Guarded Fresh Goals: Dependency-Directed Introduction of Fresh Logic Variables

EVAN DONAHUE, University of Tokyo, Japan

We extend miniKanren with a new fresh form that combines logic variable instantiation with unification. We show how combining these operations exposes information about the dependencies between logic variables that permits a variety of optimizations and heuristics. These optimizations include minimizing the introduction of new logic variables, eagerly exploiting ground terms, and reordering conjuncts according to the interdependencies between relations that share logic variables. We demonstrate improved performance in a collection of relational program synthesis tasks and describe possible avenues for further exploration within this architectural framework.

ACM Reference Format:
Evan Donahue. 2021. Guarded Fresh Goals: Dependency-Directed Introduction of Fresh Logic Variables. In . ACM, New York, NY, USA, Article 5, 15 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Relational program synthesis is a central topic of research in the miniKanren community due in part to the language's strengths in this area [5–7, 9]. At the same time, the combinatorial nature of the task places great demands on the underlying synthesis engine in terms of speed and search efficiency. Much work has been dedicated to exploring various search strategies and other algorithmic enhancements to improve miniKanren's search performance [3, 5, 11–13, 15].

The central contribution of this paper is an extension to miniKanren that enables the application of several heuristics that significantly improve the performance of a relational interpreter on a variety of program synthesis tasks. Several of those heuristics, such as reordering conjuncts more fairly, prioritizing goals with ground arguments, and reusing logic variables to avoid extending the substitution are related to recent work [2, 5, 12]. Others, particularly interleaving parallel relations are, to our knowledge, novel in this work.

The work described in this paper was conducted in the SmallKanren dialect of miniKanren implemented in Pharo Smalltalk. SmallKanren is a first-order dialect that supports an extensible constraint system [1], a debugger [14], and tabled relations [4]. The code examples in this paper were transliterated from Smalltalk code into a Scheme-like pseudocode for greater readability by the wider miniKanren community.

2 RELATIONAL PROGRAM SYNTHESIS

Program synthesis, in a miniKanren context, involves specifying program behavior in terms of a set of input/output values and allowing the synthesis system to search for a program that produces each output given its respective input.

Consider the following pseudocode for a program synthesis relation, which will be referenced throughout the remainder of the paper:

miniKanren 2021, August 26 2021, Online

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

```
(define synthesizeo (program input-1 output-1 input-2 output-2 ...)
  (fresh ()
    (evalo program input-1 output-1)
    (evalo program input-2 output-2)
    ...))
```

Listing 1. Pseudocode for synthesizeo

Following prior work on relational program synthesis in miniKanren, *evalo* is a relational implementation of the Scheme **eval** function, which accepts an expression and an environment and produces the value corresponding to the expression evaluated in the context of the supplied environment. Due to miniKanren's ability to "run backwards," not only can the *evalo* relation evaluate the same range of expressions as the non-relational **eval** function, it can also, when supplied with output values, synthesize all possible expressions that evaluate to those values in a given environment.

By sharing the *program* logic variable among all calls to *evalo*, the *synthesizeo* relation constrains program to satisfy each pair of input/output values. Inputs are supplied as environments with variables pre-bound to the input values, while outputs are passed as return values to the *evalo* relations.

A simple implementation of *evalo* might look as follows:

```
(define evalo (expression environment out)
  (conde (quoteo expression out)
         (lookupo expression environment out)
         ...
         (conso expression environment out)))
```

3 GUARDED FRESH GOALS

3.1 Motivation

Any non-trivial miniKanren program will contain numerous instances of the **fresh** form for instantiating fresh logic variables. In its simplest implementation, however, **fresh** makes no use of several pieces of information that the implementation can use as heuristics to significantly speed up program synthesis.[1]

Consider the common miniKanren idiom:

```
(define some-relation (q)
  (fresh (a b)
    (== q (cons a b))
    ...))
```

If q is a ground pair, a and b will be bound to its contents, allowing destructuring of complex input terms. If q is a free logic variable, it will be bound to a newly created pair and a and b will in turn correspond to the subterms of the new pair. The duality of interpretations of this idiom depending on whether q is free or bound is part of what gives miniKanren its ability to "run backwards."

[1]The implementation details described in this paper are, of necessity, partly a consequence of the affordances of the Smalltalk language. Lacking macros, SmallKanren requires the user to wrap all potentially recursive calls in explicit, user-level **fresh** goals to avoid infinite recursion, as is done in [8]. As a result, managing **fresh** goals implicitly manages all potentially recursive goals, which is why this paper focuses its discussion on the **fresh** form in particular. Other miniKanren implementations with other architectures may need to manage other forms as well, such as relation definition, although such treatment is likely to be analogous and require little additional effort.

However, the standard implementation of **fresh** as defined in [4] does not make use of several pieces of information useful for improving the performance of the overall search.

3.1.1 Reusing Logic Variables. First, considering the case in which q is ground, every relation that accesses subterms of q must do so by creating new free logic variables and binding them to those subterms. Repeated variable binding will lead to an ever expanding substitution, increasing the cost of the *walk* operation in all future unifications. Consider the above relation evaluated in the following substitution:

```
((q . (cons c 1)))
```

Assuming c is a logic variable, under the normal idiom, the substitution would become:

```
((b . 1) (c . a) (q . (cons c 1)))
```

Suppose, however, that it was possible to transform the **fresh** goal into something of the following form (slightly abusing the notation of **fresh** to allow for constants in the variable declaration form):

```
(define some-relation (q)
  (fresh (c 1)
    ...))
```

The logic variable and even the ground term are simply reused directly by subgoals of the **fresh** and the unification and corresponding extensions to the substitution would be avoided.

3.1.2 Prioritizing Relations with Ground Arguments. The groundedness of q can be an important heuristic in determining when to execute the subgoals of the **fresh** goal. If q is bound, then its value is necessarily constrained relative to the case in which it is free and the **fresh** goal may therefore fail sooner than if q was free. As such, given the choice between expanding the subgoal of a **fresh** that destructures an existing ground term and one that creates a new binding and searches within an unconstrained space, it may in cases be preferable to prioritize the former over the latter.

Consider the synthesis problem starting from the following partial specification of the *map* function in which underscores represent "holes" in the program for the synthesis engine to fill:

```
(define map (f xs)
  (if (null? xs) '()
      (cons (f (car xs)) (map _ _))))
```

If the interpreter begins by trying to synthesize the arguments to the recursive call directly, it will find that they are completely unconstrained, leading to a very large search space. If, however, the interpreter begins by starting to evaluate the already ground body of the recursive step, waits until it encounters the variable lookup expressions f or xs in the body, and then attempts to synthesize the arguments to the recursive call using the values that must be returned by the variable lookups, the search will be considerably more constrained.

Byrd et al. [5] reports significant performance gains from manually controlling this execution order in the interpreter, however it is possible to arrive at a similar execution order automatically by prioritizing conjuncts with ground inputs over those whose inputs are completely free, as this will prioritize execution of the body, then extension of the environment, and finally the evaluation of the operands with which that environment is extended. This heuristic and its application are discussed in more detail in 6.

3.1.3 Interleaving Interdependent Goals. Using normal left-to-right conjunction, the first call to *evalo* must complete before subsequent *evalo* calls can be evaluated. However, because the only information available to constrain the search is present in the input/output pairs, the first call to *evalo* must operate entirely without the benefit of subsequent input/output pairs. It may therefore

be desirable to interleave the subgoals of these two calls to *evalo* such that subexpressions of the final *program* term selected within the first *evalo* call can immediately be tested against all available input/output information from subsequent calls.

Consider the following pair of input/output examples for synthesizing an *nth* function that returns either the nth element in a list or #f if the index is out of bounds:

```
(synthesizeo
 program
 '(1 ((a) (b))) '(a)
 '(1 ()) #f)
```

Because the first output value is a list, *synthesizeo* may initially be tempted to attempt to synthesize a program beginning with **cons**. Such an attempt will succeed an unbounded number of times while considering the first input/output pair, albeit not in a manner that generalizes to the next pair. No program that begins with **cons** can ever synthesize the `\#f` value in the following input/output pair.

A "fair" search that considered both input/output pairs in tandem as it proceeded through the search would be able to prune this infinite branch and others like it immediately. Crucially, however, this fairness is not fairness at the level of individual disjunctions or conjunctions, but rather at a higher semantic level. It is a fairness between calls to *evalo* itself and between their corresponding subgoals.

Writing a "fair" interleaving *evalo* would be considerably more complex than simply translating an existing functional **eval** implementation into relational terms and calling it repeatedly as is done in *syntehsizeo*. However, if the system recognizes, when it first attempts to destructure *program* into its subexpressions, that *program* is a free variable, it has the opportunity to delay evaluation until more is known about the constraints on *program*. Even if *program* itself is never directly bound or otherwise constrained, as would be the case in the example above, it may be useful to make all assumptions that will be made about *program* at the same time. By executing, for instance, the subgoals from each of the *evalo* calls that attempt to bind the first subexpression of *program* to **cons**, the search can check that each of these calls to *evalo* is expecting a pair before attempting to synthesize any further subexpressions. Such delaying and promoting, guided by the dependencies on shared logic variables and depending on the heuristics used, automatically recovers the interleaved version of *evalo* described above without sacrificing the interpretability of the *evalo* relation. This problem of needing to interleave relations that are easier to write separately arises in other areas as well, and so may have wider applicability than just to program synthesis [7].

3.2 Guarded Fresh Goal Syntax

To make efficient use of the information mentioned in the previous section, we define a new type of **fresh** form that implicitly captures this information. Using **guarded-fresh**, the above idiomatic use of **fresh** would have the following syntax:

```
(define some-relation (q)
  (guarded-fresh ((q (A . B))
                  ...)))
```

This form specifies that q should be unified with the pair of newly instantiated logic variables A and B and then any subgoals should be executed with direct access to the new logic variables. For clarity when discussing the implementation, the following "desugared" notation will be used for the remainder of this paper:

```
(define some-relation (q)
```

```
(guarded-fresh ((q '(A . B))
                (lambda (a b)
                  ...)))
```

Listing 2. Minimal working example of guarded-fresh syntax

The **guarded-fresh** goal consists of two parts. The first argument is a binding form similar to that of **let**. Logic variables or ground terms in the local environment are bound to patterns, which are interpreted by the form. The syntax of these patterns is analogous in this instance to the generic Scheme reader with the exception that symbols starting with capital letters are interpreted as fresh logic variables. The binding `(q '(A . B))` indicates that q is to be bound in the current substitution to a pair containing two fresh logic variables, A and B. The second argument to **guarded-fresh** is a **lambda** expression that accepts as many arguments as there are logic variables in all of the binding expressions combined.

A naive implementation of this form would simply unify q with the pair `(A . B)` and then invoke the **lambda** with arguments A and B. However, such an implementation makes no use of the dependency information that is made explicit by this syntax. Because subgoals resulting from evaluation of the **lambda** term containing A or B depend on the contents of q, this form exposes information needed to decide when to execute those subgoals based on the groundedness of q or on the timing of the execution of other subgoals that also depend on q.[2]

4 IMPLEMENTATION

In this section, we describe the high-level intuition behind the implementation of **guarded-fresh**. We defer a more detailed discussion of the implementation details to A.

4.1 The Schedule

The SmallKanren implementation of **guarded-fresh** extends the central miniKanren package datastructure, which contains the substitution and constraint store, with an additional data structure we will refer to as the "schedule:"

`(substitution constraint-store schedule)`

SmallKanren implements the schedule as a simple queue, although see 6 for remarks on alternative implementations.

4.2 Adding to the Schedule

When **guarded-fresh** goals are first encountered, their binding forms are immediately unified in the current substitution and a tuple containing dependency information and the **lambda** expression are added to the schedule. Returning to the minimal working example 2, q is immediately unified with `(A . B)`, and the tuple `((q) (A B) (lambda (a b) . . . ))` is added to the schedule.

4.2.1 External Variables. The list `(q)` represents the "external" variables that correspond to the left-hand-sides of the **guarded-fresh** binding form. They are called "external" variables because they represent variables that already exist in the environment outside the **guarded-fresh**. External variables are used to determine when the **guarded-fresh** depends on ground terms or bound variables, which guides the rearrangement of the schedule to make best use of ground information.

[2]Lozov and Boulytchev [12] describes the use of a similar type of dependency information collected through static program analysis at the relation level. It is possible that a similar type of static analysis could afford the same benefits described in this paper to a program written using the conventional miniKanren idioms, depending on the affordances of the host language.

If, for instance, q is already bound, this **guarded-fresh** will have no external variables and its external variable list will be empty.[3] The significance of an empty list will be explained in 4.3.

4.2.2 Internal Variables. The list $(A\ B)$ represents the "internal" variables that correspond to the newly instantiated logic variables generated from the capitalized symbols in the interpreted binding patterns. These internal variables will be passed as arguments into the **lambda** term when the tuple is removed from the schedule. If q is bound prior to this removal, including by other **guarded-fresh** goals that also depend on q, these internal variables will be used along with the unifier to extract the ground terms from the binding of q to which they correspond, and these ground terms will be passed to the **lambda** term instead. This latter procedure is discussed in more detail in A.

4.2.3 Lambda Term. The **lambda** term in the tuple represents the continuation that will return the subgoal of the **guarded-fresh** form when supplied with the internal variables on which that subgoal depends.

4.3 Removing from the Schedule

Because **fresh** goals in SmallKanren contain all recursive calls, each step in the search will return a package with unifications added to the substitution, constraints added to the constraint store, and potentially recursive **fresh** goals added to the schedule.[4] If the schedule is empty, then the package can be returned as an answer, as there exist no more conjuncts to constrain it. However, so long as the schedule is non-empty, there remain additional conjuncts and the stream represented by the package remains incomplete.

In order to expand this incomplete stream, it is necessary to select one tuple from the schedule, evaluate its **lambda** term by supplying the corresponding internal variables, and then evaluate the returned goal in the context of the current package. If the schedule is a queue, the simplest strategy is simply to pick the first tuple, with one caveat: when a tuple is removed from the queue, each of its external variables must be removed from the list of external variables for each tuple remaining in the schedule.

Tuples with empty external variable lists have no dependencies on which they are waiting, and so should be prioritized above even the first tuple in the schedule. This mechanism is what creates the higher-level fair interleaving discussed in 3.1.3. Even if q has not been bound by program inputs, all tuples that depend only on q or already bound external variables will fire in succession, independent of the order in which they were added to the schedule. Correspondingly, subgoals of *evalo* that depend on the same subexpression will likewise fire in succession, bringing each subgoal's ground data to bear on each step of the synthesis.

5 EVALUATION

This section compares the performance of a relational interpreter implemented using **guarded-fresh** goals to one implemented using standard left-to-right conjunct evaluation. SmallKanren, and by extension any relational interpreter written in SmallKanren, is implemented using Smalltalk as the host language. However, the interpreters used in these experiments interpret a restricted version of Scheme.

[3]Note, however, that although the **guarded-fresh** binds q immediately, a subsequent **guarded-fresh** that also depends on q should not therefore mistake q for a ground term supplied externally to the program. Differentiating between variables bound by other **guarded-fresh** goals and those bound to primary inputs to the program is an important subtlety which will be further discussed in A

[4]The link between **fresh** goals and recursive relations in SmallKanren is an artifact of the affordances of the host language and arises from a lack of macros as in [8]. Other host languages may differ, and will have to adjust this description accordingly.

The functions chosen for synthesis belong to three classes: quines, logical functions (*and* and *xor*), and recursive functions of lists (*zip*, *map*, *append*). Quines were chosen for comparison with past work, and due to their unusual quine property as it pertains to relational program synthesis. The logical functions were chosen because they are relatively small functions that nevertheless require a larger number of input/output examples to fully specify, which it was assumed would benefit the guarded interpreter's ability to make early use of ground information in the input/output pairs. The recursive functions were chosen because the ability to exploit ground information in the form of the partial program during the recursive step was likewise thought to benefit the guarded interpreter. We were not able to conceive of any a priori classes of functions for which we believed the guarded clauses would be a liability, however the quines case proved interesting in this regard.

The interpreters perform limited canonicalization to rule out some expressions that, while valid Scheme, are generally unhelpful in synthesizing recursive functions. For instance, **`car`** and **`cdr`** are prevented from synthesizing a cons as a direct child.[5] Consequently, the results here are not comparable to similar synthesis experiments in other host languages, and should be taken only as a comparative benchmark between two program synthesis tasks in the same host language using the same interpreter architecture.

All experiments were run on a Lenovo ThinkPad T520 laptop with a 2.4GHz Intel Core i7-2760QM CPU running Ubuntu 18.04. To measure performance, we used the Pharo *bench* command, which runs the synthesis as many times as possible in one second with a minimum of one execution and averages the results. As such, the longer times reported were the result of a single run. However, given the magnitudes involved and the variances observed during testing, we do not believe additional runs would substantially change our conclusions.

5.1 Smallest Partial Function Synthesized

In this section, we report the smallest partial functions required for both the regular interpreter and the guarded interpreter to synthesize a complete and correct implementation. For each function name, we report the smallest partial program required for each interpreter using two notational conventions: underscores (_) represent free logic variables and digits (1, 2, etc.) represent variable lookups of the first, second, etc. argument to the synthesized function. 5 minutes was the cutoff used to terminate searches early, as allowing the interpreters to run much beyond that occasionally caused the Smalltalk image to lock up. Blank table entries indicate that the entire program was synthesized with no partial program hints. Function names are in bold if the guarded interpreter resulted in improved performance on this program completion metric. Time-based metrics are reported in the next section.

Function	Left-to-Right Fresh	Guarded Fresh
quine		
and		
xor		
zip	`(if (null? 2) '() (cons (cons (car 1) (car 2)) (_ _ _)))`	`(if _ '())`
map	`(if (null? 2) 2 (cons (1 (car 2)) (_ _ _)))`	`(if (null? 2) 2 (_ (_ _) _))`
append	`(if (null? 1) 2 (cons (car 1) (_ _ _)))`	`(if _ 2 _)`

[5] This canonicalization is accomplished with several specialized *evalo* relations that evaluate only a subset of all possible forms.

Note that the recursive step proved particularly difficult for the left-to-right synthesis system, and it was only ever able to synthesize a recursive step if prompted to synthesize an **`apply`**.[6]

5.2 Timed Comparison Using Identical Partial Programs

Times in this section were reported using the greatest partial program required by either interpreter to synthesize the function in question. Orders of magnitude improvements are reported for each function, with positive orders with boldface function names indicating improved performance with the guarded interpreter and negative orders with a plain font face indicating reduced performance.

Function	Left-to-Right Fresh	Guarded Fresh	Orders of Magnitude Improvement
quine	.84 seconds	11 seconds	-1
and	.3 seconds	.16 seconds	0
xor	54 seconds	.4 seconds	2
zip	1:35	.58 seconds	2
map	2:47	8.8 seconds	1
append	4:47	2.5 seconds	2

Predictably, the guarded interpreter, which was able to synthesize complete programs using smaller partial programs was faster when measured on the same partial recursive programs. Surprisingly, the non-recursive *xor* program was also significantly faster in the guarded interpreter, and even more surprisingly, quine performance degraded significantly when using guarded goals.

6 DISCUSSION

As demonstrated in 5, guarded clauses frequently offer a significant performance increase over a static left-to-right conjunct order. However, measuring performance in miniKanren is complex due to the fact that even small, seemingly innocuous changes can alter the search behavior enough to result in performance increases or decreases of several orders of magnitude. One surprising example of this is that adding multiple no-op fresh goals to *applyo* and *ifo* significantly improved the performance of both interpreters as in the following[7]:

```
(define ifo (expr env out)
  (fresh ()
    (fresh ()
      ...)))
```

Beyond quines, the dramatic differences in performance between the left-to-right interpreter and the guarded interpreter on synthesizing the recursive steps of programs were likely due in significant part to the eager evaluation of ground terms for synthesizing recursive programs. By the time the recursive step is to be synthesized, most of the function has already been synthesized, and so can be exploited eagerly by the **`guarded-fresh`** goals evaluating the recursive steps. The fact

[6] Although it is not apparent in this syntax, the interpreters used an explicit **`apply`** function that allowed the partial program to specify an explicit function application as opposed to other function-like keywords.

[7] The left-to-right interpreter used classic fresh goals, whereas the guarded interpreter used "fair" fresh goals handled by the schedule like **`guarded-fresh`** goals but without the dependency tracking behavior. Performance increases may have been due to delaying the expensive *ifo* and *applyo* relations long enough to find an earlier answer. The fair nature of the queue-based schedule may have helped to delay those relations for longer. A third interpreter composed entirely of these "fair" **`fresh`** goals was also tested, but performed universally worse than either of the others, possibly because these goals did not unify their terms immediately, as did **`guarded-fresh`** goals, nor did they make great use of the goal ordering done by the programmer, spending a great deal of time exploring unconstrained parts of the search space, such as by expanding the environment with fresh logic variables.

that the left-to-right interpreter was unable to synthesize a single recursive step without significant prompting supports this theory, and it agrees with the analysis of [5].

However, the significant performance difference on *xor*, which is not recursive, suggests that eager use of ground information is not the only dimension of improvement. We suspect that the interleaving of the *evalo* relations to more fairly apply the information contained in the input/output pairs may be more responsible for this improvement. However, additional test functions and a more fine-grained analysis of the individual optimizations would help clarify their relative contributions.

7 RELATED WORK

The strategy of using guard clauses to capture dependency information has a long history in logic programming, with applications to improving search performance [10] and enabling the coordination of parallel processors [16, 17]. Although this is to our knowledge the first attempt to bring this syntactic strategy to miniKanren, our objectives in doing so intersect with several existing lines of work.

There has been much work within the miniKanren community on improving search performance for various types of problems [13, 15]. This work belongs to that broad category as well, but it is worth comparing it more specifically to several lines of methodologically similar work on conjunct reordering and thematically similar work on improving the performance of relation program synthesis systems.

Lozov and Boulytchev [11, 12] describe an approach to conjunct reordering similar in some respects to the reordering achieved by **`guarded-fresh`** goals, although with a different focus. The authors describe a "naive fair conjunction" that behaves like **`guarded-fresh`** without the dependency information derived from the external variables. This fair conjunction avoids divergence by default and serves as the basis for further heuristic optimizations. In particular, the authors perform static analysis at the relation level and track runtime information to identify structurally recursive relations that are safe to evaluate eagerly for significant performance gains on some types of search problems. **`guarded-fresh`** as currently implemented does not identify such relations, but performs other forms of reordering, such as interleaving subrelations of *evalo* not discussed in [12]. Consequently, there may be room to improve **`guarded-fresh`** by incorporating the recursion tracking described in that work.

Interestingly, the choice of a queue as the default schedule datastructure yields behavior similar to this fair conjunction in that the queue also guarantees convergence and for the same reasons. Every goal added to the schedule, ignoring reordering due to dependency heuristics, will eventually be run, which is the key condition that avoids divergence. If the schedule was a list, it would be possible to add and remove goals indefinitely without ever reaching a goal deeper in the list, potentially diverging as a result. The effect of the dependency reordering heuristics on this divergence avoiding property—both those currently implemented and those enabled by the general guarded architecture—remains a topic of ongoing research.

Ballantyne [2] describes a "set-var-val" optimization that dynamically detects unifications of fresh variables performed before non-deterministic branching and mutates the internal variable state to avoid unnecessary walking of the substitution. This optimization performs a similar function to the variable reuse optimization of **`guarded-fresh`**. **`guarded-fresh`** does not have the constraint that its substitution-avoiding properties depend on the positioning of branching **`conde`** forms, but it also handles the variables covered by the set-var-val optimization less efficiently than that optimization due to the need for an initial walk. It is therefore likely the two could be productively combined.

Byrd et al. [5] describes several optimizations to the interpreter architecture itself. First, it describes a **`conde`**1 form equivalent to the groundedness-checking behavior of **`guarded-fresh`** in that it executes its subgoals immediately if all arguments are ground, and suspends them otherwise.

Supplying **guarded-fresh** with patterns consisting of just single variables as in the following has a similar effect, albeit with the addition of the dependency-tracking machinery:

```
(guarded-fresh (q Q)
  ...)
```

Byrd et al. [5] also describes specific optimizations to the *applyo* relation that involve checking for groundedness across several conjoined relations, including those that handle the operator and those that handle the operands of an **apply** expression, prioritizing whichever has ground terms. **guarded-fresh** performs a similar although not identical type of groundedness analysis automatically, although that analysis is subject to the heuristics built into the **guarded-fresh** implementation, which is still a subject of active research.

Interestingly, we were unable to improve performance with our **conde1** implementation, suggesting that the benefits of **conde1** may already be entailed in the implementation of **guarded-fresh**. We were also unable to improve performance by designing heuristics to make **guarded-fresh** mimic the groundedness seeking behavior of the manual **apply** optimizations. However, because [5] attests to their efficacy there may be room to further improve **guarded-fresh** using a better understanding of the logic captured by those manual optimizations.

8 CONCLUSION

We described a new fresh form that captures grounding and dependency information and an architecture for exploiting that information to improve the performance of the miniKanren search in a program synthesis context. We demonstrated significant performance improvements on a variety of program synthesis tasks and described several additional avenues of potential further research on dynamically reordering conjuncts using grounding and dependency information to improve the performance of relational program synthesis systems.

In future work, we anticipate expanding our focus to other classes of problems, including exploring the possibility of interleaving conjoined goals in another common miniKanren program architecture involving the parallel invocation of two different relations with mutually entangled dependencies.

Chirkov et al. [7] discusses the case of a Javascript synthesis system composed of a parser and an interpreter that mutually constrain one another. The authors suggest that writing the parser and the interpreter as separate relations is easier and more maintainable, while interleaving the relations might make synthesis faster. **guarded-fresh** goals, possibly supplemented by additional heuristics, might allow the relations to be written separately and subsequently automatically interleaved by the implementation in the same manner as *evalo*. Likewise, semantic parsers for natural language may also benefit as they exhibit the same heterogeneous, bipartite structure.

9 ACKNOWLEDGMENTS

We thank Jason Hemann, Michael Ballantyne, Petr Lozov, and Nada Amin for their advice, explanations, encouragement, and discussion during the planning, writing, and submission phases of this piece. We especially thank Will Byrd for his extensive feedback and comments. We also thank the anonymous reviewers for their detailed suggestions.

REFERENCES

[1] Claire E Alvis, Jeremiah J Willcock, Kyle M Carter, William E Byrd, and Daniel P Friedman. 2011. cKanren miniKanren with constraints. (2011).

[2] Michael Ballantyne. 2020. Faster miniKanren [Source Code]. (2020). https://github.com/michaelballantyne/faster-miniKanren

[3] David C Bender, Lindsey Kuper, William E Byrd, and Daniel P Friedman. 2009. Efficient representations for triangular substitutions: A comparison in miniKanren. *Unpublished manuscript* (2009).
[4] William Byrd. 2010. *Relational Programming in Minikanren: Techniques, Applications, and Implementations.* Ph.D. Dissertation. Indiana University.
[5] William E Byrd, Michael Ballantyne, Gregory Rosenblatt, and Matthew Might. 2017. A unified approach to solving seven programming problems (functional pearl). *Proceedings of the ACM on Programming Languages* 1, ICFP (2017), 1–26.
[6] 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*. ACM, 8–29.
[7] Artem Chirkov, Gregory Rosenblatt, Matthew Might, and Lisa Zhang. 2020. A Relational Interpreter for Synthesizing JavaScript. In *Proceedings of the miniKanren and Relational Programming Workshop.*
[8] Jason Hemann Daniel P Friedman. 2013. μKanren: A Minimal Functional Core for Relational Programming. In *Proceedings of the 2013 Workshop on Scheme and Functional Programming.* http://webyrd.net/scheme-2013/papers/HemannMuKanren2013.pdf
[9] Jason Hemann and Daniel P Friedman. 2020. Some Novel miniKanren Synthesis Tasks. In *Proceedings of the miniKanren and Relational Programming Workshop.*
[10] Sverker Janson and Seif Haridi. 1991. Programming paradigms of the Andorra kernel language. *SICS Research Report* (1991).
[11] Petr Lozov and Dmitry Boulytchev. 2020. On Fair Relational Conjunction. (2020).
[12] Peter Lozov and Dmitry Boulytchev. 2021. Efficient fair conjunction for structurally-recursive relations. In *Proceedings of the 2021 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation.* 58–73.
[13] Kuang-Chen Lu, Weixi Ma, and Daniel P Friedman. 2019. Towards a miniKanren with fair search strategies. In *Proceedings of the 2019 miniKanren and Relational Programming Workshop.* 1–15.
[14] Gregory Rosenblatt, Lisa Zhang, William E Byrd, and Matthew Might. 2019. First-order miniKanren representation: Great for tooling and search. In *Proceedings of the miniKanren and Relational Programming Workshop.* 16.
[15] Cameron Swords and Daniel Friedman. 2013. rKanren: Guided search in miniKanren. In *Proceedings of Scheme Workshop.*
[16] Kazunori Ueda. 1985. Guarded horn clauses. In *Conference on Logic Programming*. Springer, 168–179.
[17] Kazunori Ueda and Masao Morita. 1990. A new implementation technique for flat GHC. In *Logic programming*. 3–17.

A IMPLEMENTATION OF GUARDED FRESH

An example will be useful for illustrating the implementation of the **`guarded-fresh`** goals. Consider the implementation of the *conso* relation in a relational interpreter:

```
(define conso (expr env out)
  (guarded-fresh ((expr '(cons CarExpr CdrExpr))
                  (out '(CarVal . CdrVal)))
                 (lambda (car-expr cdr-expr car-val cdr-val)
                   (conj (evalo car-expr env car-val)
                         (evalo cdr-expr env cdr-val)))))
```

Listing 3. Implementation of conso subrelation of relational interpreter

Given a synthesis problem with two input/output pairs, the top level *evalo* will be called two times, one for each pair, as illustrated in 1. In the branch of the search tree in which the interpreter attempts to synthesize a **`cons`** expression, *conso* will be evaluated two times, again one for each input/output pair. The first call to the guarded fresh goal in the *conso* relation will essentially unify *expr* with a triple and *out* with a pair, each containing the appropriately instantiated logic variables. It will then push a 3-tuple of the external variables from the left hand sides of the bindings (*expr* and *out*), the internal variables from the right hand patterns (*CarExpr*, *CdrExpr*, *CarVal*, and *CdrVal*), and the closure representing the **`lambda`** term into the schedule, yielding a tuple of the form:

```
((expr)
 (CarExpr CdrExpr CarVal CdrVal)
```

```
(lambda (car-expr cdr-expr car-val cdr-val)
  (conj (evalo car-expr env car-val)
        (evalo cdr-expr env cdr-val))))
```

Note that if *out* is ground, it will not appear in the external variables list. Tuples with empty lists of external variables represent **guarded-fresh** goals in which all terms are ground. Such tuples may be executed immediately, ahead of other tuples in the schedule. When any tuple is removed from the schedule for any reason, its closure will be applied to the list of internal variables and the resulting goal will be evaluated in the current substitution.

With only one input/output pair, the resulting execution order will be similar to that of a naive implementation that makes no use of the dependencies between multiple calls to *evalo* that all depend on the same *expr* variable. Most of the interesting behavior only becomes apparent with multiple parallel calls to *evalo*.

A.1 Adding to the Schedule

Consider the following implementation of the **guarded-fresh** procedure that adds new goal tuples to the schedule:

```
1  (define guarded-fresh (vars+patterns body)
2    (lambda (package)
3      (let ((lhs-vars (map car vars+patterns))
4            (rhs-patterns (map hydrate-pattern (map cdr vars+patterns)))
5            (bindings (map (lambda (b) (walk-binding b package)) lhs-vars))
6            (walked (map cdr bindings))
7            (substitution (unify walked rhs-patterns (new-substitution)))
8            (next-package (unify walked
9                                 (walk-recursive walked substitution)))
10           (internal-vars (map (lambda (v) (walk v substitution))
11                               (extract-internal-vars rhs-patterns)))
12           (external-vars
13            (map car (filter (lambda (b)
14                               (or (var? (cdr b))
15                                   (in-schedule? (car b))))) bindings))))
16       (if (null? external-vars)
17           ((apply body internal-vars) next-package)
18           (extend-schedule
19            `(,external-vars ,internal-vars ,body) next-package)))))
```

Listing 4. Implementation of primary functionality for guarded-fresh form

This procedure performs the first two optimizations described in the introduction: it avoids adding unnecessary fresh variables to the substitution, preferring instead to reuse the logic variables already present, and it consumes ground terms eagerly if all external variables bound in a **guarded-fresh** form are ground.

The first part of this procedure up to line 12 is primarily concerned with unifying ground information from the pattern in the package substitution. The second part, from line 12 to the end, either evaluates the subgoals immediately if all terms are ground, or else adds the goal to the schedule.

Because the same *expr* logic variable is passed to each call to *conso*, it creates a dependency between the two **guarded-fresh** goals in each call by virtue of their shared external variable. The first call to *conso* unifies *expr* with the **cons** expression and pushes its tuple onto the schedule. The

following line-by-line description of the code therefore considers in detail the execution of the second call, assuming the first has already pushed its tuple into the schedule, as this second call best illustrates the important subtleties.

```
(let ((lhs-vars (map car vars+patterns))
      (rhs-patterns (map hydrate-pattern (map cdr vars+patterns)))
```

Using the example of the *conso* relation, lines 3-4 extract the variables from the left hand side of the **guarded-fresh** binding forms and the patterns from the right hand side, substituting capitalized symbol names with logic variables.

```
(bindings (map (lambda (b) (walk-binding b package)) lhs-vars))
```

Line 5 invokes a specialized form of *walk*. *walk-binding* returns not simply the value to which a variable is bound, but a pair containing the variable-value association. In the case of free variables or ground terms, the pair contains two copies of the walked value.

The first call to *conso* would have found *expr* free and *out* ground, the walked bindings would have returned (*expr* . *expr*) and (*out* . *out*) respectively. However, as the first call has now bound *expr*, the second call will walk the binding to find (*expr* . (**cons** *CarExpr CdrExpr*)). The value of retrieving *expr* itself from the substitution, as opposed to only its bound value, is that it will be needed later to determine whether the earlier call to *conso* is waiting on *expr*, which will differentiate this binding created by a previous **guarded-fresh** goal from a variable bound by program inputs.

```
(walked (map cdr bindings))
```

Line 6 extracts the cdrs of the walked pairs, yielding a list of the values that would have been returned by the standard *walk* procedure.

```
(substitution (unify walked rhs-patterns (new-substitution)))
(next-package (unify walked (walk-recursive walked substitution)))
```

Lines 7-8 extract any ground information from the pattern that needs to be added to the substitution, such as the **cons** symbol, and adds that information into the substitution to produce the next substitution. This is accomplished through a reuse of the standard unifier.

Line 7 unifies the walked external variables, in this case *expr* and *out*, with their corresponding patterns in an empty substitution. This temporary substitution will contain binding information that describes the relationship between the patterns in the guard forms and the external variables bound to those patterns, and will be used to determine which information from the patterns should be bound in the primary substitution and conversely which information from the walked variables should be reused in the subgoals of the **guarded-fresh**. The creation of this substitution however depends crucially on the order in which the patterns and the external variables are unified, and this order is implementation dependent.

The terms must be unified so that if two variables are bound in this new substitution, walking either one yields the variable contained in the external variable terms and not the pattern. In other words, the ground term bound to *expr* by the first call to *conso*, namely (**cons** *CarExpr*1 *CdrExpr*1), will be unified with the new pattern (**cons** *CarExpr*2 *CdrExpr*2), resulting in a substitution of the form[8]:

```
((CarExpr2 . CarExpr1) (CdrExpr2 . CdrExpr1))
```

[8] 1 and 2 function here as indicators of which call to *conso* instantiated the given copy of *CarExpr* or *CdrExpr*, and are not part of the variable name.

This substitution can subsequently be used in two directions. The first direction allows us to avoid unnecessarily unifying free variables in the primary substitution.

In line 8, we extract any ground terms present in the pattern that need to be bound in the primary substitution. This is accomplished by unifying the walked term (**cons** *CarExpr2 CdrExpr2*) with itself walked recursively in the new substitution, and then unifying in the primary substitution the walked variables with their own corresponding recursively walked values from the temporary substitution. If the temporary substitution contains no new bindings (or only bindings between variables), then the walked variables will each walk to themselves, and so their unifications in the primary substitution will short circuit due to both values and variables being *eq*.

In this case, since *CarExpr2* and *CdrExpr2* walk to *CarExpr1* and *CdrExpr1*, the *next-package* term reduces to the no-op unification (**==** (**cons** *CarExpr1 CdrExpr1*) (**cons** *CarExpr1 CdrExpr1*)), which unifies without walking either logic variable (as they are *eq?*).

```
(internal-vars (map (lambda (v) (walk v substitution))
                    (extract-internal-vars rhs-patterns)))
```

The second direction allows us to reuse the variables bound in the primary substitution in the subgoals of this expression. On line 10, we extract the internal variables *CarExpr2* and *CdrExpr2* from the current pattern and walk them in the substitution to yield *CarExpr1* and *CdrExpr1*, with which they would have unified had we naively unified the entire pattern in the primary substitution. Since *CarExpr2* and *CdrExpr2* are still unbound and not present in any relations, having been only recently instantiated, we can simply use *CarExpr1* and *CdrExpr1* as our internal variables in their place and save the cost of unifying these free variables in the primary substitution. The *internal-vars* therefore simply become *CarExpr1* and *CdrExpr1* for all future uses, although note that had either one been ground in the primary substitution, then one of our internal "variables," which we eventually pass into the lambda expression, would now be a ground term that would later be passed into the closure.

```
(external-vars
 (map car (select (lambda (b)
                    (or (var? (cdr b))
                        (in-schedule? (car b))))) bindings))))
```

Beginning on line 12, we move from the first optimization that avoids unnecessary unifications to the second, which determines when to eagerly evaluate goals based on the groundedness of the input terms. The list of external variables is computed using the list of bindings and the distinction between terms bound by guard clauses and terms bound previously that we discussed above. The list of external variables is extracted from the cars of the bindings list, subject to one of two conditions.

If the cdr representing the bound value is a variable (which may be itself), then the variable contains no ground information, and so is an external variable. If the candidate external variable is bound to a ground term, but is also in the external variable list of an existing tuple in the schedule, then it must have been free when the first tuple that depended on it was added to the schedule, otherwise that tuple would have eagerly consumed the ground information. The existence of a tuple depending on an external variable in the schedule implies that its binding in the substitution was created by a **guarded-fresh** goal and not by program input.[9]

[9]One caveat is that if the tuple that created the binding has already left the schedule, the binding will be indistinguishable from a ground term. This would arise if the *evalo* calls were not direct siblings, but were made at different points in the program. This is not necessarily a problem, as the effect would be to cause the later calls to *evalo* to eagerly consume as many bindings as the earlier *evalo* calls had added to the substitution, essentially causing it to hurry and catch up with the earlier calls, after which it would interleave as normal.

```
(if (null? external-vars)
    ((apply body internal-vars) next-package)
    (extend-schedule `(external-vars internal-vars body) next-package))
```

Finally, on line 16, If the *external-vars* list is empty, then all input terms are definitively ground, and the lambda term can be invoked immediately. Otherwise, the triple of external variables, internal variables, and the closure are pushed onto the schedule.

A.2 Removing from the Schedule

Compared to the process for adding fresh goals to the schedule, the process for removing them is comparatively simple. At the same time, this stage is open to many possible heuristics that determine how to select which goal to expand next.

When it becomes necessary to select a new fresh goal from the schedule to expand, goals are chosen in the order dictated by the schedule implementation. By default, SmallKanren uses the naive fair queue-based ordering for guarded goals. The exception to this order is that if the list of external variables has been made empty since it was added to the schedule, the corresponding goal triple will be prioritized over other fresh goals. The primary means by which a list of external variables can be made empty is via a mechanism tied to the removal of tuples from the schedule.

As each guarded tuple is removed from the schedule, all of its external variables are removed from any guarded fresh goal in the schedule that contains them. Significantly, since all *conso* fresh goals depend on the same *expr* external variable representing an expression in the synthesized program, as soon as the first *conso* is evaluated, *expr* is removed from all subsequent *conso* fresh goals in the schedule, meaning they will be prioritized over other goals in the schedule regardless of the schedule order. The SmallKanren implementation would in this case, having removed *expr* from all *conso* goals, subsequently evaluate the other *conso* goal, regardless of its position in the schedule. Concretely, consider the following schedule containing two calls to *conso* interleaved with one call to variable *lookupo*:

```
(((expr) (CarExpr1 CdrExpr1 CarVal1 CdrVal1) (lambda ...))
 ((env) (VarName EnvValue) (lambda ...))
 ((expr) (CarExpr1 CdrExpr1 CarVal1 CdrVal1) (lambda ...)))
```

After the first goal is removed from the schedule, *expr* is removed from all subsequent goals:

```
(((env) (VarName EnvValue) (lambda ...))
 (() (CarExpr1 CdrExpr1 CarVal1 CdrVal1) (lambda ...)))
```

On subsequent expansion steps the goals with the newly empty list of external variables will be selected above the call to *lookupo* even though it is not first in the queue.

Adding to and removing from the schedule constitute the primary implementation details of **guarded-fresh** clauses. The Smalltalk implementation from which the numbers in this paper were reported will be made available through the author's website in connection with this paper.