A Surprisingly Competitive Conditional Operator

miniKanrenizing the Inference Rules of Pie

BENJAMIN STRAHAN BOSKIN, Indiana University
WEIXI MA, Indiana University
DAVID THRANE CHRISTIANSEN, Galois, Inc
DANIEL P. FRIEDMAN, Indiana University

Because miniKanren programs can run forwards and backwards, interpreters written in miniKanren can perform program synthesis. Because program synthesis for dependently typed languages is also proof synthesis, miniKanren implementations of dependently typed languages can be used for proof search. Unfortunately, cond^e's unaided complete interleaving depth-first search does not yield a practical proof search due to the sheer amount of computation required. A new conditional operator, called cond^p, gives users a means to drop irrelevant goals from search. We demonstrate that cond^p provides sufficient control over the search process to perform synthesis far more quickly.

Additional Key Words and Phrases: miniKanren, Dependent Types, Program Synthesis

1 INTRODUCTION

miniKanren, described in *The Reasoned Schemer, 2nd Ed.* [2018] (TRS2) by Friedman, Byrd, Kiselyov, and Hemann, is a relational programming language. A function is a special case of a relation, so functional programs can be readily translated into miniKanren. The process of translating a function into a miniKanren relation is called *miniKanrenization*. As Byrd et al. [2017, 2012] demonstrate, miniKanrenizing functional programs makes it possible to run programs *backwards*, finding elements of the domain that compute given elements of the range. As a result, interpreters written in miniKanren are automatically program synthesis tools, and miniKanren proof checkers are proof search tools. This power comes from the complete interleaving depth-first search performed by miniKanren's flagship conditional operator, cond^e, and the ability to place unification variables, here simply referred to as *variables*, in place of expressions and proofs.

The power of miniKanren's search comes, however, at a steep cost: the search space grows exponentially with respect to the problem. While many miniKanren programs are small enough that the exponential growth does not present issues, unaided search is impractical for many interesting programs. The exponential growth is a result of disjunctions, because cond^e must examine each disjunct, or *line* in miniKanren parlance. We present a new conditional operator for miniKanren, called cond^p, which allows users to preemptively recognize irrelevant lines, and *prune* them from search.

During execution, miniKanren maintains a *substitution*, which is a data structure that associates variables with their values. Internally, the distinctness of variables is established using allocation effects and pointer equality tests—a typical implementation uses distinct zero-length vectors. A substitution combined with zero-length vectors is not, however, particularly easy for most people to read. *Reification* is the process of building legible *values* that contain all ground data associated with a variable in a substitution. When reification is performed in full, any variables remaining in the reified term are replaced with symbols such as $\__0$, $\__1$, etc., for readability, where the same symbol is given to variables that co-refer. cond^p allows users to perform *partial reification* mid-computation, which applies the current substitution but leaves the underlying representation of fresh variables

Authors' addresses: Benjamin Strahan Boskin, Indiana University, bboskin@indiana.edu; Weixi Ma, Indiana University, mvc@indiana.edu; David Thrane Christiansen, Galois, Inc, dtc@galois.com; Daniel P. Friedman, Indiana University, dfried@indiana.edu.

2018. XXXX-XXXX/2018/9-ART
https://doi.org/

in the resulting values. This allows cond^p lines to be pruned when they are guaranteed to fail, while freshness of variables remains known.

The effect of dropping cond^p lines is similar to that of using disequality constraints, described by Comon [1990], because both methods, like unification, can limit search. Disequality constraints, however, cannot prevent lines from being executed when performed on fresh variables, and never change the number of lines included in search. cond^p lets programmers create flexible search structures, without needing effects on variables nor the state.

When cond^p is used carefully, it generates the same results as cond^e but faster, having restricted the search space to lines that might succeed. In addition, new search strategies become possible, such as including special-case lines that are not always used, or multiple versions of similar lines that are never used at the same time. With great power, however, comes great responsibility—when used without care, cond^p programs can betray their cond^e predecessors, and fail unexpectedly if too many lines are dropped from search.

In Section 2, we describe the usage of cond^p and its implementation, for which an understanding of TRS2 will suffice. In Section 3, we show runtime comparisons between cond^p and cond^e, using different miniKanrenizations of a lightweight, dependently typed language called Pie as the example. Pie is described by Friedman and Christiansen [2018] in *The Little Typer* (TLT). In Section 4, we further discuss Pie, outlining a miniKanren implementation, and show how computation-heavy inference rules can be miniKanrenized. In dependent type theory, a program that inhabits a type is a proof of the proposition expressed by its type, so miniKanren-style program synthesis is also a form of proof search. Pie is complex enough, however, that cond^e's unaided complete interleaving depth-first search is impractical. For this section, familiarities with miniKanren interpreters, inference rules, and dependent types are assumed. We use Pie as an example, primarily because it is complicated enough that cond^p is able to provide significant improvement. The version of miniKanren we use to implement Pie uses hash tables to represent substitutions, as well as attributed variables, described by Huitouze [1990], for disequality constraints, adding to the language described in TRS2.

All code shown can be found at https://github.com/bboskin/SFPW2018.

2 A NEW CONDITIONAL OPERATOR

We present cond^p, a new conditional operator, that lets users control search, with the ability to *prune* irrelevant lines from search. While cond^p does not improve the order of time complexity, the number of (possibly recursive) goal calls is greatly decreased, which reduces running time and space usage.

A problem with typical conditional operators that drop goals from search, like miniKanren's cond^a and Prolog's `cut`, is that the order in which goals are written dictates which goals are pruned: all goals below the first line with a sucessful first goal are pruned. With cond^p, however, search is restricted by user-defined functions, called *suggestion functions*. Suggestion functions use partially-reified variables to decide which cond^p lines should be included in search. Because suggestion functions are *not* written in miniKanren, they are able to use tricks unavailable to miniKanren programs. These tricks include obtaining knowledge of variable freshness, as well as anything useful that can be done in the host language.

Suggestion functions can be thought of as a way to tell a relation when certain cond^p lines are and are not relevant. Suppose you're walking through your town, passing many establishments, and looking for a particular grocery store. You hope to enter as few establishments as possible other than this particular grocery store. If you were using cond^e to get there, however, you'd stop in each one and make sure they weren't your intended destination. You might even continue to look after finding the store, with purchased groceries already in your arms. The suggestion functions used with cond^p let miniKanren relations ignore irrelevant establishments, on their way to a particular destination.

The goal when designing suggestion functions is to convey how search would be performed by hand. When performing informal search, paths are often left out because they will soon be irrelevant. Everything knowable by human searchers should be expressible to miniKanren. With suggestion functions, lines that are guaranteed to fail can be preemptively dropped from search, as would be done in a by-hand search. The fact that a line *can* be used should not necessarily imply that it *is* used.

In addition, each `cond`^{p} line is given a *key* that allows suggestion functions to refer to lines to attempt. The process of taking suggestions and dropping lines accordingly happens at each entrance to a `cond`^{p}. This results in a search tree that is tailored to the situation at hand.

2.1 Differences in relevance

miniKanren queries run both *forwards* and *backwards*. A query runs forwards when input variables are *ground data*, which is data that does not contain variables. Input variables are arguments that strongly dictate the direction of computation, and can include proof terms, and source language expressions to evaluate. A query runs backwards when input variables either are or contain variables.

Because the placement of variables in a query can vary, however, suggestion functions must do more than suggest `cond`^{p} lines. They must discern the relative importance of variables, and favor suggestions from certain variables as needed. Input variables are generally sufficient to guide search when queries are run forwards, and `cond`^{p} recognizes this. A hierarchy of relevance for suggestions is available: there are preliminary variables that are *always* considered, and there are variables that are *maybe* considered, only when the previous level suggests to keep going. The result of considering variables, or applying suggestion functions to values, is a *list of suggestions*.

Returning to the example of grocery shopping: suppose you know that you want to go grocery shopping at a *particular* store in a town that has many stores. (This is similar to having a variable, `destination`, that has ground data associated with it.) Alternatively, it may only be known that grocery shopping is the task at hand (where `destination` is fresh, while another variable, `task`, has ground data). In this example, `destination` is the variable that you *always* consider, and `task` is only *maybe* considered, when `destination` isn't helpful. Although knowing your `task` may not limit your options to a single store, it can restrict the number of stores to consider.

Variables that are *maybe* considered are only used when the keyword `use-maybe` is included in the suggestions coming from the previous level. If the suggestions from the *always* variables include `use-maybe` then the first *maybe* level is considered. Then, for each *maybe* level, if the suggestions from that level include `use-maybe`, then the next *maybe* level is also considered. These relevance hierarchies are similar to the constraint-based compile-time modes developed by Overton et al. [2002] for Mercury, but `cond`^{p} and its modes are used entirely at runtime.

Without the ability to establish differences in relevance, optimal pruning could not coexist with generality. In the case of interpreters, for example, the suggestion functions desired for output variables are less strict than those for input variables, to maintain that all possible evaluations remain available. If such suggestions made for generality were not ignorable, then more `cond`^{p} lines than necessary would often be suggested. Establishing differences in relevance among variables is a crucial part of using `cond`^{p}, maintaining that optimal pruning is possible, while all possible paths to solutions are preserved.

Because the goal when writing suggestion functions is to minimize the number of `cond`^{p} lines that are included in search, it is easy to write suggestion functions that over-prune. When designing suggestion functions, especially as multiple levels of relevance are in play, one needs to keep in mind what is known about each relevant variable, and ensure that in all cases, the set of lines that *might* succeed is a subset of the set of lines that are actually

```
(conde
  (g ...)
  ...)
```

```
(condp
    (((f x) ...) ...)
  (key g ...)
  ...)
```

Fig. 1. Syntax of cond^e and cond^p

```
(defrel (swap-someo ls o)
  (conde
    ((== '() ls) (== '() o))
    ((fresh (a d res)
       (== `(,a . ,d) ls)
       (== `(,a . ,res) o)
       (swap-someo d res)))
    ((fresh (a d res)
       (== `(,a . ,d) ls)
       (== `(novel . ,res) o)
       (swap-someo d res)))))
```

```
> (run* (q r)
    (swap-someo q `(novel ,r)))

'(((novel _0) _0)
  ((novel _0) novel)
  ((_0 _1) _1)
  ((_0 _1) novel))
```

Fig. 2. Definition of swap-some^o and an example query

suggested. As long as this property is ensured, programs written with cond^p can be trusted to behave like their cond^e predecessors.

The syntax of cond^p and cond^e, shown in Figure 1, differ only slightly. Each cond^p line has a `key` at the front, used as its identifier by suggestion functions. In addition, cond^p has a *prelude*, where all suggestion functions and variables that should be used for suggestions are listed. Each level of relevance is wrapped in parentheses, and as many variables as desired can be used at each level. Each tuple `(f x)`, where `f` is a suggestion function and `x` is a variable, becomes an application, but is performed after `x` has been partially reified in the current substitution.

The idea to give each line a special first element initially came from rKanren, by Swords and Friedman [2013]. rKanren has a special operator, called cond^r, where cond^r lines begin with numerical weights, which collectively determine an ordering in which lines are executed.

2.2 An introductory example of cond^p

Consider the relation swap-some^o, shown in Figure 2, which takes a list and swaps an uncertain number of its elements with the symbol `novel`. In the first cond^e line, `ls` is the empty list, and since there are no values to swap, `o` is the empty list. In the second cond^e line, `ls` is a pair, the `car` of `ls` is the `car` of `o`, and recursion is performed on the `cdr` of `ls`, `d`, and the `cdr` of `o`, `res`. In the final cond^e line, the `car` of `ls` is swapped with the symbol `novel`, which is the `car` of `o`, and recursion is again performed on `d` and `res`.

We now turn swap-some^o into a cond^p relation that decreases the number of lines used for each goal expression. To do this, we must first choose keys for the three cond^e lines, and then define suggestion functions using these keys, writing one for `ls` and one for `o`. For this example, we use upper-case letters for keys, to make them stand out. The key `BASE`, arbitrarily chosen, is used for the first line, which is the base case of this recursive relation;

```
(define (ls-keys-init ls)                              (define (ls-keys ls)
  (cond                                                  (cond
    ((var? ls) '(BASE KEEP SWAP))                          ((var? ls) '(use-maybe))
    ((null? ls)  '(BASE KEEP SWAP))    simplifies to:      ((null? ls) '(BASE))
    ((pair? ls) '(BASE KEEP SWAP))                         ((pair? ls) '(KEEP SWAP))
    (else '(BASE KEEP SWAP))))                             (else '())))

(define (o-keys-init ls)                               (define (o-keys o)
  (cond                                                  (cond
    ((var? ls) '(BASE KEEP SWAP))                          ((var? o) '(BASE KEEP SWAP))
    ((null? ls) '(BASE KEEP SWAP))     simplifies to:      ((null? o) '(BASE))
    ((pair? ls) '(BASE KEEP SWAP))                         ((pair? o)
    (else '(BASE KEEP SWAP))))                              (if (or (var? (car o))
                                                                    (eqv? 'novel (car o)))
                                                                '(KEEP SWAP)
                                                                '(KEEP)))
                                                           (else '())))
```

Fig. 3. Suggestion functions for swap-some^p

KEEP is used for the second line, where the `car` of `ls` is kept; and SWAP is used for the last line, where the `car` of `ls` is swapped with the symbol novel. Using these keys, we define the suggestion functions shown in Figure 3. First, we write a prototype suggestion function for `ls`, `ls-keys-init`, that always suggests every swap-some^p line, and will cause behavior that is identical to cond^e. Then, for any cond^p line that is guaranteed to fail when `ls` has a certain value, that line is dropped from the list of suggestions offered in that case in the final suggestion function, and use-maybe is used when necessary.

The reasoning for `ls-keys` is as follows: when `ls` is fresh, use-maybe is suggested, because `ls` has no information, and o may have more to offer. When `ls` is `'()`, only BASE can succeed. When `ls` is a pair, BASE is guaranteed to fail, but both KEEP and SWAP can succeed. Finally, if `ls` is neither a variable, `'()`, nor a pair, then no lines can succeed, and the empty list of keys is returned.

Next, a similar method is applied to o, starting with a suggestion function that suggests all lines, and eliminating those that are guaranteed to fail. Because of the way we wrote `ls-keys`, we also know that if `o-keys` is being used, then `ls` is a fresh variable. (For this relation, however, we certainly could have placed `o-keys` in the *always* category instead, and only *maybe* used `ls-keys`.)

When o is fresh, then all three lines are suggested, as they can all succeed. When o is `'()`, then only BASE is suggested. When o is a pair, however, then more analysis is needed. If its `car` is either the symbol `novel` or a fresh variable, then both KEEP and SWAP can succeed. If the `car` is any other ground term, however, then only KEEP is suggested, because SWAP is guaranteed to fail. Otherwise, no lines are suggested. The final cond^p definition, swap-some^p, is shown in Figure 4.

Next we show the effect that variables have on the conditional structures produced at each step of a cond^p, shown in Figure 5. Consider the following query: (run* q (swap-some^p q `(book novel))).

Initially, `ls` is fresh, and o is a list whose `car` is book. So, `ls-keys` suggests use-maybe, and `o-keys`, then suggests the only line that can succeed, which is KEEP. Conceptually, the resulting goal is equivalent to Step 1 in Figure 5.

```
(defrel (swap-some^p ls o)
  (cond^p
     (((ls-keys ls))
      ((o-keys o)))
    (BASE (== '() ls) (== '() o))
    (KEEP (fresh (a d res)
            (== `(,a . ,d) ls)
            (== `(,a . ,res) o)
            (swap-some^p d res)))
    (SWAP (fresh (a d res)
            (== `(,a . ,d) ls)
            (== `(novel . ,res) o)
            (swap-some^p d res)))))
```

Fig. 4. cond^p definition of swap-some^o

```
Step 1:
  (cond^e
    ((fresh (a d res)
       (== `(,a . ,d) ls)
       (== `(,a . ,res) o)
       (swap-some^p d res))))
```

```
Step 2:
  (cond^e
    ((fresh (a d res)
       (== `(,a . ,d) ls)
       (== `(,a . ,res) o)
       (swap-some^p d res)))
    ((fresh (a d res)
       (== `(,a . ,d) ls)
       (== `(novel . ,res) o)
       (swap-some^p d res))))
```

```
Step 3:
  (cond^e
    ((== '() ls) (== '() o)))
```

Fig. 5. Conditional structures for step-some^p

In the resulting call to swap-some^p, `ls` is still fresh, and `o` is a pair whose `car` is `novel`. So, `o-keys` suggests that both KEEP and SWAP be used, reflecting the fact that `novel` could have been either an element of `ls`, or the result of a swap. The resulting goal is conceptually equivalent to Step 2.

Finally, in the two identical recursions made in Step 2, `ls` is still fresh, and `o` is `'()`. This results in a goal that is conceptually equivalent to Step 3, and no more recursions are performed.

The definition of swap-some^o written in Figure 2 begs for some simplification, however, because of the overlapping uses of `fresh` between the second and third lines. One might prefer to write swap-some^o as shown in Figure 6. Check your understanding by converting the second definition of swap-some^o into a relation with nested occurrences of cond^p.

2.3 Another wire to connect

Chapter 10 and "Connecting the Wires" in TRS2 describe an implementation of miniKanren with only equality constraints. To use cond^p, the macros shown in Figure 7 can be added to that implementation, available at miniKanren.org, as though a part of "Connecting the Wires."

cond^p expands to a goal expression, a function taking a substitution. After a substitution has been passed, suggestions are collected using the macro `collect`: starting with the first list of the prelude, `((f x) ...)`, each

```
(defrel (swap-some^o ls o)
  (cond^e
    ((== '() ls) (== '() o))
    ((fresh (a d res)
       (== `(,a . ,d) ls)
       (cond^e
         ((== `(,a . ,res) o))
         ((== `(novel . ,res) o)))
       (swap-some^o d res)))))
```

Fig. 6. An alternate definition of swap-some^o

```
(define-syntax collect
  (syntax-rules ()
    ((collect s) '())
    ((collect s ((f0 x0) ...) ((f x) ...) ...)
     (let ((ulos (append (f0 (walk* x0 s)) ...)))
       (if (memv 'use-maybe ulos)
           (append ulos (collect s ((f x) ...) ...))
           ulos)))))

(define-syntax cond^p
  (syntax-rules ()
    ((cond^p (((f x) ...) ...) (key g ...) ...)
     (lambda (s)
       (let ((los (collect s ((f x) ...) ...)))
         ((disj (if (memv 'key los) (conj g ...) fail) ...) s))))))
```

Fig. 7. The definition of cond^p

variable `x` is given to a function `walk`*, which performs partial reification. The resulting partially-reified value is then passed to its suggestion function, `f`. Typically, each `x` is unique, and each `f` is unique, but uniqueness is not required. The suggestions gleaned from each list is an *unfinished list of suggestions*, `ulos`. If `use-maybe` is present in `ulos`, then the complete list of suggestions includes, at a minimum, suggestions from the next relevance level as well, and is built using recursion. If `ulos` does not contain `use-maybe`, however, then the collecting of suggestions stops, at which point the complete list of suggestions, `los`, has been formed.

Using `los`, the suggested cond^p lines are put into a disjunction, while non-suggested lines are replaced with `fail`, which is equivalent to dropping them entirely. In the version of cond^p shown in Figure 7, the dropping of lines is only *staged*. By playing with the implementation, however, this can, of course be changed. Readers who want to use cond^p are encouraged to experiment with these options, and to create the cond^p that they find most suited to their needs! Variations that we have explored include using a helper macro to prevent introducing extraneous `fail`s, and replacing lists of suggestions with sets of suggestions, using Racket sets.

3 BENCHMARKING COND^p WITH IMPLEMENTATIONS OF PIE

We present some examples of the effect that well-chosen suggestion functions can have on the time taken for miniKanren to execute a query by comparing three miniKanren implementations of Pie. These Pie implementations have been developed towards the implementation using cond^p, and to confirm the effectiveness of cond^p. First, we have developed a full implementation of Pie's inference rules against which a backwards implementation can be tested, but which cannot run backwards, because it uses cond^u, and thus is not a part of the tests shown. Next, we have constructed Pie^e, an implementation of a subset of the Pie language using cond^e, that (in theory) performs program synthesis but (in practice) is too slow to be useful. Then, we have designed and implemented Pie^p, the faster implementation using cond^p. Finally, we have used Pie$^e/c$, which is Pie^e with three added guards using disequality constraints, which prevent serious goal calls. Comparing Pie^p and Pie$^e/c$ has confirmed that Pie^p is competitive even in the presence of miniKanren constraints. The results are shown visually in Figure 8, and the code for the programs used can be found in Figures 15, 16, and 17 of Appendix A. Because Pie is a new language, these programs are not provided for comprehension, but are merely provided as Pie^p's benchmarks.

3.1 Understanding the data

The tests are run roughly in increasing difficulty, and both Pie^e and Pie$^e/c$ reach a difficulty threshold beyond which they no longer complete their execution in a reasonable amount of time. For these queries, a 'reasonable' amount of time is 5 minutes, although many of them have been left running for several hours and had remained unsolved by Pie^e. Pie^p has such a threshold as well, when asked to synthesize nontrivial lambda terms that satisfy proofs, because this causes lots of computation to be performed with many fresh variables, at which point cond^p's pruning is able to do less. Such tests are not shown in these charts, however, since neither Pie^e nor Pie^e/c compare with its performance at that point.

In the third and final chart, Pie^e is left out, as it no longer completes, meaning that it had not terminated after running for over 12 hours. These programs, however, are where Pie^p proves itself to be faster than Pie$^e/c$. A query taking 15 seconds for Pie^p takes 30 seconds for Pie$^e/c$, and a query taking 90 seconds for Pie^p takes over 600 seconds for Pie$^e/c$.

These comparisons show that cond^p is worth considering adding to the miniKanren toolkit, as it allows a programmer's domain-specific insights to be encoded in programs, cutting out unnecessary computation.

4 A RELATIONAL IMPLEMENTATION OF PIE'S INFERENCE RULES

A logic has three fundamental components: the subjects about which it reasons; the forms of judgment that can be made about these subjects; and the rules, or *inference rules*, that permit new acts of judgment on the basis of prior acts of judgment. Classical First Order Logic (FOL), for example, reasons about propositions, which are built from atomic propositions and connectives such as $\wedge$ and $\Rightarrow$. FOL makes judgments about the truth and falsity of propositions, and uses rules such as modus ponens and the principle of the excluded middle to reach these judgments. In a dependent type theory, the subjects of reasoning are drawn from an open-ended grammar of expressions, and the forms of judgment include that an expression is a type; that two expressions are the same type; that one expression inhabits some type; and that two expressions are the same with respect to their type. In dependent type theory, propositions are expressed as types, and the judgment that a proposition is true is the judgment that another expression inhabits it.

Making a language suitable for machine implementation requires that certain aspects be made more explicit. In the case of Pie, the explicit version of the language is specified with inference rules.

Following these rules, we have reimplemented Pie in miniKanren. While an independent implementation increases confidence that the rules are a correct specification of the language, the choice of miniKanren as an implementation language additionally enables synthesis where behavior is specified by types, rather than by test cases.

Types in Pie are named by *type constructors*, which introduce new types. Each type in Pie may have *data constructors*, often simply referred to as *constructors*, as well as *eliminators*. Constructors are the most direct

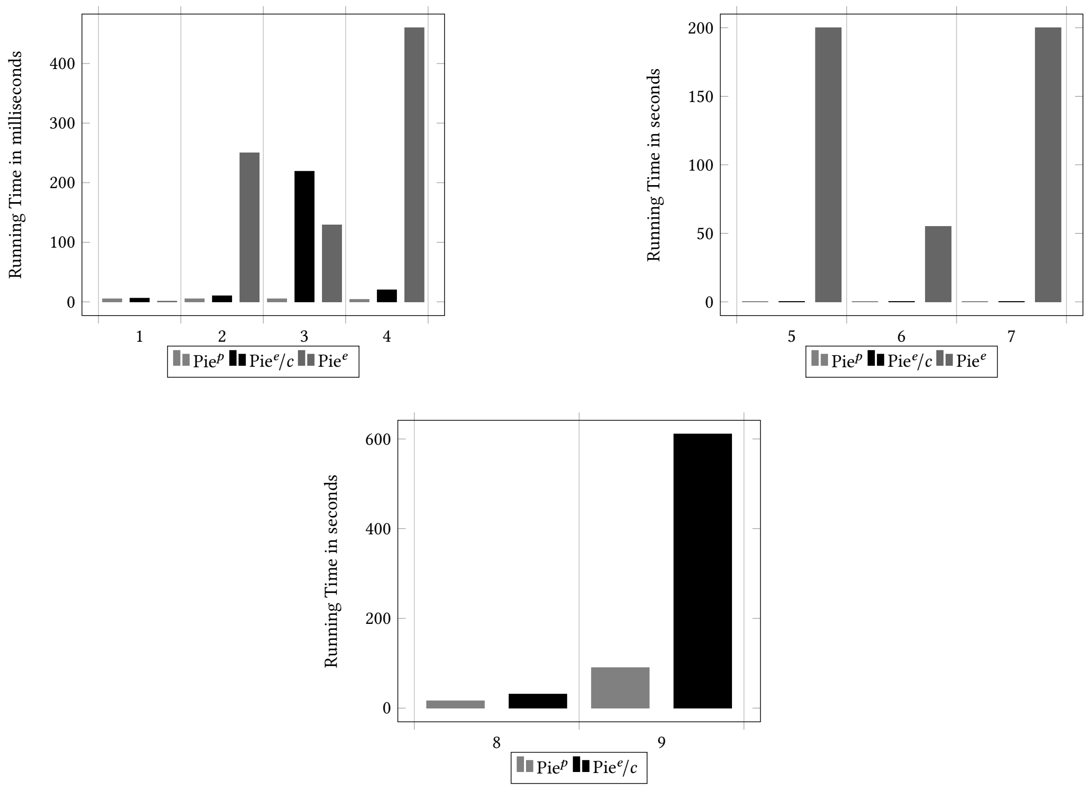

Fig. 8. Comparing running times Pie^p, Pie^e, and Pie^e/c

means of producing members of the type, while eliminators expose the information underneath constructors, allowing members of the type to be used to construct members of arbitrary other types.

The subset of Pie that we consider has six type constructors, three of which construct *dependent types*, which are types that contain expressions that are not themselves types. There is no separation between the syntax of types and the syntax of expressions in Pie. The types of this subset are:

- `Atom`, which is similar to Lisp's symbols, and has an infinite number of constructors, each of which is a symbol preceded by `'`, called a *tick mark*
- `Trivial`, which is Pie's unit type, and has one nullary constructor, `sole`, and no eliminators
- `Nat`, with nullary constructor `zero`, unary constructor `add1`, and inductive eliminator `ind-Nat`
- `=`, which is a dependent type with unary constructor `same`, and inductive eliminator `ind-=`
- Π, which is a dependent type with constructor λ, and is eliminated with function application
- Σ, which is a dependent type with binary constructor `cons` and two unary eliminators `car` and `cdr`
- and `U`, short for *universe*, in which the constructors are the other type constructors, namely `Atom`, `Trivial`, `Nat`, `=`, Π, and Σ.

While some of these types correspond closely to the features of languages in the Scheme family, others may be less familiar. The equality type (`=` X *from* *to*) is a type whose inhabitants are proofs that *from* and *to* are equal expressions of type X. The dependent function type (Π ((x *Arg*)) R) is a type whose inhabitants are functions that take *Arg*s as inputs, and return Rs, where the precise argument supplied has been substituted for x

Form of Judgment	Name	Meaning
$\Gamma \vdash expr$ **type** $\rightsquigarrow expr^e$	Typehood	In Γ, *expr* is a type, and elaborates to $expr^e$
$\Gamma \vdash expr \in T \rightsquigarrow expr^e$	Checking	In Γ, *expr* is a T, and elaborates to $expr^e$
$\Gamma \vdash expr$ **synth** $\rightsquigarrow$ `(the` $T\ expr^e$`)`	Synthesis	In Γ, *expr* elaborates to $expr^e$, and is of type T
$\Gamma \vdash T_1 \equiv T_2$ **type**	Type sameness	In Γ, T_1 and T_2 are equivalent types
$\Gamma \vdash expr_1 \equiv expr_2 : T$	Sameness	In Γ, $expr_1$ and $expr_2$ are equivalent and of type T

Fig. 9. The forms of judgment in Pie

in R. The dependent pair type `(Σ ((x A)) D)` is a type whose inhabitants are pairs whose `car`s are As, and whose `cdr`s are Ds, where x has been substituted in D for the `car` of the pair. To maintain the logical consistency of Pie, `U` is not a `U`.

Pie uses bidirectional typechecking [Pierce and Turner 2000], which requires that some expressions be annotated with their types. To accommodate this, there is another form, `the`, which is used to add type annotations to expressions. For example, λ expressions need type annotations to clarify the expected type of their argument. An expression e is annotated as a T with an expression of the form `(the T e)`.

Pie is based on the Intuitionistic Type Theory of Martin-Löf [1982, 1984]. Accordingly, the types Π and Σ represent quantifiers. Π is interpreted as universal quantification, and Σ is interpreted as existential quantification, where the `car` of a Σ is the witness and the `cdr` demonstrates that the witness fulfills the desired property. These two types, in addition to the = type and the natural numbers, allow many interesting proofs to be written in this small language.

We now show the process by which a set of inference rules can be translated to a set of miniKanren relations, and then how such a set of relations can be merged into a single relation that represents a form of judgment, using cond^p. Farka et al. [2018] provide a more formal and detailed exposition of the relationship between inference rules of dependently typed languages and relational programs, and the translation between the two.

4.1 Judgments in Pie

There are several forms of judgment in Pie. In the judgments shown in Figure 9, Γ is a context, *expr*, $expr^e$, etc. are expressions, and T, T_1, etc. are types.

Because Pie is intended to be implementable in a language in which programs run only forwards, the first three forms of judgment explicitly separate *inputs* from *outputs*, with outputs occurring after the bent arrow, $\rightsquigarrow$. These outputs are typically more-explicit versions of a corresponding input; however, the type synthesis judgment additionally returns the type that was discovered. The process of producing a more-explicit program from a less-explicit source text is referred to as *elaboration*. Adding an ^{e} to the name of a variable, such as changing *expr* to $expr^e$, is used to convey that an expression has been elaborated (this ^{e} should not, however, be confused with the ^{e} in cond^e, which stands for *every*). Forms of judgment without outputs are realized by programs that merely succeed or fail, yielding no further information, while forms of judgment with outputs are realized by programs that may fail, in which success additionally provides the output.

The first form of judgment, typehood, is that an expression is a type. A program that checks this form of judgment is a program that succeeds when provided with a type. The second form of judgment, checking, is that an expression can be checked to have some given type, realized by a program that returns the elaborated version of the inhabitant on success. The third form of judgment, synthesis, is that a type can be discovered by inspecting an expression. On success, synthesis returns the discovered type in addition to the elaborated expression.

The last two forms of judgment respectively indicate that two expressions are the same type, or when they are the same with respect to their type. Because the sameness rules for the function type, the unit type, the pair

$$\frac{\Gamma \vdash \mathit{mid} \in X \rightsquigarrow \mathit{mid}^e \quad \Gamma \vdash \mathit{from} \equiv \mathit{mid}^e : X \quad \Gamma \vdash \mathit{mid}^e \equiv \mathit{to} : X}{\Gamma \vdash (\texttt{same}\ \mathit{mid}) \in (=\ X\ \mathit{from}\ \mathit{to}) \rightsquigarrow (\texttt{same}\ \mathit{mid}^e)}\ \textsc{EqI} \tag{1}$$

$$\frac{\Gamma, x : \mathit{Arg} \vdash r \in R \rightsquigarrow r^e}{\Gamma \vdash (\lambda\ (x)\ r) \in (\Pi\ ((x\ \mathit{Arg}))\ R) \rightsquigarrow (\lambda\ (x)\ r^e)}\ \textsc{FunI} \tag{2}$$

$$\frac{\Gamma \vdash a \in A \rightsquigarrow a^e \quad \Gamma \vdash d \in D[a^e/x] \rightsquigarrow d^e}{\Gamma \vdash (\texttt{cons}\ a\ d) \in (\Sigma\ ((x\ A))\ D) \rightsquigarrow (\texttt{cons}\ a^e\ d^e)}\ \Sigma\mathrm{I} \tag{3}$$

$$\frac{\Gamma \vdash \mathit{expr}\ \mathbf{synth} \rightsquigarrow (\texttt{the}\ X_1\ \mathit{expr}^e) \quad \Gamma \vdash X_1 \equiv X_2\ \mathbf{type}}{\Gamma \vdash \mathit{expr} \in X_2 \rightsquigarrow \mathit{expr}^e}\ \textsc{Switch} \tag{4}$$

Fig. 10. Inference rules for checking

type, and the identity type all take types into account to enable more expressions to be the same, we implement these rules using normalization by evaluation (NbE), described by Berger and Schwichtenberg [1991], in which expressions are first interpreted into values that contain no latent computation, and then are *read back* into the syntax of that value's normal form. Abel [2013] describes how NbE can be used for dependent types by making the read-back procedure take types into account to perform η-expansion. A tutorial on implementing NbE in Racket is available from the third author's Web site.[1]

4.2 Developing a relation

The judgment that we demonstrate miniKanrenization with is checking, to be realized by $\texttt{check}^o$. To judge $\Gamma \vdash \mathit{expr} \in T \rightsquigarrow \mathit{expr}^e$ is to judge that in a context Γ, an expression *expr* has type T, and *expr* elaborates to the expression expr^e. $\texttt{check}^o$ directly handles three Pie expressions: `same`, λ, and `cons`. When $\texttt{check}^o$ is given any other expression, it uses $\texttt{synth}^o$ to synthesize a type, and then uses $\equiv\texttt{-type}^o$ to determine that the synthesized and expected types are equivalent. The inference rules used to justify the type checking judgment are shown in Figure 10, and the miniKanrenized inference rules of EqI, FunI, ΣI, and Switch are shown in Figure 11.

The first rule we miniKanrenize is EqI, which describes how $\texttt{check}^o$ handles `same` terms. The rule EqI has three premises: first ensuring that *mid* is an X, and then that mid^e, the result of elaborating *mid*, is equivalent to both *from* and *to*, also of type X. Then, $(\texttt{same}\ \mathit{mid})$ is confirmed to be of type $(=\ X\ \mathit{from}\ \mathit{to})$, and elaborates to $(\texttt{same}\ \mathit{mid}^e)$. It is not necessary to check that *from* and *to* have type X, because the type being checked against is assumed to have already been checked for typehood. Following this description of EqI, and assuming a miniKanrenization of $\equiv$, called $\equiv^o$, we define $\texttt{EqI}^o$.

The order in which goals appear in a conjunction can play a large role in that conjunction's behavior. Rozplokhas and Boulytchev [2018] demonstrate this by improving the performance of miniKanren programs by dynamically altering the order of goals within conjunctions, to prevent divergence. In general, however, a rule of thumb is to always put serious goal calls, i.e., goals that involve recursions, below simple goals. This may mean doing the last step of an inference rule earlier than would be naturally written, as with the third unification in $\texttt{EqI}^o$, $(\texttt{==}\ \texttt{`(same ,mid}^e\texttt{)}\ \texttt{expr}^e)$. Because inference rules are conjunctions, however, this is fine, and helps ensure that as much information is carried into recursions as is possible, which is especially important when $\texttt{cond}^p$ is being used.

[1] http://davidchristiansen.dk/tutorials/nbe

```
(defrel (EqIᵒ Γ expr T exprᵉ)
  (fresh (X from to mid midᵉ)
    (== `(same ,mid) expr)
    (== `(= ,X ,from ,to) T)
    (== `(same ,midᵉ) exprᵉ)
    (checkᵒ Γ mid X midᵉ)
    (≡ᵒ Γ X from midᵉ)
    (≡ᵒ Γ X midᵉ to)))

(defrel (FunIᵒ Γ expr T exprᵉ)
  (fresh (x r y Arg R Rˢ Γ^ rᵉ Arg_v)
    (non-reserved-Pie-symbolᵒ x)
    (non-reserved-Pie-symbolᵒ y)
    (== `(λ (,x) ,r) expr)
    (== `(Π ((,y ,Arg)) ,R) T)
    (== `(λ (,x) ,rᵉ) exprᵉ)
    (substᵒ x y R Rˢ)
    (valofᵒ Γ Arg Arg_v)
    (extend-Γᵒ Γ x Arg_v Γ^)
    (checkᵒ Γ^ r Rˢ rᵉ)))

(defrel (ΣIᵒ Γ expr T exprᵉ)
  (fresh (a d x A D aᵉ Dˢ dᵉ)
    (non-reserved-Pie-symbolᵒ x)
    (== `(cons ,a ,d) expr)
    (== `(Σ ((,x ,A)) ,D) T)
    (== `(cons ,aᵉ ,dᵉ) exprᵉ)
    (checkᵒ Γ a A aᵉ)
    (substᵒ aᵉ x D Dˢ)
    (checkᵒ Γ d Dˢ dᵉ)))

(defrel (switch-exprᵒ Γ expr T o)
  (fresh (t)
    (synthᵒ Γ expr `(the ,t ,o))
    (≡-typeᵒ Γ T t)))

(defrel (switch-Tᵒ Γ expr T o)
  (fresh (t)
    (≡-typeᵒ Γ T t)
    (synthᵒ Γ expr `(the ,t ,o))))
```

Fig. 11. miniKanrenized inference rules for check^o

Next, we miniKanrenize the rule FunI, for handling λ terms. FunI has one premise, which says that if in an extended context Γ, where x is added as an *Arg*, the expression r checks to be an R and elaborates to r^e, then in Γ, the expression $(\lambda\ (x)\ r)$ is a $(\Pi\ ((x\ Arg))\ R)$, and elaborates to $(\lambda\ (x)\ r^e)$.

For FunI, we assume the existence of a relation `extend-Γ`^{o}, which takes a context, a variable, and a type for that variable, and relates them to an extended context. In our implementations of Pie, contexts hold the *values* of types rather than the syntactic expressions that denote them, so the relation `valof`^{o} is used to evaluate the type `Arg`. `valof`^{o} is the first stage of normalization by evaluation, and the second stage is `read-back`^{o}, where values are brought back to syntactic expressions that are always Pie normal forms.

In addition, there are important details left implicit in FunI that need to be explicit in its miniKanrenization. There are new variables introduced, that need to be confirmed to be unreserved symbols. Pie's `zero`, for example, cannot be a formal parameter. We can use an assumed predicate `non-reserved-Pie-symbol`^{o}, to ensure this. In addition, although they are both x in FunI, the new lexical variables in the λ and Π expressions may be different. These variables must be made the same, as they will now share an entry in the extended context. A relation to perform capture-avoiding substitution, therefore, is needed, and we assume the existence of a relation `subst`^{o}, which takes an expression e, a variable to replace, x, an expression to replace x with, a, and a final expression, o, and performs substitution.

In our miniKanren implementation, `subst`^{o} uses `gensym` to create a new variable when needed. For some relations, uses of `gensym` cause running backwards to be impossible. In `subst`^{o}, whether or not `gensym` is used is driven by the input variable e, depending on whether or not a formal parameter of e occurs in the free variables of the substitution term. The only case in which `gensym` could cause `subst`^{o} to fail is when both the input and output expressions are ground, when a fresh variable is needed to be generated because of the input expression, and the name of that fresh variable is decided in the output expression. Because `subst`^{o} is never called with two ground terms in the Pie implementation, however, and because $\texttt{subst}^o$ is not a relation that users can directly use, it is not a concern. In general, `gensym` *cannot* be used willy-nilly. Following these guidelines, we define $\texttt{FunI}^o$.

Next, we miniKanrenize the rule ΣI, which handles `cons` pairs. ΣI has two premises, which require that the `car` and the `cdr` of the given pair are both well-typed. Further, since the type describing the `cdr` is polymorphic for any x of type A, it needs to be instantiated with the elaborated `car` before typechecking the `cdr`. When both premises are satisfied, the pair (`cons` a d) elaborates to (`cons` a^e d^e), and has the type (Σ ((x A)) R).

By following the description of the rule, and assuming the predicate $\texttt{non-reserved-Pie-symbol}^o$, as well as the relation performing capture-avoiding substitution, $\texttt{subst}^o$, we define $\Sigma\mathrm{I}^o$.

Finally, we miniKanrenize the SWITCH rule, which is used when an expression for which a type can be synthesized is checked. When an expression other than a `same`, λ, or `cons` expression is given to the type checker, a type for the expression is synthesized, and the synthesized type must be equivalent to the expected type. For this definition, we assume a miniKanrenization of type sameness.

Following these guidelines, we are left with two different definitions of SWITCH: $\texttt{switch-expr}^o$ and $\texttt{switch-T}^o$ shown in Figure 11. Both of these definitions are reasonable because the two serious goals, which use $\texttt{synth}^o$ and $\equiv\texttt{-type}^o$, use overlapping sets of variables.

Which of these definitions of SWITCH yields optimal performance? Assuming that all judgments are defined using $\texttt{cond}^p$, variables should always contain as much information as possible. Depending on the freshness of `expr`, T, and `o`, $\texttt{synth}^o$ and $\equiv\texttt{-type}^o$ each can help the other perform efficient search. When `expr` is partially ground, $\texttt{synth}^o$ needs no help and should be performed first. When `expr` is fresh, however, since the variable `t` is also fresh if $\texttt{synth}^o$ is used first, and `o` may be fresh, there is minimal information directing the choice of $\texttt{synth}^o$ lines unless $\equiv\texttt{-type}^o$ is performed first. If T is partially ground, then the information gained about `t` using $\equiv\texttt{-type}^o$ can offer $\texttt{synth}^o$ some information. Both $\texttt{switch-expr}^o$ and $\texttt{switch-T}^o$ are useful, and we definitely want to be able to use both. Because $\texttt{cond}^p$ lets us dynamically choose between one or the other, we get to have our Pie and eat it too.

4.3 Designing suggestion functions

Because $\texttt{check}^o$'s input expression is the most relevant variable to dictate which $\texttt{cond}^p$ line is used, we *always* use its suggestion function. When the input expression is fresh, denoted below by the predicate `var?`, suggestions come from a different suggestion function that examines the other two variables together. When the input variable is sufficiently ground, however, only one line needs to be suggested: if it is a `same`, λ, or a `cons`, then the corresponding line is suggested, and otherwise `switch-expr`. The suggestion function for the input expression is defined in Figure 12. The function `expr-memv?` is a generic way to see if a given value is part of a family described by a list of forms, such as `'(same λ cons)`.

Using convenient keys for our $\texttt{cond}^p$ lines allows us to streamline our suggestion functions, as is seen in the first `match` line above of `check-expr-table`. The keys used for $\texttt{check}^o$ are: `same`, `cons`, λ, `switch-expr`, and `switch-T`.

Next, we define suggestion functions for the expected type and output expression. Because the expected type can restrict lines more than the output expression, `check-T-table` suggests whether or not `o` should be used. By letting these variables drop irrelevant $\texttt{cond}^p$ lines, but keeping all lines that might succeed, we get the functions shown in Figure 13.

Now that we have the required suggestion functions defined, as well as the rules for $\texttt{check}^o$ defined as miniKanren relations, we can define $\texttt{check}^o$ as the miniKanren relation shown in Figure 14.

```
(define ((expr-memv? ls) e)
  (and (pair? e) (memv (car e) ls)))


(define (check-expr-table expr)
  (match expr
    ((? var?) '(use-maybe))
    ((? (expr-memv? '(same λ cons)))
     `(,(car expr)))
    (else '(switch-expr))))
```

Fig. 12. The suggestion function for the input expression to check, with helper functions

```
(define (check-T-table T)
  (match T
    ((? var?) '(use-out))
    (`(= ,X ,from ,to) '(switch-T same))
    (`(Π ((,x ,A)) ,R) '(switch-T λ))
    (`(Σ ((,x ,A)) ,D) '(switch-T cons))
    (else '(switch-T))))
```

```
(define (check-o-table e)
  (match e
    ((? var?)
     '(same λ cons switch-expr))
    ((? (expr-memv? check-exprs))
     `(,(car e) switch-T))
    (else '(switch-T))))
```

Fig. 13. Suggestion functions for T and o

```
(defrel (checkᵒ Γ expr T exprᵉ)
  (condᵖ
      (((check-expr-table expr))
       ((check-T-table T))
       ((check-o-table exprᵉ)))
    (same (EqIᵒ Γ expr T exprᵉ))
    (λ (FunIᵒ Γ expr T exprᵉ))
    (cons (ΣIᵒ Γ expr T exprᵉ))
    (switch-expr (switch-exprᵒ Γ expr T exprᵉ))
    (switch-T (switch-Tᵒ Γ expr T exprᵉ))))
```

Fig. 14. Definition of check^o

4.4 You can't just follow the rules

In Pie, each syntactic form is either checked against a type or has a type synthesized for it. This uniqueness, however, is a global property that is not mentioned in each rule. In particular, SWITCH should not be used for same, λ, and cons, because they are meant to be handled by the other three rules: EQI, FUNI, and ΣI. This is typically established with a miniKanren expression using disequality constraints, or can be enforced with cond^p. In either method, however, information is added to the miniKanrenization that is not present in the inference rule.

There are several advantages to using $\texttt{cond}^p$ instead of disequality constraints to make such nuances explicit. One is that miniKanrenized inference rules are able to remain closer in resemblance to their origins this way. The notion of when a rule is to be used becomes part of the infrastructure of the judgment itself, rather than through changing the meaning of a single rule, or giving extra burden to variables. The need of disequality constraints for preventing when lines are executed can be eliminated with $\texttt{cond}^p$. In addition to the symbolic simplicity, $\texttt{cond}^p$ permits more creative experimentation with adding new lines, as they can be dropped from search when desired.

In our experimentation with Pie, we have found that constraints expected to be seen in the results of full reification should be enforced with disequality constraints. An example of this is using $\texttt{symbol}^o$ to enforce that the `x` in `(λ (x) ...)` be a symbol. Constraints used merely as guards to prevent lines from being executed for efficiency, however, are more suited for $\texttt{cond}^p$.

5 CONCLUSION

The designers of search-based programming tools strive to find a balance between making a tool that is easy to use, and one that models the smart ways in which humans approach problems. Without $\texttt{cond}^p$, miniKanren achieves the first goal, and $\texttt{cond}^p$, only adding the work of a few small suggestion functions, brings miniKanren closer to achieving the second goal. When faced with many options as to how to proceed towards a solution, the first step made by humans, which is so basic that it is typically unconscious, is to forget immediately about the options that are, for the moment, irrelevant. This is what $\texttt{cond}^p$ gives programmers: the ability to drop irrelevant goals from search. It does not, however, support reasoning as sophisticated as, for example, purpose-built systems such Lindblad and Benke [2006]'s Agsy, nor does Pie^p regularly solve proof goals as interesting as those solvable through human-directed tactics, such as the techniques described by Chlipala [2013]. $\texttt{cond}^p$ is easy to use, and provides miniKanren programmers with a straightforward method to improve the search performed by $\texttt{cond}^e$ which, despite its innocence, pays off.

We hope that demonstrating our enhancement of search using $\texttt{cond}^p$ leads to other novel approaches to miniKanren operators. The capability of transitioning between miniKanren and a host language may bring about many new or unexpected results.

Additionally, we would like to add nominal unification to this project, to improve how Pie^p finds equivalence between α-equivalent terms, using a method described by Ma et al. [2018].

ACKNOWLEDGMENTS

In addition, we thank Michael Ballantyne for bringing attributed variables to miniKanren constraints. His implementation, which can be found at https://github.com/michaelballantyne/faster-miniKanren, largely inspired the miniKanren implementation behind Pie^p.

REFERENCES

Andreas Abel. 2013. *Normalization By Evaluation: Dependent Types and Impredicativity.* Springer, Institut für Informatik, Ludwig-Maximilians-Universität München. OCLC: ocn436029031.

U. Berger and H. Schwichtenberg. 1991. An Inverse of the Evaluation Functional for Typed Lambda-Calculus. IEEE Comput. Sco. Press, 203–211. https://doi.org/10.1109/LICS.1991.151645

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 (Aug. 2017), 8:1–8:26. https://doi.org/10.1145/3110252

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

Adam Chlipala. 2013. *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant.* The MIT Press.

Hubert Comon. 1990. Solving Symbolic Ordering Constraints. *International Journal of Foundations of Computer Science* 1, 4 (Dec. 1990), 387–411.

František Farka, Ekaterina Komendantskya, and Kevin Hammond. 2018. Proof-relevant Horn Clauses for Dependent Type Inference and Term Synthesis. *ICLP* (2018). arXiv: 1804.11250.

Daniel P. Friedman, William E. Byrd, Oleg Kiselyov, and Jason Hemann. 2018. The Reasoned Schemer, Second Edition. The MIT Press. (2018).

Daniel P. Friedman and David Thrane Christiansen. 2018. The Little Typer. The MIT Press. (2018).

Serge Le Huitouze. 1990. A New Data Structure for Implementing Extensions to Prolog. In *Programming Language Implementation and Logic Programming (Lecture Notes in Computer Science)*. Springer, Berlin, Heidelberg, 136–150. https://doi.org/10.1007/BFb0024181

Fredrik Lindblad and Marcin Benke. 2006. A Tool for Automated Theorem Proving in Agda. In *Types for Proofs and Programs*, Jean-Christophe Filliâtre, Christine Paulin-Mohring, and Benjamin Werner (Eds.). Springer Berlin Heidelberg, 154–169.

Weixi Ma, Jeremy Siek, David Christiansen, and Daniel Friedman. 2018. *Efficiency of a Good but Not Linear Nominal Unification Algorithm.* EasyChair Preprints. EasyChair. https://doi.org/10.29007/9x4c

P. Martin-Löf. 1982. Constructive Mathematics and Computer Programming. In *Proc. Of a Discussion Meeting of the Royal Society of London on Mathematical Logic and Programming Languages*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 167–184.

Per Martin-Löf. 1984. *Intuitionistic Type Theory.* Number 1 in Studies in Proof Theory Lecture Notes. Bibliopolis, Napoli. OCLC: 12731401.

David Overton, Zoltan Somogyi, and Peter J. Stuckey. 2002. Constraint-based Mode Analysis of Mercury. In *Proceedings of the 4th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP '02)*. ACM, New York, NY, USA, 109–120. https://doi.org/10.1145/571157.571169

Benjamin C. Pierce and David N. Turner. 2000. Local Type Inference. *ACM Trans. Program. Lang. Syst.* 22, 1 (Jan. 2000), 1–44. https://doi.org/10.1145/345099.345100

Dmitri Rozplokhas and Dmitri Boulytchev. 2018. Improving Refutational Completeness of Relational Search via Divergence Test. In *Proceedings of the 20th International Symposium on Principles and Practice of Declarative Programming (PPDP '18)*. ACM, New York, NY, USA, 18:1–18:13. https://doi.org/10.1145/3236950.3236958

Cameron Swords and Daniel P. Friedman. 2013. rKanren: Guided Search in miniKanren *(SFPW '13)*. http://www.schemeworkshop.org/2013/papers/Swords2013.pdf

A CODE FOR BENCHMARKS FROM SECTION 3

In these programs, the relation `pie`^{o} is an interface to the judgments of Pie, which behaves like the Pie interpreter but is relational, rather than functional. It takes a list of Pie expressions, `prog`, and relates those expressions to `o`, a list of the results of elaborating all expressions in `prog` that are not top-level definitions. In the case where all provided expressions are top-level definitions, `o` will be `'()`. Top-level definitions are made with `claim/define` forms, which combine Pie's `claim` and `define` forms into a single expression.

A.1 Describing the queries

Programs 1 through 4 are short Pie programs: none of them involve top-level definitions, nor inductive eliminators.

Programs 5 through 7 are slightly more involved. Program 5 uses three top-level definitions. The first definition is a polymorphic identity function, `foo`, whose type is concretely given, but whose body is a variable. The second definition is another polymorphic identity function, `bar`, that is entirely ground. Finally, the third is a short proof that shows that `foo` and `bar` should behave the same, and a definition for `foo` is synthesized.

Program 6 uses two top-level definitions, both of which have concrete types and unknown bodies. The first type describes a function `f` that takes two `Nat`s and returns a `Nat`. The second type describes a proof of commutativity of `f`, called `f-comm`. The query synthesizes both a function body for `f`, and a proof for `f-comm`.

Program 7 uses two top-level definitions. The first is arithmetic addition, `+`, which inductively eliminates the first number given. The second definition is a proof that for all `n`, `((+ zero) n)` is the same `Nat` as `n`. The proof is entirely ground, but the definition of `+` has one variable in its body, the value returned when `n` is zero. The value needed here is, therefore, clarified by the proof provided, and a value to finish the definition of `+` is synthesized.

Program 8 and Program 9 consider the same definition of `+` as Program 7, and a proof that for all `n`, `((+ n) zero)` is the same `Nat` as `n`. Using dependent types, this is a much more invovled proof than the one used in Program 7, because it requires induction. Program 8 leaves the definition of `+` as a variable, and asks for an expression that satisfies the given proof. The definition of `+` for Program 9 is all ground data, however, and only typechecking is needed.

1.
```
(run 1 (q r)
  (pieᵒ `((add1 (add1 ,q)))
    `((the Nat ,r))))

'((zero (add1 (add1 zero))))
```

2.
```
(run 1 type
  (pieᵒ
    `(((the ,type ;; type is synthesized.
         (λ (x)
           x))
       (add1 zero)))
    '((the Nat (add1 zero)))))

'(((Π ((_₀ Nat)) Nat)))
```

3.
```
(run 1 pair
  (pieᵒ
    `(((the (Π ((x Nat))
              Atom)
         (λ (n)
           'hello))
       (car ,pair))) ;; pair is synthesized.
    `((the Atom 'hello))))

'(((the (Σ ((_₀ Nat)) Nat)
     (cons zero zero))))
```

4.
```
(run 1 q
  (pieᵒ
    '(((the
         (Π ([x (Σ ([x Nat])
                  (= Nat x x))])
           Nat)
         (λ (pr)
           (car pr)))
       (the (Σ ([x Nat])
              (= Nat x x))
         (cons (add1 zero)
           (same (add1 zero))))))
    q))

'(((the Nat (add1 zero))))
```

Fig. 15. Small examples of Pie evaluation

5.
```
(run 1 fun
  (pieᵒ
    `((claim/define bar
        (Π ((X U))
          (Π ((x X)) X))
        ,fun) ;; fun is synthesized.
      (claim/define foo
        (Π ((X U))
          (Π ((x X))
            X))
        (λ (X)
          (λ (x) x)))
      (claim/define foo=bar
        (Π ((Z U))
          (Π ((z Z))
            (= Z ((foo Z) z)
                 ((bar Z) z))))
        (λ (A)
          (λ (a)
            (same a)))))
    '()))

'(((λ (x) (λ (x) x))))
```

6.
```
(run 1 (fun proof)
  (pieᵒ
    `((claim/define f
        (Π ((n Nat))
          (Π ((m Nat))
            Nat))
        ,fun) ;; fun is synthesized.

      (claim/define f-comm
        (Π ((n Nat))
          (Π ((m Nat))
            (= Nat ((f n) m)
                   ((f m) n))))
        ,proof)) ;; proof is synthesized.
    '()))

'(((λ (m) (λ (var) zero))
   (λ (_0) (λ (_1) (same zero)))))
```

7.
```
(run 1 base
  (pieᵒ
    `((claim/define +
        (Π ((n Nat))
          (Π ((m Nat))
            Nat))
        (λ (n)
          (λ (m)
            (ind-Nat n
              (λ (n) Nat)
              ,base ;; base is synthesized.
              (λ (x)
                (λ (res) (add1 res)))))))
      (claim/define +-zero-l
        (Π ((n Nat))
          (= Nat n ((+ zero) n)))
        (λ (n) (same n))))
    '()))

'((m))
```

Fig. 16. Examples of syntheses driven by types

8.
```
(run 1 body
  (pie^o
    `((claim/define +
        (Π ([n Nat])
          (Π ([m Nat])
            Nat))
        (λ (n)
          (λ (m)
            ,body))) ;; fun is synthesized
      (claim/define +-zero-r
        (Π ([n Nat])
          (= Nat n
                 ((+ n) zero)))
        (λ (n)
          (ind-Nat n
            (λ (n)
              (= Nat n
                     ((+ n) zero)))
            (same zero)
            (λ (n-1)
              (λ (IH)
                (ind-= IH
                  (λ (?)
                    (λ (_)
                      (= Nat (add1 n-1)
                             (add1 ?))))
                  (same (add1 n-1)))))))))
    '()))

'(n)
```

9.
```
(run 1 q
  (pie^o
    `((claim/define +
        (Π ([n Nat])
          (Π ([m Nat])
            Nat))
        (λ (n)
          (λ (m)
            (ind-Nat n
              (λ (z) Nat)
              m
              (λ (n-1)
                (λ (res)
                  (add1 res)))))))
      (claim/define +-zero-r
        (Π ([n Nat])
          (= Nat n ((+ n) zero)))
        (λ (n)
          (ind-Nat n
            (λ (n)
              (= Nat n ((+ n) zero)))
            (same zero)
            (λ (n-1)
              (λ (IH)
                (ind-= IH
                  (λ (?)
                    (λ (_)
                      (= Nat (add1 n-1)
                             (add1 ?))))
                  (same (add1 n-1)))))))))
    '()))

'(_0)
```

Fig. 17. Longer Pie evaluations involving +

B SOLUTION TO THE EXCERCISE AT THE END OF SECTION 2.2

An alternate swap-some^p, using nested cond^{p}s.

First, suggestion functions for the outer and inner cond^{p}s:

```
(define (ls-keys-outer ls)
  (cond
    ((var? ls) '(use-maybe))
    ((null? ls) '(BASE))
    ((pair? ls) '(REC))
    (else '())))

(define (o-keys-outer o)
  (cond
    ((var? o) '(BASE REC))
    ((null? o) '(BASE))
    ((pair? o) '(REC))
    (else '())))
```

```
(define (ls-keys-inner ls)
  (cond
    ((var? ls) '(use-maybe))
    (else '(KEEP SWAP))))

(define (o-keys-inner o)
  (cond
    ((var? o) '(KEEP SWAP))
    ((pair? o)
     (if (or (var? (car o))
             (eqv? 'novel (car o)))
         '(KEEP SWAP)
         '(KEEP)))
    (else '())))
```

Then, the final definition of swap-some^p with nested cond^{p}s, using the above suggestion functions:

```
(defrel (swap-someᵖ ls o)
  (condᵖ
      (((ls-keys-outer ls))
       ((o-keys-outer o)))
    (BASE (== '() ls) (== '() o))
    (REC (fresh (a d res)
           (== `(,a . ,d) ls)
           (condᵖ
               (((ls-keys-inner ls))
                ((o-keys-inner o)))
             (KEEP (== `(,a . ,res) o))
             (SWAP (== `(novel . ,res) o)))
           (swap-someᵖ d res)))))
```