Accelerating Program Synthesis in miniKanren

Robert Zinkov
University of Oxford
Oxford, UK
zinkov@robots.ox.ac.uk

Michael Ballantyne
Northeastern University
Boston, MA, USA
ballantyne.m@northeastern.edu

Gregory L. Rosenblatt
University of Alabama at Birmingham
Birmingham, AL, USA
gregr@uab.edu

William E. Byrd
University of Alabama at Birmingham
Birmingham, AL, USA
webyrd@uab.edu

While many logic programming systems like miniKanren are highly expressive, they suffer from long and unpredictable running times. The challenge comes from the search algorithm being usually an uninformed search. Through the domain of program synthesis we show that it possible to greatly speedup this search by guiding it using example programs.

1 Introduction

In the miniKanren[1] programming language, we often pose relations in a compostional manner, where more complicated relations are defined in terms of simpler relations. These simple relations include $\equiv$ defining constraints and **cond**^{e} for defining a logical disjunction over clauses. In **cond**^{e} we have to make a decision over the order we explore each of the clauses. If we pick well, we quickly find an answer that satisfies our constraints. If we pick less well, we hope we quickly find a constraint violation and backtrack immediately. If we pick badly, it could take a long while to get an answer at all.

Now, it can feel like there might an optimal way to arrange the clauses but this runs into several problems. Firstly, our instincts of which clause to try first are in many settings not obvious or intuitive. Secondly, what ordering might make sense in the absence of constraints might not make sense in their presence. But most importantly, the best ordering is highly contextual and when dealing with relations which are recursively-defined the best ordering could be different from call to call.

To deal with this issue we take a cue from existing machine learning work[3, 2] and learn a search heuristic to guide the search. More specifically, we learn a probabilistic model to choose which clause to pick given the context the choices exist within.

As an example consider a grammar like the lambda-calculus. Suppose we are trying to generate the operand e_2 of a function application. We can use the context of being the operand in the expression to decide what program we generate. To decide what expressions to generate we might look at a set of example programs and count in the operand position what is most likely a symbol, a function abstraction, or a function application. Then we should bias our search to look more like example programs. Particularly, if we are trying to generate programs that resemble a given set of example programs.

This can be reasonable for small grammars like the lambda calculus, but what should we do if we end up in a context we have never seen before? In that case, we don't want to say an expression shouldn't be generated even if we don't possess an example. So we assume every choice was seen at least once or what is sometimes called Laplacian smoothing.

We train this model by collecting a dataset of (context, choice) pairs. These are created in a domain-specific way from example programs that be converted into these (context, choice) pairs in an entirely

Submitted to:
TeaseLP

$$e \quad := \quad x \mid (e_1\ e_2) \mid (\lambda\ (x)\ e_1)$$

Figure 1: Example grammar

offline manner. For each pair we count what choices were made per context. These counts are then saved into a table that our implementation of **cond**^{e} accesses selecting clauses from most to least probable, thus guaranteeing the search stays complete.

```
(define zip
  (lambda (ls1 ls2)
    (if (null?  ls1)
      '()
      (cons (cons (car ls1) (list (car ls2)))
        (zip (cdr ls1) (cdr ls2))))))
```

```
(((app-rand* cdr) . 2)
((app-rator var) . 1)
((car var) . 2)
((cdr var) . 2)
((cons-e1 car) . 1)
((cons-e1 cons) . 1)
((cons-e2 app) . 1)
((cons-e2 list) . 1)
((if-alt cons) . 1)
((if-conseq quoted-datum) . 1)
((if-test null?) . 1)
((lambda if) . 1)
((letrec-rhs lambda) . 1)
((list car) . 1)
((null? var) . 1)
((top-level letrec) . 1))
```

2 Experiments

We show a preliminary evaluation in the table below. The following results come from trying to complete a program synthesis task. Synthesis is defined as satisfying a query in a relational interpreter written in miniKanren. We first try to synthesise programs where an expert chooses the ordering of the different rules in the grammar, and then with our method where we use the probabilities of a learned model to select the grammar rule.

The interpreter is for a subset of the Scheme language, and we are thus able to train using a collection of programs sourced from the Little Schemer and the Seasoned Schemer.

Table 1: Times for completing synthesis test in seconds

Program	Expert ordering	n-gram directed ordering
append	0.9809	0.5906
reverse	–	0.0115
rember	0.0103	0.007
foldr	2.3881	0.0064

We show that in isolation this optimisation yields significant speedups on several programs. For some programs, like reverse, the original program did not find a solution even when given several minutes.

3 Future Work

The present method is highly specialised to the program synthesis task of Barliman, but there is a significant possibility that if we had a dataset of queries and query answers, we could generalise our results for miniKanren programs in general.

References

[1] William E Byrd, Michael Ballantyne, Gregory Rosenblatt & Matthew Might (2017): *A unified approach to solving seven programming problems (functional pearl)*. *Proceedings of the ACM on Programming Languages* 1(ICFP), p. 8.

[2] Woosuk Lee, Kihong Heo, Rajeev Alur & Mayur Naik (2018): *Accelerating search-based program synthesis using learned probabilistic models*. In: *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ACM, pp. 436–449.

[3] Wojciech Zaremba, Karol Kurach & Rob Fergus (2014): *Learning to discover efficient mathematical identities*. In: *Advances in Neural Information Processing Systems*, pp. 1278–1286.