining conjunction and breadth-first search. The resulting programs thus leverage those strategies without suffering interpretive overhead.

Synthesis. 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]. Semantics-Guided Synthesis (SemGuS) [Kim et al. 2021] is a framework for program synthesis with user-defined semantics specified in Constrained Horn Clauses, which resemble relational interpreters. However, existing SemGus implementations do not specialize to sketches or lift synthesis through nested layers of interpretation.

8 Conclusion

The design of multi-stage miniKanren took several iterations. The initial version allowed programmers to accumulate code for a `later` stage but required them to carefully manage fallback by non-relational inspection of whether terms were ground. Eventually, this experiment led to a more intuitive design tailored to relational programming, with new constructs to help with nondeterminism in the first stage as well as compilation of closures. A key feature is automatic fallback: a staged relation can be used as a code generator during staging or can be deferred entirely to the runtime, with the semantic correspondence automatically established by erasure of staging annotations. The new design has been motivated by writing relational interpreters, and has proven fruitful for other applications including a relational recognizer for context-free grammars.

Multi-stage miniKanren fits within a trend towards more generality and meta-reuse in synthesis systems; e.g., parameterizing over the object language [Kim et al. 2021; Martins et al. 2019; Polozov and Gulwani 2015]. We anticipate that our approach to specializing based on partially known expressions and collapsing added layers of interpretation via staging could be a fruitful technique for improving performance in other such synthesis systems.

Acknowledgments

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. Laura Zharmukhametova helped explore applications of an early version of multi-stage miniKanren. Chung-chieh Shan helped Michael Ballantyne adapt the well-known staging example of exponentiation to multi-stage miniKanren, which appears as our "pow8-backward" benchmark. Alex Bai helped explore potential connections to SemGuS. We thank Anastasiya Kravchuk-Kirilyuk for feedback on drafts and Adam Chlipala, Steve Chong, and Matthias Felleisen for insightful discussions.

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. The research of Jason Hemann has been partially supported by NSF grant CCF-2348408. The research of Nada Amin has been partially supported by NSF grant 2303983. The research of Michael Ballantyne has been partially supported by NSF grants SHF 2116372 and 2315884.

Artifact Availability

Multi-stage miniKanren is developed as open-source software at https://github.com/namin/staged-miniKanren. A snapshot of the implementation, our benchmark suite, and instructions for reproducing our results are packaged as an artifact available on Zenodo [Ballantyne et al. 2025].

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 (April 2001), 171–211. doi:10.1142/S0129054101000448

Sergei Abramov and Robert Glück. 2002. Principles of inverse computation and the universal resolving algorithm. In *The essence of computation*. Lecture notes in computer science, Vol. 2566. 269–295. doi:10.1007/3-540-36377-7_13

Michael Ballantyne, Mitch Gamburg, and Jason Hemann. 2024. Compiled, Extensible, Multi-language DSLs (Functional Pearl). *Proc. ACM Program. Lang.* 8, ICFP, Article 238 (Aug. 2024). doi:10.1145/3674627

Michael Ballantyne, Rafaello Sanna, Jason Hemann, William E. Byrd, and Nada Amin. 2025. Multi-Stage Relational Programming Artifact. doi:10.5281/zenodo.15233194

Alan Bawden. 1999. Quasiquotation in Lisp. In *Proc. Workshop on Partial Evaluation and Semantics-Based Program Manipulation*. 4–12. https://www.brics.dk/NS/99/1/BRICS-NS-99-1.pdf

Maurice Bruynooghe, Michael Leuschel, and Konstantinos Sagonas. 1998. A polyvariant binding-time analysis for off-line partial deduction. In *Proc. European Symposium on Programming*. 27–41. doi:10.1007/BFb0053561

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

Francisco Bueno, Manuel Hermenegildo, Pedro López, and Germán Puebla. 1996. *The Ciao Preprocessor*. Technical Report CLIP 1/06. The Computational logic, Languages, Implementation, and Parallelism (CLIP) Lab at IMDEA Software Institute. https://ciao-lang.org/legacy/files/ciao/ciao-1.15/13954560f564e08056ce53d86ebffad604b000dd/CiaoDE-1.15-1653-g1395456_ciaopp.pdf

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 (Aug. 2017). doi: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 *Proc. Workshop on Scheme and Functional Programming*. 8–29. doi:10.1145/2661103.2661105

Cristiano Calcagno, Walid Taha, Liwen Huang, and Xavier Leroy. 2003. Implementing multi-stage languages using ASTs, gensym, and reflection. In *Proc. Generative Programming and Component Engineering*. 57–76. doi:10.1007/978-3-540-39815-8_4

Artem Chirkov, Gregory Rosenblatt, Matthew Might, and Lisa Zhang. 2020. A relational interpreter for synthesizing JavaScript. In *Proc. miniKanren and Relational Programming Workshop*. 123–155. http://hdl.handle.net/2047/D20413639

Stephen-John Craig, John P. Gallagher, Michael Leuschel, and Kim S. Henriksen. 2005. Fully automatic binding-time analysis for Prolog. In *Proc. Symposium on Logic-Based Program Synthesis and Transformation*. 53–68. doi:10.1007/11506676_4

Zachary DeVito, James Hegarty, Alex Aiken, Pat Hanrahan, and Jan Vitek. 2013. Terra: a multi-stage language for high-performance computing. In *Proc. Programming Language Design and Implementation*. 105–116. doi:10.1145/2491956.2462166

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.

John Gallagher. 1986. Transforming logic programs by specialising interpreters. In *Proc. European Conference on Artificial Intelligence*. 313–326.

Yuichiro Hanada and Atsushi Igarashi. 2014. On cross-stage persistence in multi-stage programming. In *Proc. Functional and Logic Programming*. 103–118. doi:10.1007/978-3-319-07151-0_7

P. G. Harrison and H. Khoshnevisan. 1992. On the synthesis of function inverses. *Acta Informatica* 29, 3 (1992), 211–239. doi:10.1007/BF01185679

Jason Hemann, Daniel P. Friedman, William E. Byrd, and Matthew Might. 2016. A small embedding of logic programming with a simple complete search. In *Proc. Symposium on Dynamic Languages*. 96–107. doi:10.1145/2989225.2989230

Jun Inoue and Walid Taha. 2016. Reasoning about multi-stage programs. *Journal of Functional Programming* 26, Article e22 (2016). doi:10.1017/S0956796816000253

Ulrik Jørring and William L. Scherlis. 1986. Compilers and staging transformations. In *Proc. Principles of Programming Languages*. 86–96. doi:10.1145/512644.512652

Ramana Joshi and William E. Byrd. 2021. metaKanren: Towards a metacircular relational interpreter. In *Proc. miniKanren and Relational Programming Workshop*. 47–73. http://hdl.handle.net/1807/110263

Jinwoo Kim, Qinheping Hu, Loris D'Antoni, and Thomas Reps. 2021. Semantics-guided synthesis. *Proc. ACM Program. Lang.* 5, POPL, Article 30 (Jan. 2021). doi:10.1145/3434311

Oleg Kiselyov. 2014. The design and implementation of BER MetaOCaml. In *Proc. Functional and Logic Programming*. 86–102. doi:10.1007/978-3-319-07151-0_6

Oleg Kiselyov, William E. Byrd, Daniel P. Friedman, and Chung-chieh Shan. 2008. Pure, declarative, and constructive arithmetic relations (declarative pearl). In *Proc. Functional and Logic Programming*. 64–80. doi:10.1007/978-3-540-78969-7_7

Oleg Kiselyov, Chung-chieh Shan, Daniel P. Friedman, and Amr Sabry. 2005. Backtracking, interleaving, and terminating monad transformers (functional pearl). In *Proc. International Conference on Functional Programming*. 192–203. doi:10.1145/1086365.1086390

Dmitry Kosarev and Dmitry Boulytchev. 2016. Typed embedding of a relational language in OCaml. *Proc. Workshop on ML*. doi:10.48550/arXiv.1805.11006

Michael Leuschel, Stephen J. Craig, Maurice Bruynooghe, and Wim Vanhoof. 2004a. Specialising interpreters using offline partial deduction. In *Program Development in Computational Logic*. Lecture notes in computer science, Vol. 3049. 340–375. doi: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. doi:10.1017/S1471068403001662

Yannis Lilis and Anthony Savidis. 2019. A survey of metaprogramming languages. *ACM Comput. Surv.* 52, 6, Article 113 (Oct. 2019). doi:10.1145/3354584

Petr Lozov, Ekaterina Verbitskaia, and Dmitry Boulytchev. 2019. Relational interpreters for search problems. In *Proc. miniKanren and Relational Programming Workshop*. 43–57. http://nrs.harvard.edu/urn-3:HUL.InstRepos:41307116

Petr Lozov, Andrei Vyatkin, and Dmitry Boulytchev. 2018. Typed relational conversion. In *Proc. Trends in Functional Programming*. 39–58. doi:10.1007/978-3-319-89719-6_3

Ruben Martins, Jia Chen, Yanju Chen, Yu Feng, and Isil Dillig. 2019. Trinity: an extensible synthesis framework for data science. *Proc. VLDB Endow.* 12, 12 (Aug. 2019), 1914–1917. doi:10.14778/3352063.3352098

Matthew Might, David Darais, and Daniel Spiewak. 2011. Parsing with derivatives: A functional pearl. In *Proc. International Conference on Functional Programming*. 189–195. doi:10.1145/2034773.2034801

Peter-Michael Osera and Steve Zdancewic. 2015. Type-and-example-directed program synthesis. In *Proc. Programming Language Design and Implementation*. 619–630. doi:10.1145/2737924.2738007

Oleksandr Polozov and Sumit Gulwani. 2015. FlashMeta: a framework for inductive program synthesis. In *Proc. Object-Oriented Programming, Systems, Languages, and Applications*. 107–126. doi:10.1145/2814270.2814310

Tiark Rompf and Martin Odersky. 2010. Lightweight modular staging: A pragmatic approach to runtime code generation and compiled DSLs. In *Proc. Generative Programming and Component Engineering*. 127–136. doi: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. doi:10.1145/2184319.2184345

Dmitry Rozplokhas, Andrey Vyatkin, and Dmitry Boulytchev. 2020. Certified semantics for relational programming. In *Proc. Asian Symposium on Programming Languages and Systems*. 167–185. doi:10.1007/978-3-030-64437-6_9

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

Armando Solar-Lezama. 2009. The sketching approach to program synthesis. In *Proc. Asian Symposium on Programming Languages and Systems*. 4–13. doi:10.1007/978-3-642-10672-9_3

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. doi:10.1016/S0743-1066(96)00068-4

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. doi:10.1017/CBO9781139194129

Walid Taha. 1999. *Multi-Stage Programming: Its Theory and Applications*. Ph. D. Dissertation. Advisor(s) Tim Sheard.

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

Ekaterina Verbitskaia, Danil Berezun, and Dmitry Boulytchev. 2020. An empirical study of partial deduction for miniKanren. In *Proc. miniKanren and Relational Programming Workshop*. 13–21. http://hdl.handle.net/2047/D20413639

Received 2024-11-15; accepted 2025-03-06