Little Languages for Relational Programming

Daniel W. Brady   Jason Hemann   Daniel P. Friedman
Indiana University
{dabrady,jhemann,dfried}@indiana.edu

Abstract

The miniKanren relational programming language, though designed and used as a language with which to teach relational programming, can be immensely frustrating when it comes to debugging programs, especially when the programmer is a novice. In order to address the varying levels of programmer sophistication, we introduce a suite of different language levels. We introduce the first of these languages, and provide experimental results that demonstrate its effectiveness in helping beginning programmers discover and prevent mistakes. The source to these languages is found at https://github.com/dabrady/LittleLogicLangs.

Categories and Subject Descriptors D.3.2 [*Language Classifications*]: Applicative (functional) languages, Constraint and logic languages; D.2.5 [*Testing and Debugging*]: Debugging aids

Keywords miniKanren, microKanren, Racket, Scheme, relational programming, logic programming, macros

1. Introduction

miniKanren is a family of embedded domain-specific language for relational (logic) programming with over 40 implementations in at least 15 different languages, including ones in Clojure, Haskell, Ruby, and C#. Much of the current development, however, is carried out in Scheme, Clojure, and Racket (see http://minikanren.org).

In addition to the industrial [3, 13, 18] and academic [1, 4, 17, 20] uses, miniKanren has also been used as a teaching language. It has been successfully used to introduce students to logic programming, both through the textbook *The Reasoned Schemer* [11] and as a part of the curriculum in Indiana University Bloomington's undergraduate and graduate programming languages courses [10].

The relational programming paradigm differs significantly from functional or imperative programming, and is difficult for beginning students. With miniKanren programming, this holds even for students already familiar with the embedding language (e.g. Scheme, Racket).

Debugging miniKanren programs is frequently one of the most frustrating aspects for new programmers. Debugging is a difficult problem in programming generally, and can be time consuming and tedious. Debugging miniKanren carries additional challenges above those of many other languages. miniKanren is implemented as a shallow embedding, and historically its implementations have been designed to be concise artifacts of study rather than featureful and well-forged tools. As a result, implementers have given little attention to providing useful and readable error messages to the user at the level of their program. What error handling there is, then, is that provided by default with the host language. This has the negative impact of, in the reporting of errors, communicating details of the miniKanren implementation with the user's program. This is a problem common to many shallow embedded DSLs [12].

This can leave the programmer truly perplexed. What should be syntax errors in the embedded language are instead presented as run-time errors in the embedding language. This makes bugs more difficult to track down. Run-time errors may manifest some distance from the actual source of the error. A poor error message can cause a programmer to look for bugs far from the actual source of the problem, and perhaps accidentally break correct code in a misguided attempt to fix the problem. Moreover, the mixing of miniKanren implementation and user program means that often the user must have some knowledge of the miniKanren implementation to understand the reported error.

The promise of domain-specific languages [2] is that we can more quickly map a solution to code in a language specifically tailored to the problem than in a more general-purpose language. As it stands in miniKanren, the user is forced back to thinking in a general-purpose language when an error arises, precisely when a domain-specific language would be most useful. miniKanren presents an additional complication, though: miniKanren is designed specifically to be a DSL in which the programmer *does* have access to the entirety[1] of the host language.

While a programmer will most often only use the primitives defined in miniKanren itself, the language allows her access to non-miniKanren code of the host language. This is an intended feature of miniKanren, and does have its uses on occasion (e.g. `build-num` from the relational arith-

[1] Except vectors, which are used in the implementation and of necessity should not be used by the programmer as miniKanren terms.

metic suite). So syntactically restricting the programmer to miniKanren primitives is not a sufficient solution.

It is, however, unfortunate to allow this specialized language feature to make miniKanren programming across-the-board so much more difficult, when programmers, especially beginning ones, will often only use the primitives defined in the miniKanren language in their programs. Our solution is to abandon a one-size-fits-all approach, and instead embrace a suite of different *language levels* [9] of increasing sophistication and freedom that come with additional burdens on the programmer. We propose a small series of little languages organized into a tiered system that provides the programmer with development environments of varying degrees of restriction for writing miniKanren relations. Towards these ends we have made significant progress, laying much of the groundwork for the tasks to come (outlined in section 7).

Our paper makes the following contributions:

- We propose a series of languages meant to teach relational programming where each successive programming language exposes more of the complexities of miniKanren by allowing more of the embedding language.
- We present the first language in this series, a very restricted miniKanren implementation with a suite of syntax macros designed to give the programmer precise and descriptive error messages when writing relational programs.
- We discuss design details for the second language in this proposed series. It is a language that is meant to be transitionary, extending the first language level in ways that facilitate the acquisition of skills the programmer may find useful when working in the increasingly freer environments of the tiers above.
- We also present the last little languages of this series: two implementations of the full miniKanren language, one minimally restricted and the other completely free of restrictions.
- We demonstrate the variety of errors these macros prevent and provide experimental evidence showing how they can be used to the advantage of beginning and seasoned logic programmers alike.

We begin by offering a brief refresher on the miniKanren language. Then, we present a situation that is representative of the kinds of debugging a miniKanren programmer of any skill level is likely to encounter and that proves rather unfriendly to new students.

2. The miniKanren language

Here, we briefly recapitulate the operators and operations of miniKanren. We begin by describing the operators, and conclude with an example of their usage. A more thorough introduction to miniKanren can be found in *The Reasoned Schemer* [11].

A miniKanren program is a goal. A goal is run in an initial, empty state, and the result is formatted and presented to the user. A goal is a function that takes a state and returns a stream (a somewhat lazily-evaluated list) of answers. This goal may be the combination of several subgoals, either their conjunction or disjunction. In the pure subset of the original miniKanren, we have one *atomic* goal constructor, $\equiv$. A goal constructor such as $\equiv$ takes arguments, in this instance two terms u and v, and returns a goal. Applying that goal to a given state returns a stream, possibly empty. The goal constructed from $\equiv$ succeeds when the two terms u and v *unify*, that is, when they can be made syntactically equal relative to a binding of free variables.

Our implementation of miniKanren also includes *disequality constraints*, introduced with the miniKanren operator $\neq$. Disequality constraints are in some sense a converse of goals constructed with $\equiv$. In a given state, a disequality constraint between two terms u and v fails if, after making u and v syntactically equal, the state has not changed. Otherwise, the disequality constraint succeeds, but if another, later goal causes them to become syntactically equal, failure will result.

Individual goals constructed with $\equiv$ and $\neq$ are in and of themselves only so useful. To write more interesting programs, we need a mechanism by which we can build the conjunction and disjunction of goals. The operator that allows us to build these more complex goals is `conde`. `conde` takes as arguments a sequence of *clauses*. A clause is a sequence of goal expressions, and for the execution of a `conde` clause to succeed the conjunction of all of its goals must succeed. The clauses of the `conde` are executed as a nondeterministic disjunction; for the `conde` to succeed, at least one of its clauses must succeed. A `conde` expression evaluates to a goal that can succeed or fail.

Often, when executing a miniKanren program, we need to introduce auxiliary logic variables. The miniKanren operator `fresh` allows us to do this. `fresh` takes a list of variable names, and a sequence of goal expressions; new variables with those names are introduced and lexically scoped over the conjunction of the goals. Like `conde`, a `fresh` expression evaluates to a goal that can succeed or fail.

Because miniKanren is an embedded DSL, we utilize the host language's ability to define and invoke (recursive) functions to build goal constructors and invoke (recursive) goals. Goals constructed from these user-defined goal constructors can be used wherever goals created from the primitive miniKanren operators can be used.

Finally, we use `run` to execute a miniKanren program. `run` takes a maximal number n of desired answers, a variable name, typically q, and a sequence of goal expressions. A new variable is lexically scoped to the name q; this is the variable with respect to which the final answers will be presented. The program to be executed is taken as the goal that is the conjunction of the goal expressions provided to `run`. The run* operator is similar to run, except that instead of a maximal number of answers, we request *all* of the answers.[2]

Consider the following miniKanren program and it's execution:

```
> (define (no-tago tag l)
    (conde
      ((≡ '() l))
      ((fresh (a d)
         (≡ `(,a . ,d) l)
         (≠ a tag)
         (no-tago tag d)))))
> (run 2 (q)
    (fresh (x y)
      (≡ `(,x ,y) q)
      (no-tago x `(a ,y b))))
(((_.0 _.1) (≠ ((_.0 _.1)) ((_.0 a)) ((_.0 b)))))
```

The Racket program no-tago is a user-defined goal constructor; it takes two arguments and returns a goal. This is a conde with two clauses. The first clause consists of a single goal, the requirement that l be '(). The second clause too consists of a single goal. This goal is created from fresh; it requires that two new variables a and d be introduced, and that three things then be the case: that l decompose into two parts a and d, that a not be equal to tag, and that no-tago hold over tag and d.

In the invocation of this program we ask run for at most two answers, with respect to some variable q. The program itself is a single goal that is the result of a fresh. We freshen two new variables x and y, and require two things be the case: that q be the same as a list of x and y, and that no-tago hold of x and the list `(a ,y b). There is in fact only one result to this query. The result is a list containing both the final answer, and a list of the disequality constraints on the answer. The answer itself is a representation of the list (x y); since x and y remain *fresh* in the final answer, they are printed in miniKanren's representation of fresh variables. Fresh variables in miniKanren are represented as _.n, for a zero-based integer index n. The list of disequality constraints ensures that variable x be distinct from variable y, and from symbols 'a and 'b.

With these operators, we are equipped to implement relatively complicated miniKanren programs. The canonical implementation adds impure operators for committed-choice and "if-then-else" behavior, as well as debugging printing operators. Other implementations add more sophisticated run primitives and additional constraints.

3. The problem at present

Suppose one is writing a relation to generate the infinite set of natural numbers as defined by the Peano axioms. Peano numbers are a simple way of representing the natural numbers using only a zero value and a successor function; here, 'z represents the number zero, '(s . z) the number one, '(s s . z) the number two, and so on. We would hope that when running peano for 9 answers, we would get an output similar to that below.

```
> (run 9 (q) (peano q))
'(z
  (s . z)
  (s s . z)
  (s s s . z)
  (s s s s . z)
  (s s s s s . z)
  (s s s s s s . z)
  (s s s s s s s . z)
  (s s s s s s s s . z))
```

Upon opening up a Racket REPL and loading up miniKanren, we flesh out our definition of peano:

```
> (define peano
    (λ (n)
      (cond
        (≡ 'z n)
        ((fresh (n-)
           (= `(s . ,n-) n)
           (peano n-))))))
```

Since Racket accepts this definition, we can use it to try and execute the program.

```
> (run 9 (q) (peano q))
ERROR ⇒
 ...lang/mk.scm:596:24: application:  not a procedure;
 expected a procedure that can be applied to arguments
  given: 'z
  arguments...:
   '(((#(q) #(q))) () () () () ())
```

This error message is not very helpful. Intriguing, though, is its reported source: mk.scm. This error is not reported as coming directly from any code we've written, but rather from the implementation of miniKanren itself: it is a Racket-level exception, though caused by miniKanren code. Since it is unlikely that the code we just wrote somehow broke our miniKanren implementation, we can assume that the real source of the bug is in our definition of peano. Taking another look at our implementation, we discover what we believe to be the 'true' source of the error:

```
(define peano
  (λ (n)
    (cond
      (≡ 'z n)
      ((fresh (n-)
         (= `(s . ,n-) n)
         (peano n-))))))
```

We were erroneously using Racket's equality operator = instead of the miniKanren unification operator ≡; this typo is a common mistake. Correcting the issue should give us a working definition of peano:

[2] In the case of an infinite stream of answers, the execution will appear to hang, and must be aborted. But here's the rub: how does one determine if a program has produced an infinite stream of answers, or merely an obscenely large one?

```
> (define peano
    (λ (n)
      (cond
        (≡ 'z n)
        ((fresh (n-)
           (≡ `(s . ,n-) n)
           (peano n-))))))
> (run 9 (q) (peano q))
ERROR ⇒
 ...lang/mk.scm:596:24: application:  not a procedure;
 expected a procedure that can be applied to arguments
  given: 'z
  arguments...:
   '(((#(q) #(q))) () () () () ())
```

...but it doesn't. That is the same error: did we not just fix the problem? Apparently, the problem we fixed, while certainly a bug, is not the root of this particular error. So, since the error message certainly doesn't help us, we need to scrutinize our code a bit more. Where is the issue, here?

```
(define peano
  (λ (n)
    (cond
      (≡ 'z n)
      ((fresh (n-)
         (≡ `(s . ,n-) n)
         (peano n-))))))
```

Aha! In the definition of our relation we used `cond`, not the miniKanren primitive `conde`. This is another common error, but hopefully now our debugging is complete.

```
> (define peano
    (λ (n)
      (conde
        (≡ 'z n)
        ((fresh (n-)
           (≡ `(s . ,n-) n)
           (peano n-))))))
> (run 9 (q) (peano q))
ERROR ⇒
 ...lang/mk.scm:653:18: ≡: arity mismatch;
 the expected number of arguments does not match the given
 number
  expected: 2
  given: 1
  arguments...:
   '(((#(q) #(q))) () () () () ())
```

This time, we at least get a different error message. Not a particularly helpful one, we admit: we're still seeing Racket-level exceptions filtering up through the DSL. This is a frustrating and tiring experience for the uninitiated, and irksome, at the very least, to a relational programming expert. We push forward, and unmask what will turn out to be the final bug in this bit of code:

```
(define peano
  (λ (n)
    (conde
      (≡ 'z n)
      ((fresh (n-)
         (≡ `(s . ,n-) n)
         (peano n-))))))
```

The `conde` form takes a sequence of goal sequences, but what was intended to be the first sequence is merely a single goal expression: it is missing a set of parentheses. Parenthesis mistakes such as these are also a frequent source of errors in miniKanren programming, as in Racket programming generally. These are especially troublesome when they pass compilation and manifest only at run-time. After correcting this, our relation can now, at last, generate results.

```
> (define peano
    (λ (n)
      (conde
        ((≡ 'z n))
        ((fresh (n-)
           (≡ `(s . ,n-) n)
           (peano n-))))))
> (run 9 (q) (peano q))
'(z
  (s . z)
  (s s . z)
  (s s s . z)
  (s s s s . z)
  (s s s s s . z)
  (s s s s s s . z)
  (s s s s s s s . z)
  (s s s s s s s s . z))
```

The exceptions we have seen thus far have been thrown by Racket from the code that the miniKanren DSL is *generating*, not by any code expressly written by the user. Racket is left to interpret both our miniKanren code and the miniKanren implementation as a single Racket program. This, to a certain extent, undermines the purpose of a DSL.

Ideally, our miniKanren implementation will check for syntax mistakes like the foregoing when a program is defined. miniKanren users should not be confronted with host-level exceptions when writing or executing their relations. As it stands, miniKanren implementations do not check for such things, and there is nothing syntactically wrong as far as *Racket*, the host language, is concerned. Therefore, the code generated by miniKanren compiles, and mistakes such as these slip through to run-time, where our program's source and the miniKanren implementation are blended together in Racket.

4. Our approach

We see great potential in a system that provides the user with control over how much non-relational code their programs can contain. The purest of miniKanren environments would completely disallow non-relational pieces of code, while at the other end of the spectrum a boundless miniKanren would embrace the blending of functional and relational constructs.

Such a safety system would have other benefits, as well: a strict programming environment would not only help the user in the creation of valid relational programs, but could also be used to aid the mechanical transformation process of Racket code to miniKanren relations. As the user grows more comfortable with the relational style of programming, she may choose to give herself more non-relational freedom by removing the training wheels, so to speak, and switching to a less restricted environment.

We envision this safety system as a series of *language levels* having four tiers. The first three tiers are packaged with a standard library of miniKanren relations that is expanded and restricted at various levels.

- At the lowest level sits the purest miniKanren environment (section 5.2): no non-relational code allowed, but the library provides convenience functions (e.g. `conso`, `caro`) to help the user prepare for short-hand notations (e.g. `quasiquote` syntax) that are prohibited at this level. All definitions must reduce to either literals, goals, or relations.
- One tier higher (section 7.1), the environment is now slightly tolerant of non-relational code, and adds a suite of pure arithmetic relations [16] to the standard library, allowing the user to work with numbers in a purely relational manner for the first time. The programmer now has the ability to use the more advanced, short-hand notations for relations like `conso`; namely, those which require `quasiquote`.
- The next tier, described in section 5.1, allows the user to intermingle Racket code with miniKanren code, at their own risk. With non-relational code comes non-relational definitions, increased complexity, and the potential for insidious bugs due to a less-restrictive environment.
- The final tier (also in section 5.1), the highest and freest level, holds bare-bones miniKanren: all code allowed everywhere and no miniKanren-specific syntax checking enforced. Users must rely solely on host-level error handling. (This is the current state of miniKanren.) The standard library is still available, and has grown to include a variety of more sophisticated, powerful relations.

We have reimplemented miniKanren as a Racket language, utilizing Racket's excellent language definition facilities. Using this implementation, we reduce the process of working with this DSL to a single `require` statement:

```
Welcome to DrRacket, version 6.1  [3m].
Language: miniKanren; memory limit: 128 MB.
> (require miniKanren)
> (define succeed (≡ #f #f))
> (run 1 (q) succeed)
'(_.0)
```

This version of miniKanren is completely bare-bones: no syntax checking has been provided.

It is also possible to declare miniKanren as the chosen language for a Racket file:

```
;; in aFile.rkt
#lang miniKanren
(define succeed (≡ #f #f))
```

```
Welcome to DrRacket, version 6.1  [3m].
Language: miniKanren; memory limit: 128 MB.
> (run 1 (q) succeed)
'(_.0)
```

In addition, we provide an implementation of miniKanren with a microKanren core [14], located in the `miniKanren/micro` collection. This language comes with a minimal amount of syntax macros sitting between the user and the implementation, macros that take advantage of Racket's powerful `syntax-parse` macro system [6].

For example, the system catches the missing set of parentheses from the `conde` situation visited above, rejecting it as invalid syntax:

```
> (require miniKanren/micro)
> (define peano
    (λ (n)
      (conde
        (≡ n 'z)
        ((fresh (n-)
           (≡ n `(s . ,n-))
           (peano n-))))))
ERROR ⇒ conde: expected a goal expression
  parsing context:
   while parsing a sequence of goals in: ≡
```

In the following section, we provide details on the two modular implementations of miniKanren and our implementation of the lowest language level, a little language dubbed `freshman`.

These implementations are built as Racket language modules, meaning the user can simply type

```
#lang language-level
```

and begin programming in the specified language level. The language can also be loaded into a Racket file using a standard `require` statement.

5. Implementation

Modularizing the miniKanren embedded DSL was facilitated by the design of Racket's language model. In this model, syntax parsing is divided into two discrete layers: the *reader* layer, which turns a sequence of characters into lists, symbols, and other constants; and the *expander* layer, which processes those lists, symbols, and other constants to parse them as an expression. In effect, this division of labor makes defining a language at the expander layer of syntax parsing, while simultaneously sharing the reader layer of Racket, a relatively simple matter.

The modules we present are just such languages; they utilize the Racket reader while providing additional rules for parsing syntax at the expander layer. In this way, our little languages take full advantage of the system created by the developers of Racket, and with minimal effort provide suitable extensions or restrictions to that system.

At the root of our macro system is a set of miniKanren-specific syntax classes. These classes describe what it means to be an artifact of the miniKanren language (e.g. a *goal-expr*, a *relation*, etc.), and, along with the astonishing power of `syntax-parse`, are directly responsible for the simplicity and extensibility of our system. For more on syntax classes, see "Fortifying Macros" by Culpepper and Felleisen. [6]

We provide the following syntax classes to be used to enforce syntax-checking of miniKanren artifacts:

- `goal-expr`—A *goal-expression* is an expression that reduces to a goal.
- `goal-cons`—A *goal-constructor* is a function of one or more arguments that returns a goal.
- `goal-seq`—A *goal-sequence* is a sequence of goals to be evaluated.
- `relation`—A *relation*, for our purposes, is a function of one or more arguments whose body reduces either to a goal or another relation.

To demonstrate their use, take the following definition of the `fresh` form from the microKanren kernel:

```
(define-syntax fresh
  (syntax-rules ()
    ((_ () g0 g* ...) (conj+ g0 g* ...))
    ((_ (x0 x* ...) g0 g* ...)
     (call/fresh
      (λ (x0)
        (fresh (x* ...) g0 g* ...))))))
```

Imposing the miniKanren syntax-checking on `fresh` is as simple as converting it to `syntax-parse` notation and utilizing the provided miniKanren syntax classes appropriately:

```
(define-syntax (fresh stx)
  (syntax-parse stx
    ((_ () g0:goal-expr g*:goal-expr ...) #'(conj+ g0 g* ...))
    ((_ (x0:id x*:id ...) g0:goal-expr g*:goal-expr ...)
     #'(call/fresh
          (λ (x0)
            (fresh (x* ...) g0 g* ...))))))
```

The remainder of this section is dedicated to describing in detail the various languages we currently provide in our miniKanren relational development environment.

5.1 Collection: `miniKanren`

This collection provides a pair of top-level miniKanren implementations: a module-based implementation of miniKanren, as described by Friedman et. al. in *The Reasoned Schemer* [11]; and a miniKanren with a minimal functional core, described in full detail by Hemann and Friedman in their 2013 Scheme Workshop paper [14].

The `miniKanren` collection is itself a language—it is the first module-based implementation in the above list. In addition to the relational features of the canonical miniKanren implementation, it also provides *disequality constraints*. The inclusion of such constraints in our language levels is currently being discussed and may change in future work. This implementation has been designed to run in both Chez Scheme and Racket; it is implemented as a thin layer of a handful of Racket function aliases atop the Scheme implementation.

In this collection, the `micro` module provides a version of miniKanren with a microKanren core. On top of the core is a layer of usage macros that provides the familiar interface and behavior of miniKanren. An end user could in principle program directly in the language of the microKanren core, however the language is too low-level to be practical for many real programs.

This language has also been supplemented with a small selection of syntax macros specifically designed for use with the miniKanren DSL. These macros do not remove the programmer's ability to intermingle functional and relational code (e.g. `cond` is a valid construct), but they do, however, restrict the programmer's ability to use functional code within such relational constructs like `conde` and `fresh`. Because the syntax classes are attached to the miniKanren code constructs themselves, the syntax-checking is only applied to the bodies of those constructs.

5.2 Collection: `mk`

This collection houses our tiered set of language levels. Each level is a restricted (and sometimes an extended) version of a miniKanren with a microKanren kernel. The miniKanren root implementation is functionally equivalent to the one provided by the `miniKanren/micro` collection; however, this one is written entirely in Racket and has no Scheme dependencies.

Unlike with the `miniKanren` collection, this collection does not double as a language module itself; one cannot simply require `mk` or use `mk` as their `#lang` language as they could with `miniKanren`: they must choose a language module contained within this collection.

This collection houses the lowest little language in our level system: `freshman`. Like the top-level miniKanren implementations, it has been implemented as a Racket module that can be used via `#lang` or `require`.

`freshman` is designed to be a purely relational miniKanren, in which all user-defined functions must be relations, and all definitions must reduce to either a relation or literal value (excluding a function); top-level goals are disallowed. The standard `define` now introduces bindings strictly for literals; `freshman` introduces a new special form, `define-relation`, with which to define relations.

```
> (require mk/freshman)
> (define non-relation 'literal )
> (define-relation the-answer 42)
ERROR ⇒ define-relation: expected a relation of one or more
  arguments in: 42
> (define-relation not-the-answer (λ (x) x))
ERROR ⇒ define-relation: expected a goal-expression or
  expected a relation of one or more arguments
  parsing context:
   while parsing a relation of one or more arguments in: X
> (define-relation the-real-answer (λ (x) (≡ x 42)))
>
```

This relational purity only goes so deep: currently, no method is in place for keeping functional code from being used in the place of arguments to relations. For example, the expression `(conso (lambda (x) x) foo bar)` is valid in `freshman`. The reason for this is the pattern used for parsing goals (defined in `mk/lib/mk-stx.rkt`):

```
(define-syntax-class goal-expr
  #:description "a␣goal-expression "
  (pattern (p:goal-cons y:expr ...+)))
```

Restrictions are currently placed only on the operator; the arguments can be any valid *Racket* expression. In future work we plan to further flesh out the faculties needed to fully enforce a purely relational environment.

All relations must begin with either a `fresh` or a `conde`, and we enforce the convention that the relation identifier end in the letter 'o'. If a relation is defined that does not follow convention, an exception is thrown that suggests an alternative identifier that *does* adhere to convention.

```
> (define relation (≡ #f #t))
> (define relation
    (λ (x)
      (fresh (y)
        (≡ x y))))
ERROR ⇒ define: relation identifier must end in -o,
 suggested change: relation -> relationo in:
  (define relation (λ (x) (fresh (y) (≡ x y))))
```

This identifier convention is enforced only on relations; definitions that reduce merely to goals are exempt from scrutiny.

`freshman` relations must also have at least one argument; side-effects are not allowed at this level, and without them a nullary relation would be utterly useless.

5.3 Analysis and Results

In an effort to measure the effectiveness of the current state of our system, we took a semester's worth of miniKanren code written by Indiana University undergraduates last year and ran it on the `miniKanren/micro` and `mk/freshman` language levels of our system. There were 561 relations in total that were fed to the system, divided among 84 students. We collated the syntax errors generated at both the `minikanren/micro` and `mk/freshman` levels, keeping a count of the distinct exceptions caught, then analyzed the data. The counts are shown in Table 1 below.

The left-column identifies the language level at which the errors listed in the middle column occurred. The right-most column contains a breakdown of the frequency of the errors caught at each level. As you can see, `freshman` caught the same types of errors as `micro`; however, it caught more of them, and also caught a few new ones; these fall below the horizontal line under the `freshman` section. In total, the most restrictive language level caught nearly twice the number of syntax errors as did the least restrictive language level. The discrepancy in types of errors caught is due to the variance in the restrictions placed on the user at each language level.

Table 1. Error spread for the top-level miniKanren's

Level	Error	Count
`micro`	did you mean conde?	1
	**X* may not be a goal constructor	17
	expected identifier	1
	*expected a goal-expression	27
	expected a goal-expression	4
	Pure relation errors:	6
	Blended relation errors:	44
	Total errors:	**50**
`freshman`	did you mean conde?	4
	X may not be a goal constructor	23
	*expected identifier	1
	expected identifier	2
	*expected a goal-expression	28
	expected a goal-expression	4
	`define` expected λ	18
	relation id must end in -o	1
	Pure relation errors:	10
	Restricted relation errors:	71
	Total errors:	**81**

We compared the errors reported by the syntax macros to the reports generated by an autograder currently being used to evaluate student miniKanren submissions at Indiana University. We found that the vast majority of the student code that generated syntax errors *also* generated Racket-level exceptions when allowed to run in an unrestricted environment. This suggests that many, if not all, such exceptions were completely preventable given a proper programming environment.

A few of the error categories highlight areas of the system that need improvement. '*X* may not be a goal constructor', in particular, is thrown every place a `goal-expr` is expected but a variable or other expression is given. The current algorithm for determining if an expression evaluates to a goal is very simple, and very dumb: if the operator identifier of the expression does not end in -o, the `goal-expr` is rejected. This check, of course, will always fail if there is no operator, as in the case of a variable or some other value, regardless of if it actually evaluates to a goal. Once a more intelligent algorithm is developed, this error will only be reported when a non-relational expression is given in place of an expected `goal-expr`.

Relatedly, 'expected a goal-expression' errors are thrown when users attempt to use variables in the place of `goal-exprs`, whether or not these variables actually refer to goals. The reason, again, being `goal-exprs` lack of smarts: it has no knowledge of which bindings in the current environment point to goals (or, indeed, knowledge of any bindings at all); and because there is currently no value difference between a miniKanren goal and a standard Racket function (they both evaluate to `#<procedure>`), it cannot perform any value checking that would distinguish the two.

These issues are common to both `miniKanren/micro` and `freshman` and are discussed at length in section 7.2.

Crunching the numbers a bit more, we find that 88 percent of errors caught at the `micro` level were caused by 'blended relations', that is, miniKanren relations that attempted to utilize non-relational features of the host language. This number is, however, severely bloated due to the issues with how goal-expressions are identified, discussed above.

Attempted use of restricted features account for roughly the same percentage of `freshman` errors; these restricted features mainly include the usage of non-relational code (a `let` statement, perhaps) and the ability to define top-level goals. For example, if one were to try and define the canonical `succeed` or `fail` relations, `(define succeed (== #f #f))` and `(define fail (== #f #t))` respectively, a '`define` expected λ' exception would occur. Errors resulting from the use of such restricted features are viewed as evidence of the importance of proper programming environments and the benefits gained from using our fail-fast system: miniKanren programmers are alerted of any syntax mistakes or potential dangers as close to their source as possible.

Though no formal user study has been attempted to assess what improvements our system make to the experience of debugging miniKanren relations, in appendix A we provide the output of a REPL session in which a severely broken and blended miniKanren relation, `member?o`, is nursed back to relational health by exclusively following the error messages thrown by `freshman`.

6. Related Work

The microKanren kernel itself bears a strong relationship to the kernel of Spivey and Seres' "Embedding Prolog in Haskell" [19], and to Oleg Kiselyov's Sokuza Kanren [15]. Like microKanren, both of these works have a basic model of conjunction, disjunction, introducing new logic variables, and an operator to perform unification.

Our work here has been explicitly modeled on the DrRacket model of teaching languages. We feel a strong analogy between the way the teaching languages of DrRacket aid beginning students in writing functional programs and the way these teaching languages aid the programmer in writing relational programs [9]. It seems possible that their method for introducing computer programming to students early in their education might, with the appropriate languages, features and guidance, help students learn relational programming as well [8].

7. Conclusions and further work

Through the restrictions in this tiered series of little relational languages, we can provide to users of any skill level more descriptive error messages to aid development. We found that when programming at these levels, most run-time errors encountered by novice programmers became instead straightforward syntax errors.

From this experiment it is clear that many common mistakes made by miniKanren users are preventable, provided they are made in an environment that can handle them. The programmer can choose a language level that suits their relational needs, depending on the types of functional freedoms they wish to have in their programs. miniKanren initiates, who are presumably unfamiliar with either the relational programming paradigm or the syntax of miniKanren, can make use of the tiered system in such a way that helps them learn the language. Starting at the most restricted level, `freshman`, they may choose to 'level up' as they become more comfortable with writing miniKanren programs and find themselves wanting to take advantage of the embedding language features in order to write increasingly complex relations.

The very act of writing this document brought to light strengths, weaknesses, and potentialities that were heretofore hidden from view, and many design decisions were modified. This is a trend that will surely continue, as miniKanren and its rapidly expanding community of relational programmers are still in their infancy. Below, we present our vision for the 'missing link', as it were, in our system of relational language levels, as well as improvements and further work to be done.

7.1 `sophomore`: Level up

Here we introduce the not-yet-implemented second little language, `sophomore`. `sophomore` is intended to sit as a middle level between `freshman` and full miniKanren. We imagine the `sophomore` DSL extending `freshman` in ways that give the programmer more freedoms with the miniKanren language, naturally increasing the range of legal relations the programmer can write. The following design ideas represent our expectations as to what trade-offs `sophomore` should make between safety and flexibility, though these may change with discoveries as we implement and test our designs.

The user is now granted a very limited ability to blend functional and relational code in their programs: the programmer will have the ability to write non-relational code, but only in a non-relational context. That is to say, functional and relational pieces of code can coexist in the same program, interacting with each other, but the programmer may not use a non-relational construct like `(map pred '(1 2 3))` inside of a relational one, like `fresh` or `conde`.

The standard library packaged with this level has been augmented to include the relational arithmetic library. This library provides a variety of numeric predicates and relations that perform such functions as basic mathematical operations like addition and subtraction, in addition to more involved arithmetic like logarithms and inequalities. These relations operate on an encoding of little-endian binary numbers, which facilitates relational arithmetic.

In order to facilitate programming with this suite, the standard library at this level also provides `build-num`, a function that translates decimal numbers to little-endian binary numbers.

```
> (build-num 11)
'(1 1 0 1)
> pluso
#<procedure:pluso>
> (run 1 (q) (pluso (build-num 2) (build-num 9) q))
'((1 1 0 1))
```

7.2 Improvements

There is one issue that has been uncovered during the course of this experiment. It affects both the `miniKanren/micro` and `mk/freshman` language levels, and has to do with the types of expressions valid at each level.

Prior to running this experiment, `freshman` users were unable to define top- level goals, such as `succeed` or `fail`. This was discussed, and we decided this was the desired behavior: top-level goals should be unavailable to `freshman` users, as their main concern is with writing well-formed relations. The discussion surrounding this issue, however, brought to light another: the 'purity guards', if you will, placed upon the `conde` and `fresh` forms only allow `goal-expr`s in their bodies, and fail to recognize when a top-level variable was defined with a `goal-expr`.

In other words, `freshman` users cannot use variables in the place of `goal-expr`s. Future versions of `freshman` will somehow need to keep track of bound goals as they are defined such that they can be recognized by the syntax-checker.

Future work must also be done in the area of not only restricting the presence of host-language features in miniKanren programs, but also the usage of the Racket standard library. If we are going to prevent the programmer from using a function, the simplest way would be to not provide the function in the first place. Though a simple idea, actually decoupling the standard library from the user may prove to be quite a feat. An effective and perhaps more easily implemented alternative might be to simply restrict user access to the *bindings* of the Racket functions.

We mention the 'miniKanren standard library' multiple times throughout this paper: this has not yet been designed nor implemented, and so more needs to be done in this area, as well. The relational arithmetic suite that will be introduced into this library by `sophomore` exists, but needs to be formally documented along with the rest of this project.

7.3 Dedicated miniKanren tools

We believe that a better-tailored programming environment that supports proper development and maintenance tools would help novice users prevent or eliminate many of their most common errors. Even unsophisticated programming environments, that offer a bare minimum of programmer comfort (e.g. syntax highlighting), help programmers avoid many bugs. [7]

DrRacket provides an excellent graphical environment for developing programs using the Racket programming languages, featuring a sophisticated debugger, an algebraic stepper, and support for user-defined plug-ins, in addition to source highlighting for syntax and run-time errors. Embedded languages such as miniKanren, however, have syntax that extends or otherwise differs from the host language, and tools meant for the host language do not, and cannot, naïvely map to such extensions.

As it stands, no tools exist that have been tailored to developing in the miniKanren relational programming language. By extending the development environment of miniKanren to match the language extensions [5], we hope to change this, providing the first steps toward a sophisticated development environment for working in miniKanren.

A. Relational debugging with `freshman`

```
Welcome to DrRacket, version 6.1 [3m].
Language: mk/freshman; memory limit: 128 MB.
> (define member?o
   (λ (x ls out)
     (conde
       ((≡ '() ls) (≡ #f out))
       ((fresh (a d)
          (≡ `(,a . ,d) ls)
          (cond
            ((equal? x a) #t)
            ((not (equal? x a)) (member?o x d out))))))))
. fresh: did you mean "conde"?
  parsing context:
   while parsing a goal constructor
   while parsing a goal-expression in: cond
> (define member?o
   (λ (x ls out)
     (conde
       ((≡ '() ls) (≡ #f out))
       ((fresh (a d)
          (≡ `(,a . ,d) ls)
          (conde
            ((equal? x a) #t)
            ((not (equal? x a)) (member?o x d out))))))))
. conde: equal? may not be a goal constructor
  (identifier doesn't end in -o)
  parsing context:
   while parsing a goal constructor
   while parsing a goal-expression
   while parsing a sequence of goals in: equal?
> (define member?o
   (λ (x ls out)
     (conde
       ((≡ '() ls) (≡ #f out))
       ((fresh (a d)
          (≡ `(,a . ,d) ls)
          (conde
            ((≡ x a) #t)
            ((not (equal? x a)) (member?o x d out))))))))
. conde: expected a goal-expression
  parsing context:
   while parsing a sequence of goals in: #t
> (define member?o
   (λ (x ls out)
     (conde
       ((≡ '() ls) (≡ #f out))
       ((fresh (a d)
          (≡ `(,a . ,d) ls)
          (conde
            ((≡ x a) (≡ #t out))
            ((not (equal? x a)) (member?o x d out))))))))
. conde: not may not be a goal constructor
  (identifier doesn't end in -o)
  parsing context:
   while parsing a goal constructor
   while parsing a goal-expression
   while parsing a sequence of goals in: not
> (define member?o
   (λ (x ls out)
     (conde
       ((≡ '() ls) (≡ #f out))
       ((fresh (a d)
          (≡ `(,a . ,d) ls)
          (conde
            ((≡ x a) (≡ #t out))
            ((≠ x a) (member?o x d out))))))))
> (define list-uniono
   (λ (s1 s2 out)
     (conde
       ((≡ '() s1) (≡ s2 out))
       ((fresh (a d)
          (≡ `(,a . ,d) s1)
          (fresh (b)
            (member?o a s2 b)
            (conde
              ((≡ b #t) (list-uniono d s2 out))
              ((≡ b #f)
               (fresh (res)
                 (≡ `(,a . ,res) out)
                 (list-uniono d s2 res))))))))))
> (run1 (q) (fresh (a b) (≡ q `(,a ,b)) (list-uniono a b '(1 2))))
'((() (1 2)))
> (run3 (q) (fresh (a b) (≡ q `(,a ,b)) (list-uniono a b '(1 2))))
'((() (1 2)) ((1 2) ()) ((1) (1 2)))
> (run9 (q) (fresh (a b) (≡ q `(,a ,b)) (list-uniono a b '(1 2))))
'((() (1 2))
  ((1 2) ())
  ((1) (1 2))
  ((1) (2))
  ((1 1) (1 2))
  ((2) (1 2))
  ((2 1) (2))
  ((1 2) (2))
  ((1 1 1) (1 2)))
```

References

[1] C. E. Alvis, J. J. Willcock, K. M. Carter, W. E. Byrd, and D. P. Friedman. cKanren: miniKanren with constraints. *Scheme and Functional Programming*, 2011.

[2] J. Bentley. Programming pearls: little languages. *Communications of the ACM*, 29(8):711–721, 1986.

[3] C. Brozefsky. core.logic and SQL killed my ORM, 2013. URL http://www.infoq.com/presentations/Core-logic-SQL-ORM.

[4] W. E. Byrd, E. Holk, and D. P. Friedman. miniKanren, live and untagged: Quine generation via relational interpreters (programming pearl). In *2012 Workshop on Scheme and Functional Programming*, Sept. 2012.

[5] J. Clements and K. Fisler. "Little language" project modules. *Journal of Functional Programming*, 20:3–18, 1 2010. ISSN 1469-7653. . URL http://journals.cambridge.org/article_S0956796809990281.

[6] R. Culpepper and M. Felleisen. Fortifying macros. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, pages 235–246, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-794-3. . URL http://doi.acm.org/10.1145/1863543.1863577.

[7] M. Ducassé and J. Noyé. Logic programming environments: Dynamic program analysis and debugging. *The Journal of Logic Programming*, 19 - 20, Supplement 1(0):351 – 384, 1994. ISSN 0743-1066. . URL http://www.sciencedirect.com/science/article/pii/0743106694900302. Special Issue: Ten Years of Logic Programming.

[8] M. Felleisen, R. B. Findler, M. Flatt, and S. Krishnamurthi. The TeachScheme! project: Computing and programming for

every student. *Computer Science Education*, 14(1):55–77, 2004.

[9] R. B. Findler, C. Flanagan, M. Flatt, S. Krishnamurthi, and M. Felleisen. Drscheme: A pedagogic programming environment for scheme. In *Programming Languages: Implementations, Logics, and Programs*, pages 369–388. Springer, 1997.

[10] D. Friedman. C311/B521/A596 programming languages, 2014. URL `https://cgi.soic.indiana.edu/~c311/doku.php?id=home`.

[11] D. Friedman, W. E. Byrd, and O. Kiselyov. *The Reasoned Schemer*. MIT Press, Cambridge, Mass, 2005. ISBN 9780262562140.

[12] J. Gibbons. Functional programming for domain-specific languages. *Central European Functional Programming-Summer School on Domain-Specific Languages (July 2013)*, 2013.

[13] D. Gregoire. Web testing with logic programming, 2013. URL `http://www.youtube.com/watch?v=09zlcS49zL0`.

[14] J. Hemann and D. Friedman. microkanren: A minimal functional core for relational programming. In *Proceedings of Scheme Workshop*, 2013.

[15] O. Kiselyov. The taste of logic programming, 2006. URL `http://okmij.org/ftp/Scheme/misc.html#sokuza-kanren`.

[16] O. Kiselyov, W. E. Byrd, D. P. Friedman, and C. Shan. Pure, declarative, and constructive arithmetic relations (declarative pearl). In J. Garrigue and M. Hermenegildo, editors, *Proceedings of the 9th International Symposium on Functional and Logic Programming*, volume 4989 of *LNCS*. Springer, 2008.

[17] J. P. Near, W. E. Byrd, and D. P. Friedman. αleanTAP: A declarative theorem prover for first-order classical logic. In *Proceedings of the 24th International Conference on Logic Programming (ICLP 2008)*, volume 5366 of *LNCS*, pages 238–252. Springer-Verlag, Heidelberg, 2008.

[18] R. Senior. Practical core.logic, 2012. URL `http://www.infoq.com/presentations/core-logic`.

[19] J. Spivey and S. Seres. Embedding Prolog in Haskell. In *Proceedings of Haskell Workshop*, volume 99, pages 1999–28, 1999.

[20] C. Swords and D. Friedman. rKanren: Guided search in miniKanren. In *Proceedings of Scheme Workshop*, 2013.